Process Description and Control

Process:the concept

Process = a program in execution

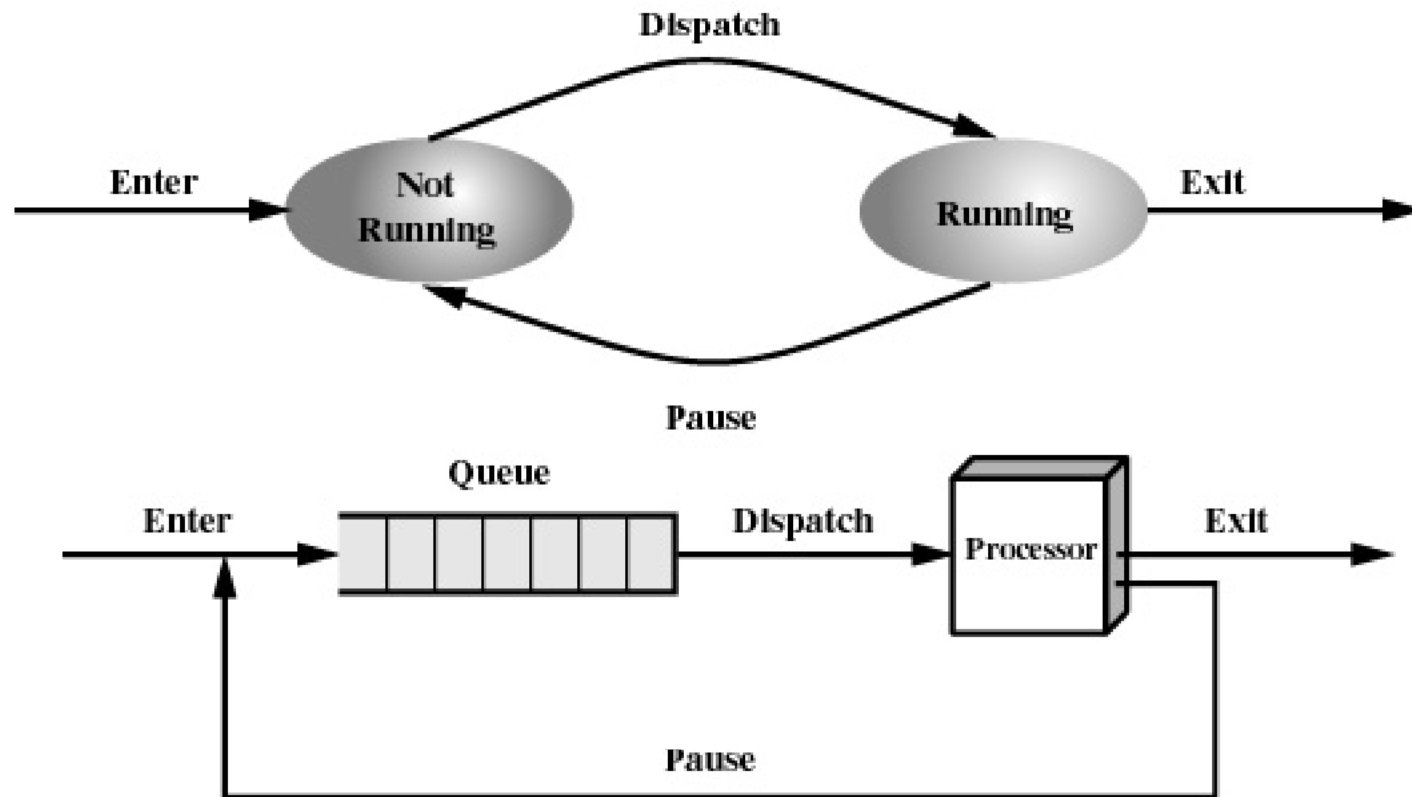
- Example processes:
  - OS kernel
  - OS shell
  - Program executing after compilation
  - www-browser

Process management by OS :

- Allocate resources
- Schedule: interleave their execution (watch for processor utilization, response time)
- Interprocess communication, synchronization (watch for deadlocks; interleaving, nondeterminism imply increased difficulties!)

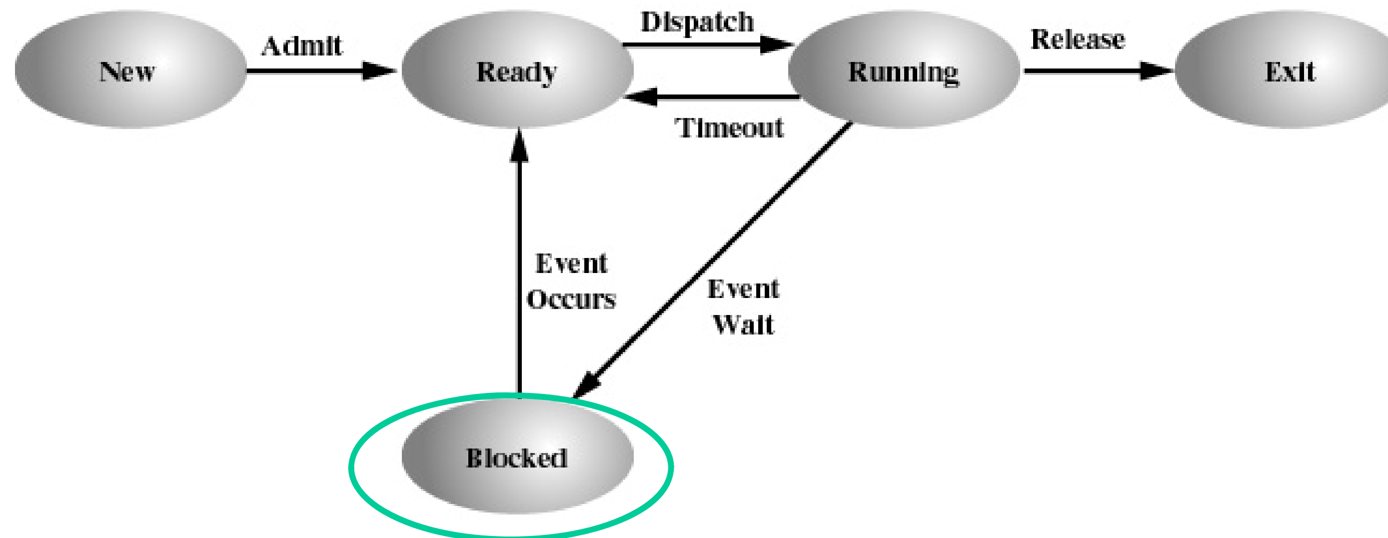
Two-State Process Model

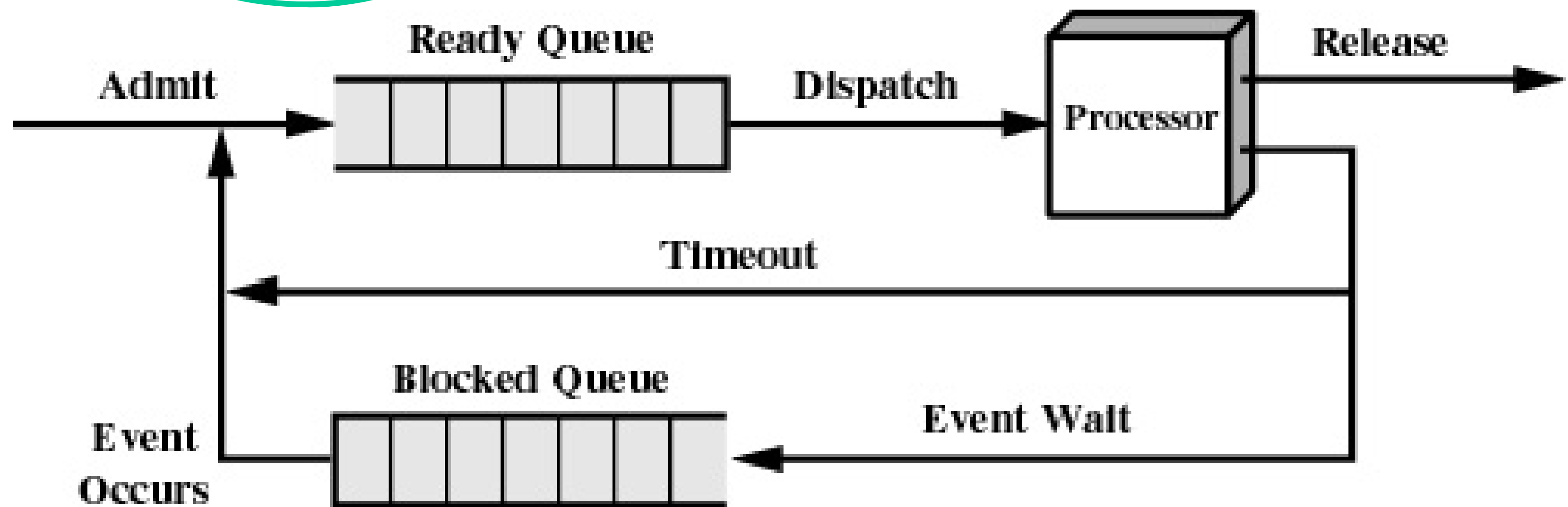
- Process may be in one of two states
  - Running, Not-running
  - Not-Running Process in a Queue



