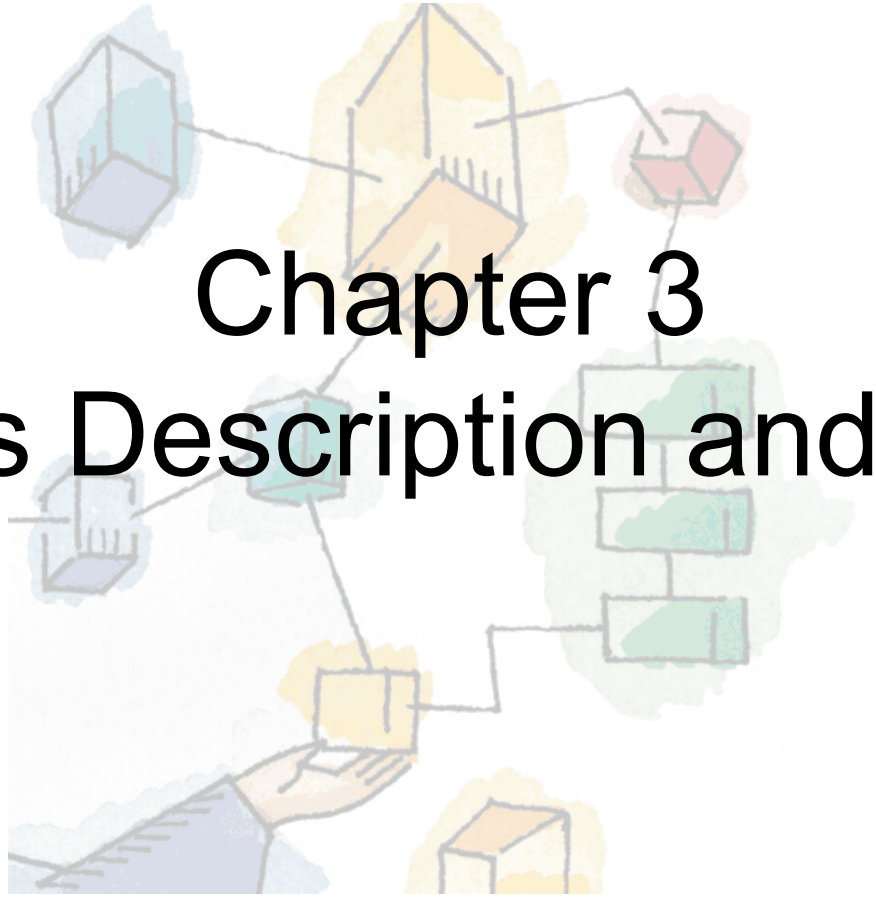
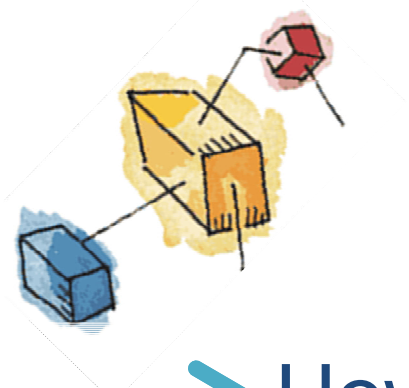
Operating Systems:
Internals and Design Principles
William Stallings

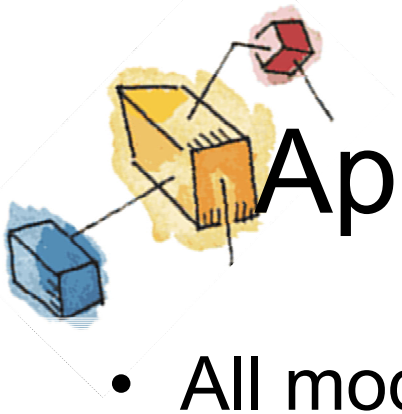


Chapter 3
Process Description and Control



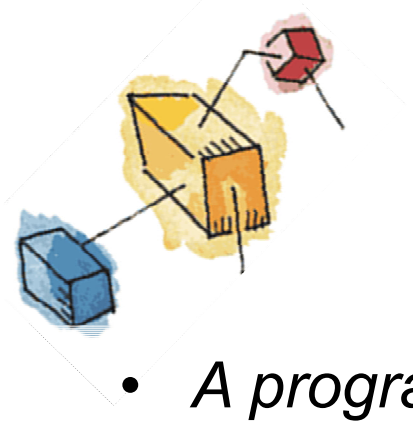
Roadmap

- How are processes represented and controlled by the OS?
  - A flavour of creating a process in UNIX
  - ***Process states*** which characterize the behaviour of processes
  - ***Data structures*** used to manage processes
  - Ways in which the OS uses these data structures to control process execution
  - Discuss process management in UNIX



Applications vs. Processes

- All modern OS rely on a model in which the execution of an application corresponds to the existence of one or more processes.
- Example: single-user systems such as Windows and mainframe system such as IBM's mainframe OS, z/OS, are built around the ***concept of process***.



What is a *"process"*?

- *A program in execution*
- An instance of a program running on a computer
- The entity that can be assigned to and executed on a processor
- A unit of activity characterized by the execution of a sequence of instructions, a current state, and an associated set of system resources

Process Elements

- A process is comprised of:
  - Program code
    - which may be shared with other processes that are executing the same program
  - A set of data associated with that code
  - Execution context of the program, containing all information the OS needs to manage the process
- When the processor begins to execute the program code, we refer to this executing entity as a ***process***.

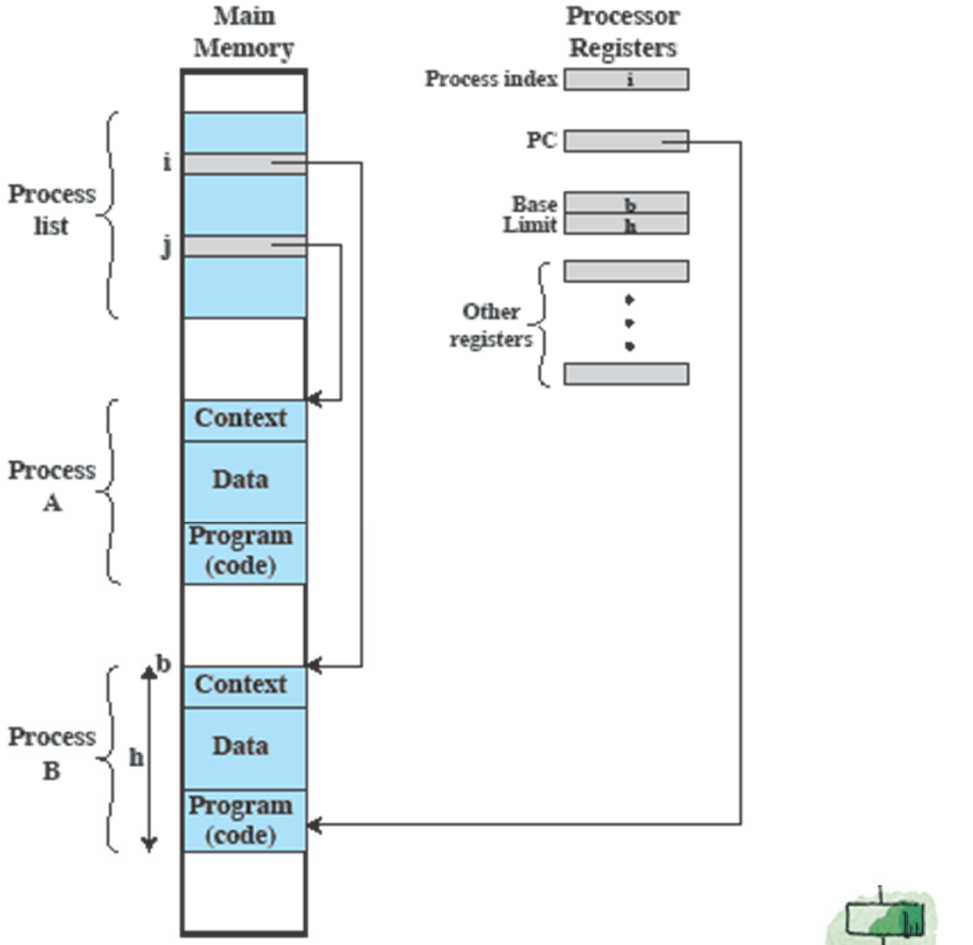
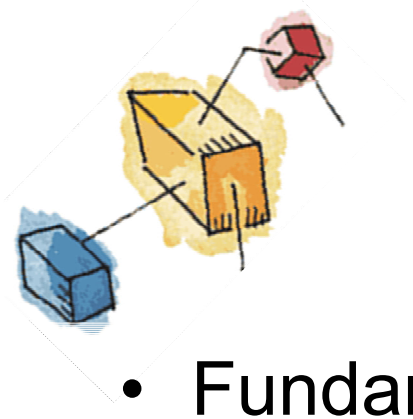
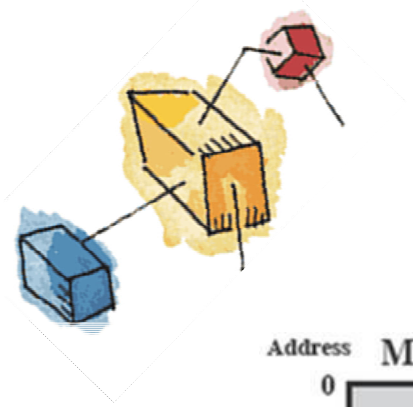


Figure 2.8 Typical Process Implementation

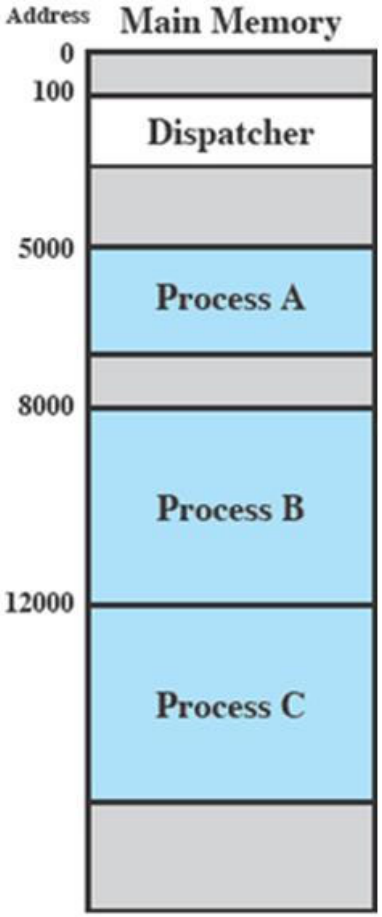


Requirements of an Operating System

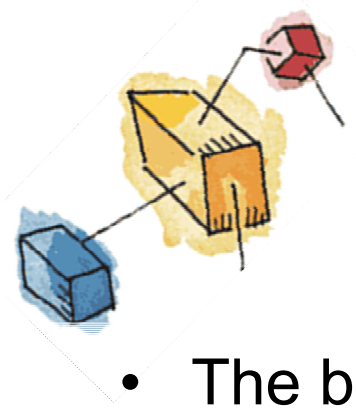
- Fundamental Task: ***Process Management***
- The Operating System must
  - Interleave the execution of multiple processes
  - Allocate resources to processes and protect the resources of each process from other processes
  - Enable processes to share and exchange information
  - Enable synchronization among processes



Example: Interleaved Execution of Processes



- Consider three processes being executed
- Plus a ***dispatcher*** - a small program which switches the processor from one process to another
- All are in memory



Trace from Processes' Point of View

- The behavior of an individual process can be characterized by listing the sequence of instructions that execute for that process: a *trace* of the process.

