CSCI-GA.2250-001

Operating Systems

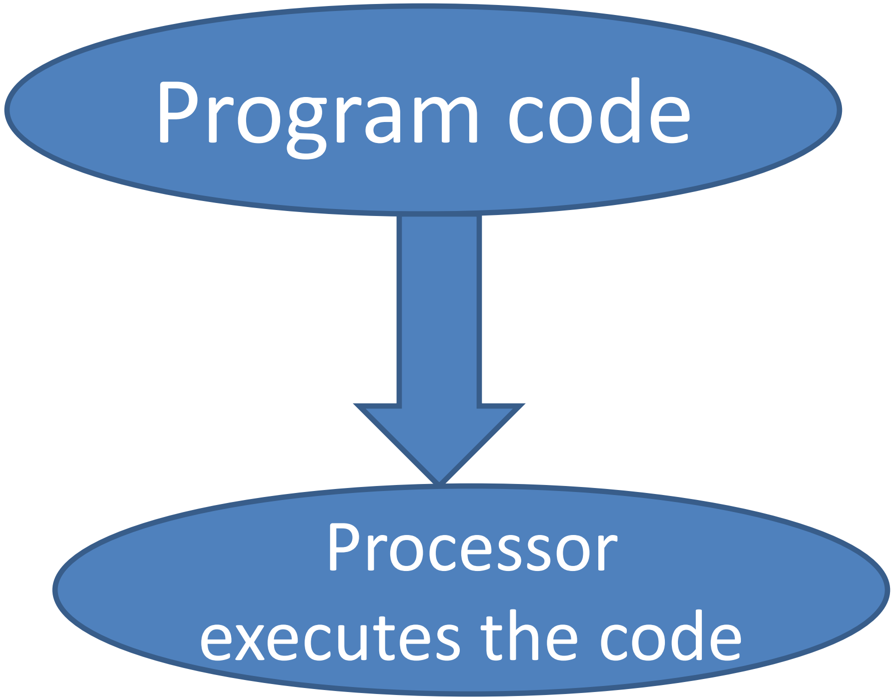
Lecture 2:

Processes and Threads - Part 1

Mohamed Zahran (aka Z)
mzahran@cs.nyu.edu
http://www.mzahran.com

OS Management of Application Execution

- Resources are made available to multiple applications
- The processor is switched among multiple applications so all will appear to be progressing
- The processor and I/O devices can be used efficiently



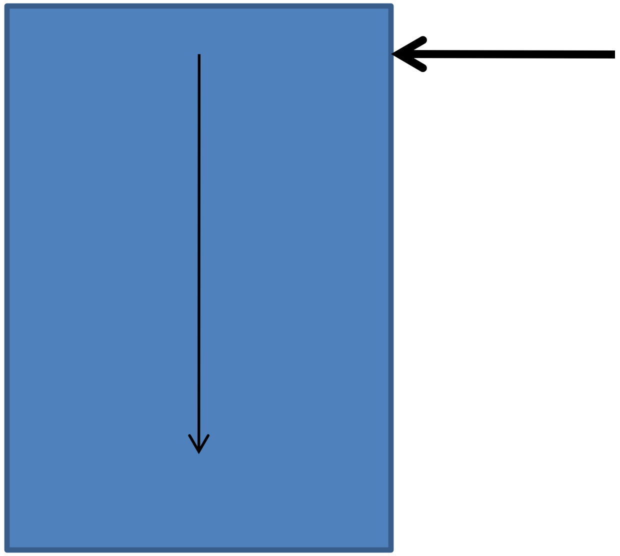
Program code

Processor executes the code

When the processor begins to execute the program code, we refer to this executing entity as a ***process***

What Is a Process?

An abstraction of a running program



Program Counter

The Process Model

- A process is an instance of an executing program and includes
  - Program counter
  - Registers
  - Variables
  - …
- A process has a program, input, output, and state.

If a program is running twice, does it count as two processes? or one?

Process Element

- While the program is executing, this process can be uniquely characterized by a number of elements, including:

Where do we store all this info?

identifier

state

priority

program counter

memory pointers

context data

I/O status information

accounting information

Process Control Block

- Contains the process elements
- It is possible to interrupt a running process and later resume execution as if the interruption had not occurred
- Created and managed by the operating system
- Key tool that allows support for multiple processes

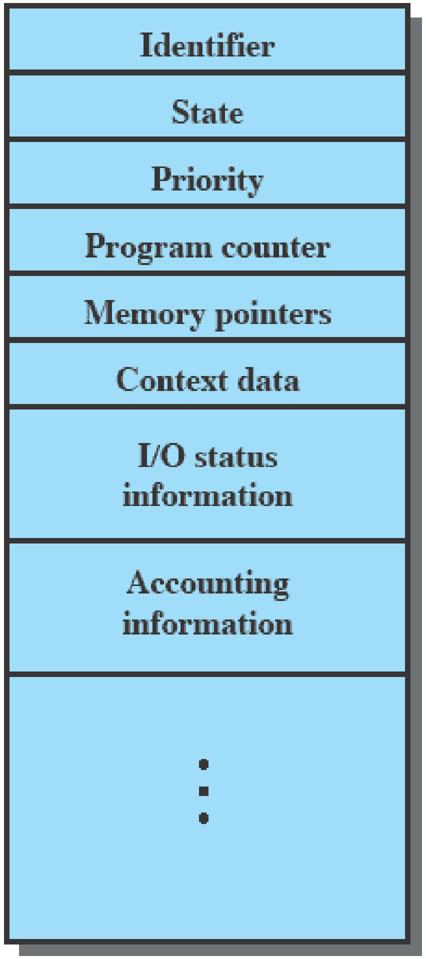
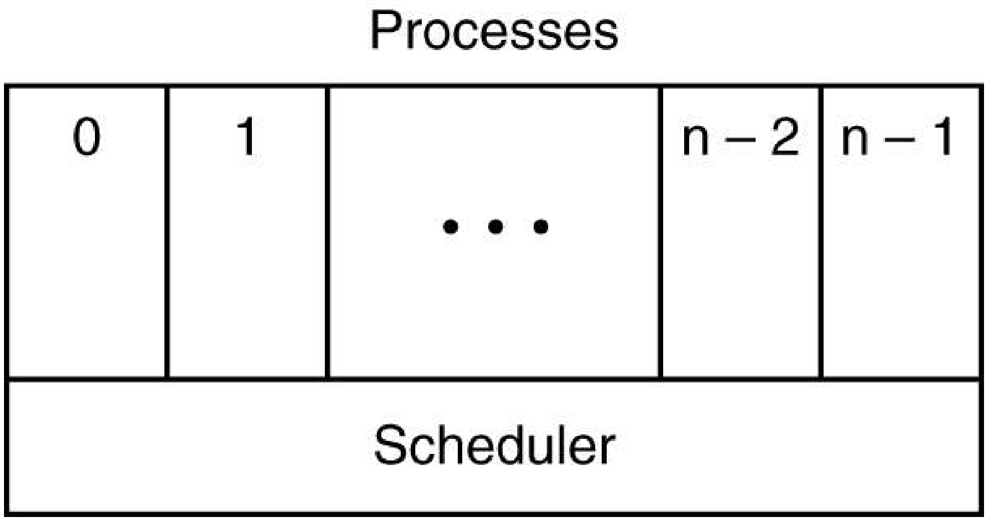
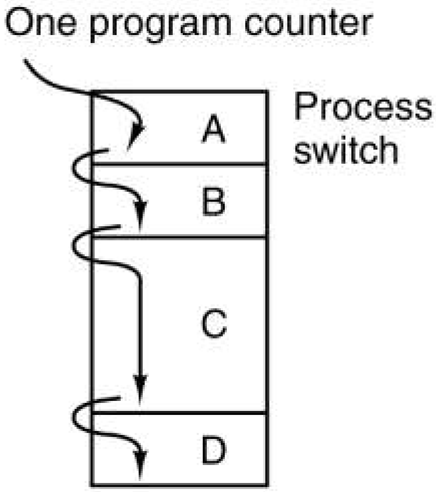


Figure 3.1 Simplified Process Control Block

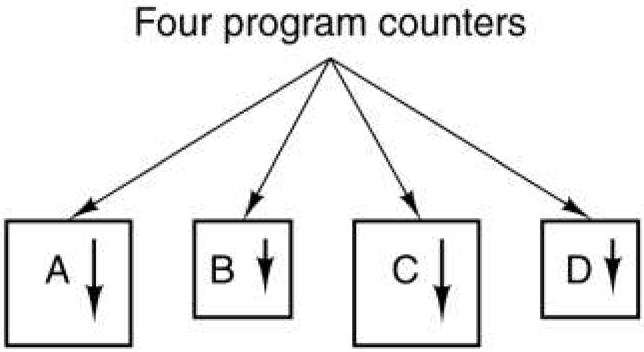
Multiprogramming

- One CPU and several processes
- CPU switches from process to process quickly

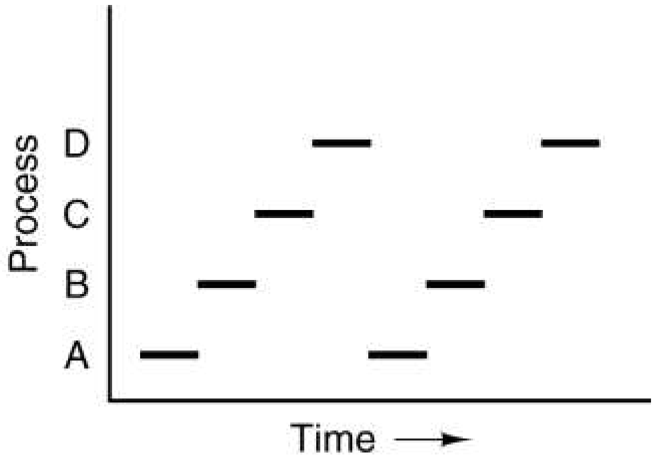




(a)