(b) Queuing diagram

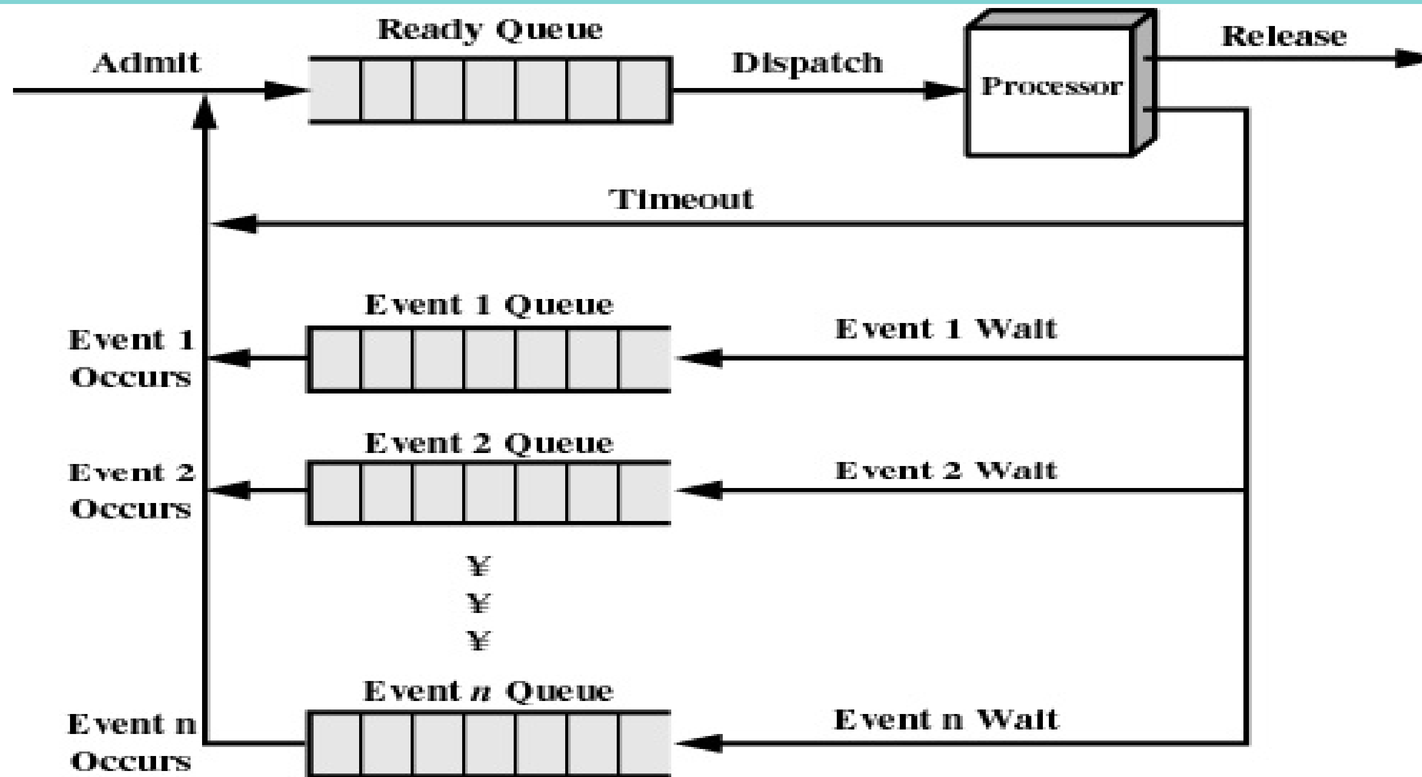
What are not-running processes doing?





(a) Single blocked queue

Actually there are more queues …



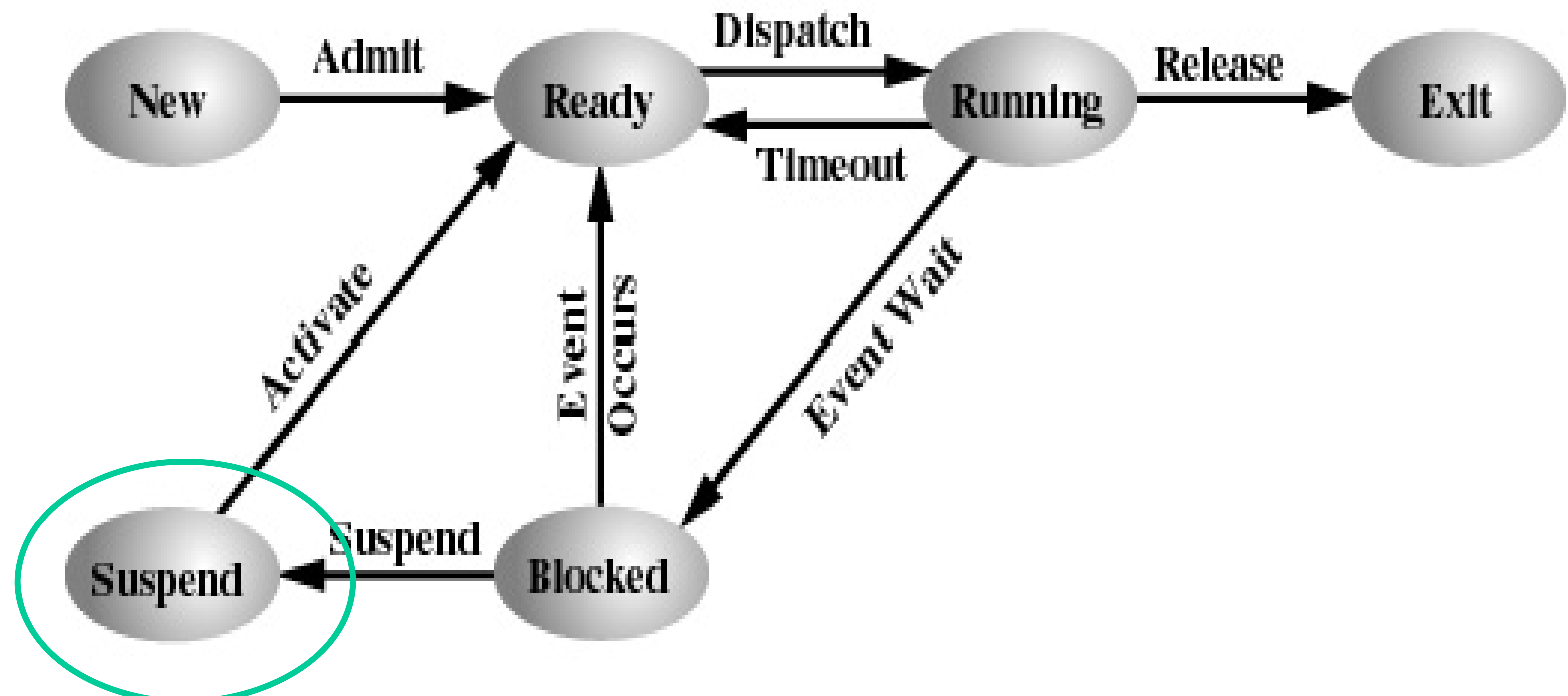
(b) Multiple blocked queues

Observe: I/O's much slower than CPU
What if all processes are waiting for IO and all memory is allocated?
Or if not enough memory for all processes to execute?
Or if …

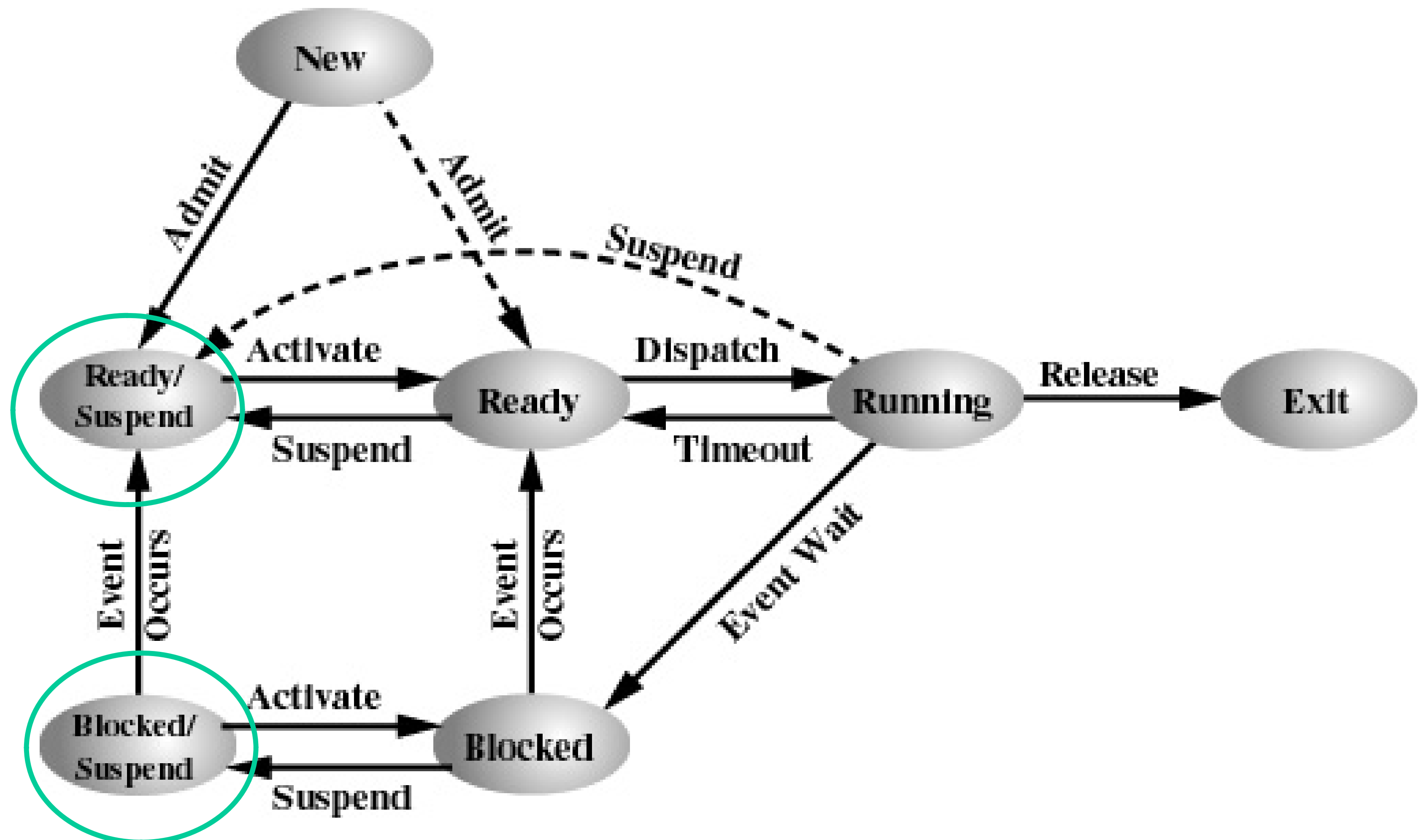
Suspended Processes

Idea: Swap these processes to disk to free up more memory (to admit new processes)

- Blocked state becomes suspend state when swapped to disk



Two Suspend States



(b) With Two Suspend States

Reasons for Process Suspension

Swapping	The operating system needs to release sufficient main memory to bring in a process that is ready to execute.
Other OS reason	The operating system may suspend a background or utility process or a process that is suspected of causing a problem. e.g. to prevent deadlock
Interactive user request	A user may wish to suspend execution of a program for purposes of debugging or in connection with the use of a resource.
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 interval.
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 descendents.

