Chapter 6

Process Management

Last lecture review

- Von Neumann computer comprises of
  - CPU (alu + control unit)
  - Memory unit
  - Devices
  - Bus
- Boot-strapping
- Interrupts and interrupt handling
- Trap mechanism (more explanation today)

Requesting Service from OS

- Kernel functions are invoked by "trap"
- System call
  - Process traps to OS interrupt handler
  - Supervisor mode set
  - Desired function executed
  - User mode set
  - Returns to application

Requesting Svc: System Call

Steps in making a system call

- There are 11 steps in making the system call read (fd, buffer, nbytes)

Revisiting the trap Instruction (H/W)

executeTrap(argument) {
  setMode(supervisor);
  switch(argument) {
    case 1: PC = memory[1001]; // Trap handler 1
    case 2: PC = memory[1002]; // Trap handler 2
    .
    .
    .
    case n: PC = memory[1000+n]; // Trap handler n
  }
}

- The trap instruction dispatches a trap handler routine atomically
- Trap handler performs desired processing
- "A trap is a software interrupt"
Process Management

Process Management Tasks
- Define & implement the essential characteristics of a process and thread
  - Algorithms to define the behavior
  - Data structures to preserve the state of the execution
- Define what “things” threads in the process can reference – the address space (most of the “things” are memory locations)
- Manage the resources used by the processes/threads
- Tools to create/destroy/manipulate processes & threads

OS organization

Process management (...ctd)
- Tools to time- multiplex the CPU – Scheduling the (Chapter 7)
- Tools to allow threads to synchronize the operation with one another (Chapters 8-9)
- Mechanisms to handle deadlock (Chapter 10)

Introduction
- Scenario
  - One process running
  - One/more process performing I/O
  - One/more process waiting on resources
  - Most of the complexity stems from the need to manage multiple processes

Introduction
- Process Manager
  - CPU sharing
  - Process synchronization
  - Deadlock prevention
Process Manager Overview

- **Program**
- **Process**

Abstract Computing Environment

- **File Manager**
- **Device Manager**
- **Memory Manager**
- **Resource Manager**
- **Scheduler**
- **Process Description**

**Process Components**

- **Program**
  - defines behavior
- **Data**
- **Resources**
- **Process Descriptor**
  - keeps track of process during execution

Process Descriptor

<table>
<thead>
<tr>
<th>Field</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Internal program</td>
<td>The internal name of the program such as <em>n</em> in Chapter 9 of the book.</td>
</tr>
<tr>
<td>Owner</td>
<td>A process has an owner identified by the owner's process identifier.</td>
</tr>
<tr>
<td>Parent</td>
<td>A pointer to the previous descriptor of the parent process.</td>
</tr>
<tr>
<td>Sibling</td>
<td>A pointer to a list of sibling processes of this process.</td>
</tr>
<tr>
<td>Resource managers</td>
<td>A pointer to a list of resources managed by the process. Some resources may be mapped into a list.</td>
</tr>
<tr>
<td>Readable/Present</td>
<td>The descriptor is a handle to an element of the address space.</td>
</tr>
<tr>
<td>Allocatable</td>
<td>A description of the allocatable memory list.</td>
</tr>
<tr>
<td>Protection errors</td>
<td>A description of the protection errors incurred by the process.</td>
</tr>
<tr>
<td>CPU status registers</td>
<td>A list of bits indicating whether the CPU has entered the running status.</td>
</tr>
<tr>
<td>COP/COP registers</td>
<td>A copy of each of the COP/COP registers at the last time the process entered the running status.</td>
</tr>
</tbody>
</table>

Process Address Space

- Defines all aspects of process computation
- **Program**
- **Variables**
- ...
- Address space is generated/defined by translation

Creating an Executable Program

- Separate objects each relative to 0
- One large program
- Y = (X+Y)

Maps relative address space to physical memory location

- Generates separate object code modules
- Relocates modules one behind other
- Relocates addresses of all but first
- Relates external reference to library calls and external modules

Basic Memory Hierarchy

- Fastest
- Access Speed
- Primary Memory, $M_p$
- Secondary Memory, $M_s$
- Slowest
Basic Memory Hierarchy...
- At any point in the same program, element can be in
  - Secondary memory $M_2$
  - Primary memory $M_1$
  - Registers $M_0$

- Consistency is a Problem
  - $M_0 = M_1 + M_2$ (code vs data)
  - When does one make them consistent?
  - How?

