CS 333
Introduction to Operating Systems

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

Jonathan Walpole
Computer Science
Portland State University
Lecture 2 overview

**OS-Related Hardware & Software**
- Complications in real systems
- Brief introduction to
  - memory protection and relocation
  - virtual memory & MMUs
  - I/O & Interrupts

**The “process” abstraction**
- Process scheduling
- Process states
- Process hierarchies
- Process system calls in Unix
Why its not quite that simple ...

- The basic model introduced in lecture 1 still applies, but the following issues tend to complicate implementation in real systems:
  - Pipelined CPUs
  - Superscalar CPUs
  - Multi-level memory hierarchies
  - Virtual memory
  - Complexity of devices and buses
Pipelined CPUs

Execution of current instruction performed in parallel with decode of next instruction and fetch of the one after that.
Superscalar CPUs
What does this mean for the OS?

- **Pipelined CPUs**
  - more complexity in taking a snapshot of the state of a running application
  - more expensive to suspend and resume applications

- **Superscalar CPUs**
  - even more complexity in capturing state of a running application
  - even more expensive to suspend and resume applications
  - support from hardware is useful i.e. precise interrupts

- More details, but fundamentally the same task
- **The BLITZ CPU is not pipelined or superscalar**
  - BLITZ has precise interrupts
The memory hierarchy

- **2GHz processor → 0.5 ns clock cycle**
- **Data/instruction cache access time → 0.5ns – 10 ns**
  - This is where the CPU looks first!
  - Memory this fast is **very expensive**!
  - Size ~64 kB- 1MB (too small for whole program)
- **Main memory access time → 60 ns**
  - Slow, but cheap
  - Size 512 MB – 1GB+
- **Magnetic disk → 10 ms, 200+ Gbytes**
Terminology review - metric units

<table>
<thead>
<tr>
<th>Exp.</th>
<th>Explicit</th>
<th>Prefix</th>
<th>Exp.</th>
<th>Explicit</th>
<th>Prefix</th>
</tr>
</thead>
<tbody>
<tr>
<td>$10^{-3}$</td>
<td>0.001</td>
<td>milli</td>
<td>$10^3$</td>
<td>1,000</td>
<td>Kilo</td>
</tr>
<tr>
<td>$10^{-6}$</td>
<td>0.000001</td>
<td>micro</td>
<td>$10^6$</td>
<td>1,000,000</td>
<td>Mega</td>
</tr>
<tr>
<td>$10^{-9}$</td>
<td>0.000000001</td>
<td>nano</td>
<td>$10^9$</td>
<td>1,000,000,000</td>
<td>Giga</td>
</tr>
<tr>
<td>$10^{-12}$</td>
<td>0.000000000001</td>
<td>pico</td>
<td>$10^{12}$</td>
<td>1,000,000,000,000</td>
<td>Tera</td>
</tr>
<tr>
<td>$10^{-15}$</td>
<td>0.000000000000001</td>
<td>femto</td>
<td>$10^{15}$</td>
<td>1,000,000,000,000,000,000</td>
<td>Peta</td>
</tr>
<tr>
<td>$10^{-18}$</td>
<td>0.0000000000000000001</td>
<td>atto</td>
<td>$10^{18}$</td>
<td>1,000,000,000,000,000,000,000</td>
<td>Exa</td>
</tr>
<tr>
<td>$10^{-21}$</td>
<td>0.000000000000000000001</td>
<td>zepto</td>
<td>$10^{21}$</td>
<td>1,000,000,000,000,000,000,000,000</td>
<td>Zetta</td>
</tr>
<tr>
<td>$10^{-24}$</td>
<td>0.00000000000000000000001</td>
<td>yocto</td>
<td>$10^{24}$</td>
<td>1,000,000,000,000,000,000,000,000,000</td>
<td>Yotta</td>
</tr>
</tbody>
</table>
Who manages the memory hierarchy?

