February 1971

The Design of a Meta-System for Measurement and Simulation of Time-Sharing Computers

Andrew S. Noetzel

University of Pennsylvania

Follow this and additional works at: http://repository.upenn.edu/cis_reports

Recommended Citation


This paper is posted at ScholarlyCommons. http://repository.upenn.edu/cis_reports/803
For more information, please contact libraryrepository@pobox.upenn.edu.
The Design of a Meta-System for Measurement and Simulation of Time-Sharing Computers

Abstract
This report presents the results of a study of a measurement-simulation technique for evaluating modifications to time-sharing systems. The technique consists of using raw data, obtained by measurement of an operational system, as the input to a simulation model of the variant version of the system. The results of the research are reported in the form of a detailed design of a system that will perform this evaluation. It is called the Meta-System.

The Meta-System consists of three parts: event-recording mechanisms in the time sharing system that record a 'system event trace', a pre-processor of the system event trace that decomposes it into independent 'task event traces' and the simulator that accepts these as input.

The significant problem in the design of such a system is that of ensuring that the representations of the systems operation (the task event traces) are valid for use by the simulation model. Various solutions for the particular cases of timesharing system operation are provided in the design of each of the three stages of the Meta-System.

It is shown that the Meta-System may be designed to extract representations of the time-sharing systems' operation at any one of several levels. These are called the 'levels of Meta-System awareness of system operation'. The extent of the simulation model is specified by the level, as well as the class of modifications that may be evaluated in the model.

The particular Meta-System designed in this study was developed to operate with a model time-sharing system, whose features have been abstracted from those of several modern systems. These features include demand paging, multi-tasking, dynamic loading, and a virtual access method. The description of the model system is included in the report.

Keywords
event analyzer, file structure, input/output simulators, measurement-simulation, memory allocation, meta-system, random access, simulation, time sharing

Comments

This technical report is available at ScholarlyCommons: http://repository.upenn.edu/cis_reports/803
TECHNICAL REPORT

THE DESIGN OF A META-SYSTEM FOR MEASUREMENT AND SIMULATION OF TIME-SHARING COMPUTERS

by

Andrew S. Noetzel

February 1971

Submitted to the
Office of Naval Research
Information Systems Branch
Arlington, Virginia

under
Contract N00014-67-A-0216-0014
Research Project NR 049-153

Reproduction in whole or in part is permitted for any purpose of the United States Government

Moore School Report No. 71-15
This report presents the results of a study of a measurement-simulation technique for evaluating modifications to time-sharing systems. The technique consists of using raw data, obtained by measurement of an operational system, as the input to a simulation model of the variant version of the system. The results of the research are reported in the form of a detailed design of a system that will perform this evaluation. It is called the Meta-System.

The Meta-System consists of three parts: event-recording mechanisms in the time-sharing system that record a 'system event trace', a pre-processor of the system event trace that decomposes it into independent 'task event traces' and the simulator that accepts these as input.

The significant problem in the design of such a system is that of ensuring that the representations of the systems operation (the task event traces) are valid for use by the simulation model. Various solutions for the particular cases of time-sharing system operation are provided in the design of each of the three stages of the Meta-System.

It is shown that the Meta-System may be designed to extract representations of the time-sharing systems' operation at any one of several levels. These are called the 'levels of Meta-System awareness of system operation'. The extent of the simulation model is specified by the level, as well as the class of modifications that may be evaluated in the model.
<table>
<thead>
<tr>
<th>KEY WORDS</th>
<th>LINK A</th>
<th></th>
<th>LINK B</th>
<th></th>
<th>LINK C</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Event analyzer</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>File structure</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Input/Output simulators</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Measurement-simulation</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Memory allocation</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Meta-system</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Random access</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Simulation</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Time sharing</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
13. ABSTRACT (continued)

The particular Meta-System designed in this study was developed to operate with a model time-sharing system, whose features have been abstracted from those of several modern systems. These features include demand paging, multi-tasking, dynamic loading, and a virtual access method. The description of the model system is included in the report.
Abstract

THE DESIGN OF A META-SYSTEM FOR MEASUREMENT AND SIMULATION OF TIME-SHARING COMPUTERS

This report presents the results of a study of a measurement-simulation technique for evaluating modifications to time-sharing systems. The technique consists of using raw data, obtained by measurement of an operational system, as the input to a simulation model of the variant version of the system. The results of the research are reported in the form of a detailed design of a system that will perform this evaluation. It is called the Meta-System.

The Meta-System consists of three parts: event-recording mechanisms in the time-sharing system that record a 'system event trace', a pre-processor of the system event trace that decomposes it into independent 'task event traces' and the simulator that accepts these as input.

The significant problem in the design of such a system is that of ensuring that the representations of the systems operation (the task event traces) are valid for use by the simulation model. Various solutions for the particular cases of time-sharing system operation are provided in the design of each of the three stages of the Meta-System.

It is shown that the Meta-System may be designed to extract representations of the time-sharing systems' operation at any one of several levels. These are called the 'levels of Meta-System awareness of system operation'. The extent of the simulation model is specified by the level, as well as the class of modifications that may be evaluated in the model.

The particular Meta-System designed in this study was developed to operate with a model time-sharing system, whose features have been abstracted from those of several modern systems. These features include demand paging, multi-tasking, dynamic loading, and a virtual access method. The description of the model system is included in the report.
**TABLE OF CONTENTS**

<table>
<thead>
<tr>
<th>Chapter 1: Introduction and Summary</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>The Meta-System</td>
<td>1</td>
</tr>
<tr>
<td>Summary of Research Reported</td>
<td>2</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Chapter 2: Background, Motivation, and Previous Work</th>
<th>13</th>
</tr>
</thead>
<tbody>
<tr>
<td>Evaluation and Measurement of Time-sharing System Performance</td>
<td>13</td>
</tr>
<tr>
<td>Criteria for Evaluating Batch Processing Systems</td>
<td>13</td>
</tr>
<tr>
<td>Quantification of Criteria</td>
<td>15</td>
</tr>
<tr>
<td>Effects of Evaluation in the Development of Batch Processing Systems</td>
<td>16</td>
</tr>
<tr>
<td>Difference Between Batch and Time-Sharing Evaluation</td>
<td>18</td>
</tr>
<tr>
<td>Response Time Constraint</td>
<td>19</td>
</tr>
<tr>
<td>Intrinsic Problem of Time-Sharing Systems</td>
<td>21</td>
</tr>
<tr>
<td>Sophistication of System Design</td>
<td>24</td>
</tr>
<tr>
<td>Isolation of User Requirements From Hardware</td>
<td>24</td>
</tr>
<tr>
<td>Criteria for Time-Sharing Systems</td>
<td>27</td>
</tr>
<tr>
<td>Measurement Techniques</td>
<td>30</td>
</tr>
<tr>
<td>Measurement of Utilization</td>
<td>32</td>
</tr>
<tr>
<td>Approach to the Measurement Problem</td>
<td>33</td>
</tr>
<tr>
<td>The Meta-System</td>
<td>38</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Chapter 3: Overview of Meta-System Design</th>
<th>39</th>
</tr>
</thead>
<tbody>
<tr>
<td>The Measurement Function of the Meta-System</td>
<td>39</td>
</tr>
<tr>
<td>The Evaluation Function of the Meta-System</td>
<td>41</td>
</tr>
<tr>
<td>Possibility of an Automatic Meta-System</td>
<td>42</td>
</tr>
<tr>
<td>Topic</td>
<td>Page</td>
</tr>
<tr>
<td>----------------------------------------------------------------------</td>
<td>------</td>
</tr>
<tr>
<td>Functional Description of the Meta-System</td>
<td>44</td>
</tr>
<tr>
<td>The Recording Mechanisms</td>
<td>44</td>
</tr>
<tr>
<td>The Preprocessor</td>
<td>45</td>
</tr>
<tr>
<td>The Simulator</td>
<td>47</td>
</tr>
<tr>
<td>Levels of Meta-System Awareness of System Operation</td>
<td>48</td>
</tr>
<tr>
<td>Design Problems of the Meta-System</td>
<td>50</td>
</tr>
<tr>
<td>Task Identifications in the Event Trace</td>
<td>52</td>
</tr>
<tr>
<td>Representation of Processor Time Requirements</td>
<td>53</td>
</tr>
<tr>
<td>Specification of Memory Requirements</td>
<td>55</td>
</tr>
<tr>
<td>Entrances and Exits to System Routines</td>
<td>56</td>
</tr>
<tr>
<td>Volumes of Data Recorded: The Concept of Class of Subroutines</td>
<td>58</td>
</tr>
<tr>
<td>Modeling System Routines</td>
<td>62</td>
</tr>
<tr>
<td>Chapter 4: The Time-Sharing System Description</td>
<td>64</td>
</tr>
<tr>
<td>The Hardware</td>
<td>64</td>
</tr>
<tr>
<td>The Processor</td>
<td>65</td>
</tr>
<tr>
<td>The Main Memory</td>
<td>66</td>
</tr>
<tr>
<td>The Secondary Storage</td>
<td>68</td>
</tr>
<tr>
<td>I/O Channels and Devices</td>
<td>69</td>
</tr>
<tr>
<td>The Generalized Operating System</td>
<td>69</td>
</tr>
<tr>
<td>Resident Parts of the Operating System</td>
<td>70</td>
</tr>
<tr>
<td>The Operation of I/O Devices</td>
<td>77</td>
</tr>
<tr>
<td>Management of Channel Queues by the I/O Control Subsystem</td>
<td>81</td>
</tr>
<tr>
<td>The Channel-Complete Interrupt Response</td>
<td>84</td>
</tr>
<tr>
<td>Topic</td>
<td>Page</td>
</tr>
<tr>
<td>----------------------------------------------------------------------</td>
<td>------</td>
</tr>
<tr>
<td>The Task Initiator</td>
<td>87</td>
</tr>
<tr>
<td>The File System: Logical I/O</td>
<td>88</td>
</tr>
<tr>
<td>The File Structure</td>
<td>91</td>
</tr>
<tr>
<td>Address Transformation of User Files</td>
<td>93</td>
</tr>
<tr>
<td>Logical I/O Macros and Operations</td>
<td>97</td>
</tr>
<tr>
<td>The Virtual Access Method</td>
<td>104</td>
</tr>
<tr>
<td>VAM Files</td>
<td>105</td>
</tr>
<tr>
<td>Economization of Secondary Storage</td>
<td>109</td>
</tr>
<tr>
<td>The Loader</td>
<td>112</td>
</tr>
<tr>
<td>Resolution of Address Constants</td>
<td>112</td>
</tr>
<tr>
<td>Loader Operation</td>
<td>117</td>
</tr>
<tr>
<td>Loader Operation with the Virtual Access Method</td>
<td>120</td>
</tr>
<tr>
<td>The Page Loader</td>
<td>124</td>
</tr>
<tr>
<td>Multi-Tasking</td>
<td>124</td>
</tr>
<tr>
<td>Higher Level Macros</td>
<td>129</td>
</tr>
<tr>
<td>Summary</td>
<td>132</td>
</tr>
<tr>
<td>Chapter 5: Extracting the Event Traces</td>
<td>134</td>
</tr>
<tr>
<td>The Hardware Event Trace</td>
<td>136</td>
</tr>
<tr>
<td>Recording the SHET</td>
<td>138</td>
</tr>
<tr>
<td>SHET Example</td>
<td>138</td>
</tr>
<tr>
<td>Processor of the SHET</td>
<td>143</td>
</tr>
<tr>
<td>Example of Task Hardware Event Trace</td>
<td>150</td>
</tr>
<tr>
<td>The Physical Event Trace</td>
<td>152</td>
</tr>
</tbody>
</table>
# TABLE OF CONTENTS (continued)

<table>
<thead>
<tr>
<th>Topic</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>Definition of the SPET</td>
<td>153</td>
</tr>
<tr>
<td>Extracting the SPET</td>
<td>155</td>
</tr>
<tr>
<td>Example of the SPET</td>
<td>156</td>
</tr>
<tr>
<td>The Processor of the SPET</td>
<td>160</td>
</tr>
<tr>
<td>Task Physical Event Trace</td>
<td>161</td>
</tr>
<tr>
<td>The Logical Event Trace</td>
<td>165</td>
</tr>
<tr>
<td>Definition of the SLET</td>
<td>166</td>
</tr>
<tr>
<td>Identification of Entry and Exit Points</td>
<td>170</td>
</tr>
<tr>
<td>Recording the SLET</td>
<td>175</td>
</tr>
<tr>
<td>Definition of Logical Level</td>
<td>176</td>
</tr>
<tr>
<td>Example of SLET Recording</td>
<td>178</td>
</tr>
<tr>
<td>Preprocessor of the SLET</td>
<td>181</td>
</tr>
<tr>
<td>Example of TLET</td>
<td>182</td>
</tr>
<tr>
<td>The Relocation Event Trace</td>
<td>187</td>
</tr>
<tr>
<td>Definition of the Relocation Event Trace</td>
<td>189</td>
</tr>
<tr>
<td>Recording the SRET</td>
<td>190</td>
</tr>
<tr>
<td>Example of SRET Recording</td>
<td>190</td>
</tr>
<tr>
<td>Preprocessor of the SRET</td>
<td>193</td>
</tr>
<tr>
<td>Example of the TRET</td>
<td>194</td>
</tr>
<tr>
<td>Chapter 6: Design of the Simulator</td>
<td></td>
</tr>
<tr>
<td>The Simulator</td>
<td></td>
</tr>
<tr>
<td>Input to the Simulator</td>
<td></td>
</tr>
<tr>
<td>Output of the Simulator</td>
<td></td>
</tr>
</tbody>
</table>
### TABLE OF CONTENTS (continued)

<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>The Basic Simulator - Overview</td>
<td>200</td>
</tr>
<tr>
<td>The Clockworks</td>
<td>201</td>
</tr>
<tr>
<td>The Event Analyzer</td>
<td>204</td>
</tr>
<tr>
<td>The Hardware Model</td>
<td>205</td>
</tr>
<tr>
<td>Model of Random Access Device Operations</td>
<td>208</td>
</tr>
<tr>
<td>The Models of Event Response Routines</td>
<td>211</td>
</tr>
<tr>
<td>Replacements of Hardware Instructions</td>
<td>213</td>
</tr>
<tr>
<td>Main Memory Allocation by the Model</td>
<td>213</td>
</tr>
<tr>
<td><strong>Chapter 7: Examples of Meta-system Operation</strong></td>
<td></td>
</tr>
<tr>
<td>Method of Running Examples</td>
<td>219</td>
</tr>
<tr>
<td>Results from the Actual System</td>
<td>220</td>
</tr>
<tr>
<td>Investigation of Hardware Device Change Using the Physical Event Trace</td>
<td>222</td>
</tr>
<tr>
<td>Investigation of Change of Scheduling Discipline Using the Physical Event Trace</td>
<td>223</td>
</tr>
<tr>
<td>Evaluation of Alternative Logical I/O Routines Using the Logical Level Meta-system</td>
<td>225</td>
</tr>
<tr>
<td>Evaluation of More Complex Scheduling Disciplines Using the Logical Level Meta-system</td>
<td>226</td>
</tr>
<tr>
<td>Evaluation of System Memory Management Using the Relocation Event Trace</td>
<td>228</td>
</tr>
<tr>
<td><strong>Chapter 8: Results and Conclusions</strong></td>
<td></td>
</tr>
<tr>
<td>Uniqueness of the Work</td>
<td>232</td>
</tr>
<tr>
<td>Future Research</td>
<td>233</td>
</tr>
<tr>
<td><strong>Appendix A</strong> Location of Event Recording Mechanisms</td>
<td>233</td>
</tr>
<tr>
<td><strong>Appendix B</strong> Glossary</td>
<td>237</td>
</tr>
<tr>
<td></td>
<td>252</td>
</tr>
</tbody>
</table>
# LIST OF FIGURES

<table>
<thead>
<tr>
<th>Figure</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>1-1</td>
<td>The Meta-system</td>
<td>2</td>
</tr>
<tr>
<td>1-2</td>
<td>Conceptualization of Meta-system Level</td>
<td>2</td>
</tr>
<tr>
<td>2-1</td>
<td>Response Time as a Function of Utilization</td>
<td>20</td>
</tr>
<tr>
<td>2-2</td>
<td>Isolation of User Requirements from Hardware</td>
<td>26</td>
</tr>
<tr>
<td>2-3</td>
<td>Types of Meta-systems</td>
<td>35</td>
</tr>
<tr>
<td>3-1</td>
<td>The Meta-system</td>
<td>43</td>
</tr>
<tr>
<td>3-2</td>
<td>Meta-system Levels due to Subroutine Structure</td>
<td>60</td>
</tr>
<tr>
<td>4-1</td>
<td>Resident Parts of the Operating System</td>
<td>71</td>
</tr>
<tr>
<td>4-2</td>
<td>Page Fault Interrupt Response Routine</td>
<td>74</td>
</tr>
<tr>
<td>4-3</td>
<td>Paging Channel-Complete Interrupt Response Routine</td>
<td>75</td>
</tr>
<tr>
<td>4-4</td>
<td>Physical I/O Initiation</td>
<td>79</td>
</tr>
<tr>
<td>4-5</td>
<td>Examples of Path Sets</td>
<td>83</td>
</tr>
<tr>
<td>4-6</td>
<td>I/O Channel Complete Interrupt Response</td>
<td>85</td>
</tr>
<tr>
<td>4-7</td>
<td>I/O Wait and Terminal Response Wait Macros</td>
<td>89</td>
</tr>
<tr>
<td>4-8</td>
<td>Terminal I/O Complete Interrupt Response</td>
<td>90</td>
</tr>
<tr>
<td>4-9</td>
<td>Two Level Directory of Index Sequential File</td>
<td>92</td>
</tr>
<tr>
<td>4-10</td>
<td>Example of Generalized File</td>
<td>94</td>
</tr>
<tr>
<td>4-11</td>
<td>Abstract Structure of Syscat</td>
<td>94</td>
</tr>
<tr>
<td>4-12</td>
<td>Structure of Syscat</td>
<td>96</td>
</tr>
<tr>
<td>4-13</td>
<td>Logical I/O-Get</td>
<td>99</td>
</tr>
<tr>
<td>4-14</td>
<td>Open Macro</td>
<td>101</td>
</tr>
<tr>
<td>4-15</td>
<td>Physical Operations in File Open and Get</td>
<td>102</td>
</tr>
<tr>
<td>LIST OF FIGURES (continued)</td>
<td>Page</td>
<td></td>
</tr>
<tr>
<td>-----------------------------</td>
<td>------</td>
<td></td>
</tr>
<tr>
<td>4-16 Vam Get Macro</td>
<td>108</td>
<td></td>
</tr>
<tr>
<td>4-17 Operation of Vam Get</td>
<td>110</td>
<td></td>
</tr>
<tr>
<td>4-18 Address-Specifying Information in Object Modules</td>
<td>116</td>
<td></td>
</tr>
<tr>
<td>4-19 Resolution of External Symbols</td>
<td>119</td>
<td></td>
</tr>
<tr>
<td>4-20 Program Load Operation</td>
<td>122</td>
<td></td>
</tr>
<tr>
<td>4-21 Physical Events in the Load Operation</td>
<td>123</td>
<td></td>
</tr>
<tr>
<td>4-22 The Page Loader</td>
<td>125</td>
<td></td>
</tr>
<tr>
<td>4-23 The Create Macro</td>
<td>128</td>
<td></td>
</tr>
<tr>
<td>4-24 Task Wait and Post Macros</td>
<td>130</td>
<td></td>
</tr>
<tr>
<td>Figure 5-1 Preprocessor of SLET</td>
<td>149</td>
<td></td>
</tr>
<tr>
<td>5-2 Preprocessor of SPET</td>
<td>162, 163</td>
<td></td>
</tr>
<tr>
<td>5-3 Interface Choices for SLET</td>
<td>167</td>
<td></td>
</tr>
<tr>
<td>5-4 Subroutine Exits</td>
<td>173</td>
<td></td>
</tr>
<tr>
<td>5-5 Recording I/O Routines</td>
<td>177</td>
<td></td>
</tr>
<tr>
<td>5-6 Preprocessor of SLET</td>
<td>183, 184</td>
<td></td>
</tr>
<tr>
<td>5-7 Interface for Relocation Event Trace</td>
<td>188</td>
<td></td>
</tr>
<tr>
<td>Figure 6-1 Outline of Simulator</td>
<td>201</td>
<td></td>
</tr>
<tr>
<td>6-2 The Clockworks</td>
<td>203</td>
<td></td>
</tr>
<tr>
<td>6-3 The Event Analysis Routine</td>
<td>206</td>
<td></td>
</tr>
<tr>
<td>6-4 Subroutines for Simulated Hardware Instructions</td>
<td>210, 212</td>
<td></td>
</tr>
<tr>
<td>6-5 Software Model of I/O Request Macro</td>
<td>214</td>
<td></td>
</tr>
<tr>
<td>6-6 Page Re-use Times in Task Trace</td>
<td>216</td>
<td></td>
</tr>
<tr>
<td>6-7 Modification of Memory Demand by the Simulator</td>
<td>218</td>
<td></td>
</tr>
<tr>
<td>Figure</td>
<td>Description</td>
<td>Page</td>
</tr>
<tr>
<td>---------</td>
<td>-------------------</td>
<td>------</td>
</tr>
<tr>
<td>7-1</td>
<td>New Open</td>
<td>227</td>
</tr>
<tr>
<td>7-2</td>
<td>New Get</td>
<td>227</td>
</tr>
</tbody>
</table>
# LIST OF TABLES

<table>
<thead>
<tr>
<th>Table</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>5-1</td>
<td>Example of SHET</td>
<td>140,141</td>
</tr>
<tr>
<td>5-2</td>
<td>Extracted Events of Task 1</td>
<td>146</td>
</tr>
<tr>
<td>5-3</td>
<td>THET for Task 1</td>
<td>151</td>
</tr>
<tr>
<td>5-4</td>
<td>Example of SPET</td>
<td>157,158</td>
</tr>
<tr>
<td>5-5</td>
<td>TPET for Task 1</td>
<td>163</td>
</tr>
<tr>
<td>5-6</td>
<td>Example of SLET</td>
<td>178,179</td>
</tr>
<tr>
<td>5-7</td>
<td>TLET for Task 1</td>
<td>185</td>
</tr>
<tr>
<td>5-8</td>
<td>Example of SRET</td>
<td>190,191</td>
</tr>
<tr>
<td>5-9</td>
<td>TRET for Task 1</td>
<td>194</td>
</tr>
<tr>
<td>7-1</td>
<td>Utilization Figures for the Actual System</td>
<td>223</td>
</tr>
<tr>
<td>7-2</td>
<td>Results of Simulation Run With Fast Paging Device</td>
<td>224</td>
</tr>
<tr>
<td>7-3</td>
<td>Simulation Run With FCFS Scheduling</td>
<td>225</td>
</tr>
<tr>
<td>7-4</td>
<td>Simulation Run With New Logical I/O Routines</td>
<td>227</td>
</tr>
<tr>
<td>7-5</td>
<td>Simulation Run With Complex Scheduling Discipline</td>
<td>229</td>
</tr>
</tbody>
</table>
CHAPTER 1
INTRODUCTION AND SUMMARY

The design and implementation of multiprogrammed, time sharing computer systems continues long after the system is put into use. A tool is needed that will measure and evaluate the computer system while it is in operation, as an aid to further development or optimization for a particular usage. The design of such a tool is described in detail in this report. It is called the Meta-system.

The uniqueness of the Meta-system is due to the coalescing of two widely used techniques - on-line measurement, and simulation - into one system. Measurement is performed by extracting raw representations of a computer system's operation (from that system) using software techniques only. Evaluation of the system is based on input of the measured performance characteristics to a simulation model that exercises modified hardware-software versions of the system. All the potential modifications to the system are evaluated in the context of the task load of the system, as extracted from the operational system.

This report presents the design of the system. When implemented, the Meta-system will have a wide range of applicability. System designers will be able to optimize system performance by measuring the effects of modifications to hardware devices, system configurations, software residences, and operating system routines. Users of the system will obtain benchmark comparisons for various configurations and scheduling policies.
The Meta-system

The Meta-system consists of a set of routines which extract measurements from an operational time-sharing system, and a simulation model of that system. The raw data is extracted from the system in the form of a 'system event trace'. A preprocessor decomposes this event trace into event traces that represent the operation of each task within the system. These 'task event traces' become the input for the simulation model of the variant version of the system.

Together with the system designers, the Meta-system represents the measurement and evaluation loop that results in design improvements to the system. The Meta-system loop is shown in Figure 1-1.

The detailed design of the Meta-system, that is presented in this report and the feasibility of its implementation that may be inferred from the design, is the first important result of this research.

The second result is the specification of means and procedures for obtaining performance measurements of one system that will be useful in the simulation of some other system. This result is the generalization of the detailed design, and could be the basis for a theory of execution-simulation of computer systems. A brief discussion of the Meta-system technique is given before the results are stated more explicitly.

First, consider the measurements of one system that are useful for simulating another system. The measurements must not be the final results, such as the utilization factors of various hardware components, since these will be obtained from the simulation model. Rather, they
Figure 1-1
The meta-system

Figure 1-2
Conceptualization of Meta-system Level
will be measurements that can be interpreted by the simulation model -
frequencies of occurrence of the operations, for example. The simula-
tion model may then allocate a different time interval for each opera-
tion, and different utilization factors will be obtained. New resource
allocation techniques, as well as other system algorithms that influence
the resource allocation, may be investigated in the simulation model.

The measurements taken from the operational system will therefore
be measurements of the user task demand for various system resources,
which is related to the allocation of the resources through the opera-
tion of the system.

Next, it must be noted that a task's demand for system hardware
resources cannot be represented independently of the system on which the
task is run because the hardware resources vary from system to system.
The same is true for the demand for other resources (macros, algorithms,
tables, etc.) except at the highest level of specification - that of
machine-independent languages. This level, however, is a limiting case
of possible resource demand representations.

The result that is important to the theory of execution-simulation,
is that it is possible to find representations of user task demand that
are relatively independent of the system on which the task was run.

Relatively independent demand representations are representations
which remain valid for a system that is within a specified class of
modifications of the system from which the representations were taken.
One concrete example may help make the concept of relative independence
clearer. A task may be represented as a series of I/O operations. The
number and frequency of the I/O operations are functions of the size
of the data block that is involved in a single I/O operation. Then if
it is granted that every system on which the task will run will have
the same block size in its mass storage, the sequence of demands for
the I/O operations is a valid representation of the task's demand, even
if the speed of the I/O device is changed, or if the configuration of
the system or the scheduling of the device is altered so the wait time
until the operation begins is varied.

A further result is that these relatively independent represen-
tations of task demand may be taken from several levels of the system
organization. To each such level there corresponds a simulation model
of part of the system, that can be operated from the demand representa-
tion taken from that level.

The Meta-system levels are shown in Figure 1-2. The representa-
tion of the time-sharing system in this figure is quite arbitrary. It
roughly corresponds to the levels of logical complexity of the informa-
tion-processing capabilities of the system, which are greatest for the
parts of the system that directly communicate with the user, and least
at the level of the hardware. Representations of system operation taken
from one level are used as the input to a simulator of all parts of the
system below that level, including the hardware. Several such levels
of measuring and simulating the system are possible.

**Summary of the Research Reported**

The research encompassed by this report began as a broad study of
the problem of measurement and evaluation of time-sharing systems, and
converged to the concept of the Meta-system after the philosophy of measurement and evaluation was made precise.

A few definitions are necessary to specify the evaluation capabilities of the Meta-system.

An evaluation function of a class of computer systems is a mapping from the class into the set \{yes, no\}.

An evaluation of a computer system is the application of the evaluation function to the system.

Thus, an evaluation requires specification of the evaluation function. Implicit in the specification of the evaluation function is an action - either purchase, implementation, or modification of a system - to which the yes or no will apply.

The evaluation function may be decomposed into two functions. One is a mapping from the class of computer systems into a finite dimensional space. The dimensions of the space are called evaluation criteria. When this mapping is specified algorithmically, it is called the measurement function.

The second component of the evaluation function - the mapping from the evaluation criteria to \{yes, no\} - is called the utility function. This function will, in general, differ for each evaluator of a computer system.

The Meta-system will enable an evaluation of a computer system by specification of the measurement function (including the definition of the evaluation criteria space) and automatic application of that function to a computer system. The subjective utility function is
included within the Meta-system only if the human evaluator is considered part of the Meta-system.

The measurement function explicitly developed in this report, has the following evaluation criteria for its range:

1. Utilization of the processor
2. Utilization of the paging channel
3. Utilization of the I/O channels
4. A set of points representing average response time (for a particular task) with \( 1, 2, \ldots, n \) tasks active in the system.

The utilization of a particular device is the ratio of the time the device is active (working) during a period, to the total length of the period.

The technique of implementing the measurement function may be readily extended to include the following criteria:

1. Efficiency of the processor
2. Efficiency of memory
3. Efficiency of the paging channel

The efficiency of a device is defined as the ratio:

\[
\frac{\text{total time-idle}}{\text{total time}} - \frac{\text{time-overhead time}}{\text{total time}}
\]

Overhead time is the amount of time the device is not available, yet not performing a function directly related to the requirements of a task. Scheduling tasks and preparing devices for use are examples of overhead functions.

\(^1\) The measurement function is used to obtain these numerical quantities from a hypothetical system via simulation. Examples of the results of such measurement are presented in Chapter 7.
The results of the initial part of the study includes a review of other works being done in system measurement and evaluation, as well as some preliminary Meta-system design possibilities. These results are presented in Chapter 2 of this report. Chapter 2 is not central to the Meta-system design, and may be omitted by the reader interested in the results of Meta-system design effort.

A summary of all the significant aspects of the Meta-system design which are developed in detail in the latter parts of this report (Chapters 5 and 6), are summarized in Chapter 3. The important techniques developed are the following:

1. The definitions of four different event traces, necessary for measuring the systems operation at four different levels. These levels are:
   a) the hardware level
   b) the physical level
   c) the logical level
   d) the relocation level

   Each of these levels measures different degrees of the organization of the operating system. The definitions of the event traces are given in Chapter 5. The locations of the event-recording mechanisms in the operating system for each of these levels are shown in Appendix A.

2. A technique for decomposing these system event traces to remove the influence of the part of the system which will be simulated. This is accomplished in a preprocessor of
the event trace. The design of the preprocessors of the event traces listed above, is given in Chapter 5.

3. A technique for representing the processor requirements of a task in terms of processor time without memory interference. The preprocessor converts from the actual processor time - which may have been extended because of I/O interference in the memory, to the 'pure' processor time representation. This technique is included in the discussion of the Preprocessor of the Hardware Event Trace - Chapter 5.

4. A technique for simulating memory utilization by the tasks. This technique allows the simulator to follow the memory utilization data available in the task event traces for as long as it remains valid. It then uses the record of task memory allocations as a guide in the generation of new task memory requirements. This is shown in the simulator design, Chapter 6.

5. A technique for locating the significant subroutines that define a level in the structure of the operating system. This will allow generalization of the concept of Meta-system levels beyond the four levels explicitly developed in this report. This is described in Chapter 3 - Levels of Meta-system Awareness of System Operation'.
6. A technique for recording the entrances and exits to the set of subroutines that define a level. This will enable the preprocessor to eliminate from the event trace, the subroutine calls that result from the subroutines defining a level. The task event traces may thereby be greatly simplified. The secondary calls will be replaced by the simulator. This technique is described in Chapter 5, in the description of Logical Event Trace.

As the concept of the Meta-system evolved, several large timesharing systems were analyzed in detail, with a view towards implementing the Meta-system on them. These systems are: TS03 on the RCA Spectra 70/46, TSS on the IBM 360/75, and the Multics system on the GE 645. An implementation of the Meta-system on the Spectra/70 was originally intended; however, it became clear that the level of effort required for the implementation is so large that it will require continued efforts beyond the presently completed design phase.\(^2\) The design effort has concentrated on greater generality than could be achieved on any one system.

It was found necessary to develop a model of the various timesharing systems for two reasons. First, in order for the operation of the system routines to be replicated in the simulation model, they must

\(^2\) Even though the simulation model used in the design of the system was available. This model, however, besides representing an obsolete version of the system and not being supported by the system software, was not designed to meet the specific needs as a Meta-system simulator. The Meta-system simulator is presented in Chapter 6.
be abstracted at a level that includes all of the resource-allocation decisions made by them. The outlines of the system routines presented in the model time-sharing system are examples of this level of abstraction. Second, the model is necessary to precisely demonstrate the design and operation of the Meta-system.

The generalized model of the time-sharing systems that was developed is presented in Chapter 4. The model includes features common to all of the systems, such as interrupt-driven schedulers, demand paging, recursive and reentrant supervisor routines. It also includes some specific features that may be found within these systems, such as generalized hierarchical files and the Virtual Access Method. The Virtual Access Method is broadly representative of the class of complex logical I/O functions that appear in these systems, and it includes the operation of the dynamic loader.

With the aid of the model, and with reference to the original systems in particular instances, several system-independent representations of user task resource demand were developed. Each such representation, which takes the form of an event trace, is taken from a different level of the computer system's operation. For each of these levels, the method of recording the events is demonstrated on the model time-sharing system. Also, the preprocessor of the event trace, which prepares the event trace for input to the simulator, is developed. All of these results are presented in Chapter 5, along with examples of operation of the system.
Lastly, the simulator of the system, which accepts the measurements of task demand as input, was developed, and its operation tested with several sets of circumstances in the original system. The structure of the simulator follows that of the hardware interrupt mechanism, and software Interrupt Analyzer, but it operates on events rather than interrupts. The entire design of the simulator is described in Chapter 6.

The results of a few of the hand simulations of the entire Meta-system that were extensively used during the development of the Meta-system, are presented in Chapter 7. These provide some indications of the capabilities of the Meta-system, but they are not exhaustive.

Chapter 8 contains a statement of the conclusions of the work, and an outline of several directions of future research.

The detailed design of the Meta-system presented in Chapters 5 and 6 is predicated upon an understanding of the operation of the model time-sharing system described in Chapter 4. However, all of the significant design problems presented in these two chapters are reviewed, in less detail and without reference to the model system, in Chapter 3.

Appendix A shows the locations within the operating system described in Chapter 4, at which the event-recording mechanisms are placed for each of the event traces. A glossary of the most important terms used is presented in Appendix B.
CHAPTER 2
BACKGROUND, MOTIVATION AND PREVIOUS WORK

Evaluation and Measurement of Time Sharing Systems

Before the problem of measuring the performance of a time-sharing system can be approached, the criteria for evaluating such a system must be established. This will require, at least, definitions of the concepts of 'computing power' or 'work per unit time' that are precise enough to enable measurement of these quantities.[32] The work that has been done in evaluating batch-processing computer systems should provide some basis for the specification of this aspect of the time-sharing evaluation criteria. Then, examination of the differences in the structure and requirements of time sharing and batch processing systems, should enable specification of the time-sharing evaluation criteria to be completed.

Criteria for Evaluation of Batch Processing Systems

The evaluation of computer systems in general is a difficult problem. With the proliferation of computing devices in the early '60's it was recognized that standard measures were required to compare the capabilities of the various machines.[33] At first, only a few hardware parameters, such as memory cycle time and execution time for a particular instruction, were available as measures of performance. Then it became clear that throughput - the real indication of system performance - was dependent upon the work the machine was doing, and some problem-dependent schemes of estimating performance were developed from
the hardware parameters. A few examples are the estimates of processor speed based on a 'mix' of instructions that represent processor operation in a particular problem type, and the execution time of a 'kernel' problem typical of a given problem environment.\[A3,Cl] An evaluation approach which encompasses the entire machine operation is that of defining several problems to be 'benchmarks' or standard problems, and then timing their operation on different machines.\[Hi2] If these problems are written in high-level languages, the time will also reflect the efficiency of the compiler and the compiled code, thus measuring the software of the system as well, even though this was not the primary intent of the benchmark tests.

