Chapter 6

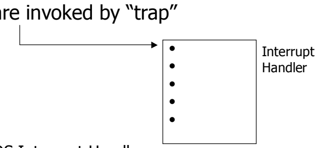
Process Management

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)

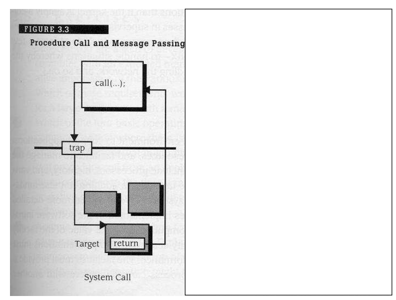
Requesting Service from OS

- 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



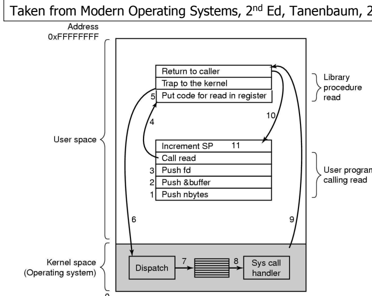
Revisiting the `trap` Instruction (H/W)

```
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"

Steps in making a system call

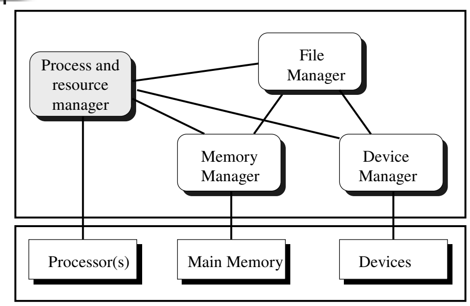
Taken from Modern Operating Systems, 2nd Ed, Tanenbaum, 2001



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

Process Management

OS organization



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

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)

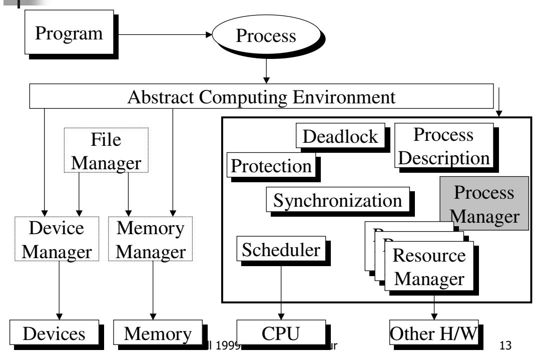
Introduction

- 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



Process components

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

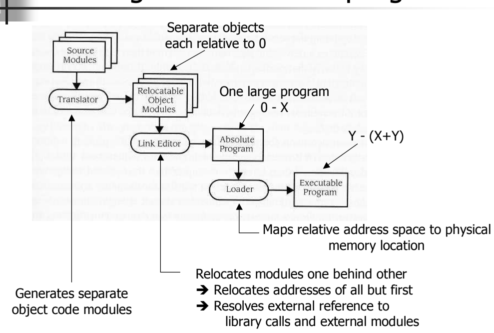
Process Descriptor

FIELD	DESCRIPTION
Internal process name	An internal name of the process, such as an integer or table index, used in the operating system code.
State	The process's current state.
Owner	A process has an owner (identified by the owner's internal identification such as the login name). The descriptor contains a field for storing the owner identification.
Parent process descriptor	A pointer to the process descriptor of this process's parent.
List of child process descriptors	A pointer to a list of the child processes of this process.
List of reusable resources	A pointer to a list of reusable resource types held by the process. Each resource type will be a descriptor of the number of units of the resource.
List of consumable resources	Similar to the reusable resource list (see Section 6.3.2).
List of file descriptors	A special case of the reusable resource list.
Message queue	A special case of the consumable resource list.
Protection domain	A description of the access rights currently held by the process (see Chapter 14).
CPU status register content	A copy of each of the CPU status registers at the last time the process exited the running state.
CPU general register content	A copy of each of the CPU general registers at the last time the process exited the running state.

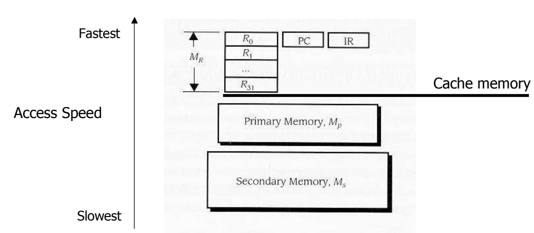
Process Address Space

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

Creating an executable program



Basic Memory Hierarchy



Basic Memory Hierarchy…

- At any point in the same program, element can be in
  - Secondary memory M_S
  - Primary memory M_P
  - Registers M_R
- Consistency is a Problem
  - $M_S \neq M_P \neq M_R$ (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, $M_{R_j} = M_{P_j} = M_{S_j}$

How can an instruction be in a register ?

Consistency Problem…

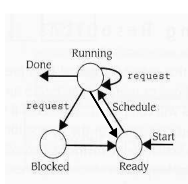
- Is Data a Problem ???
  - YES
  - Variable temporarily stored in register has value added to it
  - Therefore, $M_{R_j} \neq M_{P_j}$
- On context switch, all registers are saved
  - Therefore, current state is saved

Sample Scenario…

- Suppose ‘MOV X Y’ instruction is executed
  - → $M_{P_y} \neq M_{S_y}$
- On context switch, is all of a process’ memory flushed to M_S ?
  - No, only on page swap
- Hence, $env_{process} = (M_R + M_S) + (\ldots)$
- Note:
  - Flushing of memory frees it up for incoming process
    => Page Swap

Process States

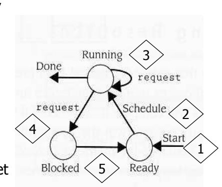
- Focus on Resource Management & Process Management
- Recall also that part of the process environment is its ***state***



State Transition Diagram

Process States…

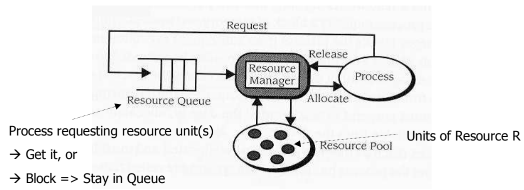
1. When process enters ‘Ready’ state, it must compete for CPU. Memory has already been allocated
2. Process has CPU
3. Process requests resource that is immediately available →NO blocking
4. Process requests resource that is NOT yet available
5. Resource allocated, memory re-allocated?



State Transition Diagram

Resources & Resource Manager

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



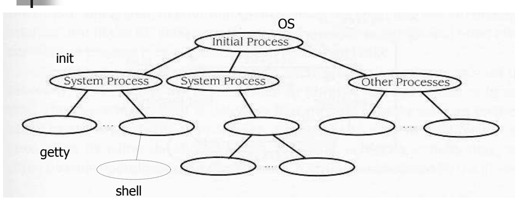
Resource Descriptor

- 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

Parts of a Resource Descriptor

FIELD	DESCRIPTION
Internal resource name	An internal name for the resource used by the operating system code. **/dev/...**
Total units	The number of units of this resource type configured into the system. **6**
* Available units	The number of units currently available. **3**
List of available units	The set of available units of this resource type that are available for use by processes. **A, B, C**
List of blocked processes	The list of processes that have a pending request for units of this resource type. **Only if * = 0**

Process Hierarchy



- Conceptually, this is the way in which we would like to view it
- Root controls all processes i.e. Parent

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

- `Fork`UNIX
  - 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

Prog
x, y : int
Proc A
ref x & y
Proc B
ref x & y
Fork "A"
Fork "B"

