

Lecture 5:

Process Description and Control
Multithreading
Basics in Interprocess communication
Introduction to multiprocessors

Process:the concept

Process = a program in execution

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

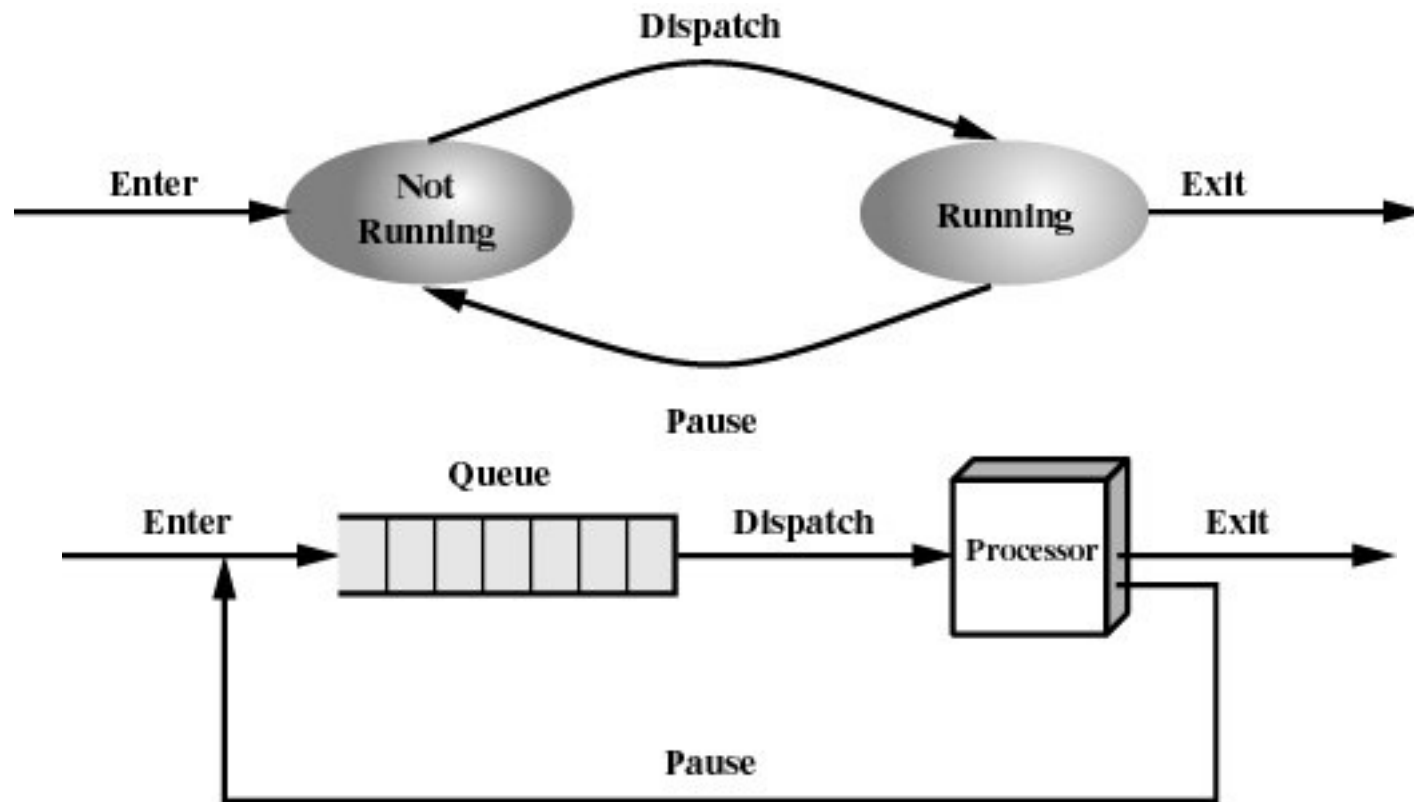
Process management by OS :

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

Process' lifetime:

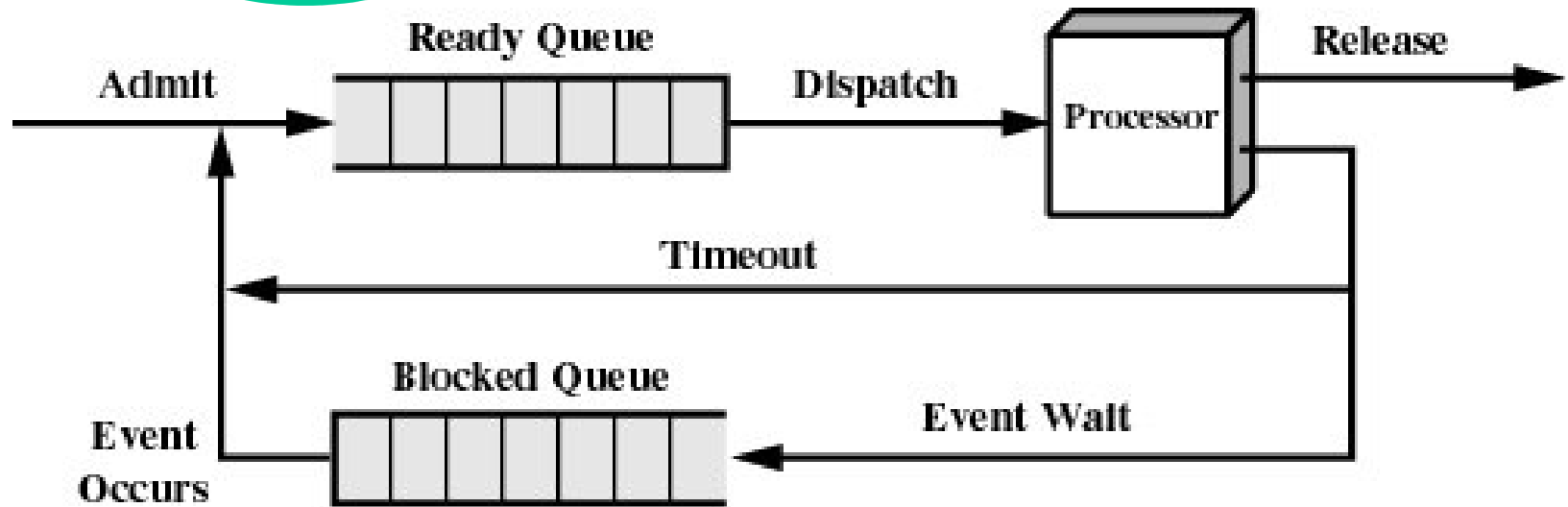
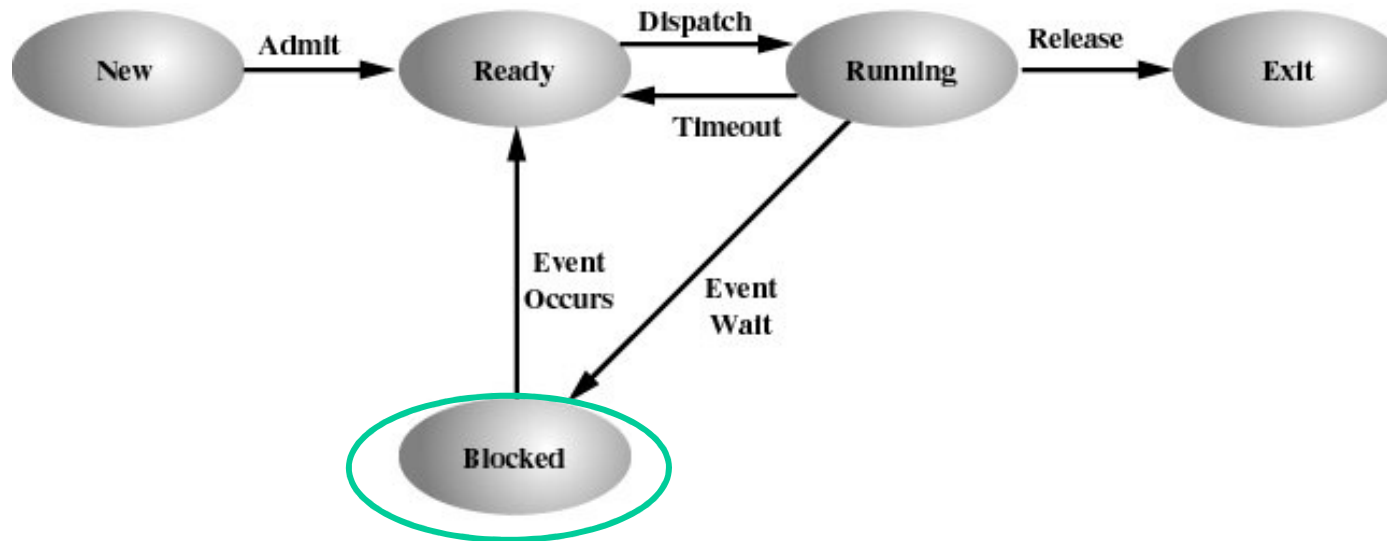
1. Two-State Process Model