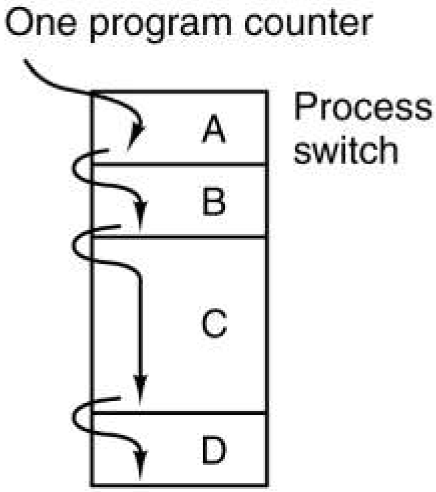
(b)



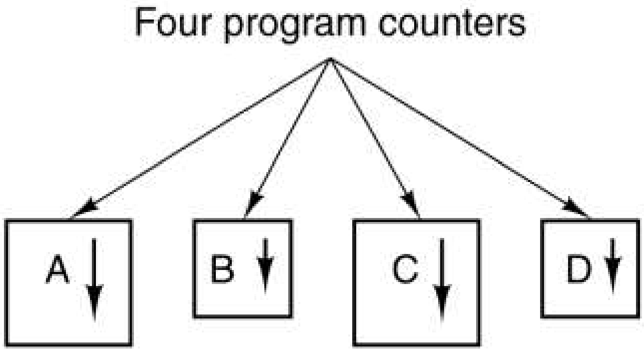
What Really Happens

What We Think It Happens

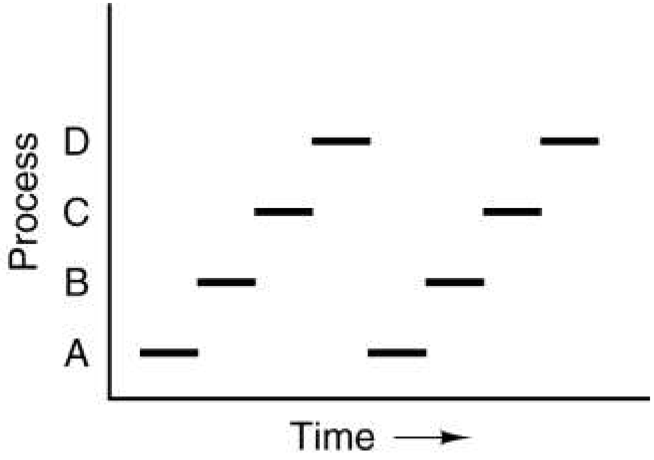
If we run the same program several times, will we get the same execution time?



(a)



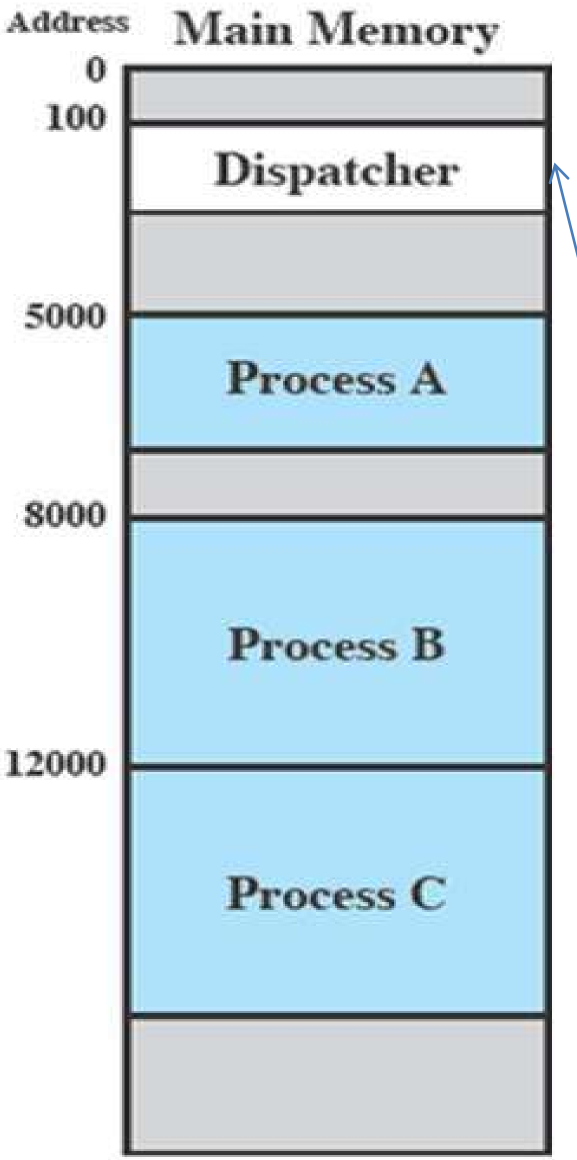
(b)



What Really Happens

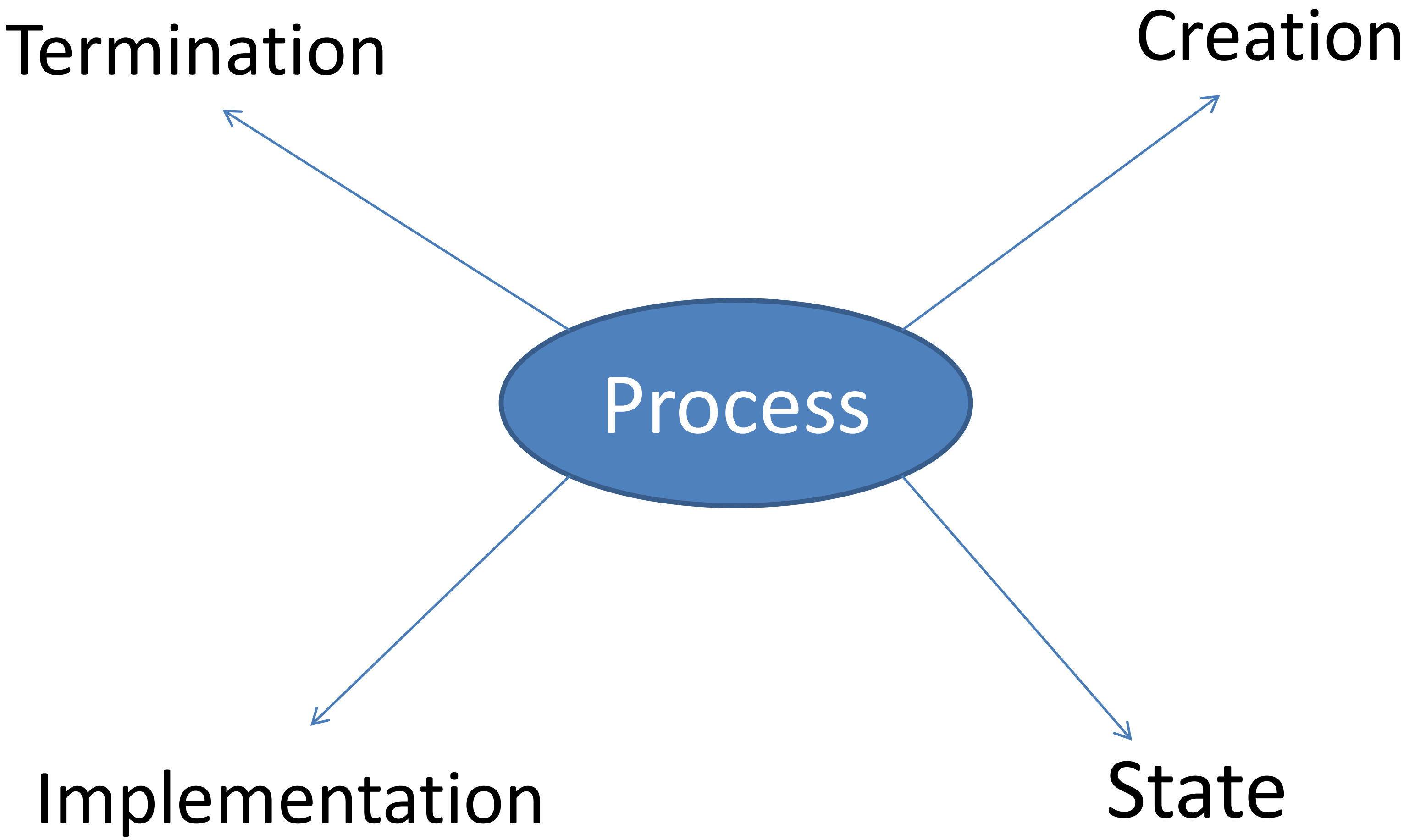
What We Think It Happens

Example



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

Small program that switches the processor from one process to another (also called Scheduler)