These computing measurements were reported as a set of performance characteristics for the various task types: 'scientific' tasks - which use a great deal of processor time, 'text processing' tasks - which largely use the I/O channels and devices and have relatively little processor need, and special jobs such as matrix inversion, which have a large main memory requirement.

In general, the results of the benchmark problems differed for each different type of benchmark task used for comparison - as did the instruction mix and kernel comparisons. Therefore the evaluation of any particular computer could not be completed until the characteristics of the problems it was to process were known.

These measurements of computing performance enable evaluation of a system by comparing it with other systems, in the context of a given class of problems. The purpose of the evaluation is the selection of
a system to process problems of that class. Another reason for evaluating a computer system is to assist in making design decisions. In this application, a performance characteristic of a computer is not compared with a similar performance characteristic of a competitor's machine, but with the performance of some modified version of the system being developed. The degree of the performance difference in the two versions of the system, together with consideration of the cost difference, will determine which of the versions will be selected as the basis of further development. The criteria for this evaluation should be consistent with criteria used by a user or purchaser of the system, which are based on comparisons between diverse systems.

Quantifications of Criteria

The general, non-standardized performance criteria that has always been used for batch systems is throughput. Indications of throughput have been quantified in two ways: speed of the hardware components, and utilization of the components. Speed is primarily an attribute of the hardware, but is also somewhat dependent upon the large operating system software packages, such as compilers. The speed of a compiler is measured in terms of both speed of compilation and efficiency of compiled code. The degree of utilization of the hardware is due to both the scheduling software of the operating system and the configuration - how well the speeds of the hardware components are matched. The effects of the speed and utilization measurements of batch processing systems can be seen in the development of the batch processing systems.
Effects of Evaluation in the Development of Batch Processing Systems

From the outset, batch processing measurements and evaluation were directed towards the technological aspects of the computer's performance. The search for meaningful evaluation criteria did not begin with questions about whether the computer was serving useful functions or what is the 'work' of a computer. Instead, it was implicitly assumed that there was a direct relationship between doing useful work and keeping the system busy reading input cards and spewing out reams of paper.

The particular criteria of effectiveness centered on the CPU, since it was obvious that the real work of a computer is performed by the CPU. Since the peripheral devices - the card reader and printer - were inexpensive compared to the CPU, significant performance improvement could be economically obtained by increasing the capacity of the peripherals. This creates a better match between CPU and peripheral speeds, and thus increases the utilization of the CPU.

CPU utilization was also increased by the introduction of job-swapping. [Fl,K2] This is most effective when there is a large difference in the speeds of the CPU and the I/O units as in the case when the I/O is done on serial-access files (tapes). A task which is waiting for the completion of a long-period I/O operation is taken out of main memory and put in a backup memory, and another job taken from the backup memory and processed. This keeps the CPU busy during periods of I/O, less the time it takes to swap one job out and load another. The greater overall utilization of the CPU and the I/O devices that is realized is due to parallel processing, in the sense that a CPU operation for one task and
I/O operation for another are taking place simultaneously. The cost is that of having special hardware devoted to swapping.

Another more sophisticated solution to the problem of achieving greater CPU utilization was the implementation of multiprogramming systems. [F7,S5,R1] Although achieving greater overall utilization of the system hardware by parallel processing, they further focused the criteria of system performance on the CPU utilization, because they are implemented at the expense of having a main memory large enough to contain several tasks. In a multiprogrammed system, all but a few of the tasks in main memory are idle at any time, which means there is a large amount of memory idleness in the system. "Memory idleness" however was not a concept linked to poor throughput. Until the development of the most recent systems, much less attention has been paid to the cost of memory idleness than CPU idleness.

The efforts to solve the problem of memory fragmentation showed a concern for memory utilization as a parameter of system performance. Fragmentation is the process by which pieces of memory become unusable because they are not matched to the requirements of current programs. One solution, called shuffling or core compaction, is to periodically move the active tasks into contiguous areas of main memory. This required new hardware to make programs relocatable and takes some CPU time, thus increasing overhead.

The memory fragmentation problem can be alleviated, (but not eliminated [R3]) without shuffling, by the implementation of paging. With this scheme, the several programs of a multiprogramming system can
be loaded into whichever blocks of main memory are available. Special hardware is required to convert the addresses referenced by each program of such a system into the actual physical locations. A further economy of memory size is afforded by demand paging. [D2,L3,Sl0,R2] In demand paging systems, active programs can be executed from main memory without being entirely resident there. (In fact, some uni-programmed systems have been developed using demand paging to allow a small physical memory to be efficiently used. See [F5,K4].)

It can therefore be seen that the evolution of batch processing systems was accompanied by the displacement of evaluation criteria from concern for CP utilization to concern for utilization of each system component. The utilization of any component is relative to

a) the cost of the component

b) the available tradeoffs with utilization of other devices.

Considering these two factors at once leads to the concept of a 'balanced system' in which any change in the utilization of any component would result in deterioration of throughput. [D8] This balance point is not easily measured and not well defined, but it may serve as the ultimate criteria in system evaluation.

Differences Between Batch and Time-Sharing Evaluation

There are several major differences between batch processing systems and time sharing systems, with respect to performance criteria. First, and most obviously, is the presence of a response time constraint for time sharing systems. The second difference, which represents a serious difficulty in attempting to evaluate a time-sharing system, is
that the problems in design and evaluation of a time-sharing system are largely intrinsic problems, i.e., they deal with the communication between the human and the computer. The problems of a batch processing system, their solutions, and therefore the evaluations of those systems are mostly technological, dealing with the machine hardware only, and almost always solvable by advances in technology. (The distinction between intrinsic and technological problems was made by Saltzer in [Sl].)

Third, the increased sophistication of the time-sharing system design demands increased sophistication in measurement. Lastly, the increased isolation of users demands from the hardware imposes special requirements on measurement techniques.

**Response Time Constraint**

Many mathematical analyses of time-sharing systems [C4, C5, K3] have used the assumption that the input task inter-arrival time is a random variable with an exponentially distributed probability density function with mean wait time \( \frac{1}{\lambda} \). The service time - which is the processor time required by the task - is also assumed to be exponentially distributed, with mean \( \frac{1}{\mu} \). The processor is a single-server.

With these assumptions, the following results are immediate:

The utilization factor of the processor is \( p = \frac{\lambda}{\mu} \). The average queue length for the processor, including the task on the processor, is \( \frac{\lambda}{\mu - \lambda} \). The average time on the processor queue and in service, \( \frac{1}{\mu - \lambda} \), is called the response time. The response time, normalized by the mean service time is

\[
\frac{\text{response time}}{\text{service time}} = \frac{\mu}{\mu - \lambda} = \frac{1}{1 - p}
\]
A plot of the normalized response time, as a function of the utilization is shown in Fig. 2-1.
With this simple model of a time-shared computer, the number of users active on the system is proportional to \( \lambda \), and since \( p = \frac{\lambda}{\mu} \), \( p \) is also proportioned to the number of active users. Figure 2-1 can be assumed to show normalized response time as a function of the number of users on the system. The tradeoff between the response time and number of users is obvious.

With respect to evaluating time-sharing systems, the significant problem is how can systems operating at different points of this curve be compared?

A consideration that alleviates this problem is the following: if the response time is small enough that the results appear quasi-instantaneously to the user at the terminal, no further reduction in response time is necessary. Once response times are reduced to approximately one second, any further reduction becomes a luxury, and an evaluation procedure should not judge favorably any sacrifice of utilization to achieve further reduction of response times.

**Intrinsic Problems of Time Sharing Systems**

Time sharing systems operate over a broad range of application. A dedicated time-sharing system, for which the American Airlines Sabre system [D3] is the classical example, provides responses to a limited number of commands or queries. The user may not use the system in the traditional way general-purpose computers are used: he may not write his own programs.
IBM's Quicktran system [11] is an example of a system which allows the user to write and run his own FORTRAN programs, but he is restricted to the single language, and his capacity to use the system for other resources, such as information storage and retrieval, is severely limited. This system is designed for the scientific user who wishes to program interactively.

Other time-sharing systems have been designed for scientific computation capability in several languages, or for business data processing with the COBOL language. [38]

Comparisons made between two different types of time-sharing systems can only be meaningful if each system is measured doing its own work, and if the evaluation criteria are independent of the nature of the task. And any evaluation criteria which is meaningful must derive from measures of the quality and quantity of the work performed.

One area of application common to many time-sharing systems is program development. An experimental study by Schatzoff, Tsao, and Wiig [33] demonstrates how systems that are widely diverse in capability can be compared in effectiveness of aiding program development. In this study, several parameters of program development effort were compared for a time sharing system and a batch processing system. If a broad view of time-sharing systems is taken, in which the batch processing system is considered a time sharing system with a very long response time, the study may be viewed as a comparison of time-sharing systems. The interactivity required in the definition of a time-sharing system is a necessity for program development, whether it is performed from a
console or not. The parameters of the program development effort that were compared are the following:

1. Elapsed time - from start to completion of the problem.
2. Analysis time - time spent in programming, analysis and debugging.
3. Programmer’s time - Analysis time plus keypunch and terminal time.
4. Computer time
5. Number of compilations.
6. Total cost - programmer’s time and machine time.

The results of this experiment are of no general significance, except that such comparisons between systems can be made. (For the record, the results indicated a smaller elapsed time for time sharing but longer analysis and program time. Computer time used was the same for each case, although a much greater number of compilations was made under time sharing. The total cost for time-sharing was about 50% higher than the batch case. The results are not general, since the batch processing machine was faster, and the time-sharing system - CTSS - a primitive one.)

The choice of parameters selected for this comparison is of significance. Only one of the six parameters - computer time - is directly related to the hardware. In trying to find a basis for making improvements in the other areas, the tradeoffs between them must be understood and evaluated. Thus, all of the following must be resolved.
• What is the value of ease of use by an inexperienced user, especially in light of possible tradeoff of ease of use by experienced users?

• What is a short elapsed time to completion worth in terms of greater total effort and cost?

These are the intrinsic problems of time-sharing systems, that make their evaluation difficult.

**Sophistication of System Design**

In serial batch processing systems, the computation power of the system was shown by the hardware parameters indicating size and speed. With the advent of multiprogramming and multiprocessing, complex schemes have been devised to maximize the utilization of the hardware. Some of these algorithms, when used under the load conditions for which they were designed, provide large hardware utilization. When the demand is not matched to the algorithm, they allow severe performance degradation. Hence, in the overall evaluation, the hardware parameters of these systems are much less relevant.

The measurement parameters used to replace the hardware parameters must show the effectiveness of these resource allocation algorithms. General effectiveness measures applicable to the wide variety of hardware and software techniques of resource allocation are non-existent.

**Isolation of User Requirements From Hardware**

From the point of view of measuring the systems, there is one other essential difference between serial batch processing systems and more advanced multiprogrammed systems, not necessarily restricted to
time-sharing systems. This is the increasing isolation of the users processing requirements from actual hardware requirements. The trend can be seen in a brief review of the evolution of operating systems.

In the earliest computer systems, when one user was operating the machine at a time, his programs had direct control over the hardware. (Fig. 2-2a) Early operating systems, in addition to scheduling the batch jobs, provided routines that aided the user in control of hardware. (Fig. 2-2b) With the introduction of multiprogramming, it became imperative that the system exercise complete control of the hardware, to prevent user programs from interfering with resources allocated to another process. The operating system represented in Fig. 2-2c is actually the 'task scheduler' or 'resource allocation' routine - since it schedules the tasks for operation when resources are available, and allocates the resources to the task in response to the tasks requirements.

Increasingly, operating systems provided users with information processing routines - file handling capabilities, language processors, and mathematical routines, for example. Most of these other functions have not been written to be an integral part of the system, but are handled by the operating system as a user program. Some of these, however, such as the file-handling capabilities, have significant bearing on the resource allocation procedures of the system, since they are a transformation from the information requirements of the user to the requirements for system resources. They can be integrated with the resource allocation scheme. (See Fig. 2-2d.)
a) Primordial Computer System

b) Early Operating System

c) Multiprogrammed Operating System

d) Late Operating System

Figure 2-2 Isolation of User Requirements From Hardware
The evolution of such general purpose systems is in the direction of isolating the user further from actual hardware demands. This occurs by defining new system functions (service routines - written as user programs), and integrating the older service routines with the resource allocation functions at the lower levels.

The evaluation of such a system involves the evaluation of several levels of transformations of user demands into further requirements and ultimately into hardware utilization.

When such a system is measured, it must be subjected to a load of user requirements defined at some level, and the response to the load measured at some other level. The selection of the user requirement definition, the level of measurement, the level at which the response is measured, and the definition of performance at that level, are some of the complex measurement problems presented by this type of system. They can be resolved for the measurement of one system, but the results will not be comparable to solutions obtained for any significantly different system.

Criteria for Time-Sharing Systems

The differences between time-sharing system and batch processing systems are significant enough to conjecture that evaluation of the time-sharing systems will be a different and more complex problem. The most important criteria for evaluating these systems, from the point of view of a designer or prospective purchaser (as opposed to a user of the system, whose scope of interest is more limited) are listed below. The criteria are not listed in order of their importance, since each evalua-
tor will weigh the criteria according to his own concept of their importance.

1. The number and type of standard software packages available. Usually, the system will be oriented toward a particular usage, which will be reflected by the type of language processors and utility programs. Systems with many language processors will pay a price in terms of performance.

2. The number of simultaneous users the system can support. Generally, the number of lines by which users communicate with the system can be expanded by simple hardware additions. The limiting factor on the number of simultaneous users is then the degradation of response time to an unacceptable level due to an overload of processing demand. The 'number of users vs. response time' characteristic is then a single criteria by which one of the factors may be chosen, if the other is fixed. This characteristic generally has one fairly well defined region, beyond which performance degradation is severe as users are added. This region may be taken to define both response time and maximum number of users. In modern time-sharing systems, the response time thus defined is less than two seconds. The response time criteria can be overlooked, then, by calling this adequate, and considering the corresponding number of users as the evaluation criteria.

3. Cost. All other criteria are relative to this.

4. Expansion capability. A criterion important to many users is the degree of ease by which the user can integrate his own subsystems into the overall system, and the performance of the system with the
expanded operating system.

5. Speed and Utilization. The hardware criteria used for batch processing systems are still applicable. If a system must be evaluated in the absence of the 'number of users vs. response time' characteristic, the utilization data may be used to estimate this characteristic. Such a situation arises when a system under development is to be evaluated.

6. Other criteria such as reliability, maintainability, and support. These factors may be quantified theoretically during system design, but they are usually not believed until substantiated by statistics obtained during a significant period of experience. Reliability figures for new systems are never very favorable, hence are not used in evaluating these systems. A 'fail-soft' or 'graceful degradation' feature is a special requirement for time-sharing, but quantifications of this feature have not been generalized to be applicable to diverse systems.

An attempt to design a system to measure each of these criteria shows that the only criteria amenable to formal quantitative specification and measurement, besides cost, are the technical performance criteria 2) and 5). The other criteria are still evaluated subjectively. However, even the objective criteria are evaluated with respect to the task requirements of the system, which are, in general, different for each user.
Measurement Techniques

The evaluation criteria of time-sharing systems are still largely subjective and are usually expressed qualitatively. However, some criteria can be expressed quantitatively - speed, utilization and response time. These quantitative criteria have been measured often. The most extensive measurements have been made by measurement systems that have been implemented and used during the design of the computer system.

Measurement has been found to be indispensable at every step in the development of a computing system. Hypothetical systems as well as real systems have been measured. The general plans for a system are guided by the analysis of theoretical system models. The results of these analyses are measurements of the operation of the hypothetical system. As the plans for the system become more refined, analysis by theoretical models becomes unfeasible, necessitating the development of simulation models. [F3,F6,H3,H5,K1,M6,N2-5,02,S12,R6] It has been reported in several cases [C2,02] that a major drawback of design by simulation has been that the model could not be completed before the actual design decisions had been made and implemented.

Performance has been measured on several different levels of system operation, both by simulation and the measurement of systems during implementation. Some of these measurements are the following:

(1) CPU measurement at the instruction level to quantify the speed of execution of the processor. The resultant figure for speed of processor will depend upon the 'instruction mix' deemed to be most
representative of the actual operation of the system. The 'mix', 'kernel' and benchmarks have been discussed previously. The limitations of these measurements are discussed in [C1].

Several hardware measurement devices have been implemented in order to monitor a variety of hardware functions indicating processor speed and utilization. Both UCLA's Snuper Computer [E1] and a System Performance Activity Recorder developed by IBM for measurement of the 360/67 time sharing system [S4] will gather data on the frequency of instruction use in an operational system.

(2) Program speed. Since the performance of the system will depend upon the language processors and service routines, some measurements of the speed of operation of these routines have been made. [P6]

(3) Subsystem performance. Throughput rates and response time for I/O subsystems have been evaluated by analytical models. [A2,P4]

(4) User console measurements. The system throughput will be ultimately dependent upon the user behavior at the console. Several studies on existing time sharing systems have been made of the user behavior - distribution of think time, type in time, etc. [C3,S7] These results were necessary to act as guidelines for system design, and input to simulations. [D7,G5] The user behavior at the console depends upon the characteristics of his problem, the format of his input, and the ease of response. It will not be possible to apply results of user behavior on one system to another, except in general terms, unless the interactive system on each are the same.
(5) Overall system utilization. Of all the above measures, the throughput of the system is best correlated with the amount of time each of the system components spends performing useful - that is, user demanded - functions. If the utilizations are known, they will indicate the areas in which system performance is lacking and areas of possible improvement.

Measurement of Utilization

Measurements of system utilization have been made by simulation, hardware measurement devices, on-line recording and condensation of measurement data \([S7,S8,S12]\) and on-line recording of events for off-line condensation and analysis \([C2,C3]\).

Aside from the hardware performance measurement devices, the most comprehensive measurement schemes reported are those implemented for design analysis of the GECOS-II batch operating system for the GE-635 computer. These are reported by Cantrell and Ellison in \([C3]\). First, the accounting program records the I/O channel time used by each user task, as well as the processor time and time-of-day of beginning and end of the task. Second, a trace program records, in a circular list, every major system event and the time at which it took place. A continuous trace is also recorded on tape. This has proved extremely useful in finding several large inefficiencies. Third, individual programs are analyzed by running them in a uniprogrammed mode, to obtain a complete I/O and compute time profile. Fourth, a profile of the execution pattern of a particular program is obtained by a high density sampling of the program counter. The program is interrupted at a high rate during its
operation, and the distribution of interrupt addresses, which is proportional to the amount of time the program spent in each locality, is recorded. Fifth, a condensation of the processor and of channel time used for each system and user program is printed every two seconds. The distribution of main memory among the active programs is also printed each time a new task is initiated or terminated.

Implementation of the GECOS-III time-sharing and batch processing system reported by Campbell and Heffner [C2] was done with the aid of a similar major event trace, recorded during system operation and analyzed later. Utilization data was also condensed at periodic intervals, and either displayed on a cathode ray tube or printed.

**Approach to the Measurement Problems**

The problem addressed by this work is the design of a system that will evaluate the performance of time-sharing systems in terms of meaningful criteria. The system will be called the Meta-system. The design constraints of the Meta-system will be established. Each of the possible applications of the Meta-system will impose major constraints on the measurement techniques, and the measurable evaluation criteria. Three possible applications of the Meta-system are:

1) **System Selection.** A prospective user needs criteria to help select, of several diverse systems, the one most suitable for his applications.

2) **System Optimization During Design.** A measurement and evaluation procedure for unimplemented systems is needed to aid in design decisions.
loaded, nor the utilization of any part of the system.

This Meta-system is only as good to the user as the benchmark tasks are approximations to the problems he will actually be running.

2. System Optimization During Design. The Meta-system designed for this purpose will be significantly different from the first case. First of all, it will be necessary to measure and evaluate parts of the system when the operation of the total system is not available. The measurement system must interface with the part of the system to be measured.

Secondly, the evaluation procedure must be applied to representations of systems which have not yet been implemented. The evaluation procedure should work on a simulated system as well as a real one.

Lastly, the relationship between internal statistics and the overall load vs. response time characteristic must be established.

Figure 2-3b represents a Meta-system for system design. The 'subsystem' to be evaluated is either a cluster of hardware components with some level of operating system, or else a simulation model of any subpart of the system that can be isolated as a logical entity. The simulated external system is a generator of requirements or demands to the subsystem. The requirements it makes of the subsystem is a function of the subsystems response, hence the feedback 'response' loop from the subsystem.

The problem of normalizing the measurements of two different systems to ensure that they are responding to the same demand is present in the construction of this system also. The simulated external system
must generate the same basic set of demands to the subsystem at each trial. This set of demands must represent demands which would be generated as a result of some realistic external usage of the system. It is difficult for the system designer to obtain the advance knowledge required to meet these conditions.

3. System Optimization by User. A Meta-system for optimization of a system by a user has the advantage of having well-defined demand characteristics available. It must be capable of evaluating a model of a proposed system, which is the operational system, modified by the user. The model of the system must be as detailed as the models used in the design stage, and include the entire system, because in most cases, modifying any part of the system will create different operating characteristics throughout the whole system. The changes that will be made to this model will be only of the class of changes that a user can effect.

The model need not be designed with the flexibility of the model for the design Meta-system. Figure 2-3c is an outline of the Meta-system for system optimization by the user.

Meta-systems of the first class have not been implemented, largely because of the expense of designing and running tests with many users at terminals simultaneously. Also, manufacturers are reluctant to allow users to run such tests when they can sell systems based on their own tests and promises.

The theoretical approach to the design of such Meta-systems is clear, and no great simplifications are evident. One possible simplification lies in the construction of a system to simulate the operation
of many on-line users. This Meta-system has been implemented in several cases, but no implementation is general enough to become operational on more than one computing system. This is because each system must be addressed by a different command language, and contains different problem languages. Furthermore, the implementation of the hardware interface with the system is a task large enough to invalidate the claim of simplification.

The Meta-System

The Meta-system developed in this work will find its primary application in the third area - system optimization by the user, although it will be implemented during the design stage.

This Meta-system could be designed to apply to a particular operational system. However, the expense involved in analysis for the information-gathering portion of the system and construction of the simulator part of the system may well make this economically unjustifiable. Therefore, the Meta-system will be developed along with the development of the operating system. The measurement portion of the Meta-system is more easily implemented while the system is being developed, and the system simulator that is used as a design aid can be modified to fit the requirements of the user Meta-system.
CHAPTER 3
OVERVIEW OF THE META-SYSTEM DESIGN

This chapter is an overview of the Meta-System design. It begins with the considerations of measurement and evaluation that lead to the Meta-System concept. Then, a brief functional description of the Meta-System is given. Last, some of the specific problems involved in the design are reviewed.

The level of detail of this overview does not require an understanding of the specific system for which this Meta-System was designed, which is presented in the next chapter. The concepts reviewed in this chapter are those that would be applicable to the detailed design of a Meta-System for any one computer system. When the specific design problems of this Meta-System are reviewed in the latter parts of the chapter, the specific problems are related to the class of difficulties that will be present in any attempt to create this type of execution-simulation system.

Meta-System Overview

The Meta-System may be conceived of as a feedback loop on an operating computer system, in which the functions of measurement, evaluation, and modification take place. These functions are discussed in the following sections.

The Measurement Function of the Meta-System

The study of measurement techniques of operational systems resulted in the following set of requirements for the measurement func-
tion of the Meta-System.

1. It should be implemented by software techniques. The measurement of the logical or decision-making functions of the operating system will require recognition of specific data items, and decision-making capabilities in the measurement devices. Also, the measurement device must be capable of handling various measurements and conditions of operation. A software device is therefore indicated. To avoid the expense of an additional processor the measurement software will be multiprogrammed with the system being measured.

2. It should introduce little artifact.

3. It should record all information of interest. The complete specification of the information of interest cannot be achieved until the entire Meta-System, including the system modifications to be evaluated, are specified.

4. It should be amenable to flexible off-line analysis (i.e., information must be detailed).

5. It should be flexible, so that the same general approach could accommodate new and more specific areas of investigation.

The choice of the measurement that meets all of these constraints is the event trace. The event trace will be more precisely defined later. For the moment, it is a record of the important events in a computer systems operation. The events are the operation of particular hardware devices within the system, or calls on significant subroutines.
It is recorded at various points in the operating system by writing a small amount of data specifying the nature of the event, and the time at which the event occurs, into a buffer area in main memory. The buffer is written to external storage whenever it becomes full.

If the events are properly defined, the event trace may be an extensively detailed record of the systems operation. It is, however, raw data, and will be processed by the evaluation function of the Meta-System.

The Evaluation Function of the Meta-System

The following possibilities for the evaluation function of the Meta-System are apparent:

1. If the event trace is a record of the time of the beginnings and the ends of each interaction of the user tasks, the evaluation function may condense this data to obtain the response time distributions for the user tasks.

2. If the event trace is a record of the beginnings and terminations of utilization periods of the hardware devices, the evaluation function may condense this data to create a record of the utilization factor each system component.

In either of these cases, the actual evaluation will be obtained by comparing the condensed data with some standard. Since the purpose of the evaluation is to determine the modifications that will improve the systems performance, the standard for evaluation should be the same data taken from variant versions of the system, especially modified versions in which performance might be improved.
The evaluation function should satisfy these two requirements: it should indicate potential performance improvements to the system designers, and it should indicate the performance of the modified systems, without the expense of performing the system modifications.

The Meta-System described thus far has the form indicated in Figure 3-1. The evaluation function contains a 'trial modification' loop in which measurements of variant versions of the system are obtained. Another possibility for the evaluation function suggests itself:

3. The event trace may be a record of the user tasks' demand for the various system resources, and the evaluation function may be a simulator of the system providing the system response to the demand, in terms of condensed data, such as 1) and 2) above.

This is the evaluation function that will be used in the Meta-System design.

Possibility of an Automatic Meta-System

Ultimately, the evaluation of the system is performed by the system designers in the Meta-System loop. It is conceivable that the Meta-System might automatically perform the optimization process of selecting the most beneficial set of modifications to the system from a given class of allowable modifications. However, this would require that either:

a) the performance function over the set of modifications as independent variables must be continuous, and have no local maxima except the maximum value; or

b) the performance function can be evaluated over all
Figure 3-1
The Meta-system

combinations of allowable modifications.

Neither of these conditions can be met for systems as complex as modern multiprogrammed computer systems. Many simulations and actual measurements of computer systems show few regular and predictable variations of performance with even simple parameter changes, and combinations of hardware and software modifications can be expected to cause performance vary even more erratically. [N2, O2] In the second case, an exhaustive search for maxima would require an overwhelmingly large amount of computation as long as the range of allowable modifications is wide enough to be useful.
Functional Description of the Meta-system

The three parts of the Meta-system described in this report are: the recording mechanisms, the preprocessor of the event trace, and the simulator. The following paragraphs provide a functional description of each. This will also provide an overview of the operation of the entire Meta-system.

The Recording Mechanisms

Measurement of the system operation is performed by small open subroutines embedded within the operating system. The subroutines record the significant events in the systems operation. An event is composed of the following data items:

1. the time
2. an identification of the task (task number) for which the event occurred
3. the identification of the type of event
4. data associated with the event.

A few examples of event types are the following: (In each case the time and type are recorded. Task number is recorded for each event type except 'idle'. The data field may or may not be recorded, depending upon the event type.)

The 'on' event, signifying a task gaining control of the processor. No data is associated with this event.

The 'idle' event indicating the beginning of a processor idle period. No task number or data need be recorded with this event.
The 'I/O-req' event, which is recorded when a task (or a system routine) requests a physical I/O operation. The data associated with this event is a unique representation of the physical address (e.g. device, cylinder, track) involved in the operation. An 'I/O-req' event is not synonymous with the initiation of an I/O device because the system may delay the actual operation.

The page fault or 'pf' event, indicating the necessity for a demand paging operation. The data, in this case, is the virtual address of the page required.

The event of a request for a Logical I/O operation, or 'LI/O'. This event is recorded when a task calls on a system routine to perform a Logical I/O operation. The data associated with this event is the logical specification of the record required (e.g. filename, record name). The data is, essentially, the input parameters to the LI/O routine.

Other events complete the specification of the system's operation. The set of all events that occur during the operation of the system, recorded in the order they occur, is known as the system event trace.

The Preprocessor

The system event trace is first pre-processed before becoming the input to the simulation model. Both the preprocessor and the model are run off-line.

The preprocessor accepts the system event trace in mass storage as input. The preprocessor has two functions:
1. To decompose the system event trace into event traces representing the resource demand of each task.

2. To 'purify' the task event trace. Since the task event traces will become input to a simulator of part of the system, the effects that part of the system in those traces should be removed before the traces are used as simulator input. The preprocessor does this. Several examples of 'system' influence (i.e., the system to be simulated) in the representation of 'system' demand, will be shown later.

The output of the preprocessor is a set of task event traces - one for each task that was active during the period the event recording mechanism was operating. The events in the task event traces are much like those of the system event trace except that:

a) The task numbers are not recorded in the events, since each event in a trace is of the same task.

b) The time of each event is adjusted to be relative to the operation of that task only.

An example of the second function of the preprocessor - removing system influence in the event trace - is as follows: One event in a system event trace is a call on a Logical LI/O routine. The LI/O routine calls on the physical I/O routine, and the Physical I/O event is recorded. This call on the PI/O routine is not due to the task, because the task specified its I/O demand at the Logical level. The
physical I/O call must be considered due to the system, and is removed by the preprocessor of the system event trace.

The Simulator

The simulator accepts the task event traces as input. The simulation model must include the operation of all of the system from the level at which the events in the trace are recorded, to the hardware of the system. Specifically, the simulator consists of:

a) A Clockwork, which selects the next event from the task traces and increments the simulation time.

b) An Event Analyzer: the analog of the interrupt analyzer in the actual system.

c) The Event Response Routine: models of the operating system routines.

d) The Hardware Section: Representations of the system hardware devices.

The output of the simulation is model is the data that allows evaluation of the system and isolation of areas of possible improvement. This data consists of:

1. Utilization factors of the various devices.

2. Response time characteristics for the task interactions.

The utilization data is recorded in the simulation model, by summing the simulated operating and idle times of each hardware device. Response times are calculated by the difference between the simulator time at which the first 'on' event of the task is accepted by the model, and the simulator time at which the 'terminate' event is accepted.
This data, obtained from the model, may be compared with the same data taken directly from the operating systems. An example of this data collection from the actual system is presented in the description of the Hardware Event Trace, in Chapter Five.

Levels of Meta-System Awareness of System Operation

It is obvious that the level of detail of the simulation will depend upon the class of modifications that is being contemplated.

Since the event trace, after some pre-processing, will become the input to the simulation model, the definition of the events in the trace will depend upon the extent of the simulation model. If the events in the trace are representations of some aspect of the original system's operation (such as the operation of a hardware device), and that aspect of the system is altered in the simulation model (i.e., the device characteristics are changed), then the event trace is irrelevant and useless to the simulation. To be useful, the events must rather be representations of the user tasks' requirements or demand for that aspect of the systems operation. The term 'system resource' which usually indicates the hardware device of the system, may be extended to include any 'aspect' of the system's operation that may be of interest. Specifically it may include system service macros, the scheduling routine, the loader, or a compiler. Therefore, the definition of the events in the trace are seen to depend upon the definition of system resource that is used for the specification of resource demand.
Thus, it can be seen that all of the following are inter-related:

the class of trial modifications
the extent of the simulation model
the definition of the events in the trace
the definition of system resource used to specify resource demand.

In the course of the design of the Meta-System it became apparent that these four quantities could be specified at several different levels, which could best be differentiated by calling them different levels of Meta-System awareness of the system operation.

At the lowest level of awareness, only changes in speed or configurations of the hardware devices are potential modifications, and only the hardware and the scheduler of the hardware devices need be included in the simulation model. Any program calling for a hardware operation will be considered a user program, and the user programs' demand for system resources is the demand for hardware operations. The events in the trace, in this case, will be occurrences of the requests for hardware operations.

At the highest level of Meta-System awareness - total awareness of system and user programs - any modification to the real system may be made to the simulation model, since the simulation will be total and the model as complex as the entire system. The events in the trace will be defined in terms of instructions or commands written at the terminals, and the system resource defined as all of the programs that respond to these commands.
Between these two levels, several more practical levels of Meta-System complexity may be demonstrated.

The design of the Meta-System will include the determination of the Meta-System level. The factors that determine the Meta-System level, listed above, must be selected to be mutually compatible. The set of events in the trace must be a representation of the demand for system resources that is independent of the allocation of the resources. Thus, the resource may be altered in the simulation model, while the demand for the resource remains constant.

In this work, the outline for the selection of Meta-System level is presented, along with the design of the framework of the Meta-System, which is independent of the level. The procedure is largely demonstrated by specific examples: four levels of Meta-System awareness are presented in the succeeding chapters, and considerations that allow generalization of the definitions, so that higher levels of awareness may be established, are reviewed.

Design Problems of the Meta-System

The analysis of the operation of time-sharing systems for purposes of implementing the Meta-System centered on isolating representations of user task demand for system resources that are independent of the allocation of the resources.

A Fortran program, being a logical, machine-independent specification of a problem, is one such representation. It is the representation of the demand for the logical, or information, resources of the system - the highest level of system intelligence. But to determine the demand
for other logical or physical mechanisms that the Fortran program implies, requires knowledge of the transformation (provided by the system) of this logical specification to the lower level resources. The demand for the hardware resources that is represented by the program can be known through either running the problem on the system to be evaluated, or performing a detailed simulation of both the operation of the Fortran compiler and the execution of the code generated by the compiler. The motivation for the Meta-System design is to avoid the level of effort required in either of these alternatives. The high-level language representation of a task's requirements is too general for use in the Meta-System.

Other lower level representations of the task demand are obtained by viewing part of execution of the Fortran problem to be the 'task' (even though the instructions being executed may have been coded by a system's programmer; as would occur during the compile phase) and the rest as the 'system'. The user task demand is given by the calls on the system functions.

Lower level representations of task demand, however, are not easily separated from the operation of the system. There are feedbacks from system to task: allocation of system resources to the task modifies the task demand for system resources. Most of the problems encountered in the design of the Meta-System are due to the system influence in the 'pure' or system-independent representations of user task demand. The system influence is removed, in each case, by one or a combination of the following techniques:
a) Construction and placement of the event recording mechanisms in the system to either exclude the system influence, or include supplementary information so that it may be removed later.

b) Removing the system-influence in the preprocessor of the event trace.

c) Carrying the system influence into the simulator, but designing the simulator to neutralize its effect.

The following paragraphs outline some of the problems encountered in defining, extracting, and using representations of resource demand, and the solutions to them. Other problems, relating to the efficiency and practicality of the technique are also discussed.

**Task Identifications in the Event Trace**

A basic function of a multiprogramming operating system is the scheduling of each task's use of system resources. Thus, the way in which the events of each task are intermingled in time is due solely to the influence of the system.

In another instance of system operation - specifically, the one provided by the simulator - one task may run faster or slower, relative to the performance of the others. The simulator must therefore view the representations of each task's demand separately.

The first way in which the representation of resource demand is purified, then, is to separate the resource demand of each task, into 'task event traces', in a preprocessor of the simulator input. In order to do this, each event in the system event trace must be identified
with a task.

The event recording mechanisms are therefore placed in positions within the operating system at which an identification of the task is known. This is no great obstacle to the implementation of the measurement portion of the Meta-System. In most cases, the Task Control Block for the task being operated on is immediately available to the operating system routine. When it is not, some unique representation of the task such as task number or TCB address, is always maintained by the system, and may be used as the task identification.