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



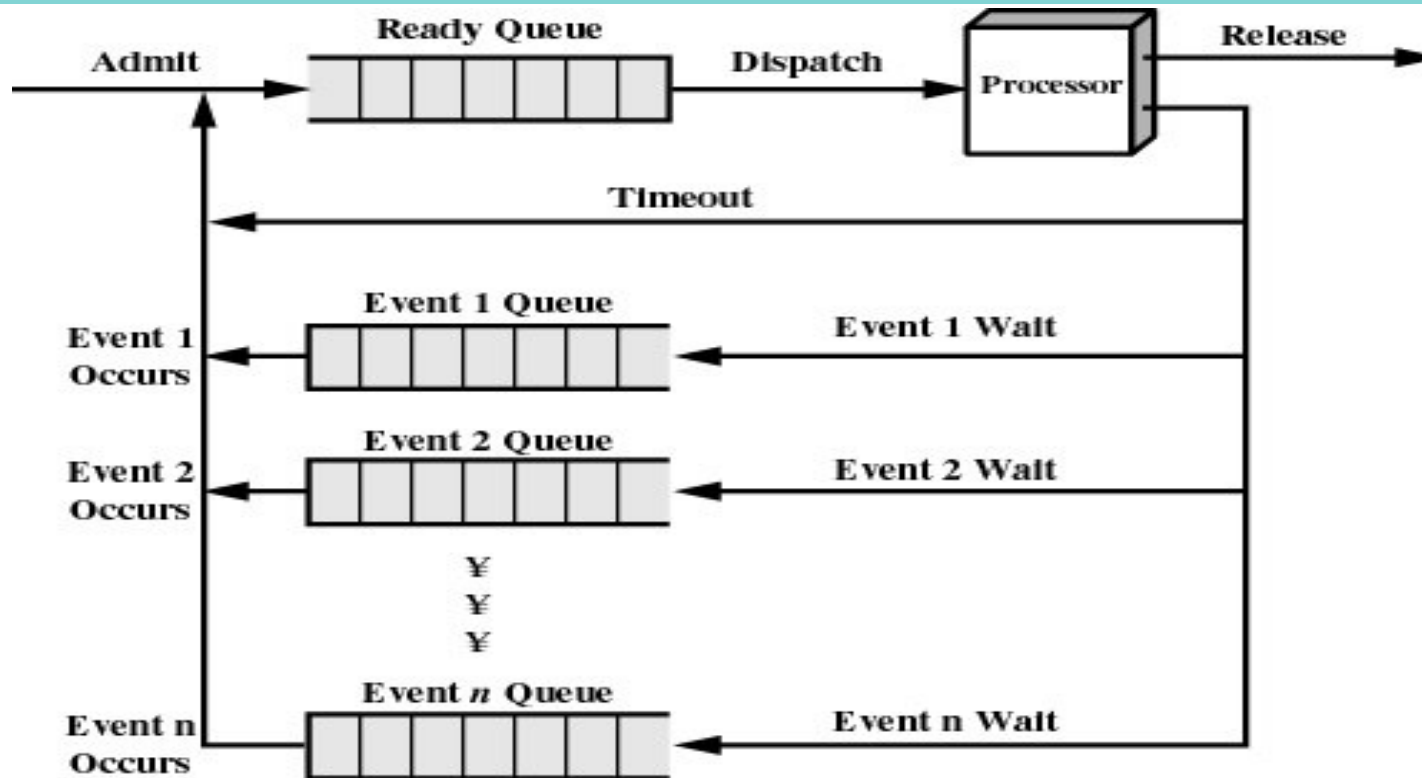
(b) Queuing diagram

2. What are not-running processes doing?



(a) Single blocked queue

3. Actually there are more queues to wait on ...



Observe: I/O's much slower than CPU

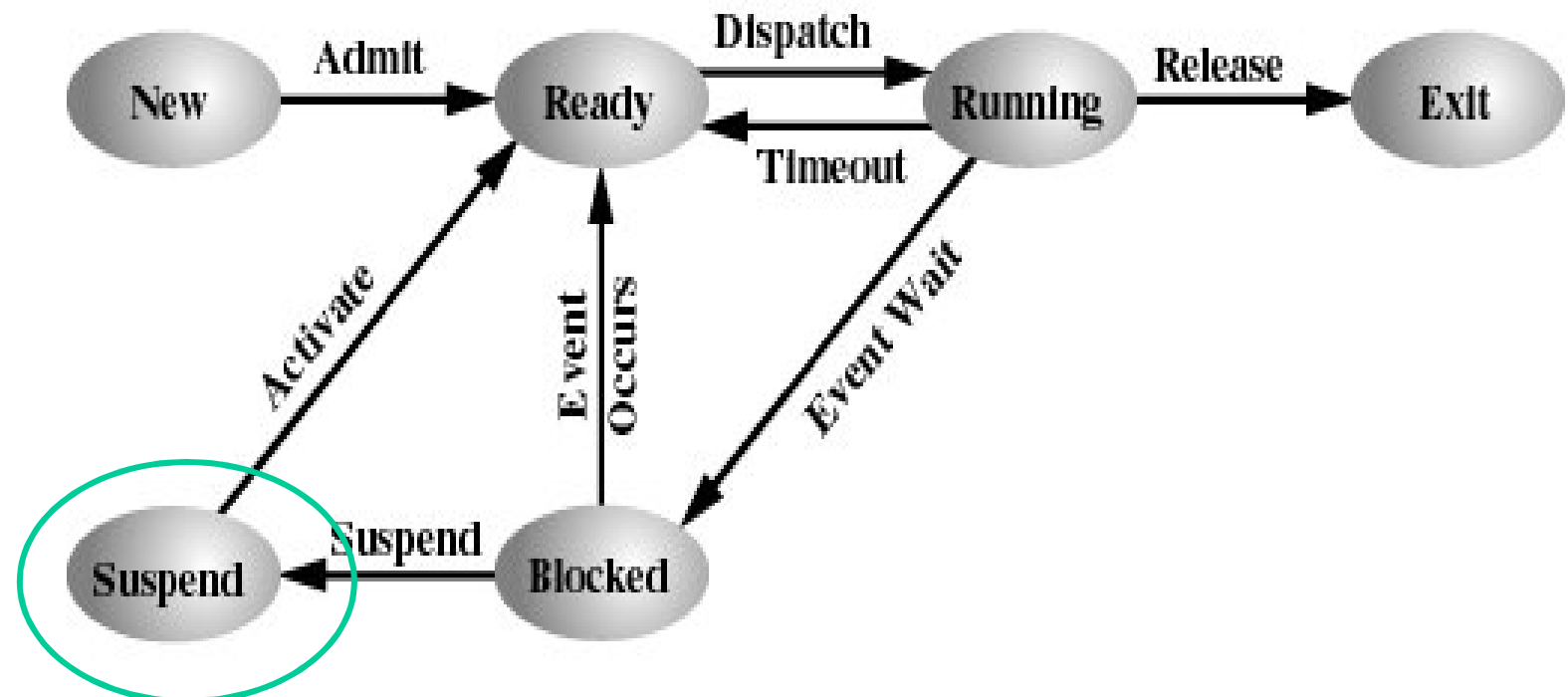
Question: What if all processes are waiting for IO and all memory is allocated?

Or if not enough memory for all processes to execute?

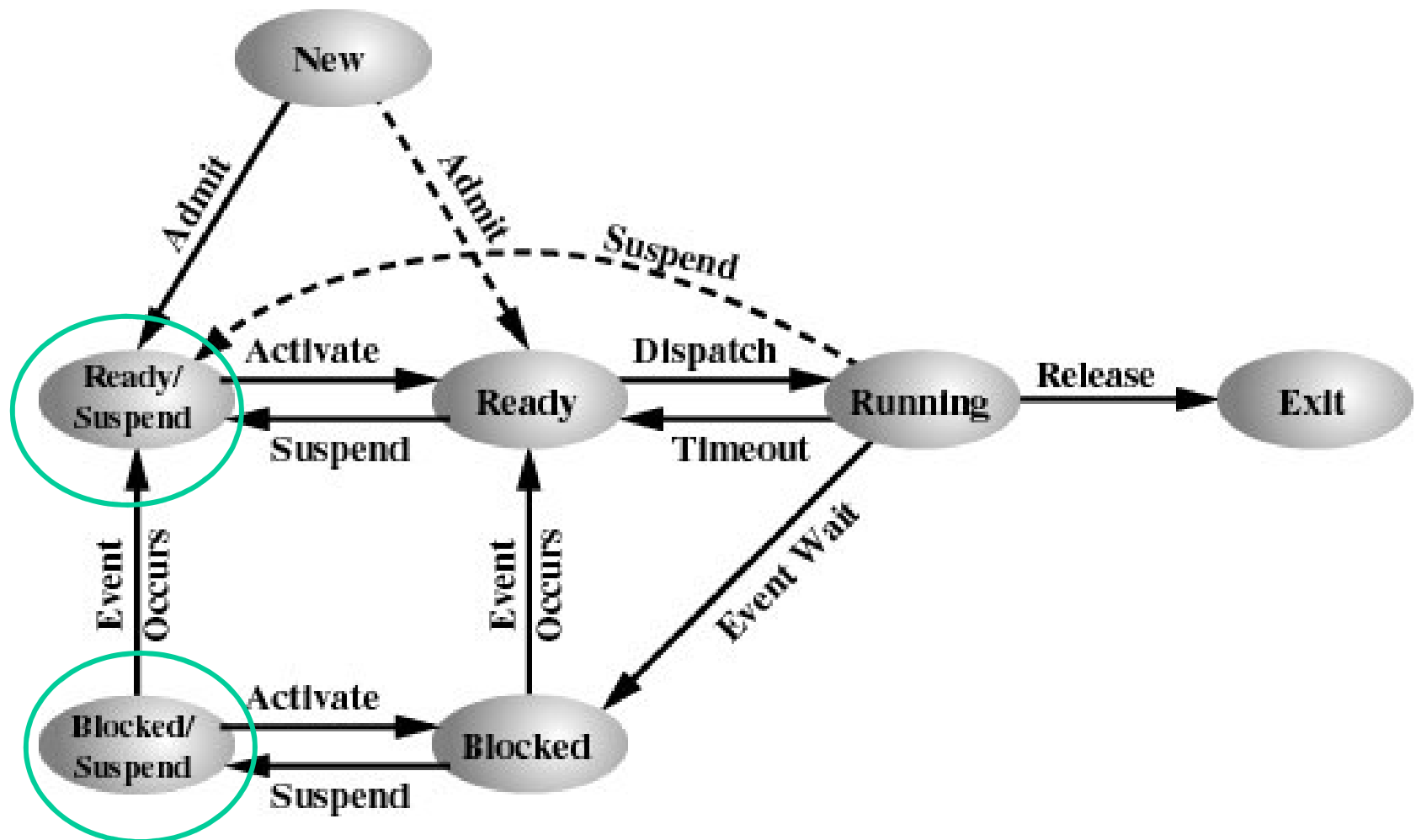
Or if ...