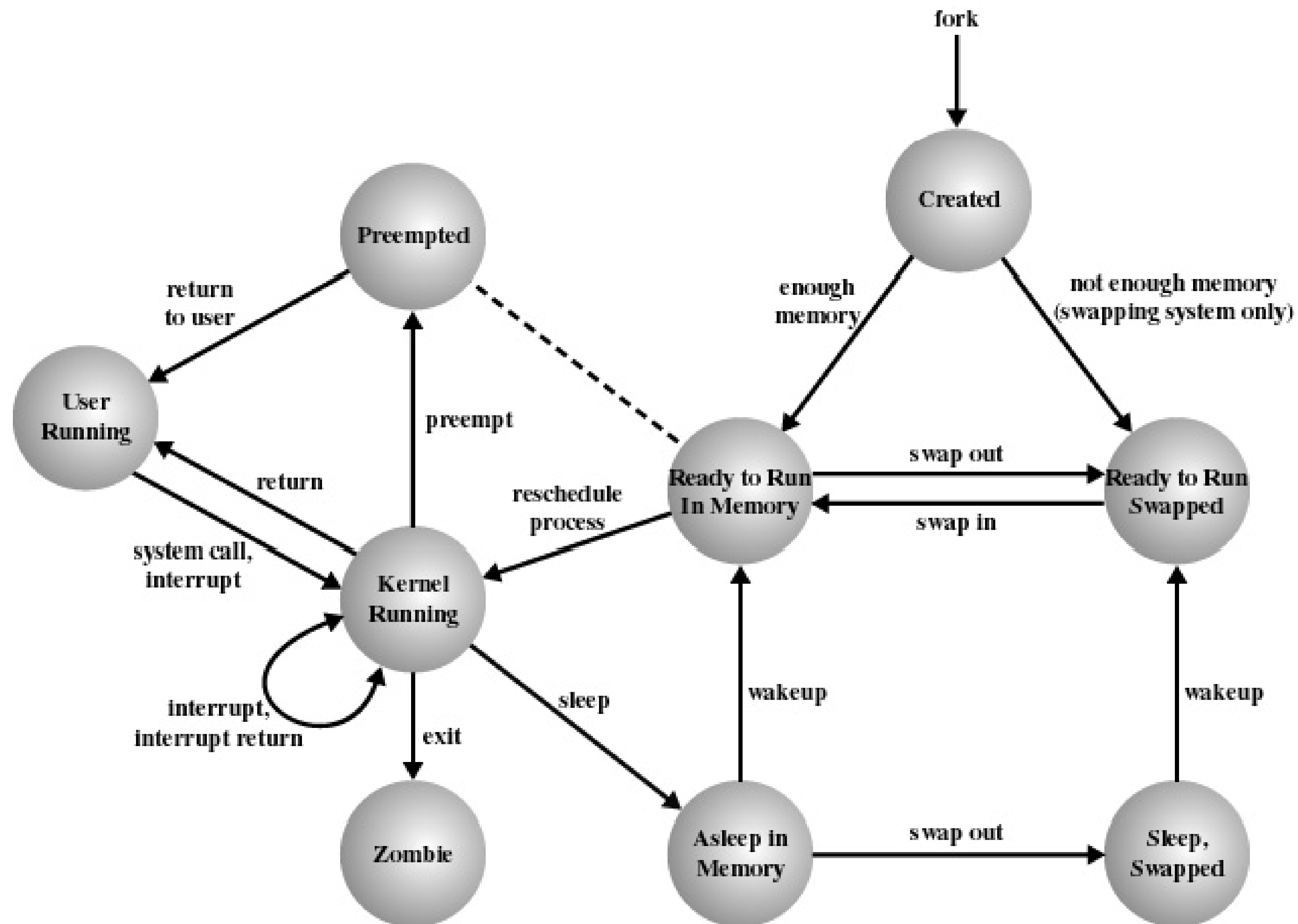
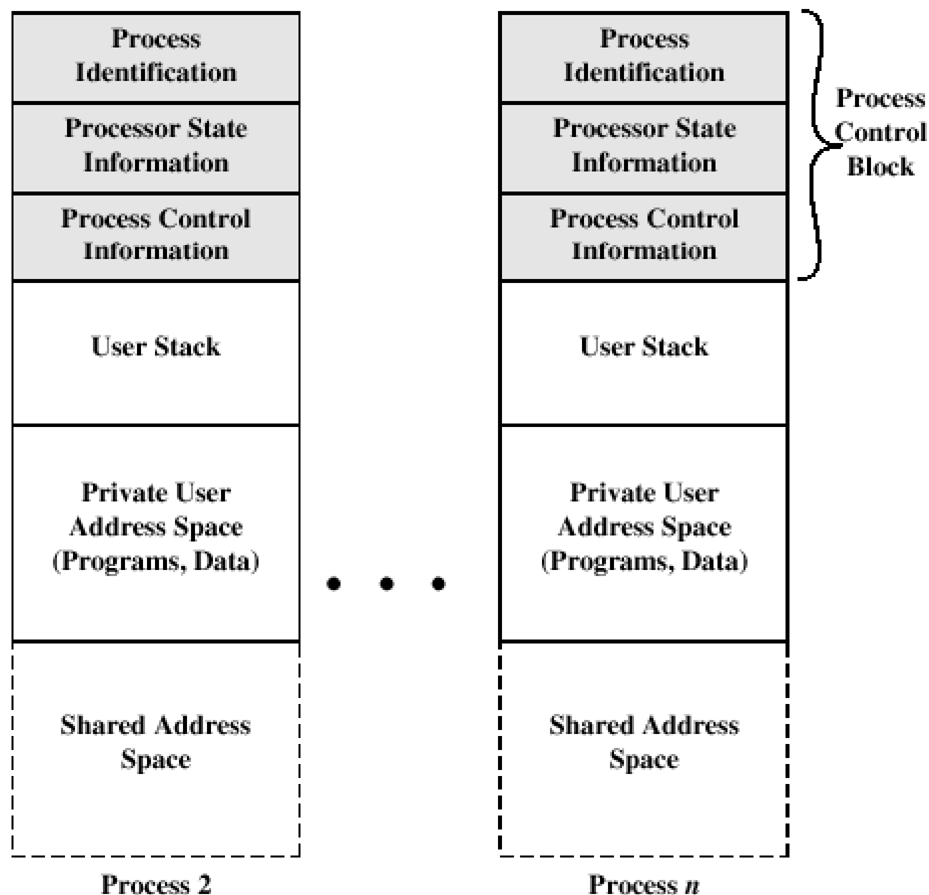


Figure 3.16 UNIX Process State Transition Diagram

OS Control Structures



gure 3.12 User Processes in Virtual Memory

What a process needs in order to execute (process' image):

- Program
- Data
- Stack
- Process Control Block (context; for multiprogramming)

The OS must keep:

- Information about the current status of each process and resource
- Tables are constructed for each entity the operating system manages

Process Table (and other OS tables)

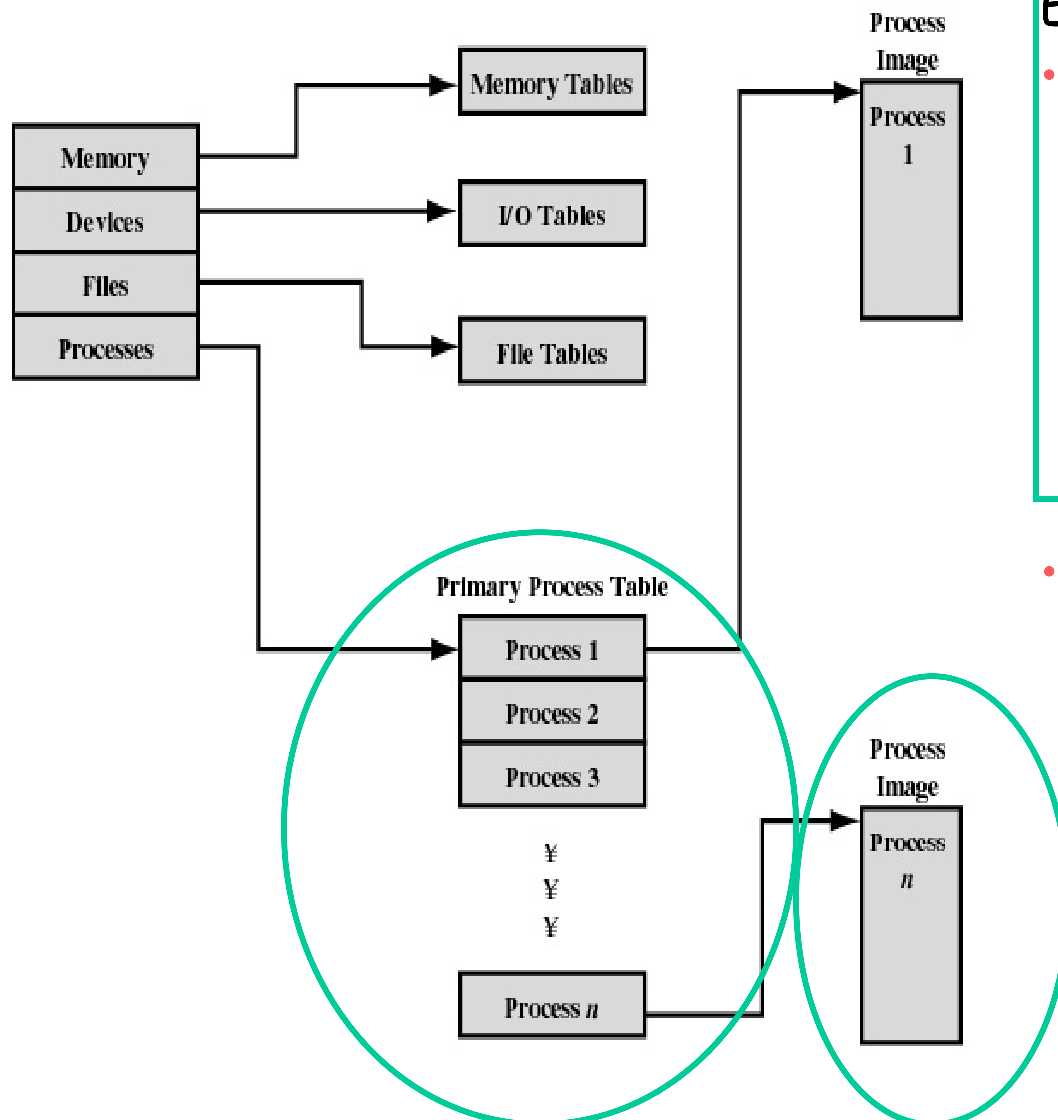
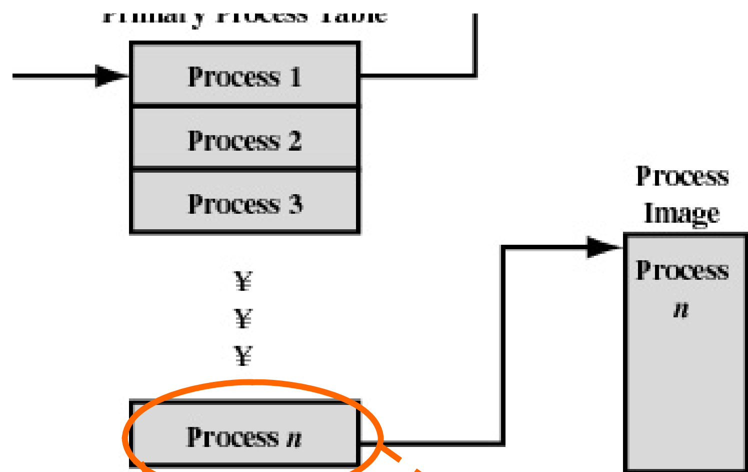