(a) Trace of Process A	(b) Trace of Process B	(c) Trace of Process C
5000	8000	12000
5001	8001	12001
5002	8002	12002
5003	8003	12003
5004		12004
5005		12005
5006		12006
5007		12007
5008		12008
5009		12009
5010		12010
5011		12011

5000 = Starting address of program of Process A
8000 = Starting address of program of Process B
12000 = Starting address of program of Process C

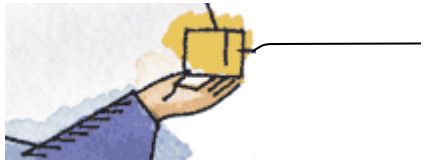
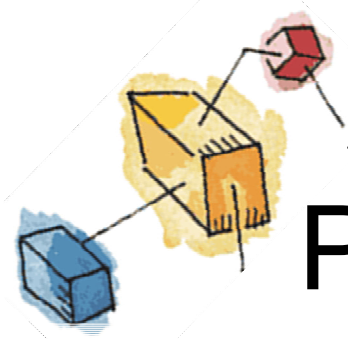
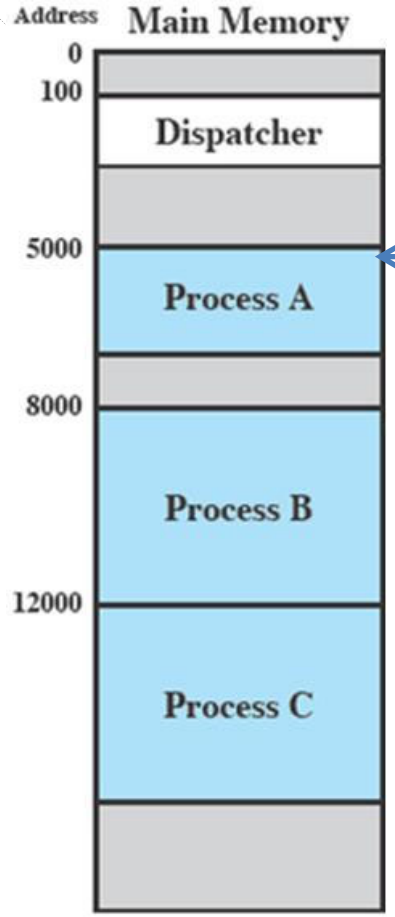


Figure 3.3 Traces of Processes of Figure 3.2

A Combined Trace from Processor's Point of View





Timeout

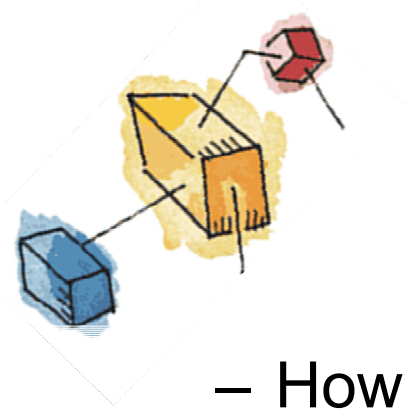
The behavior of the processor can be characterized by showing how the traces of the various processes are interleaved.

1	5000	A
2	5001	
3	5002	
4	5003	
5	5004	
6	5005	
	Timeout	
7	100	D
8	101	
9	102	
10	103	
11	104	
12	105	
13	8000	B
14	8001	
15	8002	
16	8003	
	I/O Request	
17	100	D
18	101	
19	102	
20	103	
21	104	
22	105	
23	12000	C
24	12001	
25	12002	
26	12003	
27	12004	
28	12005	
	Timeout	
29	100	D
30	101	
31	102	
32	103	
33	104	
34	105	
35	5006	A
36	5007	
37	5008	
38	5009	
39	5010	
40	5011	
	Timeout	
41	100	D
42	101	
43	102	
44	103	
45	104	
46	105	
47	12006	C
48	12007	
49	12008	
50	12009	
51	12010	
52	12011	
	Timeout	

100 = Starting address of dispatcher program

Shaded areas indicate execution of dispatcher process;
first and third columns count instruction cycles;
second and fourth columns show address of instruction being executed

Figure 3.4 Combined Trace of Processes of Figure 3.2

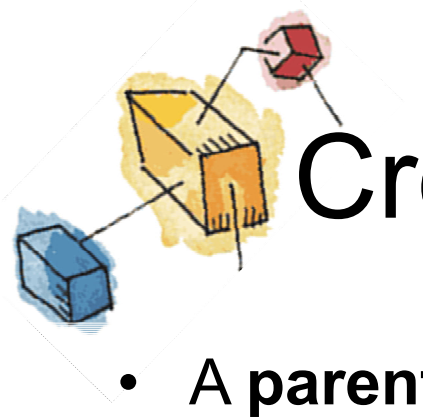


Roadmap

- How are processes represented and controlled by the OS?

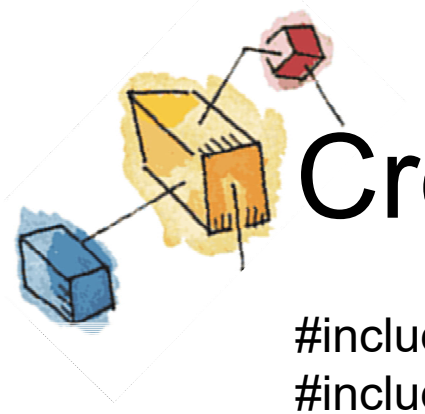
→ A flavour of creating a process in UNIX

- ***Process states*** which characterize the behaviour of processes
- ***Data structures*** used to manage processes
- Ways in which the OS uses these data structures to control process execution
- Discuss process management in UNIX



Creating Process in UNIX

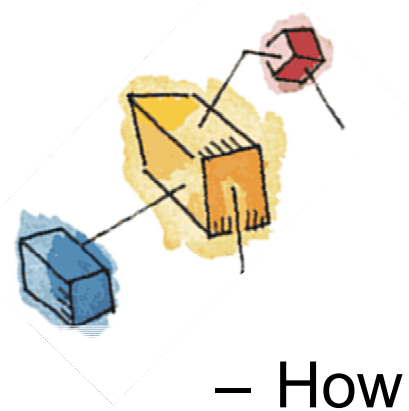
- A **parent** process can create a **child** process by means of the ***system call***, **`fork( )`**.
- After creating the process, the OS can do one of the following, as part of the dispatcher routine:
  - Stay in the parent process
    - Control returns to the point of the fork call of the parent
  - Transfer control to the child process
    - The child process begins executing at the same point in the code as the parent, namely at the return from the fork call
  - Transfer control to another process



Creating Process in UNIX

```
#include <iostream>
#include <unistd.h>
using namespace std;

int x = 10;

int main(void)
{   int pid;
    pid = fork(); /* create a new process */
    cout << "(1) x = " << x << " in " << pid << endl;
    if ( pid == 0 ) {
        cout << "(2) x = " << x << " in " << pid << endl;
        x = x + 5;
        cout << "(3) x = " << x << " in " << pid << endl;
    }
    cout << "(4) x = " << x << " in " << pid << endl;
}
```

What is possible output sequence of the program?

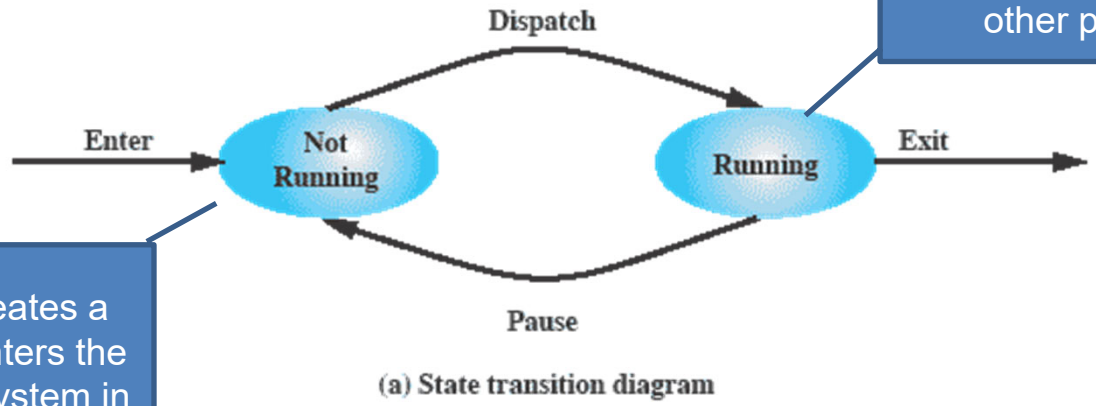


Roadmap

- How are processes represented and controlled by the OS?
- A flavour of creating a process in UNIX
- ***Process states*** which characterize the behaviour of processes
- ***Data structures*** used to manage processes.
- Ways in which the OS uses these data structures to control process execution
- Discuss process management in UNIX

Two-State Process Model

- A process is either being executed by a processor or not, so this is the simplest possible model: a process may be in one of two states
  - Running
  - Not-running



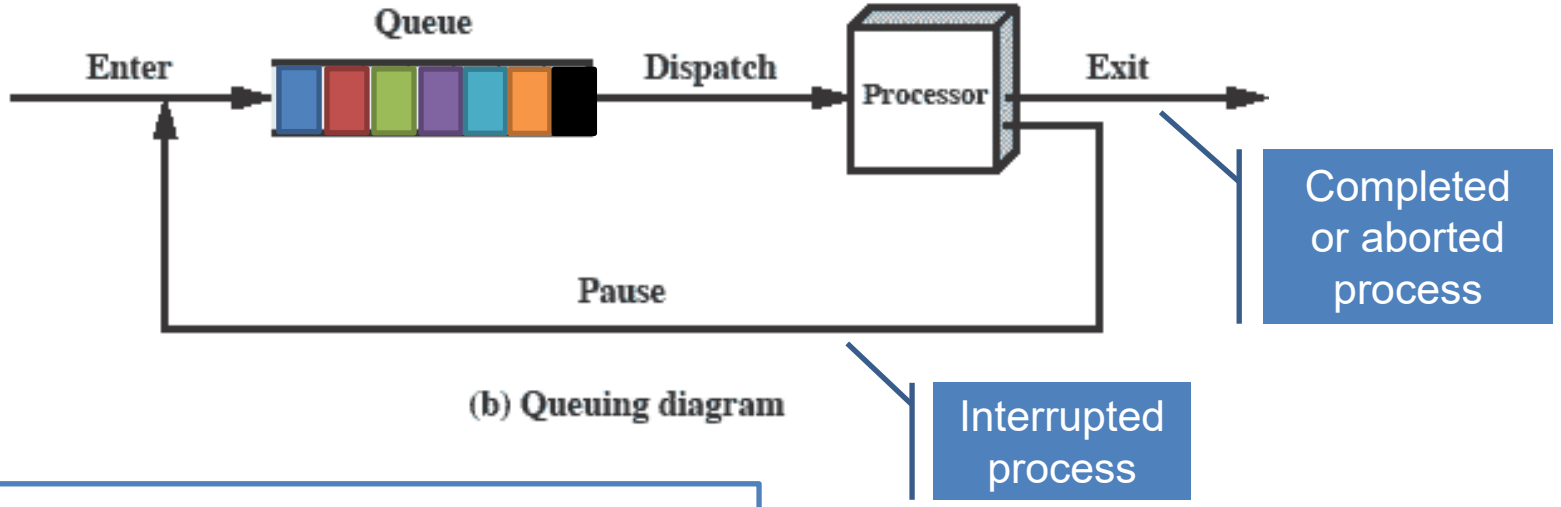
(a) State transition diagram

From time to time, the currently running process may be blocked or interrupted and the dispatcher will select some other process to run.

