CS399 New Beginnings
Jonathan Walpole
OS-Related Hardware & Software

The Process Concept
Lecture 2 Overview

OS-Related Hardware & Software
- complications in real systems
- brief introduction to memory protection, relocation, virtual memory and I/O

The process abstraction
- process scheduling
- process states
- process hierarchies
- process-related system calls in Unix
Recap from Class 1

Why do we need a timer device?
Why do we need an interrupt mechanism?
Why do we need privileged instructions?
Why are system calls different to procedure calls?
How are system calls different to interrupts?
Why is memory protection necessary?
How can the OS switch among applications?
Real World Complexity

Our simple model applies, but real systems have more complexity

- 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

Fetch unit → Decode unit → Holding buffer → Execute unit

Fetch unit → Decode unit → Execute unit

Fetch unit → Decode unit → Execute unit
What does this mean for the OS?

Pipelined CPUs
- more complexity in capturing 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 and has precise interrupts
The memory hierarchy

2GHz processor $\rightarrow$ 0.5 ns clock cycle
Data/instruction cache access time $\rightarrow$ 0.5ns – 10 ns
This is where the CPU looks first!
Memory this fast is very expensive!
Size: too small for whole program

Main memory access time $\rightarrow$ 60 ns
Slow, but cheap
Size 1GB+

Magnetic disk $\rightarrow$
10 ms, 200+ Gbytes
Managing the Memory Hierarchy

Movement of data from main memory to cache
- Cache *lines* loaded on demand automatically
- Placement and replacement policy fixed by hardware

Movement of data from cache to main memory affected by OS
- Instructions for “flushing” the cache
- 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 and file system calls
Memory Hierarchy Challenges

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, such as main memory and disk?
Other Memory-Related Issues

How do you protect one application’s area of memory from that of another application?

How do you *relocate* an application in memory?

- How does the programmer know where the program will ultimately reside in memory?
Memory Protection & Relocation

Memory protection – the basic ideas

Virtual vs physical addresses
- Address range in each application starts at 0
- Applications use virtual addresses, but hardware and OS translate them automatically into physical addresses

Base register concept
- Get the CPU to interpret address indirectly via a base register
- Base register holds starting, or base address
- Add base value to address to get a real address before main memory is accessed

Relocation is simple
- Just change the base register value!
Paged virtual memory

The same basic concept, but ...
- Supports non-contiguous allocation of memory
- Allows processes to grow and shrink dynamically
- Requires hardware support for page-based address translation
  Sometimes referred to as a memory management unit (MMU) or a translation lookaside buffer (TLB)

Much more on this later ...
I/O Devices

A simplified view of a computer system
Structure of a Pentium System
Program-Device Interaction

Devices vs device controllers vs device drivers
- Device drivers are part of the OS (ie. software)
- Programs call the OS which calls the device driver
- Device drivers interact with device controllers using special IO instructions or by reading/writing controller registers that appear as memory locations
- Device controllers are hardware that communicate with device drivers via interrupts
Device to Program Interaction

Diagram:

1. CPU
2. Disk controller
3. Interrupt controller
4. Disk drive

Steps:
1. Interrupt
2. Dispatch to handler
3. Return

Current instruction:
Next instruction:

Interrupt handler
Types of Interrupt

Timer interrupts
- Allows OS to regain control of the CPU
- One way to keep track of time

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

Program generated (traps & faults)
- Address translation faults (page fault, TLB miss)
- Programming errors: seg. faults, divide by zero, etc.
- System calls like read(), write(), gettimeofday()
System calls

System calls are the mechanism by which programs invoke the OS
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()
System Call Implementation

User-level code

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

Library code

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

```asm
_read:
    LOAD r1, @SP+2
    LOAD r2, @SP+4
    LOAD r3, @SP+6
    TRAP Read_Call
```
Read(fd,nbytes,buffer) System Call
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
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 Processes

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

- Prog1 has CPU
- Prog2 is suspended
Switching Among Processes

Saving all the information about a process allows a process to be *temporarily suspended* and later *resumed*.
Switching Among Processes

Saving all the information about a process allows a process to be *temporarily suspended* and later *resumed*.
Switching Among Processes

Program instructions operate on operands in memory and in registers

Memory

- Prog1 Code
- Prog1 Data
- Prog1 State

Prog2 Code

Prog2 Data

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

CPU

ALU

SP  PC

Prog2 has CPU

Prog1 is suspended
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 Scheduler

Lowest layer of process-structured OS
- handles interrupts & scheduling of processes
Sequential processes only exist above that layer

<table>
<thead>
<tr>
<th>Processes</th>
<th>0</th>
<th>1</th>
<th>...</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>

33
Process States

1. Process blocks for input
2. Scheduler picks another process
3. Scheduler picks this process
4. Input becomes available
How Are Processes Created?

Events that cause process creation:
- System startup
- 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 process creates child process
  - each process is assigned a unique identifying number or process ID (PID)
  - system calls for communicating with and waiting for child processes

Child processes can create their own child processes
  - UNIX calls this hierarchy a "process group"
How Do Processes Terminate?

Conditions that 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 (parent and child), 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
Process Creation Example

csh (pid = 22)

...  

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

...
Process Creation Example

csh (pid = 22)

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

csh (pid = 24)

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

csh (pid = 22)

\[
\begin{align*}
\text{...} \\
\text{pid} &= \text{fork()} \\
\text{if (pid == 0) } \{ \\
\quad &\text{// child...} \\
\text{...} \\
\quad &\text{exec();} \\
\text{\}} \\
\text{else } \{ \\
\quad &\text{// parent} \\
\quad &\text{wait();} \\
\text{\}} \\
\text{...}
\end{align*}
\]

csh (pid = 24)

\[
\begin{align*}
\text{...} \\
\text{pid} &= \text{fork()} \\
\text{if (pid == 0) } \{ \\
\quad &\text{// child...} \\
\text{...} \\
\quad &\text{exec();} \\
\text{\}} \\
\text{else } \{ \\
\quad &\text{// parent} \\
\quad &\text{wait();} \\
\text{\}} \\
\text{...}
\end{align*}
\]
Process Creation Example

csh (pid = 22)

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

ls (pid = 24)

```
//ls program
main(){
   //look up dir
   ...
}
```
Process Creation (fork)

Fork creates a new process by *copying* the calling process.

The new process has its own:
- Memory address space (copied from parent)
  - Instructions (same program as parent!)
  - Data
  - Stack
- Register set (copied from parent)
- Process table entry in the OS
Before Next Class

Do the reading for next week’s class
Finish project 1 – Introduction to BLITZ