Some events are caused by the system only, and yet must be recorded in order to complete specification of task demand information. For example, the event of the processor beginning to idle need not be associated with any task, when it is recorded. Later, the 'idle' event will be used as a 'task relinquishes processor' event. The preprocessor of the event trace, having knowledge of which task is on the CP, will complete the specification of the event.

Representation of Processor Time Requirements

A task's demand for the processor is given by the number and type of instruction executions it requires. Since it is neither practical nor necessary to count and simulate the execution of the individual instructions a task's processor demand is taken to be the processor time required by the task.

(1) Modifications internal to the processing unit will not be considered here.
The measurement of the time a task spends in the processing, however, is influenced by the amount of memory interference due to I/O operations into memory, taking place simultaneously with processing. Therefore, the tasks processor demand will be defined to be the time the execution of the task would take if no memory interference were present. The system influence due to simultaneity is eliminated in the preprocessor of the event trace. The preprocessor calculates the 'pure' processing time, as follows: Let \( m \) be the memory speed (cycles per second) and \( c \) be the average fraction of memory cycles needed by the processor. Then with no memory interference, processing for a period of \( t \) seconds will spend \( ct \) seconds utilizing the memory.

Now suppose I/O operations taking \( k \) bytes per second are being performed in the background. The time for the processors use of memory will be expanded by a factor of \( \frac{m}{m-k} \). The total time \( t' \) taken to perform the original \( t \) seconds of processing is:

\[
t' = (1 - c + \frac{cm}{m-k}) \cdot t
\]

The quantity \( (1 - c + \frac{cm}{m-k}) \) will be called the memory interference factor \( f \).

Each of the event traces taken from the system will contain the actual time taken on the processor, \( t' \). But, in order to isolate the task requirement for processor time - \( t' \), the trace must also contain an indication of the amount of I/O being performed simultaneously with processing, so that \( f \) may be known during each interval of processing.
time. Each event trace is constructed to contain some account of the I/O activity and the calculation of \( t \) from \( t' \) and \( f \) is performed in off-line processing of the event trace.

**Specification of Memory Requirements**

The specification of the hardware resource requirements made by the user programs - either directly or via a call on a system routine - are generally quite unambiguous. The specification of memory requirements is an exception.

A task's memory requirement is actually one word - instruction or data - at a time. For obvious reasons, memory must be allocated in larger units - in paging systems, one page or block at a time. The requirement for a page of memory - when the page is not allocated, will result in an unambiguous specification of demand - the page fault. But the system cannot know whether the demand still exists one memory cycle after the page has been allocated. Hence, the system itself specifies when pages should be de-allocated. It will generally do this by assigning a probabilistic value to the demand for the page and deallocating the block when either one of these conditions is met:

a) When it becomes known that the page is no longer needed;

b) When some other task has a demand for the memory block occupied by the page, which is greater than the probabilistic demand for the page;

c) When it is known that the page will not be needed for a period, and it is likely that condition b) will be met before the end of the period.
If these deallocation judgments are made optimally, a page fault for a particular page will not occur soon after the deallocation of that page.

The record of page allocations and deallocations, then, is an inexact specification of the task’s demand for memory: it shows a large degree of system influence. However, it is the only record of memory demand available without special hardware to monitor memory utilization. This example of system influence is not removed during preprocessing of the event trace, but is removed by the simulation model, and removed only when necessary.

The simulator will, in general, handle memory allocation differently from the allocation shown in the event trace. If, during the simulation, the task trace shows a page fault for a page that the simulator has already allocated, the page fault event is simply skipped. On the other hand, if the simulation model deallocates a page when it was not deallocated in the real system, the simulator must impose a potential page fault on itself. It replaces the page fault by evaluating the page re-use time as a random variable. The specification of the random variable is made from the average value of the page re-use times of the previous and next re-use times for the page that are available in the event trace.

Entrances and Exits to System Routine

Higher levels of specification of user demand for system resources are demands for system functions. The events indicating these functions
are recorded at the entrances to the routines performing the functions. The operating system is written as a set of recursive subroutines so that a call on one system routine may result in calls on several others. If the original call on a system routine is taken to be an indication of user program demand, then these secondary calls, which are not made directly by the user program, may not be considered user demand. The events corresponding to these calls are system-contributed data, and must be eliminated from the event trace. A method of distinguishing user program calls from system program calls is required.

In order to distinguish system program calls from user program calls on the system functions, both the entrances to and exits from the system routines are recorded as events. Off-line processing of the event traces will remove the secondary calls that occur between the entrance event and exit event of a particular routine.

In order to place these event recording mechanisms in the system, the entry and exit points of the system routine of interest must be identified. The identification of entry points is straightforward. The identification of the exit points of system routines is, in general, a difficult problem. Each transfer of the following types must be analyzed to determine whether it should be considered an exit from the subroutine:

a) Transfer to the return address provided by the standard subroutine call.

b) Transfer to any address provided as a parameter to the subroutine.
a) Subroutine structure of O.S.

b) Classes of Subroutines

c) Definition of Meta-System Levels

Figure 3-2
Meta-System Levels Due to Subroutine Structure
as events. The routines that are called only from higher-level routines within the Meta-system awareness, need not be recorded, even though they may have been recorded in a previous, lower-level Meta-system.

The set of levels of Meta-system awareness that may be chosen for the subroutine structure of the example is the following:

a) any of the 7 subsets of \([K,I,D]\), the lowest classes (excluding the empty set)

b) \([GHJ, I]\) or \([GHJ, I, D]\).

If the class \(GHJ\) is chosen, then the lower class, \(K\), need not be recorded since it occurs only as a consequence of \(GHJ\). The other lower class, \(I\), must be recorded, since it is called from above \(GHJ\).

c) \([ABCEF, GHJ]\)

The user programs call on this set of system program classes. All other classes result from calls on this set, therefore, calls on these other classes need not be recorded.

It must be remembered, however, that the Figure 3-2b is a structural representation of Meta-system classes, and therefore provides only an estimate of the number of events which will actually be recorded. The volume of recorded data will depend upon the frequency with which control

---

1 This does not imply that the entrances and exits to every subroutine of such a class must be recorded, because a finer analysis (e.g., the operating system structure of Figure 3-2a) may show that only several of the routines of a class are called from above the Meta-system level. The analysis by class is a first approximation to specify a set of routines that need not be recorded. It is true, however, that if one routine of a class is included within the model, then each routine of the class, and all classes below it, must be included in the model.
passes through the meta-system level. Also, the calls on the subroutine classes are recorded at the entrance to the subroutines. Thus, if the meta-system level crosses one arrow leading into a class, the level will in fact cross the other arrows into that class as well, whether or not this is intended in the definition of the level. For example, in Figure 3-2c, two levels are shown. One, the higher level, is inefficient, since some of the calls on the GHJ class result from the previously-recorded ABCEF class. In this case, a call sequence from the user program to F to G is recorded twice. The lower meta-system level is efficient.

Modeling System Routines

The simulation model will include the models of some of the system routines. In order to preserve the economy inherent in simulation (as opposed to implementation and testing) the model of these routines will have to be somewhat simplified. Yet the important aspects of their operation must be duplicated.

In particular, the decisions that ultimately result in hardware resource allocation must be included within the model.

Simplified versions of the significant aspects of system routines are presented in the description of the time-sharing system - Chapter 4. It has been estimated that 60% of the code in the executive of a multiprogrammed operating system exists for the purpose of error checking. It may be assumed that the paths resulting from the error checks are taken rather infrequently, therefore, they are not a great influence on the resource allocation process. These error checking paths may be
omitted from the versions of the system routines in the model. Likewise, security considerations, in file operations, do not influence the location or identification of a particular data item. It may be assumed that the frequency of file operations being blocked for security reasons is small enough not to influence the utilization data obtained in the simulation; therefore, it may be omitted from the model.
CHAPTER 4
THE TIME-SHARING SYSTEM DESCRIPTION

In this chapter, the design of a sample time-sharing system is presented. The system has been abstracted from three current time-sharing systems: TSS on the RCA Spectra 70/46 [G2-3,R4-5], TSS on the IBM 360/75 [L3,12-4], and the Multics system on the GE 645 [C10-12, D1-2, O4, V3]. The parts of a time-sharing system that most heavily influence physical resource allocation are detailed in this design. Thus, the following features are shown explicitly: multiprogramming, demand paging, multi-tasking, multiple-paths to I/O devices, dynamic loading, a hierarchical file structure, and a virtual access method. Algorithms which vary widely from system to system, such as the Task Initiator or Dispatcher, and the Page Replacement Algorithm, are not shown in any detail; however their position and function within the system are made explicit.

The Meta-system, detailed in the following chapters, will be constructed to operate with the time-sharing system presented here.

The Hardware of the Time-Sharing System

The multiprogramming system under discussion is a system with a single processor. The results that are obtained from the discussion may be extended to a multiprocessor system. The principal method of main memory allocation is demand paging. The four main hardware sections are:

1. the processor (also called the Central Processor or CP)
2. the main memory (also called executable memory or core memory)
(3) the secondary storage (also called backup or swapping memory)

(4) the I/O channels and devices (also called external devices)

Each will be discussed individually in the following sections.

The Processor

The processor unit of the computer system may consist of one or more general-purpose processors. If there is more than one processor, each is capable of handling all of the processing requirements of the system. For example, each processor may perform both fixed and floating point computations. Only one processor need be interrupted by non-program interrupts, and this processor is capable of interrupting any of the other processors.

In order for a task to gain control of the processor, the following operations must take place:

(a) The general-purpose registers, that are controlled by the program, must be set to the proper values for the task.

(b) Other registers of the CPU, such as those that contain information determining the task's privilege, must be set.

(c) The hardware device performing the mapping from virtual address to core address must be activated. For example, a translation table, a hardware associative memory, or a pointer to a page table in core must be loaded with the correct values for the task.

(d) The program counter, containing address of the next instruction to be executed must be set to the next address to be executed within the task. This action is actually a transfer to the task.
When a task gives up the processor, all of the above information must be saved. Switching tasks on a processor thus involves some overhead, but it is small compared to the average length of a processor application for a task.

The amount of processor time that is needed for a computation will be proportional to the amount of data traffic to and from the memory unit that the processor is using. The external devices will, in general, be rotating and will have tight tolerances on the time in which each byte must be transferred. Therefore, they will take memory cycles as they need them. The processor will then be delayed by the amount of memory cycles that are taken by the channel.

The **Main Memory**

The main memory is divided into units of uniform sized blocks. Each unit can be addressed as if it were a contiguous part of a sequential address space, by means of address mapping hardware. A set of blocks that have been allocated for a task may be isolated from all others, by the memory protection mechanism, so that computation within that set may not reference any location outside of it, and vice versa. Processing, I/O, and paging operations may be performed on the main memory simultaneously. The blocks of main memory will at all times contain the following:

1. Programs and data areas which are either being used by a processor, or are awaiting execution by a processor.
2. I/O buffer areas, which are either awaiting an I/O unit, or channel, or are transmitting their contents to, or receiving
them from a channel.

(3) Programs for which there is currently no user, but are being kept available in case they are requested. This could be because they are needed with rapid response, or because the expectancy of use is high.

(4) Programs and data areas of tasks which are awaiting completion of an I/O operation or other operation which blocks processing.

(5) Nothing useful. These are available for allocation.

A feature of modern computer systems architecture is the division of main memory into banks which are separately accessed.\[M5,M9]\ This enables channel operations to take place in one bank without interfering with processing in another. Such a system will not be considered directly, even though it adds another dimension to the problem of resource utilization. This discussion of the processing system could largely be generalized to be applicable to a system containing \(n\) memory banks, if \(n\) different memory demands are considered each time the 'main memory' demand is mentioned here. For example, if a paging operation must take place in this system, it must be determined whether any of the \(n\) memory banks in the generalized system would require the paging actions, and if so, to which should it be directed. This will depend upon the rules by which the banks interact with each other and the rest of the system, which are not readily generalizable.
The Secondary Storage

The backup memory may be either a drum or bulk core. Recent reports of systems employing large core storage as a backup to primary memory seem to indicate a significant improvement in overall performance \[J1, M1\] especially in demand paging systems.\[L2\] Most multiprogramming systems for time-sharing use still utilize the more inexpensive drum systems.

We will assume that the large core system is truly random access - any location can be accessed with the same access time, which is very small in comparison with the drum unit. The drum will have pages distributed on \(n\) different sectors from which an operation can begin periodically once every revolution time. Only one page can be read from a sector at any time. Thus, if two page requests for one sector occur simultaneously, one will be satisfied after a latency period, averaging one-half a revolution period; the other must wait one entire revolution time longer.

The main memory will be served by one paging channel. This will be generalizable to many paging channels, with some effort, as in the case of multiple memory banks.

The paging channel itself will at any time be performing one of the following activities:

1) Demand page-in, the highest priority activity. Bringing a page into main memory in order to allow a task in main memory to continue processing.
2) Pre-paging. Bringing pages of a task into main memory before the task is given the processor.

3) Page-out. Removing a page from main memory to the backup memory.

The I/O Channels and Devices

The I/O section of the system contains multiple data channels, serving random-access devices of varying sizes and speeds. The assignment of units to channels is not unique; in general, it will be possible to access one I/O unit through different channels.[L3]

The I/O units include the communications unit, which provides the interface with the user terminals. Because the terminals are very slow compared with the channels, all the terminals will be multiplexed on one communications unit, which may itself be multiplexed with other slow-speed units on one channel. Communications with the user terminals create a relatively constant load on the communications unit, because of the slowness of the terminals. The under-utilization of the communications channel or device, due to under-utilization of the terminals, must be tolerated; at least, it cannot be remedied by modifications internal to the system. Therefore, the scheduling of this channel will not be a significant part of the overall resource allocation problem.

The random access I/O devices are not as fast as the paging device, although their transfer times are of the same order of magnitude.

The Generalized Operating System

The operating system runs in a special state, called the 'master' or 'supervisor' state, as opposed to the 'slave', 'problem' or 'user'
states in which the application programs run. The master state has
capabilities not available to the problem programs, such as the ability
to issue orders directly to hardware, and to access all program and data
areas, whether external or internal. Control of the machine is transfer-
red from the user state to the processor state by means of the auto-
matic interrupt, or a special program-forced interrupt called the super-
visor call, or SVC.

The operating system allocates the various system hardware resources
to the tasks operating within the system. Another slightly different
point of view is that the operating system schedules the tasks for the
hardware resources. The operating system maintains a representation of
each task in a Task Control Block or TCB. In the following, it will be
assumed that all the information relating to a particular task, needed
by the operating system, is maintained in the TCB. Each task will also
be characterized by a mapping from virtual addresses for the virtual
memory allocated for the task, to the physical addresses of the main
memory blocks or secondary storage. This will be called the task's
Page Table.

Resident Parts of the Operating System

When an interrupt or SVC takes place, the interrupt-handling hard-
ware of the CP causes information pertaining to the interrupt, such as
the location at which the interrupt occurred, and a number identifying
the type of interrupt, to be placed in program-accessible CP registers.
The hardware then transfers control to a particular location, within the
supervisor, which is the beginning of a routine called the Interrupt
Analyzer.
Figure 4-1

Resident Parts of the Operating System
The Interrupt Analyzer identifies the type of interrupt, and transfers to the appropriate interrupt-response routine which processes the interrupt. This may be done, for example, by consulting a branch table, which contains the virtual addresses of the interrupt response routines, with the identifying number of the interrupt as an index. (See Fig. 4-1.)

Branching to an address of an interrupt response routine could be done by the hardware interrupt mechanism directly, by merely setting the location counter to the proper location within the branch table.

When a program references a virtual address for which no block has been allocated, the page fault interrupt occurs. The response to the page fault is the page-fault-interrupt-response routine, or the Paging Routine. The Paging Routine must find the address in secondary memory of the virtual memory page required. To do this, it must have available a table, which is a mapping from virtual address to secondary storage address for the task using the processor. This will be called the virtual memory map.

The paging routine must also find an available block in main memory in which to place the required page. Thus, it must have a record of the usage of all the core blocks. This will be called the main memory map. If there is no block available in core, or if the availability of blocks is too small, the paging routine must also initiate the Replacement Algorithm, which will de-allocate a block for some task.

De-allocation of a block is the logical operation of breaking the correspondence of a task and the block. If the task has not
altered the contents of the block, the operation involves nothing more than altering an entry in the table of blocks held by the task. If the task has altered the contents of the block, the contents must be saved in the backup memory before the block can be reallocated. Thus, the physical operation of reading the page out must be performed as well. Figure 4-2 shows the Paging Routine.

The **Paging-channel-complete-interrupt-response routine** (see Fig. 4-3) initiates the paging operations that were placed on the paging queue during the time the device was busy. It must also initiate the processing of tasks for which paging operations have been completed.

A channel completion which marks the end of a demand paging operation, also indicates that a program is available for use by the processor.

The paging-channel-complete-interrupt-response routine must be resident, in order to intercept the task as its paging operation is completed, and perform the actions required to place the task on the processor again. The task will not always be immediately returned to the processor, because this may not be the most efficient way that tasks may be scheduled. Therefore, the interrupt-response routine places the task on queue for the processor, and transfers to a subroutine called the **Task Initiator** which selects the next task to be given the processor, and places the task on the processor.

The Task Initiator, which is otherwise called the Dispatcher or Scheduler, is an important factor in the scheduling of all system resources, since it decides which task is to be placed on the processor.
Figure 4-2
Page Fault Interrupt Response Routine

Exit to task initiator (8 in Figure 4-3)
Figure 4-3
Paging Channel Complete Interrupt Response Routine
The Task Initiator is called under two sets of circumstances. First, some event caused by a task in the system may change the availability of the processor or the demand for it. In the previous case, for example, the paging channel completion indicates that the task which had been in paging now has a demand for the processor. Second, the operating system may call the Task Initiator at various scheduled intervals. The Task Initiator achieves this by setting an interval timer to some time value when a task is placed on the processor. At the specified time later, the interval timer will expire, causing an interrupt. The response to this interrupt will be a call on the Task Initiator, which will again decide whether to allow the task to continue on the processor, or to place another task on the processor and how much processing time to allow the task before the Task Initiator is again called. It will base its decisions on each task's position on the processor queue, its external priority, run time, amount of allocated main memory, etc. This set of decisions comprises the major part of the scheduling algorithm for the system.

Thus, the following are the parts of the operating system which must be resident by virtue of their function:

- the Interrupt Analyzer,
- the Paging Routine with its data, i.e., the main memory map and the virtual memory map,
- the Replacement Algorithm,
- the Paging-channel-interrupt-response Routine, and
- the Task Initiator.
The entire remainder of the operating system can exist in virtual memory, and be run under demand paging as a 'user' program. This includes interrupt response routines. When an interrupt occurs, for which the response routine is not resident in main memory, the transfer to the address in the branch table will result in another interrupt, for the page fault. This interrupt, and the paging operation, must be completed before the original interrupt can be serviced. The system would operate inefficiently in this manner, because paging delays would often occur for functions that should be executed immediately.

In this operating system, all the asynchronous interrupts will be serviced immediately by resident response routines.

Yet, the operating system will operate in a 'space squeezed' manner, in order to keep the maximum amount of main memory available for the users. This means that more system functions will be available only through paging, as will some system data, and directories. The program-forced interrupts, or SVC's, may or may not be met with resident response routines depending upon the frequency with which each is called.

The Operation of I/O Devices

The commands which control the I/O devices and channels are issued from the operating system, since they are privileged commands. When a user task has a need for an I/O operation, it calls for it by issuing an SVC called the Physical I/O SVC. This results in an interrupt, to which the response is the Physical I/O Response Routine, briefly called the PI/O macro. If the device is available, the PI/O
macro issues the command that initiates the operation on the device. If it is not available, the macro puts the operation on the queue for the device, for execution later, when the device becomes available.

The inputs to the PI/O macro are the physical device identification, the operation (type), and the address of the data area on the I/O device. The physical device identification is the system name for the device. This may undergo one transformation in the PI/O macro, before the hardware command is issued. The operation of the PI/O macro, from the point of view of resource allocation, is as shown in Fig. 4-4.

The random-access I/O devices are controlled by commands sent to them from the channel. The principal commands the I/O devices respond to are the following:

Seek: A seek is a mechanical positioning of the I/O device so that the physical record (track) may be read, or searched. During the positioning of the device, the channel may be idle or perform I/O operations on some other unit.

Search: A search looks for a particular logical record after the physical record has been accessed. A search involves sampling some data (the key field) on the physical track, to determine the position of the logical record field. The channel is busy during the search.

Read or Write: After the proper location on the physical track has been read, the data is written onto the track area or read from the track area.

Although the three types of commands, seek, search, and read or write are needed for a typical I/O operation, the processor need not
Initiate Physical I/O Macro
Input: Device address
       Record address
       Operation

Is device busy?

No

Initiate Physical I/O Operation

No

Path set busy?

Yes

Queue for device

Yes

Is operation to this device on path set queue?

No

Queue for path set

Exit

Figure 4-4
Physical I/O Initiation
send each command to the channel explicitly. The commands can be
organized into a **channel program** from which the channel takes each new
command when the current I/O operation is completed. The physical I/O
macro could set up such a channel program to initiate I/O each time it
is called, and then issue an 'execute channel program' instruction to the
channel.

In this operating system, the I/O operations are divided into two
steps, for which two instructions (or two channel programs) are issued.
First, a channel program consisting of only the seek command is issued.
Upon completion of the positioning operation, the channel causes a
'channel complete' interrupt signifying the completion of that channel
program. The interrupt response program then issues an instruction for
the search and read or write operations. Thus, two 'channel complete'
interrupts take place for each physical I/O operation, and the 'channel
complete' interrupt response routine takes different actions depending
upon which channel program has completed.

There are two reasons why the I/O operations occur in two steps
rather than by the execution of a single channel program. First, in
order to obtain a complete record of the utilization of the devices and
channels, it would be useful to record when the seek - for which the
channel is not utilized - ends, and the search operation begins.

Secondly, it allows greater flexibility in the I/O scheduling. If
a high priority I/O operation must be initiated on a channel that is
executing a 'seek' channel program, the 'search' part of the first opera-
tion can be delayed until completion of both 'seek' and 'search' parts
of the high-priority I/O. No channel program must be broken in order to effect this change in the order of these I/O operations.

The cost of executing the I/O operations in this fashion is an extra interrupt for each operation. In most cases, there will be no high priority interrupt, therefore, when a seek program completes, the channel-complete-interrupt-response routine will only check for the presence of a higher priority I/O operation, then issue the instruction for the already-prepared second step. Thus, the extra interrupt takes little processor time.

Management of Channel Queues by The I/O Control Subsystem

I/O devices are operated through commands issued through the channels. In general, a particular I/O device - or set of I/O devices, through their device controller - may communicate with the CP and memory via one of several channels. This allows greatly increased efficiency of I/O device operation since congestion at one channel can be alleviated by directing operations to a less utilized channel. A command to an I/O device will be delayed because of channel unavailability only if every one of the channels that could receive the command is busy. A particular advantage of a multiple-path arrangement is in the execution of seek commands during periods of heavy I/O use. Seek operations do not require constant utilization of the channel; they merely require availability of a channel for an instant in order to send the command to the device controller. Thus, if a particular I/O unit is not busy, a required seek operation for that device need be delayed only until any one of the channels that can communicate with the I/O device completes.
an operation. This will occur sooner than the completion of an operation on any one particular channel.

An I/O operation is queued for one of the two following reasons. First, the I/O device may be busy in performing a seek or search operation. In this case, the operation will be placed on a queue for the device. Secondly, all available paths to the device may be busy. The I/O operation will then join a queue for the path set to that device.

A path set of an I/O device is defined as the set of all channels which can communicate with the I/O device. We say that a channel belongs to a path set of a device if the device is connected to the channel. A channel belongs to the path set of each device that can be connected to the channel. A channel may therefore be a member of several path sets. A path set is not available if each of the channels in the path set is busy. It is available if at least one channel in it is not busy. Refer to Fig. 4-5 for an example of path sets.

There are two queues for each path set. One is a queue of seek operations to available devices belonging to the path set. This queue can become emptied as soon as one of the channels of the path set becomes available, as all the seek commands can then be sent to the devices via this channel.\(^1\) The second is a queue of search operations. When a channel becomes available, one of the search operations on queue for one of the path sets of a channel is selected. The selection procedure could be dependent upon task priority, path set priority, or position

\(^1\) As long as no two different seek operations, directed to one device of the path set, are on queue for the path set.
Queues for path sets:

Path set 1

12

123

2

23

Channel 1

Channel 2

Channel 3

1

12

12

12

123

23

23

23

I/O devices: each labeled by path set it defines

Command to device A joins queue for path set 23.

Figure 4-5

Examples of Path Sets
of the particular I/O device. This latter will determine the latency period and expected length of the operation. In general, the algorithm should favor a path set of fewer channels over larger path sets, since there will be fewer opportunities to service the smaller path sets. This selection priority will ultimately depend upon the number of devices belonging to each path set and the frequency of reference to these devices.

Channel operations are initiated when calls are made on the PI/O macro. If either the device or the path set of the device is busy when the I/O operation is requested, the operation is placed on the corresponding queue. These operations must be initiated when a device or a channel becomes free as the result of the completion of an operation.

The Channel Complete Interrupt Response

The Channel-complete-interrupt-response routine is shown in Fig. 4-6. The channel-complete-interrupt occurs at the completion of either a seek or a search and read or write channel program. If a seek program termination caused the interrupt, it means that the channel had been available to transmit the interrupt to the CP. It also means that no search operations are on queue for the channel since these operations would not have been put on queue for a channel when that channel was available. Therefore, the search and read or write operation is initiated immediately upon the interrupt marking the completion of a seek operation.

If the interrupt is for the completion of a read or write operation, then the possibility of activating other operations exists since it may be assumed that other operations were placed on queue for the
channel during the period that the channel was busy. First, other seek operations may have been completed. The interrupts for these operations were held by the devices or device controllers until the channel becomes free, and they are masked (i.e., the interrupt mechanism is inhibited) while the interrupt that marks the completion of the channel busy period is serviced. The channel complete interrupt response routine will note these interrupts, and service them by placing the search operations corresponding to them on the proper path set queue. Each of these path sets includes the channel presently terminating.

Second, there may be another operation on queue for the device just terminating. In general, it will be a new seek, but if it is directed to the same track as the operation just completed, only the search part is necessary. If it is a seek, it is initiated immediately. If a search, it is queued for the path set along with other potential search operations.

Third, new I/O operations may have been requested during the period the channel was busy. Each of these, for which the entire path set had been busy until this point, is represented by a seek operation on queue for the path set. These seek operations are initiated for each available device. If there are multiple operations directed towards one device on the path set queue, one operation is selected for initiation and the others put on the queue for the device.

Last, one search operation is selected from among the search operations on queue for this channel: the queue for the channel is the union of all the path sets that include this channel. The selection is
dependent upon I/O scheduling considerations previously mentioned.

The **Task Initiator**

The Task Initiator is the system program which decides which task will take control of the processor. It selects the task from a pool of available tasks. A task may not be available for the processor for one of several reasons:

1. **The task is either on queue for paging or is performing a paging operation because a required page is not in core.** Its return to available status is imminent.

2. **The task is awaiting data requested in an I/O operation.** During this period the blocks associated with the task may, of course, be taken by another task, or the pages may remain in core, according to the scheduling algorithm.

3. **The task is waiting for a response from a terminal.** Since this is expected to take a long time, no system resources will be allocated to it. The task is called a **dormant task**.

In case (1), the page-fault-interrupt signals the system to block the processing of the task. The page-fault-interrupt-response routine of the system then removes the task from the status of being available for the processor.

In case (2), when an active task must wait for the completion of an I/O operation, it does not enter an 'idle' loop, as it would in a batch processing system. Instead, the task issues an 'I/O Wait' SVC. This tells the system that this task should be removed from the status of being available for the processor and the task initiator may place
another task on the CP.

Likewise, when a task must await a response from the terminal, as in case (3), it issues a 'Terminal Wait' SVC. In response to the 'Terminal Wait' SVC, the system places the task in a dormant state (possibly moving it to a pageable portion of virtual memory) and transfers to the task initiator to find another task for the processor. The 'I/O Wait' and 'Terminal Wait' macros, which are the responses to the corresponding SVC's are shown in Fig. 4-7.

Tasks whose processing has been halted for paging and I/O waits are returned to the active status by the paging channel or (I/O) channel complete interrupt response routines. A dormant task is returned to active status by the routine which responds to the interrupt marking the completion of an input message from a terminal. This routine, called the Terminal-I/O-complete-interrupt-response Routine, is outlined in Fig. 4-8.

The File System: Logical I/O

The user programs in a multiprogramming system cannot have direct control over the hardware of the system. Hence they do not specify I/O operations by physical addresses of external storage devices and the addresses of data areas on the devices. Instead, the operating system assigns user files their locations on the storage devices and issues the hardware commands which use these hardware addresses. Furthermore, the operating system makes and maintains the correspondence between the user names for data and the physical locations of the data. The corres-
WAIT for I/O: input: identification of operation

Was I/O operation completed?

Place task in Status of Unavailable for processor

Yes

No

exit

to Task Initiator

Figure 4-7a

I/O Wait Macro

WAIT for terminal response:

Place task in Status of terminal I/O Wait (Dormant state)

to Task Initiator

Figure 4-7b

Terminal-Response-Wait Macro
Figure 4-8

Terminal I/O Complete Interrupt Response
pondence between the user names and physical locations of data is made in a set of system macros called Logical I/O (or LI/O) macros.

A user program specifies a data record by a (file name, index) pair. The index is the user program's alphanumeric name for the record. The system Logical I/O macros GET and PUT retrieve and store the logical records in the files. A description of the file structures of the system is necessary to understand the operation of the logical I/O macros.

The File Structure

A file is stored in external storage in three areas: a definition area, a directory, and data area. The definition contains the information concerning the way the file is structured, and the address of the file directory. The definition of a file is stored separately from the remainder of the file; it is stored on the first track (or one uniquely addressed track) of a random-access device. When a file is used for the first time, it is necessary to access a particular track to ensure that the required volume is mounted on the proper storage device. This particular track will contain the definitions of every file on the device; it is called the Volume Table of Contents (VTOC).

The file directory is the set of (index, physical address) pairs that defines a mapping from the index of each record in a file to the address of the physical residence of the record in external storage.

The file directory is a mapping from indices to physical addresses. It may be structured as a multi-level mapping, if it is too large to be contained in a single physical data area (track). If the indexes of records within the file are arranged in alphanumeric order, and the
set of (index, physical address) pairs extends over more than one track, a two level mapping is necessary. The second level will be the pairs that define the mapping, extending over as many tracks as are necessary. The first level is the set of (max-index, second level track address) pairs, where max-index is the largest (and last) index on the corresponding second level track.

Figure 4-9
A Two-Level Directory of an Index Sequential File
The records of the file are stored on the data tracks. If more than one record is stored on a single track the records on the track are ordered lexicographically by their indices.

The file structure is recursively extendable in the following way: the records of one file may become the file definitions of another. Thus, in Fig. 4-10, the file B may be viewed as an extended record of file A. In this case we say that file A is the predecessor of file B and B is the successor of file A. The predecessor function P is a mapping from a file to its predecessor.

\[ A = P(B) \]

A user program may specify file B by the qualified file name A.B. The combination of the file A and all of its successor files may be considered as a generalized file. The directory of the generalized file is the combination of file A and all the lower-level file directories. The records of the generalized file are the union of all records of the lowest-level files within the generalized file.

Most of the files encountered in the time-sharing environment are small enough to be represented by a single-level directory. The discussion of the Logical I/O operations following will consider only this case, although they may be readily extended for the cases of multi-level files.

Address Transformation of User Files

The operating system maintains a list of all the files in the system in a System Catalog (Syscat). The System Catalog is a special, multi-level generalized file. It takes the form of a nonhomogeneous
Figure 4-10: Generalized file A

Figure 4-11: Abstract structure of Syscat
tree, that is, starting from the root of the Syscat (the file whose
definition is not a record of another file), the number of files that
must be searched in order to reach an end point (which is a user file)
is not the same for each end point. Figure 4-11 is a representation of
the Syscat as a tree.

A particular file may be specified by a fully qualified file name;
that is, the name may be composed of the concatenation of several file
names each of which specifies the branch that is taken in the Syscat
tree in the search for that file. For example, in Figure 4-11, the
file name A.B.E.F.H. is a fully qualified name; it specifies one user
file. The name A.C. is partially qualified; it specifies a file of user
files.

The name of the user that owns a particular file appears as some
part of the fully qualified file name. A file named after an owner of
any file therefore appears at some level of the Syscat. This file is
the root of a subtree of the Syscat tree, which defines a generalized
file whose elements are all the files owned by that user. This root will
be called the user's file of files (FOF).

The first level of Syscat is a file with a multi-level directory;
the first level of the directory is resident in system virtual memory,
and the remaining directory levels are resident on an external storage
device. The records of the Syscat are the file definitions of user
private and shared files.

Figure 4-12 is a representation of a useful System Catalog. It
has two levels for each of the files that are owned by a user, and one
Figure 4-12
Structure of Syscat
level for the common files. The records of the first level of the
Syscat are therefore either the definitions of the FOF's for the
various users or the definitions of common files.

This is the Syscat structure that will be used to demonstrate
the operation of the Logical I/O macros. It has the features of a
multi-level structure, and yet, for most time-sharing applications, two
levels of file structure are sufficient. The generalization to an
arbitrary multi-level structure is straightforward.

**Logical I/O Macros and Operations**

In general, a user program requests I/O operations by issuing
an SVC for the GET and PUT macros. These macros bring or store a record
from or to the file, respectively.

The input to each macro consists of

(a) the virtual memory address to or from which the record
    is to be moved,

(b) the file name, and

(c) the index of the record within the file.

During the time that a file is active for I/O operations, the
system maintains information relevant to the status of the file in a
**File Control Block (FCB)**. The FCB contains the file name, a pointer to
the file definition or file directory, and the status information re-
garding those parts of the file that can be found in virtual memory as
a result of previous I/O activities. The latter is referenced by the
Logical I/O macros to determine how many, and which physical I/O opera-
tions must be executed for the macros.
The GET and PUT macros assume that files to be operated on have previously been opened. Otherwise, an error condition will result. Basically, the file opening operation has the effect of placing the file definition and the address of the directory of a file F in the FCB. The macro 'GET F, X' will then perform the following steps:

(a) Bring the directory of F into virtual memory.

(b) Locate the track containing the record whose index is X in the directory.

(c) Bring this track into virtual memory.\(^1\)

(d) Relocate the record X in virtual memory.

The next time the GET macro is called, 'GET Y, F' for example, it will not be necessary to perform step (a) because the directory of F already resides in virtual memory. Likewise, it is not necessary to perform step (b) if record Y is on the same track as record X, since this track is resident in virtual memory.

If the record is not found in the file, an output parameter of the macro is set to this effect. The GET macro is outlined in Fig. 4-13.

The GET macro therefore determines which part of the file, if any, is resident in virtual memory before issuing physical I/O instructions to bring the file directory or data track into virtual memory.

---

\(^1\) It is not imperative that the entire track be brought into virtual memory, since the hardware may search for the required record and bring just that into virtual memory. However, this method will conform to the method of greatest efficiency for sequential processing of the file. In sequential processing, it is assumed that if record X is obtained, records Y and Z, which are on the same track, will be required imminently, therefore, they might as well be obtained with record X by reading the entire track.
Since a file must be opened before its GET macro is called, the user task prepares the file for activity by issuing an OPEN SVC. The response to this SVC is the OPEN macro which is outlined in Fig. 4-14.