When the OS creates a new process, it enters the process into the system in the Not Running state.

Queuing Diagram

- Processes that are not running are kept in some sort of queue, waiting for their turn to execute.



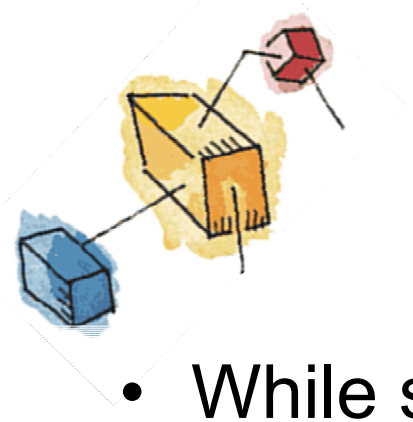
(b) Queuing diagram

Processes moved by the dispatcher to the CPU then back to the queue until the process has completed or aborted.

Process Birth and Death

Creation	Termination
New batch job	Normal Completion
Interactive log-on	Memory unavailable
Created by OS to provide a service	Bounds violation
Spawned by existing process	Protection error
	Arithmetic error
	Operator or OS Intervention
	Parent termination
	Parent request

See tables 3.1 and 3.2 for more



Five-State Process Model

- While some processes in the Not Running state are ready to execute, others may be blocked (e.g., waiting for an I/O operation to complete).

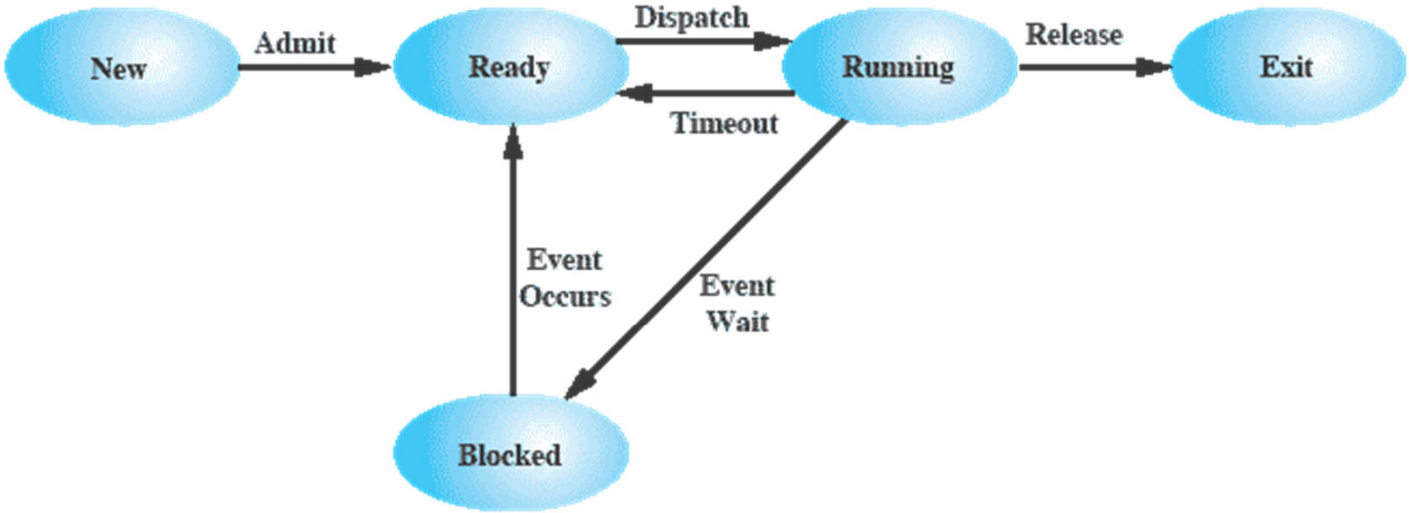
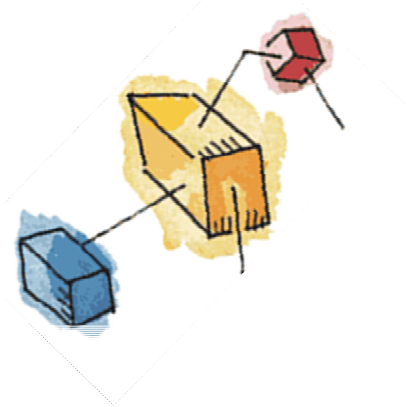
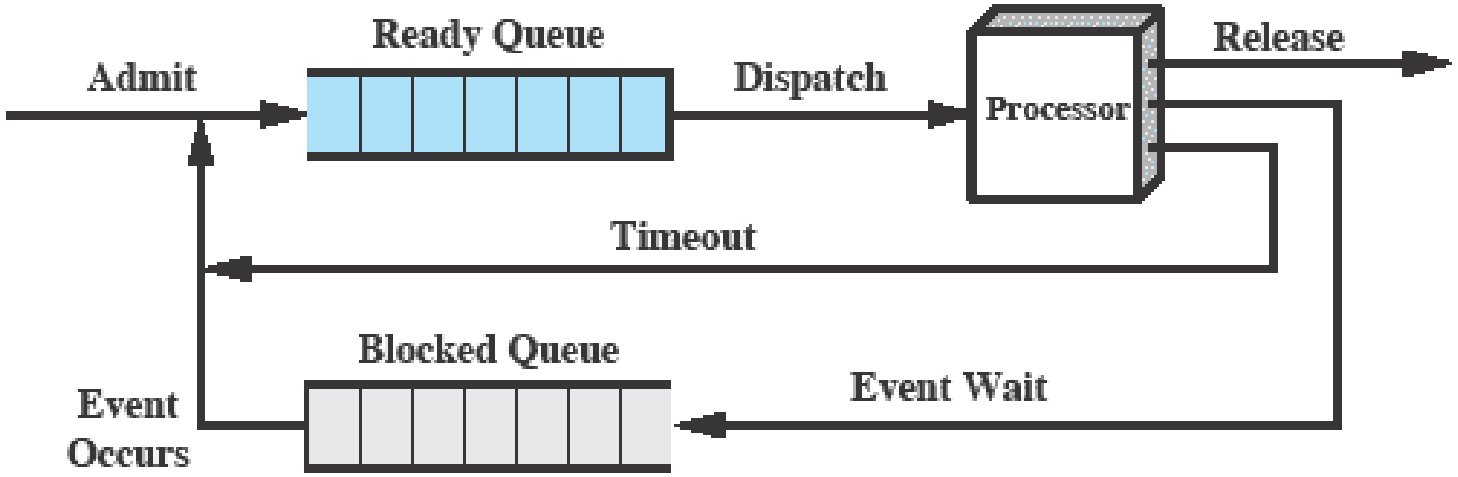