Process

Termination

Creation

Implementation

State

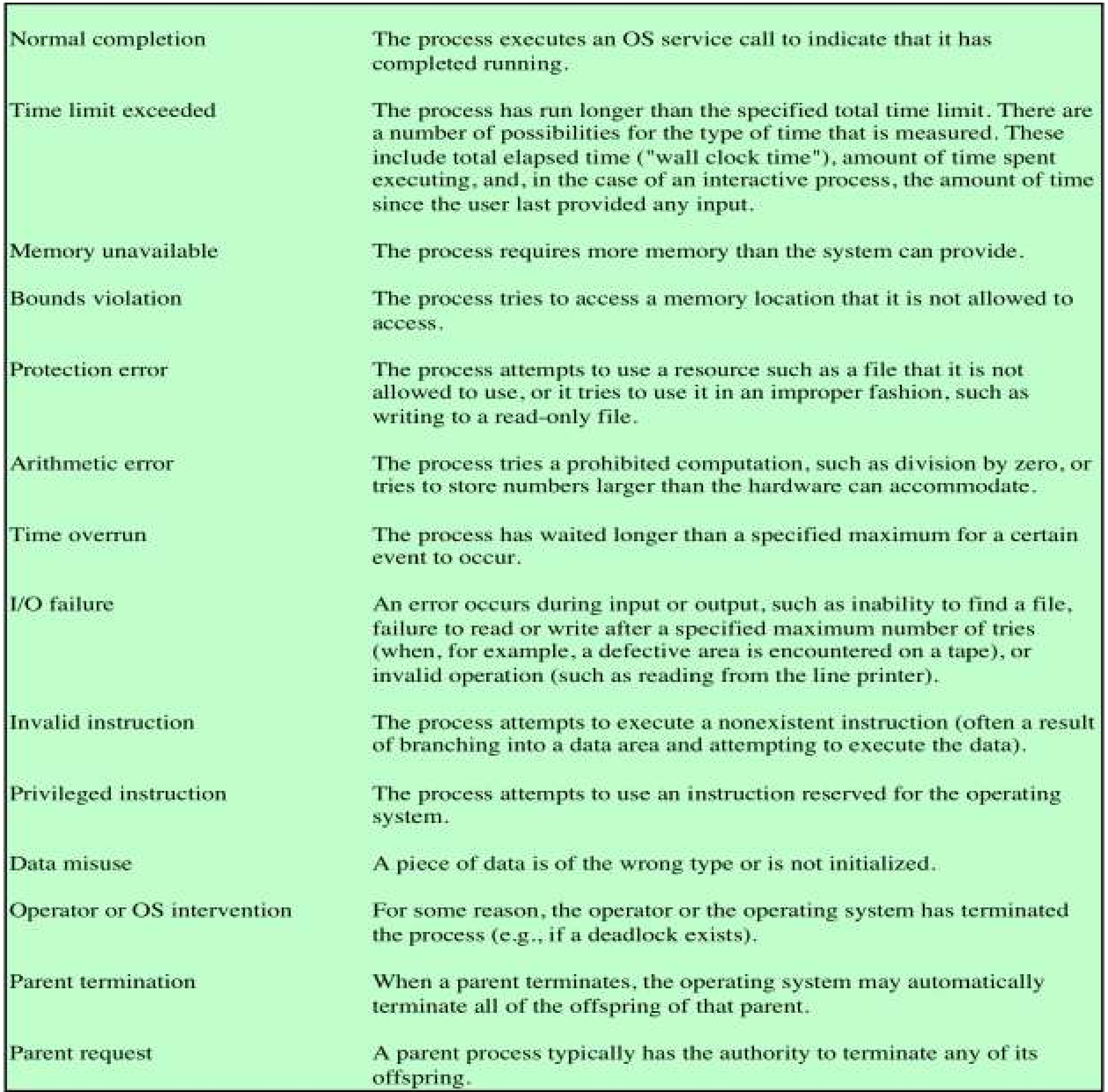
Process Creation

- System initialization
  - At boot time
  - Foreground
  - Background (daemons)
- Execution of a process creation system call by a running process
- A user request
- A batch job
- Created by OS to provide a service
- Interactive logon

Process Termination

- Normal exit (voluntary)
- Error exit (voluntary)
- Fatal error (involuntary)
- Killed by another process (involuntary)

Process Termination: More Scenarios

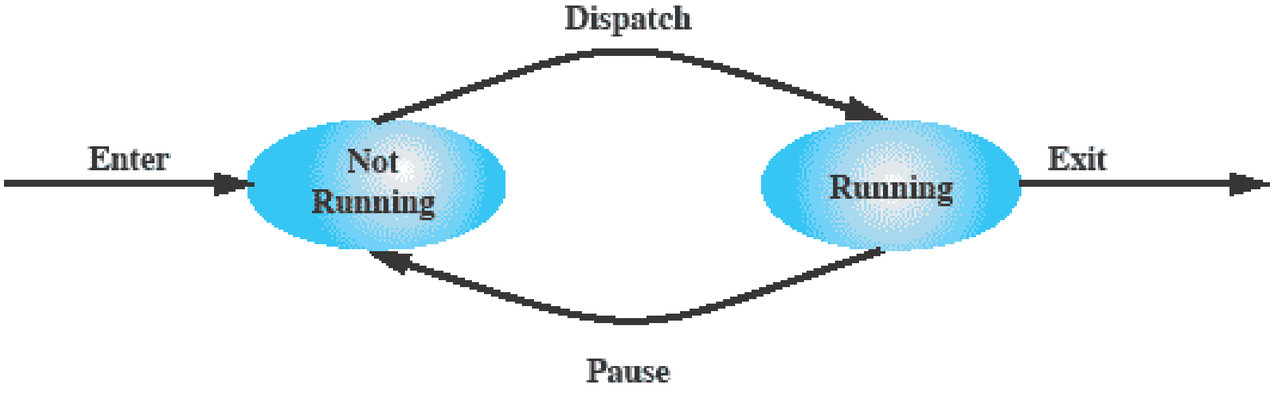


Normal completion	The process executes an OS service call to indicate that it has completed running.
Time limit exceeded	The process has run longer than the specified total time limit. There are a number of possibilities for the type of time that is measured. These include total elapsed time ("wall clock time"), amount of time spent executing, and, in the case of an interactive process, the amount of time since the user last provided any input.
Memory unavailable	The process requires more memory than the system can provide.
Bounds violation	The process tries to access a memory location that it is not allowed to access.
Protection error	The process attempts to use a resource such as a file that it is not allowed to use, or it tries to use it in an improper fashion, such as writing to a read-only file.
Arithmetic error	The process tries a prohibited computation, such as division by zero, or tries to store numbers larger than the hardware can accommodate.
Time overrun	The process has waited longer than a specified maximum for a certain event to occur.
I/O failure	An error occurs during input or output, such as inability to find a file, failure to read or write after a specified maximum number of tries (when, for example, a defective area is encountered on a tape), or invalid operation (such as reading from the line printer).
Invalid instruction	The process attempts to execute a nonexistent instruction (often a result of branching into a data area and attempting to execute the data).
Privileged instruction	The process attempts to use an instruction reserved for the operating system.
Data misuse	A piece of data is of the wrong type or is not initialized.
Operator or OS intervention	For some reason, the operator or the operating system has terminated the process (e.g., if a deadlock exists).
Parent termination	When a parent terminates, the operating system may automatically terminate all of the offspring of that parent.
Parent request	A parent process typically has the authority to terminate any of its offspring.

Process State

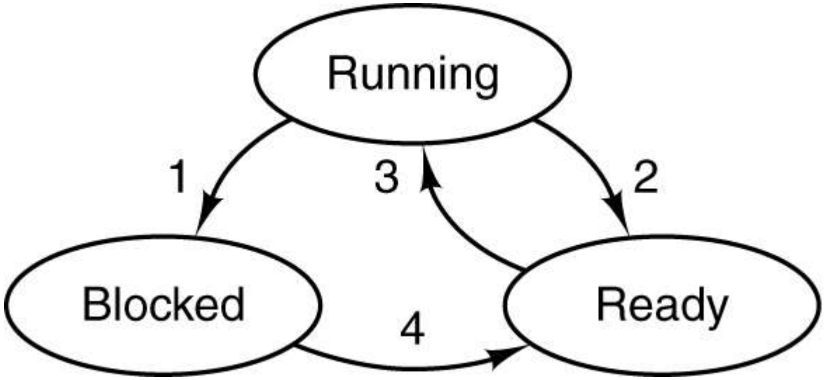
- Depending on the implementation, there can be several possible state models.

- The Simplest one: Two-state diagram



(a) State transition diagram

Process State:
Three-State Model



1. Process blocks for input
2. Scheduler picks another process
3. Scheduler picks this process
4. Input becomes available

