Chapter 2 - (First Part)

Processes and Threads

Slide Credits:
Jonathan Walpole
Andrew Tanenbaum
Lecture overview

**Processes**
- Process Scheduler
- Process States
- Process Hierarchies
- Relevant Unix System Calls

**Threads**
- Comparison to Processes
- Examples
- User-Level Thread Package
Processes

A process is a program in execution.

Program
Description of how to perform an activity
  Instructions and static data values

Process
A snapshot of a program in execution.
  • Memory
    (Instructions, Data, Runtime Stack)
  • CPU state (Registers, PC, SP, etc.)
  • Operating system state
    (open files, accounting statistics, etc.)
Virtual Address Space

Each process runs in its own \textit{virtual memory address space}.
Which consists of...

\textbf{Text} – the program code (usually read-only)

\textbf{Data space} – variables (initialized/uninitialized)

\textbf{Stack space} – used for function calls

Invoke the same program multiple times?
... Results in the creation of multiple, distinct address spaces.
Process Switching

In its simplest form, a computer performs instructions on operands. Registers are used to hold values temporarily to speed things up.
Process Switching

In its simplest form, a computer performs instructions on operands. Registers are used to hold values temporarily to speed things up.

Program 1 is running
Process Switching

Saving all the information about a process allows a process to be *temporarily suspended.*
Process Switching

Saving all the information about a process allows a process to be temporarily suspended.
Process Switching

Saving all the information about a process allows a process to be *temporarily suspended*.

*Program 2 now has the CPU*
Process Switching

Saving all the information about a process allows a process to be *temporarily suspended*. 

![Diagram of memory and CPU components]

- Memory: Program Code, Program Data
- CPU: ALU, Regs, SP, PC

Save the state of program 2
Process Switching

Saving all the information about a process allows a process to be *temporarily suspended.*
Process Switching

Saving all the information about a process allows a process to be temporarily suspended.

*Program 1 has the CPU*
Why use the process abstraction?

- Multiprogramming of four programs in the same address space
- Conceptual model of 4 independent, *sequential processes*
- Only one program is active at any instant
The Role of the Scheduler

Lowest layer of process-structured OS
  *handles interrupts & scheduling of processes*

Above that layer are sequential processes
Process States

Possible states of a process:

- RUNNING
- BLOCKED
- READY

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

Skeleton of what the lowest levels of the OS do when an interrupt occurs

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.
How Can Processes Be Created?

Events that create processes...

- System initialization
- Initiation of a batch job
- Execution of a “process creation” system call
  (from another process)
- User request to create a new process
Process Hierarchies

Parent creates a child process.

Special system calls for communicating with and waiting for child processes

Each process is assigned a unique identifying number
The “Process ID” or “pid”.

Child processes can create their own child processes.
Forms a hierarchy
UNIX calls this a “Process Group”

Windows has no concept of process hierarchy.
“All processes are created equal.”
How do Processes Terminate?

Conditions which terminate processes...

- Normal exit (voluntary)
- Error exit (voluntary)
- Fatal error (involuntary)
- Killed by another process (involuntary)
Process creation in UNIX

All processes have a unique process id
getpid(), getppid() allow processes to get their information

Process creation
fork() creates an exact copy of the process
identical with exception of the return value of fork()
exec() replaces an address space with a new program
system() like CreateProcess()

Process termination, signaling
signal(), kill() allows a process to be terminated or have specific signals sent to it
Example: Process Creation in UNIX

csh (pid = 22)

```c
...  
pid = fork()  
if (pid == 0) {
    // child...  
    ...  
    exec();  
}  
else {
    // parent  
    wait();  
}  
...  
```
Example: Process Creation in UNIX

csh (pid = 22)
...

\texttt{pid = fork()}
\texttt{if (pid == 0) \{ // child...}
     ...
     \texttt{exec();}
     \}
else \{ // parent
     \texttt{wait();}
     \}
...

csh (pid = 24)
...

\texttt{pid = fork()}
\texttt{if (pid == 0) \{ // child...}
     ...
     \texttt{exec();}
     \}
else \{ // parent
     \texttt{wait();}
     \}
...

Example: Process Creation in UNIX

```
csh (pid = 22)
...
pid = fork()
if (pid == 0) {
    // child...
    ...
    exec();
}
else {
    // parent
    wait();
}
...

csh (pid = 24)
...
pid = fork()
if (pid == 0) {
    // child...
    ...
    exec();
}
else {
    // parent
    wait();
}
...
```
Example: Process Creation in UNIX

```c
... pid = fork();
if (pid == 0) {
    // child...
    ...
    exec();
}
else {
    // parent
    wait();
}
...
```

csh (pid = 22)

```c
... pid = fork();
if (pid == 0) {
    // child...
    ...
    exec();
}
else {
    // parent
    wait();
}
...
```

csh (pid = 24)
Example: Process Creation in UNIX

```
csh (pid = 22)
...
pid = fork()
if (pid == 0) {
    // child...
    ...
    exec();
} else {
    // parent
    wait();
}
...

ls (pid = 24)
//ls program
main(){
    // look up dir
    ...
}
```

"
What other process state does the OS manage?

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

Fields of a process table entry
What about the OS?

Is the OS a process?
It is a program in execution, after all …
Does it need a process control block?
Who manages its state when it’s not running?
Threads

Processes have the following components:
- an address space
- a collection of operating system state
- a CPU context … or *thread of control*

On multiprocessor systems, with several CPUs, it would make sense for a process to have several CPU contexts (threads of control)

Multiple threads of control can run in the same address space on a single CPU system too!

