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

Execution snapshot of three single-threaded processes (No Virtual Memory)

5000 = Starting address of program of Process A
5001 = Starting address of program of Process B
5002 = Starting address of program of Process C
12000 = Starting address of program of Process C
Combined Traces
(Actual CPU Instructions)

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 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 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) Queueing 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

(a) Single blocked queue
Implementation of Processes

- A process's 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
  - 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

<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>Process state</td>
<td>File descriptors</td>
</tr>
<tr>
<td>Stack pointer</td>
<td>Priority</td>
<td>User ID</td>
</tr>
<tr>
<td>Process state</td>
<td>Scheduling parameters</td>
<td>Group ID</td>
</tr>
<tr>
<td>Priority</td>
<td>Process ID</td>
<td></td>
</tr>
<tr>
<td>Scheduling parameters</td>
<td>Parent process</td>
<td></td>
</tr>
<tr>
<td>Process ID</td>
<td>Process group</td>
<td></td>
</tr>
<tr>
<td>Parent process</td>
<td>Signals</td>
<td></td>
</tr>
<tr>
<td>Process group</td>
<td>Time when process started</td>
<td></td>
</tr>
<tr>
<td>Signals</td>
<td>CPU time</td>
<td></td>
</tr>
<tr>
<td>Time when process started</td>
<td>CPU time used</td>
<td></td>
</tr>
<tr>
<td>CPU time used</td>
<td>Child's CPU time</td>
<td></td>
</tr>
<tr>
<td>Child's CPU time</td>
<td>Time of next alarm</td>
<td></td>
</tr>
</tbody>
</table>

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

Note: ignoring synchronization issues for now

Multithreaded Restaurant with more worker threads

Finite-State Machine Model
(Input Events)

Non-Blocking actions

External activities

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

Thread Usage

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)