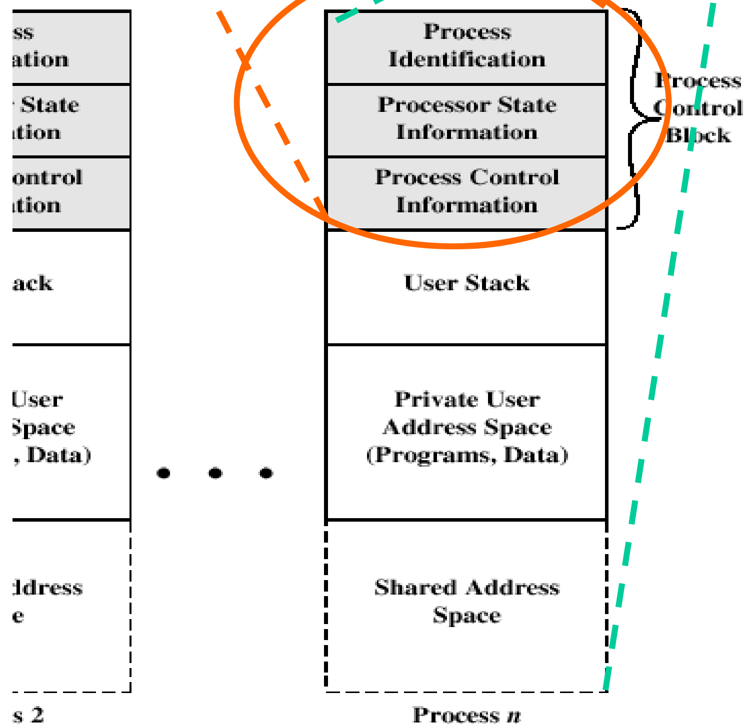
Figure 3.10 General Structure of Operating System Control Tables

Each entry:

- Attributes necessary for its management
  - Process ID
  - Process state
  - Location in memory
  - Etc: process control block

- Other tables hold resource-specific info (zoom into later):
  - How main/secondary memory is allocated,
  - I/O device status, buffer in memory,…
  - File status, location, attributes, …

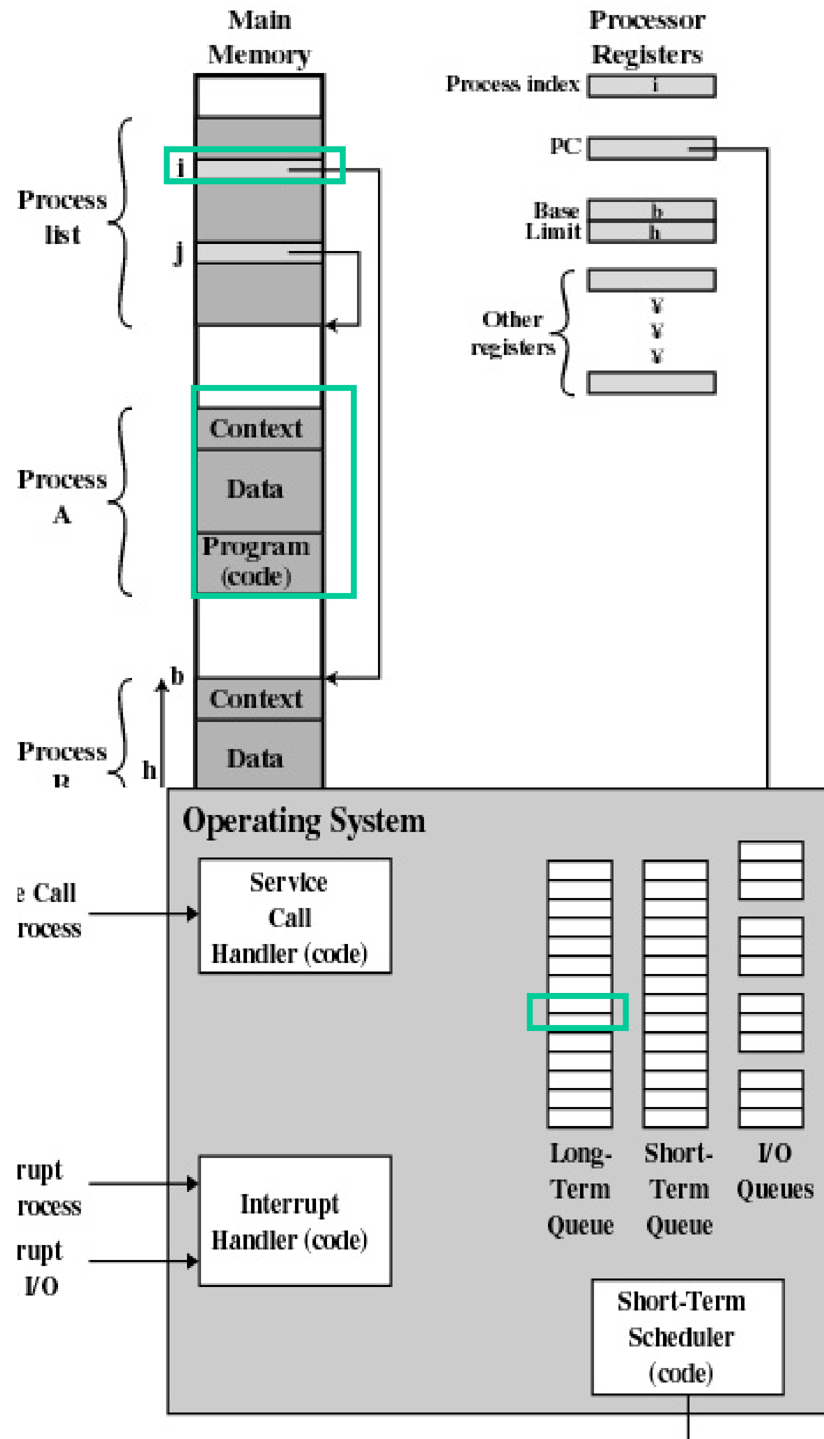




Process Control Block

PCB contains:

- Identifiers (process, parent process, user, …)
- Processor State Information (register values: must be copied and restored in state transitions: running↔ready,…)
- Other Process Control Information:
  - Scheduling and State Information (priority, event awaited, …)
  - Process memory tables
  - Resources (open files, ownership, …)
  - Links (to other process in a queue, …)
  - Privileges
  - …



Process Creation

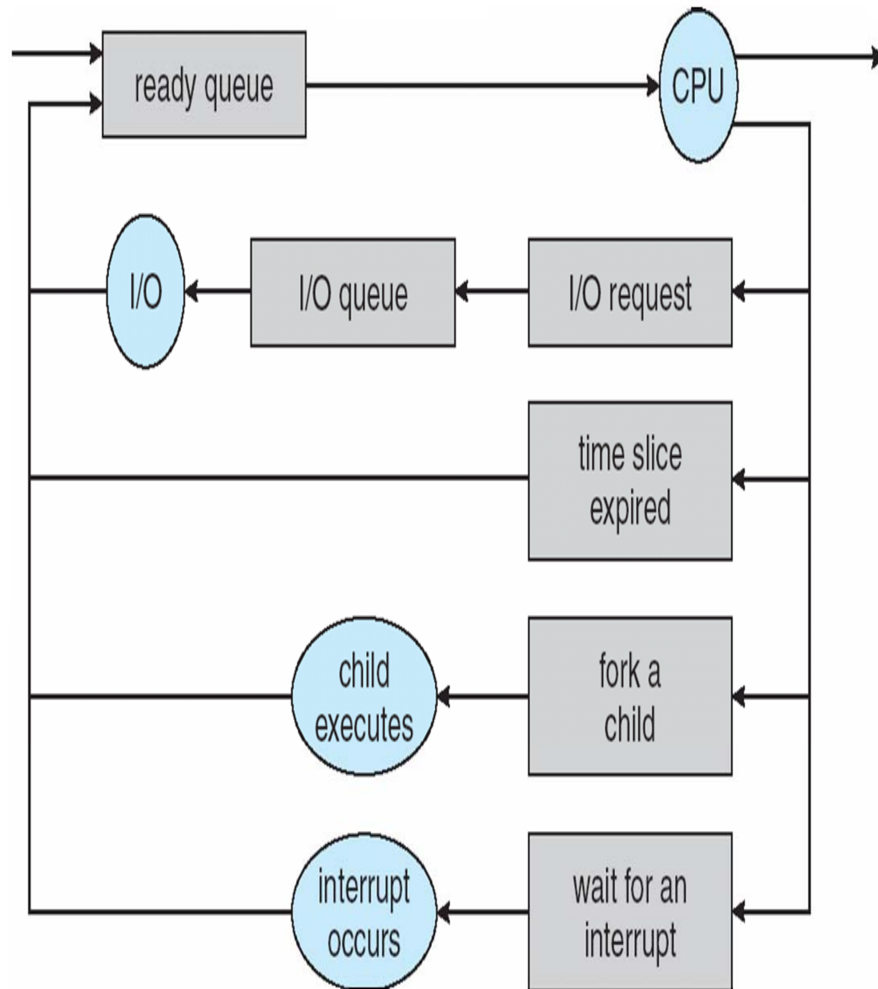
Examples:

- Execution of a compiled user program
- User logs on (shell starts executing)
- Process creation to provide a service (e.g. printing: user executes lpr)
- A process creates another process (e.g. shell creates lpr; user programs can create processes, as well)

OS must:

- Assign a unique process identifier
- Allocate space for the process
- Initialize process control block
- Set up appropriate linkages
  - "Include" the process in the system (in some queue(s), …)

