CS 333
Introduction to Operating Systems

Class 2 - OS-Related Hardware & Software
The Process Concept

Jonathan Walpole
Computer Science
Portland State University
Lecture overview

OS-Related Hardware & Software
  Memory protection and relocation
  Virtual memory & MMUs
  I/O & Interrupts

Processes
  Process scheduling
  Process states
  Process hierarchies
  Process system calls in Unix
Memory protection and relocation ...

- Memory protection (basic ideas)
  - virtual vs physical addresses
    - address range in each application starts at 0
  - “base register” used to convert each virtual address to a physical address before main memory is accessed
  - address is compared to a “limit register” to keep memory references within bounds

- Relocation
  - by changing the base register value

- Paged virtual memory
  - same basic concept, but more powerful (and complex)
Base & Limit Registers (single & multiple)
Virtual memory and MMUs

- **Memory management unit (MMU)**
  - hardware provided equivalent of base registers
  - at the granularity of “pages” of memory, say 2kB, i.e., lots of them!
  - supports relocation at page granularity
  - applications need not occupy contiguous physical memory

- **Memory protection**
  - limit registers don’t work in this context
  - per-page and per-application protection registers

- **Relocation and protection occur at CPU speeds!**
What about I/O devices?

A simplified view of a computer system
Structure of a large Pentium system
What about I/O devices?

A simplified view of a computer system
How do programs interact with devices?

- **Devices vs device controllers vs device drivers**
  - device drivers are part of the OS
  - programs call the OS which calls the device driver

- **Device drivers interact with device controllers**
  - either using special IO instructions
  - or by reading/writing controller registers that appear as memory locations

- **Why protect access to devices by accessing them indirectly via the OS?**
How do devices interact with programs?

- Interrupts
Different types of interrupts

- **Timer interrupts**
  - Allows OS to maintain control
  - One way to keep track of time

- **I/O interrupts**
  - Keyboard, mouse, disks, etc...

- **Hardware failures**

- **Program generated (traps)**
  - Programming errors: seg. faults, divide by zero, etc.
  - System calls like read(), write(), gettimeofday()
Timer interrupts

- OS can ask timer device to interrupt after a specified time period has elapse

- Interrupt invokes timer interrupt handler which invokes OS “scheduler”

- OS can take the opportunity to save the current application and restore a different one
  - context switch
Why use traps for system calls?

- The Operating System is just a program!
- It must have the privilege to manipulate the hardware
  - set base and limit registers for memory protection
  - access devices
  - set and clear mode bit to enable privilege
- If user programs execute with the mode bit clear, and do not have privilege to set it, how can they invoke the OS so that it can run with the mode bit set?
  - That’s what traps do ... set the mode bit and begin execution at a specific point in memory (in the OS!)
System calls

- *System calls* are the mechanism by which programs communicate with the O.S.
- Implemented via a TRAP instruction
- Example UNIX system calls:
  - open(), read(), write(), close()
  - kill(), signal()
  - fork(), wait(), exec(), getpid()
  - link(), unlink(), mount(), chdir()
  - setuid(), getuid(), chown()
The inner workings of a system call

User-level code

Process usercode
{
  ...
  read(file, buffer, n);
  ...
}

Library code

Procedure read(file, buff, n)
{
  ...
  read(file, buff, n)
  ...
}

_read:
  LOAD r1, @SP+2
  LOAD r2, @SP+4
  LOAD r3, @SP+6
  TRAP Read_Call
Steps in making a system call
What about disks and file storage?

Structure of a disk drive
Disks and file storage

- **Manipulating the disk device is complicated**
  - hide some of the complexity behind disk controller, disk device driver

- **Disk blocks are not a very user-friendly abstraction for storage**
  - contiguous allocation may be difficult for large data items
  - how do you manage administrative information?

- **One application should not (automatically) be able to access another application’s storage**
  - OS needs to provide a “file system”
File systems

File system - an abstraction above disk blocks
What about networks?

- Network interfaces are just another kind of shared device/resource

- Need to hide complexity
  - send and receive primitives, packets, interrupts etc
  - protocol layers

- Need to protect the device
  - access via the OS

- Need to allocate resources fairly
  - packet scheduling
The Process Concept

- **Process** – 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 (program instructions, static and dynamic data values)
    - CPU state (registers, PC, SP, etc)
    - operating system state (open files, accounting statistics etc)
Process address space

- Each process runs in its own virtual memory address space that consists of:
  - *Stack space* - used for function and system calls
  - *Data space* - variables (both initialized and uninitialized)
  - *Text* - the program code (usually read only)

- Invoking the same program multiple times results in the creation of multiple distinct address spaces
Running a process on a CPU

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

- Saving all the information about a process allows a process to be *temporarily suspended*
Switching among multiple processes

- Saving all the information about a process allows a process to be *temporarily suspended*.
Switching among multiple processes

- Saving all the information about a process allows a process to be *temporarily suspended*
Switching among multiple processes

- Saving all the information about a process allows a process to be *temporarily suspended*
Switching among multiple processes

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

```
Memory

Program Code

Program Data

CPU

ADD R1, R3

ALU

Program State

Program State

Program 1 has 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 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

<table>
<thead>
<tr>
<th>Processes</th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
<td>...</td>
<td>n-2</td>
<td>n-1</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Scheduler</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Process states

- Possible process states
  - 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

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

- Skeleton of what the lowest levels of the OS do when an interrupt occurs
How do processes get created?

Principal events that cause process creation

- System initialization
- Initiation of a batch job
- User request to create a new process
- Execution of a process creation system call from another 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 or process ID (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 a copy of a process with the lone 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();  
}  
...```

...
Process creation in UNIX example

```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)
```
Process creation in UNIX example

csh (pid = 22)

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

csh (pid = 24)

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

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();
    }
...
Process creation in UNIX example

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 count</td>
<td>Pointer to code 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>Session</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>System CPU time</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Time of next time</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?
What to do before next class

- Reading for next week’s class - pages 100-110
- Finish project 1 - Introduction to SPANK