Processes and Threads
Learning Outcomes

• An understanding of fundamental concepts of processes and threads
Major Requirements of an Operating System

• Interleave the execution of several processes to maximize processor utilization while providing reasonable response time

• Allocate resources to processes

• Support interprocess communication and user creation of processes
Processes and Threads

• Processes:
  • Also called a task or job
  • Execution of an individual program
  • “Owner” of resources allocated for program execution
  • Encompasses one or more threads

• Threads:
  • Unit of execution
  • Can be traced
    • list the sequence of instructions that execute
  • Belongs to a process
    • Executes within it.
Execution snapshot of three single-threaded processes (No Virtual Memory)

Figure 3.1  Snapshot of Example Execution (Figure 3 at Instruction Cycle 13)
Logical Execution Trace

5000  8000  12000
5001  8001  12001
5002  8002  12002
5003  8003  12003
5004  12004
5005  12005
5006  12006
5007  12007
5008  12008
5009  12009
5010  12010
5011  12011

(a) Trace of Process A  (b) Trace of Process B  (c) Trace of Process C

5000 = Starting address of program of Process A
8000 = Starting address of program of Process B
12000 = Starting address of program of Process C

Figure 3.2  Traces of Processes of Figure 3.1
Combined Traces
(Actual CPU Instructions)

What are the shaded sections?

<p>| | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>5000</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>5001</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>5002</td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>5003</td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>5004</td>
<td></td>
</tr>
<tr>
<td>6</td>
<td>5005</td>
<td></td>
</tr>
<tr>
<td>7</td>
<td>100</td>
<td>Time out</td>
</tr>
<tr>
<td>8</td>
<td>101</td>
<td></td>
</tr>
<tr>
<td>9</td>
<td>102</td>
<td></td>
</tr>
<tr>
<td>10</td>
<td>103</td>
<td></td>
</tr>
<tr>
<td>11</td>
<td>104</td>
<td></td>
</tr>
<tr>
<td>12</td>
<td>105</td>
<td></td>
</tr>
<tr>
<td>13</td>
<td>8000</td>
<td></td>
</tr>
<tr>
<td>14</td>
<td>8001</td>
<td></td>
</tr>
<tr>
<td>15</td>
<td>8002</td>
<td></td>
</tr>
<tr>
<td>16</td>
<td>8003</td>
<td></td>
</tr>
<tr>
<td>17</td>
<td>100</td>
<td>I/O request</td>
</tr>
<tr>
<td>18</td>
<td>101</td>
<td></td>
</tr>
<tr>
<td>19</td>
<td>102</td>
<td></td>
</tr>
<tr>
<td>20</td>
<td>103</td>
<td></td>
</tr>
<tr>
<td>21</td>
<td>104</td>
<td></td>
</tr>
<tr>
<td>22</td>
<td>105</td>
<td></td>
</tr>
<tr>
<td>23</td>
<td>12000</td>
<td></td>
</tr>
<tr>
<td>24</td>
<td>12001</td>
<td></td>
</tr>
<tr>
<td>25</td>
<td>12002</td>
<td></td>
</tr>
<tr>
<td>26</td>
<td>12003</td>
<td></td>
</tr>
<tr>
<td>27</td>
<td>12004</td>
<td>Time out</td>
</tr>
<tr>
<td>28</td>
<td>12005</td>
<td></td>
</tr>
<tr>
<td>29</td>
<td>100</td>
<td></td>
</tr>
<tr>
<td>30</td>
<td>101</td>
<td></td>
</tr>
<tr>
<td>31</td>
<td>102</td>
<td></td>
</tr>
<tr>
<td>32</td>
<td>103</td>
<td></td>
</tr>
<tr>
<td>33</td>
<td>104</td>
<td></td>
</tr>
<tr>
<td>34</td>
<td>105</td>
<td></td>
</tr>
<tr>
<td>35</td>
<td>5006</td>
<td></td>
</tr>
<tr>
<td>36</td>
<td>5007</td>
<td></td>
</tr>
<tr>
<td>37</td>
<td>5008</td>
<td></td>
</tr>
<tr>
<td>38</td>
<td>5009</td>
<td></td>
</tr>
<tr>
<td>39</td>
<td>5010</td>
<td></td>
</tr>
<tr>
<td>40</td>
<td>5011</td>
<td></td>
</tr>
<tr>
<td>41</td>
<td>100</td>
<td>Time out</td>
</tr>
<tr>
<td>42</td>
<td>101</td>
<td></td>
</tr>
<tr>
<td>43</td>
<td>102</td>
<td></td>
</tr>
<tr>
<td>44</td>
<td>103</td>
<td></td>
</tr>
<tr>
<td>45</td>
<td>104</td>
<td></td>
</tr>
<tr>
<td>46</td>
<td>105</td>
<td></td>
</tr>
<tr>
<td>47</td>
<td>12006</td>
<td></td>
</tr>
<tr>
<td>48</td>
<td>12007</td>
<td></td>
</tr>
<tr>
<td>49</td>
<td>12008</td>
<td></td>
</tr>
<tr>
<td>50</td>
<td>12009</td>
<td></td>
</tr>
<tr>
<td>51</td>
<td>12010</td>
<td></td>
</tr>
<tr>
<td>52</td>
<td>12011</td>
<td></td>
</tr>
</tbody>
</table>