Representation of Process Scheduling /when to switch a running process



Clock interrupt

process has executed for the maximum allowable time slice

I/O interrupt

Memory fault

memory address is in virtual memory so it must be brought into main memory

Trap

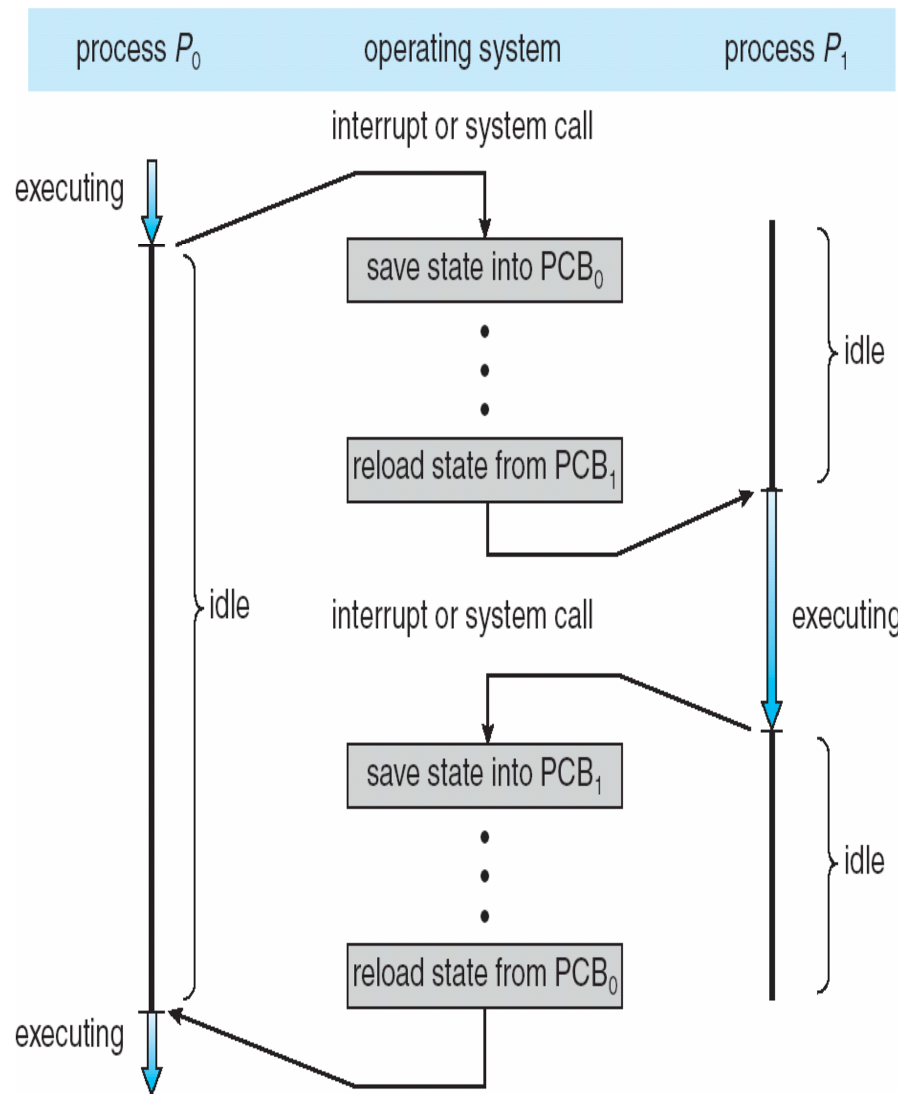
error occurred

may cause process to be moved to Exit state

Supervisor call (or system call)

such as file open

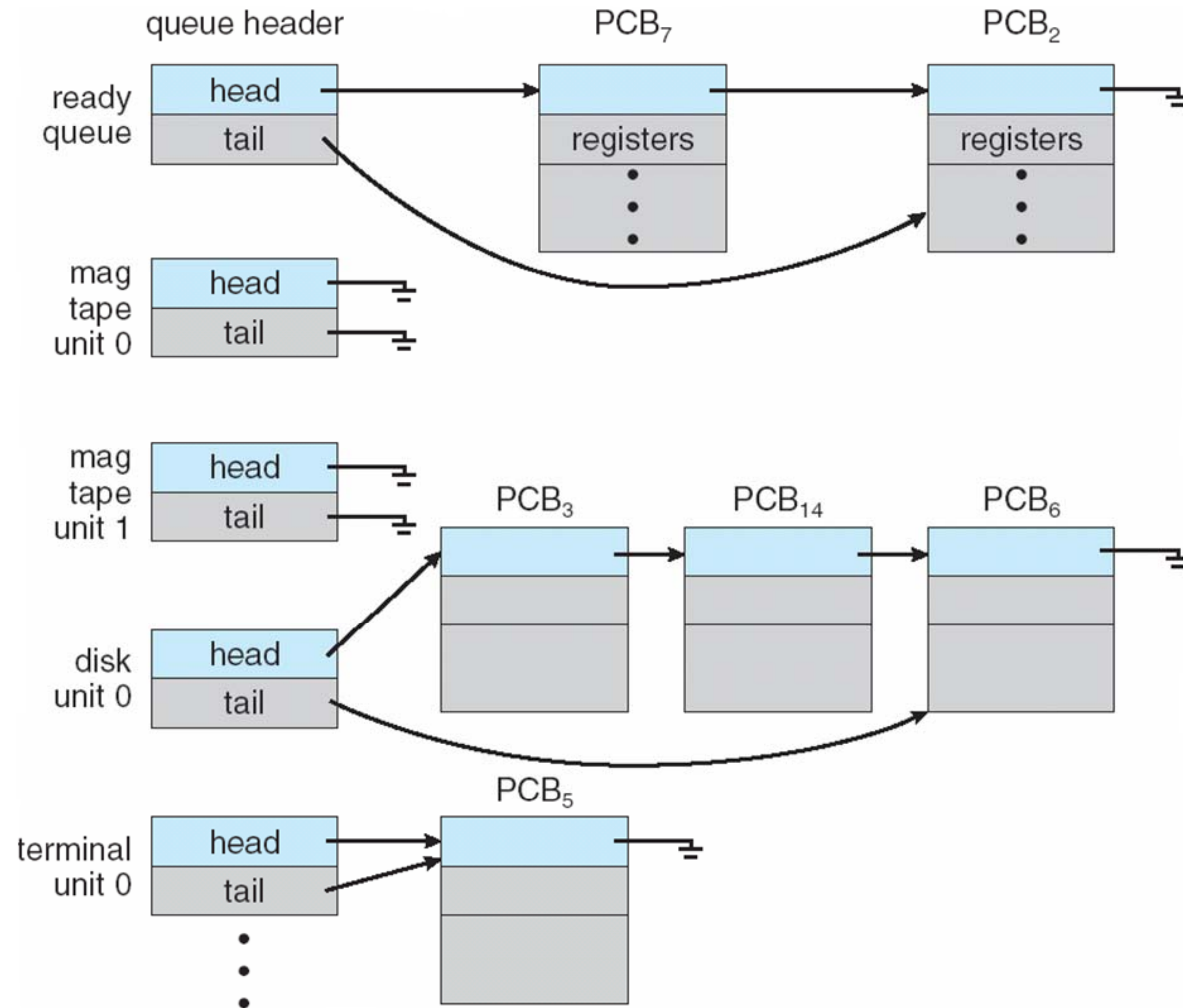
Change of Process State (context switching)



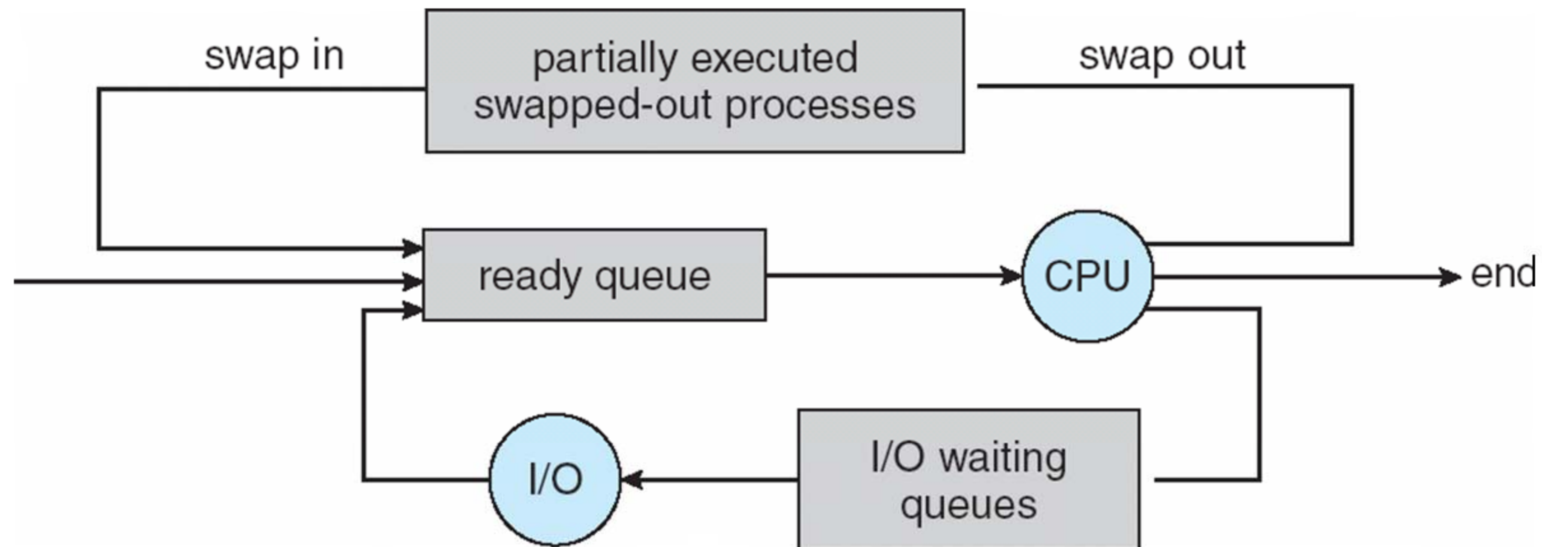
running→other

- Save context of processor including program counter and other registers (in PCB)
- Move PCB to appropriate queue (ready, blocked, …)
- Select another process for execution
- Update the PCB of the process selected
- Update memory-management data structures
- Restore context (in processor) of the selected process

Ready Queue And Various I/O Device Queues



Addition of Medium Term Scheduling (swap-out/suspension)



Interprocess Communication

- Processes within a system may be **independent** or **cooperating**
- Reasons for cooperating processes:
  - Information sharing
  - Computation speedup
  - Modularity, Convenience
- Cooperating processes need **interprocess communication (IPC)**
- Two models of IPC
  - Shared memory
  - Message passing

(more in connection with process synchronization/coordination)

Threads

Processes and Threads

multithreading: more than one entities can possibly execute in the same resource- (i.e. process-) environment (and collaborate better)