Process State
Five-State Model

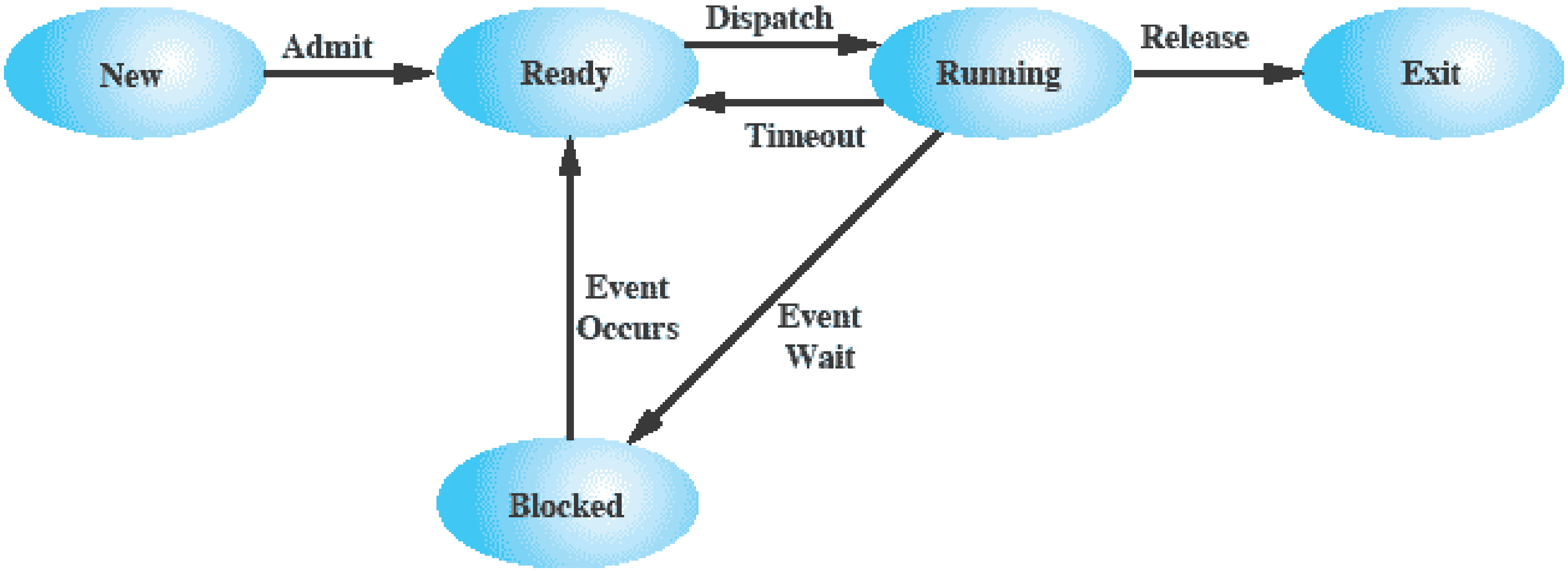
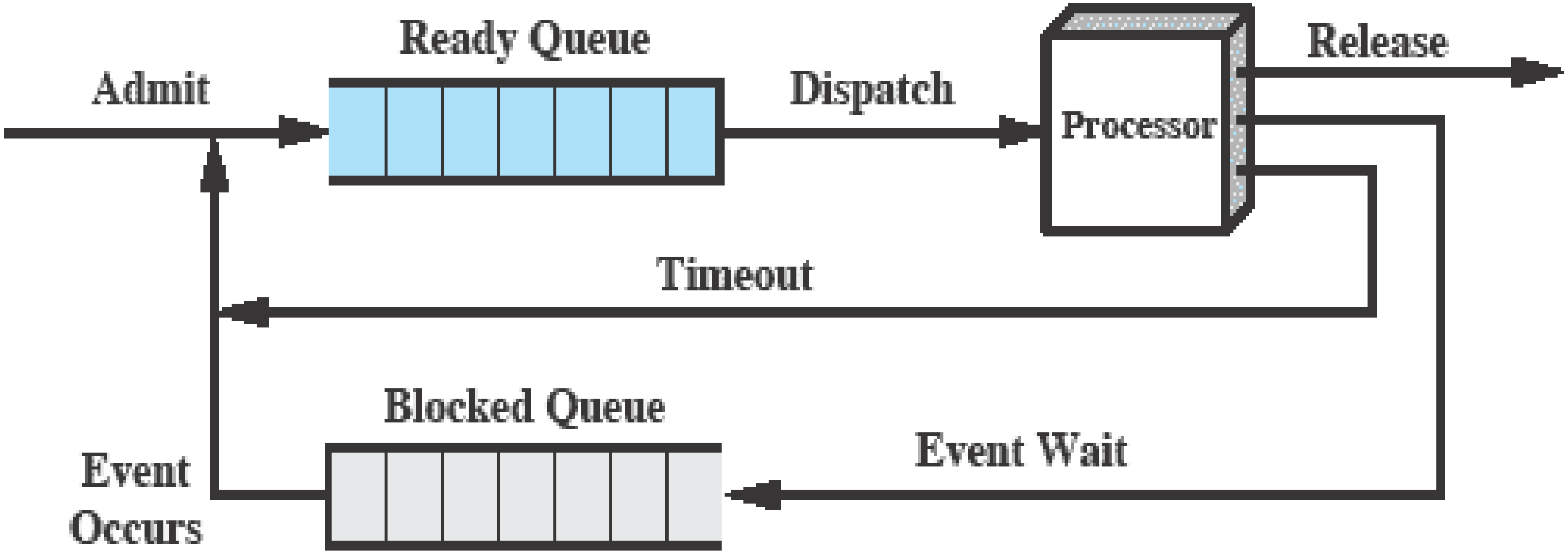