100 = Starting address of dispatcher program
shaded areas indicate execution of dispatcher process;
first and third columns count instruction cycles;
second and fourth columns show address of instruction being executed

Figure 3.3 Combined Trace of Processes of Figure 3.1
Summary: The Process Model

- Multiprogramming of four programs
- Conceptual model of 4 independent, sequential processes (with a single thread each)
- Only one program active at any instant
Figure 4.1  Threads and Processes [ANDE97]
Process and thread models of selected OSes

• Single process, single thread
  • MSDOS
• Single process, multiple threads
  • OS/161 as distributed
• Multiple processes, single thread
  • Traditional UNIX
• Multiple processes, multiple threads
  • Modern Unix (Linux, Solaris), Windows

Note: Literature (incl. Textbooks) often do not cleanly distinguish between processes and threads (for historical reasons)
Process Creation

Principal events that cause process creation

1. System initialization
   • Foreground processes (interactive programs)
   • Background processes
     • Email server, web server, print server, etc.
     • Called a daemon (unix) or service (Windows)

2. Execution of a process creation system call by a running process
   • New login shell for an incoming ssh connection

3. User request to create a new process

4. Initiation of a batch job

Note: Technically, all these cases use the same system mechanism to create new processes.
Process Termination

Conditions which terminate processes
1. Normal exit (voluntary)
2. Error exit (voluntary)
3. Fatal error (involuntary)
4. Killed by another process (involuntary)
Implementation of Processes

• A processes’ information is stored in a process control block (PCB)
• The PCBs form a process table
  • Reality can be more complex (hashing, chaining, allocation bitmaps,...)
# Implementation of Processes

<table>
<thead>
<tr>
<th>Process management</th>
<th>Memory management</th>
<th>File management</th>
</tr>
</thead>
<tbody>
<tr>
<td>Registers</td>
<td>Pointer to text segment</td>
<td>Root directory</td>
</tr>
<tr>
<td>Program counter</td>
<td>Pointer to data segment</td>
<td>Working directory</td>
</tr>
<tr>
<td>Program status word</td>
<td>Pointer to stack segment</td>
<td>File descriptors</td>
</tr>
<tr>
<td>Stack pointer</td>
<td></td>
<td>User ID</td>
</tr>
<tr>
<td>Process state</td>
<td></td>
<td>Group ID</td>
</tr>
<tr>
<td>Priority</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Scheduling parameters</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Process ID</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Parent process</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Process group</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Signals</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Time when process started</td>
<td></td>
<td></td>
</tr>
<tr>
<td>CPU time used</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Children’s CPU time</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Time of next alarm</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

## Example fields of a process table entry
Process/Thread States

- Possible process/thread states
  - running
  - blocked
  - ready
- Transitions between states shown

1. Process blocks for input
2. Scheduler picks another process
3. Scheduler picks this process
4. Input becomes available
Some Transition Causing Events

Running $\rightarrow$ Ready
- Voluntary `Yield()`
- End of timeslice

Running $\rightarrow$ Blocked
- Waiting for input
  - File, network,
- Waiting for a timer (alarm signal)
- Waiting for a resource to become available
Scheduler

- Sometimes also called the *dispatcher*
  - The literature is also a little inconsistent on with terminology.
- Has to choose a *Ready* process to run
  - How?
    - It is inefficient to search through all processes
The Ready Queue

(b) Queuing diagram
What about blocked processes?

