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

5000  8000  12000
5001  8001  12001
5002  8002  12002
5003  8003  12003
5004  8004  12004
5005  8005  12005
5006  8006  12006
5007  8007  12007
5008  8008  12008
5009  8009  12009
5010  80010  12010
5011  80011  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
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/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 terminology.
- Has to choose a *Ready* process to run
  - How?
  - It is inefficient to search through all processes

The Ready Queue

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
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

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

Threads

The Thread Model

- Separating execution from the environment.

Per process items
- Address space
- Global variables
- Open files
- Child processes
- Pending alarms
- Signals and signal handlers
- Accounting information

Per thread items
- Program counter
- Registers
- Stack
- State

Threads Analogy

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

Customer arrives → Take Order → Fry Cook → Assemble Order → Fry Finish → Serve Customer → Wait for Customer → Blocking operation → delay all activities

Multithreaded Restaurant

Customer arrives → Take Order → Fry Cook → Assemble Order → Fry Finish → Start Fries → Serve Customer → Wait for Customer → Note: Ignoring synchronization issues for now

Multithreaded Restaurant with more worker threads

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

Finite-State Machine Model (Event-based model)

Input Events → Fries Cook → Fries Finish → Start Fries → 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

Thread 1, Thread 2, Thread 3 → Process → Thread 1’s stack → Thread 2’s stack → Thread 3’s stack → Kernel → Each thread has its own stack
Thread Model

- Local variables are per thread
  - Allocated on the stack
- Global variables are shared between all threads
  - 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

```
while (TRUE) {
    get_next_request(&buf);
    handle_work(&buf);
}
```

(a) Dispatcher thread

```
while (TRUE) {
    wait_for_work(&buf);
    look_for_page_in_cache(&buf, &page);
    if (page not in cache) {
        read_page_from_disk(&buf, &page);
        return page to user;
    }
}
```

(b) Worker thread – can overlap disk I/O with execution of other threads

Thread Usage

```
Dispatcher thread
Worker thread
```

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)