The OPEN macro calls the GET macro to obtain the definition of the file F. The input parameters to this call are the file name and the record index in the form: GET P(P'),F. The OPEN macro then places the relevant data from the file definition into the FCB, and the file F is thus opened.

If a FOF was the input to the call on the OPEN macro, the file P(FOF) is in fact the Syscat. If the file F is not found in the file P(F), a search for the file F is initiated throughout Syscat (first level), considering the file F to be an unqualified file name.

The entire operation of the OPEN and GET macros is summarized in an example shown in Fig. 4-15. It is assumed that the user, for whom the task is defined, and the owner of the files needed by the task are the same. The first time a user opens a file, the user's FOF is opened. The Syscat is searched, via a GET macro specifying the user's name. The operation (1) in Fig. 4-15 brings the second level of the Syscat into virtual memory. The operation (2) then brings in the Syscat record, which is the user's FOF definition.

Next, the file A is opened. This occurs by means of a GET OF,A macro. The operation (3) in Fig. 4-15 brings in the directory of the

---

1 This is not necessarily the case. The two individuals' names may, however, be equated in the LOGON process, and the owner's name will then be available in the TCB. The operation is subject to security and privacy procedures not covered here.
Figure 1.11
OPEN macro
OF. This is the listing of the names and locations of all the user files. This listing will remain resident in virtual memory and serve as the User Catalog (Userrcat). From then on, all user-specified file operations will first consult the Userrcat before searching for file names in Syscat.

Having found the external address of file A in the Userrcat, the file definition of file A is brought into virtual memory by the operation (4) of the figure. File A is then opened. Next, the Userrcat is modified to replace the external address of the file definition with the virtual address of file A's on FCB.

Lastly, the Get macro GET A,X involves two operations: (5) and (6) in Fig. 4-15. The operation (5) will now find the external address of the directory of file A in Userrcat and bring the directory into virtual memory. The operation (6) will bring the track containing the required record X into virtual memory. Later, when the GET macro is called again for a record in file A, not all of the physical I/O operations that occurred during this call on the GET macro will be necessary.

The PUT macro restores a data record to the file. Its operation is quite similar to that of the GET macro. It will not be made explicit here.

The Virtual Access Method

Until this point, the logical file operations discussed were operations on medium to small files whose records were usually smaller than the standard block size. To handle large files an extension of the preceding methods may be realized by replacing the single physical
I/O operation for the one-level directory by several physical I/O operations for multi-level file directories. However, a more significant way of performing Logical I/O on large files is the structuring of files for the Virtual Access Method (VAM).

**VAM Files**

The records of a VAM file may extend over several pages of virtual memory. One page of a VAM record contains the directory to the specific items in the remainder of the record. This directory in the record will be called the Table of Contents (TOC). The records are, in general, variable length. In calling for a VAM record, the user does not specify the virtual address at which he would like the retrieved record to reside, instead, the access method (macro) obtains all the virtual address space that is needed, so and provides the user task with the virtual location of the record. The access method also provides the task with the location of the TOC for the record. The TOC will contain addresses which are relative to the beginning of the record. The user task can therefore locate the specific items of interest in the virtual memory that was allocated for the VAM record.

Thus, retrieving a VAM record for a user is equivalent to allocating virtual memory for an entire file and providing the user the file directory, yet not loading the entire allocated space. The user may then access selected "records" within the "file", causing only the needed portions of the VAM record transferred into main memory.

The efficiency of the virtual access method is obvious, because in addition to the Table of Contents, only the parts of a record that
are needed are brought into main memory.

The operation of this access method parallels the paging technique. When a page of virtual memory has not been allocated in core, the page table entry for that page is marked "not in core". The important address in this page table entry is then the address of the page on the paging device. This physical address is used by the Paging Routine to initiate a physical I/O operation to the paging device. The same technique is now extended to include pages residing on external devices. When virtual memory is allocated for these pages, page table entries are assigned to the pages; but these entries are marked "not in virtual memory" and the address associated with the entries are their addresses on the external storage device. When a page marked "not in core" is accessed, a page fault interrupt occurs, resulting in an I/O operation from the paging device. When a page marked "not in virtual memory" is accessed, a page fault interrupt will also occur. In this case the paging operation that results will be a read operation from the external storage device.

The operation of the Virtual Access Method will be made explicit in the following:

The variable length VAM records may belong to a file whose structure is similar to that of previously described files. No indication of record lengths are contained in the file directory. Since a VAM record's TOC may also be of variable length, it must be made to fit into a fixed length format which is standard for the entire file. A way of doing this is to define a TOC for the TOC - essentially con-
structing a TOC in the form of a multi-level directory. Henceforth, 'TOC' for a VAM record will refer to only the first, fixed length portion of the TOC. Starting from the TOC, then, the accessing of any particular item within the record may occur in a multi-step operation.

The remainder of the VAM record will be on the physical tracks following the TOC track, in the address space of the external random access device. Then, with the physical (external) address of the beginning of the VAM record known, and the logical address (with respect to the beginning of the record) of a data element known, the address of the track containing the data element will be known.\footnote{This restriction is included to simplify the operation of the VAM to allow use of the previously defined file structure and macros. What the VAM GET macro actually needs is a mapping from the logical addresses internal to a record, to the physical addresses of the tracks on which these addresses reside. Without this restriction, this mapping, for each record, must necessarily be contained in the file directory not made.}

Figure 4-16 is an outline of the steps of the VAM GET macro. First, the VAM GET macro calls on a system routine to allocate enough virtual memory to contain the TOC for a record of this file. Then, the TOC which contains the record length will be brought into virtual memory. The accessing of the VAM TOC may be performed by the regular GET routine previously described. With the virtual memory allocated for the TOC of record A, the 'GET F, A' operation will be assumed to bring the TOC of record A into virtual memory. This is operation (1) in Fig. 4-17.

Next, enough virtual memory is allocated to contain the entire record, and the page table entries for this virtual memory are con-
VAM GET macro.

Input: file name F
      record index X

allocate virtual memory for TOC

GET F, X

allocate virtual memory for entire record

construct page table entries for allocated pages; addresses are external storage addressed

Mark page table entries 'not in virtual memory'

exit

Figure 4-16: VAM GET macro
structured. These are 2 and 3 of Fig. 4-17. The page table entries will contain a 'not in virtual memory' flag. The external addresses of the pages are calculated from the initial external address of the record, and placed in the page table. This is operation 4 of Fig. 4-17.

**Economization of Secondary Storage**

When the active programs operate in a core memory squeezed manner, the expensive core memory is conserved by allocating to each program only the core memory that is absolutely necessary to its operation at any instant. This minimizes the time-core memory space product of the program operation. For the remainder of the program - the less active part - low-cost secondary storage is substituted for the higher-cost core memory.

At some point, even the secondary storage, which now contains the entire virtual memory of each program will become excessively large relative to its cost. The same technique used to conserve core memory can be applied again to conserve virtual memory. That is, a program will run in a virtual memory squeezed manner. Only the parts of the virtual memory that are used actively are allocated secondary storage. Of course, the operating definition of 'active' here must be considerably less stringent than the definition which allows a part of the program to reside in core. The remainder of the virtual memory is kept resident on some even less expensive mass memory devices. 

---

1 The logical extension of this is, of course, a hierarchy of memory devices. Each device in the hierarchy being bigger, slower and less expensive (per unit of information) than the devices at the level above. Programs and data will move up and down in the hierarchy, based on their utilizations over periods ranging from fractions of microseconds at the top, to years at the bottom.
Virtual Memory
(Secondary Memory)

Page Table

V.M. allocated for all tracks of VAM Record

New entries for VAM record

TOC track

Data track

Figure h-17

Operation of VAM CMP
been developed to optimize secondary storage utilization. The technique of writing user programs to be reentrant (pure procedure, in Multics terminology), for example, represents a saving of core memory only if one user requires a page of program at the same instant that another user is using that page (i.e., another user has allocated core memory for it). However, this technique results in a greater optimization of secondary storage, because only one copy of that program need be resident in secondary storage for all actual or potential users of the program.

Another economy of the time-space product of secondary storage is the **dynamic call**.

Assume a user task is executing one program, Program A, which requires the services of another program, Program B. Both of these programs must be loaded into the same virtual memory, so that they can communicate with each other. Traditionally, these programs have been loaded into the same virtual memory and linked together before the task is run, i.e., before the execution of program A has begun. Thus, the virtual memory of the task must be large enough to include both programs throughout the operation of the task.

A reduction of the time-virtual space product for the task is apparent if the program B is not loaded into the virtual memory of the task until the actual call on B takes place. The system CALL macro will perform the function of loading and linking program B, and transferring to it. The CALL macro therefore utilizes a loader. To understand the operation of the CALL macro, the loading problem will first be reviewed.
Using the above technique, the problem of resolving the internal addresses of a program when it is relocated in virtual memory can be alleviated.

The more difficult task in the resolution of address constants occurs when addresses that are used in one program, program A, are defined within another, program B. If A and B were compiled or assembled separately, the language processor of program A has no knowledge of the values of the symbols that are defined within B. Aside from the fact that these symbols are not defined within any virtual memory, their value even relative to program B is unknown. Therefore, the language processor of A leaves those externally defined symbols in the form of alphanumeric names. The values of these symbols must be determined at the time the program is loaded into virtual memory.

The data used by the loader for the above purpose will be specified more precisely.

The output of a language processor is an object module. It consists of text (i.e., the instructions and constants that comprise the program itself) and three classes of address-specifying information: relocation information, externally defined symbols and symbol definitions.

The first class of address information is the relocation information; it is a specification of locations within the program which contain addresses. The loader needs this in order to modify these locations when the program is loaded into virtual memory.
The second class is the set of external references called externally defined symbols (REFS). These are names used within program A which are defined within some other program. These alphanumeric names are placed in a REF table by the language processor. In general, there may be many points within A which contain a reference to one externally-defined symbol. Therefore, the text of the object module will contain a pointer to an entry in the REF table at each point where an external reference is made. Although the REF table entry may include pointers to each location within the program at which the address is required, this technique is usually not used. In either case, the loader must be capable of placing the proper definition of the external reference into the program. The link between the program word that requires the address and the name itself must be present.

The third class of address information is the set of entries called symbol definitions (DEFS). These are the symbolic locations within program A that may be referenced by some other program. The language processor creates a table of correspondence between the DEF's, which are represented by alphanumeric names, and their values, which are their addresses relative to program A. This table is called the DEF table.

Two object modules, containing explicit representations of the links in the REF and DEF tables are shown in Figure 4-18.

After the program has been loaded into a virtual memory, the relocation information (for internal addresses) may be discarded, since all of the internal references have been specified. Then, the REFS must be resolved. The loader cannot complete the loading of program A until
Figure h-18

Address specifying information contained within object modules.
the DEFS of program B which are REFS of program A are assigned virtual addresses. The loader allocates space for program B, and loads it into virtual memory before returning to complete the loading of program A. The REFS in program A are given their virtual addresses, and then the table of REFS is discarded. The DEF table of program A, however, must always remain available, for programs that are loaded later may contain a REF to a location within program A. Furthermore, the DEF table need not now contain relative addresses. Since each symbol in the DEF table now has a value which is a virtual address, the table is updated to contain the correspondence between DEF symbols and their virtual addresses.

Each DEF table of a loaded program is moved into a separate virtual storage area called the Task Dictionary (TDY). Each time a new program is loaded for a task, the TDY will be searched for each new REF. If every REF of the new program is found in the TDY, no recursive call on this loader will be necessary to load new programs to resolve REFS.

**Loader Operation**

With this background, the operation of the loader will be reviewed.

In order to load a program for a task into virtual memory, the loader must perform each of the following five steps, although each does not necessarily have to be completed before another is performed:

1. Allocate virtual memory for the program.
2. Move the text of the program to the virtual memory.
3. Update each relative address in the text of the program. The relocation information must be available during this step, and may be discarded after this step.
(4) Update the DEF table by completing the address of each symbol in this table. The DEF table is now attached to the TDY of the task which may be referenced by the loader during the loading of another program containing a REF to this program.

(5) Define each entry in the REF table. This involves:
(a) A search of the TDY for each REF symbol.
(b) A search of the user file catalog for the symbol if it is not found in the TDY.
(c) A search for the symbol in the Syscat if it is not found in the user file definition area.

In cases (b) and (c), the REF is to be found in a program that is not yet loaded. Thus, the REF cannot be defined until the corresponding program is loaded. Consequently, the program containing the symbol must be found and a recursive call on the loader must be made before this step can be completed. When the symbol is defined, each reference to it in the program is completed. Finally, when this is done, the REF table may be discarded.

Figure 4-19 shows how each of the REF's of program A is resolved as A is loaded. Before A is loaded, only the program 'OKIC' resides in the virtual memory of the task. (Fig. 4-19a) During loading, the REF 'SUBB' is not found in the TDY, which contains all the external symbols defined in previously loaded programs. (Fig. 4-19b) A search through the file catalog shows SUBB to be defined in program B. Therefore, program B is loaded, and its updated DEF table is added to the TDY,
a. before prog. A load.

b. using Program A load
   ref. SUBB is undefined

c. after program B is loaded.

Figure 4-19 Resolution of External Symbols
before the address within A can be completely specified. (Fig. 4-19c)

**Loader Operation with the Virtual Access Method**

The preceding outline of a loader shows that the loader is a heavy user of the I/O operations. To achieve whatever economies of machine utilization are possible during these I/O operations, the loader will use the virtual access method (VAM). When the virtual access method is employed, the data is physically moved only as the data is used, one page at a time. In this case, a page of a program will be brought into virtual memory only when it is accessed in execution of the program; the program is forced into execution before it is loaded. The loader, however, must resolve the program's addresses after the text is brought in from the external device. This implies that the loader operation must be distributed; it will operate on one page of a program at a time, as these pages are brought in from external storage.

If the loader is called to load one page at a time, it must have all the address-specifying information available in virtual memory until each page of the program is loaded. The address-specifying information for each page must be available in physical memory as the page is loaded.

Two separate load operations are seen. One is the preparation for the loading of a program; this is called once for each program to be loaded. The text of the program is not moved during this operation. This will be called the **Program Load operation**.

The second load operation is the **Page Load operation**. It will be performed as each page of the program to be loaded is accessed for the first time. The obvious way to implement this is to call the **Page Load**
operation in response to a page fault.

The Program Load macro, which uses the VAM, is outlined in Fig. 4-20. Using the name of the program as an argument, the load operation calls the VAM GET. This macro returns the virtual address allocated to the program, the length of the program, and the virtual address of the Table of Contents. The TOC, in the case of an object module record, contains the addresses of the object module relocation-specifying information (including DEF table and REF table). The load operation then moves the DEF and REF tables to the TDY, so that they may be referenced later during the page load operation. As soon as this information is accessed, a page fault occurs and an implicit call on the VAM paging routine occurs. This brings the page containing the program relocation-specifying information into core memory.

Since the VAM call provided the virtual address at which the program is to be loaded, the loader can make the correspondence between each relative address within the program and its virtual address. In particular, the DEF table for the program can be completed. The loader does this. The REF table may be completed when the REFS are needed, at the time of the call for Page Load operation.

Lastly, since the page faults that result in calls on the VAM may not be programs which must be loaded, as well as data, the VAM Page Load operation must be informed whether the page contains data or program. This information is left in the page table. Along with the 'not in core' and the 'not in virtual memory' bits, there will be an
Figure 4-20

Program LOAD operation
Figure 4-21
Physical Events in the LOAD Operation
'unprocessed by loader' bit for each page that contains program sections.\footnote{Not every page in the text of a program will require loading operations, because programs can be written in part or whole without containing any address constants. For example, reentrant code contains no addresses of modifiable data.}

The Page Loader

When a page fault occurs during the operation of a program, the Page-fault-interrupt-response Routine checks the 'not in WM' bit of page table. If it is set, the paging routine initiates a physical I/O operation for the page. When the physical read I/O operation is completed, the paging operation checks the 'unprocessed by loader' bit for the page. If it is set, a loading process must begin. The system program which handles the loading of program pages is called the Page Loader.

The page loader finds the relocation information corresponding to this page in the TDY. The address computation for each specified word in the relocation information table is performed. If any address requires the definition of a REF, the TDY is searched for the REF. If this search is not successful, first the user catalog, and then the system catalog are searched for the symbol definition. If it is found, the program containing the symbol must be loaded in order that the REF may be resolved.

The operation of the page loader is outlined in Figure 4-22.

Multi-tasking

Sometimes when one task is required to perform several logical functions it is useful for the task to create another task to perform one or more of these logical functions. This will result in a reduction in the total time required for the original task, since some functions...
The page loader

Figure 4-22
that it requires will be performed in parallel with it. The original
task and the created task may run in parallel even on a single-processor
system, by competing for system resources. One consequence is, for
example, that one task may take the processor while the other is waited
for an I/O operation.

A task creates another by calling on the 'Create' macro. The input
parameters to this macro are all the data that is needed to define a new
task. Most importantly, a virtual address representing the starting
point for the new task is required. Other data, such as the page table
for the new task, file access and security data, may be copied directly
from the TCB of the originating task. In fact, the created task may
have only a dummy TCB, containing little more than a pointer to the
originating task's TCB.

The Create macro may operate in one of several ways, depending
upon how its starting location is specified. This may be specified by
one of the following three ways: (1) as a literal name of a program,
(2) as the SVC number of a system program, and (3) as the virtual
address of a program. In the first case, the Create macro searches the
Syscat for the name of the program, locates it, and transfers to it.
In each case the Create macro performs the transfer by constructing a
small program to receive the return of the routine, then inserting
either the SVC or the transfer instruction in that program, and then
transferring to the created program.
Figure 4-23 is a representation of the operation of the Create macro. The three general methods of specification of the starting point for the created task are deciphered by this macro and the program representing the beginning of the created task is shown. The operations of the created task occur in a subroutine in this representation and the program for the created task consists of a transfer instruction to the subroutine and a return point.

After it is run, the created task must be removed from the system by issuing a 'Destroy' macro. The Destroy macro works in an analogous way to the Create macro. It will not be detailed here.

If one task spawns another task by means of the Create macro, the two tasks must again communicate with each other; either to inform each other of their progress in the execution of their particular function, or else to indicate completion of their logical functions. In order to achieve communication and synchronization, two more macros are necessary. The created task will use the 'Post' macro which simply places the information that a particular part of its operation is completed into the TCB of the creating task. The creating task will issue a 'Task Wait' operation, which indicates that it will cease processing until the 'posted' indication is made.

Use of the Task Wait and Post macros is not restricted to just created tasks; they can be used to synchronize any two tasks. The input parameters to the Post macro are the task number or other identification of the task to be communicated with, and an identification of the posted information. The input to the Task Wait macro is an identi-
CREATE: input is starting location, address, SVC number, or name

- Set up TCB for created task -

- Initialize new task: set up return -

- V.M. address ?
  - N -
    - SVC number ?
      - N -
        - starting location is named
      - Y -
        - Put 'NAME' in instruction 'LD'
        - Put 'CALL NAME' in instruction INS
  - Y -
    - Put transfer instructions in INS
    - Put SVC number in INS

- Put address INS in TCB

- Task Initiator

Figure 4-23 - The CREATE Macro
ification of the event to be awaited. The Task Wait and Post macros are shown in Figure 4-24.

Higher Level Macros

Macros that have been described are compounded to form higher-level macros. Usually a set of higher-level macros is provided by the system to be immediately available to the user at the terminal. The names of these macros and their associated parameters make up the command language of the system. The user at the terminal will also have the capability of conversing with any of his own programs in his own language or the language of a shared language processor. To eliminate the ambiguity associated with the simultaneous use of several languages, the user will prefix each command to the system with some special character. When the interrupt marking the completion of an input from the terminal occurs, the system can decide whether the input was a command to the system. If it was, the Terminal-I/O-complete-interrupt-response Macro will initiate the routine servicing this command.

A typical command that a user at a terminal would call is the 'Execute' command, directing the system to run a specified program. The parameters of the 'Execute' command are the file name of the file containing the program, the name or the entry point of the program, and a specification of where the data for the program is to be found.

Assume, for example, that the data parameters of the 'Execute' command specify program P in file F with data D in file G. The command may be written: Execute F,P,G,D.
a) Task Wait Macro

b) 'Post' Macro

Figure 4-24
Task Wait and Post Macros
It will have the effect of performing the following macros:

OPEN F
LOAD F,P
OPEN G
GET D
transfer to P

In fact, the 'Execute' macro itself will be coded in a very similar manner.

The implication of this type of coding is that each macro will be performed sequentially. Some possible saving of time is apparent if the functions of loading the program and obtaining the data are performed in parallel. Thus, the 'Execute' macro could be written as follows:

CREATE task t to GET G,D
OPEN F
LOAD F,P
WAIT for t to obtain data
transfer to P

It will be even more efficient for program P itself to issue the 'Task Wait' SVC at the instant the data is needed. If program P has been coded without knowledge of any assumptions regarding where this data is to be obtained, it will contain a simple GET call. The GET macro must be made aware that the data could have been called for in a separate operation. This will require a more complex GET macro. Here, it may be assumed that program P contains a different type of GET macro; one that essentially contains the 'Task Wait' macro. Thus, the 'Execute' macro is as follows:
CREATE task t to GET G,D
OPEN F
LOAD F,P
transfer to P
(Program P will 'Task Wait' for t.)

The above expression of the 'Execute' macro can be used for generating a task trace.

Summary

The discussion in this section has been an outline of some of the basic operations of a paged multiprogramming system, with emphasis on its resource allocation functions. Some indication of the implementation of the system has been presented by specific examples. Of course, not all of the logical functions of the system have been shown; the complete system will be composed of more macros similar to those given here, and higher level functions calling on those macros.

The system could have been presented by emphasizing other points of view. For example, as an interface between the tasks' logical requirements and the hardware, the operating system may be considered as consisting of two parts. One part is the task monitor, which received the tasks' demand. The other part is the supervisor, which allocates the hardware resources. With this differentiation, the implementation of the system functions becomes easier, because it enables requests for system resources by system programs, to be addressed to the task monitor. The request to the task monitor is a logical operation; therefore the program need not be concerned with scheduling the hardware operations that ensue.
Still, many parts of the design of an operating system have not been examined. For example, in this discussion, virtual memory was simply assumed to be a very large address space, and each task's active part of virtual memory was a set of pages of the address space. In order to easily manipulate these sets of pages, and define them to be either private or shared, the virtual memory may be segmented. The policies used for the segmentation and the manipulation of the segments are integrally related to the hardware which decodes addresses; several different versions of this hardware are in common use. The management of virtual memory involves essentially the referencing and setting of values in tables, as opposed to the data-transfer operations associated with management of main memory. Virtual memory management may be simplified by segmentation. However, the main memory management problems, where the concern of resource allocation lies, is not greatly alleviated by segmentation. Segmentation schemes have therefore been overlooked in this description.

Security, privacy, reliability, backup and fail-safe are some of the other capabilities that must be integrated into this design. Each of these presents a large design problem in itself.
CHAPTER 5

EXTRACTING THE EVENT TRACES

User resource demand can be measured at several levels of system operation, depending upon the definition of resource that is applied. In the Meta-system design, the definition of resource reflects the level of the Meta-system's awareness of system operation: if a system program or function is considered a resource the events representing the demand for the resource - the calls for the program or function - are extracted from the system for the purpose of obtaining the input to a simulator. That program or function will be in the simulation model, and it may then be investigated to determine if some modification to it will result in more efficient overall operation.

This chapter discusses the process of obtaining event traces from the system that has been described. Four different types of event traces are demonstrated, corresponding to four levels of system measurement -- these are:

- The hardware event trace
- The physical event trace
- The logical event trace
- The relocation event trace

Each main section of the chapter describes one of these event traces. Each description contains:
1) The definition of the event trace.

2) The method of extracting the trace.

3) An example of the event trace. The examples of the different event traces will be taken from a single instance of system operation, so that the various traces may be compared.

4) A description of the processor of the event trace that decomposes it into several task event traces.

5) An example of the task event trace. The sample task trace will be taken from the example of the system event trace.

The examples of the event traces included in this chapter are taken from a period of system operation in which five tasks are active, and then each becomes inactive. The tasks are scheduled according to a priority discipline; the effects of the scheduler will be seen as tasks one and two gain the processor and paging device ahead of the other tasks on queue for these resources. The recording of the sample event traces may be observed by tracing through the system routine flowcharts in Appendix A.

The specific problems of extracting and processing these event traces will be treated in sections in which the problems are encountered.
Table 5-1
Example of System Hardware Event Trace (SHET)

<table>
<thead>
<tr>
<th>Time</th>
<th>Task</th>
<th>Event</th>
<th>Time</th>
<th>Task</th>
<th>Event</th>
</tr>
</thead>
<tbody>
<tr>
<td>3</td>
<td>5</td>
<td>I/O-beg-seek</td>
<td>99</td>
<td>5</td>
<td>I/O-beg-seek</td>
</tr>
<tr>
<td>3</td>
<td>1</td>
<td>on</td>
<td>3</td>
<td>3</td>
<td>on</td>
</tr>
<tr>
<td>4</td>
<td>-</td>
<td>idle</td>
<td>105</td>
<td>-</td>
<td>idle</td>
</tr>
<tr>
<td>5</td>
<td>3</td>
<td>pg-end</td>
<td>110</td>
<td>1</td>
<td>pg-end</td>
</tr>
<tr>
<td>1</td>
<td>pg-beg-r</td>
<td></td>
<td>3</td>
<td>3</td>
<td>pg-beg-r</td>
</tr>
<tr>
<td>3</td>
<td>on</td>
<td></td>
<td>1</td>
<td>on</td>
<td></td>
</tr>
<tr>
<td>8</td>
<td>-</td>
<td>idle</td>
<td>114</td>
<td>2</td>
<td>on</td>
</tr>
<tr>
<td>20</td>
<td>1</td>
<td>pg-end</td>
<td>115</td>
<td>-</td>
<td>idle</td>
</tr>
<tr>
<td>4</td>
<td>pg-beg-r</td>
<td></td>
<td>122</td>
<td>4</td>
<td>I/O-end-seek</td>
</tr>
<tr>
<td>1</td>
<td>on</td>
<td></td>
<td>1</td>
<td>I/O-beg-seek</td>
<td>I/O-beg-seek</td>
</tr>
<tr>
<td>21</td>
<td>-</td>
<td>idle</td>
<td>114</td>
<td>2</td>
<td>I/O-beg-crch</td>
</tr>
<tr>
<td>28</td>
<td>5</td>
<td>I/O-end-seek</td>
<td>125</td>
<td>3</td>
<td>pg-end</td>
</tr>
<tr>
<td>5</td>
<td>I/O-beg-crch</td>
<td></td>
<td>2</td>
<td>pg-beg-r</td>
<td></td>
</tr>
<tr>
<td>35</td>
<td>4</td>
<td>pg-end</td>
<td>140</td>
<td>2</td>
<td>pg-end</td>
</tr>
<tr>
<td>1</td>
<td>pg-beg-r</td>
<td></td>
<td>2</td>
<td>on</td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>on</td>
<td></td>
<td>141</td>
<td>2</td>
<td>pg-beg-r</td>
</tr>
<tr>
<td>37</td>
<td>4</td>
<td>I/O-beg-seek</td>
<td>144</td>
<td>4</td>
<td>I/O-end-crch</td>
</tr>
<tr>
<td>4</td>
<td>idle</td>
<td></td>
<td>4</td>
<td>on</td>
<td></td>
</tr>
<tr>
<td>43</td>
<td>5</td>
<td>I/O-end-crch</td>
<td>146</td>
<td>-</td>
<td>idle</td>
</tr>
<tr>
<td>5</td>
<td>on</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>44</td>
<td>5</td>
<td>I/O-beg-seek</td>
<td>149</td>
<td>5</td>
<td>I/O-end-crch</td>
</tr>
<tr>
<td>5</td>
<td>idle</td>
<td></td>
<td>5</td>
<td>I/O-beg-crch</td>
<td>I/O-beg-crch</td>
</tr>
<tr>
<td>48</td>
<td>4</td>
<td>I/O-end-seek</td>
<td>167</td>
<td>1</td>
<td>I/O-end-seek</td>
</tr>
<tr>
<td>4</td>
<td>I/O-beg-crch</td>
<td></td>
<td>1</td>
<td>I/O-beg-crch</td>
<td>I/O-beg-crch</td>
</tr>
<tr>
<td>50</td>
<td>1</td>
<td>pg-end</td>
<td>170</td>
<td>2</td>
<td>pg-end</td>
</tr>
<tr>
<td>3</td>
<td>pg-beg-r</td>
<td></td>
<td>4</td>
<td>pg-beg-r</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>on</td>
<td></td>
<td>2</td>
<td>on</td>
<td></td>
</tr>
<tr>
<td>51</td>
<td>-</td>
<td>idle</td>
<td>174</td>
<td>-</td>
<td>idle</td>
</tr>
<tr>
<td>65</td>
<td>3</td>
<td>pg-end</td>
<td>182</td>
<td>1</td>
<td>I/O-end-crch</td>
</tr>
<tr>
<td>1</td>
<td>pg-beg-r</td>
<td></td>
<td>1</td>
<td>on</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>on</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>68</td>
<td>5</td>
<td>I/O-end-seek</td>
<td>184</td>
<td>1</td>
<td>I/O-beg-seek</td>
</tr>
<tr>
<td>5</td>
<td>I/O-beg-crch</td>
<td></td>
<td>-</td>
<td>idle</td>
<td></td>
</tr>
<tr>
<td>70</td>
<td>4</td>
<td>I/O-end-crch</td>
<td>185</td>
<td>4</td>
<td>pg-end</td>
</tr>
<tr>
<td>4</td>
<td>on</td>
<td></td>
<td>2</td>
<td>pg-beg-r</td>
<td></td>
</tr>
<tr>
<td>78</td>
<td>4</td>
<td>on</td>
<td>4</td>
<td>on</td>
<td></td>
</tr>
<tr>
<td>80</td>
<td>1</td>
<td>pg-end</td>
<td>187</td>
<td>-</td>
<td>idle</td>
</tr>
<tr>
<td>3</td>
<td>pg-beg-r</td>
<td></td>
<td>5</td>
<td>end-I/O-crch</td>
<td>end-I/O-crch</td>
</tr>
<tr>
<td>1</td>
<td>on</td>
<td></td>
<td>5</td>
<td>on</td>
<td></td>
</tr>
<tr>
<td>81</td>
<td>4</td>
<td>on</td>
<td>189</td>
<td>5</td>
<td>I/O-end-seek</td>
</tr>
<tr>
<td>4</td>
<td>I/O-end-seek</td>
<td></td>
<td>-</td>
<td>idle</td>
<td></td>
</tr>
<tr>
<td>82</td>
<td>4</td>
<td>idle</td>
<td>5</td>
<td>I/O-end-seek</td>
<td>I/O-end-seek</td>
</tr>
<tr>
<td>95</td>
<td>3</td>
<td>pg-end</td>
<td>187</td>
<td>1</td>
<td>I/O-beg-crch</td>
</tr>
<tr>
<td>1</td>
<td>pg-beg-r</td>
<td></td>
<td>1</td>
<td>I/O-end-seek</td>
<td>I/O-end-seek</td>
</tr>
<tr>
<td>3</td>
<td>on</td>
<td></td>
<td>3</td>
<td>I/O-beg-crch</td>
<td>I/O-beg-crch</td>
</tr>
<tr>
<td>98</td>
<td>5</td>
<td>I/O-end-crch</td>
<td>200</td>
<td>2</td>
<td>pg-end</td>
</tr>
<tr>
<td>5</td>
<td>on</td>
<td></td>
<td>2</td>
<td>on</td>
<td></td>
</tr>
<tr>
<td>Time</td>
<td>Task</td>
<td>Event</td>
<td>Time</td>
<td>Task</td>
<td>Event</td>
</tr>
<tr>
<td>------</td>
<td>------</td>
<td>------------</td>
<td>------</td>
<td>------</td>
<td>------------</td>
</tr>
<tr>
<td>204</td>
<td>5</td>
<td>I/O-end-arch</td>
<td>327</td>
<td>2</td>
<td>pg-beg-r</td>
</tr>
<tr>
<td>212</td>
<td>5</td>
<td>pg-beg-r</td>
<td>330</td>
<td>4</td>
<td>I/O-end-arch</td>
</tr>
<tr>
<td>215</td>
<td>2</td>
<td>on</td>
<td>337</td>
<td>-</td>
<td>idle</td>
</tr>
<tr>
<td>222</td>
<td>1</td>
<td>I/O-end-arch</td>
<td>339</td>
<td>1</td>
<td>I/O-end-arch</td>
</tr>
<tr>
<td>224</td>
<td>1</td>
<td>I/O-beg-arch</td>
<td>350</td>
<td>2</td>
<td>pg-end</td>
</tr>
<tr>
<td>229</td>
<td>2</td>
<td>on</td>
<td>353</td>
<td>4</td>
<td>pg-beg</td>
</tr>
<tr>
<td>230</td>
<td>1</td>
<td>I/O-end-arch</td>
<td>360</td>
<td>-</td>
<td>idle</td>
</tr>
<tr>
<td>232</td>
<td>2</td>
<td>on</td>
<td>361</td>
<td>1</td>
<td>I/O-end-arch</td>
</tr>
<tr>
<td>245</td>
<td>4</td>
<td>pg-end</td>
<td>365</td>
<td>4</td>
<td>on</td>
</tr>
<tr>
<td>249</td>
<td>5</td>
<td>pg-beg</td>
<td>380</td>
<td>1</td>
<td>pg-end</td>
</tr>
<tr>
<td>254</td>
<td>4</td>
<td>I/O-beg-arch</td>
<td>383</td>
<td>2</td>
<td>pg-beg</td>
</tr>
<tr>
<td>258</td>
<td>1</td>
<td>on</td>
<td>393</td>
<td>4</td>
<td>I/O-end-arch</td>
</tr>
<tr>
<td>260</td>
<td>5</td>
<td>I/O-end-arch</td>
<td>398</td>
<td>2</td>
<td>pg-end</td>
</tr>
<tr>
<td>264</td>
<td>5</td>
<td>pg-beg-r</td>
<td>410</td>
<td>-</td>
<td>idle</td>
</tr>
<tr>
<td>265</td>
<td>5</td>
<td>on</td>
<td>420</td>
<td>2</td>
<td>pg-beg-w</td>
</tr>
<tr>
<td>267</td>
<td>4</td>
<td>I/O-end-arch</td>
<td>425</td>
<td>1</td>
<td>I/O-end-arch</td>
</tr>
<tr>
<td>269</td>
<td>4</td>
<td>I/O-end-arch</td>
<td>440</td>
<td>2</td>
<td>pg-end</td>
</tr>
<tr>
<td>284</td>
<td>2</td>
<td>I/O-end-arch</td>
<td>446</td>
<td>1</td>
<td>I/O-end-arch</td>
</tr>
<tr>
<td>285</td>
<td>1</td>
<td>on</td>
<td>450</td>
<td>4</td>
<td>I/O-end-arch</td>
</tr>
<tr>
<td>290</td>
<td>5</td>
<td>I/O-end-arch</td>
<td>470</td>
<td>4</td>
<td>idle</td>
</tr>
<tr>
<td>293</td>
<td>5</td>
<td>on</td>
<td>471</td>
<td>-</td>
<td>idle</td>
</tr>
<tr>
<td>295</td>
<td>2</td>
<td>I/O-end-arch</td>
<td>480</td>
<td>1</td>
<td>I/O-end-arch</td>
</tr>
<tr>
<td>300</td>
<td>1</td>
<td>I/O-end-arch</td>
<td>500</td>
<td>1</td>
<td>I/O-end-arch</td>
</tr>
<tr>
<td>305</td>
<td>2</td>
<td>on</td>
<td>502</td>
<td>1</td>
<td>I/O-end-arch</td>
</tr>
<tr>
<td>315</td>
<td>1</td>
<td>I/O-end-arch</td>
<td>510</td>
<td>1</td>
<td>idle</td>
</tr>
<tr>
<td>317</td>
<td>1</td>
<td>on</td>
<td>552</td>
<td>1</td>
<td>idle</td>
</tr>
</tbody>
</table>
Shortly after being placed on the processor, at time 4, task 1 references a page which is not in core - a page fault interrupt occurs. The Interrupt Analyzer (Fig. A-1) transfers to the Page Fault Interrupt Response Routine (Fig. A-2). This routine initiates paging operations when the paging device is available, however, in this case, the paging device is busy. Therefore, it enqueues the paging operation on the paging device queue, and transfers to the Task Initiator, which selects a new task to use the processor (Fig. A-3).