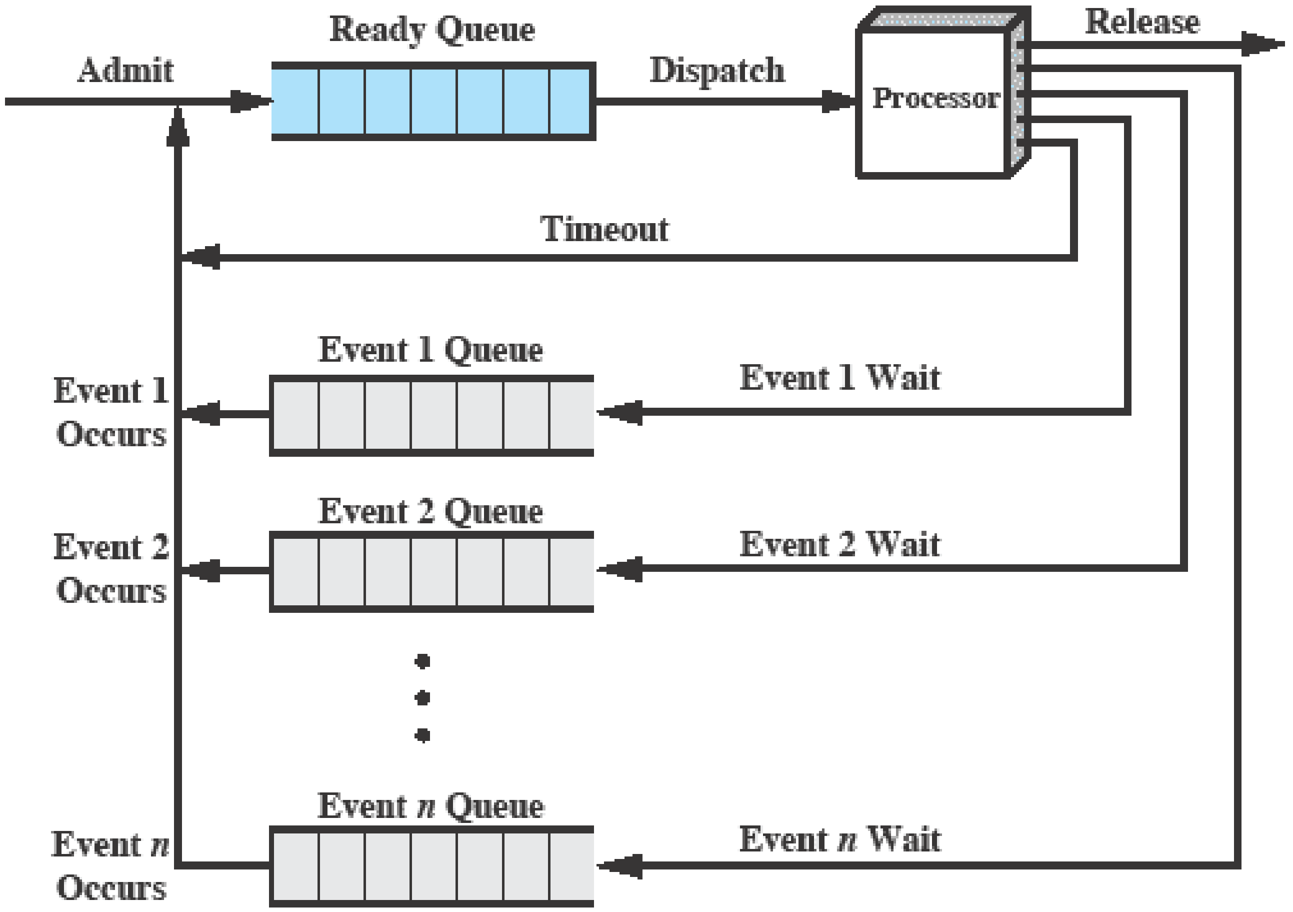
Figure 3.6 Five-State Process Model

Using Queues to Manage Processes



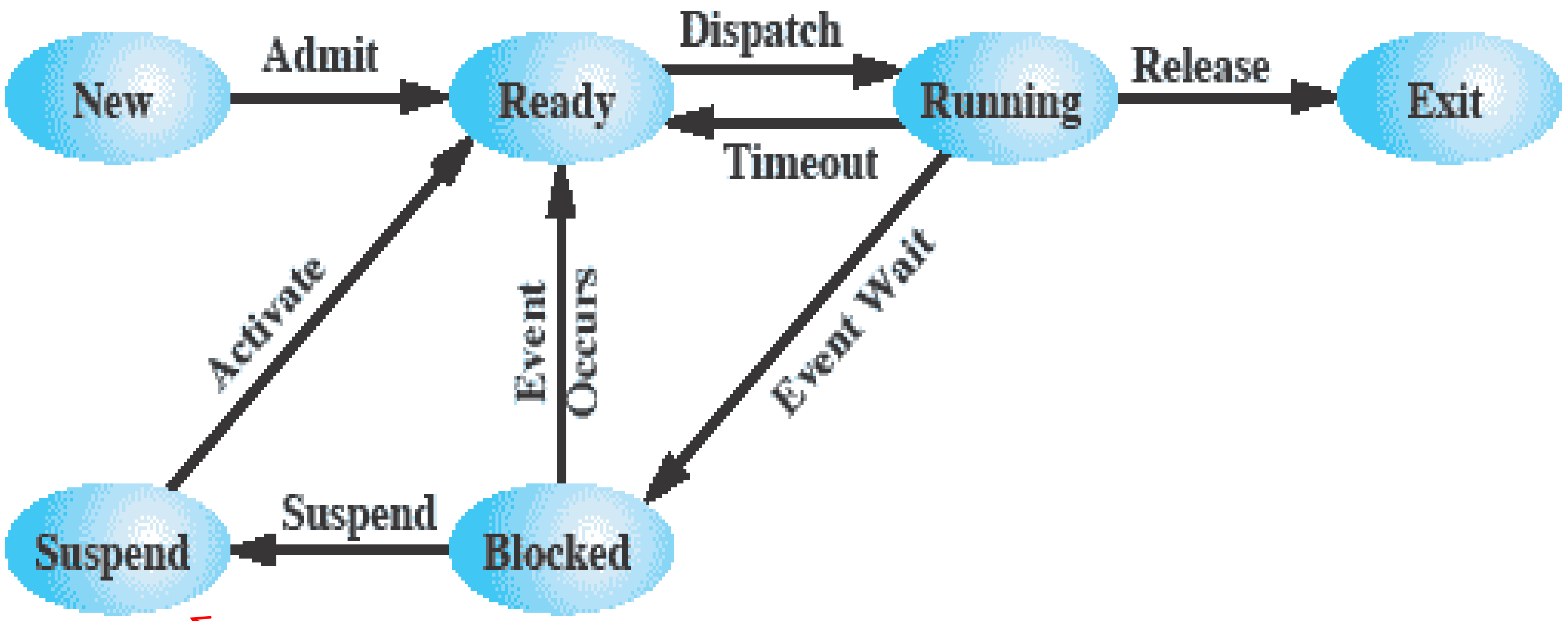
(a) Single blocked queue

Using Queues to Manage Processes



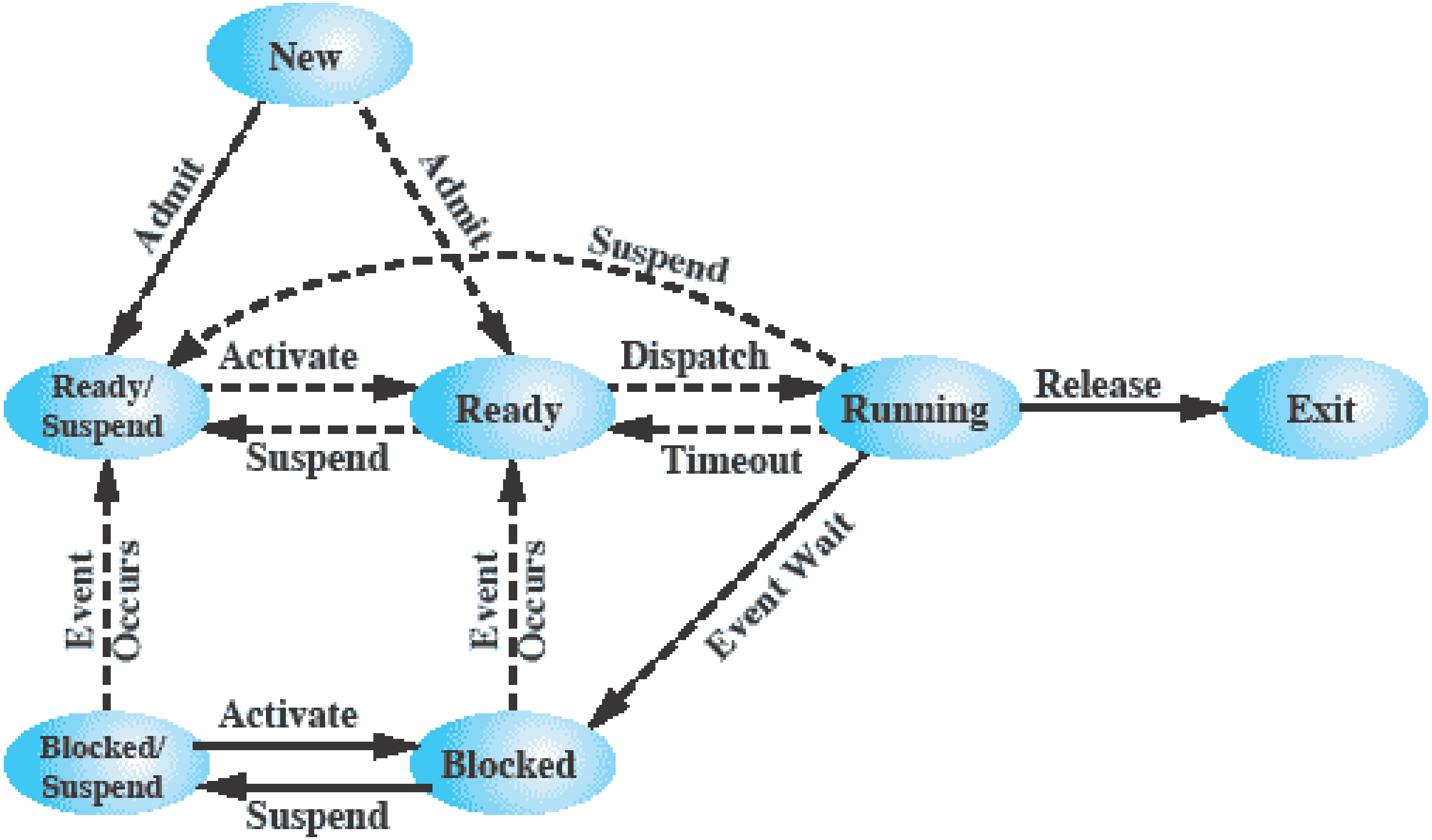
(b) Multiple blocked queues

One Extra State!



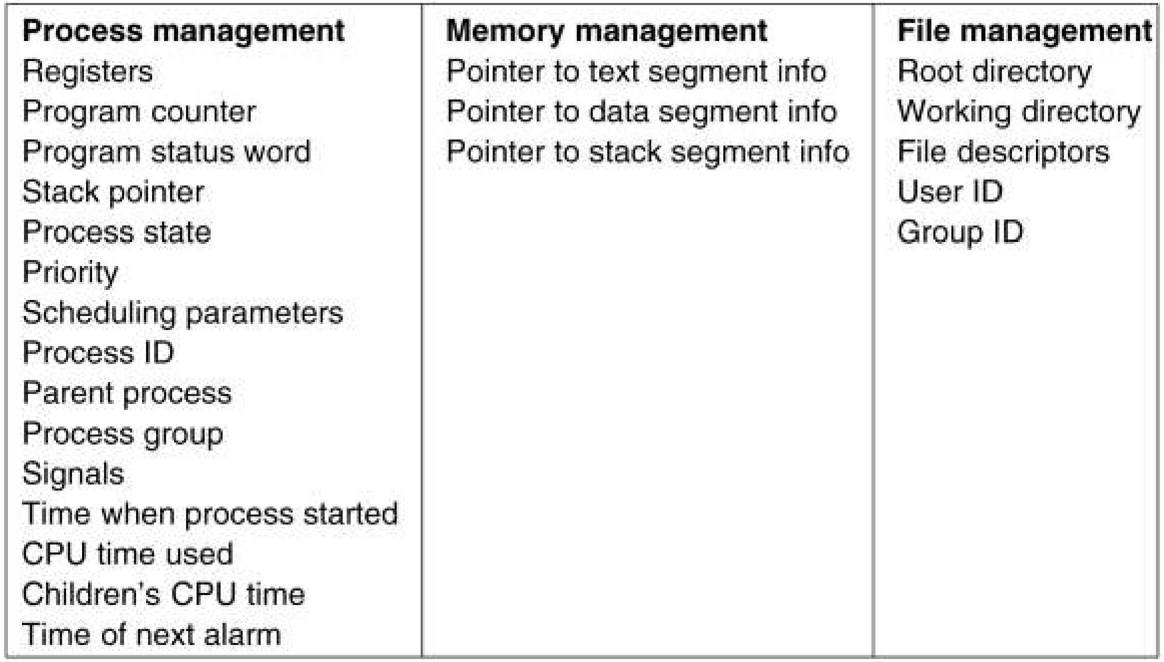
Swapped to disk

One Extra State!

