Lecture 2:
Processes and Threads - Part 1

Mohamed Zahran (aka Z)
mzahran@cs.nyu.edu
http://www.mzahran.com
OS Management of Application Execution

• Resources are made available to multiple applications
• The processor is switched among multiple applications so all will appear to be progressing
• The processor and I/O devices can be used efficiently
When the processor begins to execute the program code, we refer to this executing entity as a *process*.
What Is a Process?

An abstraction of a running program

Program Counter
The Process Model

• A process is an instance of an executing program and includes
  – Program counter
  – Registers
  – Variables
  – ...

• A process has a program, input, output, and state.

If a program is running twice, does it count as two processes? or one?
Process Element

- While the program is executing, this process can be uniquely characterized by a number of elements, including:

  - identifier
  - state
  - priority
  - program counter
  - memory pointers
  - context data
  - I/O status information
  - accounting information

Where do we store all this info?
Process Control Block

- Contains the process elements
- It is possible to interrupt a running process and later resume execution as if the interruption had not occurred
- Created and managed by the operating system
- Key tool that allows support for multiple processes

Figure 3.1  Simplified Process Control Block
Multiprogramming

- One CPU and several processes
- CPU switches from process to process quickly

<table>
<thead>
<tr>
<th>Processes</th>
<th>0</th>
<th>1</th>
<th>( \cdots )</th>
<th>( n-2 )</th>
<th>( n-1 )</th>
</tr>
</thead>
<tbody>
<tr>
<td>Scheduler</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
What Really Happens: One program counter switches between processes A, B, C, and D.

What We Think It Happens: There are four separate program counters, each handling a process (A, B, C, D), with time progression shown on the right graph.
If we run the same program several times, will we get the same execution time?

What Really Happens

What We Think It Happens
Example

Small program that switches the processor from one process to another (also called Scheduler)
Process Creation

• System initialization
  – At boot time
  – Foreground
  – Background (daemons)
• Execution of a process creation system call by a running process
• A user request
• A batch job
• Created by OS to provide a service
• Interactive logon
Process Termination

- Normal exit (voluntary)
- Error exit (voluntary)
- Fatal error (involuntary)
- Killed by another process (involuntary)
<table>
<thead>
<tr>
<th>Scenario</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Normal completion</td>
<td>The process executes an OS service call to indicate that it has completed running.</td>
</tr>
<tr>
<td>Time limit exceeded</td>
<td>The process has run longer than the specified total time limit. There are a number of possibilities for the type of time that is measured. These include total elapsed time (&quot;wall clock time&quot;), amount of time spent executing, and, in the case of an interactive process, the amount of time since the user last provided any input.</td>
</tr>
<tr>
<td>Memory unavailable</td>
<td>The process requires more memory than the system can provide.</td>
</tr>
<tr>
<td>Bounds violation</td>
<td>The process tries to access a memory location that it is not allowed to access.</td>
</tr>
<tr>
<td>Protection error</td>
<td>The process attempts to use a resource such as a file that it is not allowed to use, or it tries to use it in an improper fashion, such as writing to a read-only file.</td>
</tr>
<tr>
<td>Arithmetic error</td>
<td>The process tries a prohibited computation, such as division by zero, or tries to store numbers larger than the hardware can accommodate.</td>
</tr>
<tr>
<td>Time overrun</td>
<td>The process has waited longer than a specified maximum for a certain event to occur.</td>
</tr>
<tr>
<td>I/O failure</td>
<td>An error occurs during input or output, such as inability to find a file, failure to read or write after a specified maximum number of tries (when, for example, a defective area is encountered on a tape), or invalid operation (such as reading from the line printer).</td>
</tr>
<tr>
<td>Invalid instruction</td>
<td>The process attempts to execute a nonexistent instruction (often a result of branching into a data area and attempting to execute the data).</td>
</tr>
<tr>
<td>Privileged instruction</td>
<td>The process attempts to use an instruction reserved for the operating system.</td>
</tr>
<tr>
<td>Data misuse</td>
<td>A piece of data is of the wrong type or is not initialized.</td>
</tr>
<tr>
<td>Operator or OS intervention</td>
<td>For some reason, the operator or the operating system has terminated the process (e.g., if a deadlock exists).</td>
</tr>
<tr>
<td>Parent termination</td>
<td>When a parent terminates, the operating system may automatically terminate all of the offspring of that parent.</td>
</tr>
<tr>
<td>Parent request</td>
<td>A parent process typically has the authority to terminate any of its offspring.</td>
</tr>
</tbody>
</table>
Process State