The Task Initiator finds no task ready to take the processor, therefore, it places an 'idle' event in the SNET and proceeds to idle. No task is associated with the idle event. It is entered as follows:

<table>
<thead>
<tr>
<th>Time</th>
<th>Task no.</th>
<th>Event</th>
</tr>
</thead>
<tbody>
<tr>
<td>4</td>
<td>--</td>
<td>idle</td>
</tr>
</tbody>
</table>

The paging device, which was being used by task 3 completes operation at time 5, causing an interrupt. The paging channel complete interrupt response routine is called (Fig. A-3). First, the page-end event is placed in the SNET.

This event is the following:

<table>
<thead>
<tr>
<th>Time</th>
<th>Task no.</th>
<th>Event</th>
</tr>
</thead>
<tbody>
<tr>
<td>5</td>
<td>3</td>
<td>pg-end</td>
</tr>
</tbody>
</table>

Then, since a paging operation was on queue for the paging device, the paging device is activated again, and the event of page-begin is recorded on the SNET. Thus the next event on the SNET is:

<table>
<thead>
<tr>
<th>Time</th>
<th>Task no.</th>
<th>Event</th>
</tr>
</thead>
<tbody>
<tr>
<td>5</td>
<td>1</td>
<td>pg-beg-r</td>
</tr>
</tbody>
</table>
The paging-channel routine then transfers to the Task Initiator, which finds that task 3, having completed a page-in, is now available for the processor. The 'on' event is recorded before task 3 begins processing.

<table>
<thead>
<tr>
<th>Time</th>
<th>Task no.</th>
<th>Event</th>
</tr>
</thead>
<tbody>
<tr>
<td>5</td>
<td>3</td>
<td>on</td>
</tr>
</tbody>
</table>

**Processor of the SHET**

It is obvious that the utilization of each of the hardware sections of the computer system can be calculated from the SHET. The technique is as follows. The SHET is scanned and the time intervals between the 'on' or 'begin' event for each device and the 'off', 'idle' or 'end' events are summed. However, the SHET does not contain the information from which the amount of processor time spent in "overhead" can be determined, and how much of the time is for processing directly on the requirements of the task. An event trace which does measure the amount of processing time spent in overhead is conceptually possible, by recording events marking every entry and exit to each system routine. The time spent in system routines defined as "overhead" is the overhead contribution of the system. The event trace that records overhead in this way is an extension of the event trace that has been defined, but the number of events that would be recorded in this measurement is an order of magnitude greater than the number of events showing hardware utilization.

Furthermore, the measurement of overhead is not central to the design of the meta-system. The purpose of extracting event traces is to design an overall system that will aid in increasing throughput of the computer system and increases or decreases in overhead are not strongly correlated with increased throughput. It is the combination of overhead and idleness
that affects throughput, not either of these quantities alone. Also, the overhead operations are seen to be a necessity. One improvement in system design that may result from use of the Meta-system is a decrease in overhead processing time, but no significant decreases are anticipated. Since the overhead itself is a smaller factor than either task processing time or idleness, the total effect of a reduction in overhead time will not result in a significant change in throughput.

In this system, a compromise is adopted. The overhead time is considered partly as processing -- when a particular task is assumed to be on the processor, and partly as idleness -- when no one task can be identified to be on the processor.

The processor that is necessary to measure hardware utilization therefore, being rather straightforward and somewhat irrelevant, will not be described.

A processor of the SHET, which breaks up the SHET into the various Task Hardware Event Traces (THETs) will be developed. Later, modified versions of this processor will be used to decompose higher level event traces. The task event traces obtained in those cases will be used as inputs to the simulator, in those cases the processor of the system event trace will be a preprocessor of the simulator input. Therefore, the processor of the SHET will also be called a Preprocessor, even though no utility is seen for the output of the SHET Preprocessor as an input to the simulator.
The input to the preprocessor is the System Hardware Event Trace, expressed as a sequence of ordered triplets \((\text{time}_i, \text{task}_i, \text{event}_i)\), where \(i = 1,2,3,...\)

The output of the preprocessor is a set of \(n\) TGETs, for the \(n\) jobs on the system during the time the SHET was recorded. Each TGET is a sequence of (time, event) ordered pairs. The \(j^{th}\) event in the \(i^{th}\) task trace is the event \((t_{ij}, e_{ij})\).

The algorithm of the preprocessor is approximated by the following rules:

1. The system event trace is scanned. If the current event in the scan is labeled task \(k\), the event is placed on the event trace for task \(k\).

2. If the current event of the scan is 'on' or 'idle', the trace for the task which had previously been on the processor is located, and an 'off' event - with the same time as the current event in the scan - is placed in that task trace.

The task trace obtained by these steps may be plotted along a time scale. An example of these steps, performed on the sample SHET of Table 5-1 is presented in Table 5-2. The events marking the 'begin' and 'end' of a device utilization period are connected by a solid line. Events occurring at approximately the same time are written on one line.

This 'extracted' event trace is not a pure representation of the tasks resource demand. The gaps in the continuum of utilization of some part of the system by a task are due to scheduling of the tasks' requirements by the system; they are not an effect of task demand. They are removed by the preprocessor, by redefining the time at which the events occur. The \textit{task time} of each event in a task trace is the time of the event relative to other events of the task. Intervals of task time
Table 5-2

Extracted Events of Task 1

<table>
<thead>
<tr>
<th>Time</th>
<th>Events</th>
<th>Time</th>
<th>Events</th>
</tr>
</thead>
<tbody>
<tr>
<td>3</td>
<td>on</td>
<td>333</td>
<td>I/O beg-seek, off</td>
</tr>
<tr>
<td>4</td>
<td>off</td>
<td>420</td>
<td>I/O end-seek</td>
</tr>
<tr>
<td>5</td>
<td>pg-beg</td>
<td>440</td>
<td>I/O beg-seek, off</td>
</tr>
<tr>
<td>20</td>
<td>pg-end, on</td>
<td>446</td>
<td>I/O end-seek</td>
</tr>
<tr>
<td>21</td>
<td>off</td>
<td>480</td>
<td>I/O beg-src</td>
</tr>
<tr>
<td>35</td>
<td>pg-beg</td>
<td>500</td>
<td>I/O end-src, on</td>
</tr>
<tr>
<td>50</td>
<td>pg-end, on</td>
<td>502</td>
<td>I/O beg-seek, off</td>
</tr>
<tr>
<td>51</td>
<td>off</td>
<td>540</td>
<td>I/O end-seek, on</td>
</tr>
<tr>
<td>65</td>
<td>pg-beg</td>
<td>552</td>
<td>I/O beg-src</td>
</tr>
<tr>
<td>80</td>
<td>pg-end, on</td>
<td>582</td>
<td>I/O end-src, on</td>
</tr>
<tr>
<td>81</td>
<td>off</td>
<td></td>
<td>I/O beg-seek, off</td>
</tr>
<tr>
<td>95</td>
<td>pg-beg</td>
<td></td>
<td>I/O end-seek, on</td>
</tr>
<tr>
<td>110</td>
<td>pg-end, on</td>
<td></td>
<td>I/O beg-src</td>
</tr>
<tr>
<td>112</td>
<td>off</td>
<td></td>
<td>I/O end-src, on</td>
</tr>
<tr>
<td>122</td>
<td>I/O beg-seek</td>
<td></td>
<td>I/O beg-seek, off</td>
</tr>
<tr>
<td>168</td>
<td>I/O end-seek</td>
<td></td>
<td>I/O end-seek, on</td>
</tr>
<tr>
<td>182</td>
<td>I/O beg-src</td>
<td></td>
<td>I/O beg-seek, off</td>
</tr>
<tr>
<td>184</td>
<td>I/O end-src</td>
<td></td>
<td>I/O end-seek</td>
</tr>
<tr>
<td>200</td>
<td>I/O beg-rch</td>
<td></td>
<td>I/O beg-seek, off</td>
</tr>
<tr>
<td>222</td>
<td>I/O end-src</td>
<td></td>
<td>I/O end-seek</td>
</tr>
<tr>
<td>224</td>
<td>I/O beg-seek</td>
<td></td>
<td>I/O end-src, on</td>
</tr>
<tr>
<td>229</td>
<td>I/O end-seck</td>
<td></td>
<td>I/O beg-rch</td>
</tr>
<tr>
<td>250</td>
<td>I/O end-rch</td>
<td></td>
<td>I/O beg-seek, off</td>
</tr>
<tr>
<td>258</td>
<td>I/O beg-rch</td>
<td></td>
<td>I/O end-seek</td>
</tr>
<tr>
<td>265</td>
<td>I/O beg-rch</td>
<td></td>
<td>I/O beg-seek, off</td>
</tr>
<tr>
<td>284</td>
<td>I/O end-rch</td>
<td></td>
<td>I/O end-seck</td>
</tr>
<tr>
<td>285</td>
<td>I/O beg-rch</td>
<td></td>
<td>I/O end-rch, on</td>
</tr>
<tr>
<td>295</td>
<td>I/O end-rch</td>
<td></td>
<td>I/O beg-seek, off</td>
</tr>
<tr>
<td>315</td>
<td>I/O beg-rch</td>
<td></td>
<td>I/O end-rch, on</td>
</tr>
<tr>
<td>317</td>
<td>I/O beg-rch</td>
<td></td>
<td>I/O beg-seek, off</td>
</tr>
<tr>
<td>339</td>
<td>I/O beg-rch</td>
<td></td>
<td>I/O end-rch, on</td>
</tr>
<tr>
<td>360</td>
<td>I/O end-rch</td>
<td></td>
<td>I/O beg-rch</td>
</tr>
<tr>
<td>361</td>
<td>I/O end-rch</td>
<td></td>
<td>I/O end-rch, on</td>
</tr>
<tr>
<td>365</td>
<td>pg-beg</td>
<td></td>
<td>I/O beg-rch</td>
</tr>
<tr>
<td>380</td>
<td>pg-end, on</td>
<td></td>
<td>I/O end-rch, on</td>
</tr>
</tbody>
</table>
represent processing of the task. Each task trace begins at task time zero, and every interval of task time represents the task's use of the processor.

The I/O operations that take place simultaneously with processing cause processor degradation due to cycle-stealing. The processing times that are shown for each of the tasks do not really represent the task's processor demand. The effects of simultaneity can be filtered out by adjusting each segment of processing time to be the amount of time that that same processing would have taken if no I/O action were taking place. This adjustment to the task event traces must be done while all the I/O operation information is available, that is, when the SHET is being decomposed.

The complete algorithm for the SHET preprocessor therefore includes conversion from system (real) time to task time, and compensation for processing delays due to simultaneity. The complete algorithm is shown in Figure 5-1. The following is a description of the flowchart and the definitions of the parameters and variables.

The processing of each event on the SHET begins with the placement of the event descriptors into the following processor locations:

\[
\begin{align*}
t &: \text{real time of the current event} \\
n &: \text{task number of the event} \\
e &: \text{the event descriptor}
\end{align*}
\]

Other locations maintained by the processor are the following:

\[
\begin{align*}
p &: \text{the task number of the task on the processor (zero if the processor is idle)}
\end{align*}
\]
l : the real time of the last event
f : the memory interference factor

The processor also maintains several locations for each task. For the $i^{th}$ task, these are:

- $k_i$ : the number of events on task trace $i$. This is a pointer to the next event on that trace.
- $r_i$ : the real time of the last event on the task trace.
  (This is the real time that corresponds to the task time $t_{ik_i}$.)
- $a_i$ : the accumulated processor time since the last event on the task trace.

Three subroutines are defined to simplify the operation of the SHEET preprocessor.

$U(i)$ will update the accumulated processor time of task $i$, when task $i$ is on the processor. $U(i)$ performs the following operation:

$$a_i + \frac{t_i - t}{f} \rightarrow a_i$$

$UP(i)$ updates the task trace for task $i$ when an event is to be placed on the task trace. The task time of the new event is the old trace time plus the accumulated processor time. The effect of $UP(i)$ is the following:

$$t_{ik_i} = t_{i(k_i-1)} + a_i \rightarrow t_{ik_i}$$

$$0 \rightarrow a_i$$
Figure 5-1

Processor of the CPU
UT(i) updates the task trace when the task time of the new event is the task time of an event on the trace, plus a real time interval. It is used to place an event on the task trace, whose position is a fixed real time from another event. The effect of this subroutine is:

\[ t_1(k_1 - 1) + t - r_1 \rightarrow t_1k_1 \]

\[ 0 \rightarrow r_1 \]

The individual "on" and "off" events for each processor application period are not recorded on the task traces. Processor use is summed in location \( a_i \) (by \( U(p) \)) over the intervals marking the beginnings and ends of the I/O and paging device operation. This is due to the assumption that the gaps in processor utilization -- when no other events occur -- are due to the scheduling of the tasks by the system, and are not due to the tasks' requirements.

**Example of the THET**

The result of applying the SHET preprocessor to the SHET example (Table 5-1) is a series of THETs. The THET for task 1 is presented in Table 5-3. The times given in this event trace are the task times, adjusted for memory interference. Task 1 is seen to be heavily engaged in paging at first, as it builds up a working set of pages, and then it becomes heavily engaged in I/O activity.
Table 5-3

Task Hardware Event Trace (METH) For Task 1

<table>
<thead>
<tr>
<th>Time</th>
<th>Events</th>
<th>Time</th>
<th>Events</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>on</td>
<td>238</td>
<td>I/O src-h-end</td>
</tr>
<tr>
<td>1</td>
<td>off</td>
<td>238</td>
<td>on</td>
</tr>
<tr>
<td>1</td>
<td>pg-beg</td>
<td>240</td>
<td>I/O seek-beg</td>
</tr>
<tr>
<td>16</td>
<td>pg-end</td>
<td>240</td>
<td>off</td>
</tr>
<tr>
<td>16</td>
<td>on</td>
<td>264</td>
<td>I/O seek-end</td>
</tr>
<tr>
<td>17</td>
<td>off</td>
<td>264</td>
<td>I/O src-beg</td>
</tr>
<tr>
<td>17</td>
<td>pg-beg</td>
<td>285</td>
<td>I/O src-end</td>
</tr>
<tr>
<td>32</td>
<td>pg-end</td>
<td>285</td>
<td>on</td>
</tr>
<tr>
<td>32</td>
<td>on</td>
<td>286</td>
<td>pg-beg</td>
</tr>
<tr>
<td>33</td>
<td>off</td>
<td>301</td>
<td>pg-end</td>
</tr>
<tr>
<td>31</td>
<td>pg-beg</td>
<td>301</td>
<td>on</td>
</tr>
<tr>
<td>46</td>
<td>pg-end</td>
<td>304</td>
<td>I/O seek-beg</td>
</tr>
<tr>
<td>46</td>
<td>on</td>
<td>304</td>
<td>off</td>
</tr>
<tr>
<td>47</td>
<td>off</td>
<td>341</td>
<td>I/O seek-end</td>
</tr>
<tr>
<td>47</td>
<td>pg-beg</td>
<td>341</td>
<td>I/O src-beg</td>
</tr>
<tr>
<td>62</td>
<td>pg-end</td>
<td>361</td>
<td>I/O src-end</td>
</tr>
<tr>
<td>62</td>
<td>on</td>
<td>361</td>
<td>on</td>
</tr>
<tr>
<td>64</td>
<td>I/O seek-beg</td>
<td>367</td>
<td>I/O seek-beg</td>
</tr>
<tr>
<td>64</td>
<td>off</td>
<td>367</td>
<td>off</td>
</tr>
<tr>
<td>91</td>
<td>I/O seek-end</td>
<td>401</td>
<td>I/O seek-end</td>
</tr>
<tr>
<td>104</td>
<td>I/O src-beg</td>
<td>401</td>
<td>I/O src-beg</td>
</tr>
<tr>
<td>104</td>
<td>on</td>
<td>421</td>
<td>I/O src-end</td>
</tr>
<tr>
<td>106</td>
<td>I/O src-beg</td>
<td>421</td>
<td>on</td>
</tr>
<tr>
<td>106</td>
<td>off</td>
<td>423</td>
<td>I/O src-beg</td>
</tr>
<tr>
<td>122</td>
<td>I/O seek-end</td>
<td>461</td>
<td>I/O seek-end</td>
</tr>
<tr>
<td>122</td>
<td>I/O src-beg</td>
<td>461</td>
<td>I/O src-beg</td>
</tr>
<tr>
<td>144</td>
<td>I/O src-end</td>
<td>473</td>
<td>I/O src-end</td>
</tr>
<tr>
<td>144</td>
<td>on</td>
<td>473</td>
<td>on</td>
</tr>
<tr>
<td>146</td>
<td>I/O seek-beg</td>
<td>503</td>
<td>off</td>
</tr>
<tr>
<td>146</td>
<td>off</td>
<td></td>
<td></td>
</tr>
<tr>
<td>151</td>
<td>I/O seek-end</td>
<td></td>
<td></td>
</tr>
<tr>
<td>151</td>
<td>I/O src-beg</td>
<td></td>
<td></td>
</tr>
<tr>
<td>176</td>
<td>I/O src-end</td>
<td></td>
<td></td>
</tr>
<tr>
<td>176</td>
<td>on</td>
<td></td>
<td></td>
</tr>
<tr>
<td>180</td>
<td>I/O seek-beg</td>
<td></td>
<td></td>
</tr>
<tr>
<td>180</td>
<td>off</td>
<td></td>
<td></td>
</tr>
<tr>
<td>187</td>
<td>I/O seek-end</td>
<td></td>
<td></td>
</tr>
<tr>
<td>187</td>
<td>I/O src-beg</td>
<td></td>
<td></td>
</tr>
<tr>
<td>206</td>
<td>I/O src-end</td>
<td></td>
<td></td>
</tr>
<tr>
<td>206</td>
<td>on</td>
<td></td>
<td></td>
</tr>
<tr>
<td>207</td>
<td>I/O seek-beg</td>
<td></td>
<td></td>
</tr>
<tr>
<td>207</td>
<td>off</td>
<td></td>
<td></td>
</tr>
<tr>
<td>218</td>
<td>I/O seek-end</td>
<td></td>
<td></td>
</tr>
<tr>
<td>218</td>
<td>I/O src-beg</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
The Physical Event Trace

The next trace to be developed will make manifest the user tasks' demand for the various hardware resources. It differs from the SHET in that it contains the events of the tasks' requests for the various hardware devices, rather than the actual allocation of the devices. For this trace, the I/O operations are recorded at the level of the Physical I/O macros, rather than at the level of the hardware instructions, hence it will be called the System Physical Event Trace (SPET).

In determining which events represent demand for system resources, as opposed to the allocation of them, it is useful to distinguish two types of requests for resources:

1. Requests from which the length of the utilization period of the resource may be determined. Examples are: the paging device, and the I/O devices.

2. Requests in which the utilization period of the resource is not specified. The utilization period is determined only by observing the duration of the tasks use of the resource. Examples are: the processor, and the main memory.

The utilization period of a hardware device is often dependent upon the state of the device at the time it is requested. In determining the utilization period of a device from the request, then, it is implied that a model of the device is available. This is the case in the development of the SPET: later it will be seen that the usefulness of the SPET will be in a simulator of the system hardware.
The events of the SPET indicate the utilization period of the memory and processor resources, and the requests for the paging and I/O devices. The latter must be specified in such a way that the utilization periods of the devices may be determined with the aid of a model of the devices.

**Definition of the SPET**

The elements of the SPET are ordered quadruples \((t, n, e, q)\). \(t\) is the real time of the event, and \(n\) the task number. These two elements are the same as those of the SNET.

The event descriptor, \(e\), is a member of the set to be defined. The last element, \(q\), is the 'data' element. It is a supplement to the specification of the type of event. The range of values that \(q\) will take on will differ for each type of event.

The following is the set of event descriptions for the SPET:

For events representing memory demand:

'pf' - a page fault event

'pg-in' - a page-in event

For events representing I/O demand:

'I/O-req-r' - the event of a request for a read I/O operation

'I/O-req-w' - a request for a write operation

For events representing processor demand:

'wake' - an event representing the completion of a terminal response, for which a task has been waited. It marks the request for a processor for an interaction.

'I/O wait' - an event marking the suspension of processing until the completion of an I/O operation
'sleep' - the event of termination of a task, or wait for
terminal response
'task wait' - the event of suspension of processing until
another task posts data
'post' - the event of posting data for another task
'create' - the event of the creation of another task
'destroy' - the event of the termination of a task by a task
System events that are necessary to complete the specification of
the demand for the processor and memory:
'on' - the event of the allocation of the processor to a task,
as in the SHET.
'idle' - the event marking the beginning of a processor idle
period.
'pg-de' - the event marking the deallocation of a main memory
block for a task.
'pg-out' - an event marking the beginning of the operation
of writing a page back to the secondary storage.
The memory demand of tasks is not well specified since the system
contributes to the definition of memory demand as it allocates memory.
Tasks signal increased memory demand, but not decreased demand.
The processor demand is assumed not to decrease during a paging
operation, but it is assumed to decrease when it explicitly issues a
'wait' for the termination of an I/O operation, or wait for a task.
Therefore, 'I/O wait' and 'task wait' are specifications of decreased
processor demand, but a page fault is not.
The events of the SFET indicate the utilization period of the memory and processor resources, and the requests for the paging and I/O devices. The latter must be specified in such a way that the utilization periods of the devices may be determined with the aid of a model of the devices.

Definition of the SFET

The elements of the SFET are ordered quadruples \((t, n, e, q)\). \(t\) is the real time of the event, and \(n\) the task number. These two elements are the same as those of the SHET.

The event descriptor, \(e\), is a member of the set to be defined. The last element, \(q\), is the 'data' element. It is a supplement to the specification of the type of event. The range of values that \(q\) will take on will differ for each type of event.

The following is the set of event descriptions for the SFET:

For events representing memory demand:

'pf' - a page fault event
'pg-in' - a page-in event

For events representing I/O demand:

'I/O-req-r' - the event of a request for a read I/O operation
'I/O-req-w' - a request for a write operation

For events representing processor demand:

'wake' - an event representing the completion of a terminal response, for which a task has been waited. It marks the request for a processor for an interaction.
'I/O wait' - an event marking the suspension of processing until the completion of an I/O operation
'sleep' - the event of termination of a task, or wait for terminal response
'task wait' - the event of suspension of processing until another task posts data
'post' - the event of posting data for another task
'create' - the event of the creation of another task
'destroy' - the event of the termination of a task by a task

System events that are necessary to complete the specification of the demand for the processor and memory:

'on' - the event of the allocation of the processor to a task, as in the SHET.

'idle' - the event marking the beginning of a processor idle period.

'pg-de' - the event marking the deallocation of a main memory block for a task.

'pg-out' - an event marking the beginning of the operation of writing a page back to the secondary storage.

The memory demand of tasks is not well specified since the system contributes to the definition of memory demand as it allocates memory. Tasks signal increased memory demand, but not decreased demand.

The processor demand is assumed not to decrease during a page out operation, but it is assumed to decrease when it explicitly issues a 'wait' for the termination of an I/O operation, or wait for a task. Therefore, 'I/O wait' and 'task wait' are specifications of decreased processor demand, but a page fault is not.
The events of the SPET indicate the utilization period of the memory and processor resources, and the requests for the paging and I/O devices. The latter must be specified in such a way that the utilization periods of the devices may be determined with the aid of a model of the devices.

**Definition of the SPET**

The elements of the SPET are ordered quadruples \((t, n, e, q)\). \(t\) is the real time of the event, and \(n\) the task number. These two elements are the same as those of the SNET.

The event descriptor, \(e\), is a member of the set to be defined. The last element, \(q\), is the 'data' element. It is a supplement to the specification of the type of event. The range of values that \(q\) will take on will differ for each type of event.

The following is the set of event descriptions for the SPET:

For events representing memory demand:

'pf' - a page fault event

'pg-in' - a page-in event

For events representing I/O demand:

'I/0-req-r' - the event of a request for a read I/O operation

'I/0-req-w' - a request for a write operation

For events representing processor demand:

'wake' - an event representing the completion of a terminal response, for which a task has been waited. It marks the request for a processor for an interaction.

'I/O wait' - an event marking the suspension of processing until the completion of an I/O operation
'sleep' - the event of termination of a task, or wait for
  terminal response
'task wait' - the event of suspension of processing until
  another task posts data
'post' - the event of posting data for another task
'create' - the event of the creation of another task
'destroy' - the event of the termination of a task by a task

System events that are necessary to complete the specification of
the demand for the processor and memory:

'on' - the event of the allocation of the processor to a task,
  as in the SHET.
'idle' - the event marking the beginning of a processor idle
  period.
'pg-de' - the event marking the deallocation of a main memory
  block for a task.
'pg-out' - an event marking the beginning of the operation
  of writing a page back to the secondary storage.

The memory demand of tasks is not well specified since the system
contributes to the definition of memory demand as it allocates memory.
Tasks signal increased memory demand, but not decreased demand.

The processor demand is assumed not to decrease during a paging
operation, but it is assumed to decrease when it explicitly issues a
'wait' for the termination of an I/O operation, or wait for a task.
Therefore, 'I/O wait' and 'task wait' are specifications of decreased
processor demand, but a page fault is not.
The events of the SPET indicate the utilization period of the memory and processor resources, and the requests for the paging and I/O devices. The latter must be specified in such a way that the utilization periods of the devices may be determined with the aid of a model of the devices.

**Definition of the SPET**

The elements of the SPET are ordered quadruples \((t, n, e, q)\). \(t\) is the real time of the event, and \(n\) the task number. These two elements are the same as those of the SHET.

The event descriptor, \(e\), is a member of the set to be defined. The last element, \(q\), is the 'data' element. It is a supplement to the specification of the type of event. The range of values that \(q\) will take on will differ for each type of event.

The following is the set of event descriptions for the SPET:

For events representing memory demand:

'pf' - a page fault event

'pg-in' - a page-in event

For events representing I/O demand:

'I/0-req-r' - the event of a request for a read I/O operation

'I/0-req-w' - a request for a write operation

For events representing processor demand:

'wake' - an event representing the completion of a terminal response, for which a task has been waited. It marks the request for a processor for an interaction.

'I/O wait' - an event marking the suspension of processing until the completion of an I/O operation
'sleep' - the event of termination of a task, or wait for
terminal response
'task wait' - the event of suspension of processing until
another task posts data
'post' - the event of posting data for another task
'create' - the event of the creation of another task
'destroy' - the event of the termination of a task by a task

System events that are necessary to complete the specification of
the demand for the processor and memory:

'on' - the event of the allocation of the processor to a task,
as in the SHET.
'idle' - the event marking the beginning of a processor idle
period.
'pg-de' - the event marking the deallocation of a main memory
block for a task.
'pg-out' - an event marking the beginning of the operation
of writing a page back to the secondary storage.

The memory demand of tasks is not well specified since the system
contributes to the definition of memory demand as it allocates memory.
Tasks signal increased memory demand, but not decreased demand.

The processor demand is assumed not to decrease during a paging
operation, but it is assumed to decrease when it explicitly issues a
'wait' for the termination of an I/O operation, or wait for a task.
Therefore, 'I/O wait' and 'task wait' are specifications of decreased
processor demand, but a page fault is not.
Each of these will have data associated with them. The significance of the \( q \) entries for each event are as follows:

For the 'pf', 'pg-in', 'pg-de' and 'pg-out' events, the \( q \) element is the virtual address of the page involved in the operation.

For 'I/O-req-r', I/O-req/w', and 'I/O wait' the \( q \) element is the identification of the devices, and track on the device to which the operation directed.

For the 'task wait' event the \( q \) element is the identification of the data that is being waited.

For the 'post' event the \( q \) element is the identification of the task for which the post is taking place, and the identification of the data being posted.

For 'create' and 'destroy' the task number of the task being created or destroyed.

For the 'wake' and 'sleep' events, the real time (time of day) at which the event occurs.

The 'on', and 'idle' events have a null \( q \) element.

**Extracting the SPET**

The events of the SPET are recorded in those locations in the operating system at which the operating system receives the requests for the resource. The recording locations in the system that has been developed are presented in Appendix A.

The 'pf', 'I/O-req', 'I/O wait', 'sleep', 'wake', 'create', 'destroy', 'task wait' and 'post' events all correspond to a task signaling -- via a macro -- a requirement for a particular resource.
They are recorded at the entrance to their respective macros - in Figures A-2, A-4, A-6a, A-6b, A-7, A-14b respectively. The other page demand events 'pg-in', 'pg-de', and 'pg-out', are recorded when their particular function occurs; as is seen in Figure A-2 for the latter two of these, and Figure A-3 for the first. The 'on' and 'idle' events are recorded in the Task Initiator - Figure A-3 - as in the hardware event trace.

**Example of the SPET**

The same case of system operation that produced the sample hardware event trace (Table 5-1) is used to demonstrate the SPET recording. The physical event trace taken from this system operation is shown in Table 5-1.

At the time the trace is initiated, (time 0) several tasks are in progress. Since these interactions began before the trace, the amount of resources allocated to these tasks will not be contained within the trace. (These fragmented interactions are not useful for a representation of task demand, but they are carried along because of the simplicity of the recording mechanism.)

At time 1 in the trace - 9:30:40 A.M. in real time, task 1 has a completion of the terminal. It is a response for which the task was waiting. The 'wake' event is recorded for the task (Figure A-7).

<table>
<thead>
<tr>
<th>Time</th>
<th>Task</th>
<th>Event</th>
<th>Data</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
<td>wake</td>
<td>09030410</td>
</tr>
</tbody>
</table>

The event does not indicate task 1 is given the processor; it indicates task 1 is ready for the processor. Task 5, which had been on the processor, continues and generates a request for a read operation.
## Table 5-4
### Example of System Physical Event Trace (SPET)

<table>
<thead>
<tr>
<th>Time</th>
<th>Task</th>
<th>Event</th>
<th>Data</th>
<th>Time</th>
<th>Task</th>
<th>Event</th>
<th>Data</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
<td>Wake</td>
<td></td>
<td>140</td>
<td>2</td>
<td>on</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>5</td>
<td>I/O req-read</td>
<td>51</td>
<td>141</td>
<td>2</td>
<td>pf</td>
<td>202</td>
</tr>
<tr>
<td>5</td>
<td>1</td>
<td>I/O Wait</td>
<td>51</td>
<td>144</td>
<td>4</td>
<td>on</td>
<td></td>
</tr>
<tr>
<td>7</td>
<td>1</td>
<td>idle</td>
<td></td>
<td>146</td>
<td>4</td>
<td>pf</td>
<td></td>
</tr>
<tr>
<td>8</td>
<td>3</td>
<td>idles</td>
<td></td>
<td>170</td>
<td>2</td>
<td>on</td>
<td></td>
</tr>
<tr>
<td>20</td>
<td>1</td>
<td>on</td>
<td></td>
<td>174</td>
<td>2</td>
<td>pf</td>
<td></td>
</tr>
<tr>
<td>21</td>
<td>1</td>
<td>pf</td>
<td>102</td>
<td>182</td>
<td>1</td>
<td>on</td>
<td></td>
</tr>
<tr>
<td>35</td>
<td>4</td>
<td>on</td>
<td></td>
<td>184</td>
<td>1</td>
<td>I/O req-r</td>
<td>12</td>
</tr>
<tr>
<td>37</td>
<td>4</td>
<td>I/O req-w</td>
<td>41</td>
<td>185</td>
<td>4</td>
<td>on</td>
<td></td>
</tr>
<tr>
<td></td>
<td>4</td>
<td>I/O Wait</td>
<td>41</td>
<td>187</td>
<td>4</td>
<td>pf</td>
<td></td>
</tr>
<tr>
<td>43</td>
<td>5</td>
<td>on</td>
<td></td>
<td>188</td>
<td>5</td>
<td>on</td>
<td></td>
</tr>
<tr>
<td>44</td>
<td>5</td>
<td>I/O req-r</td>
<td>52</td>
<td>189</td>
<td>5</td>
<td>I/O req-write</td>
<td>54</td>
</tr>
<tr>
<td>5</td>
<td>1</td>
<td>I/O Wait</td>
<td>52</td>
<td>190</td>
<td>5</td>
<td>on</td>
<td></td>
</tr>
<tr>
<td>51</td>
<td>1</td>
<td>pf</td>
<td></td>
<td>192</td>
<td>5</td>
<td>pf</td>
<td></td>
</tr>
<tr>
<td>65</td>
<td>3</td>
<td>on</td>
<td></td>
<td>209</td>
<td>5</td>
<td>on</td>
<td></td>
</tr>
<tr>
<td>78</td>
<td>3</td>
<td>pf</td>
<td>103</td>
<td>210</td>
<td>5</td>
<td>pf</td>
<td></td>
</tr>
<tr>
<td>80</td>
<td>1</td>
<td>on</td>
<td></td>
<td>215</td>
<td>2</td>
<td>on</td>
<td></td>
</tr>
<tr>
<td>81</td>
<td>1</td>
<td>pf</td>
<td></td>
<td>222</td>
<td>1</td>
<td>on</td>
<td></td>
</tr>
<tr>
<td></td>
<td>4</td>
<td>I/O req-r</td>
<td>42</td>
<td>224</td>
<td>1</td>
<td>I/O req-read</td>
<td>13</td>
</tr>
<tr>
<td></td>
<td>4</td>
<td>I/O Wait</td>
<td>42</td>
<td>230</td>
<td>5</td>
<td>on</td>
<td></td>
</tr>
<tr>
<td>82</td>
<td>4</td>
<td>idles</td>
<td></td>
<td>232</td>
<td>5</td>
<td>pf</td>
<td></td>
</tr>
<tr>
<td>95</td>
<td>3</td>
<td>on</td>
<td></td>
<td>249</td>
<td>2</td>
<td>I/O req-r</td>
<td>21</td>
</tr>
<tr>
<td>98</td>
<td>5</td>
<td>on</td>
<td></td>
<td>250</td>
<td>2</td>
<td>I/O Wait</td>
<td>21</td>
</tr>
<tr>
<td>99</td>
<td>5</td>
<td>I/O req-r</td>
<td>53</td>
<td>251</td>
<td>4</td>
<td>on</td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>1</td>
<td>I/O Wait</td>
<td>53</td>
<td>253</td>
<td>1</td>
<td>I/O req-r</td>
<td>14</td>
</tr>
<tr>
<td>3</td>
<td></td>
<td>on</td>
<td></td>
<td>254</td>
<td>1</td>
<td>I/O req-r</td>
<td>14</td>
</tr>
<tr>
<td>105</td>
<td>3</td>
<td>sleep</td>
<td>093040</td>
<td>267</td>
<td>4</td>
<td>I/O req-r</td>
<td>43</td>
</tr>
<tr>
<td></td>
<td>3</td>
<td>pg-de</td>
<td>362</td>
<td>267</td>
<td>4</td>
<td>I/O Wait</td>
<td>43</td>
</tr>
<tr>
<td></td>
<td>3</td>
<td>pg-de</td>
<td>305</td>
<td></td>
<td>267</td>
<td>4</td>
<td>I/O Wait</td>
</tr>
<tr>
<td></td>
<td>-</td>
<td>idle</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>110</td>
<td>1</td>
<td>on</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>111</td>
<td>1</td>
<td>create</td>
<td>2</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>114</td>
<td>1</td>
<td>I/O req-r</td>
<td>11</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>I/O Wait</td>
<td>11</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>2</td>
<td>on</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>115</td>
<td>2</td>
<td>pf</td>
<td>201</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Time</td>
<td>Task</td>
<td>Event</td>
<td>Data</td>
<td>Time</td>
<td>Task</td>
<td>Event</td>
<td>Data</td>
</tr>
<tr>
<td>------</td>
<td>------</td>
<td>------------</td>
<td>------</td>
<td>------</td>
<td>------</td>
<td>------------</td>
<td>------</td>
</tr>
<tr>
<td>235</td>
<td>1</td>
<td>I/O Wait</td>
<td>15</td>
<td>500</td>
<td>1</td>
<td>idle</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>idle</td>
<td></td>
<td>502</td>
<td>1</td>
<td>I/O req-r</td>
<td>19</td>
</tr>
<tr>
<td>300</td>
<td>5</td>
<td>Sleep</td>
<td>093041</td>
<td>1</td>
<td>I/O Wait</td>
<td>19</td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>pg-de</td>
<td>501</td>
<td></td>
<td></td>
<td></td>
<td>idle</td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>pg-de</td>
<td>502</td>
<td></td>
<td>552</td>
<td>1</td>
<td>on</td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>pg-de</td>
<td>503</td>
<td></td>
<td>582</td>
<td>1</td>
<td>Sleep</td>
<td>093042</td>
</tr>
<tr>
<td></td>
<td></td>
<td>idle</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>315</td>
<td>1</td>
<td>on</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>317</td>
<td>1</td>
<td>I/O req-r</td>
<td>16</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>1</td>
<td>I/O Wait</td>
<td>16</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>on</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>327</td>
<td>2</td>
<td>pf</td>
<td>204</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>idle</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>330</td>
<td>4</td>
<td>on</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>337</td>
<td>4</td>
<td>pf</td>
<td>403</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>idle</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>350</td>
<td>2</td>
<td>on</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>353</td>
<td>2</td>
<td>pf</td>
<td>205</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>idle</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>360</td>
<td>1</td>
<td>on</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>361</td>
<td>1</td>
<td>pf</td>
<td>105</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>idle</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>365</td>
<td>4</td>
<td>on</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>380</td>
<td>1</td>
<td>on</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>383</td>
<td>1</td>
<td>I/O req-r</td>
<td>17</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>1</td>
<td>I/O Wait</td>
<td>17</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>on</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>393</td>
<td>4</td>
<td>I/O req-w</td>
<td>43</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>398</td>
<td>4</td>
<td>I/O Wait</td>
<td>43</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>on</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>409</td>
<td>2</td>
<td>Post</td>
<td>1, 10</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>410</td>
<td>2</td>
<td>Destroy</td>
<td>2</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>pg-de</td>
<td>201</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>pg-de</td>
<td>202</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>pg-de</td>
<td>203</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>pg-de</td>
<td>204</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>pg-de</td>
<td>205</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>idle</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>440</td>
<td>1</td>
<td>on</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>443</td>
<td>1</td>
<td>Task Wait</td>
<td>2, 10</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>446</td>
<td>1</td>
<td>I/O req-r</td>
<td>18</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>1</td>
<td>I/O Wait</td>
<td>18</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>idle</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>470</td>
<td>4</td>
<td>on</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>471</td>
<td>4</td>
<td>Sleep</td>
<td>093041</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>pg-de</td>
<td>401</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>pg-de</td>
<td>402</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>pg-de</td>
<td>403</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
to device/track 51. The task then waits for the end of this operation, and the Task Initiator selects task 1 for the processor. The events recorded on the SPET are (from Figures A-4, A06a, and A-3):

<table>
<thead>
<tr>
<th>Time</th>
<th>Task</th>
<th>Event</th>
<th>Data</th>
</tr>
</thead>
<tbody>
<tr>
<td>3</td>
<td>5</td>
<td>I/O-req-r</td>
<td>51</td>
</tr>
<tr>
<td>3</td>
<td>5</td>
<td>I/O-wait</td>
<td>51</td>
</tr>
<tr>
<td>3</td>
<td>1</td>
<td>on</td>
<td>--</td>
</tr>
</tbody>
</table>

Task 1 is placed on the processor with no pre-paging operation. The implication is that the task will begin processing in a routine that is already resident - most likely, a system routine. Soon thereafter, task 1 accesses a location that is not resident - location 101. A page fault occurs, and the Task Initiator can find no new task for the processor. The following events are placed on the SPET (Figures A-2 and A-3).

