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 8010 12010
5011 8011 12011

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

<table>
<thead>
<tr>
<th>Time (ns)</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1833</td>
</tr>
<tr>
<td>1</td>
<td>1833</td>
</tr>
<tr>
<td>2</td>
<td>1833</td>
</tr>
<tr>
<td>3</td>
<td>1833</td>
</tr>
<tr>
<td>4</td>
<td>1833</td>
</tr>
<tr>
<td>5</td>
<td>1833</td>
</tr>
<tr>
<td>...</td>
<td></td>
</tr>
</tbody>
</table>

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 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,...)

Example fields of a process table entry

Process/Thread States

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

Process management
- Registers
- Program counter
- Program status word
- Stack pointer
- Process state
- Priority
- Scheduling parameters
- Process ID
- Parent process
- Process group
- Signals
- Time when process started
- CPU time used
- Children’s CPU time
- Time of next alarm

Memory management
- Pointer to text segment
- Pointer to data segment

File management
- Root directory
- Working directory
- File descriptors
- User ID
- Group ID

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 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

![Diagram](image)

 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>Accounting info</td>
</tr>
<tr>
<td>Signals and signal handlers</td>
<td></td>
</tr>
</tbody>
</table>

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

Threads Analogy

A hamburger restaurant
Single-Threaded Restaurant

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

Blocking operations delay all activities

Multithreaded Restaurant

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

Note: Ignoring synchronisation issues for now

Finite-State Machine Model
(Event-based 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

Thread 1's stack

Thread 2
Thread 3

Process

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
  - 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

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

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)