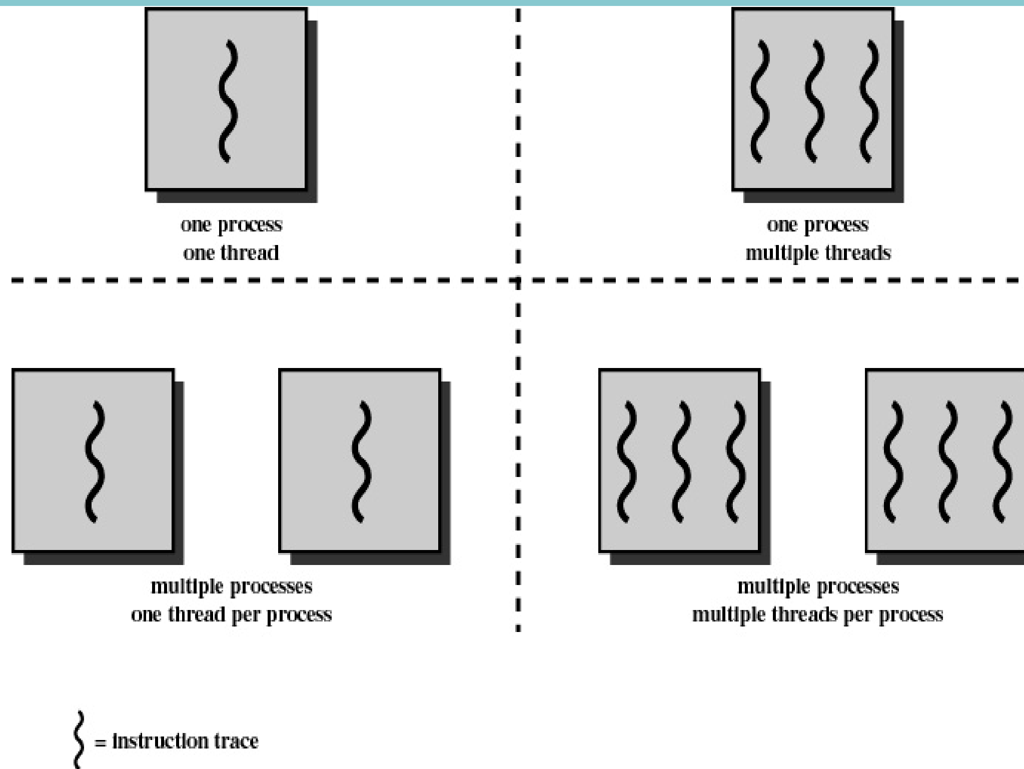


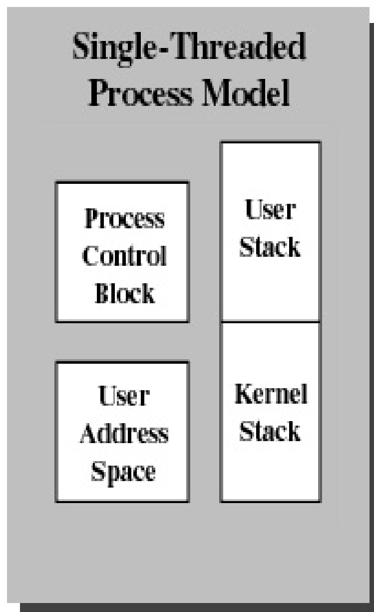
Figure 4.1 Threads and Processes [ANDE97]

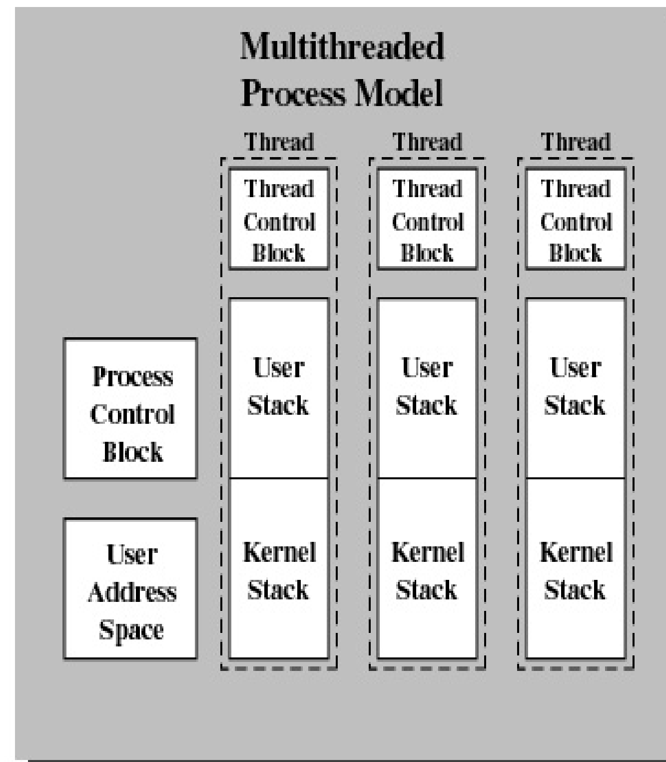
Unit of ...

- ... dispatching is referred to as a thread
- ... resource ownership is referred to as a process or task

A process has … A thread has …

- a virtual address space which holds the process image
- global variables, files, child processes, signals and signal handlers





- an execution state, stack and context (saved when not running)
- access to the memory and resources of its process
  - all threads of a process share this
- some per-thread static storage for local variables

Suspension and termination

- Suspending a process involves suspending all threads of the process since all threads share the same address space
- Termination of a process, terminates all threads within the process

Benefits of Threads

- Since threads within the same process share memory and files, they can communicate with each other without invoking the kernel
- May allow parallelization within a process:
  - I/O and computation to overlap (remember the historical step from uniprogramming to multiprogramming?)
  - concurrent execution in multiprocessors
- Takes less time to
  - create/terminate a thread than a process
  - switch between two threads within the same process

Uses of Threads

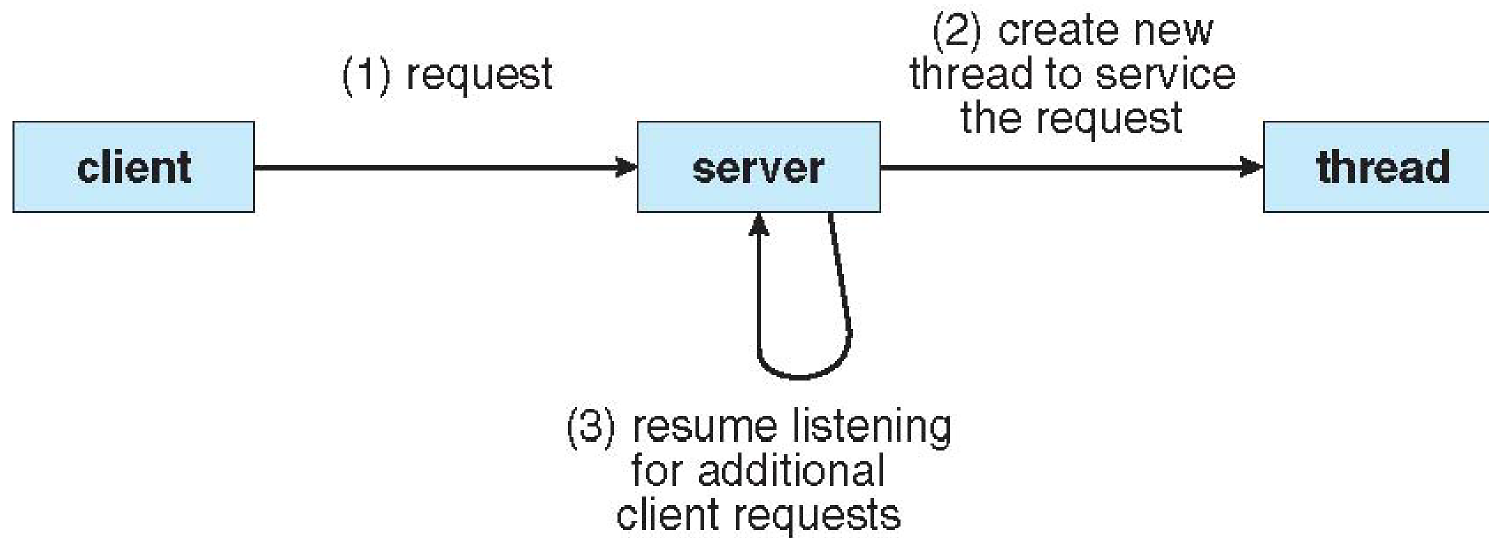
- Overlap foreground (interactive) with background (processing) work
- Asynchronous processing (e.g. backup while editing)
- Speed execution (parallelize independent actions)
- Modular program structure (must be careful here, not to introduce too much extra overhead)

- Cf. multicores/multiprocessor systems

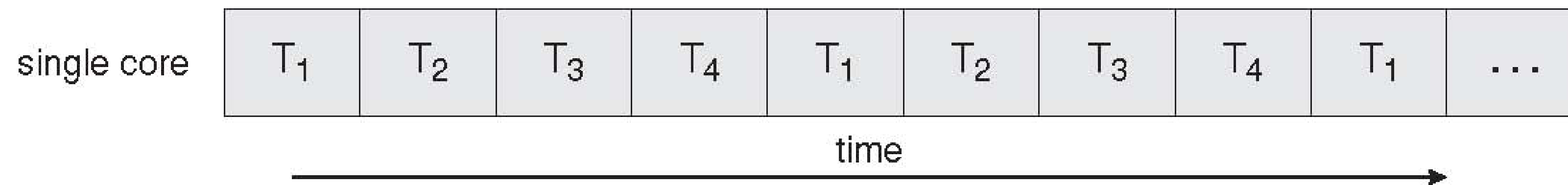
Multicore Programming

- Multicore systems putting pressure on programmers, challenges include
  - **Dividing activities**
  - **Balance**
  - **Data splitting**
  - **Data dependency**
  - **Testing and debugging**