<table>
<thead>
<tr>
<th>Time</th>
<th>Task</th>
<th>Event</th>
<th>Data</th>
</tr>
</thead>
<tbody>
<tr>
<td>4</td>
<td>1</td>
<td>pf</td>
<td>101</td>
</tr>
<tr>
<td>4</td>
<td>-</td>
<td>idle</td>
<td>--</td>
</tr>
</tbody>
</table>

The significant differences between the SPET and the SHET are obvious. Both show that task 1 came on the processor at time 3, and the processor became idle at time 4, and that therefore task 1 used the processor for 1 time unit. However, the SPET also shows that task 1 gained the processor because task 5 waited for I/O; and that task 1 left for a demand paging operation.

The SPET also shows the beginnings and endings of interactions. Tasks 3, 5, 4, and 1 wait for a terminal response at times 102, 300, 471, and 582 respectively.
Inter-relationships of tasks are also shown. Task 1 creates task 2 at time 111. Task 2 is created to destroy itself after it completes its function. It posts its data for task 1 at time 409, and then destroys itself. Task 1 waits for the results of task 2 at time 443, (see Fig. A-14), but they have already been posted, so processing does not stop.

The Processor of the SPET

The SPET preprocessor scans the SPET and decomposes it into task traces of the various tasks. Aside from taking the events representation task demand and placing them on the task traces, it must perform the following functions.

1) Convert the time of the event to task time. This is done the same way as it was done in the SHET processor.

2) Adjust the time of the task on the processor to compensate for the effects of memory interference. The physical event trace does not contain the events showing the actual operation of the I/O devices. Therefore, the SPET preprocessor must generate these events from the I/O demand events. It does this by placing an 'I/O-beg' event on the SHET at a time $\Delta t_q$ after the time of the request event. The delay time, $\Delta t_q$, is the average delay before the data-transfer operation begins for the device $q$ (which is specified by the $q$ element of the event). When the 'I/O-beg' event is to be processed, the memory interference factor $I$ is recalculated, according to the data-transfer rate of the device. If this factor then exceeds some threshold value, which would cause severe degradation of per-
formance, it may be assumed that the I/O operation was postponed due to device or channel unavailability. The 'I/O-beg' event is then placed on the SHET at some time later - after the end of the next I/O operation.

At the time the 'I/O-beg' event is being processed, the 'I/O-end' event is placed on the SHET at time \( t + \Delta t_r \), where \( \Delta t_r \) is the average data-transfer time for device specified by \( q \). When this event is processed, the value of \( f \) used by the preprocessor is again decreased.

The complete SPET preprocessor is presented in Figure 5-2. The definitions of the variables and subroutines, are the same as those of the SHET preprocessor in the last section.

The output of the SPET preprocessor is a set of Task Physical Event Traces - one for each active task on the system at the time the SPET was recorded.

**Task Physical Event Trace**

The Task Physical Event Trace is a record of a task's utilization of the devices whose utilization cannot be predicted from the call on those devices. For those devices whose operation is predictable in this way, the trace will include only the call on the device and the time at which the call occurred, relative to time of the operation of the unpredictable devices. Thus, since the processor is the main resource whose operation time is unpredictable, the event trace consists of a record the time of occurrence of the call: for I/O and paging devices, relative to the processing time of the task. The memory utilization time is just as unpredictable as the processor's, but since memory allocation
Figure 5-2
Preprocessor of SPET
Figure 5-2 (continued)
without the processor is always the consequence of the scheduler rather than the task, the memory allocation time may be considered subsumed by processor time.

A sample TPET is given in Table 5-5. This is the TPET of task 1, taken from the sample SPET of Table 5-4. The Hardware Event Trace for this task was shown in Table 5-3.

Table 5-5

Task Physical Event Trace (TPET) for Task 1

<table>
<thead>
<tr>
<th>Time</th>
<th>Event</th>
<th>Data</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>Wake</td>
<td>093040</td>
</tr>
<tr>
<td>1</td>
<td>pf</td>
<td>101</td>
</tr>
<tr>
<td>2</td>
<td>pf</td>
<td>102</td>
</tr>
<tr>
<td>3</td>
<td>pf</td>
<td>103</td>
</tr>
<tr>
<td>4</td>
<td>pf</td>
<td>104</td>
</tr>
<tr>
<td>5</td>
<td>Create</td>
<td>2</td>
</tr>
<tr>
<td>8</td>
<td>I/O req-r</td>
<td>11</td>
</tr>
<tr>
<td>8</td>
<td>I/O wait</td>
<td>11</td>
</tr>
<tr>
<td>10</td>
<td>I/O req-r</td>
<td>12</td>
</tr>
<tr>
<td>10</td>
<td>I/O wait</td>
<td>12</td>
</tr>
<tr>
<td>12</td>
<td>I/O req-r</td>
<td>13</td>
</tr>
<tr>
<td>12</td>
<td>I/O wait</td>
<td>13</td>
</tr>
<tr>
<td>16</td>
<td>I/O req-r</td>
<td>14</td>
</tr>
<tr>
<td>16</td>
<td>I/O wait</td>
<td>14</td>
</tr>
<tr>
<td>17</td>
<td>I/O req-r</td>
<td>15</td>
</tr>
<tr>
<td>17</td>
<td>I/O wait</td>
<td>15</td>
</tr>
<tr>
<td>19</td>
<td>I/O req-r</td>
<td>16</td>
</tr>
<tr>
<td>19</td>
<td>I/O wait</td>
<td>16</td>
</tr>
<tr>
<td>20</td>
<td>pf</td>
<td>105</td>
</tr>
<tr>
<td>23</td>
<td>I/O req-r</td>
<td>17</td>
</tr>
<tr>
<td>23</td>
<td>I/O wait</td>
<td>17</td>
</tr>
<tr>
<td>27</td>
<td>Task wait</td>
<td>2, 1</td>
</tr>
<tr>
<td>29</td>
<td>I/O req-r</td>
<td>18</td>
</tr>
<tr>
<td>29</td>
<td>I/O wait</td>
<td>18</td>
</tr>
<tr>
<td>31</td>
<td>I/O req-r</td>
<td>19</td>
</tr>
<tr>
<td>31</td>
<td>I/O wait</td>
<td>19</td>
</tr>
<tr>
<td>61</td>
<td>Sleep</td>
<td>333410</td>
</tr>
<tr>
<td>61</td>
<td>pg-de</td>
<td>101</td>
</tr>
<tr>
<td>61</td>
<td>pg-de</td>
<td>102</td>
</tr>
<tr>
<td>61</td>
<td>pg-de</td>
<td>103</td>
</tr>
<tr>
<td>61</td>
<td>pg-de</td>
<td>104</td>
</tr>
<tr>
<td>61</td>
<td>pg-de</td>
<td>105</td>
</tr>
</tbody>
</table>
The Logical Event Trace

By viewing the TPET, no reasons can be seen for any of the physical operations that take place. The user tasks seem to be generating them randomly. Yet, within a multiprogrammed environment, the operating system actually issues the calls for physical I/O operations. The computation time evident in the user task trace that immediately preceded the call for the I/O operation, was not computation within a user program, but computation within a system routine administering the I/O request generated by the user program. The user program specifies the I/O requirements at the logical level.

Suppose that changes in the operation of various system logical I/O routines are to be investigated. The calls on the system logical I/O operations then become the user demand parameter of interest, rather than the PI/O operations that result from the Logical I/O Specification.

These calls must be recorded in an event trace, which will be called the System Logical Event Trace or SLET. The physical I/O operations that result from these logical events will be generated by the simulator of the logical I/O routines; recording them in the SLET will not be necessary.

It should be noted that system logical I/O routines can be modified only in such a way that the user program calls on them are not affected. Otherwise, the modification would entail reprogramming the
user programs. Thus, the calls on the L I/O routines are valid representations of the user programs demand for logical I/O operations, and will remain valid even if those routines are modified.

Definition of the SLET

The interface at which the logical events are recorded is at a higher level in the system organization than the level at which the physical events are recorded. In Figure 5-3a, it is seen that the SLET is recorded at the entrance to the L I/O routines. The calls on the physical I/O routines occur as a consequence of these L I/O calls.

Yet, aside from the logical I/O routines, there are many other system routines which call on the physical I/O level. These routines are considered part of the user programs. A more accurate representation of the entire system is shown in Fig. 5-3b in which the user programs are divided into 'user' and 'system' parts. If the SLET is recorded at interface 2 of the figure, all of the physical I/O events that result from the calls on these systems programs will be missing. Therefore, the simulator must include a model of each of these programs, so that its effect on the hardware may be recovered. The modelling of all these programs is too large a task to be considered.

Choosing the SLET at interface 1 will result in a collection of all the calls on the logical I/O routines, together with all of the calls on the physical I/O routines that are not made from the logical I/O level. There are problems in implementing this level in a large system.

Because the OS is structured as an interrupt-driven system, it is easy enough to record all of the I/O's that occur by placing the recording mechanism at the entrance to the physical I/O subroutines. In
Figure 5-3 Interface Choices for SLET
order to record only a subset of them, one of these two alternatives must be chosen.

1. Either the recording device for the P I/O event must be placed at the point within the system program at which the physical I/O routine is called (except for the calls from the logical routines).

2. Or else the recording will be made at the entrance to the P I/O routine; but whenever the call is made from the logical I/O subsystem, it will be suppressed by an operation within the logical I/O routine prior to the transfer to the physical I/O routine.

The first alternative requires that all the places within the system, at which a physical I/O operation takes place - which may be many - be located, and a call on the recording subroutine be inserted at each such location. These modifications are to be made in the part of the system of which the model is to be unaware - this means more system routines must be investigated, than are necessary for the construction of the model. Furthermore, it may not be possible to use one unified recording method for all these systems programs, because they may operate under diverse security and privilege constraints.

On the other hand, implementation of the second alternative requires that the call on the physical I/O subroutine be modified, to include a "no record" parameter, and each of the P I/O calls in the logical I/O subsystem set this parameter. Each P I/O subroutine must check this parameter to determine whether the event it is to be recorded. These operations increase the load that the event recording subsystem places on the
system operation. Also, if the passing of the "no record" parameter is not done carefully, by means of the parameter-transfer mechanism provided by the program-generated interrupt technique, the results will be invalid. For example, if the technique of setting a flag in the P I/O routine before the transfer to it is used, the intervention of asynchronous interrupts will cause the flag to be reset to unwanted condition.

The concern for generality in the recording technique - aside from the specific problems of implementation - dictates that neither of these alternatives for the development of the SLET should be chosen. The SLET, imposed upon the SPET, is just one of a sequence of event traces, that may be defined, each imposed upon, or supplementing, a lower trace. The convention, once begun, that a call on a lower level program must be accompanied by an indication of whether the call must be recorded, will lead to complexities in later traces, when the recording information may have to be passed through several possible recording levels.

Therefore, the SLET will be composed of the entire SPET, as well as the Logical I/O events 'begin L I/O' - recorded at the entrance to the logical I/O routine. This is a combination of interface 1 and the SPET interface in Fig. 5-3b. The hardware demand events which result from the logical I/O operation will now be doubly represented - once by the L I/O events and once by the P I/O event. The P I/O events will be purged from the SLET by a preprocessor, just as the preprocessor of the SPET purged it of system contributed information.
Identification of Entry and Exit Points

The SLET preprocessor, however, is not equipped to decide which of the hardware operations following a 'begin L I/O' event are due to the operation of the L I/O program, and which occur after the return to the user program. To make this determination would require at least as much information about the operation of the L I/O program as would be contained in a full model of it. This is not feasible. The SLET, therefore, must include the events of both entry and exit from logical I/O subroutines. Within the SLET, these will appear as parenthesis around the physical events that result from the operation of the logical event trace.

The ability to identify the entrances and exits of each routine may be considered a constraint on the method in which the logical I/O routines must be constructed. If certain conventions of modularity in coding techniques are followed in order to facilitate system debugging and maintenance, the constraint will be met. If transfers are made among system routines in such a way that entrances and exits of a routine are not clear, much greater effort must be expended in order to implement this event trace.

The following is an exposition of the methodology for locating the points at which the recording mechanisms are to be placed. A standard subroutine call is one in which a program transfers to a subroutine and provides the subroutine with the return address at the same time. The return address is the current contents of the program counter - the address of the instruction that would have been executed next, had it not been for the transfer. Examples of standard subroutine calls are - the SWC
and the 'branch and link' instruction. This return address will be called the standard return. Other return addressed may be passed to the subroutine as parameters.

Generally, the entry points of the logical I/O routine are easily found. User programs call the L I/O routines only through the SVC call, and the entry point is the address in the Branch Table. System routines may branch directly to the routine. In the worst case, a search of all the privileged system code may be necessary to identify each entry to a system routine.

The identification of the exit points of the system routines of interest is more difficult.

The system of subroutines that made up the central portion of the operating system is recursive rather than hierarchical. If routine A calls on routine B as a subroutine, B will, after completing its processing, return to the calling point in routine A. However, during its processing, routine B may call on other subroutines, which may result in calls on A and/or calls on itself. It cannot be said that B is at a lower level than A; they are both subroutines to each other.

This type of recursive structure must be implemented by using a push-down storage. When A calls on B, the variables of A, which include A's return address, must be placed in a specific location whose address is not a program constant of A. These should not be modified even if A is re-entered for a new application.

The process of storing the program variables is called pushing the pushdown store. After B returns to A, the data of A is retrieved. This is called popping the pushdown. The pushes and pops are done by small
open subroutines before and after the 'transfer to B' instruction.

In some cases, routine B will not return to the point at which it was called in routine A.

The following two cases must be distinguished:

1) Routine B is logically linked to routine A. This means that routine B has knowledge of entry or return points in A, other than those explicitly passed to it via the calling mechanism.

2) Routines A and B are linked to each other only through the standard subroutine calls.

In the first case, if A calls on B, in a non-standard call (i.e. without recording the return address) and B returns to the call point, then B must have the return address of A as a program constant. No other program can make this call on B. If A makes a standard call on B without specifying return points in the parameters it sends to B, and B returns to one point in A other than the call point, then B contains an address constant, the return point in A. Again, no other program may make this call on B. Under each of these cases, routine B - with respect to this entry point - must be considered part of routine A.

In the second case, routine B is independent of program A. Four cases can be distinguished.

1) Routine B takes the standard return to routine A - the calling point (See Fig. 5-1a).

2) Routine B returns to some point in the pushdown, above the program A. For example it may return to the program that called on A, by finding its return address in the pushdown
Figure 5-4
Subroutine Exits

a) No exit
b) Pushdown exit
c) Terminal exit
d) Parameter-specified exit
e) Recording Pushdown exit
f) Recording Terminal exit
stack. (See Fig. 5-4b)

3) Routine B does not return - it simply transfers to some other program. This is a non-return transfer to a routine that is not logically linked to B. (See Figure 5-4c)

4) Routine B returns to some point specified by program A. This address will be specified in a calling parameter. (See Figure 5-4d)

In case 1, event recording mechanisms are placed at the entry and exit points of the subroutine A.

In case 2, a conditional event recording is placed at the exit from routine B, since that may or may not be an exit from routine A. If the exit is the standard return to A, the event is not recorded. If it is not, it must be determined whether the call was made from program A.

In the implementation of the event trace with calls on A, all of the possible return points to routine A will be stored as program constants of routine B, and they must be checked before the return is taken. If B is not returning to A, the event 'exit from A' is recorded. If the return is to A, the event is not recorded. This mechanism is shown in Figure 5-4e.

In case 3, routine B will transfer out of the routine, with no intention of returning. This is essentially a 'terminal processing' for the task for this interaction, because the return address to the user task is lost. The entry point to all such terminal routines is marked with an 'exit' event. The result of the 'exit' event will be to mark the exits if all of the routines that the task has entered. This is shown in Figure 5-4f.
Case 4 is the most difficult to resolve. Routine B will, in
general, expect several inputs from the calling routine. For each of
these, it must be determined -

a) Is it a return to the calling program?
b) Is it a return to a level above the call - i.e., an address
that A had been passed?
c) Is it a transfer out of B?

Whether it is case a or b is determined by examination of A. In
case a), no event recording is necessary. In case b) the event recording
is the same as in case 2. If neither a) nor b), examination of routine
B will show whether the transfer is intended to be a return call - in which
case no event need be recorded - or whether it is a terminal transfer -
which is equivalent to case 3.

The events of the SLET include all of the events of the SPET, with
the addition of two events for each logical I/O routine. The two events
mark the entrance and exit to the routine. Each of the events common
to the two event traces is recorded in the same way as was for the SPET.

Recording the SLET

Several types of logical I/O routines are presented in the sample
operating system. The events 'Get' and 'End Get' are recorded as shown
in the Get macro - Fig. A-10. 'Open' and 'End Open' are marked in the
Open macro - Fig. A-11. In each of these, it is assumed that there is one
unique exit point to the macro; therefore only one recording point is
necessary. In cases where the exit point is not unique, the recording
mechanism must be placed at each exit from the macro.
Definition of Logical Level

Beyond the Get and Open macros - the choice of which other routines must be included in order to define a logical I/O 'level' - or any other level - is somewhat arbitrary. One constraint on the definition is that once an initial set of subroutines is chosen to define the level, and the entrances and exits to these subroutines are recorded, then all the programs called between an entrance to one of these subroutines and the exit from it must belong to that level. A level will always encompass all previously defined levels, therefore, routines will be said to belong to a particular level if they belong to any level below it.

Starting with a desired set of subroutines, and applying the principle will yield a set of routines that define the level. This set is complete in the sense that once the event marking the entrance to the level is recorded, the processor may be assumed to be processing in a routine of that level until the event marking the exit from the original routine is recorded.

Another logical I/O operation, Vam Get, is shown in Fig. A-12, and similarly, the 'Vam Get' event and 'End Vam Get' events are marked. This logical I/O routine calls on the GET macro, another logical I/O routine. Thus, the sequence

```
Vam Get
Get
End Get
End Vam Get
```

will be seen in the SLET.
Figure 5-5a is an example of the Vam Get macro calling on the Get macro by the SVC interrupt mechanism and standard return. This call and return will result in the above SLET sequence. It is obvious that the logical level could be defined to be either the Get Macro, and all its consequences, without the Vam Get macro, or else the Vam Get macro, including its consequence, the Get macro.

In many cases, system programs do not call on each other by the SVC mechanism provided for user programs. The SVC, which causes an interrupt, is more wasteful of processor time than the direct transfer of control.

A direct transfer of control between the Vam Get macro and the Get macro is shown in Fig. 5-5b. The points at which the Get events are recorded could be placed so that this call on the Get macro by the Vam Get macro is not recorded, as in the figure. In this case, it is not possible to define a logical level to be just the Get macro, since at least one call on this level will not be recorded. The Vam Get macro must be included.

Figure 5-5

Recording I/O Routines
In the examples presented here, for consistency of presentation, it will be assumed that all calls on a particular level of the operating system are recorded, even though they are made from within that same level. If this is not true, in an actual case (for example, the structure of Figure 5-5a is created) it simply means a simplification of the event trace. This effect can be achieved even if design considerations dictate that structures such as Fig. 5-5b be created, by moving the event recording steps closer to the interior of the routine, where they will be taken by every call on the subroutine.

Example of SLET Recording

The example of the System Logical Event Trace is taken from the same instance of system operation that had resulted in the SLET and SPET examples. This trace will have all the events of the SPET, with the additional events of entrance and exit to the Logical I/O routines. This trace is shown in Table 5-6.

Task 5 issues the first call for a GET operation in the trace at time 5; record 01 in the file named AF is desired. The Get operation is concluded at time 98, with the occurrence of the 'End Get' event. Between these two events, task 5 issues two physical I/O calls, at times 3 and 44. The device and cylinder/track identifications for these operations are the same as those in the SLET. It may be assumed that one of these physical I/O calls is for the directory of the file, and the other for the data page.

Other occurrences of the GET call do not result in two I/O operations. Task 4 issues a Get call at time 60, and the GET operation is completed at time 144, after only one PT/0 call. At time 405, task 2
<table>
<thead>
<tr>
<th>1</th>
<th>1</th>
<th>Wake 093040</th>
<th>110</th>
<th>1</th>
<th>on</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>5</td>
<td>GET AF,01</td>
<td>111</td>
<td>1</td>
<td>create 2</td>
</tr>
<tr>
<td>3</td>
<td>5</td>
<td>I/O req-r 51</td>
<td>112</td>
<td>1</td>
<td>open KF</td>
</tr>
<tr>
<td></td>
<td>5</td>
<td>Wait 51</td>
<td></td>
<td>1</td>
<td>open FOFL</td>
</tr>
<tr>
<td>4</td>
<td>1</td>
<td>pf idle 101</td>
<td>114</td>
<td>1</td>
<td>I/O req-r 11</td>
</tr>
<tr>
<td></td>
<td>5</td>
<td>on</td>
<td></td>
<td>1</td>
<td>I/O wait 11</td>
</tr>
<tr>
<td>5</td>
<td>3</td>
<td>on</td>
<td></td>
<td>2</td>
<td>on</td>
</tr>
<tr>
<td>8</td>
<td>3</td>
<td>pf idle 301</td>
<td>115</td>
<td>2</td>
<td>pf 201</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>idle</td>
</tr>
<tr>
<td>20</td>
<td>1</td>
<td>on</td>
<td></td>
<td>140</td>
<td>2</td>
</tr>
<tr>
<td>21</td>
<td>1</td>
<td>pf idle 102</td>
<td>141</td>
<td>2</td>
<td>pf 202</td>
</tr>
<tr>
<td>35</td>
<td>4</td>
<td>on</td>
<td></td>
<td>144</td>
<td>4</td>
</tr>
<tr>
<td>36</td>
<td>4</td>
<td>put BF,02</td>
<td></td>
<td>146</td>
<td>4</td>
</tr>
<tr>
<td></td>
<td>4</td>
<td>I/O req-w 41</td>
<td></td>
<td></td>
<td>pf 401</td>
</tr>
<tr>
<td>37</td>
<td>4</td>
<td>I/O wait 41</td>
<td></td>
<td></td>
<td>idle</td>
</tr>
<tr>
<td>43</td>
<td>5</td>
<td>on</td>
<td></td>
<td>170</td>
<td>2</td>
</tr>
<tr>
<td>44</td>
<td>5</td>
<td>I/O req-r 52</td>
<td>173</td>
<td>2</td>
<td>GET DF,05</td>
</tr>
<tr>
<td></td>
<td>5</td>
<td>I/O wait 52</td>
<td></td>
<td>174</td>
<td>2</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>182</td>
<td>1</td>
</tr>
<tr>
<td>50</td>
<td>1</td>
<td>on</td>
<td></td>
<td>183</td>
<td>1</td>
</tr>
<tr>
<td>51</td>
<td>1</td>
<td>pf idle 103</td>
<td>184</td>
<td>1</td>
<td>GET FOFL,KF</td>
</tr>
<tr>
<td>65</td>
<td>3</td>
<td>on</td>
<td></td>
<td>185</td>
<td>1</td>
</tr>
<tr>
<td>78</td>
<td>3</td>
<td>pf idle 302</td>
<td>186</td>
<td>1</td>
<td>I/O wait 12</td>
</tr>
<tr>
<td></td>
<td>4</td>
<td>on</td>
<td></td>
<td>187</td>
<td>1</td>
</tr>
<tr>
<td>80</td>
<td>1</td>
<td>on</td>
<td></td>
<td>188</td>
<td>5</td>
</tr>
<tr>
<td>81</td>
<td>1</td>
<td>pf idle 104</td>
<td>189</td>
<td>5</td>
<td>I/O req-w 5h</td>
</tr>
<tr>
<td></td>
<td>4</td>
<td>on</td>
<td></td>
<td>188</td>
<td>5</td>
</tr>
<tr>
<td>82</td>
<td>4</td>
<td>Get CF,03</td>
<td></td>
<td>189</td>
<td>5</td>
</tr>
<tr>
<td></td>
<td>4</td>
<td>I/O req-r 42</td>
<td></td>
<td></td>
<td>idle</td>
</tr>
<tr>
<td>95</td>
<td>3</td>
<td>on</td>
<td></td>
<td>209</td>
<td>5</td>
</tr>
<tr>
<td>98</td>
<td>7</td>
<td>on</td>
<td></td>
<td>210</td>
<td>5</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>idle</td>
</tr>
<tr>
<td>99</td>
<td>5</td>
<td>end GET AF,01</td>
<td>215</td>
<td>2</td>
<td>on</td>
</tr>
<tr>
<td></td>
<td>5</td>
<td>I/O req-r 53</td>
<td>219</td>
<td>2</td>
<td>end GET DF,05</td>
</tr>
<tr>
<td>101</td>
<td>5</td>
<td>I/O Wait 53</td>
<td>222</td>
<td>1</td>
<td>on</td>
</tr>
<tr>
<td>103</td>
<td>3</td>
<td>on</td>
<td></td>
<td>224</td>
<td>1</td>
</tr>
<tr>
<td>105</td>
<td>3</td>
<td>sleep</td>
<td></td>
<td>225</td>
<td>1</td>
</tr>
<tr>
<td></td>
<td>3</td>
<td>pg-de 302</td>
<td>230</td>
<td>5</td>
<td>on</td>
</tr>
<tr>
<td></td>
<td>3</td>
<td>pg-de 305</td>
<td></td>
<td>231</td>
<td>5</td>
</tr>
<tr>
<td>232</td>
<td>5</td>
<td>pf</td>
<td>502</td>
<td>365</td>
<td>4</td>
</tr>
<tr>
<td>247</td>
<td>2</td>
<td>GET</td>
<td>DF,06</td>
<td>382</td>
<td>1</td>
</tr>
<tr>
<td>249</td>
<td>2</td>
<td>I/O req-r</td>
<td>21</td>
<td>383</td>
<td>1</td>
</tr>
<tr>
<td></td>
<td>4</td>
<td>I/O wait</td>
<td>21</td>
<td></td>
<td>1</td>
</tr>
<tr>
<td>254</td>
<td>1</td>
<td>on</td>
<td></td>
<td></td>
<td>4</td>
</tr>
<tr>
<td></td>
<td>1</td>
<td>end GET</td>
<td>FOFl,KP</td>
<td>390</td>
<td>4</td>
</tr>
<tr>
<td></td>
<td>1</td>
<td>end OPEN</td>
<td>KF</td>
<td>393</td>
<td>4</td>
</tr>
<tr>
<td>257</td>
<td>1</td>
<td>VAM GET</td>
<td>KF,BEG</td>
<td>398</td>
<td>4</td>
</tr>
<tr>
<td></td>
<td>1</td>
<td>GET</td>
<td>KF,BEG</td>
<td></td>
<td>2</td>
</tr>
<tr>
<td>258</td>
<td>1</td>
<td>I/O req-r</td>
<td>14</td>
<td>405</td>
<td>2</td>
</tr>
<tr>
<td></td>
<td>1</td>
<td>I/O wait</td>
<td>14</td>
<td>408</td>
<td>2</td>
</tr>
<tr>
<td></td>
<td>4</td>
<td>on</td>
<td></td>
<td>409</td>
<td>2</td>
</tr>
<tr>
<td>260</td>
<td>5</td>
<td>on</td>
<td></td>
<td>410</td>
<td>2</td>
</tr>
<tr>
<td>264</td>
<td>5</td>
<td>pf</td>
<td>503</td>
<td></td>
<td>2</td>
</tr>
<tr>
<td></td>
<td>4</td>
<td>on</td>
<td></td>
<td></td>
<td>2</td>
</tr>
<tr>
<td>266</td>
<td>4</td>
<td>GET</td>
<td>CF,07</td>
<td></td>
<td>2</td>
</tr>
<tr>
<td>267</td>
<td>4</td>
<td>I/O req-r</td>
<td>43</td>
<td></td>
<td>2</td>
</tr>
<tr>
<td></td>
<td>4</td>
<td>I/O wait</td>
<td>43</td>
<td></td>
<td>2</td>
</tr>
<tr>
<td></td>
<td>4</td>
<td>idle</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>284</td>
<td>1</td>
<td>on</td>
<td></td>
<td>440</td>
<td>1</td>
</tr>
<tr>
<td>285</td>
<td>1</td>
<td>I/O req-r</td>
<td>15</td>
<td>441</td>
<td>1</td>
</tr>
<tr>
<td></td>
<td>1</td>
<td>I/O wait</td>
<td>15</td>
<td></td>
<td>1</td>
</tr>
<tr>
<td></td>
<td></td>
<td>idle</td>
<td></td>
<td>443</td>
<td>1</td>
</tr>
<tr>
<td>290</td>
<td>5</td>
<td>on</td>
<td></td>
<td>446</td>
<td>1</td>
</tr>
<tr>
<td>297</td>
<td>5</td>
<td>Get</td>
<td>AF,08</td>
<td></td>
<td>1</td>
</tr>
<tr>
<td></td>
<td></td>
<td>idle</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>300</td>
<td>5</td>
<td>Sleep</td>
<td></td>
<td>470</td>
<td>4</td>
</tr>
<tr>
<td></td>
<td>2</td>
<td>on</td>
<td></td>
<td></td>
<td>4</td>
</tr>
<tr>
<td>301</td>
<td>2</td>
<td>end GET</td>
<td>DF,06</td>
<td>471</td>
<td>4</td>
</tr>
<tr>
<td></td>
<td></td>
<td>idle</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>315</td>
<td>1</td>
<td>on</td>
<td></td>
<td>500</td>
<td>1</td>
</tr>
<tr>
<td>316</td>
<td>1</td>
<td>end GET</td>
<td>KF,BEG</td>
<td></td>
<td>1</td>
</tr>
<tr>
<td></td>
<td>1</td>
<td>end VAM GET</td>
<td>KF,BEG</td>
<td></td>
<td>1</td>
</tr>
<tr>
<td></td>
<td></td>
<td>idle</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>317</td>
<td>1</td>
<td>I/O req-r</td>
<td>16</td>
<td>552</td>
<td>1</td>
</tr>
<tr>
<td></td>
<td>1</td>
<td>I/O wait</td>
<td>16</td>
<td></td>
<td>1</td>
</tr>
<tr>
<td></td>
<td>2</td>
<td>on</td>
<td></td>
<td>582</td>
<td>1</td>
</tr>
<tr>
<td></td>
<td></td>
<td>idle</td>
<td></td>
<td></td>
<td>1</td>
</tr>
<tr>
<td>327</td>
<td>2</td>
<td>pf</td>
<td>204</td>
<td></td>
<td>1</td>
</tr>
<tr>
<td></td>
<td>4</td>
<td>on</td>
<td></td>
<td></td>
<td>1</td>
</tr>
<tr>
<td>330</td>
<td>4</td>
<td>on</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>333</td>
<td>4</td>
<td>end GET</td>
<td>CF,07</td>
<td></td>
<td></td>
</tr>
<tr>
<td>337</td>
<td>4</td>
<td>pf</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>350</td>
<td>2</td>
<td>on</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>353</td>
<td>2</td>
<td>pf</td>
<td>205</td>
<td></td>
<td></td>
</tr>
<tr>
<td>360</td>
<td>1</td>
<td>on</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>361</td>
<td>1</td>
<td>pf</td>
<td>105</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
begins a logical Get operation, which is completed at time 408, without any physical I/O taking place. In this Get call, the required record was resident in main memory as well as virtual memory when the Get call took place.