4. Suspended Processes

- ...**Answer/Idea:** Swap these processes to disk to free up more memory (to admit new processes)
- Blocked state becomes **suspend state** when swapped to disk



4a. Two Suspend States



(b) With Two Suspend States

4b.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. |

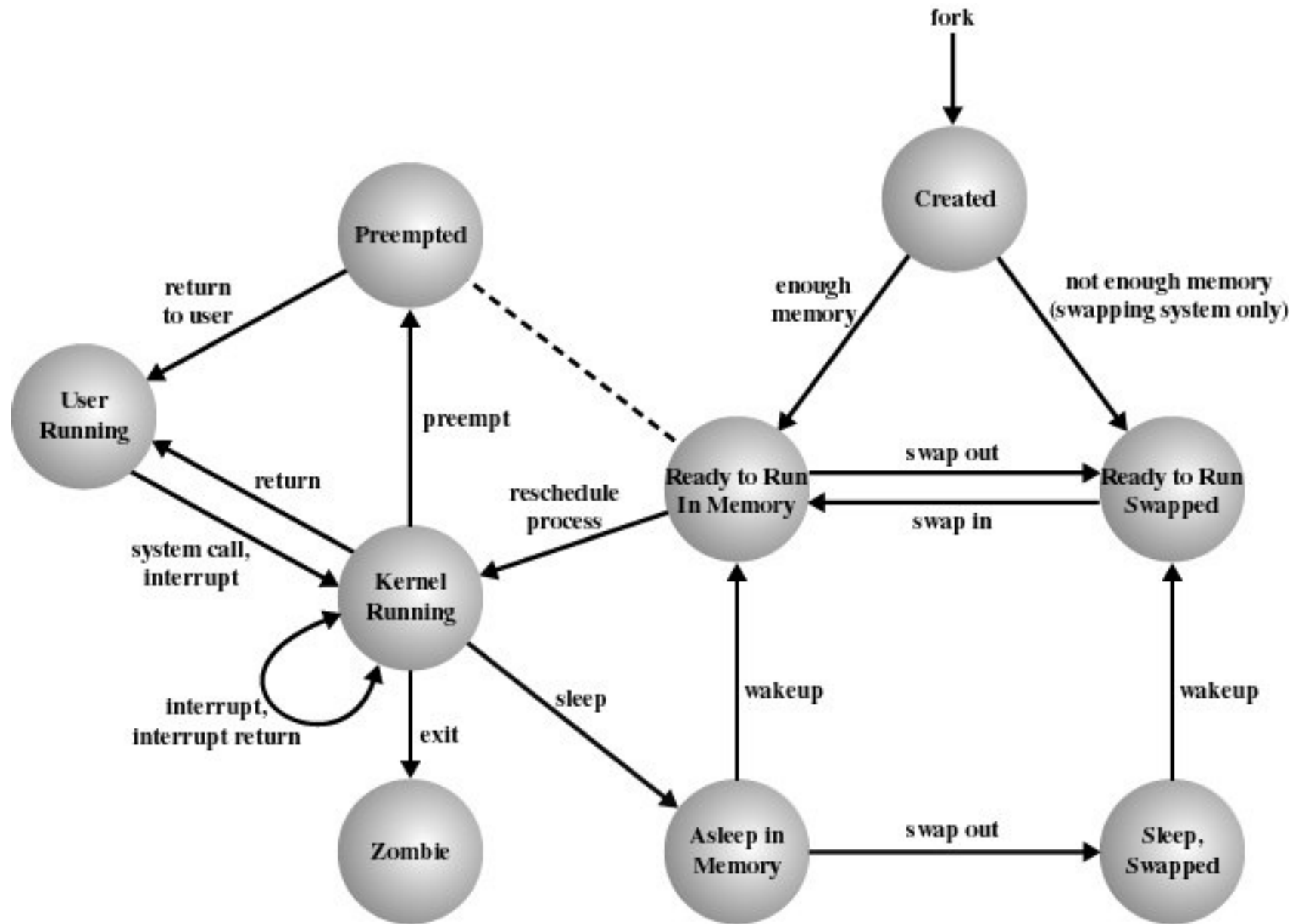
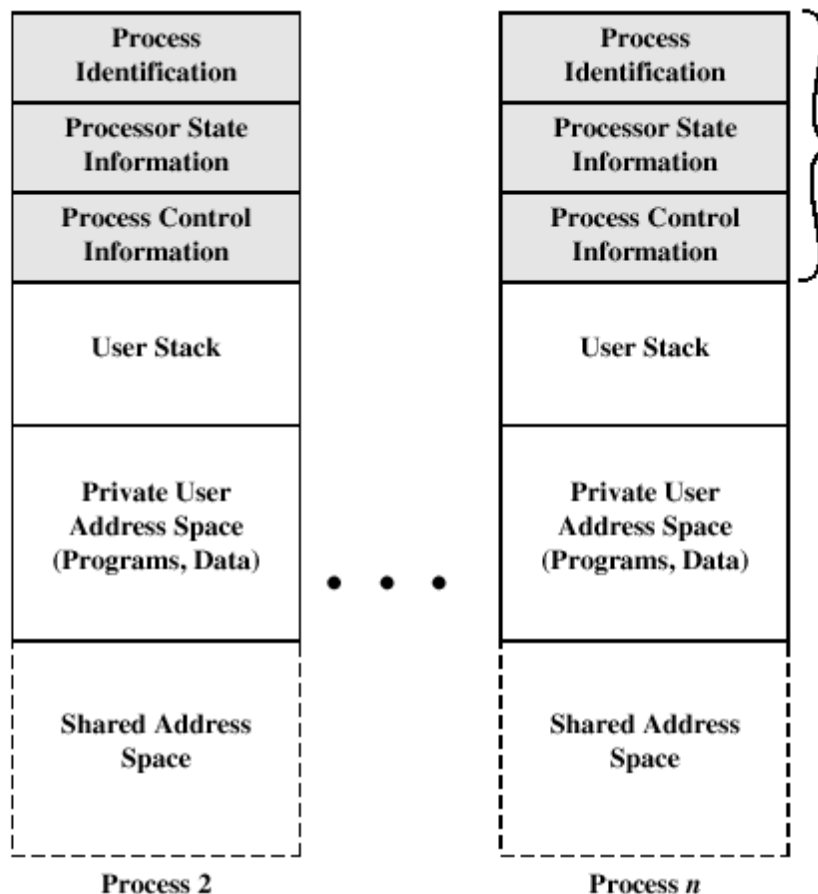


Figure 3.16 UNIX Process State Transition Diagram

OS Control Structures



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

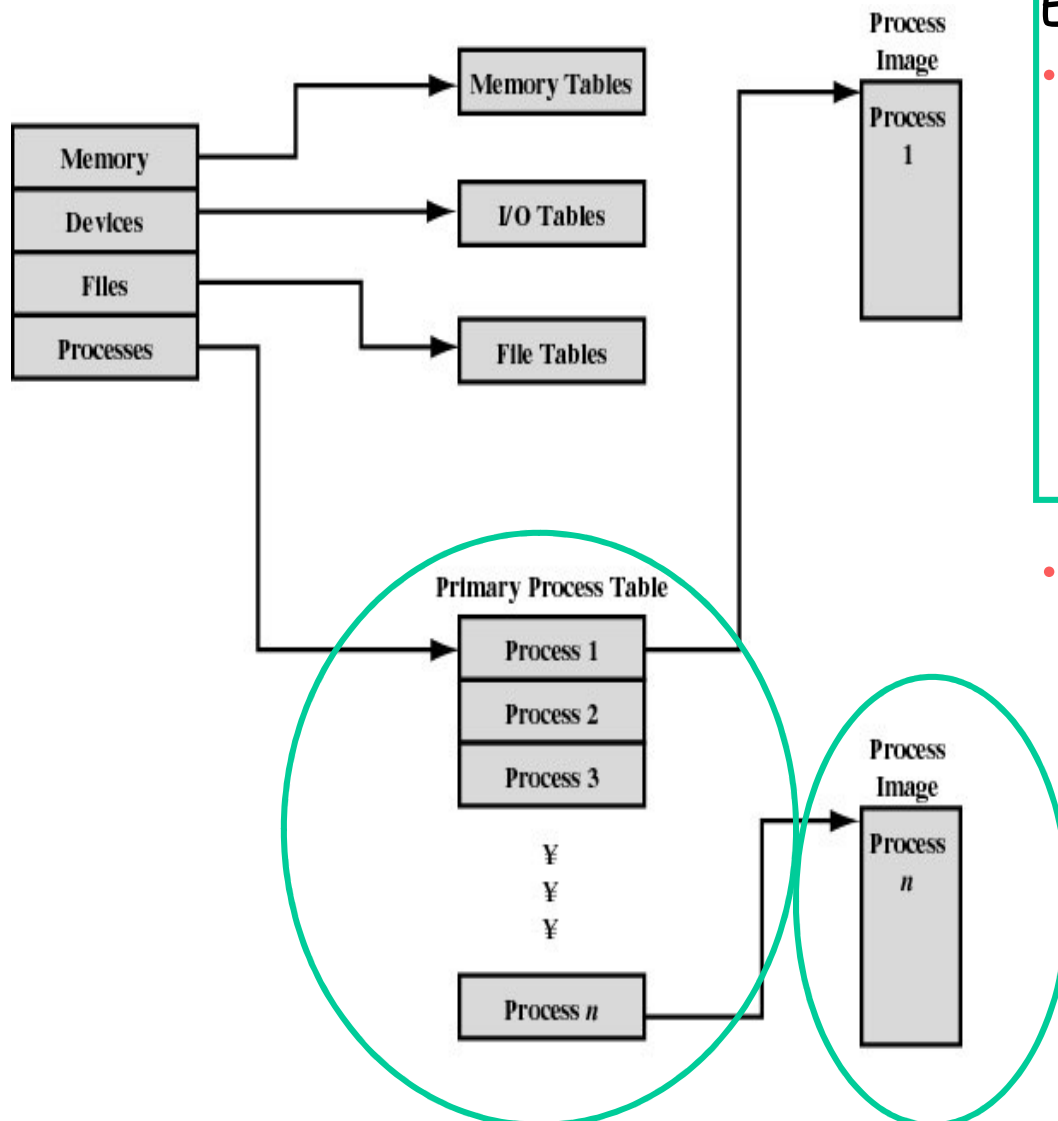
- Program
- Data
- Stack
- Process Control Block (context; for multiprogramming)

The OS must keep:

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

Figure 3.12 User Processes in Virtual Memory

Process Table (and other OS tables)



Each entry:

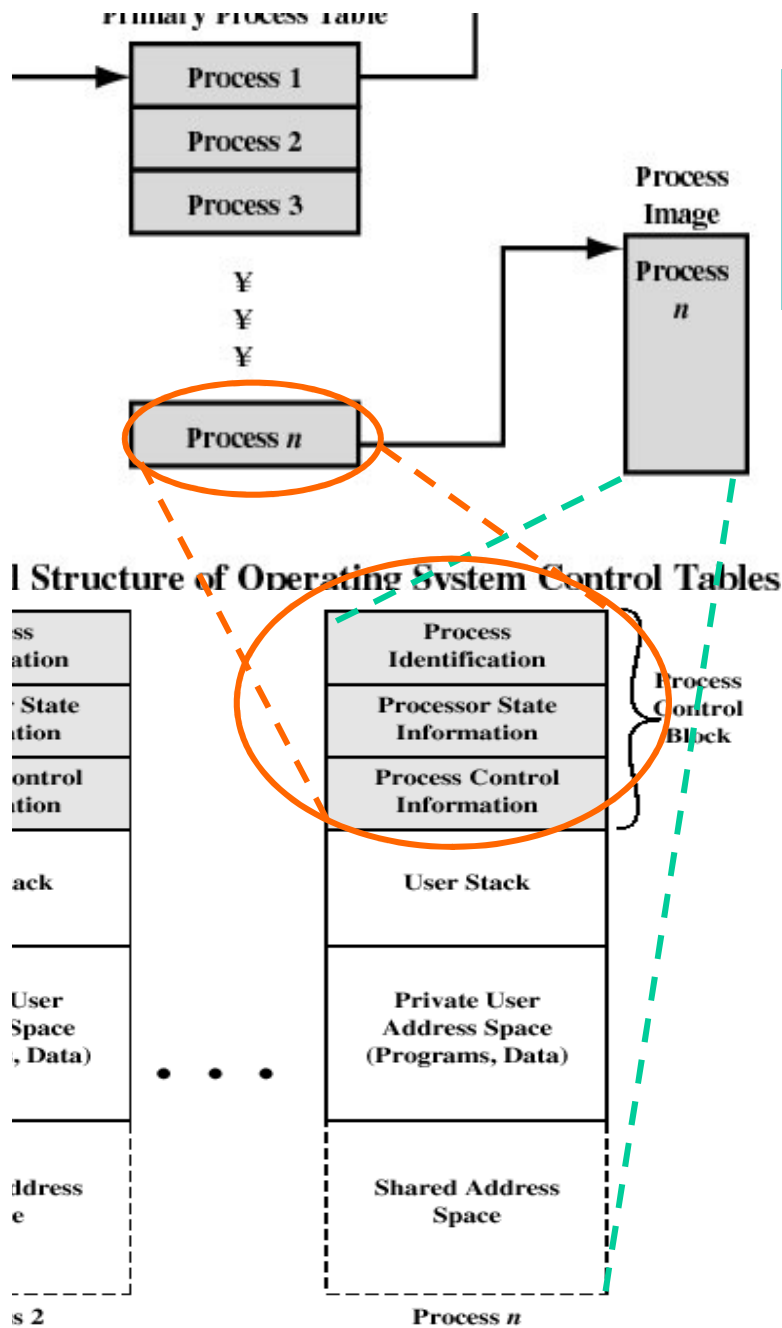
- Attributes necessary for its management
 - Process ID
 - Process state
 - Location in memory
 - Etc: process control block
- Other tables hold resource-specific info (zoom into later):
 - How main/secondary memory is allocated,
 - I/O device status, buffer in memory, ...
 - File status, location, attributes, ...

Figure 3.10 General Structure of Operating System Control Tables

Process Control Block

PCB contains:

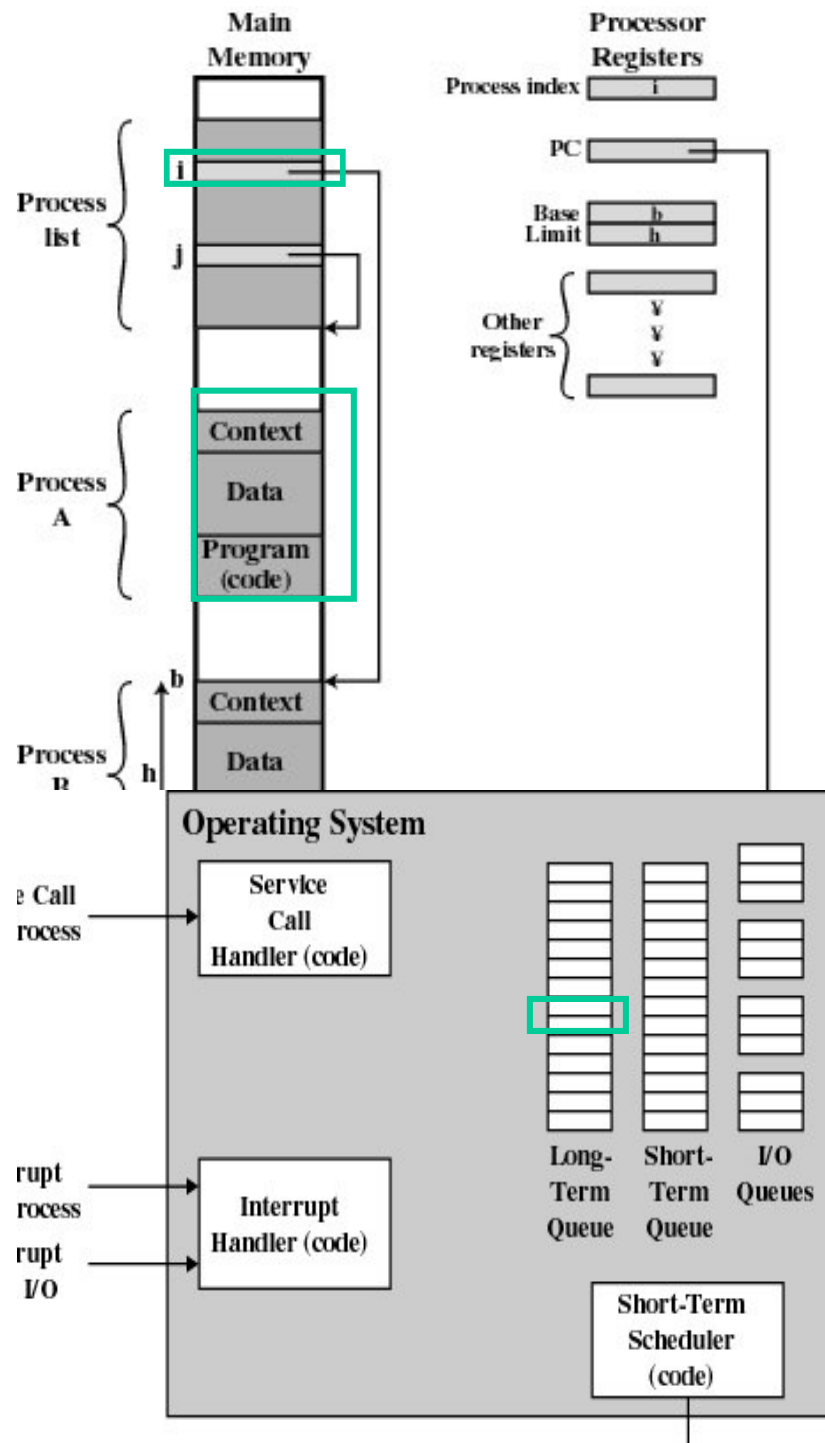
- Identifiers (process, parent process, user, ...)
- Processor State Information (register values: must be copied and restored in state transitions: running ↔ ready, ...)
- Other Process Control Information:
 - Scheduling and State Information (priority, event awaited, ...)
 - Process memory tables
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



Structure of Operating System Control Tables

Process Identification	Process Identification
Processor State Information	Processor State Information
Process Control Information	Process Control Information
User Stack	User Stack
Private User Address Space (Programs, Data)	Private User Address Space (Programs, Data)
Shared Address Space	Shared Address Space

Process Creation



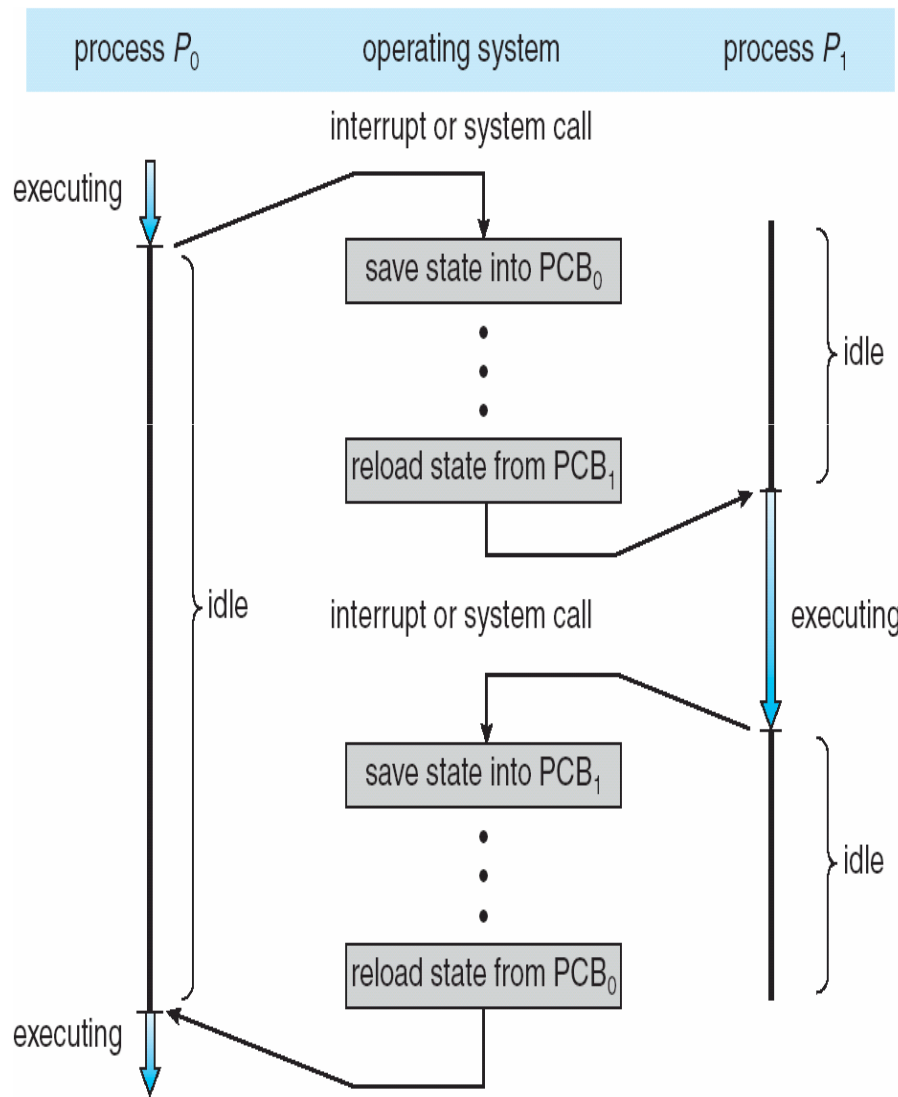
Examples:

- 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), ...)