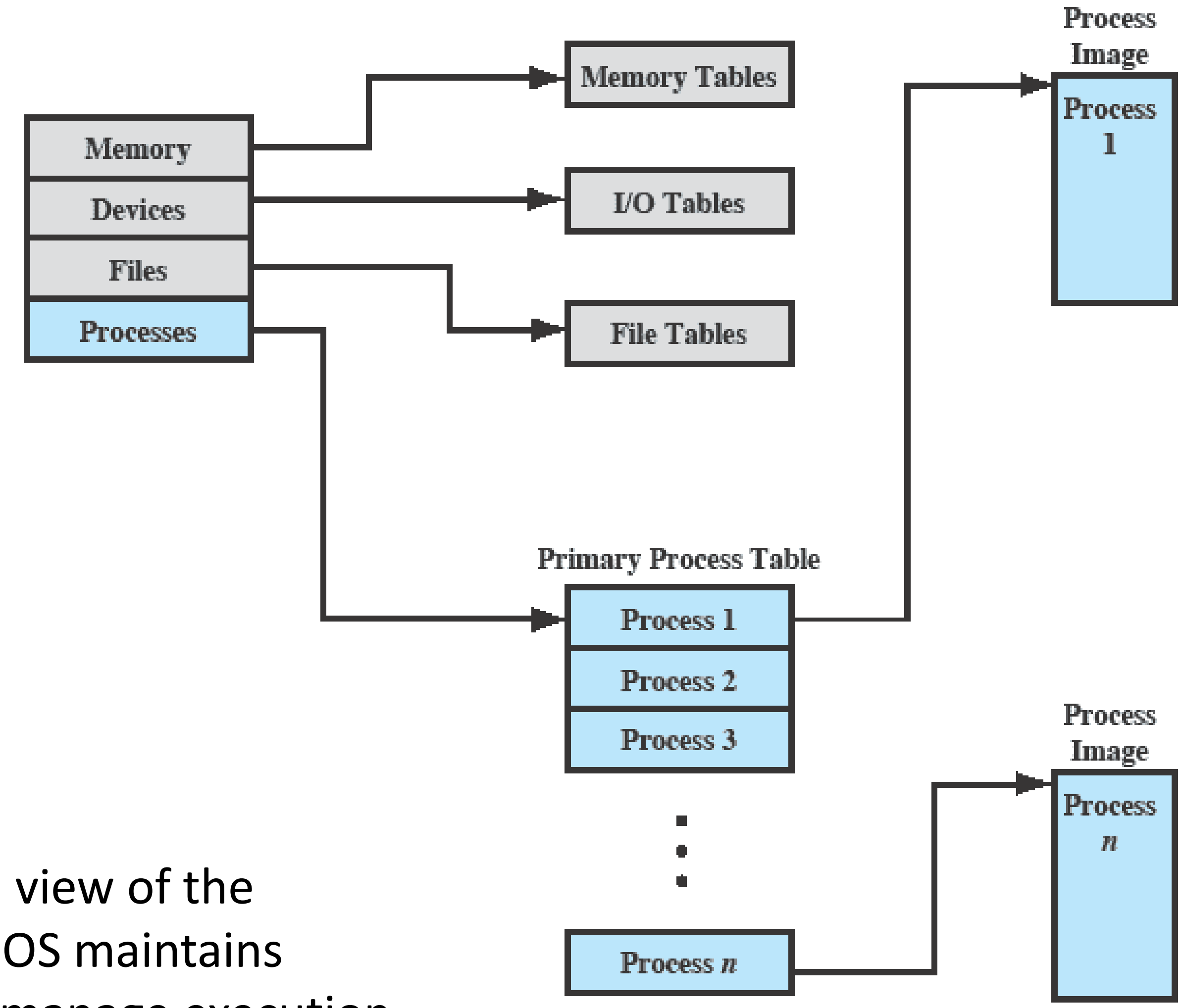


Implementation of Processes

- OS maintains a Control table (also called process table)
- An array of structures
- One entry per process



Process management	Memory management	File management
Registers	Pointer to text segment info	Root directory
Program counter	Pointer to data segment info	Working directory
Program status word	Pointer to stack segment info	File descriptors
Stack pointer		User ID
Process state		Group ID
Priority		
Scheduling parameters		
Process ID		
Parent process		
Process group		
Signals		
Time when process started		
CPU time used		
Children's CPU time		
Time of next alarm		



Memory

Devices

Files

Processes

Memory Tables

I/O Tables

File Tables

Primary Process Table

Process 1

Process 2

Process 3

Process *n*

Process Image

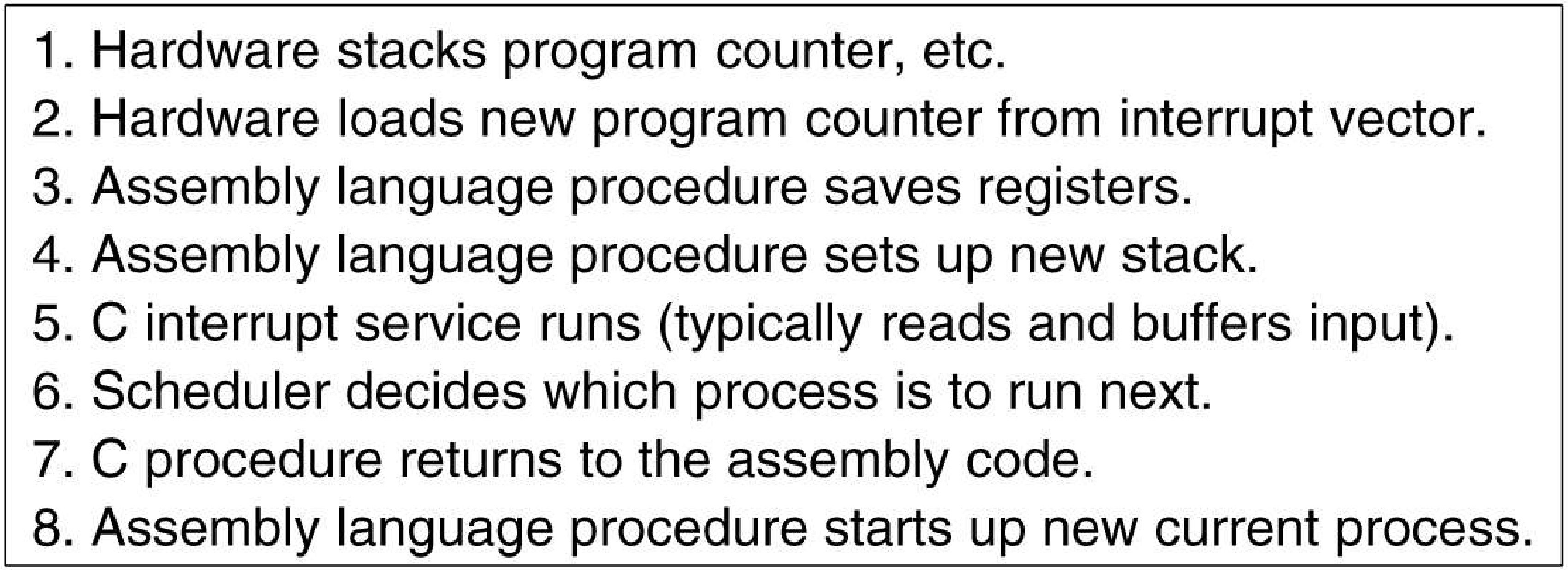
Process 1

Process Image

Process *n*

Conceptual view of the
tables that OS maintains
in order to manage execution
of processes on resources.

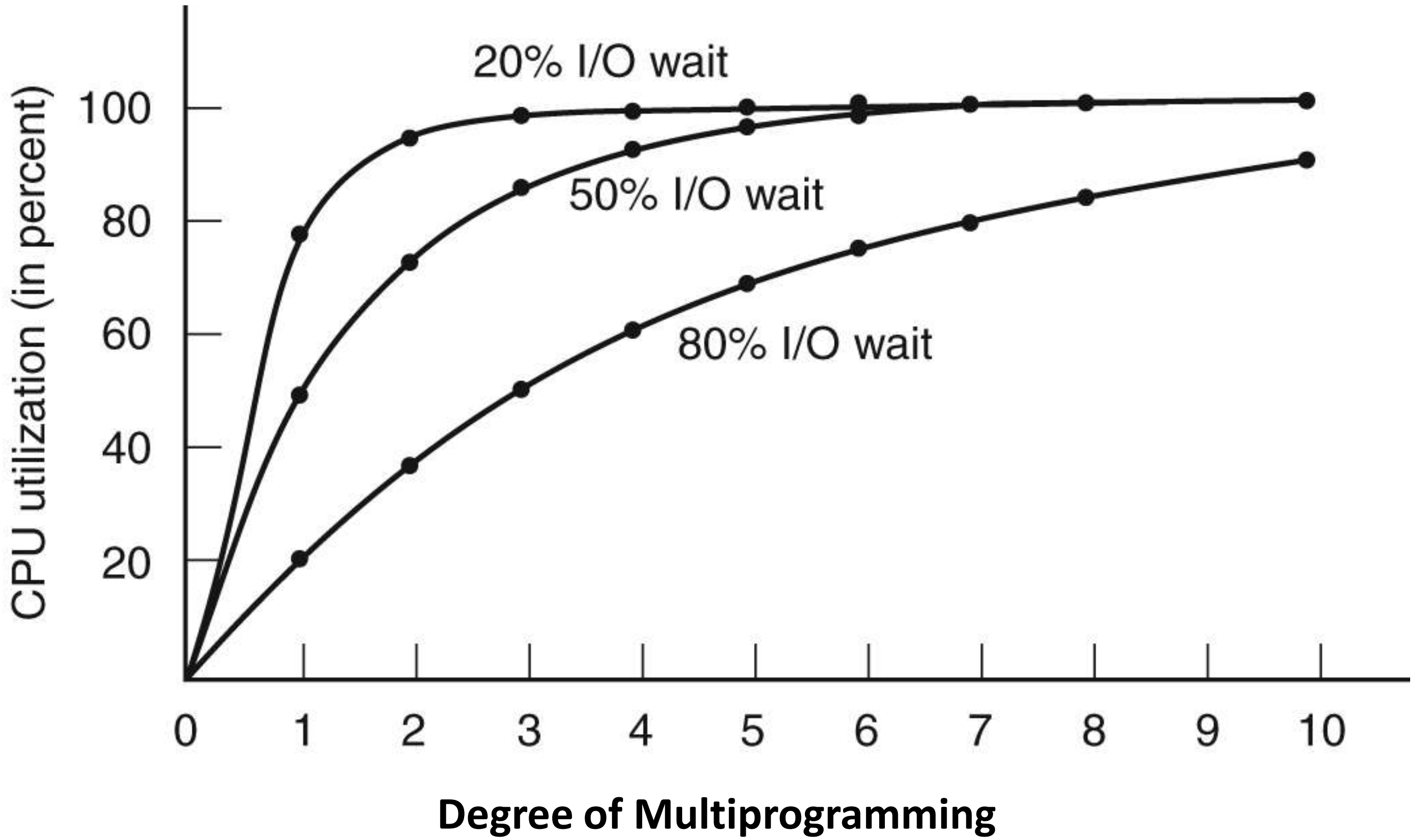
A Bit About Interrupts



1. Hardware stacks program counter, etc.
2. Hardware loads new program counter from interrupt vector.
3. Assembly language procedure saves registers.
4. Assembly language procedure sets up new stack.
5. C interrupt service runs (typically reads and buffers input).
6. Scheduler decides which process is to run next.
7. C procedure returns to the assembly code.
8. Assembly language procedure starts up new current process.