The 'Put' events that appear in the event traces are taken from the 'Put' physical I/O macro, whose operation is analogous to the 'Get' operation. This macro has not been made explicit in the definition of the system, but its presence in the event trace is a reminder that there are other physical I/O macros in any implementation. The way in which they are recorded is the same as that of the macros that have been made explicit.

Recursive calls on the system logical macros are also recorded. At time 112 task 1 issues a call for file KF to be opened. The Open macro discovers that the file of files for task 1 (FOF1) is not opened, thus the next call on the Open macro quickly occurs. This call results in the logical Get call to obtain the definition of FOF1 from the Syscat. Exits are taken from these routines in LIFO order, at times 182, 183 and 256.

Preprocessor of SLET

The SLET preprocessor will basically act similar to the SLET preprocessor. The output of this preprocessor is a set of Task Logical Event Traces (TELTS). A TELT contains a record of the task's processor utilization, each call on a logical I/O routine, and each call for a physical I/O operation that does not result from a logical I/O operation. Thus, one major function of the SLET preprocessor is to expunge from
the task traces all of the physical I/O events, whose presence in the logical event trace is redundant, since these events are the consequence of calls on the logical I/O routines. In fact, some of the events representing calls on the logical I/O routines are also redundant, since they are made from logical level programs.

The method used by the preprocessor to prevent these events from appearing in the TLET is the following. The preprocessor maintains a push down counter $d_k$ for each task $k$. This counter is incremented each time a 'begin LI/O' event for task $k$ is encountered in the SLET and is decremented at each 'end LI/O' event for the task. The counter $d_k$ will have value zero only when task $k$ is not in any LI/O routine. Physical PI/O events that occur for task $k$ while $d_k > 0$ will not be placed on the TLET for task $k$. When the 'exit' event for task $k$ appears, $d_k$ is set to zero, since the effect of the 'exit' is to indicate that task $k$ has left each of the routines whose operation is being monitored.

The SLET preprocessor will convert PI/O operations into events of device 'begin' and 'end' and insert them in the SLET, in order to recover the effects of I/O interference with the processor for memory cycles. These device begin and end events are handled by this preprocessor, just as they were in the SPET preprocessor.

The entire SLET preprocessor is shown in Figure 5-6.

**Example of the TLET**

The task logical event traces that are the output of the SLET preprocessor do not contain any indications of the operation of the logical I/O subroutines. Neither processing time, physical I/O operations, nor paging faults occurring within the subroutine are visible in the TLET.
Figure 5-6
Preprocessor of the SLET
Figure 5-6 (continued)
The event in the TLET representing the call on a LI/O routine is the only indication that that routine has been performed.

The TLET for task 1, which was extracted from the example of the SLET, is shown in Table 5-7. Comparing this task trace with the TPET of the same task in this instance of system operation (Table 5-5), it is seen that most of the physical I/O requests (all the requests whose device/cylinder-track id. is between 11 and 15, for example) are subsumed under a Logical I/O operation. The remaining physical I/O operations are located at different task times than they were in the TPET, because the processing time in the I/I/O operations is not included in the task time in the TLET. The Physical I/O operation to device/location 18, for example, takes place at time 29 in the TPET, and at time 16 in the TLET.
Table 5-7

Task Logical Event Trace (TLET) For Task 1

<table>
<thead>
<tr>
<th>Time</th>
<th>Event</th>
<th>Data</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>Wake</td>
<td>093040</td>
</tr>
<tr>
<td>1</td>
<td>pf</td>
<td>101</td>
</tr>
<tr>
<td>2</td>
<td>pf</td>
<td>102</td>
</tr>
<tr>
<td>3</td>
<td>pf</td>
<td>103</td>
</tr>
<tr>
<td>4</td>
<td>pf</td>
<td>104</td>
</tr>
<tr>
<td>5</td>
<td>Create</td>
<td>2</td>
</tr>
<tr>
<td>6</td>
<td>Open</td>
<td>KF</td>
</tr>
<tr>
<td>7</td>
<td>Vam Get</td>
<td>KF, BFIC</td>
</tr>
<tr>
<td>8</td>
<td>I/O req-r</td>
<td>16</td>
</tr>
<tr>
<td>8</td>
<td>I/O wait</td>
<td>16</td>
</tr>
<tr>
<td>9</td>
<td>pf</td>
<td>105</td>
</tr>
<tr>
<td>11</td>
<td>Vam Get</td>
<td>KF, SUB</td>
</tr>
<tr>
<td>13</td>
<td>Task Wait</td>
<td>2, 1</td>
</tr>
<tr>
<td>16</td>
<td>I/O req-r</td>
<td>18</td>
</tr>
<tr>
<td>16</td>
<td>I/O wait</td>
<td>18</td>
</tr>
<tr>
<td>18</td>
<td>I/O req-r</td>
<td>19</td>
</tr>
<tr>
<td>18</td>
<td>I/O wait</td>
<td>19</td>
</tr>
<tr>
<td>48</td>
<td>Sleep</td>
<td>093041</td>
</tr>
<tr>
<td>48</td>
<td>pg-dc</td>
<td>101</td>
</tr>
<tr>
<td>48</td>
<td>pg-de</td>
<td>102</td>
</tr>
<tr>
<td>48</td>
<td>pg-dc</td>
<td>103</td>
</tr>
<tr>
<td>48</td>
<td>pg-dc</td>
<td>104</td>
</tr>
<tr>
<td>48</td>
<td>pg-dc</td>
<td>105</td>
</tr>
</tbody>
</table>
The Relocation Event Trace

The last event trace to be developed is the event trace which includes operation of the dynamic loader of the system. We will call this the System Relocation Event Trace or SRET.

This event trace could be defined in a way analogous to the definition of the logical event trace; that is, it could merely be an extension of the preceding event trace. This technique, however, is not generalizable, since the number of events that must be recorded would soon require a significant part of the CP time, main memory, and I/O operations. Therefore, the method will be generalized, so that some of the events of lower-level event traces can be omitted as higher level events are added.

First, it was noted that in order to define a logical event trace, all of the calls on the physical level had to be recorded, because not all the calls on the physical level result from the logical level (Fig. 5-7a). Now, in order to eliminate the necessity of recording the Physical I/O events in this higher level, all the calls on any system program that call the Physical I/O level must be recorded. Thus, the remaining system routines which were considered 'user program' in the last trace, must be broken up into two groups: those that call on the PI/O level, and those that do not. All the calls on the former group must be recorded in this next level event trace. This interface is shown in Figure 5-7b. As is seen in the figure, the calls on the physical I/O level are then omitted from the trace.
Figure 5-7

Interface for Relocation Event Trace
It will be assumed that this next level event trace includes events marking the entrance and exits to all programs calling on the Physical I/O level. The only remaining programs in the demonstrated operating system that call on the physical I/O level are the loading programs. In the examples of the recording and the processing of this trace it will be assumed that the loading macros include all of the remaining physical I/O calls in the system. Again, in a real system, there will be more macros which must be included at this level, but these will be recorded in the same way as the loading macros. The smaller set of events in the trace will also suffice for the demonstration of the event preprocessor.

Definition of the Relocation Event Trace

The Relocation Event Trace will include the events marking the entrance and exits of the Load and the Page Load macros, and will not include PI/O events on the Physical Event Trace.

The following events belong to the Relocation Event Trace:

The 'on' and 'idle' events - which were defined originally in the Hardware Event Trace.

The 'pf', 'pg-in', 'pg-out', 'pg-de', 'wake', 'sleep', 'task wait', 'post', 'create' and 'destroy' events - which were defined in the Physical Event Trace.

The 'Get', 'End Get', 'Van Get', and 'End Van Get' events - which were defined in the Logical Event Trace.

The 'load', 'end load', 'Page Load' and 'End Page Load' events - defined for the Relocation Event Trace. The data element of the load and page load events are parameters of the corresponding
macros.

Recording of the SRET

Each of the previously defined events of the SRET is recorded in the same way as it was in the previous event trace. These recording locations are shown in Figures A-1 to A-11, of the sample operating system.

The additional events are shown in Figures A-11 and A-12. The 'page load' event is recorded after the branch is taken to perform page loading, rather than at the very entrance to the page fault interrupt response routine. It should be noted that the 'page fault' event is not recorded at the entrance to the page fault interrupt response routine either (Figure A-12) but that it is recorded after the page load decision has been made. If a page load is not to be performed, the page fault response of Figure A-2 takes place.

Example of SRET Recording

The system relocation event trace for the instance of system operation used in previous examples, is shown in Table 5-8.

The following should be noted in this example of the SRET:

No physical I/O operation events are included in the SRET. Task 5, for example, begins a logical operation (Get) at time 2, and completes it at time 98. In the previous trace, it was seen that this logical operation consisted of two physical operations. Here, none are recorded.

The physical I/O operations that were not included between the events marking the boundaries of the logical I/O operations, are now enclosed by events marking the relocation routines: - Load and Page Load. For example, the I/O request by task 1 at time 317 in the SRET (Table
### Table 5-8

Example of System Relocation Event Trace (SRET)

<table>
<thead>
<tr>
<th>Time</th>
<th>Task</th>
<th>Event</th>
<th>Data</th>
<th>Time</th>
<th>Task</th>
<th>Event</th>
<th>Data</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td></td>
<td>Wake</td>
<td>093040</td>
<td>144</td>
<td></td>
<td></td>
<td>idle</td>
</tr>
<tr>
<td>2</td>
<td>5</td>
<td>GET</td>
<td>AF, 01</td>
<td>146</td>
<td>4</td>
<td>pf</td>
<td>401</td>
</tr>
<tr>
<td>3</td>
<td></td>
<td>- idle</td>
<td></td>
<td>170</td>
<td>2</td>
<td>on</td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>1</td>
<td>pf</td>
<td>101</td>
<td>173</td>
<td>2</td>
<td>Get</td>
<td>DF, 05</td>
</tr>
<tr>
<td>5</td>
<td>3</td>
<td>on</td>
<td></td>
<td>174</td>
<td>2</td>
<td>pf</td>
<td></td>
</tr>
<tr>
<td>6</td>
<td></td>
<td>- idle</td>
<td></td>
<td>182</td>
<td>1</td>
<td>on</td>
<td></td>
</tr>
<tr>
<td>20</td>
<td>1</td>
<td>on</td>
<td></td>
<td>183</td>
<td>1</td>
<td>end Get</td>
<td>SYS, F0F1</td>
</tr>
<tr>
<td>21</td>
<td>1</td>
<td>pf</td>
<td>102</td>
<td>184</td>
<td>1</td>
<td>Get</td>
<td>F0F1, KF</td>
</tr>
<tr>
<td>35</td>
<td>4</td>
<td>on</td>
<td></td>
<td>185</td>
<td>4</td>
<td>on</td>
<td></td>
</tr>
<tr>
<td>36</td>
<td>4</td>
<td>Put</td>
<td>BF, 02</td>
<td>187</td>
<td>4</td>
<td>pf</td>
<td>402</td>
</tr>
<tr>
<td>37</td>
<td></td>
<td>- idle</td>
<td></td>
<td>188</td>
<td>5</td>
<td>on</td>
<td></td>
</tr>
<tr>
<td>43</td>
<td>5</td>
<td>on</td>
<td></td>
<td>189</td>
<td></td>
<td>- idle</td>
<td></td>
</tr>
<tr>
<td>44</td>
<td></td>
<td>- idle</td>
<td></td>
<td>209</td>
<td>5</td>
<td>on</td>
<td></td>
</tr>
<tr>
<td>50</td>
<td>1</td>
<td>on</td>
<td></td>
<td>210</td>
<td>5</td>
<td>pf</td>
<td>501</td>
</tr>
<tr>
<td>51</td>
<td>1</td>
<td>pf</td>
<td>103</td>
<td>4</td>
<td></td>
<td>- idle</td>
<td></td>
</tr>
<tr>
<td>65</td>
<td>3</td>
<td>on</td>
<td></td>
<td>215</td>
<td>2</td>
<td>on</td>
<td></td>
</tr>
<tr>
<td>78</td>
<td>3</td>
<td>pf</td>
<td>302</td>
<td>219</td>
<td>2</td>
<td>end Get</td>
<td>DF, 05</td>
</tr>
<tr>
<td>80</td>
<td>1</td>
<td>on</td>
<td></td>
<td>222</td>
<td>1</td>
<td>on</td>
<td></td>
</tr>
<tr>
<td>81</td>
<td>1</td>
<td>pf</td>
<td>104</td>
<td>224</td>
<td>2</td>
<td>on</td>
<td></td>
</tr>
<tr>
<td>82</td>
<td>4</td>
<td>on</td>
<td></td>
<td>230</td>
<td>5</td>
<td>on</td>
<td></td>
</tr>
<tr>
<td>83</td>
<td>4</td>
<td>Get</td>
<td>CF, 03</td>
<td>231</td>
<td>5</td>
<td>end Put</td>
<td>AF, C4</td>
</tr>
<tr>
<td>95</td>
<td>3</td>
<td>on</td>
<td></td>
<td>232</td>
<td>5</td>
<td>pf</td>
<td>502</td>
</tr>
<tr>
<td>98</td>
<td>5</td>
<td>on</td>
<td></td>
<td>247</td>
<td>2</td>
<td>GET</td>
<td>DF, 06</td>
</tr>
<tr>
<td>99</td>
<td>5</td>
<td>end Get</td>
<td>AF, 01</td>
<td>249</td>
<td>4</td>
<td>on</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>Put</td>
<td>AF, 04</td>
<td>254</td>
<td>1</td>
<td>on</td>
<td></td>
</tr>
<tr>
<td>105</td>
<td>3</td>
<td>on</td>
<td></td>
<td>258</td>
<td>4</td>
<td>on</td>
<td></td>
</tr>
<tr>
<td>109</td>
<td>3</td>
<td>Sleep</td>
<td></td>
<td>260</td>
<td>5</td>
<td>on</td>
<td></td>
</tr>
<tr>
<td>110</td>
<td></td>
<td>Create</td>
<td>2</td>
<td>264</td>
<td>5</td>
<td>pf</td>
<td>503</td>
</tr>
<tr>
<td>112</td>
<td>1</td>
<td>Open</td>
<td>KF</td>
<td>267</td>
<td>4</td>
<td>on</td>
<td></td>
</tr>
<tr>
<td>113</td>
<td>1</td>
<td>Open</td>
<td>F0F1</td>
<td>266</td>
<td>4</td>
<td>GM1</td>
<td>CF, 07</td>
</tr>
<tr>
<td>114</td>
<td>2</td>
<td>on</td>
<td></td>
<td>284</td>
<td>1</td>
<td>on</td>
<td></td>
</tr>
<tr>
<td>115</td>
<td>2</td>
<td>pf</td>
<td>201</td>
<td>285</td>
<td>1</td>
<td>- idle</td>
<td></td>
</tr>
<tr>
<td>140</td>
<td>2</td>
<td>on</td>
<td></td>
<td>290</td>
<td>5</td>
<td>on</td>
<td></td>
</tr>
<tr>
<td>141</td>
<td>2</td>
<td>pf</td>
<td>202</td>
<td>297</td>
<td>5</td>
<td>GET</td>
<td>AF, 06</td>
</tr>
<tr>
<td>Time</td>
<td>Task</td>
<td>Event</td>
<td>Data</td>
<td>Time</td>
<td>Task</td>
<td>Event</td>
<td>Data</td>
</tr>
<tr>
<td>------</td>
<td>------</td>
<td>-------</td>
<td>--------</td>
<td>------</td>
<td>------</td>
<td>-------</td>
<td>--------</td>
</tr>
<tr>
<td>299</td>
<td>5</td>
<td>exit</td>
<td></td>
<td>470</td>
<td>4</td>
<td>on</td>
<td></td>
</tr>
<tr>
<td>300</td>
<td>6</td>
<td>Sleep</td>
<td></td>
<td>471</td>
<td>4</td>
<td>end Put</td>
<td>CF, 07</td>
</tr>
<tr>
<td>301</td>
<td>2</td>
<td>on</td>
<td>DF, 06</td>
<td>500</td>
<td>1</td>
<td>on</td>
<td></td>
</tr>
<tr>
<td>315</td>
<td>1</td>
<td>end Get</td>
<td>KF, BEG</td>
<td>501</td>
<td>1</td>
<td>End Page Load</td>
<td>KF, BEG, 02</td>
</tr>
<tr>
<td>316</td>
<td>1</td>
<td>end Vam Get</td>
<td>KF, BEG</td>
<td>502</td>
<td>1</td>
<td>Page Load</td>
<td>KF, SUB, 01</td>
</tr>
<tr>
<td>317</td>
<td>1</td>
<td>Page Load</td>
<td>KF, BEG, 01</td>
<td>552</td>
<td>1</td>
<td>on</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>- idle</td>
<td></td>
<td>582</td>
<td>1</td>
<td>end Page Load</td>
<td>KF, SUB, 01</td>
</tr>
<tr>
<td>327</td>
<td>2</td>
<td>pf</td>
<td>204</td>
<td>583</td>
<td>1</td>
<td>Sleep</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>- idle</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>330</td>
<td>4</td>
<td>on</td>
<td></td>
<td>584</td>
<td>1</td>
<td>pg-de</td>
<td>101</td>
</tr>
<tr>
<td>333</td>
<td>4</td>
<td>end Get</td>
<td>CF, 07</td>
<td>585</td>
<td>1</td>
<td>pg-de</td>
<td>102</td>
</tr>
<tr>
<td>337</td>
<td>4</td>
<td>pf</td>
<td>403</td>
<td></td>
<td></td>
<td>pg-de</td>
<td>103</td>
</tr>
<tr>
<td></td>
<td></td>
<td>- idle</td>
<td></td>
<td></td>
<td></td>
<td>pg-de</td>
<td>104</td>
</tr>
<tr>
<td>350</td>
<td>2</td>
<td>on</td>
<td></td>
<td></td>
<td></td>
<td>pg-de</td>
<td>105</td>
</tr>
<tr>
<td>353</td>
<td>2</td>
<td>pf</td>
<td>205</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>- idle</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>360</td>
<td>1</td>
<td>on</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>361</td>
<td>1</td>
<td>pf</td>
<td>105</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>- idle</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>365</td>
<td>4</td>
<td>on</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>380</td>
<td>1</td>
<td>on</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>382</td>
<td>1</td>
<td>Vam Get</td>
<td>KF, SUB</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>383</td>
<td>4</td>
<td>on</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>390</td>
<td>4</td>
<td>Put</td>
<td>CF, 07</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>405</td>
<td>2</td>
<td>on</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>408</td>
<td>2</td>
<td>Get</td>
<td>DF, 07</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>409</td>
<td>2</td>
<td>end Get</td>
<td>DF, 07</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>410</td>
<td>2</td>
<td>Post</td>
<td></td>
<td></td>
<td></td>
<td>1, 10</td>
<td></td>
</tr>
<tr>
<td>410</td>
<td>2</td>
<td>Destroy</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>2 pg-de</td>
<td>201</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>2 pg-de</td>
<td>202</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>2 pg-de</td>
<td>203</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>2 pg-de</td>
<td>204</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>2 pg-de</td>
<td>205</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>- idle</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>440</td>
<td>1</td>
<td>on</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>441</td>
<td>1</td>
<td>end Get</td>
<td>KF, SUB</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>442</td>
<td>1</td>
<td>end Vam Get</td>
<td>KF, SUB</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>442</td>
<td>1</td>
<td>end Load</td>
<td>KF, SUB</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>443</td>
<td>1</td>
<td>end Page Load</td>
<td>KF, BEG, 01</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>443</td>
<td>1</td>
<td>Task Wait</td>
<td></td>
<td></td>
<td></td>
<td>2, 10</td>
<td></td>
</tr>
<tr>
<td>445</td>
<td>1</td>
<td>Page Load</td>
<td>KF, BEG, 02</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>446</td>
<td></td>
<td>- idle</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
5-6), is subsumed by the 'Page Load' event at the same time in the SRET.

Preprocessor of the SRET

The SRET preprocessor is similar to the SLET preprocessor. The differences between them are caused by the difference in their inputs - the SRET contains relocation events not present in the SLET, and the latter contains the physical events not present in the former. The SRET preprocessor will be presented by demonstrating the differences between it and the previously defined preprocessor.

The events that are peculiar to the SRET are handled in the same way that the logical events are handled by the preprocessor of the logical event trace. The 'load' and 'page load' events for task $k$ cause an increase in the counter $d_k$, and the 'end load' and 'end page load' events cause this counter to be decremented. No event - whether of the physical, logical, or relocation event trace - is placed on task $k$'s event trace while the counter $d_k$ has a non-zero value.

The absence of the physical I/O events in this trace causes a more significant problem. These events provide an indication of I/O and paging activity, from which memory interference with processing can be inferred. Without them, it cannot be determined to what degree the processing time for a task has been expanded because of the effects of memory interference.

An approximation to the memory interference factor - derived from the statistically known usage of I/O operations in the logical I/O macro - is used to replace the exact interference factors that were used in the previous trace. The approximation is performed as follows: Let $b_i$ be the average number of bytes transmitted to or from memory during the LI/O operation $i$. Let $t_i$ be the average length of time of that operation.
Then $b_1/t_1$ is the average number of memory cycles per unit time taken by this logical I/O operation. If $m$ is the memory speed, and $c$ is the fraction of memory cycles needed by the processor, then processing will be slowed by a factor of $f = 1 - c + \frac{cm}{m - b_1/t_1}$ during the period that this LI/O is in operation.

The preprocessor calculates a new memory load factor $f$ for each event that is encountered that marks the beginning of a system routine, and maintains it until the 'end' event is encountered. The memory load factor is used in $U(n)$ to determine the amount of task processing time in an interval. The load factor is applied only at the first level of recursiveness of the system routines, since recursive calls on system routines do not increase the amount of I/O operations per unit time in the routine. Therefore, the memory load is increased only when $d_k = 0$ before the 'begin' event, and it is decreased only when $d_k = 1$ before the 'end' event.

Example of the TRET

The Task Relocation Event Trace is shown in Table 5-9 for Task 1 of the example. Comparing this task trace with the TLET, (Table 5-7) it is seen that the entire task processing time has been reduced from 69 to 42 time units. The 17 time units were spent in processing within the Load and Page Load macros. The task demand for this processing time, as well as the demand for I/O and paging operations within this processing time, is represented by the Load and Page Load events at times 6, 7, 11, and 12 in the task trace. The first Load operation for the subroutine BEG, resulted in a call on the Yam Get macro to begin the loading of the program.
Table 5-9

Task Relocation Event Trace for Task 1

<table>
<thead>
<tr>
<th>Time</th>
<th>Event</th>
<th>Data</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>Wake</td>
<td>093040</td>
</tr>
<tr>
<td>1</td>
<td>pf</td>
<td>101</td>
</tr>
<tr>
<td>2</td>
<td>pf</td>
<td>102</td>
</tr>
<tr>
<td>3</td>
<td>pf</td>
<td>103</td>
</tr>
<tr>
<td>4</td>
<td>pf</td>
<td>104</td>
</tr>
<tr>
<td>5</td>
<td>Create</td>
<td>2</td>
</tr>
<tr>
<td>6</td>
<td>Open</td>
<td>KF</td>
</tr>
<tr>
<td>6</td>
<td>Load</td>
<td>KF, BEG</td>
</tr>
<tr>
<td>7</td>
<td>Page Load</td>
<td>KF, BEG, 01</td>
</tr>
<tr>
<td>8</td>
<td>Task Wait</td>
<td>2, 10</td>
</tr>
<tr>
<td>11</td>
<td>Page Load</td>
<td>KF, BEG, 02</td>
</tr>
<tr>
<td>12</td>
<td>Page Load</td>
<td>KF, SUB, 01</td>
</tr>
<tr>
<td>42</td>
<td>Sleep</td>
<td></td>
</tr>
<tr>
<td>42</td>
<td>pg-de</td>
<td>101</td>
</tr>
<tr>
<td>42</td>
<td>pg-de</td>
<td>102</td>
</tr>
<tr>
<td>42</td>
<td>pg-de</td>
<td>103</td>
</tr>
<tr>
<td>42</td>
<td>pg-de</td>
<td>104</td>
</tr>
<tr>
<td>42</td>
<td>pg-de</td>
<td>105</td>
</tr>
</tbody>
</table>
When a page of the subroutine BEG is referenced immediately after the call, a page fault for the page takes place, but, because the page is not in secondary storage, the page fault is handled by the Page Load operation. The Page Loader finds that the definition of the externally defined symbol SUB is required, and it calls on the Load operation to load the subroutine SUB before completing the loading of the BEG page. Neither the I/O operations called as a consequence of the Load, nor the Load called as a consequence of the Page Load, are shown in the TLET.
CHAPTER 6
THE DESIGN OF THE SIMULATOR

In this chapter, the simulator for a model of a variant version of the system will be developed. The simulation model will be driven by the representation of user demand that is contained in the previously developed task traces. Several different levels of simulators will be described, corresponding to the several levels of demand measurement that were presented. Unlike the levels of event trace recording, however, in which an attempt was made to eliminate the events of a lower level from the definition of a higher level, each level of simulation will be an extension of the model of the lower levels. The event recording will then take place at one level of system description; the simulator will provide remaining levels of description down to the hardware level.

This simulator could be implemented in one of several simulation languages that allow the user to provide his own input events, rather than have the simulator generate them. This description of the simulator will not focus on one of these simulation languages, nor treat the details of the interface between the processor of that language and the task event traces. Rather, the simulator will be presented in general terms, as an algorithm. The effect of the task traces as the input to the simulator will then be clear.

The Simulator

Four separate event traces for measurement of four levels of the system operation have been presented. The task traces resulting from analysis of three of those levels will be used as input to the simulator.
The event trace taken from the first level - the hardware event trace - is not useful for this purpose, since, rather than being a representation of task demand, it is a representation of the system hardware response to task demand. The simulators for the other three levels consist of one common basic part, which is the simulator of the second level - the physical I/O level. This simulator is extended to allow operation with the event traces of the two higher levels by including within it routines that simulate the higher level software functions.

The basic, PI/O level simulator will be presented in the following description.

Input to the Simulator

The input to the simulator is a set of task event traces, recorded as a system event trace on the actual system and separated by an application of the preprocessor. When the simulation model is run, the event traces must be read into a random-access storage unit so that the simulator may scan the traces in both directions in task time.

In an implementation, the simulator may also have several parameters of its operation read in as input. Here, these parameters are presented as part of the structure of the model, but they should be considered variable from run to run of the model.

Output of the Simulator

The primary factor that the simulator will measure is the response time of each interaction of the various tasks. The response time is the time from the completion of the input for a particular task (the 'awake' event) to the beginning of the wait for the terminal response (the 'sleep' event), which is synonymous with the beginning of the output to the
terminal.

The response time is recorded by writing the simulator time of every 'sleep' and 'awake' event for each task in an output area. From this point, it is a straightforward operation for a postprocessor to condense this data to obtain the means and the variances of the response times for each class of tasks.

The response time measurement is an indirect measurement - and the most significant measurement - of the utilization of each of the system components. If different versions of the simulator (or the actual system) are presented with identical representations of a fixed number of tasks, the throughputs of the different system versions will, in general, be different. That is, some versions will complete processing the entire load before others. The 'think time' part of each interaction is constant, therefore, the throughput time difference must be due to different response times in the various systems. The response time characteristic under constant load is an indication of system throughput as well as a measure of time-sharing system quality in its own right.

To effect design changes, it may be necessary to understand why throughput becomes better or worse, by observing the utilization of various parts of the system. These utilizations can be measured in the same way that utilizations are measured from the hardware event trace, since the simulator generates all the events of the hardware event trace. This use of the simulation model should be obvious, and will not be demonstrated here.
The Basic Simulator - Overview

The simulator of the time-sharing system is presented in four major parts. These are:

a) The Clockworks: the basic executive part of the simulator.
b) The Event Analysis routine: an extension of the model of the Interrupt Analyzer.
c) The hardware section: a model of the system hardware.
d) The event response routines: generally, models of the interrupt response routines.

The outline of this simulator is presented pictorially in Figure 6-1.

The Clockworks

The Clockworks (CW) is that part of the simulator which selects the next event that the model is to process, and passes it on to the model of the operating system.

CW finds the next event to be processed in one of the following two places:

a) The task traces that make up the input to the simulator. The \( i^{th} \) task trace will be called \( T_i \).
b) The Future Event Trace (FET). The events of the FET are events representing the system response to task-generated events. These events are placed on the FET by the model of the hardware.

The important simulator data maintained by CW are:

a) A location \( t \) which represents simulator time.
b) A location \( p \) indicating the task currently on the processor.
Figure 6-1

Outline of Simulator
The CW also maintains some data representing the state of each task throughout the simulated time. For the $k^{th}$ task trace, this data consists of:

a) a counter $j_k$, indicating the next event associated with the task $k$. This event, on $T_k$, consists of:

- $t_{j_k}$: the task time of the event
- $e_{j_k}$: an identification of the event
- $q_{j_k}$: the parameters of the event

b) a clock value $t_k$, which is the task time for task $k$ corresponding to the simulated real time $t$. If task $k$ is on the processor, and the clock $t$ is advanced by $n$ units, the task time $t_k$ is advanced by $n/f$, where $f$ is the interference load on the memory during this period.

The clockworks selects which event - of the next event on the Future Event Trace or the next event on the trace for task $p$ - will occur next in simulated time. The times on the FET are simulated (real) time. The times on the task traces are task times. In order to compare them to determine which will occur sooner in real time, CW must convert the task time to real time. (Figure 6-2) The location $t_n$ is calculated to be the real time of the next task event.

After selecting the next event, CW passes the event to MA. If the first event on the FET (which has index $t_1$) is taken, the locations $(n,e,q)$ take their values from $(n_{t_1}, e_{t_1}, q_{t_1})$. If not, they take their values from the trace of the active task $(p, e_p, q_p)$.
Calculate time of next event generated by task on CP:
\[(t_p - t_{ip}) \cdot f \rightarrow t_n\]

\[t_n < t_1\]

**Y**
- Take event from task trace \(p\)
  - Update proc time
    \[t_p + \left(\frac{t_n - t}{f}\right) \rightarrow t_p\]
  - Update real time
    \[t_n \rightarrow t\]
  - \(p \rightarrow k\)
  - \(e_{ip} \rightarrow e\)
  - \(q_{ip} \rightarrow q\)

**N**
- Take event from FET
  - Update proc time
    \[t_p + \left(\frac{t_{fl} - t}{f}\right) \rightarrow t_p\]
  - Update real time
    \[t_{fl} \rightarrow t\]
  - \(k_{fl} \rightarrow k\)
  - \(e_{fl} \rightarrow e\)
  - \(q_{fl} \rightarrow q\)

Figure 6-2
The Clockworks
The Event Analyzer

The model of the software includes a model of the Interrupt Analyzer. The model of this routine will receive the events selected by the Clockworks, rather than interrupts provided by the hardware. The set of events which it will receive is not in one-to-one correspondence with the set of interrupts that would occur in the system being simulated; therefore this part of the software model is called the Event Analyzer (EA), not the Interrupt Analyzer.

The EA simply transfers to the particular software model that handles the event.

In most cases, the routines will be models of the interrupt response routines. The EA must handle hardware events that have been placed on the FET by the hardware model. Models of the parts of the system identified as 'resident' must be included in the simulator of every level. Thus, a model of each of the following routines, as well as the Interrupt Analyzer, must be included in the software model:

- task initiator
- paging routine
- replacement algorithm
- page channel complete routine

These routines are some of the 'event response' software models. They will be presented after the hardware models.

The EA must also simulate the paging interrupts that take place when the required system routines are not resident.
The model contains a representation of the allocation of the pageable system virtual memory in the system's page table. This is a mapping from virtual address to either the main memory address or secondary storage address for the system's virtual memory. The Event Analyzer references the page table to find the virtual address of the simulated system routine, then checks the page table to determine whether this page is resident in core. If it is, the EA simply transfers to the proper simulated interrupt response routine.

If the page is not resident, the EA must generate a page fault for this routine. It will do this by simply transferring to the page fault event response routine, with the parameters of the task number and secondary storage address of the required page. The event that caused this page fault, however, must not be lost. Therefore, before the transfer, the EA reschedules the event by placing it as the first event on \( T_k \). (The event may actually have just been taken off \( T_k \); if so, all that is necessary is for the pointer \( i_k \) to be decremented.)

The EA operation is shown in Figure 6-3.

**The Hardware Model**

The model of the hardware is invoked in the simulator, at each place that instructions to the hardware devices are executed in the system being modeled. These are:

1) When the Task Initiator places a task on the CP, or when the Interrupt Analysis routine takes a task off the CP.

2) When the Task Initiator sets the Interval Timer

3) When the I/O initiation routine sends command to a device.
Figure 6-3
Event Analysis Routine

Event Analysis:
Inputs \((k, e, q)\)

Find address of event \(e\) response routine on branch table. Find event on SYS base table.

Is the E.R. routine marked resident?

\(\overline{\text{Y}}\) To Event Response

\(\overline{\text{N}}\) Place event on \(t_k\) at time \(t_k\)

\(\overline{\text{N}}\) To Base Fault Routine
In the first case, the process of placing a task on the CP is one of loading the CP registers and address translation hardware. This involves some overhead, which is counted as execution time in the trace of the task being placed on the processor. To the model, this process involves only the setting of the location p, to indicate the task on the processor.

When the Task Initiator sets a new value d into the interval timer, it indicates an interrupt will occur at an interval d later.

Thus, the model of the Task Initiator will place an 'interval timer runout' event on the FET at time t+d, at the point in which the actual timer is set.

The timer runout event may not occur, however, if the timer is reset before the interrupt takes place. Thus, the routine corresponding to timer reset instruction will find the 'timer runout' interrupt on the FET, and remove it. The time t+d is placed in a location y, so that the interrupt may be found, if necessary.

The channels and I/O devices in actual systems respond to both sense and activate instructions. Therefore, the input to the model of the channels and I/O devices will be simulated sense and activate instructions generated by the simulated systems routines. The model of the I/O hardware will be made explicit by an example of one typical external random access device - the disk unit. The disk unit is close to being a generalization of several random access devices -- the drum, and the mass storage unit.
Model of random access device operation

The read and write hardware instruction to the random-access unit contains the following parameters.

1) an identification of the operation to be performed
2) an identification of the unit
3) an identification of the location on the unit to which the operation is addressed.

It will be assumed that the device is part of a page-oriented system. An entire track -- which is equivalent to one page -- is to be read or written for each I/O operation. Therefore, the key field indicating the record on the track is not required.

The model will use the following parameters, which represent the static operational characteristics of the device.

1) The amount of time taken for one page transfer. This is contained in the location \( l \).

2) The distribution of the time required for disk arm movements across cylinders. This is given by the function \( S(k) \), where \( k \) is the number of cylinders to be crossed.

3) An identification of the set of channels to which the device is attached -- the path set for the device. This configuration-specifying parameter, labeled \( S \), identifies the software-maintained tables corresponding to the path set.

4) The transfer rate of the device, in bytes/second. This parameter is \( v \).
Device operation will be modeled by modifying the state of the device. The state of the device is represented by:

1) the cylinder position of the arm. This is the variable $a$.
2) the angular position of the track marker, with respect to the position of the read heads. This will be kept by a location containing the time the last data transfer operation began. This is location $t_r$.
3) Indicators marking the particular operation the device is performing. These are the following busy indicators:

   $b_k = 1$ when the device is in seek, 0 if not
   $b_t = 1$ when the device is in latency, or is transferring data, 0 if not.