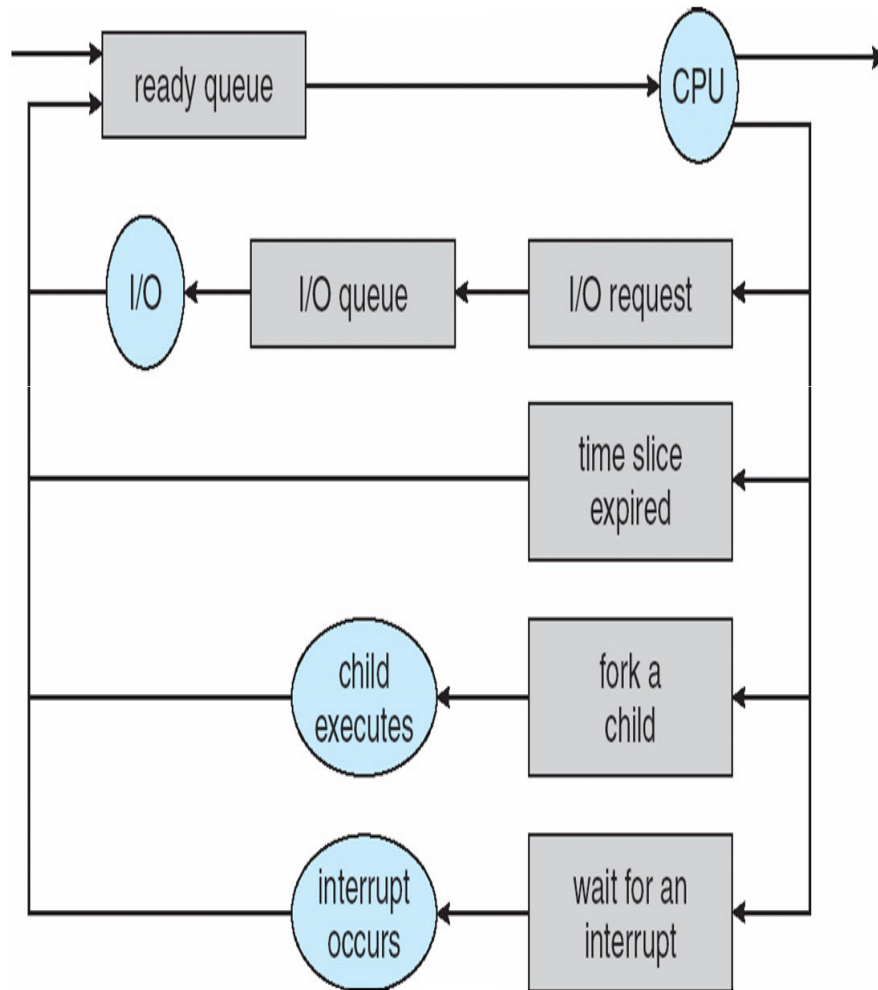
Change of Process State + new dispatch (context switching)



running \rightarrow 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

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

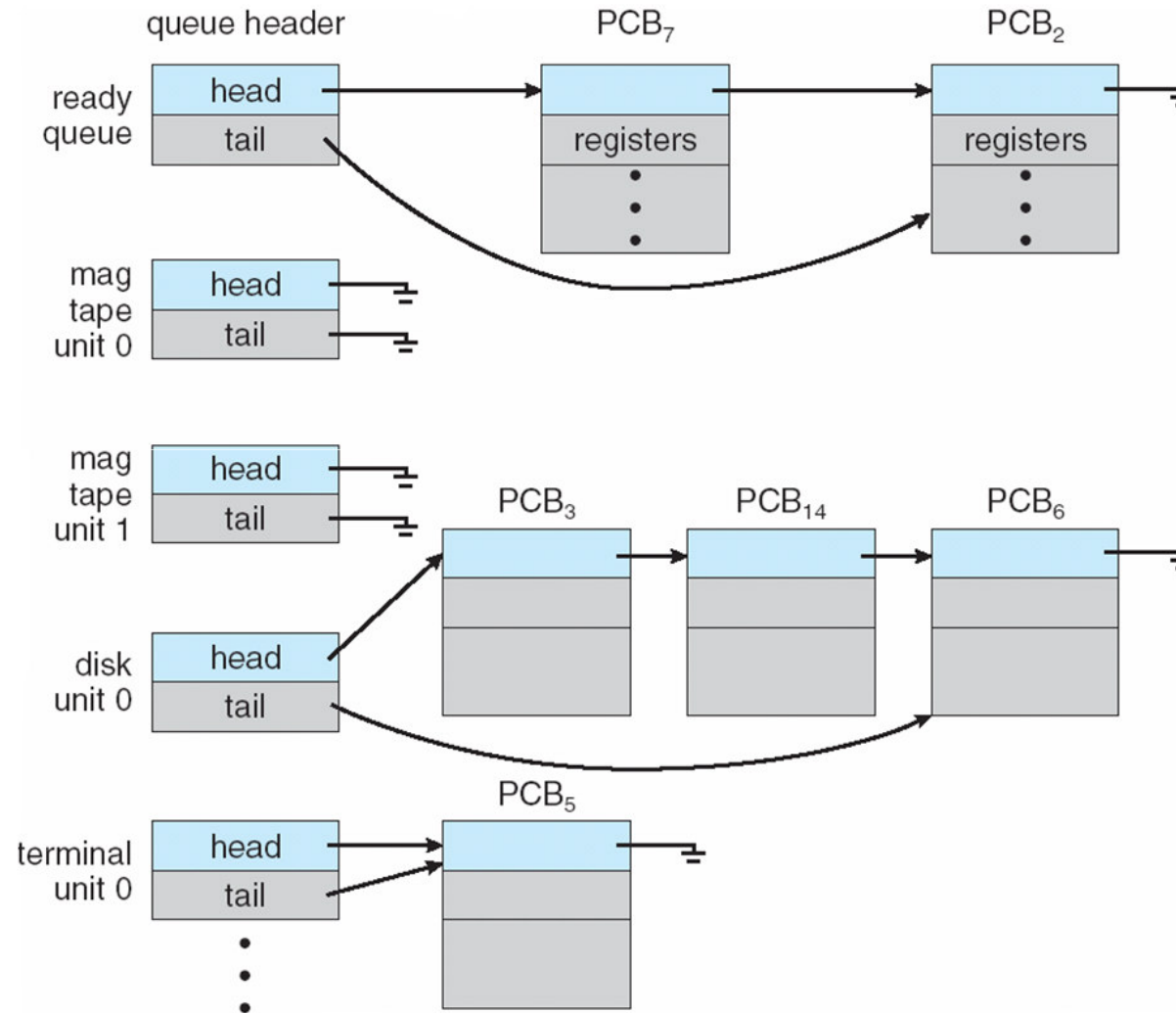
error occurred

may cause process to be moved to Exit state

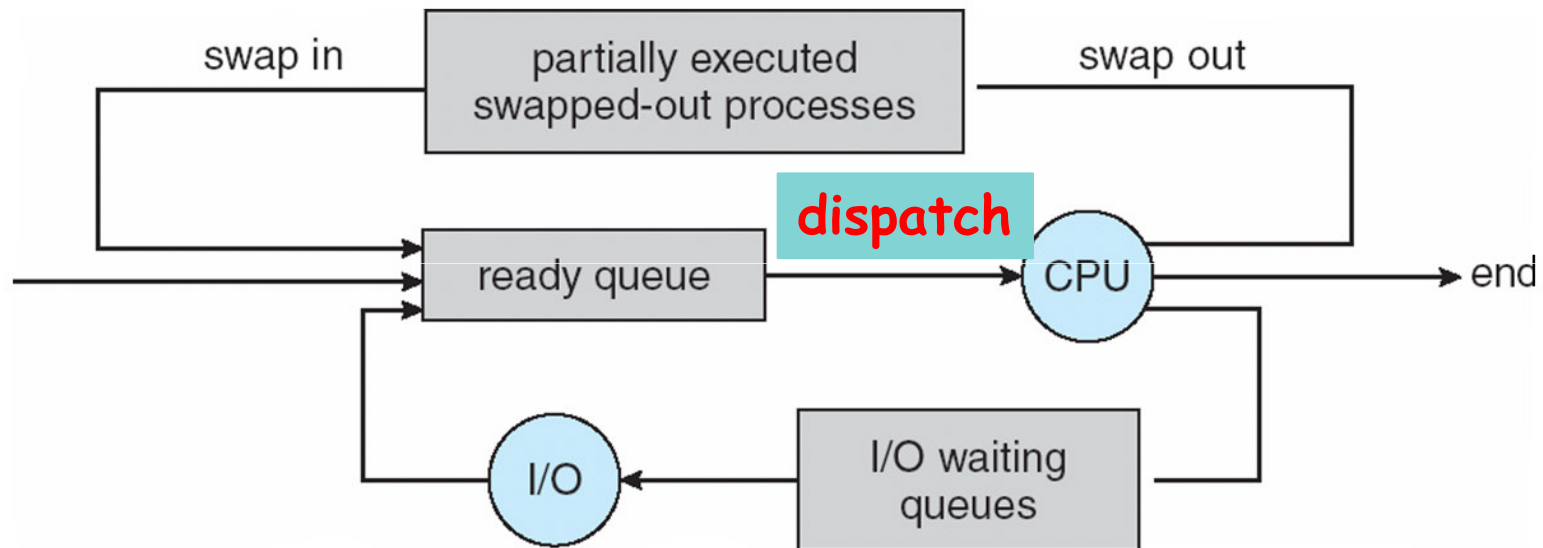
Supervisor call (or system call)

such as file open

Ready Queue And Various I/O Device Queues

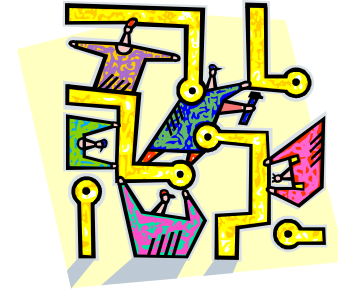


Addition of Medium Term Scheduling (swap-out/suspension)