- Movement of data from main memory to cache is under hardware control
  - cache lines loaded on demand automatically
  - Placement and replacement policy fixed by hardware
- Movement of data from cache to main memory can be affected by OS
  - instructions for “flushing” the cache
  - can be used to maintain consistency of main memory
- Movement of data among lower levels of the memory hierarchy is under direct control of the OS
  - virtual memory page faults
  - file system calls
OS implications of a memory hierarchy?

- How do you keep the contents of memory consistent across layers of the hierarchy?
- How do you allocate space at layers of the memory hierarchy “fairly” across different applications?
- How do you hide the latency of the slower subsystems?
  - Main memory?
  - Disk
- How do you protect one application’s area of memory from other applications?
- How do you relocate an application in memory?
  - How does the programmer know where the program will ultimately reside in memory?
Memory protection and relocation ...

- Memory protection – the basic ideas
  - virtual vs physical addresses
    - address range in each application starts at 0
  - Possible solution with base and limit registers
    - “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
Base & Limit Registers (single & multiple)

(a) 0xFFFFFFFF
- Base
- Limit
- User program and data
- Operating System

(b) Registers when program 2 is running
- Limit-2
- Base-2
- User-2 data
- Limit-1
- Base-1
- User-1 data
- User program
- Operating System
Memory protection and relocation ...

- **Memory protection - the basic ideas**
  - virtual vs physical addresses
    - address range in each application starts at 0
  - Possible solution with base and limit registers
    - "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)
Virtual memory and MMUs

- **Memory management unit (MMU)**
  - hardware provided equivalent of multiple base registers
  - at the granularity of “pages” of memory, say 2kB, i.e., lots of them!
  - supports relocation at page granularity by replacing high order address bits
  - 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 checking occur at CPU speed!**
What about I/O devices?

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

- Why protect access to devices by accessing them indirectly via the OS?

- Devices vs device controllers vs device drivers
  - device drivers are part of the OS (i.e., Software)
  - 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
  - Device controllers are hardware
  - They communicate with device drivers via interrupts
How do devices interact with programs?

- Interrupts
Different types of interrupts

- **Timer interrupts**
  - Allows OS to keep control after calling app’ code
  - One way to keep track of time

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

- **Hardware failures**

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

1. Push nbytes
2. Push &buffer
3. Push fd
4. Call read
5. Put code for read in register
6. Increment SP
7. Dispatch
8. Sys call handler
9. User program calling read
10. Library procedure read
11. Return to caller
What about disks and file storage?

Structure of a disk drive

Read/write head (1 per surface)
Direction of arm motion
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
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 static and dynamic allocation)
  - *Text* - the program code (usually read only)

- Invoking the same program multiple times results in the creation of multiple distinct address spaces
Switching among multiple processes

- Program instructions operate on operands in memory and (temporarily) in registers

Diagram:
- Memory: Prog1 Code, Prog1 Data, Prog2 Code, Prog2 Data, Prog2 State
- CPU: ALU, SP, PC
- Load A1, R1
- Load A2, R2
- Add R1, R2, R3
- Store R3, A3
- Prog1 has CPU
- Prog2 is suspended
Switching among multiple processes

- Saving all the information about a process allows a process to be *temporarily suspended* and later *resumed* from the same point.
Switching among multiple processes

- Saving all the information about a process allows a process to be \textit{temporarily suspended} and later \textit{resumed}.

![Diagram showing memory and CPU with Prog1 and Prog2 processes]
Switching among multiple processes

- Program instructions operate on operands in memory and in registers

```
Load A1, R1
Load A2, R2
Sub R1, R2, R3
Store R3, A3
...```

Prog2 has CPU
Prog1 is suspended
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
- Sequential processes only exist above that layer
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
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()` system calls allow processes to get their information
- Process creation
  - `fork()` system call creates a copy of a process and returns in both processes, but with a different return value
  - `exec()` replaces an address space with a new program
- Process termination, signaling
  - `signal()`, `kill()` system calls allow a process to be terminated or have specific signals sent to it
Example: process creation in UNIX

csh (pid = 22)

```
...  
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();
}
...

csh (pid = 24)

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

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

ls (pid = 24)

```c
//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>

Example 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 117-128
- Finish project 1 - Introduction to BLITZ