Figure 3.6 Five-State Process Model

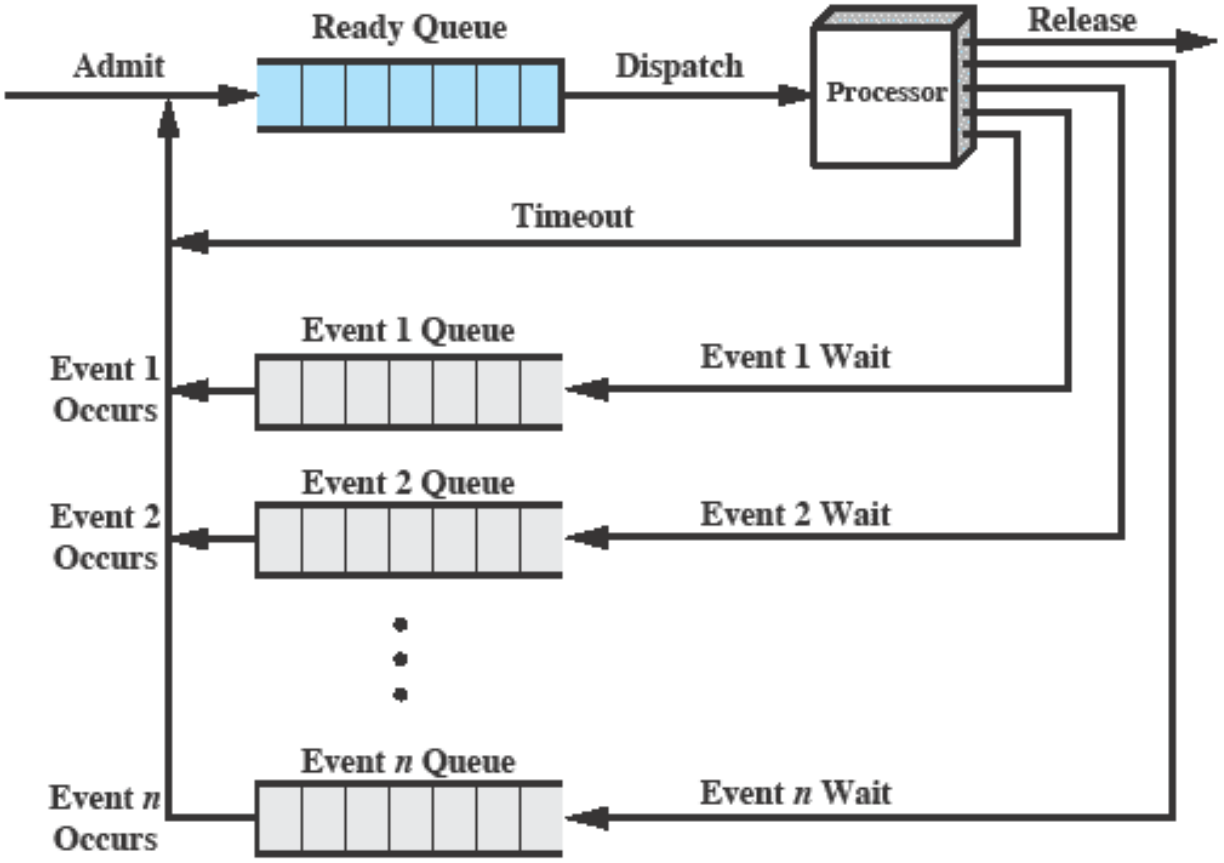


Using Two Queues

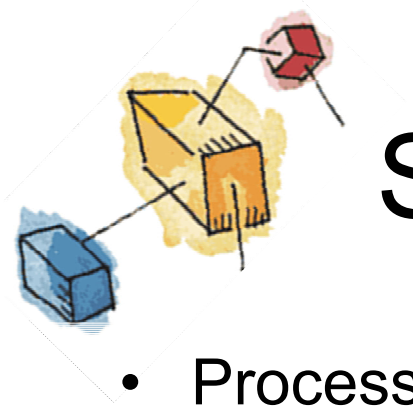


(a) Single blocked queue

Multiple Blocked Queues



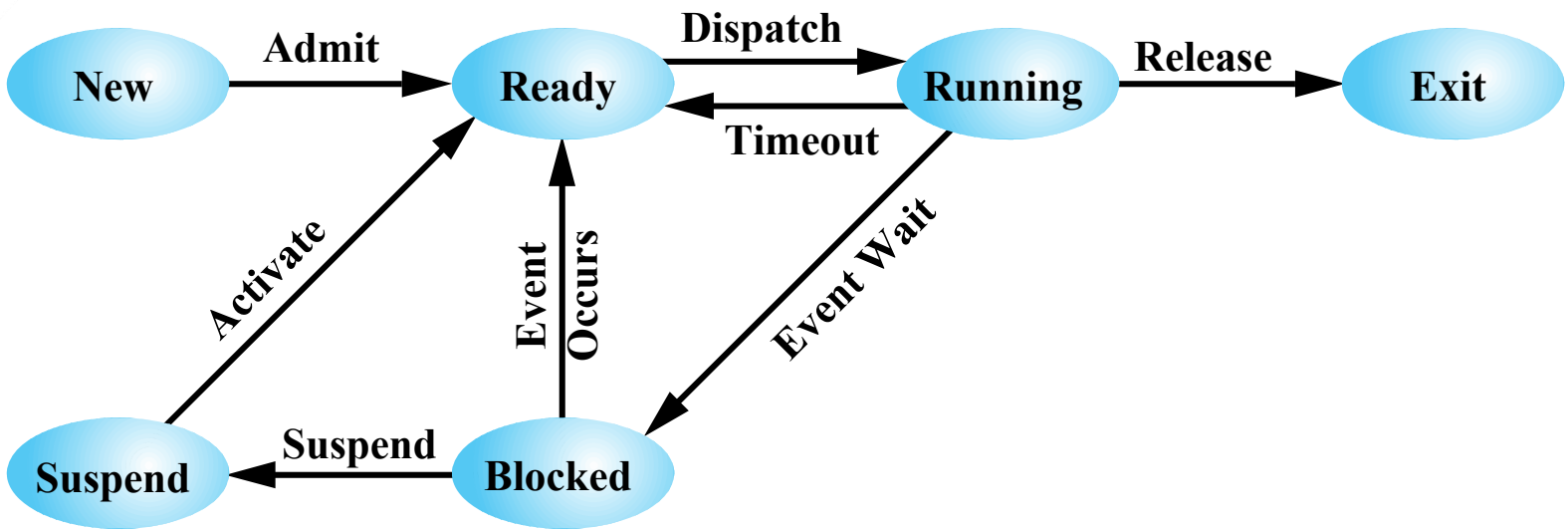
(b) Multiple blocked queues



Suspended Processes

- Processor is faster than I/O so ***all*** processes could be waiting for I/O → processor could be idle most of the time.
  - OS swaps one of the blocked processes out on to disk into a suspend queue to free up more memory.
  - The OS then brings in another process.
  - Execution then continues with the newly arrived process.
- Blocked state becomes ***Suspend*** state when swapped to disk

One Suspend State

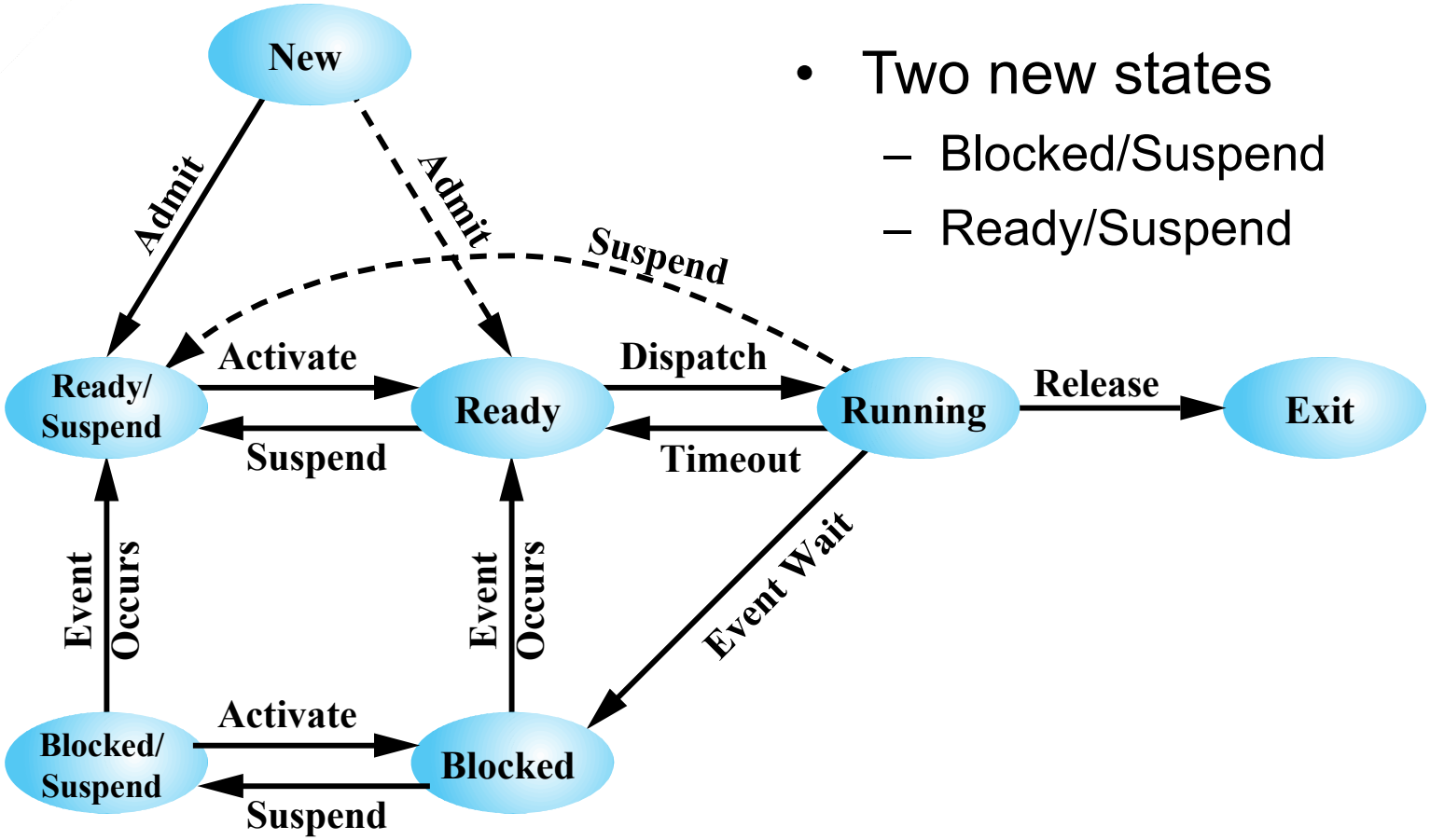


(a) With One Suspend State

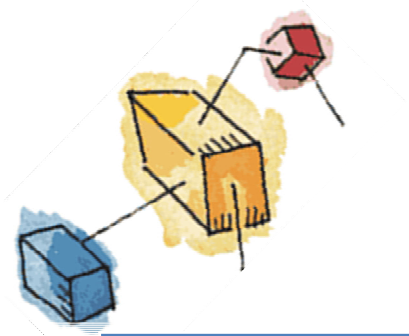
- Although each process in the Suspend state was originally blocked on a particular event, when that event occurs, the process is not blocked and is potentially available for execution.

Two Suspend States

- Two new states
  - Blocked/Suspend
  - Ready/Suspend



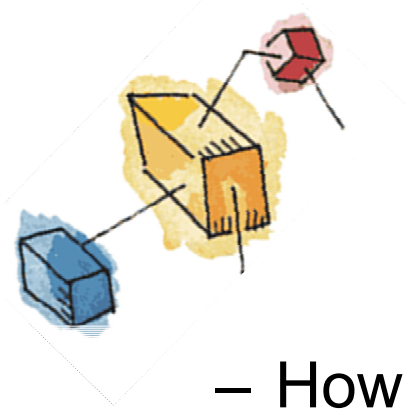
(b) With Two Suspend States



Reason for Process Suspension

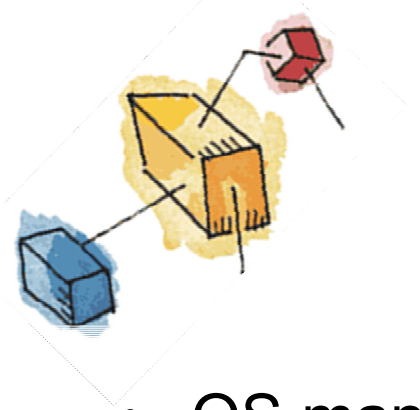
Reason	Comment
Swapping	The OS needs to release sufficient main memory to bring in a process that is ready to execute.
Other OS Reason	OS suspects process of causing a problem.
Interactive User Request	User may wish to suspend execution of a program for purpose of debugging
Timing	A process may be executed periodically (e.g., an accounting or system monitoring process) and may be suspended while waiting for the next time.
Parent Process Request	A parent process may wish to suspend execution of a descendent to examine or modify the suspended process, or to coordinate the activity of various descendants.

Table 3.3 Reasons for Process Suspension



Roadmap

- How are processes represented and controlled by the OS?
- A flavour of creating a process in UNIX
- ***Process states*** which characterize the behaviour of processes
- ***Data structures*** used to manage processes
- Ways in which the OS uses these data structures to control process execution
- Discuss process management in UNIX



Processes and Resources

- OS manages the use of system resources by processes.

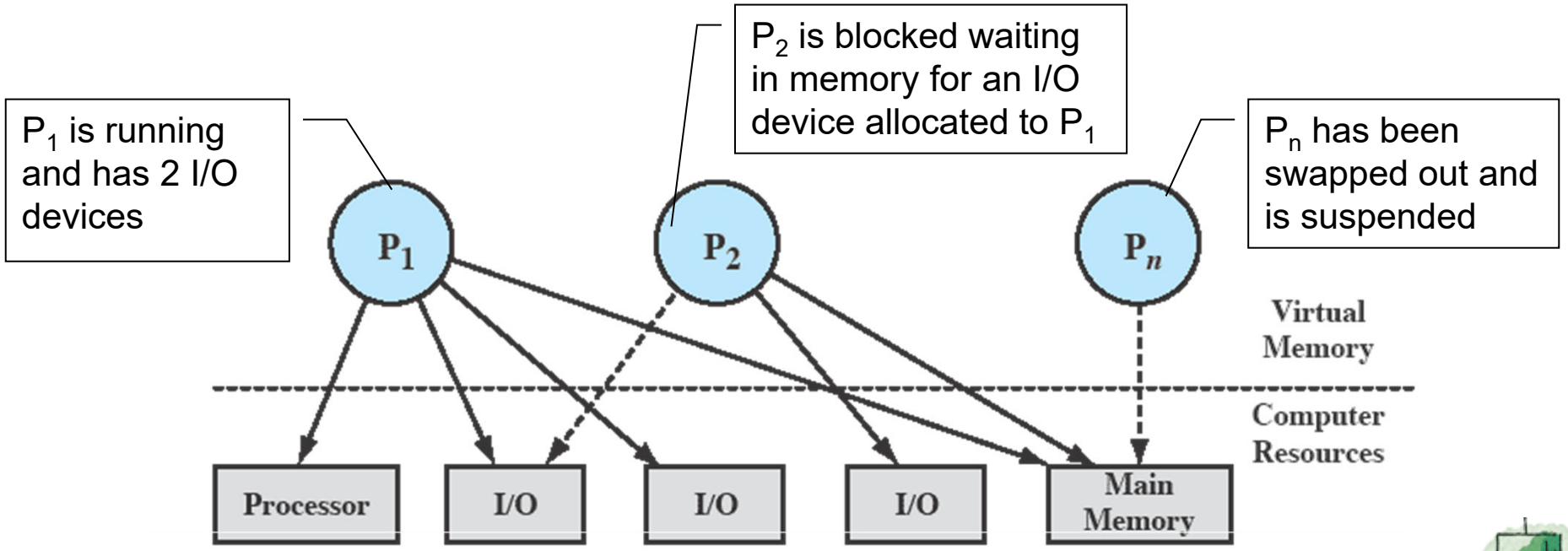
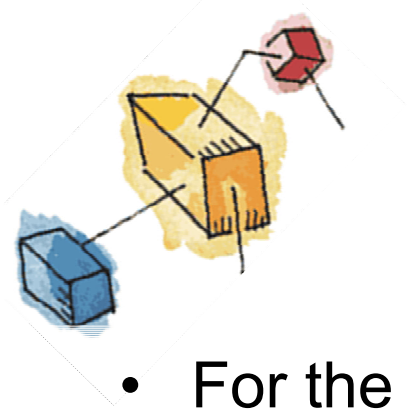


Figure 3.10 Processes and Resources (resource allocation at one snapshot in time)



Operating System Control Structures

- For the OS to manage processes and resources, it must have information about the current status of each process and resource.
- Tables are constructed for each entity the OS manages.
- What information does the OS need to control processes and manage resources for them?