• When an *unblocking* event occurs, we also wish to avoid scanning all processes to select one to make *Ready*
Using Two Queues
(b) Multiple blocked queues
Threads
The Thread Model

(a) Three processes each with one thread
(b) One process with three threads
The Thread Model – Separating execution from the environment.

<table>
<thead>
<tr>
<th>Per process items</th>
<th>Per thread items</th>
</tr>
</thead>
<tbody>
<tr>
<td>Address space</td>
<td>Program counter</td>
</tr>
<tr>
<td>Global variables</td>
<td>Registers</td>
</tr>
<tr>
<td>Open files</td>
<td>Stack</td>
</tr>
<tr>
<td>Child processes</td>
<td>State</td>
</tr>
<tr>
<td>Pending alarms</td>
<td></td>
</tr>
<tr>
<td>Signals and signal handlers</td>
<td></td>
</tr>
<tr>
<td>Accounting information</td>
<td></td>
</tr>
</tbody>
</table>

- Items shared by all threads in a process
- Items private to each thread
Threads Analogy

The Hamburger Restaurant
Single-Threaded Restaurant

Customer Arrives → Take Order → Start Fries → Fries Cook → Fries Finish → Start Burger → Burger Cooks → Burger Finished → Assemble Order → Serve Customer → Wait for Customer

Blocking operations delay all activities
Multithreaded Restaurant

Note: Ignoring synchronisation issues for now
Multithreaded Restaurant with more worker threads

Customer Arrives

Wait for Customer

Take Order

Assemble Order

Serve Customer

Start Fries

Fries Cook

Fries Finish

Start Burger

Burger Cooks

Burger Finished

Start Burger

Burger Cooks

Burger Finished

Start Burger

Burger Cooks

Burger Finished
Finite-State Machine Model
(Event-based model)

Input Events

Customer Arrives
Fries Finish
Burger Finished

Non-Blocking actions

Start Fries
Serve Customer
Start Burger

External activities

Fries Cook
Burger Cooks
Wait for Customer
Observation: Computation State

Thread Model

• State implicitly stored on the stack.

Finite State (Event) Model

• State explicitly managed by program
The Thread Model

Each thread has its own stack
Thread Model

• Local variables are per thread
  • Allocated on the stack

• Global variables are shared between all threads
  • Allocated in data section
  • Concurrency control is an issue

• Dynamically allocated memory (malloc) can be global or local
  • Program defined (the pointer can be global or local)
Thread Usage

A word processor with three threads
Thread Usage

A multithreaded Web server
Thread Usage

- Rough outline of code for previous slide
  (a) Dispatcher thread
  (b) Worker thread – can overlap disk I/O with execution of other threads

```c
while (TRUE) {
    get_next_request(&buf);
    handoff_work(&buf);
}
```
```c
while (TRUE) {
    wait_for_work(&buf)
    look_for_page_in_cache(&buf, &page);
    if (page_not_in_cache(&page)
        read_page_from_disk(&buf, &page);
    return_page(&page);
}
```
## Thread Usage

<table>
<thead>
<tr>
<th>Model</th>
<th>Characteristics</th>
</tr>
</thead>
<tbody>
<tr>
<td>Threads</td>
<td>Parallelism, blocking system calls</td>
</tr>
<tr>
<td>Single-threaded process</td>
<td>No parallelism, blocking system calls</td>
</tr>
<tr>
<td>Finite-state machine</td>
<td>Parallelism, nonblocking system calls, interrupts</td>
</tr>
</tbody>
</table>

Three ways to construct a server
Summarising “Why Threads?”

• Simpler to program than a state machine
• Less resources are associated with them than a complete process
  • Cheaper to create and destroy
  • Shares resources (especially memory) between them
• Performance: Threads waiting for I/O can be overlapped with computing threads
  • Note if all threads are compute bound, then there is no performance improvement (on a uniprocessor)
• Threads can take advantage of the parallelism available on machines with more than one CPU (multiprocessor)