Interprocess Communication

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

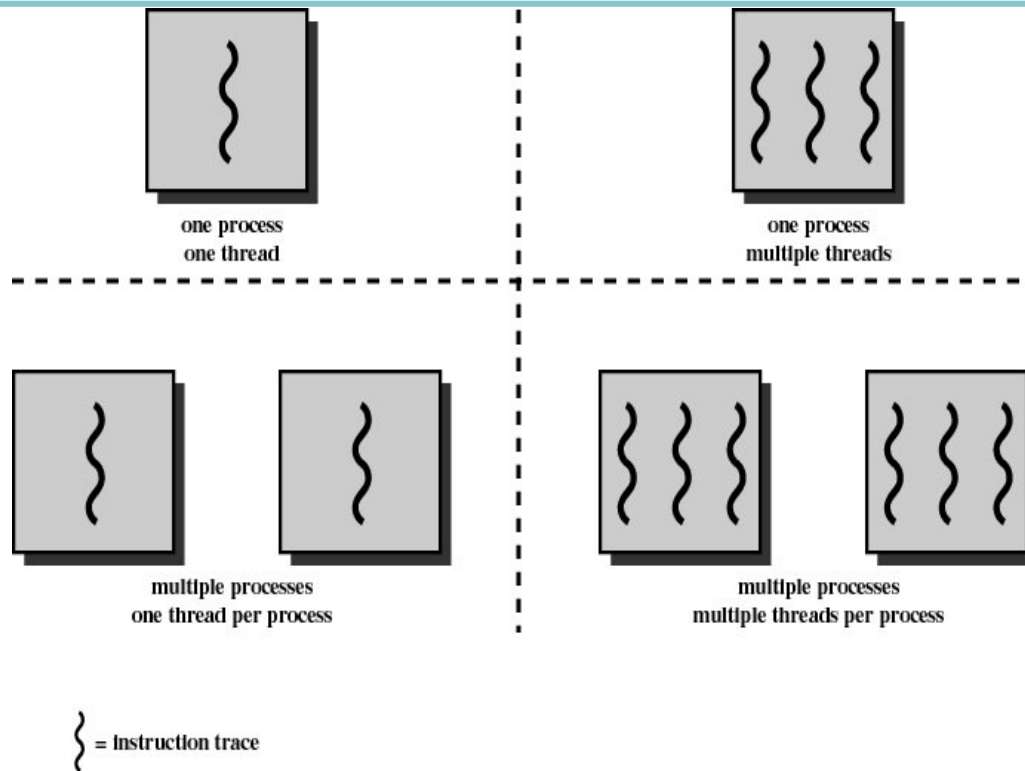


(more: to be discussed soon)

Threads

Processes and Threads

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



Unit of ...

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

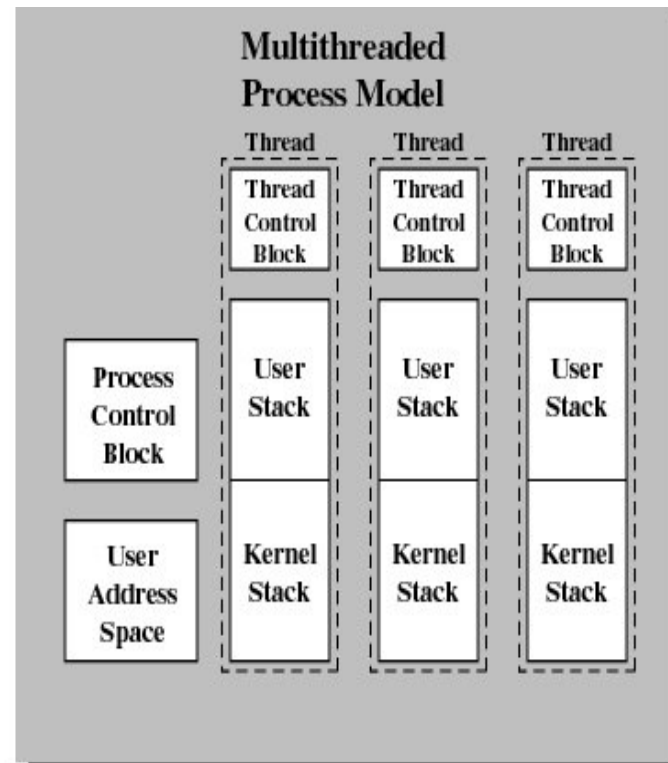
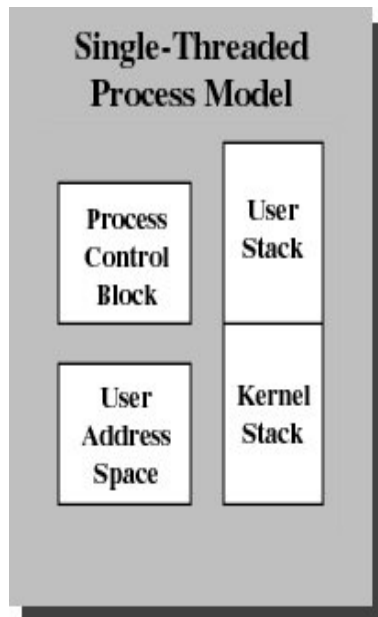
Figure 4.1 Threads and Processes [ANDE97]

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**

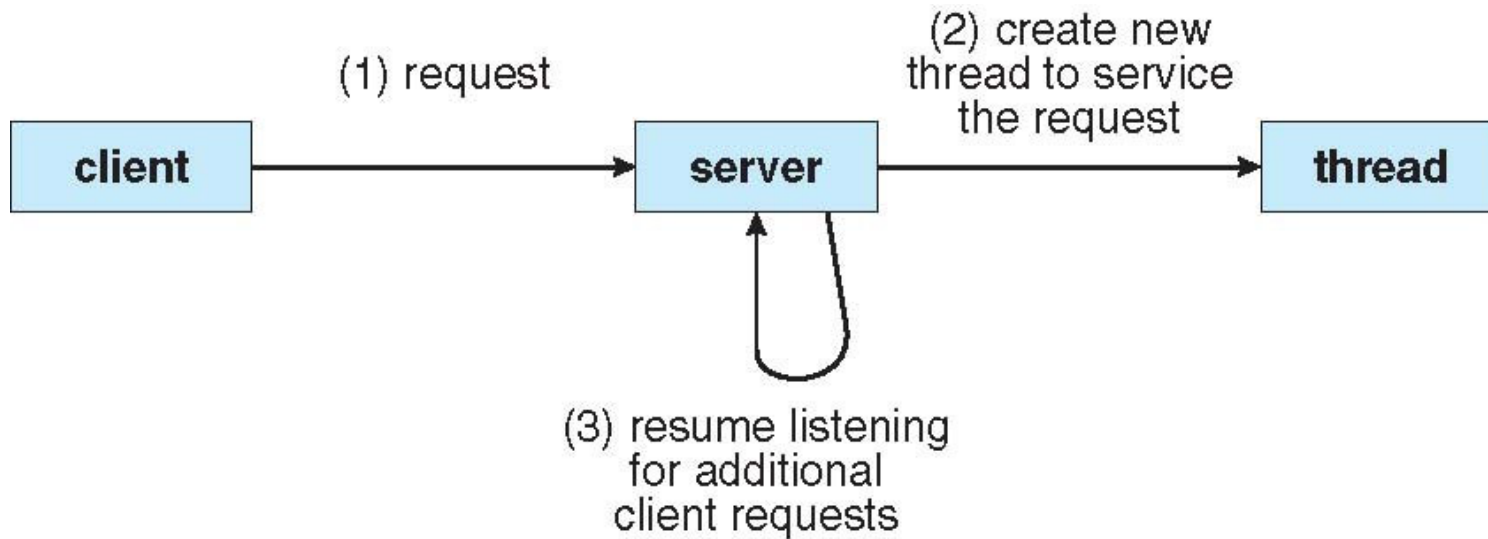
Uses of Threads

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

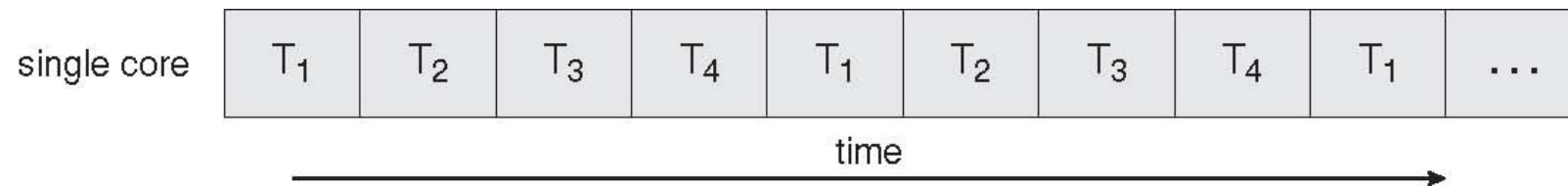
Multicore Programming

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

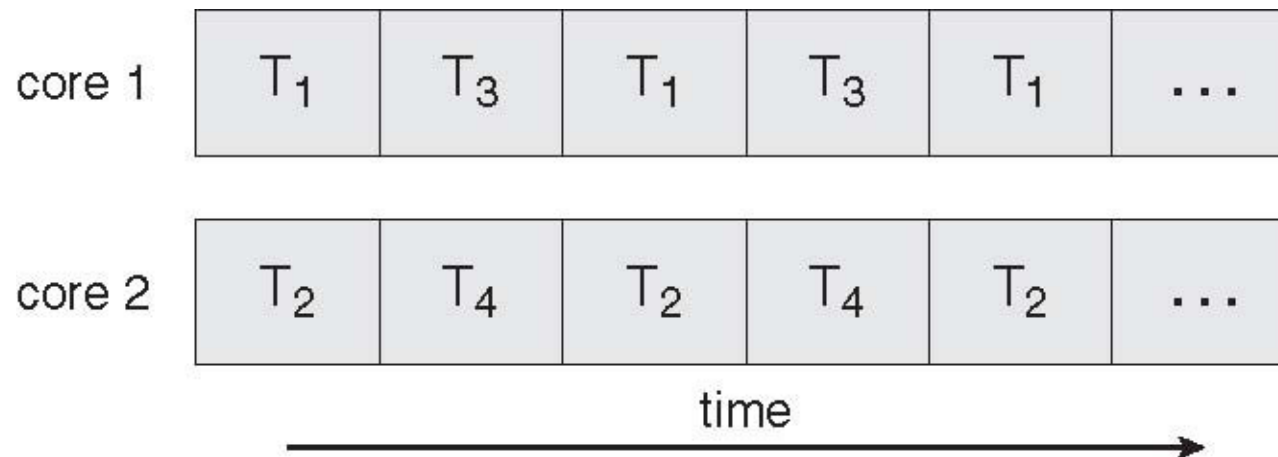
e.g. Multithreaded Server Architecture



"Concurrent" Execution on a Single-core System



Parallel Execution on a dual-core System



Question

can one have concurrency (independent threads of execution) within a process but without OS-support of threads? (i.e. no thread tables etc)

Hint: remember the step to multiprogramming on a single processor?