OS Control Tables

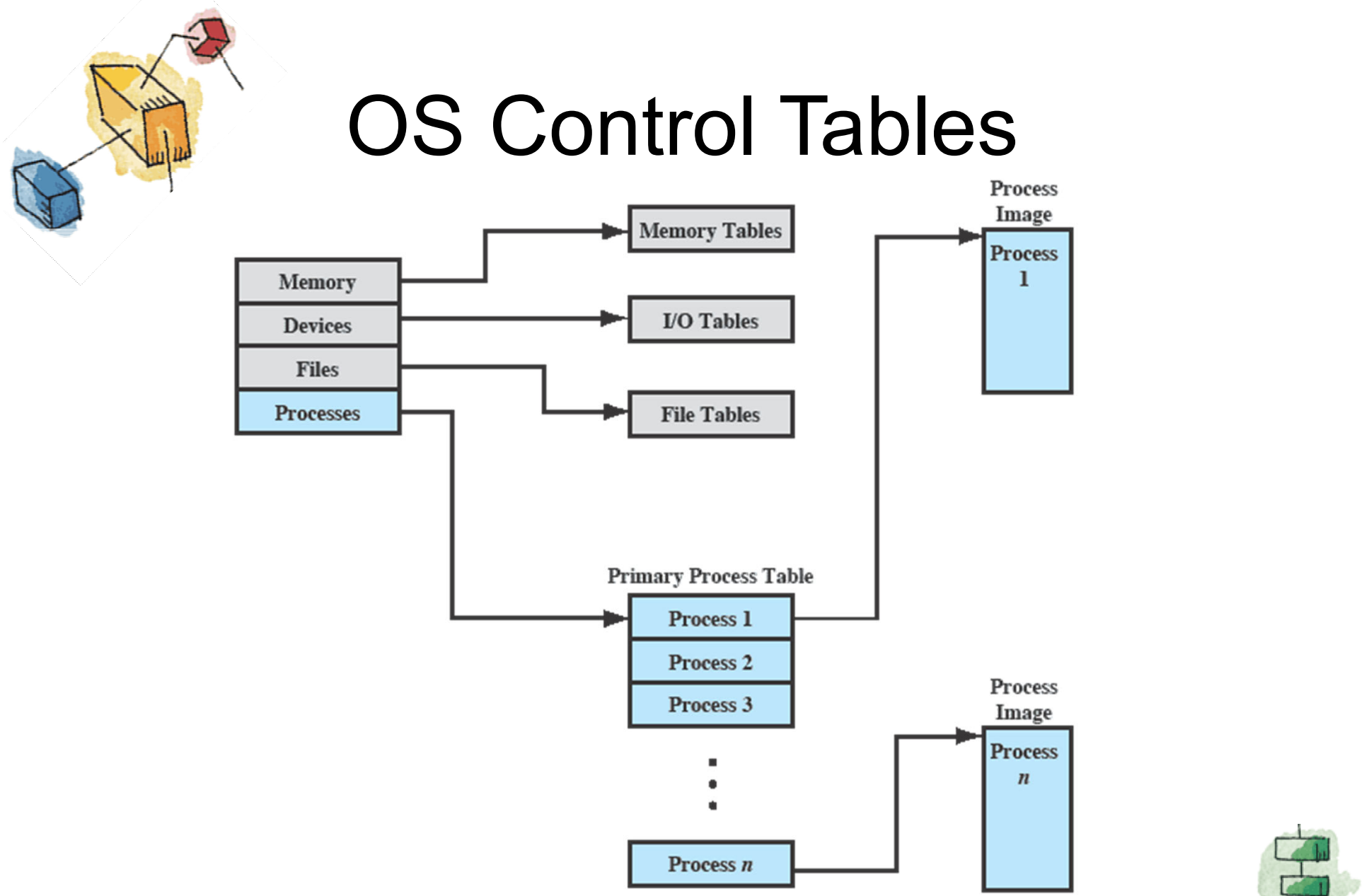
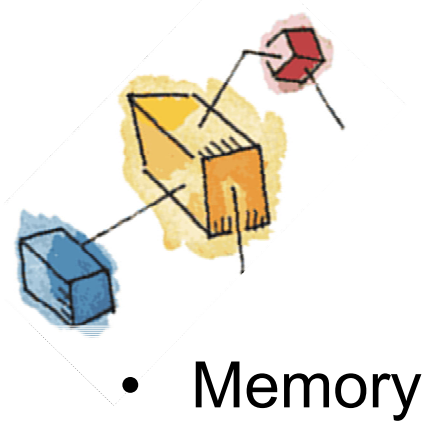
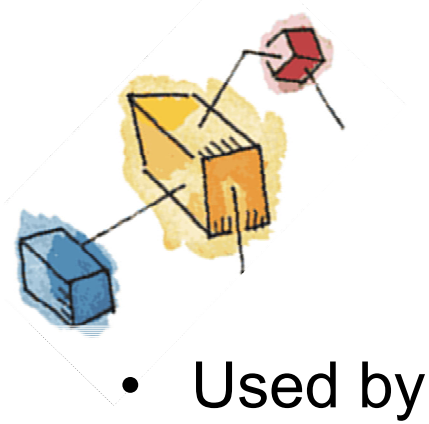


Figure 3.11 General Structure of Operating System Control Tables



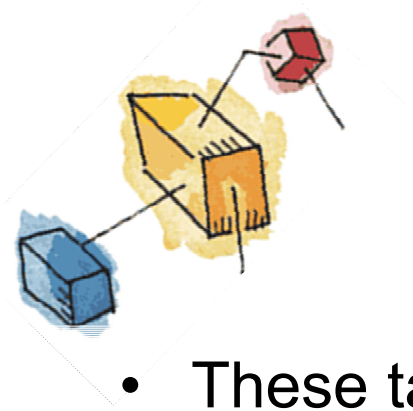
Memory Tables

- Memory tables are used to keep track of both main (real) and secondary (virtual) memory.
- Must include this information:
  - Allocation of main memory to processes
  - Allocation of secondary memory to processes
  - Protection attributes of blocks of main or virtual memory such as which processes may access certain shared memory regions
  - Information needed to manage virtual memory



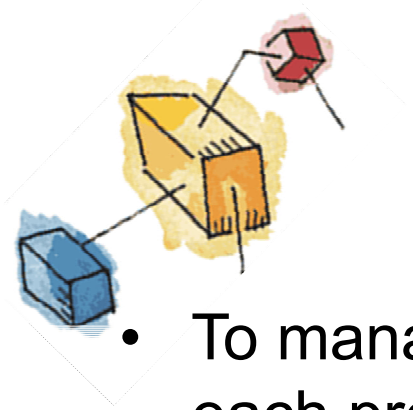
I/O Tables

- Used by the OS to manage the I/O devices and channels of the computer.
- At any given time, an I/O device may be available or assigned to a particular process.
- If an I/O operation is in progress, the OS needs to know:
  - The status of the I/O operation
  - The location in main memory being used as the source or destination of the I/O transfer



File Tables

- These tables provide information about:
  - Existence of files
  - Location on secondary memory
  - Current status
  - other attributes.
- Sometimes this information is maintained by a file management system



Process Table

- To manage and control a process, there is one entry for each process in the process table.
- Each entry points to a ***process image*** containing:

User Data

The modifiable part of the user space. May include program data, a user stack area, and programs that may be modified.

User Program

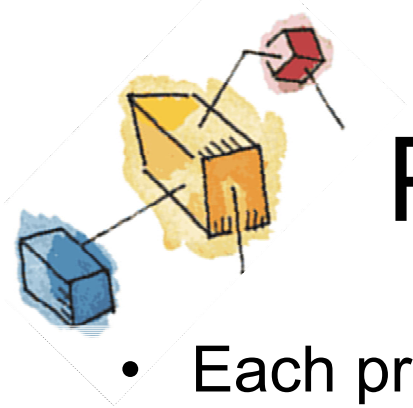
The program to be executed.

Stack

Each process has one or more last-in-first-out (LIFO) stacks associated with it. A stack is used to store parameters and calling addresses for procedure and system calls.

Process Control Block

Data needed by the OS to control the process (see Table 3.5).



Process Control Block

- Each process has associated with it a number of attributes that are used by the OS for process control.
- The attributes are stored in a data structure called a ***process control block*** (PCB), created and managed by the OS.
- It contains sufficient information so that it is possible to interrupt a running process and later resume its execution.

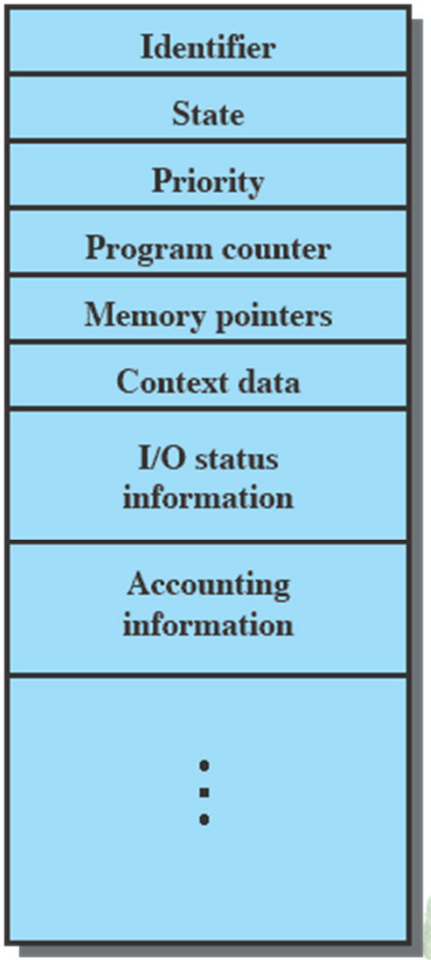
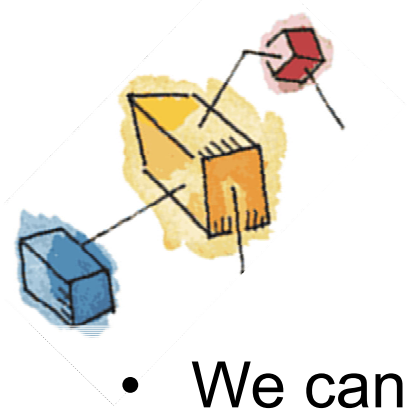
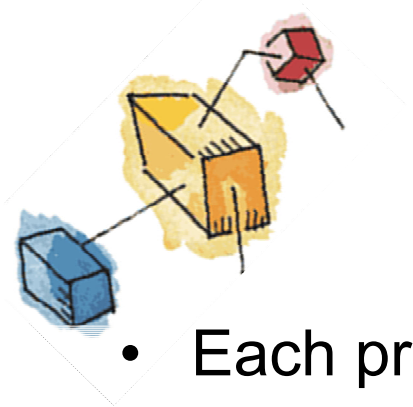