“*thread of control*” and “*address space*” are orthogonal concepts
Threads

• Threads share a process address space with zero or more other threads

• Threads have their own
  CPU State (PC, SP, register values, etc.)
  Stack

• What other OS state should be private to threads?

A traditional process can be viewed as:

*An address space with a single thread!*
Threads vs Processes

(a) Three processes each with one thread
(b) One process with three threads
# Process State vs Thread State

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

- **Items shared by all threads in a process**
- **Items private to each thread**
Independent execution of threads

Each thread has its own stack
A word processor with three threads
Processes versus threads - example

Web server receives a request for a page...

GET / HTTP/1.0
Processes versus threads - example

Web server receives a request for a page...

GET / HTTP/1.0

HTTPD

disk

Why is this not a good web server design?
Processes versus threads - example

Web server receives a request for a page...

GET / HTTP/1.0
Processes versus threads - example

Web server receives a request for a page...

GET / HTTP/1.0

HTTPD

GET / HTTP/1.0

disk
Processes versus threads - example

Web server receives a request for a page...

GET / HTTP/1.0
GET / HTTP/1.0
GET / HTTP/1.0
GET / HTTP/1.0
Threads in a web server

A multithreaded Web server
Threads in a web server

Outline of code for previous slide:

Dispatcher thread

```c
while (TRUE) {
    get_next_request(&buf);
    handoff_work(&buf);
}
```

Worker thread

```c
while (TRUE) {
    wait_for_work(&buf)
    look_for_page_in_cache(&buf, &page);
    if (page_not_in_cache(&page))
        read_page_from_disk(&buf, &page);
    return_page(&page);
}
```

(a) (b)
System structuring options

<table>
<thead>
<tr>
<th>Model</th>
<th>Characteristics</th>
</tr>
</thead>
<tbody>
<tr>
<td>Threads</td>
<td>Parallelism, blocking system calls</td>
</tr>
<tr>
<td>Single-threaded process</td>
<td>No parallelism, blocking system calls</td>
</tr>
<tr>
<td>Finite-state machine</td>
<td>Parallelism, nonblocking system calls, interrupts</td>
</tr>
</tbody>
</table>

Three ways to construct a server
Pros & Cons of Threads

Pros
• Overlap I/O with computation!
• Cheaper context switches
• Better mapping to shared memory multiprocessors

Cons
• Potential thread interactions
• Complexity of debugging
• Complexity of multi-threaded programming
• Backwards compatibility with existing code
Making Single-Threaded Code Multithreaded

There is a global variable.
The global variable is modified.
The global variable is then tested.
Making Single-Threaded Code Multithreaded

There is a global variable.
The global variable is modified.
The global variable is then tested.

Typical “C” code...

```c
i = read (file, &buff, n);
if (errno) { ...print error message... }
```
Making Single-Threaded Code Multithreaded

There is a global variable.
The global variable is modified.
The global variable is then tested.

Typical “C” code...

```c
i = read (file, &buff, n);
if (errno) { ...print error message... }
```

Now imagine that several threads are executing...
Making Single-Threaded Code Multithreaded

Thread 1

\[
\text{read (errno is set)}
\]

\[
\text{if errno... (errno is inspected)}
\]
Making Single-Threaded Code Multithreaded

Thread 1

read (errno is set)

if errno...

(errno is inspected)

Thread 2

read
Making Single-Threaded Code Multithreaded

Thread 1

- read (errno is set)
- if errno...

Thread 2

- read
- (errno is inspected)
Making Single-Threaded Code Multithreaded

Thread 1

read (errno is set)

if errno...

Thread 2

read (errno is overwritten)

if errno... (errno is inspected)
Making Single-Threaded Code Multithreaded

Threads can have private global variables
User-Level Threads

Threads can be implemented...
  • By the OS, or
  • At user level

Kernel-Level Thread Implementation

The Kernel contains the code to switch switch between different threads.

User-Level Thread Implementations

Thread scheduler runs as user code.
All thread management done by user code.
(Kernel sees only a traditional process.)
1: Implementing Threads in the Kernel

The thread switching code is in the kernel.
2: A “User-Level Threads Package”

The thread switching code is in the user address space.
User-level threads

**Advantages**

- Cheap context switch costs!
- User-programmable scheduling policy!

**Disadvantages**

- How to deal with blocking system calls!
- How to overlap I/O and computation!
Hybrid Thread Implementations

Multiplexing user-level threads onto kernel-level threads

Multiple user threads on a kernel thread

Kernel

Kernel thread

User space

Kernel space
Scheduler Activations

**Goals:**
- Mimic functionality of kernel threads
- Gain performance of user space threads

The idea - kernel upcalls to user-level thread scheduling code when it handles a blocking system call or page fault
  - User level thread scheduler can choose to run a different thread rather than blocking
  - Kernel upcalls when system call or page fault returns

Kernel assigns virtual processors to each process
  (which contains a user level thread scheduler)
  Lets user level thread scheduler allocate threads to processors

**Problem:** Relies on upcalls
  - Kernel (lower layer) calls procedures in user space (higher layer)
Summary

• Processes
• Threads

**Project 2:**

Due in 1 week!
Okay to discuss my code,
... but write your own code!!!