```
proc_A(){
    while(TRUE) {
        <compute section A1>;
        update(x);
        <compute section A2>;
        retrieve(y);
    }
}

proc_B(){
    while(TRUE) {
        retrieve(x);
        <compute section B1>;
        update(y);
        <compute section B2>;
    }
}
```

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 <u>automatically</u>
- How then, can we synchronize access to them?
- Need Sychronization Primitives

  => JOIN & QUIT

Prog
x, y : int
Porc A
  ref x & y
Proc B
  ref x & y
Fork "A"
Fork "B"

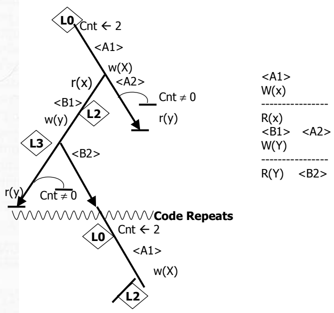
Fork, Join & Quit - Conway

- In addition to the "Fork(proc)" command, Conway also defined system calls to support process synchronization
- Join (count)
  - <u>Un-interruptable</u>

    Decrement count;
    if count ≠ 0 then Quit, else Continue
- Quit
  - Terminate process

Fork, Join, Quit example

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



A Simple Parent Program (Revisit)

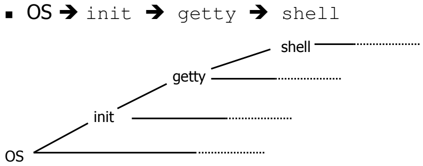
```
#include        <sys/wait.h>
#define NULL    0

int main (void){
    if (fork() = = 0){   /* This is the child process */
        execve("child",NULL,NULL);
        exit(0);           /* Should never get here, terminate */
    }
/* Parent code here */
    printf("Process[%d]: Parent in execution ...\n", getpid());
    sleep(2);
    if(wait(NULL) > 0) /* Child terminating */
        printf("Process[%d]: Parent detects terminating child \n",
                            getpid());
    printf("Process[%d]: Parent terminating ...\n", getpid());
}
```


Spawning A Child Different From Parent

- Suppose we wish to spawn a child that is <u>different</u> from the parent

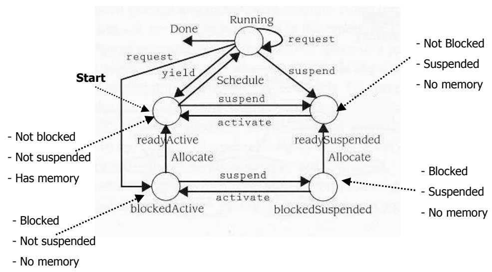
  `fork`
  `execve(…)`
- OS ➔ `init` ➔ `getty` ➔ `shell`



Factoring in additional Control Complexities

- Recall:
  - A parent process can <u>suspend</u> a child process
- Therefore, if a child is in <u>run</u> 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

Process State diagram reflecting Control



Give it a thought…

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