Think of Interrupts/Signals....

... signals (similar to interrupts) can notify a **process** that a particular event has occurred ...

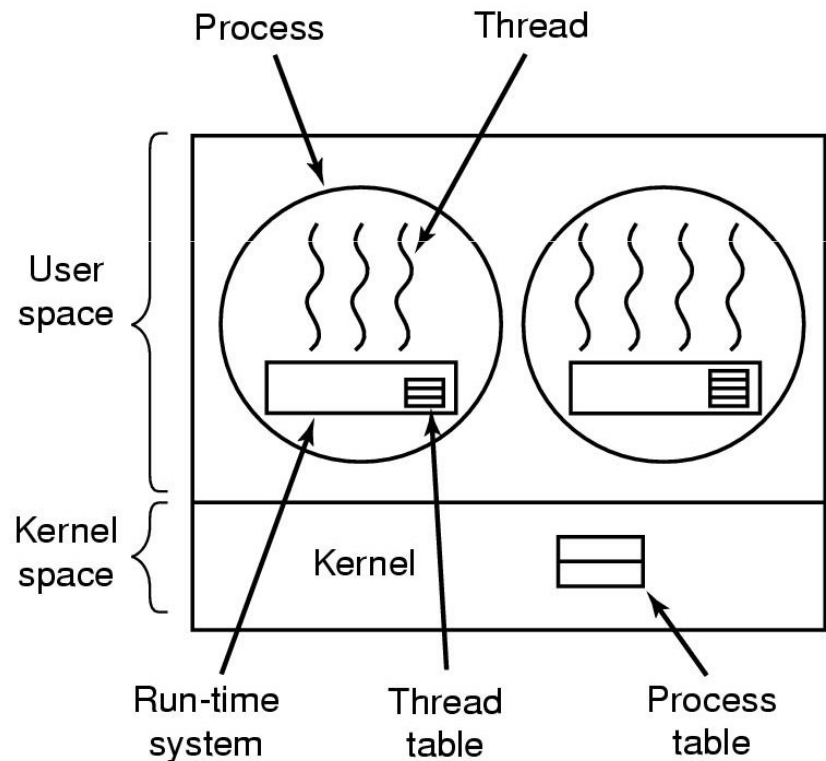
- A **signal handler** is used to process signals
 1. Signal is generated by particular event e.g. timer or non-blocking IO (cf select system call)
 2. Signal is delivered to a process
 3. Signal is handled; **handling may involve to switch between different threads** (user-level)

... and of course also voluntary relinquish of control can help in switching threads (+ e.g. setjump/longjump C instructions)



Implementing Threads in User Space

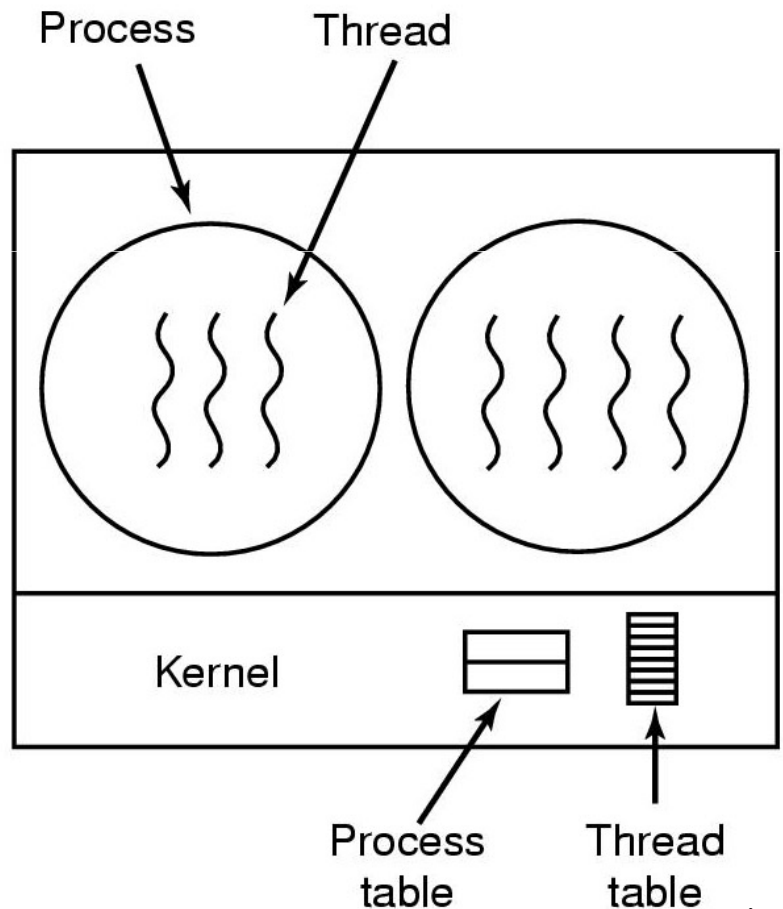
The kernel is not aware of the existence of threads;



- 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)