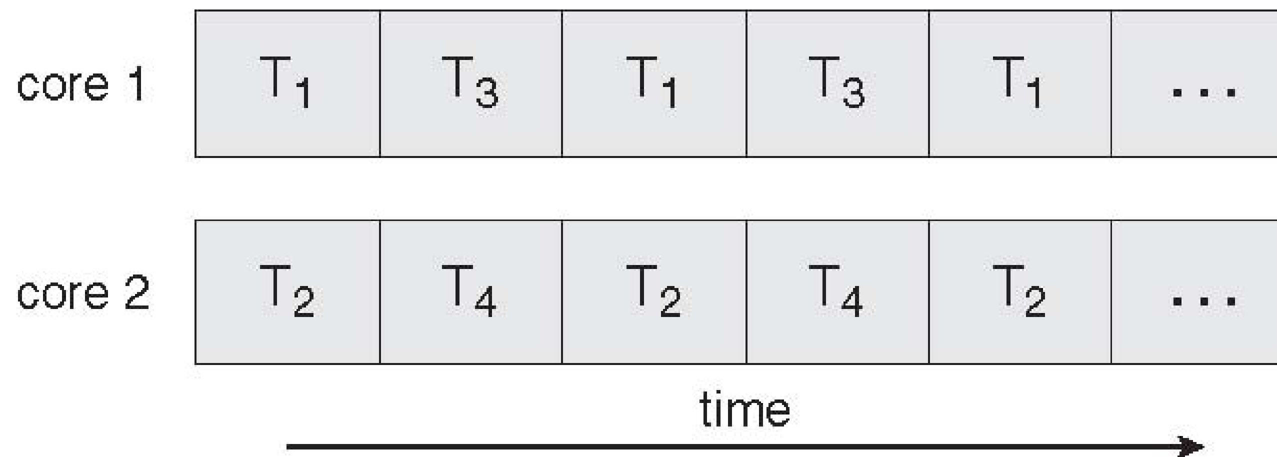
e.g. Multithreaded Server Architecture

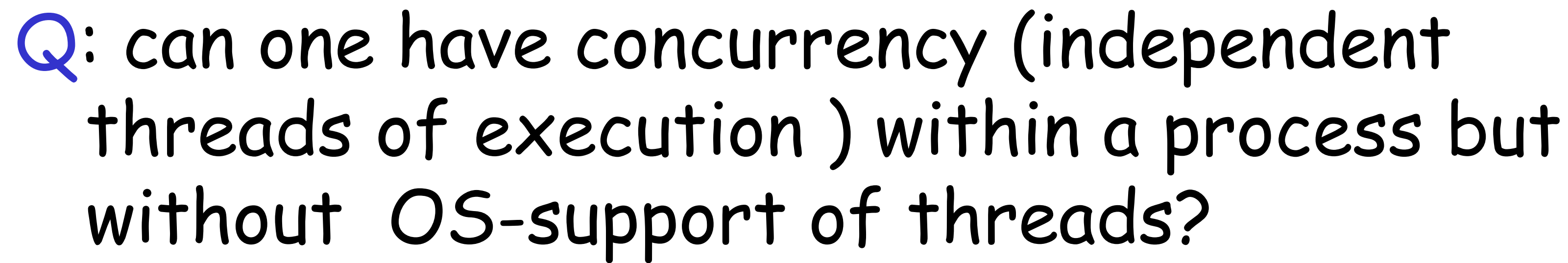


Concurrent Execution on a Single-core System



Parallel Execution on a dual-core System





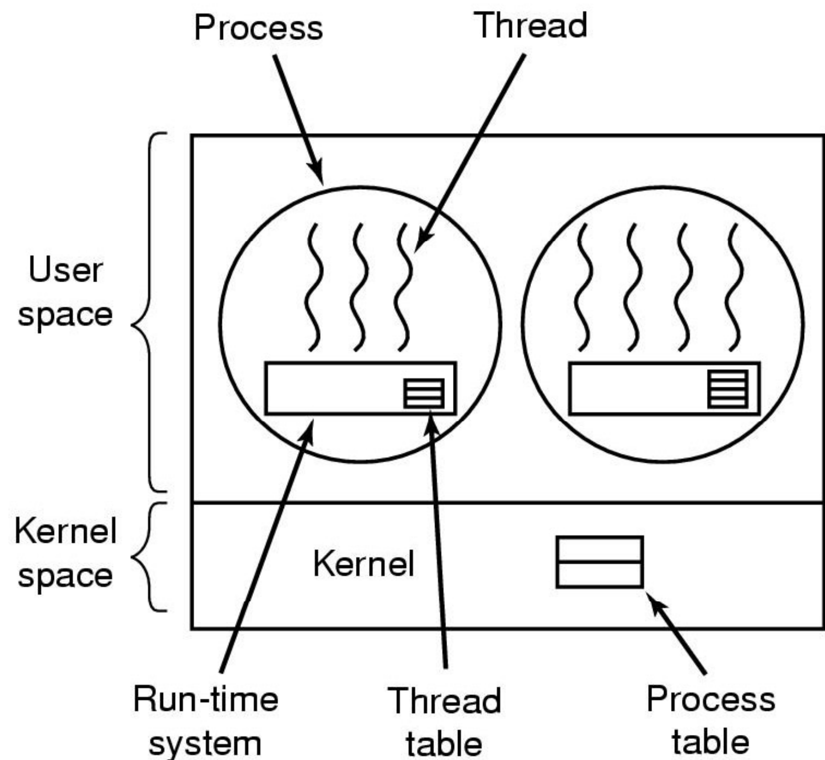
Q: can one have concurrency (independent threads of execution) within a process but without OS-support of threads?

Think of Signals

- Signals (similar to interrupts) can notify a process that a particular event has occurred
- A signal handler is used to process signals
  1. Signal is generated by particular event e.g. timer or non-blocking IO (cf select system call)
  2. Signal is delivered to a process
  3. Signal is handled

Implementing Threads in User Space

The kernel is not aware of the existence of threads;



- Run-time system (thread-library in execution) is responsible for bookkeeping, scheduling of threads
- allows for customised scheduling
- can run on any OS
- But: problem with blocking system calls (when a thread blocks, the whole process blocks; i.e other threads of the same process cannot run)

Fig: Tanenbaum, Modern OS, 2/e

Implementing Threads in the Kernel

Kernel maintains context information for the process and the threads

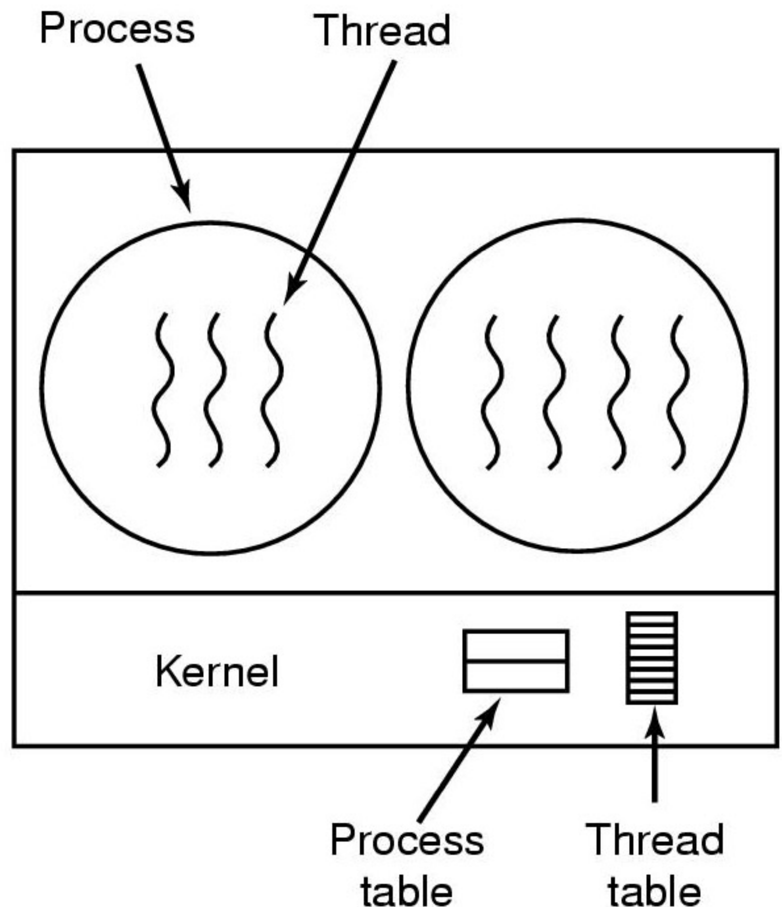
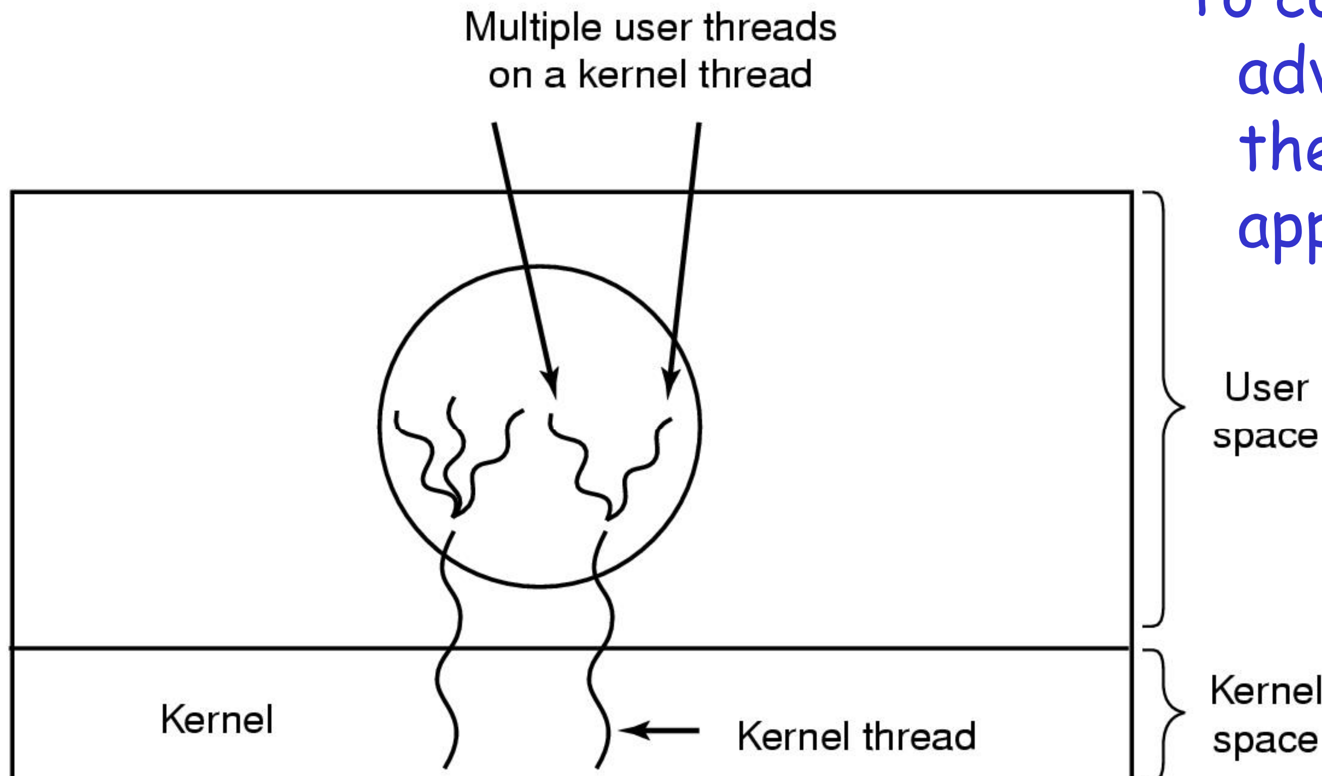


Fig: Tanenbaum, Modern OS, 2/e

- Scheduling is done on a thread basis
- Does not suffer from "blocking problem"
- Less efficient than user-level threads (kernel is invoked for thread creation, termination, switching)