Figure 3.1 Simplified Process Control Block



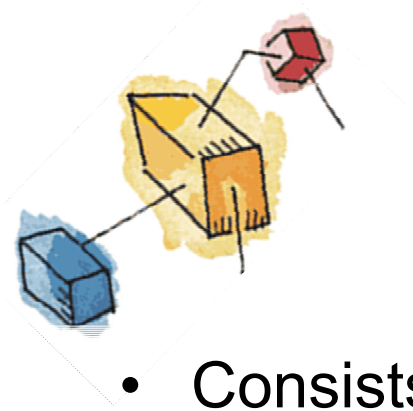
Process Attributes

- We can group the information in a PCB into three general categories:
  - Process identification
  - Processor state information
  - Process control information



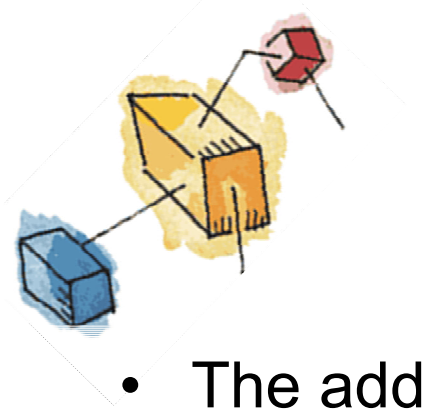
Process Identification

- Each process is assigned a unique numeric identifier.
  - Many of the tables controlled by the OS may use process identifiers to cross-reference process tables, e.g., memory tables may be organized to provide a map of main memory with an indication of which process is assigned to each region
  - When processes communicate with one another, the process identifier informs the OS of the destination of a particular communication
  - When processes are allowed to create other processes, identifiers indicate the parent and descendants of each process



Processor State Information

- Consists of processor registers' content
  - User-visible registers
  - Control and status registers
    - Program counter: address of the next instruction
    - Program status word (PSW)
      - Condition codes: result of the most recent arithmetic or logical operation (e.g., sign, zero, carry, equal, overflow)
      - Status information: e.g., interrupt enabled/disabled flags, execution mode
  - Stack pointers



Process Control Information

- The additional information needed by the OS to control and coordinate the various active processes
  - Process state
  - Priority
  - Scheduling-related info
  - Waiting event
  - Data structuring
  - … (see Table 3.5 for scope of information)

Process List Structures

- The queuing structure could be implemented as linked lists of PCBs in which pointers can be stored in the PCBs (structuring information).

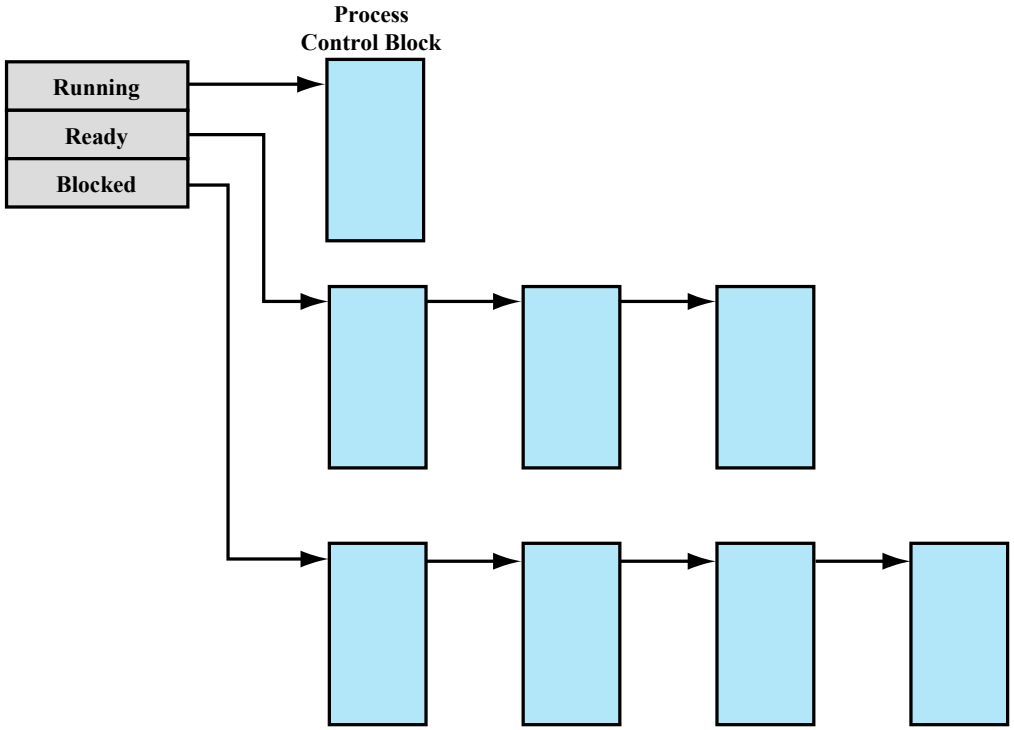


Figure 3.14 Process List Structures

Structure of Process Images in Virtual Memory

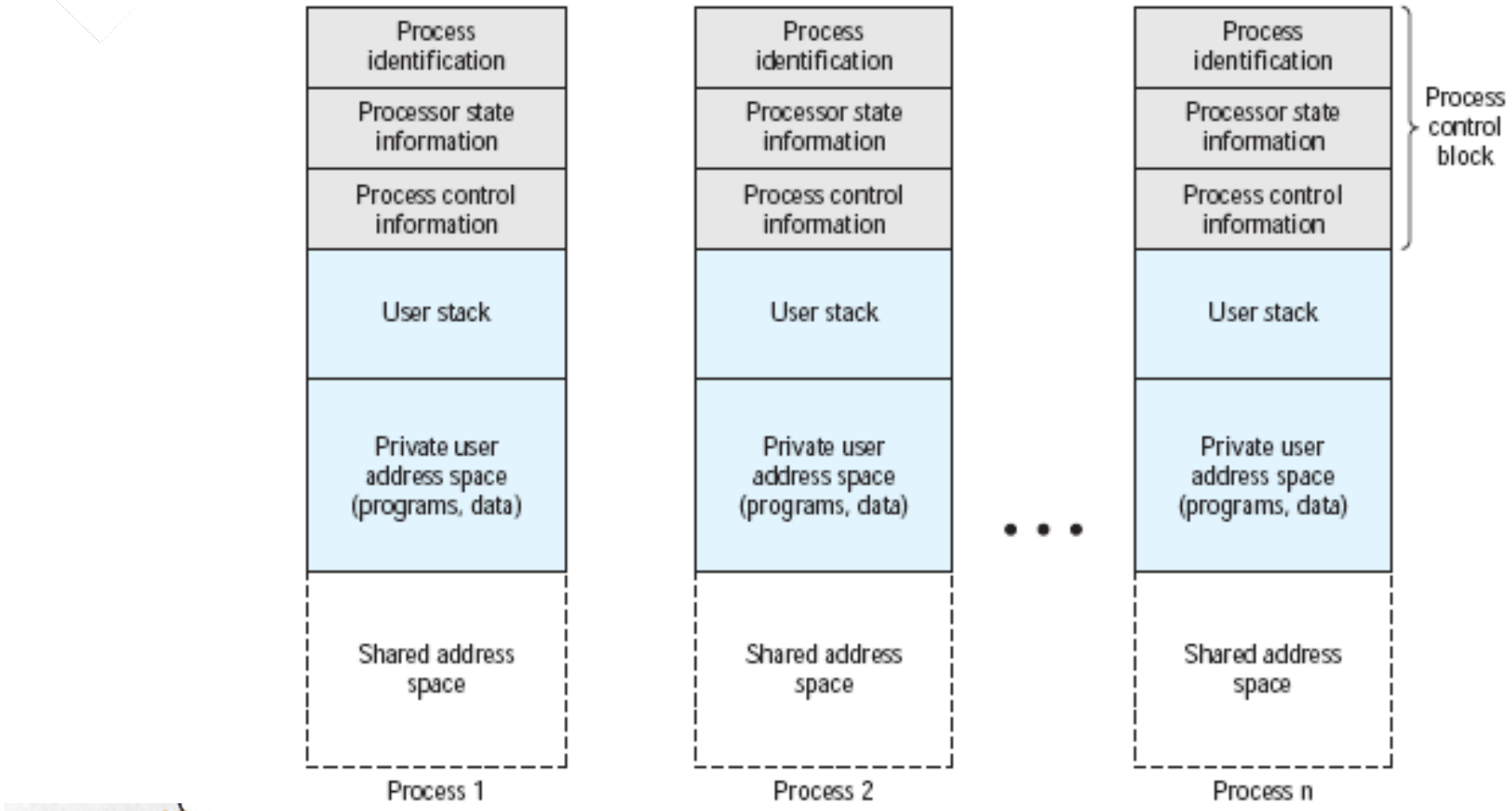
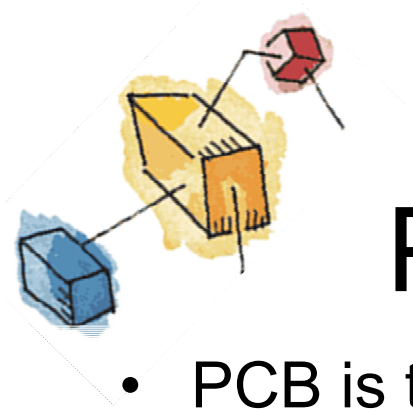
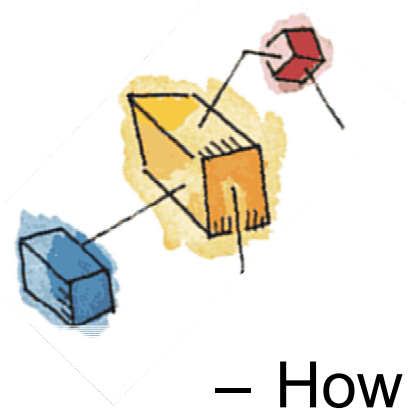


Figure 3.13 User Processes in Virtual Memory
(assume the whole process image in a contiguous range of addresses)



Role of the Process Control Block

- PCB is the most important data structure in an OS
  - Contains all of the information about a process that is needed by the OS
  - Read and/or modified by virtually every module in the OS such as scheduling, resource allocation, interrupt processing and performance monitoring
  - Defines the state of the OS
- PCB requires protection, which is difficult
  - A faulty routine could damage PCBs, which could destroy the OS's ability to manage the affected processes
  - Any design change to the PCB could affect many modules of the OS

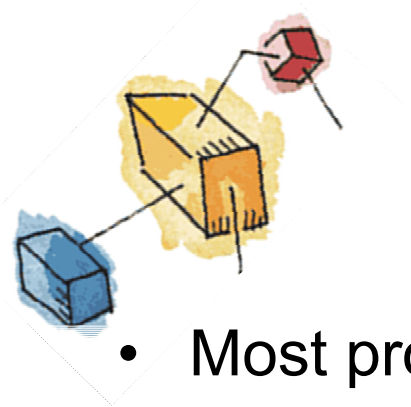


Roadmap

- How are processes represented and controlled by the OS
- A flavour of creating a process in UNIX
- ***Process states*** which characterize the behaviour of processes
- ***Data structures*** used to manage processes

