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.

Logical Execution Trace

```
<table>
<thead>
<tr>
<th>5000</th>
<th>5001</th>
<th>5002</th>
<th>5003</th>
<th>5004</th>
<th>5005</th>
<th>5006</th>
<th>5007</th>
<th>5008</th>
<th>5009</th>
<th>5010</th>
<th>5011</th>
</tr>
</thead>
<tbody>
<tr>
<td>R000</td>
<td>R001</td>
<td>R002</td>
<td>R003</td>
<td>R004</td>
<td>R005</td>
<td>R006</td>
<td>R007</td>
<td>R008</td>
<td>R009</td>
<td>R010</td>
<td>R011</td>
</tr>
</tbody>
</table>
```

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

5000 = Starting address of program of Process A
5001 = Starting address of program of Process B
12000 = Starting address of program of Process C

Figure 3.1 Snapshot of Example Execution (Figure 3.1)

Figure 3.2 Traces of Processes of Figure 3.1
**What are the shaded sections?**

**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

**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 telnet/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)
Process/Thread States

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

Some Transition Causing Events

Running → Ready
- Voluntary \texttt{Yield()}
- End of timeslice
Running → Blocked
- Waiting for input
  - File, network,
  - Waiting for a timer (alarm signal)
  - Waiting for a resource to become available

Scheduler

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

The Ready Queue

What about blocked processes?

- When an \textit{unblocking} event occurs, we also wish to avoid scanning all processes to select one to make \textit{Ready}

Using Two Queues
Implementation of Processes

- A processes’ information is stored in a process control block (PCB).
- The PCBs form a process table.
  - Sometimes the kernel stack for each process is in the PCB.
  - Sometimes some process info is on the kernel stack.
    - E.g., registers in the trapframe in OS/161.
  - Reality is much more complex (hashing, chaining, allocation bitmaps, …)

Example fields of a process table entry

Threads

The Thread Model

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

Threads Analogy

The Hamburger Restaurant
Single-Threaded Restaurant

Blocking operations delay all activities

Multithreaded Restaurant

Multithreaded Restaurant with more worker threads

Finite-State Machine Model
(Event-based model)

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)
Observation: Computation State

Thread Model
- State implicitly stored on the stack.

Finite State (Event) Model
- State explicitly managed by program

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

Thread Usage
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)