Hybrid Implementations

Multiplexing user-level threads onto kernel- level threads



To combine the advantages of the other two approaches

Fig: Tanenbaum, Modern OS, 2/e

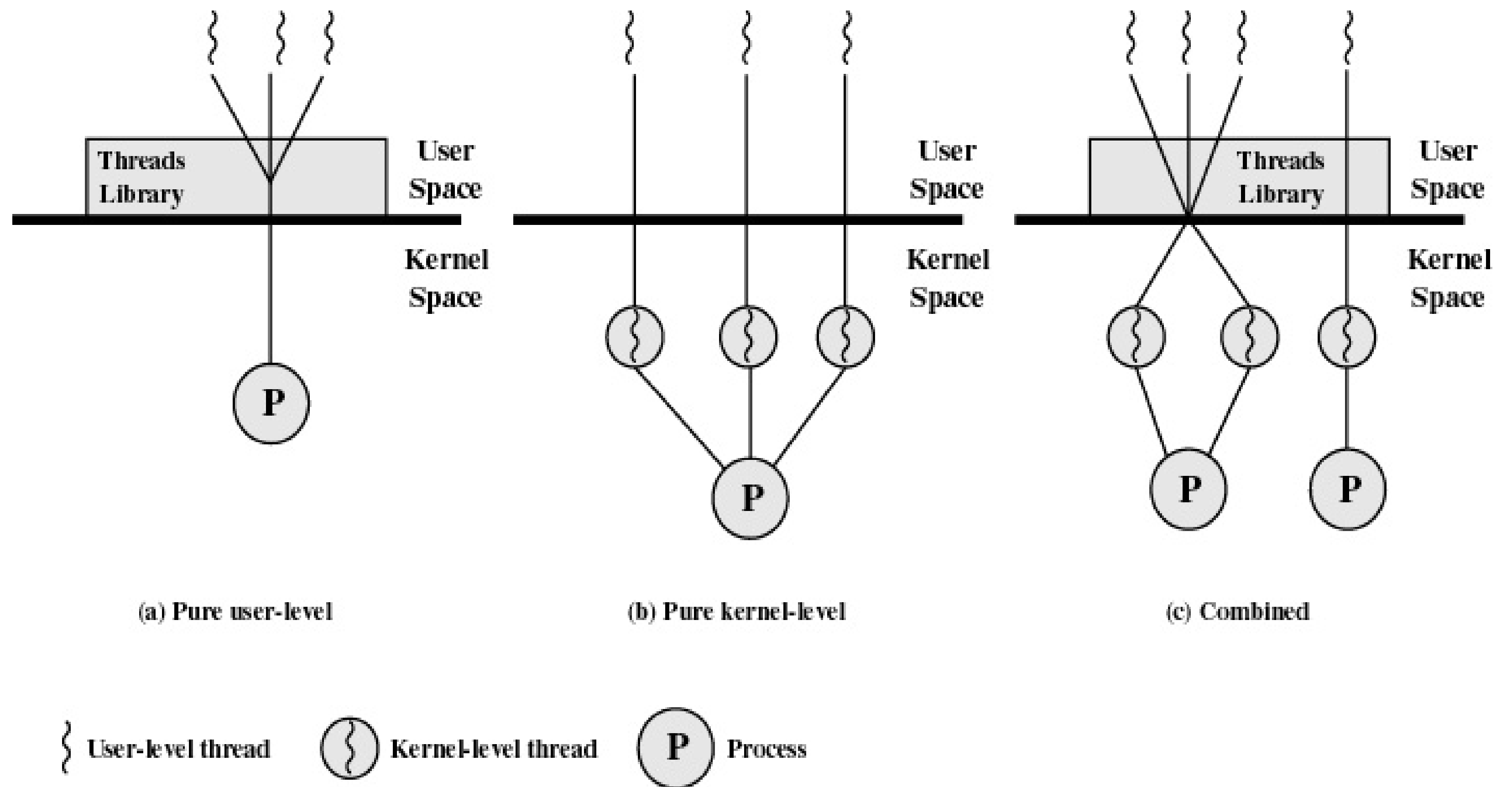


Figure 4.6 User-Level and Kernel-Level Threads

Examples:

- Posix Pthreads: (IEEE) standard:
  - Specifies interface
  - Implementation (using user/kernel level threads) is up to the developer(s)
  - More common in UNIX-like systems (linux, mac-os)
- Win32 thread library:
  - Kernel-level library, windows systems
- Java threads:
  - Supported by the JVM (*VM: a run-time system, a general concept, with deeper roots and potential future in the systems world*)
  - Implementation is up to the developers –e.g. can use Pthreads API or Win32 API, etc

Examples (cont)

- Solaris: hybrid model
  - User-level threads
  - Lightweight processes
  - Kernel threads

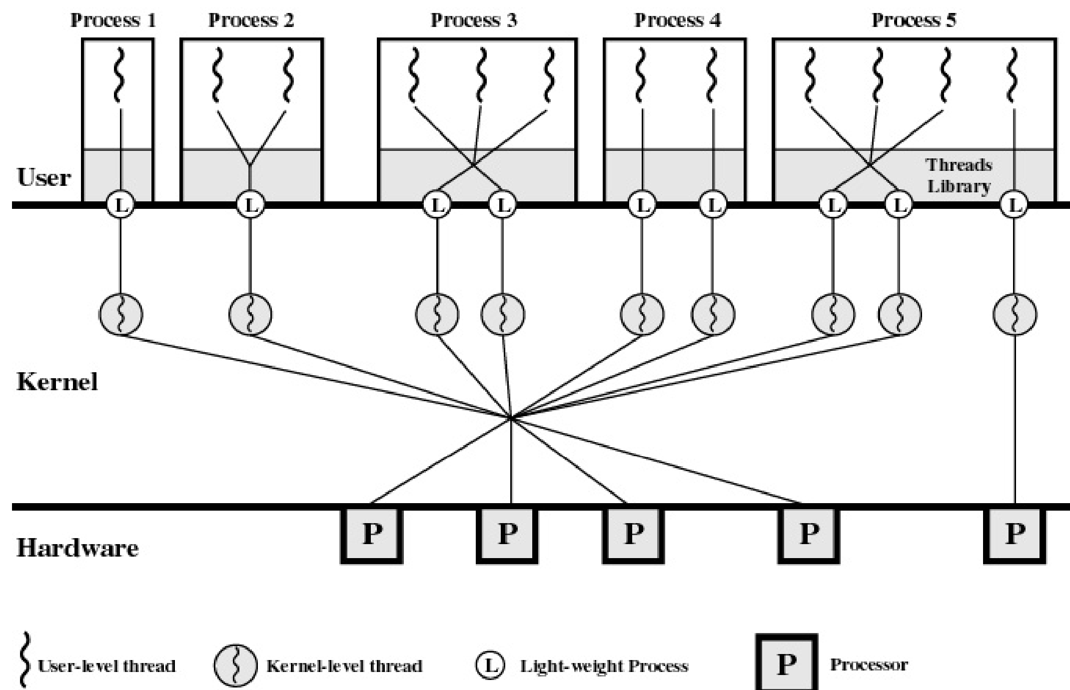


Figure 4.15 Solaris Multithreaded Architecture Example

- W2K, XP: hybrid model
  - Thread: kernel (use win32 API)
  - Fiber: user-level thread (library)

Linux Threads

- Linux refers to them as *tasks* rather than *threads*
- Thread creation is done through **clone()** system call
  - **clone()** allows a child task to share the address space of the parent task (process)

flag	meaning
CLONE_FS	File-system information is shared.
CLONE_VM	The same memory space is shared.
CLONE_SIGHAND	Signal handlers are shared.
CLONE_FILES	The set of open files is shared.

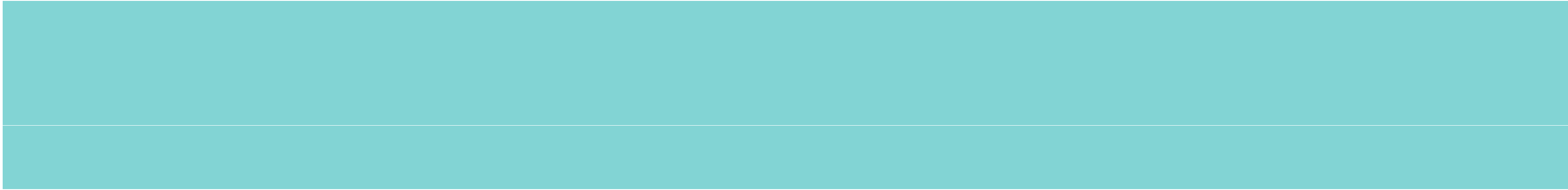
Thread Pools

- Create a number of threads in a pool where they await work
- Advantages:
  - Usually slightly faster to service a request with an existing thread than create a new thread
  - Allows the number of threads in the application(s) to be bound to the size of the pool

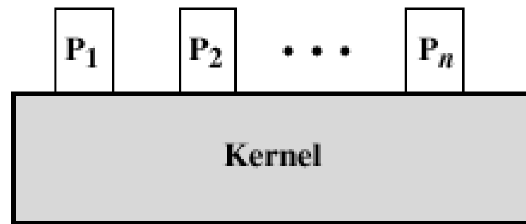
Scheduler Activations

- communication to maintain the appropriate number of kernel threads allocated to the application
  - To gain performance of user space threads
- upcalls - a communication mechanism from the kernel to the thread library/run-time system
  - Kernel assigns/allocates virtual processors (kernel threads)to each process
  - "signals" upcalls to run-time system upon blocking and unblocking of (user-level) threads
- allows an application to maintain the correct number kernel threads

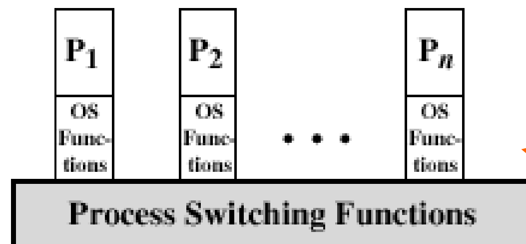
- Concern:
  Fundamental reliance on kernel (lower layer)
  calling procedures in user space (higher layer)



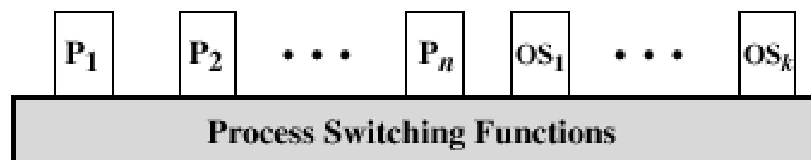
Execution of the Operating System



(a) Separate kernel



(b) OS functions execute within user processes



(c) OS functions execute as separate processes

- Non-process Kernel
  - operating system code is executed as a separate entity that operates in privileged mode
- Process-Based Operating System
  - major kernel functions are separate processes
  - Useful in multi-processor or multi-computer environment
- Execution Within User Processes
  - operating system software within context of a user process
  - process executes in privileged mode when executing operating system code