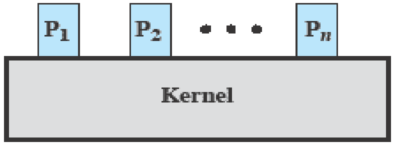
Simple Modeling of Multiprogramming

- A process spends fraction p waiting for I/O
- Assume n processors in memory at once
- The probability that all processes are waiting for I/O at once is p^n
- So -> CPU Utilization = $1 - p^n$

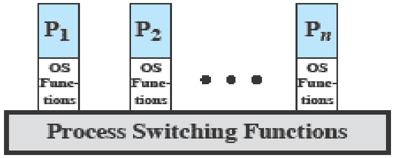


Multiprogramming lets processes use the CPU when it would otherwise become idle.

Executing the OS Itself



(a) Separate kernel



(b) OS functions execute within user processes

(c) OS functions execute as separate processes

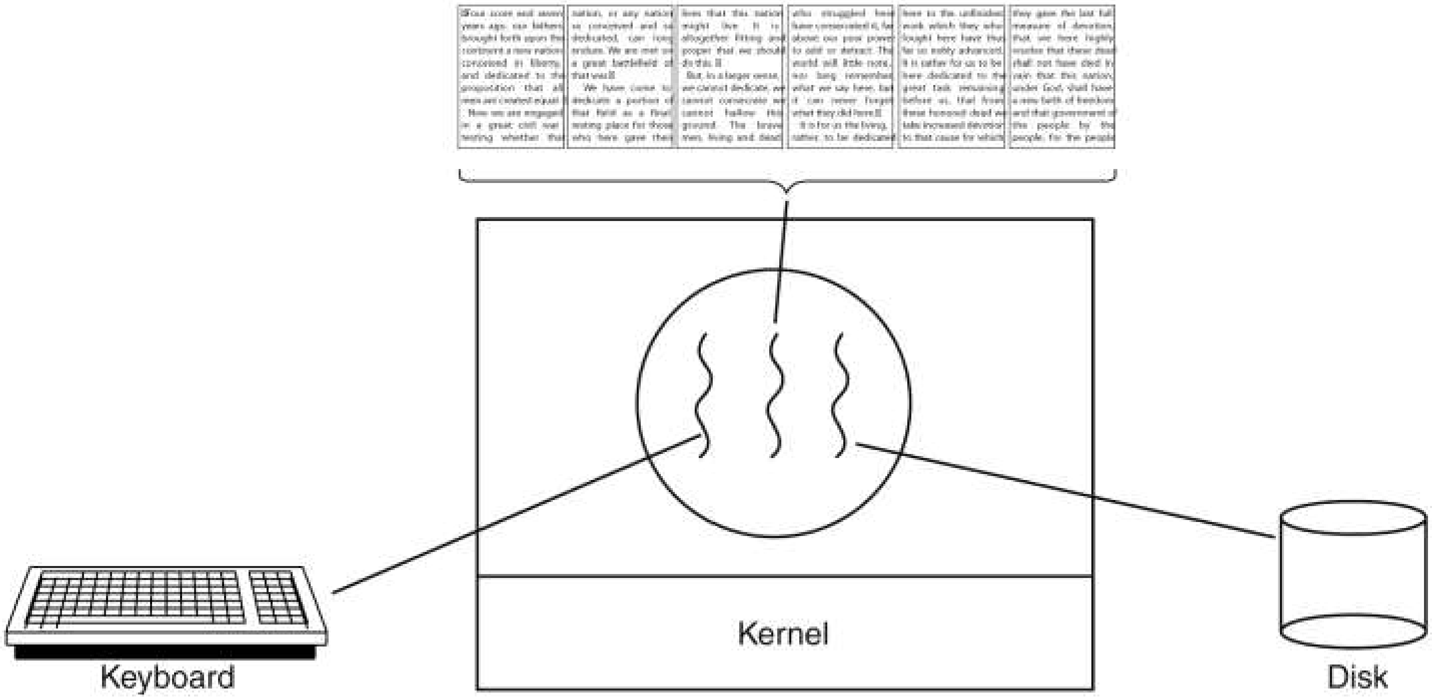
Threads

- Multiple threads of control within a process
- All threads of a process share the same address space

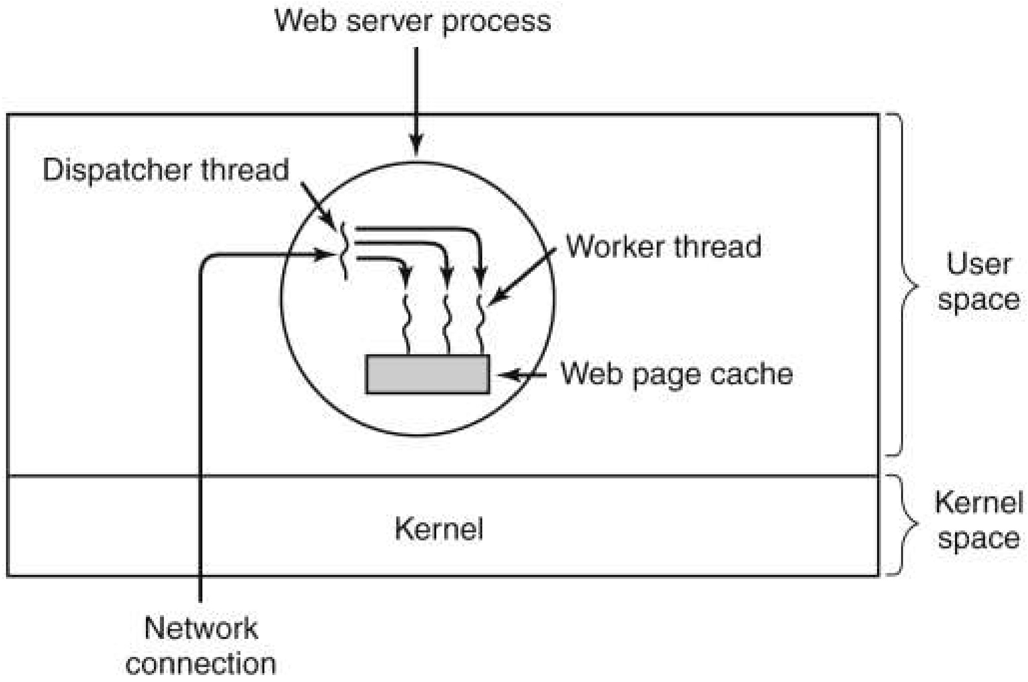
Why Threads?

- For some applications many activities can happen at once
    - With threads, programming becomes easier
    - Benefit applications with I/O and processing that can overlap
- Lighter weight than processes
    - Faster to create and restore