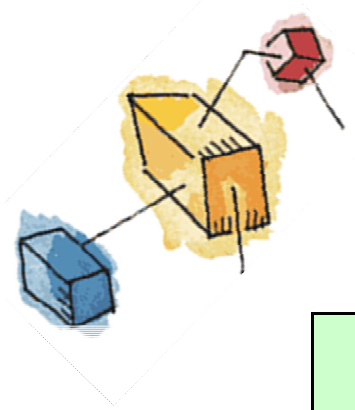
→ Ways in which the OS uses these data structures to control process execution

- Discuss process management in UNIX



Modes of Execution

- Most processors support at least two modes of execution to protect the OS and key OS tables from interference by user programs.
- User mode
  - Less-privileged mode
  - User programs typically execute in this mode
- System mode (control mode or kernel mode)
  - More-privileged mode
  - *Kernel* of the operating system (central module of an OS)
    - Loads first when the system starts
    - Resides in memory (in a protected area) all the time



Typical Functions of an OS Kernel

Process Management

- Process creation and termination
- Process scheduling and dispatching
- Process switching
- Process synchronization and support for interprocess communication
- Management of process control blocks

Memory Management

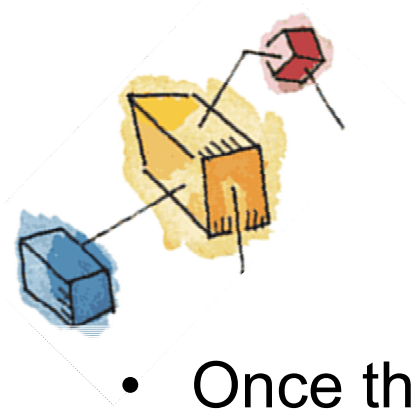
- Allocation of address space to processes
- Swapping
- Page and segment management

I/O Management

- Buffer management
- Allocation of I/O channels and devices to processes

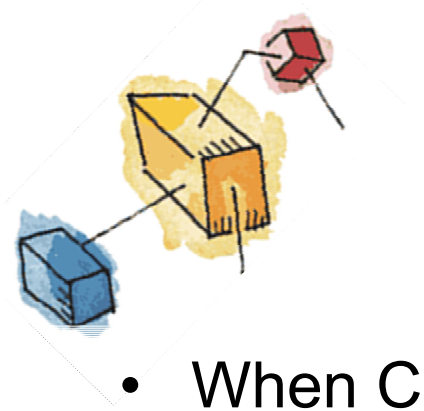
Support Functions

- Interrupt handling
- Accounting
- Monitoring



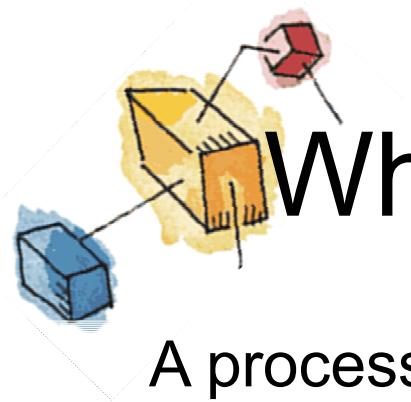
Process Creation

- Once the OS decides to create a new process, it:
  - Assigns a unique process identifier and adds a new entry to the process table
  - Allocates space for the process (process image)
  - Initializes process control block
  - Sets up appropriate linkages such as putting the new process in the Ready list
  - Creates or expands other data structures such as an accounting file for performance assessment



Process Switching

- When CPU switches to another process, the system must save the state of the old process and load the saved state for the new process (to be described).
- *Process-switch* time is considered as overhead (the system does no useful work while switching), so several issues are important
  - What events trigger a process switch?
  - What must the OS do to the various data structures to achieve such a process switch?

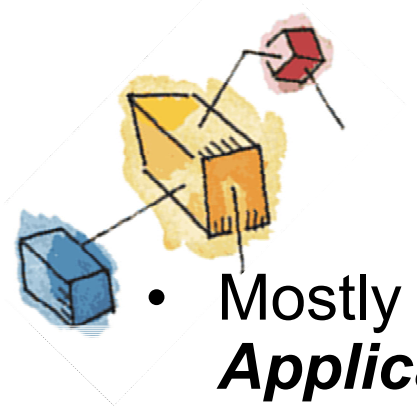


When to Switch Processes

A process switch may occur any time that the OS has gained control from the currently running process. Possible events giving OS control are:

Mechanism	Cause	Use
Interrupt	External to the execution of the current instruction, e.g., clock interrupt, I/O interrupt.	Reaction to an asynchronous external event
Trap	Associated with the execution of the current instruction, e.g., illegal file access	Handling of an error or an exception condition
System call (or supervisor call)	Explicit request, e.g., file open	Call to an operating system function

Table 3.8 Mechanisms for Interrupting the Execution of a Process



System Calls

- Mostly accessed by programs via a high-level ***Application Programming Interface*** (API).

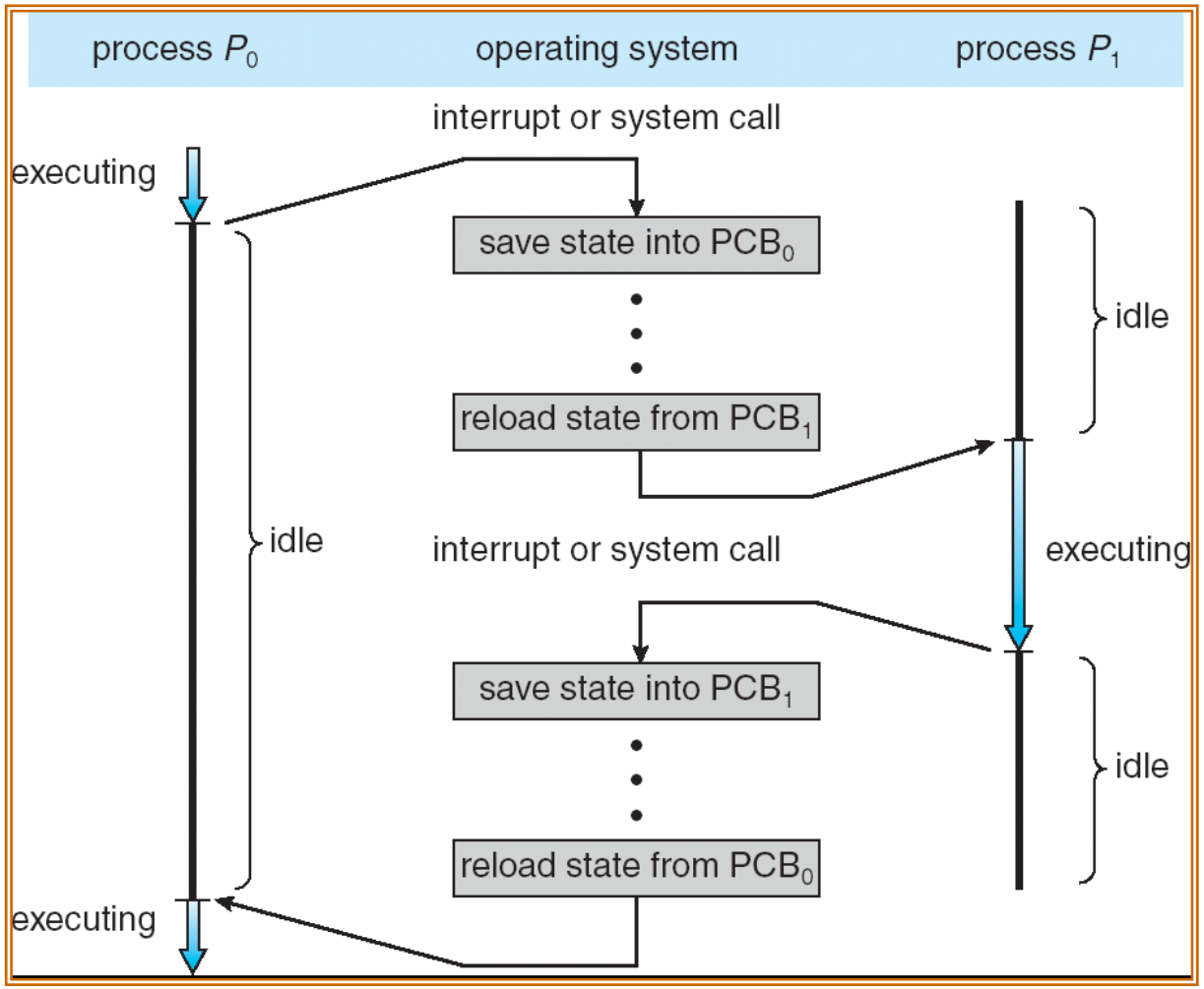
```
#include <stdio.h>
#include <string.h>
#include <iostream>

using namespace std;

int main ()
{           char ch;
            string s="";
            ch=getchar();
            while (ch!='.'){
                        s+=ch;
                        ch=getchar();
            }
 cout << "Your input is: " << s << endl;
}
```

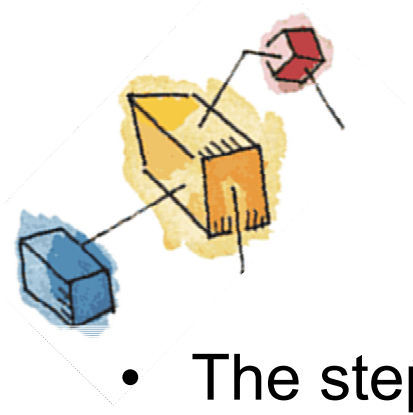
You can use command:
pgrep –f [program name]
to find the PID of this running program in UNIX.

Then use command
strace –pPID
to find all involved system calls of this program.

You will find getchar() will make read(…) system call and cout will make write (..) system call

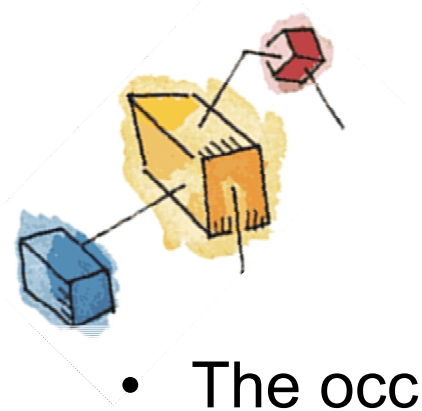
Process Switching





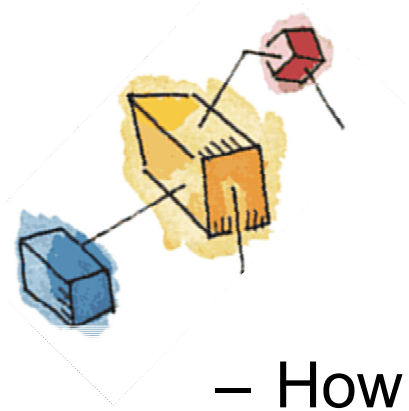
Process Switching

- The steps in a process switch are:
  1. Save context of processor including program counter and other registers
  2. Update the PCB of the process currently in the Running state
  3. Move the PCB of this process to appropriate queue – ready; blocked; ready/suspend
  4. Select another process for execution
  5. Update the PCB of the process selected
  6. Update memory management data structures
  7. Restore context of the processor to that which existed at the time the selected process was last switched out.