The device is idle when both these indicators are zero.

The routines modeling the commands to the I/O device are shown in Figure 6-4. The sense routine is a model of the instruction

'transfer if busy' u, b

where u is an identification of the device, and b the location of the conditional transfer. The subroutine merely checks the busy bits and transfers if either is set. (Figure 6-4a)

The hardware response to the Seek command is the 'channel complete' interrupt at the future time determined by the function $S(k)$. The number of tracks to be crossed is the absolute difference between the device state $a$, and the specified cylinder, $c$. The hardware model therefore places the 'channel complete' event on the PET, at time $t + S(|a - c|)$, and sets the busy indicator for the device. (Figure 6-4b)
Sense device:
inputs: device u
location b (routine for case
of device busy)

\[ b_{ku} \lor b_{tu} = 1? \]

Y

b

N

Return

a. Sense instruction

Seek:
inputs: task k
device u
cylinder c

Calculate time for end of seek:
\[ t_e = t + s_{ku}(l_u - c_l) \]

Put seek terminate event on FET
time = \( t_e \)
event = device terminate
task = k
data = u, seek

\[ b_{ku} = 1 \]

Return

b. Seek instruction

Figure 6-h
Subroutines for Simulating Hardware Instructions
The model responds to the search command by scheduling the 'search complete' event. This is done by determining when the next track mark will appear in the device operation, and placing the termination event on the FET, one revolution time \( r \) later. A new event, the 'data transfer begin' event is placed on the FET at the time the track mark first appears. The response to this new event will be to increase the memory load factor \( f \), for the data transfer interference caused by this device. (Figure 6-4c)

The Models of the Event Response Routines

With this outline of the simulation model, the simulation of the software routines that respond to the interrupts in the real system is a relatively straightforward affair. In most cases, an event response routine will look very much like a routine whose operation is being simulated. If some modification of the actual routine - from which the traces were taken - is to be evaluated, the simulation routine will, of course, be a model of that modification rather than the actual routine. Aside from this, the significant differences between the actual routines and the routines of the model are the following:

1) Whenever the actual system sends a hardware instruction to a device, the previously described hardware routines are called.

2) Some system-generated events, such as processor "on" and "off", were purged from the task traces, because it was known that they were totally dependent upon the task scheduler's decisions. Other events, namely page faults and events representing memory management, are dependent upon the tasks' requirements.
Figure 6-4c

Search Operation

1. **Search inputs task k channel h device u**

2. **Calculate** the number of rotations since the last track mark:
   
   \[ n = \left\lfloor \frac{t-\text{tr}}{r} \right\rfloor \]

3. **Place beginning of data 'transfer' event on FET** at time \( t+n_r \)

4. **Calculate time for end of each operation**:
   
   \[ t_s = t + (n+1) r \]

5. **Place 'end of search' event on FET**
   
   - Time: \( t_s \)
   - Event: \( \text{ch complete} \)
   - Task: \( k \)
   - Data: \( h \)

6. **Increase load factor I**

7. **Return**
as well as the system. Therefore, the simulation model must decide—should this event have occurred in this configuration being modeled?

Replacements of Hardware Instructions

Figure 6-5 is an example of the model of a particular operating system routine—the Physical I/O Initiate macro.

This simulated macro references the queues that the scheduler maintains (queue for devices, queue for path set). These queues are kept by the software section of the model. The commands to the hardware (sense, seek) are converted into calls on the hardware model.

Main Memory Allocation by the Model

Memory allocation must be given special consideration because the page fault events and the task trace do not represent the total memory demand. To the simulator, the significance of a page fault event in the task trace is simply that the page was required at that particular instance in task time; it is not known how long afterwards the page is needed by the task.

Upon encountering a page fault event in the task trace, the model merely checks the page table of the task to see if that interrupt would have happened in the simulated operation. If the page is marked 'not resident' the simulated page fault routine is called. If the page is marked 'resident', the page fault would not have occurred in the simulated system; it is simply skipped.

Therefore, in the cases when the model is maintaining pages in main memory during the task times that they were not in main memory in the actual system operation, the model does not face any ambiguity.
Event: PI/O req
    task = k
    data: u, c, r

Sense u, q

Is $S_u$ busy?

Seek: k, u, c

Y

Is an operation to device u on $S_u$ queue?

N

Queue operation for $S_u$

Queue operation for $S_u$

N

to CW

Queue operation for $S_u$

Queue operation for $S_u$

Y

Queue operation for $S_u$

Figure 6-5

Software Model of PI/O Request Macro
However, since the model manages main memory and virtual memory itself, it is entirely possible that it will de-allocate a page or a set of pages for a task, during task times in which these pages had been resident in the actual system operation. The consequences of such de-allocation is that a page fault for this page might now occur in the simulated operation, where none occurred on the real system. As such a page de-allocation occurs, the model must place the new page fault interrupt into the task trace. It is not placed in the Future Event Trace, because its position is fixed only in task time. The position of its occurrence in simulated time depends upon the progress of the task.

At the time that the model-generated page de-allocation takes place in the simulation, it must be estimated whether the page will be required again, and the most probable time for the re-occurrence of the demand for that page must be calculated. The only information that can be used to make this estimate is the occurrence of other instances of de-allocation and page faults for that page in the task trace.

The selection of the demand characteristics of a particular page at a particular point of simulated time must be made from a distribution of a random variable, as would be done in a pure simulation. The parameter of the distribution is taken from the incomplete available information.

Let \( T_n \) be the random variable specifying the time to the next use of an arbitrary page \( n \) of a task's working set of pages, after \( n \) is examined at \( t_0 \). The next use will more probably occur in an interval closer to \( t_0 \) than in interval further away. \( T_n \) may take on values for the entire length of the task, which is potentially very long.
Without specifying any further information about $T_n$, it must be assumed that the probability distribution for $T_n$ is exponential distribution.

$$P(T_n < (t - t_o)) = \begin{cases} 0 & t < t_o \\ 1 - e^{-\beta(t - t_o)} & t \geq t_o \end{cases}$$

The value of $\beta$, which is the mean value of $T_n$, is taken from the average of the page re-use times that are available on the task trace. One can take advantage of the similarity between the model operation and the task trace, by weighing this average in favor of the page re-use time that is closer in task time to the event under consideration.

Suppose a model-generated page de-allocation of page a takes place in the model at task time $t_o$. At task times $t_1 < t_o$ and $t_3 > t_o$, the task trace has page de-allocate events for the page. At times $t_2$ and $t_4$, the task trace contains page fault interrupts for the recovery of page a. The page re-use times $\delta_1 = t_2 - t_1$ and $\delta_2 = t_3 - t_4$ are available in the trace (see Figure 6-6).

![Figure 6-6](image)

Figure 6-6

Page Re-Use Times in Task Trace
The expected re-use time after the de-allocation of $t_o$ is:

$$\delta_o = \frac{t_3 - t_o}{t_3 - t_1} \delta_1 + \frac{t_o - t_1}{t_3 - t_1} \delta_2$$

This average is weighed in favor of the closest re-allocation time. In fact, if $t_o = t_3^-$, then $\delta_o = \delta_2$. This indicates that the model and the real system are managing memory the same way; page faults will take place at the same task time.

The model places a simulator-generated page fault for page $a$ on the task trace, at task time $t_o + \delta_o$.

The page-de-allocate routine, called by the model of the Replacement Algorithm, is shown in Figure 6-7.
Figure 6-7
Modification of Memory Demand by the Simulator
CHAPTER 7
EXAMPLES OF META-SYSTEM OPERATION

The Meta-system operation has been tested at each of the Meta-system levels of awareness of system operation, and under several different load conditions. In this chapter the results of several of these Meta-system tests are presented as examples of the capabilities of the Meta-system.

Extending the techniques presented in this report leads to the conclusion that any potential system modification may be evaluated by a Meta-system, if the Meta-system level is properly specified. The Meta-system must be constructed at the level of system awareness that encompasses all parts of the system that are affected by the modification. Only a few of this all-encompassing set of potential modifications may be given as examples, and only the explicitly developed Meta-systems are used in the examples.

Each of the developed Meta-system levels is used in one or more of the examples. In each example, the same instance of system operation is used to record the event trace. Therefore, the representation of user task demand that is input to the simulation model in each case, derives from the same set of user tasks, and the results of the simulation runs are comparable.
Method of Running Examples

The method of running the tests of the Meta-system operation is as follows. First, an arbitrary set of tasks, performing an arbitrary set of operations was selected. The flow of the set of tasks through the system routines (in Appendix A) was traced, by hand, and an event recorded each time an event recording mechanism was encountered. (In the examples given here, that set of tasks is the same as that used in Chapter 5, and the resulting event traces can therefore be found in Chapter 5.) The operation of the preprocessor was then followed in the same way, and the system event traces were decomposed into event traces of the five user tasks. Last, these five task traces were used as input to the system simulation model at the proper level, and the model's operation traced until the simulator clock reached 300. This time was chosen as an arbitrary cut-off time because soon after this time in the actual system, the tasks become depleted and the system is too lightly loaded for any results to be meaningful.

During the simulation run in which one of the features of the system is modified, the utilization of the hardware devices was recorded. The other parameter of interest in evaluation - response time - was not available in the simulation run, because the response times of most interactions was greater than 300 time units. However, a measure of the
relative increase or decrease in response time was obtained by comparing the progress of each task in the simulation interval, with its corresponding progress in the real system.

It will be assumed that at the beginning of each simulation run, each task is at the same point in its operation as it was in the real system. This assumption is necessary, because nothing else can be known in the simulation model about the state of the system at the start of the simulation run. The assumption does cause some inaccuracy in the simulation run, however. When significant results are to be obtained, the simulation must be run long enough to overcome the inaccuracy.

In each of the examples of simulation runs to evaluate system modifications, the following results will be obtained:

Utilization of the Processor
Utilization of the paging device
Utilization of the I/O channels
Response time
Percentage decrease (or increase) in the response time for each task.

The results will be compared with the corresponding figures taken from the actual system operation. The response times will be reported as percentage decrease (or increase) in the response time:
that were obtained in the original systems operation.

Results From the Actual System

The time-sharing system presented in Chapter 4 is assumed to be in operation, hence, it is called the 'actual' system. Its operation is traced, by hand, in order to generate event traces to become input to the simulator, and in order to obtain numerical data to be compared with the same data obtained from the operation of the simulated system. The instance of the actual systems operation that was used to generate the sample event traces presented throughout Chapter 5 is used in this example of the entire systems operation, as well.

A part of the time-sharing system - the specific scheduling discipline used to select tasks for the CP or the paging device - was not well specified in Chapter 4. This is because it is an area that can be left to the user to specify for his particular task load. (The disciplines are represented by the decision 'select task for CP' in the Task Initiator (Figure A-3) and 'select a paging operation' (Figure A-6)).

In the sample case, the scheduling disciplines for both the processor and the paging device divide the tasks into two priority groups. Tasks 1 and 5 are in the high-priority group, and the others in the low-priority group. The high-priority group was served before the low-priority group. Within a group, tasks were selected on a
FCTS basis. In addition, the processor was scheduled on a preemptive basis. The paging device will perform write operations when no demand page-in operations are on queue for the device.

The results from the system operations under this discipline are obtained by scanning the Hardware Event Trace, and summing the utilization periods. The results are the following:

<table>
<thead>
<tr>
<th>Time on</th>
<th>Total</th>
<th>Pct.</th>
</tr>
</thead>
<tbody>
<tr>
<td>Processor utilization</td>
<td>134</td>
<td>300</td>
</tr>
<tr>
<td>Paging device</td>
<td>230</td>
<td>300</td>
</tr>
<tr>
<td>- read</td>
<td>15</td>
<td>300</td>
</tr>
<tr>
<td>- write</td>
<td>245</td>
<td>300</td>
</tr>
<tr>
<td>I/O Channel</td>
<td>172</td>
<td>600</td>
</tr>
</tbody>
</table>

Table 7-1

Utilization Figures for the Actual System

The system seems to be paging-device-bound. The Meta-system will be used to find a solution to this condition.

Investigation of Hardware Device Change Using the Physical Event Trace

Bottlenecks in a computer system are not always easy to locate. One way to find a bottleneck is to change the speed of one device at a time, and observe the effects on the utilization of each of the components in the system, and the overall effect of the change in response time.
Because the real system seemed to be heavily engaged in paging in this case, an increase in the speed of the paging device will be investigated. For the first example, the time of a paging operation will be decreased to 12 time units/page in the model, from the original to 15 units/page in the actual system. Only the physical event trace is needed to evaluate the effects of this change since the physical-level demand for resources will not change.

The results of this simulation run, as compared to the actual system statistics, are shown in Table 7-2.

<table>
<thead>
<tr>
<th></th>
<th>Actual</th>
<th>Simulated</th>
</tr>
</thead>
<tbody>
<tr>
<td>Processor Utilization</td>
<td>44.6</td>
<td>51.3%</td>
</tr>
<tr>
<td>Paging device</td>
<td>81.3</td>
<td>64.0%</td>
</tr>
<tr>
<td>I/O channel</td>
<td>28.6</td>
<td>29.4%</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Response Time Indicators</th>
<th>Actual</th>
<th>Simulated</th>
<th>% Improvement</th>
</tr>
</thead>
<tbody>
<tr>
<td>Task 1</td>
<td>320</td>
<td>300</td>
<td>6.25</td>
</tr>
<tr>
<td>2</td>
<td>217</td>
<td>210</td>
<td>3.1</td>
</tr>
<tr>
<td>3</td>
<td>98</td>
<td>104</td>
<td>-6.1</td>
</tr>
<tr>
<td>4</td>
<td>377</td>
<td>300</td>
<td>20.4</td>
</tr>
<tr>
<td>5</td>
<td>300</td>
<td>281</td>
<td>6.3</td>
</tr>
<tr>
<td>Total</td>
<td>1312</td>
<td>1195</td>
<td>8.9</td>
</tr>
</tbody>
</table>

Table 7-2

Results of Simulation Run with Fast Paging Device

The 20% improvement in paging device speed provided an added 6.7% processor utilization. The response times are improved by an average of 8.9%. The deterioration of response time of the short-duration task 3 is a fluctuation due to new scheduling conditions.
Investigation of Change of Scheduling Discipline Using the Physical Event Traces

The effects of various scheduling disciplines can be investigated by using the Meta-System at the physical level. In the second example, the simplest possible algorithm - a non-preemptive FCFS algorithm - is used to schedule both the CP and the paging device. The Task Physical Event Traces are placed into a system simulator that follows this scheduling discipline. The results are summarized in Table 7-3.

<table>
<thead>
<tr>
<th></th>
<th>Actual</th>
<th>Simulated</th>
</tr>
</thead>
<tbody>
<tr>
<td>Processor Utilization</td>
<td>44.6</td>
<td>39.3</td>
</tr>
<tr>
<td>Paging device</td>
<td>81.3</td>
<td>75.0</td>
</tr>
<tr>
<td>I/O channel</td>
<td>28.6</td>
<td>28.3</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Response Time Indicators</th>
<th>Actual</th>
<th>Simulated</th>
<th>% Improvement</th>
</tr>
</thead>
<tbody>
<tr>
<td>Task 1 Response Time</td>
<td>275</td>
<td>300</td>
<td>-9.1</td>
</tr>
<tr>
<td>Task 2</td>
<td>217</td>
<td>212</td>
<td>2.3</td>
</tr>
<tr>
<td>Task 3</td>
<td>98</td>
<td>98</td>
<td>0.0</td>
</tr>
<tr>
<td>Task 4</td>
<td>305</td>
<td>300</td>
<td>1.2</td>
</tr>
<tr>
<td>Task 5</td>
<td>270</td>
<td>300</td>
<td>-11.1</td>
</tr>
<tr>
<td>Total</td>
<td>1165</td>
<td>1210</td>
<td>-3.9</td>
</tr>
</tbody>
</table>

Table 7-3

Simulation Run with FCFS Scheduling

As might be expected, the simpler scheduling algorithm caused a deterioration in system performance, although it allowed a marginal improvement in the response time of tasks 2, 3, and 4 (which the previous scheduler had discriminated against). The decrease in the utilization of the processor and CP in the simulation run is an indication that the choice of tasks 1 and 5 for the high-priority group was quite judicious.
Evaluation of Alternate Logical I/O Routines Using the Logical Level Meta-System

An alternate version of the basic logical I/O routines Open and Get are shown in Figure 7-1 and 7-2. This new Open routine brings the file directory as the file definition into virtual memory as well. The Get macro expects to find the directory in virtual memory and performs a physical I/O operation to obtain only the desired record.

The specifications for resource demand at the logical level are valid to evaluate the effects of using these macros in the system. The Task Logical Event Traces were used in a simulation containing the macros. The results are shown in Table 7-4.

The changes in the utilization of the hardware devices are insignificant, because in this small amount of simulation time, only one instance of the Open and Get macros occurred, and the total number of physical I/O operations that resulted from them was the same as in the original case.

This change in the logical I/O macros would have a greater effect when the Open and Get operations are separated by some processing time, and the page containing the file directory is deallocated in this interval. However, it seems the new algorithm will probably be less favorable than the original in this case.

The possibility of using either the original or the new logical I/O macros exists, with the choice being determined dynamically, at the time the Open macro is called. The decision would be based on the immediate demand for the processor and I/O devices. If the I/O load is relatively heavy and the processor demand relatively light, the original macro is
used. This set of logical I/O routines would have a greater chance of resulting in improved overall utilization. Their effectiveness could again be evaluated in the logical-level Meta-System.

<table>
<thead>
<tr>
<th></th>
<th>Actual</th>
<th>Simulated</th>
</tr>
</thead>
<tbody>
<tr>
<td>Processor Utilization</td>
<td>44.6</td>
<td>44.4</td>
</tr>
<tr>
<td>Paging device &quot;</td>
<td>81.3</td>
<td>81.3</td>
</tr>
<tr>
<td>I/O channel &quot;</td>
<td>28.6</td>
<td>28.0</td>
</tr>
<tr>
<td>Response Time Indicators</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Task 1</td>
<td>305</td>
<td>300</td>
</tr>
<tr>
<td>2</td>
<td>217</td>
<td>217</td>
</tr>
<tr>
<td>3</td>
<td>98</td>
<td>98</td>
</tr>
<tr>
<td>4</td>
<td>293</td>
<td>300</td>
</tr>
<tr>
<td>5</td>
<td>290</td>
<td>300</td>
</tr>
<tr>
<td>Total</td>
<td>1230</td>
<td>1215</td>
</tr>
</tbody>
</table>

Table 7-6
Simulation Run with New Logical I/O Routines

Evaluation of More Complex Scheduling Disciplines Using the Logical Level Meta-System

The scheduling of a task within a system is generally based on its accumulated utilization of resources and externally assigned priority. It is conceivable that scheduling could be done on the basis of the function the task is performing. For example, a task executing a Logical I/O routine is more likely to initiate a physical I/O operation, and give up the processor, than another task. Therefore, such a task should be given priority on the processor.
This scheduling philosophy was evaluated by using the logical level Meta-System. The simulated logical I/O routine assigned each task a higher priority as it entered the routine, and reset its priority as it exited from the routine. In this example, the higher priority task preempted the lower priority tasks on the processor. Scheduling within a priority group was FCFS. The paging device was scheduled FCFS. The results are shown in Table 7-5.

<table>
<thead>
<tr>
<th>Processor Utilization</th>
<th>Actual</th>
<th>Simulated</th>
</tr>
</thead>
<tbody>
<tr>
<td>Paging device</td>
<td>44.6</td>
<td>45.0%</td>
</tr>
<tr>
<td>I/O channel</td>
<td>81.3</td>
<td>81.3</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Response Time Indicators</th>
<th>Actual</th>
<th>Simulated</th>
<th>% Improvement</th>
</tr>
</thead>
<tbody>
<tr>
<td>Task 1</td>
<td>295</td>
<td>300</td>
<td>-1.6</td>
</tr>
<tr>
<td>Task 2</td>
<td>217</td>
<td>214</td>
<td>1.4</td>
</tr>
<tr>
<td>Task 3</td>
<td>98</td>
<td>89</td>
<td>11.3</td>
</tr>
<tr>
<td>Task 4</td>
<td>393</td>
<td>300</td>
<td>23.7</td>
</tr>
<tr>
<td>Task 5</td>
<td>291</td>
<td>300</td>
<td>-3.3</td>
</tr>
<tr>
<td>Total</td>
<td>1274</td>
<td>1203</td>
<td>5.6</td>
</tr>
</tbody>
</table>

Table 7-5
Simulation Run Using Complex Scheduling Discipline

It is seen that the response time of tasks 3 and 4 improves considerably, as they are now able to compete for system resources on an equal basis with tasks one and five, which had previously been the high-priority tasks. The response time of the latter tasks is slightly degraded as a result of this.
Evaluation of System Memory Management Using the Relocation Event Trace

Memory management has not been a factor in the scheduling tasks in these examples, since it was assumed that there were enough memory blocks available to satisfy the requirements of all the active tasks. In practice, however, physical memory will be limited, and memory allocation must be considered along with the scheduling of other system resources. Some of the memory allocation decisions will be made during the design of the operating system, and not all of these decisions will be based directly on the needs of the active tasks. For example, the specification of which parts of the operating system (tables and programs) will be kept resident and which will be pageable may be made without reference to the needs of the tasks.

In the last example of Meta-System operation, a memory management algorithm is evaluated. In the simulated operation, a table needed by the loader - the Extern Table - is assumed to be written out to the paging device after the first program load operation is completed. The result of this will be that another memory block will be available until the loader again requires the Extern Table.

The relocation event trace is needed to test this algorithm, since awareness of the use of the Extern Tables by the loader is required. The System Relocation Event Trace is decomposed by the preprocessor, and the resulting Task Event Traces are used as input to the simulator. For the interval of simulator operation in which the utilization figures were obtained in the previous cases (0 - 300 time units), the Extern Table was not used. Therefore the utilization figures are the
same as the original system. Continuing the simulation until task one
is completed shows the response time for task 1 worsened by 6%. (The
utilization figures were much worse, but this was due to the depletion
of the active tasks in the system.) A longer simulation run during a
high memory load period might show a compensating increase in processor
utilization due to relief of the paging load.
CHAPTER 8

RESULTS AND CONCLUSIONS

The primary result of this study is that the Meta-System - a system which extracts measurements from an operational time-sharing system and uses them in a simulator of that system - is technically feasible, and may be used both to resolve problems of several classes that arise in the development of a time-sharing system, and to tune the system to a user's needs. The detailed design of such a system, which is presented in this report, is a part of the result. The design phase indicated that the implementation of the Meta-System will require considerable effort, which will probably weigh against its implementation on any completed, operational system (especially since the benefit of the system would be diminished in this case). But if the Meta-System were to be developed along with the development of a time-sharing system, it could be a powerful tool for completing the system design and optimizing the system.

A second result of the study has the nature of a discovery about the structure of time-sharing operating systems, which may be applicable in cases other than that of the Meta-System. This result is that the user program demand for system resources may be measured at several naturally-occurring levels with the operating system. Measurements of demand taken from each of these levels is independent of the allocation of the resources which takes place beneath the level. The hardware level is the first, most obvious example. Beyond this, the selection of a set of operating system routines provides the basis of...
the definition of a level, and all system subroutines called by this
set of routines, will be included within the level. The sequence of
user program calls on these routines (i.e., calls from outside this set
of routines) may be considered a representation of user task demand.
It is said to be independent of the system, inasmuch as it is independent
of the operation of the system beneath the level. With some effort,
operating systems may be designed so that this structure is more rigidly
enforced, and the levels more easily defined.

Uniqueness of the Work

Nowhere in the current literature of computer systems has the im-
plementation of a system such as the Meta-System been reported, nor has
the design of such a system been proposed. Measurements of operational
systems are common, and so are simulations of computer systems. In
several instances, statistics taken from a predecessor computer system
are used for representations of resource demand in a simulation model
of a new system that is being developed. In these cases, however, the
statistics are input to the simulator in condensed form, and used by
the simulator to define probability distributions of the random variables
from which the actual events in the model are derived. Generally, only
the means and variances of the variables representing user task demand
are used, and each parameter is considered to be independent of all
others. Hence, all of information relating to the co-occurrences of
demand for various resources is lost.

The uniqueness of this investigation may be attributed to the fact
that it is not at all obvious that the representations of task demand
taken from a system, may be put to effective use in a model of the system. The attitude of most system designers is that simulations are useful only before the system is developed, and may be scrapped when the system becomes operational. One cause of this is, no doubt, that the approximations used to represent user demand within the simulation model, make the model inaccurate. In the Meta-System design, this deficiency is corrected.

Future Research

The work initiated in this effort may be pursued in several directions. First, since the study presented here is just the design of the Meta-System, the implementation remains. Implementation and use of the system are required before its usefulness (especially its cost effectiveness) can be fully known.

Second, the design concepts may be further developed. This exploration into the possibility of the Meta-System was oriented towards measurement and evaluation of a total general-purpose time-sharing system. The lower levels of system organization were most heavily stressed, because, being closest to the hardware allocation, they must necessarily be included within any simulation model. Also, because the functions performed at these levels are similar for many systems, the Meta-System designed at these levels will have some generality.

It should be apparent from the Meta-System levels presented here, that higher definitions of Meta-System levels may be defined. Other specific subsystems - such as the command language decoder, or debugging facility, or even a compiler - may then be included within the consciousness of a Meta-System.
Third, development of higher level Meta-Systems leads to the possibility of applying the Meta-System to the development of a dedicated on-line system. The dedicated system could be structured in a way that is similar to the time-sharing presented in this report, with the addition of one heavily-used subsystem. This Meta-System would be much more specialized than the one presented here, because it would depend heavily upon the structure of the subsystem, but it seems clear that such an effort could receive its impetus from the work done here.

Last, the research may take a theoretical direction. It could prove beneficial to the designs of future Meta-Systems, to construct theoretical models of time sharing systems, to study the requirement that representations of a systems operation - taken from within the system - remain valid even though part of the system is modified.

For example, assume a Turing Machine with a set of internal states $S$, to be a representation of the time-sharing system. Let $S'$, a subset of $S$, represent operation within the part of the system to be simulated. Each time the Turing Machine enters a state in $S'$ (from outside $S'$) record the 'total state' of the Turing Machine - the internal state and the entire tape. Call this state $(s_i, T_1)$. Record the total state of the machine again after the machine exits from the set $S'$ and call the state $(s_o, T_o)$. Now, in what ways could $(s_o, T_o)$ differ from $(s_i, T_i)$ and yet the future operation of the Turing Machine remain the same? This will define the set of allowable modifications that may be made in $S'$, and evaluated from simulation with input $(s_i, T_i)$. 
The question can be answered only after the model of the Turing Machine is refined to closer represent the operation of a time-sharing system. Development of such a model, and investigation of its behavior will be left to future research.
APPENDIX A

This appendix contains the description of the placement of the event-recording mechanisms in the operating system of the time-sharing system. All of the event-recording mechanisms of each of the four event traces are shown. Each is labeled with the event trace to which it belongs, as follows:

h : the hardware event trace
p : the physical event trace
l : the logical event trace
r : the relocation event trace

Event-recording mechanisms used in several traces have a label for each of the event traces concerned.
Figure A-1

Resident Parts of the Operating System
Page fault

Find sec. mem. address of required page

'pf' \( p_l \)

Page device available?

N

Put request on paging device queue

Y

Physical page available?

No

Replacement Algorithm

De-allocation required?

N

De-allocate page

Y

Page-out operation required?

N

Initiate page-out

Y

Initiate page-in

Exit to task initiator (p in Figure A-3)

Figure A-2
Page Fault Interrupt Response Routine
Figure A-3
Paging Channel Complete Interrupt Response Routine
Initiate Physical I/O Macro
Input: Device address
      Record address
      Operation

Is device busy?
Yes

No

Path set busy?
Yes

No

Is operation to this device on path set queue?
Yes

No

Queue for device

Queue for path set

Exit

Figure A-4
Physical I/O Initiation
Figure A-5

I/O-Channel-Complete-Interrupt Response
WAIT for I/O: input: identification of operation

Was I/O operation completed?

Yes

exit

No

Place task in Status of Unavailable for processor
to Task Initiator

Figure A-6a
I/O Wait Macro

WAIT for terminal response:

'Sleep p,l,r'

Place task in Status of terminal I/O Wait (Dormant state)
to Task Initiator

Figure A-6b
Terminal-Response-Wait Macro
Figure A-7

Terminal I/O Complete Interrupt Response
Logical I/O
GET file F, record A

- Fig. A-8 Logical I/O
GET.
Figure A-9
OPEN macro
VAM GET macro input:
- file name F
- record index X

allocate virtual memory for TOC

GET F, X

allocate virtual memory for entire record

construct page table entries for allocated pages: addresses are external storage addresses

Mark page table entries 'not in virtual memory'

end VAM GET F,X'

exit

Figure A-10

VAM GET macro
LOAD Input: program name PRG

Call VAN GET:
Return with
Address of prog:
Length of prog:
Address of TOC

Move relocation information to TDY

Complete DEF table in TDY

Mark page table entries 'unprocessed by loader'

Figure A-11
Program LOAD operation
Page fault interrupt response

not
in WM bit set?

Y
P I/O for page

N
page load

I/O Wait.

to paging operation

unprocessed by loader bit set?

Y
Use RLD in TDY to update addresses

N
end page load

exit

any unresolved REFS?

Y
is
REF in TDY?

Y
LOAD program

N
**Figure A-13 The CREATE Macro**
a) Task Wait Macro

'b' Task Wait Macro

Place i in TCB of task u

has task u been waited for i?

N

remove task u from inactive status

exit

Task Initiator

b) 'Post' Macro

Figure A-14 Task Wait and Post Macros
APPENDIX B

GLOSSARY

Artifact: the system modification that is necessary in order to measure the system, but which also modifies the operating characteristics of the system rendering the measurement inaccurate.

Backup storage (or backup memory): the storage device that is used as an extension to main memory. It generally contains parts of programs that are being executed, or whose execution is imminent, and data associated with those programs. Also called auxiliary storage.

Batch processing: a mode of operating a computer system in which a user (programmer) does not have any contact with his task until the execution of a group of tasks has been completed.

Demand paging: a paging technique, in which a part of a program is not brought into main memory until it is accessed, either in instruction fetch (execution) or data read or write.

Efficiency: the ratio of the time a device is active on any task during a sample period, to the total length of the period. The efficiency differs from utilization in that it does not include overhead time.

\[
\text{efficiency} = \frac{\text{total time} - \text{idle time}}{\text{total time} - \text{overhead time}}
\]
Evaluation: the application of an evaluation function to a particular computer system.

Evaluation criteria: the components of a real n-dimensional vector. Each component is the numerical value of some attribute of a computer system. For the evaluations performed in this work, the following attributes are used as evaluation criteria.

1. Utilization of the processor
2. Utilization of the paging channel
3. Utilization of the I/O channels
4. Average response time of a particular task

Evaluation function: a mapping from a class of computer systems into a set \{yes, no\}.

Event: 1. An occurrence within a computer system. For example, the beginning of a data-transfer from main memory to external memory.

2. The following set of data elements, which are recorded in conjunction with certain occurrences (or events according to definition 1):

   1. the time
   2. a task number
   3. a type descriptor
   4. other data, associated specific to the type of occurrence, as related by the type descriptor.
Event trace: a sequence of events, recorded during the operation of a computer system. In this work, two event traces are distinguished. They are:

1. system event trace: an event trace taken directly from the operation of a computer system, in which more than one task is in operation. The times in the events of this trace are real times.

2. task events: an event trace composed of only the events associated with one task. The times in the events of this trace are relative to the operations of that task only.

External storage: the storage media that are used to hold programs and data for long periods (minutes to years). Examples are magnetic tapes, magnetic disks and mass storage.

Main memory: the dynamic storage unit of a computer system, in which programs being executed reside. Also called core storage.

Measurement: the application of the measurement function to a computer system.

Measurement function: a mapping from a class of computer systems to the evaluation criteria space.

Multiprocessing: a mode of operating a computer system, in which there exists more than one hardware processor (CPU or CIU).
Multiprogramming: a mode of operating a computer system in which programs belonging to, or being used by, more than one task may reside in the main memory of the machine at any time. The motivation for multiprogramming is to allow the processor to switch quickly from one task to processing another, without waiting for the programs of the task to be loaded into memory.

Multitasking: the capability of a computer operating system of establishing a part of one task as an independent task, following a request from that task. The two tasks will then be able to compete for system resources enabling a degree of parallel processing, and quicker completion of the requirements of the original task. Multitasking is also known as forking.

Paging: a technique of executing programs in which only parts usually fixed-length of a program are brought into main memory at any time, and the specification of which part of the program is to be brought into main memory is not made by the programmer.

Peripheral device: a device by which data enters or leaves the computer system. Usually, these devices have slow speeds relative to the external storage devices. Examples are card readers, punches, keyboard terminals, printers.

Processor: 1. The hardware unit of the computer system in which the instructions are executed. Also called the central
processor (CP) or central processing unit (CPU).

2. A program, which when executed, transforms some particular data. Hence, it is said to be the processor of the data.

Raw data: a system event trace.

Recursiveness: the quality of a program that allows it to call on itself as a subroutine either directly or through another subroutine. In one instance of execution of a recursive program, the program may transfer to itself, and complete executing itself, then return to the first instance of its execution with the data of the first execution enhanced by the results of the second execution.

Reentrancy: the quality of routine or program that allows it to be interrupted at any point during one instance of its execution; then be executed in part or in whole in an entirely different application; then have the first instance of its execution completed, with the data of the program remaining undisturbed during the interruption.

Response time: the time interval between the time the user completes the process of entering a program or command to the computer system, and the time the computer begins outputting the results.

Task: a sequence of steps that the computer must perform in order to satisfy the requirements of a user. The steps are processor instructions, and I/O channel commands. The steps are directed
by the program being executed, but tasks and programs are not synonymous. If one program is executed for two different users, then two tasks are present.

A task is known to the operating system by the presence of a Task Control Block, or TCB. The TCB is a data area that contains all of the essential information relative to the task.

A task is usually the representation of the requirements of one user. However, if a particular user wishes to perform several logically distinct pieces of work, he may initiate several different tasks. There will then be several different TCB's, and the operating system will regard them as tasks for different users.

Task is synonymous with process.

Time sharing: A mode of operating a computer system, that has the following properties:

1. The capability of allowing multiple users to use the system simultaneously.

2. A response time that is small enough to encourage each user to wait for the computer's response to his input.

3. A capability of processing (on the CPU) each user's task, without necessarily waiting for the previous task to be completed. This is typically done by allocating each user's task a small
amount of time - called a time slice - on the processor. If a task is not completed during one time slice, it is not processed again, until other tasks have had a time slice of processing.

Utilization: the ratio of the time a device is active in a sample period, to the total length of the period. Activity of a device which might be considered overhead, is included in the 'active' time of the device.
BIBLIOGRAPHY


C2 Campbell, D.J. and W.J. Heffner, Measurement and Analysis of Large Operating Systems during System Development, AFIPS Proc. FJCC,
BIBLIOGRAPHY (continued)


BIBLIOGRAPHY (continued)


BIBLIOGRAPHY (continued)


F7 Freeman, David N. and Pearson, R.R., Efficiency vs. Responsiveness in a Multiple-Services Computer Facility, Proc. FJCC, p. 61, 1968.


G4 Gold, M.M., Time-Sharing and Batch Processing: An Experimental Comparison of Their Values in a Problem Solving Situation, CACM,
Vol. 12, No. 5, May 1969.


BIBLIOGRAPHY (continued)


BIBLIOGRAPHY (continued)