Example 1:
A Word Processor



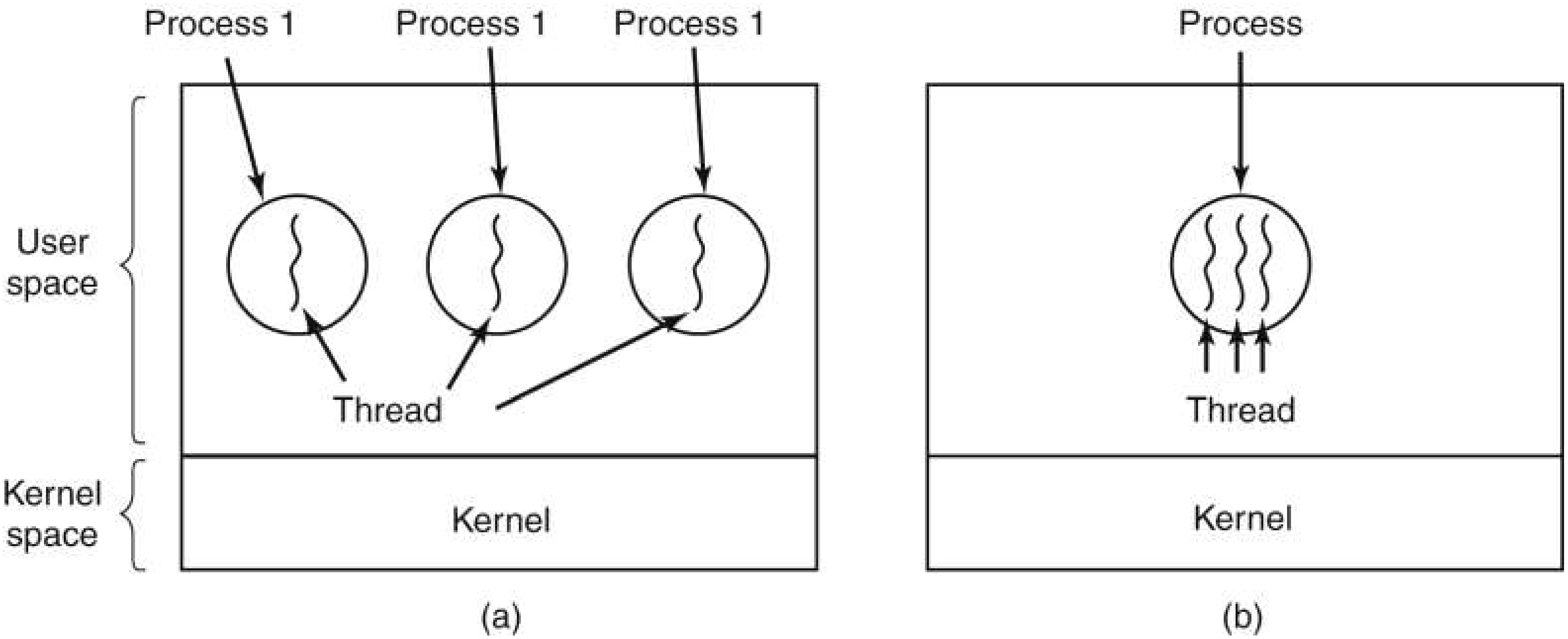
Example 2:
Multithreaded Web Server

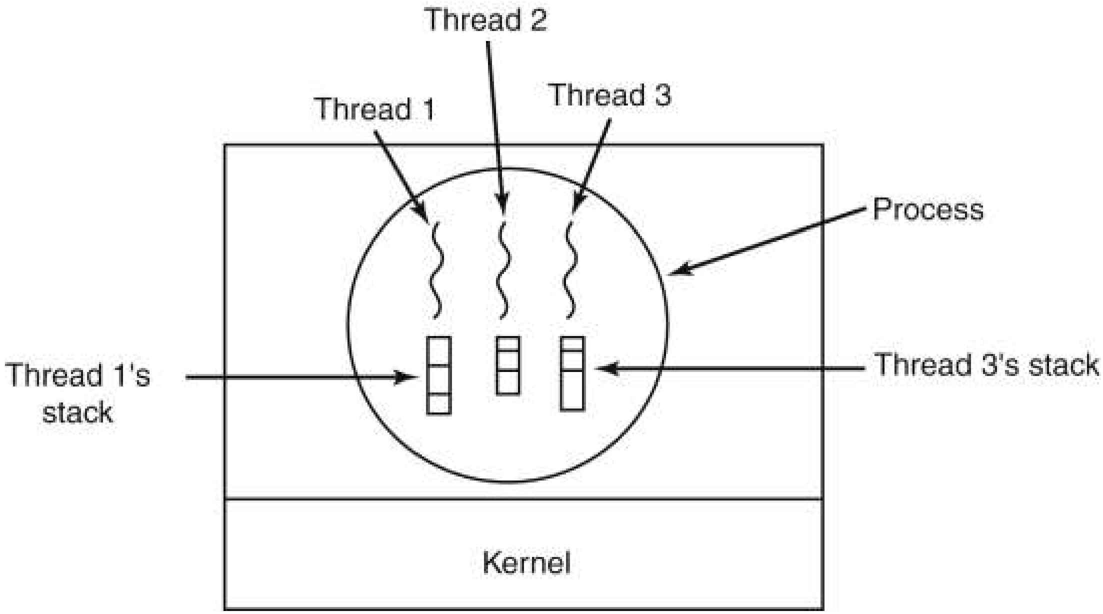


Processes vs Threads

- Process groups resources
- Threads are entities scheduled for execution on CPU
- No protections among threads (unlike processes) [Why?]
- Thread can be in any of several states: running, blocked, ready, and terminated

Per process items	Per thread items
Address space	Program counter
Global variables	Registers
Open files	Stack
Child processes	State
Pending alarms	
Signals and signal handlers	
Accounting information	


Process 1
Process 1
Process 1
Process
User space
Kernel space
Thread
Thread
Kernel
Kernel
(a)
(b)

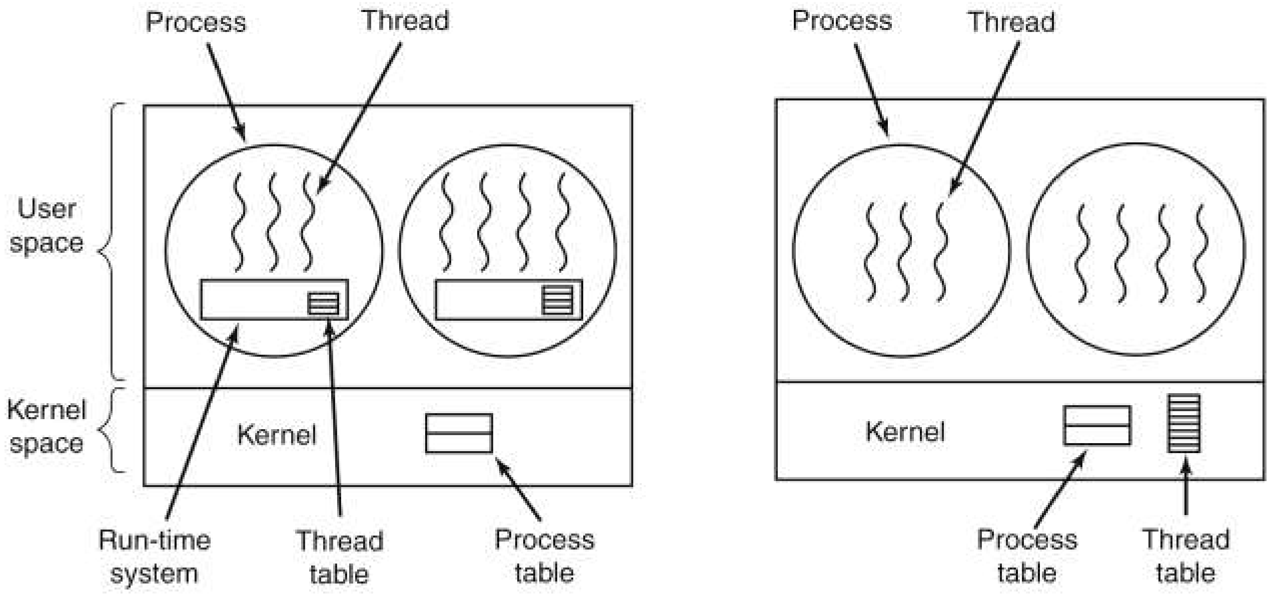


Each thread has its own stack (Why?).

Where to Put The Thread Package?

User space

Kernel space



Implementing Threads in User Space

- Threads are implemented by a library
- Kernel knows nothing about threads
- Each process needs its own private thread table
- Thread table is managed by the runtime system

Implementing Threads in User Space

Advantages

- Very fast thread scheduling
- Each process can have its own thread schduling algorithm
- Scale better

Disadvantages

- Blocking system calls can block the whole process
- Page fault blocks the whole process
- No other thread of the process will ever run unless the running thread voluntarily gives up the CPU

Implementing Threads in Kernel Space

- Kernel knows about and manages the threads
- No runtime is needed in each process
- Creating/destroying/(other thread related operations) a thread involves a system call

Implementing Threads in Kernel Space

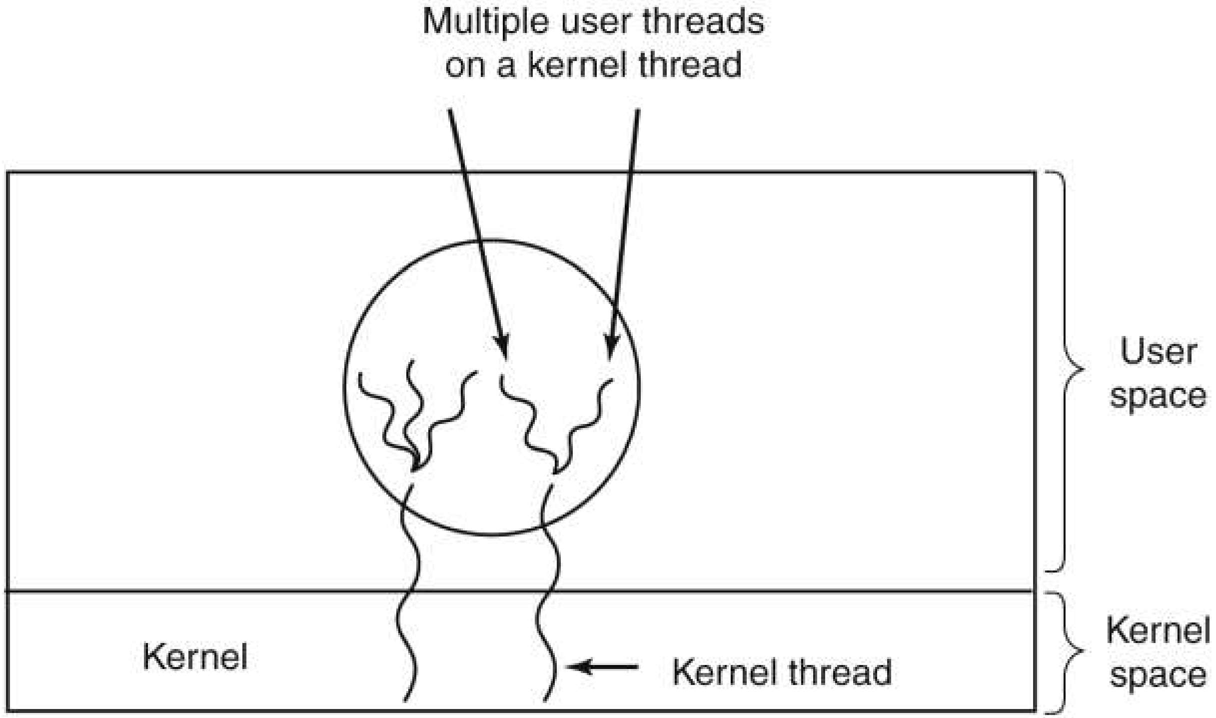
Advantages

- When a thread blocks (due to page fault or blocking system calls) the OS can execute another thread from the same process

Disadvantages

- Cost of system call is very high

Hybrid Implementation



Conclusions

- Processes is the most central concept in OS
- Process vs Thread
- Multiprogramming vs multithreading