Mode Switching

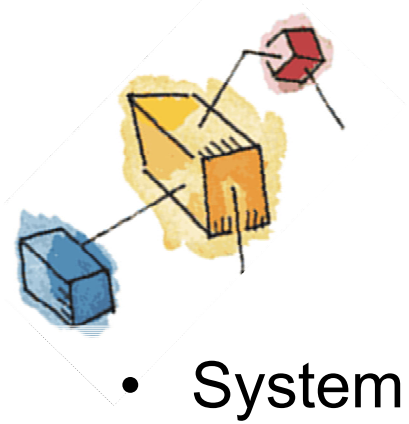
- The occurrence of an interrupt does not necessarily mean a process switch.
- It is possible that, after the processor switches from user mode to kernel mode in order to execute the interrupt handler (which may include privileged instructions), the currently running process will resume execution.
- In such a *mode switching* case, only need to save / restore the processor state information.



Roadmap

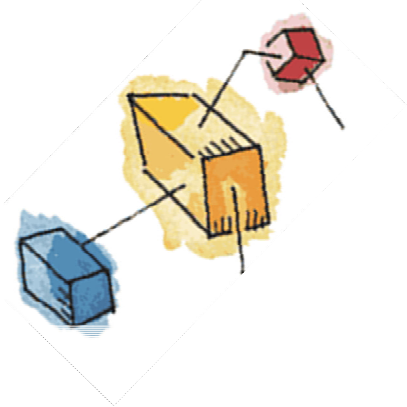
- How are processes represented and controlled by the OS
- A flavour of creating a process in UNIX
- ***Process states*** which characterize the behaviour of processes
- ***Data structures*** used to manage processes
- Ways in which the OS uses these data structures to control process execution

→ Discuss process management in UNIX



Unix

- System processes run in kernel mode
  - executes operating system code to perform administrative and housekeeping functions
- User processes
  - operate in user mode to execute user programs and utilities
  - operate in kernel mode to execute instructions that belong to the kernel
  - enter kernel mode by issuing a system call, when an exception is generated, or when an interrupt occurs



UNIX Process State Transition Diagram

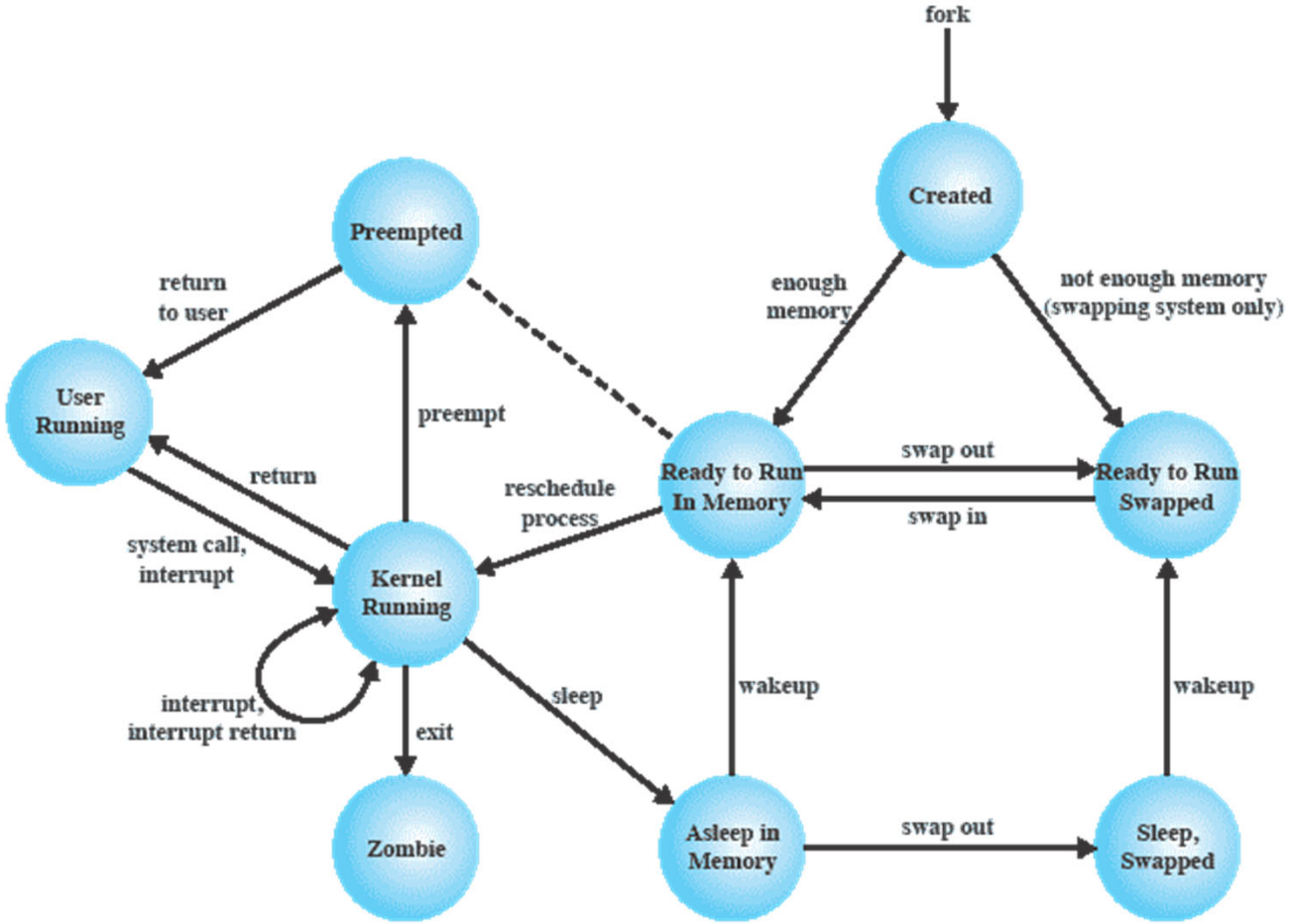
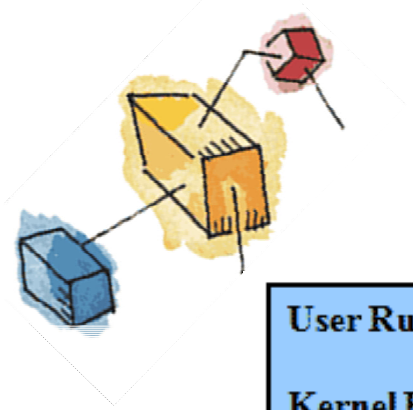
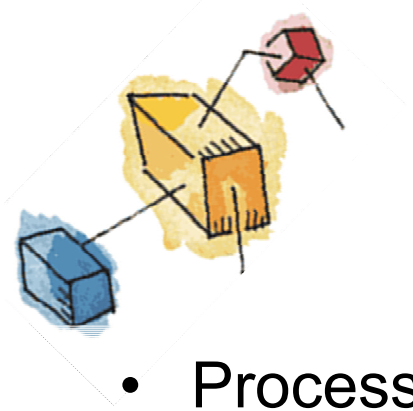


Figure 3.17 UNIX Process State Transition Diagram



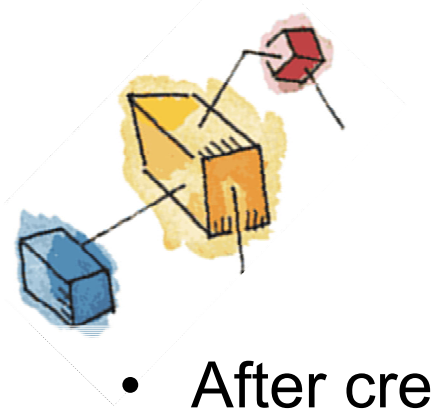
UNIX Process States

User Running	Executing in user mode.
Kernel Running	Executing in kernel mode.
Ready to Run, in Memory	Ready to run as soon as the kernel schedules it.
Asleep in Memory	Unable to execute until an event occurs; process is in main memory (a blocked state).
Ready to Run, Swapped	Process is ready to run, but the swapper must swap the process into main memory before the kernel can schedule it to execute.
Sleeping, Swapped	The process is awaiting an event and has been swapped to secondary storage (a blocked state).
Preempted	Process is returning from kernel to user mode, but the kernel preempts it and does a process switch to schedule another process.
Created	Process is newly created and not yet ready to run.
Zombie	Process no longer exists, but it leaves a record for its parent process to collect.



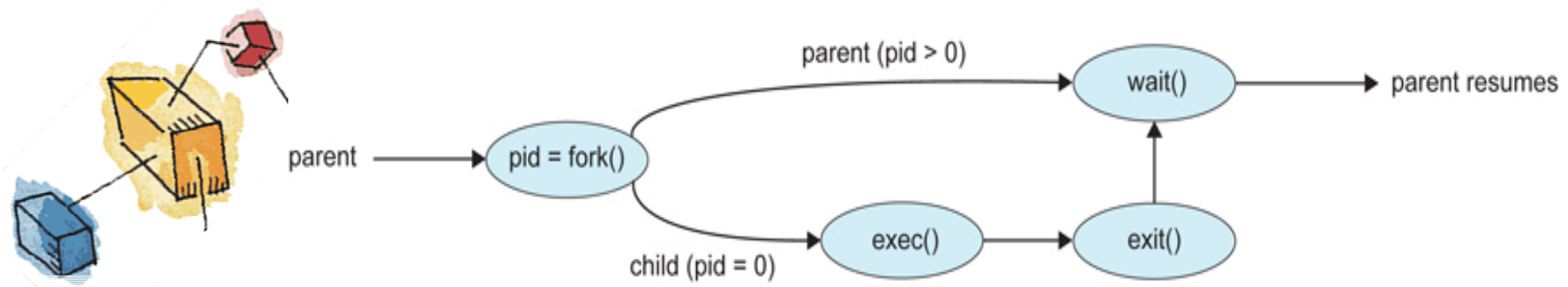
Deep Dive: Process Creation in Unix

- Process creation is by means of the kernel system call, **`fork( )`**.
- This causes the OS, in kernel mode, to:
  1. Allocate a slot in the process table for the new process.
  2. Assign a unique process ID to the child process.
  3. **Make a copy of the process image of the parent**, with the exception of any shared memory.
  4. Increment counters for any files owned by the parent, to reflect that an additional process now also owns those files.
  5. Assign the child process to the Ready to Run state.
  6. Return the ID number of the child to the parent process, and a 0 value to the child process.



After Creation

- After creating the process, the Kernel can do one of the following, as part of the dispatcher routine:
  - Stay in the parent process
    - Control returns to ***user mode*** at the point of the fork call of the parent
  - Transfer control to the child process
    - The child process begins executing at the same point in the code as the parent, namely at the return from the fork call
  - Transfer control to another process
    - Both parent and child are left in the ***Ready to Run*** state



Is there any difference of the program output if the following change is made to the if statement?

```
if ( pid == 0 ) {
        …
}
else wait (NULL);
```

```
#include <iostream>
#include <unistd.h>
using namespace std;

int x = 10;

int main(void)
{   int pid;
    pid = fork(); /* Spawn a new process */
    cout << "(1) x = " << x << " in " << pid << endl;
    if ( pid == 0 ) {
        cout << "(2) x = " << x << " in " << pid << endl;
        x = x + 5;
        cout << "(3) x = " << x << " in " << pid << endl;
    }
    cout << "(4) x = " << x << " in " << pid << endl;
}
```