- Depending on the implementation, there can be several possible state models.

- The Simplest one: Two-state diagram

(a) State transition diagram
Process State:
Three-State Model

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

Figure 3.6 Five-State Process Model
Using Queues to Manage Processes

- Admit → Ready Queue → Dispatch → Processor → Release
- Timeout
- Event Occurs → Blocked Queue → Event Wait

(a) Single blocked queue
Using Queues to Manage Processes

(b) Multiple blocked queues
One Extra State!

Swapped to disk
One Extra State!
Implementation of Processes

• OS maintains a **Control table (also called process table)**
• An array of structures
• One entry per process
Conceptual view of the tables that OS maintains in order to manage execution of processes on resources.
A Bit About Interrupts

1. Hardware stacks program counter, etc.
2. Hardware loads new program counter from interrupt vector.
3. Assembly language procedure saves registers.
4. Assembly language procedure sets up new stack.
5. C interrupt service runs (typically reads and buffers input).
6. Scheduler decides which process is to run next.
7. C procedure returns to the assembly code.
8. Assembly language procedure starts up new current process.
Simple Modeling of Multiprogramming

• A process spends fraction $p$ waiting for I/O
• Assume $n$ processors in memory at once
• The probability that all processes are waiting for I/O at once is $p^n$
• So -> CPU Utilization = $1 - p^n$
Multiprogramming lets processes use the CPU when it would otherwise become idle.
Executing the OS Itself

(a) Separate kernel

(b) OS functions execute within user processes

(c) OS functions execute as separate processes
Threads

• Multiple threads of control within a process
• All threads of a process share the same address space
Why Threads?

• For some applications many activities can happen at once
  – With threads, programming becomes easier
  – Benefit applications with I/O and processing that can overlap

• Lighter weight than processes
  – Faster to create and restore
Example 1:
A Word Processor
Example 2: Multithreaded Web Server
Processes vs Threads

- Process groups resources
- Threads are entities scheduled for execution on CPU
- No protections among threads (unlike processes) [Why?]
- Thread can be in any of several states: running, blocked, ready, and terminated
<table>
<thead>
<tr>
<th><strong>Per process items</strong></th>
<th><strong>Per thread items</strong></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>
Each thread has its own stack (Why?).
Where to Put The Thread Package?

User space

Kernel space
Implementing Threads in User Space

- Threads are implemented by a library
- Kernel knows nothing about threads
- Each process needs its own private thread table
- Thread table is managed by the runtime system
Implementing Threads in User Space

**Advantages**
- Very fast thread scheduling
- Each process can have its own thread scheduling algorithm
- Scale better

**Disadvantages**
- Blocking system calls can block the whole process
- Page fault blocks the whole process
- No other thread of the process will ever run unless the running thread voluntarily gives up the CPU
Implementing Threads in Kernel Space

• Kernel knows about and manages the threads
• No runtime is needed in each process
• Creating/destroying/(other thread related operations) a thread involves a system call
Implementing Threads in Kernel Space

Advantages
- When a thread blocks (due to page fault or blocking system calls) the OS can execute another thread from the same process

Disadvantages
- Cost of system call is very high
Hybrid Implementation

Multiple user threads on a kernel thread

Kernel thread

User space

Kernel space
Conclusions

• Processes is the most central concept in OS
• Process vs Thread
• Multiprogramming vs multithreading