Consistency Problem
- Scheduler switching out processes – Context Switch
- Is Instruction a Problem ???
  - NO
  - Instructions are never modified
  - Separate Instruction and Data space
  - Therefore, $M_0 = M_1 = M_2$

Sample Scenario...
- Suppose “MOV X Y” instruction is executed
  - $M_0 = M_1$

- On context switch, is all of a process’ memory flushed to $M_2$?
  - No, only on page swap

- Hence, $env_{process} = (M_0 + M_2) + (...)$

- Note:
  - Flushing of memory frees it up for incoming process
    $=>$ Page Swap

Consistency Problem...
- Is Data a Problem ???
  - YES
  - Variable temporarily stored in register has value added to it
  - Therefore, $M_0 = M_1$

- On context switch, all registers are saved
  - Therefore, current state is saved

Process States
- Focus on Resource Management & Process Management

- Recall also that part of the process environment is its state

Process States...
- When process enters ‘Ready’ state, it must compete for CPU. Memory has already been allocated
- Process has CPU
- Process requests resource that is immediately available => NO blocking
- Process requests resource that is NOT yet available
- Resource allocated, memory re-allocated?
Resources & Resource Manager

- 2 types of Resources
  - Reusable (Memory)
  - Consumable (Input/Time slice)

![Diagram showing resource management process]

Process requesting resource unit(s)
- Get it, or
- Block => Stay in Queue

Units of Resource R

Resource Descriptor

- Each Resource R has a Resource Descriptor associated with it (similar to the process)
  - there is a "Status" for that Resource, and
  - a Resource Manager to manage it

<table>
<thead>
<tr>
<th>Field</th>
<th>Description</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Internal resource name</td>
<td>Name of the resource requested</td>
<td></td>
</tr>
<tr>
<td>Total size</td>
<td>Number of units of the resource</td>
<td></td>
</tr>
<tr>
<td>Available units</td>
<td>Number of units available for use</td>
<td></td>
</tr>
<tr>
<td>List of available units</td>
<td>List of units available for use</td>
<td></td>
</tr>
<tr>
<td>List of blocked processes</td>
<td>List of processes blocked</td>
<td></td>
</tr>
</tbody>
</table>

Creating Processes

- Parent Process needs ability to
  - Block child
  - Activate child
  - Destroy child
  - Allocate resources to child

- True for User processes spawning child
- True for OS spawning `init, getty`, etc.
- Process hierarchy a natural,
  if `fork/exec` commands exist

UNIX fork command

- ForkUNIX
  - Shares text
  - Shares memory
  - Has its own address space
  - Cannot communicate with parent by referring variable stored in code

- Earlier definition: Fork:Conway
  - Shares text
  - Shares resources
  - Shares address space
  - Process can communicate thru variables declared in code

Cooperating Processes

```c
Prog

x, y : INT
Proc A
  while(TRUE) {
    compute section A1;
    output(x);
  }
Proc B
  while(TRUE) {
    compute section B1;
    output(y);
  }
Fork "A"
Fork "B"
```

Now processes A & B, share address space & can communicate thru declared variables

**Problem: ??**

A can write 2 times before B reads
Synchronizing Access to Shared Variables

- Shared address space allows communication through declared variables automatically
- How then, can we synchronize access to them?
- Need Synchronization Primitives

=> JOIN & QUIT

---

Fork, Join & Quit - Conway

- In addition to the "Fork(proc)" command, Conway also defined system calls to support process synchronization
- Join (count)
  - Un-interruptible
    - Decrement count:
      - if count ≠ 0 then Quit, else Continue
- Quit
  - Terminate process

---

Fork, Join, Quit example

```c
L0: count = 2;
    Compute A1;
    write(x);
    FORK(L1);
    Compute A2;
L1: JOIN(count);
    read(y);
    QUIT();
L2: read(x);
    Compute B1;
    write(y);
    FORK(L3);
    Compute B2;
    goto L3;
```

---

A Simple Parent Program (Revisit)

```c
// include <sys/socket.h>

#include <sys/socket.h>

#define MSG 0

int main(void)
{
    int s, i;
    if (sockopen(0, 0, MSG) < 0)
        /* This is the child process */
        return(1);
    printf("Child will die:");
    goto L1;
    /* Should never get here, terminate */
}

/* Parent code here */

fork()
```

---

Spawning A Child Different From Parent

- Suppose we wish to spawn a child that is different from the parent
  ```
  fork
  execve(…)
  ```
- OS ➔ init ➔ getty ➔ shell

```
```

---

Factoring in additional Control Complexities

- Recall:
  - A parent process can suspend a child process
- Therefore, if a child is in run state and goes to ready (time slice up), and the parent runs and decides to suspend the child, then how do we reflect this in the process state diagram ???
- We need 2 more states
  - Ready suspended
  - Blocked suspended
Give it a thought...

Why can a process NOT go from 'Ready Active' to 'Blocked Active'
or 'Blocked Suspended'?