Implementing Threads in the Kernel

Kernel maintains context information for the process and the threads

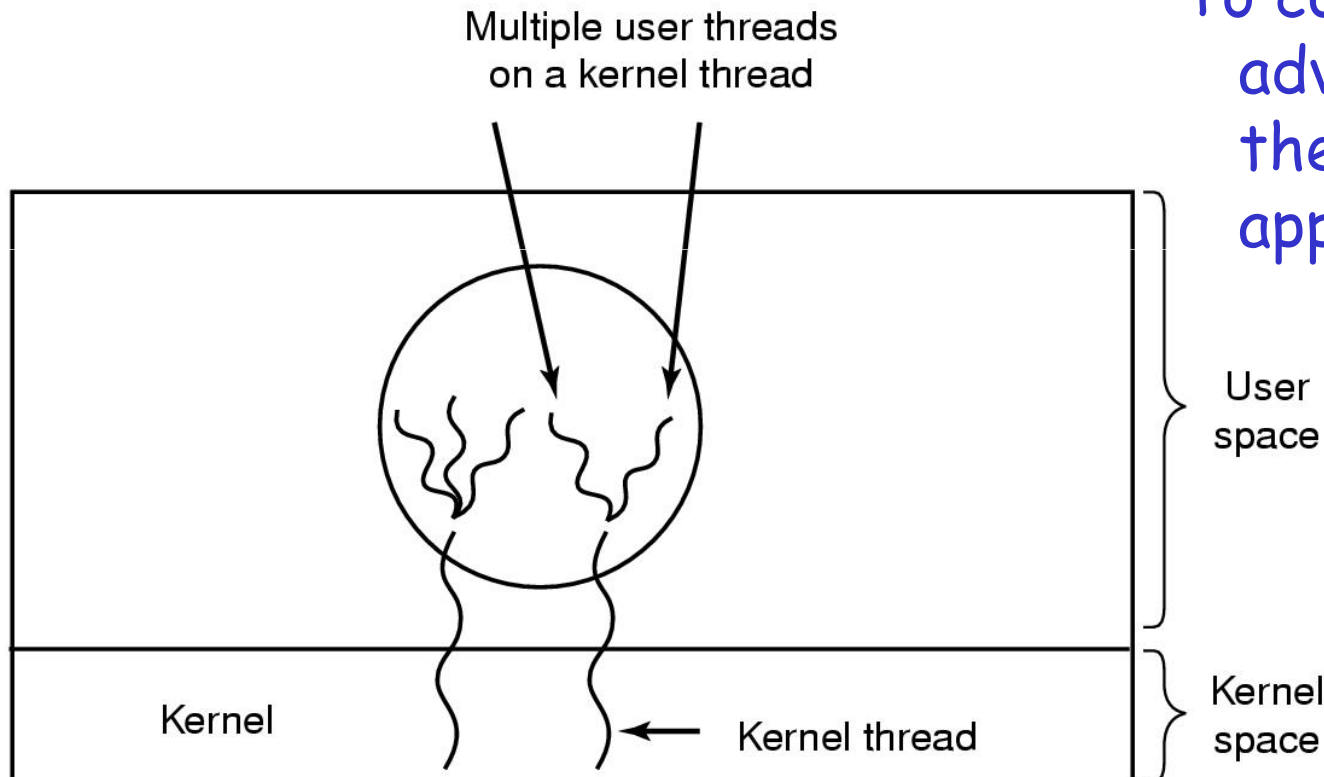


Process table Thread table
Fig: Tanenbaum, Modern OS, 2/e

- 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

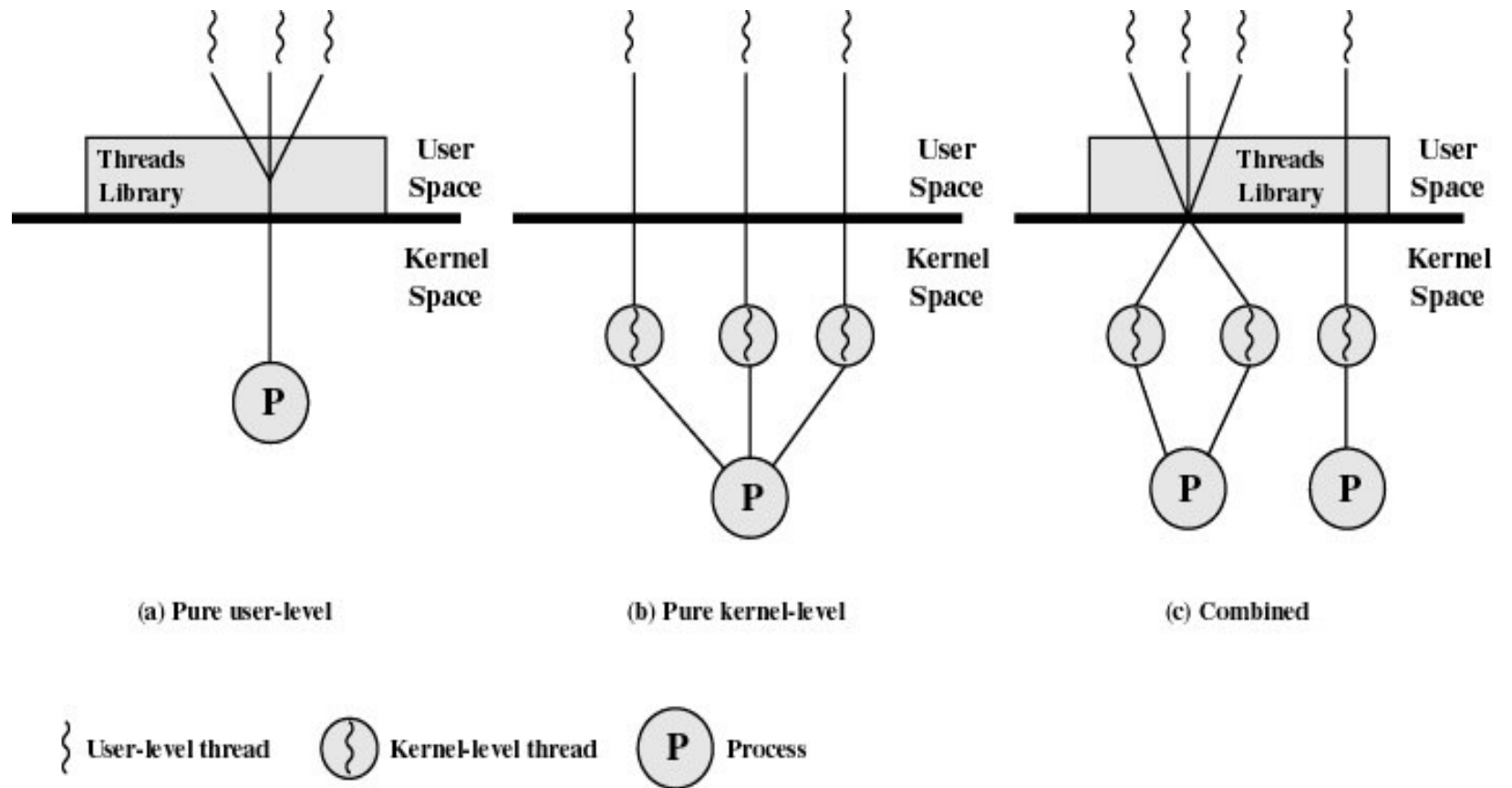


Figure 4.6 User-Level and Kernel-Level Threads

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

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

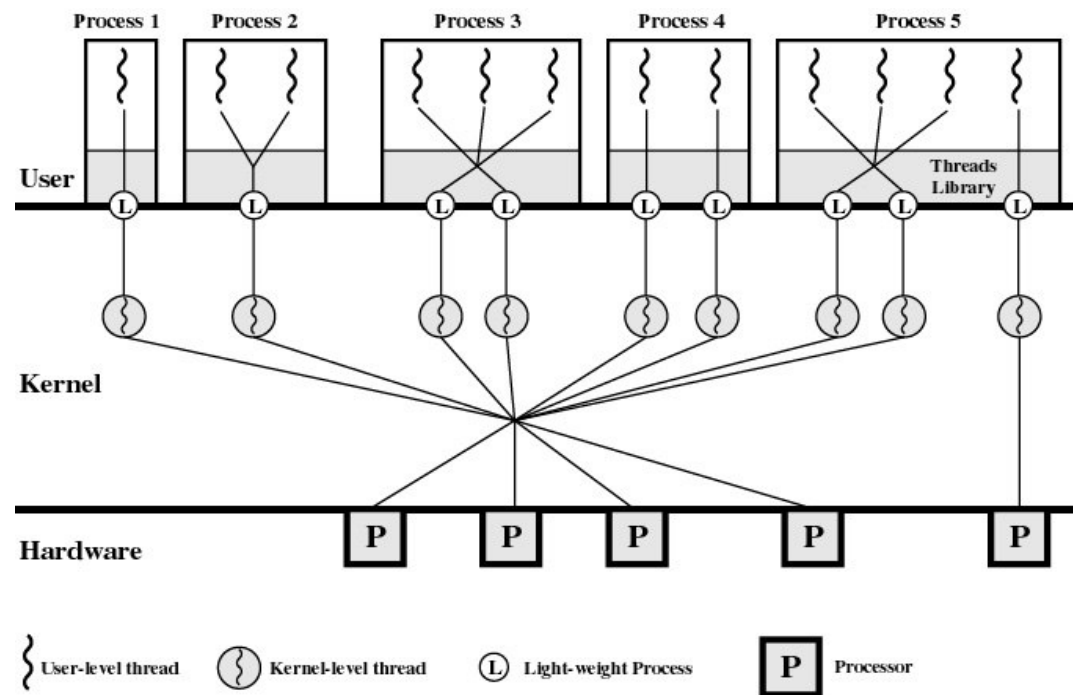


Figure 4.15 Solaris Multithreaded Architecture Example

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
<code>CLONE_FS</code>	File-system information is shared.
<code>CLONE_VM</code>	The same memory space is shared.
<code>CLONE_SIGHAND</code>	Signal handlers are shared.
<code>CLONE_FILES</code>	The set of open files is shared.

Thread Pools

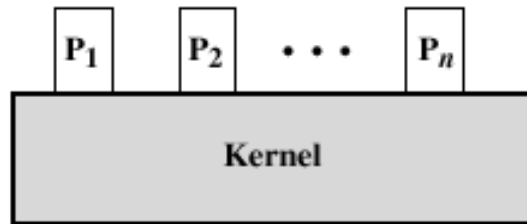
- Create a number of threads in a pool where they await work
- Advantages:
 - Usually slightly faster to service a request with an existing thread than create a new thread
 - Allows the number of threads in the application(s) to be bound to the size of the pool

Scheduler Activations

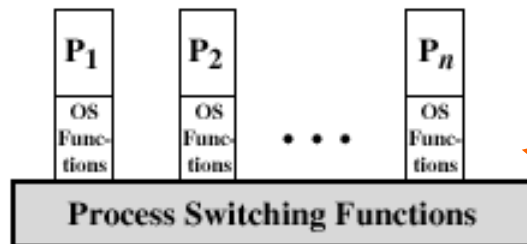
- communication to maintain the appropriate number of kernel threads allocated to the application
 - To gain performance of user space threads
- **upcalls** - a communication mechanism from the kernel to the thread library/run-time system
 - Kernel assigns/allocates virtual processors (kernel threads) 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)



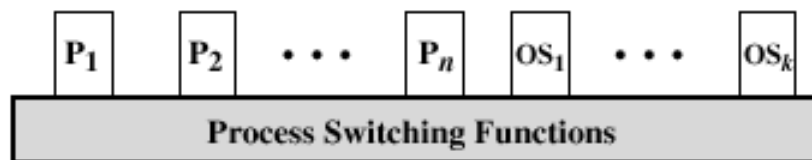
Execution of the Operating System



(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

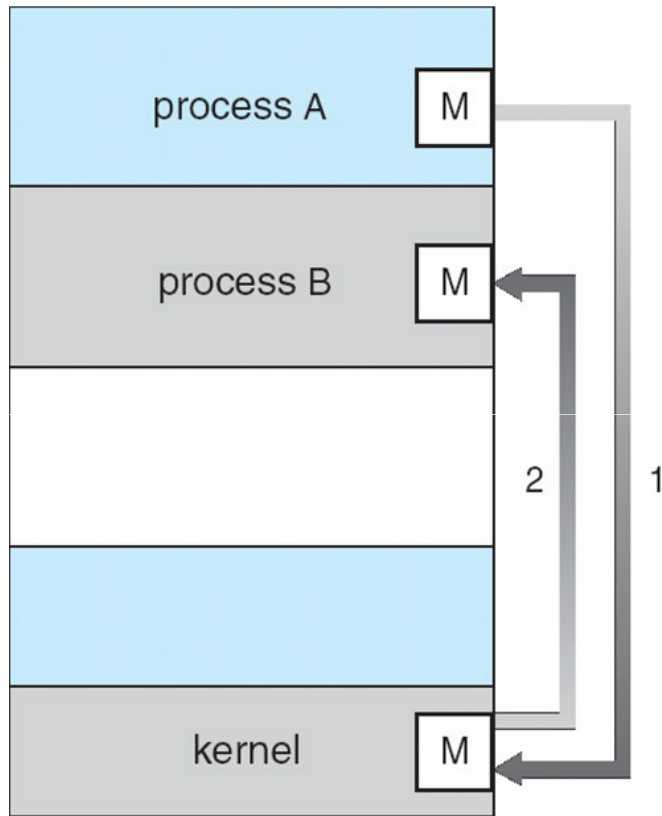
- major kernel functions are separate processes
- Useful in multi-processor or multi-computer environment

Execution Within User Processes

- operating system software within context of a user process
- process executes in privileged mode when executing operating system code

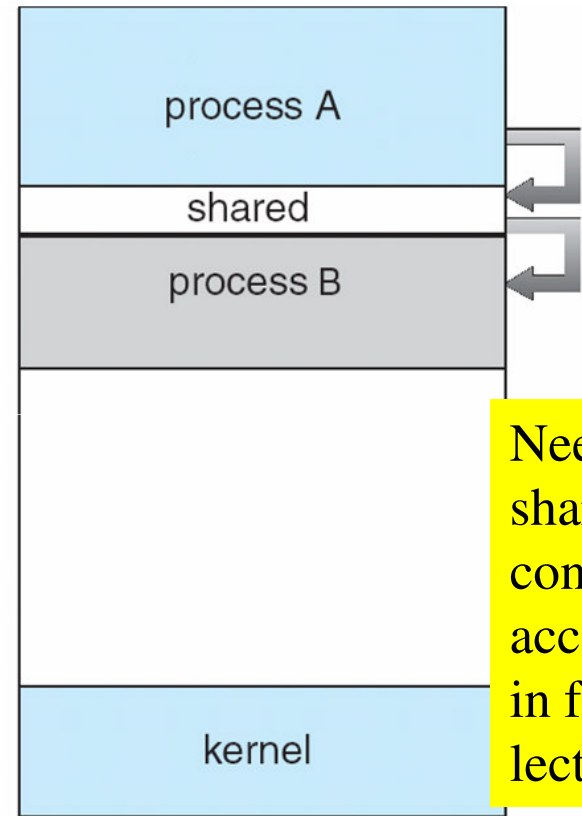
Interprocess Communication

Communication Models



(a)

Message passing



(b)

Shared memory

Need to protect shared data from concurrent accesses; more in forthcoming lectures



Interprocess Communication - Message Passing

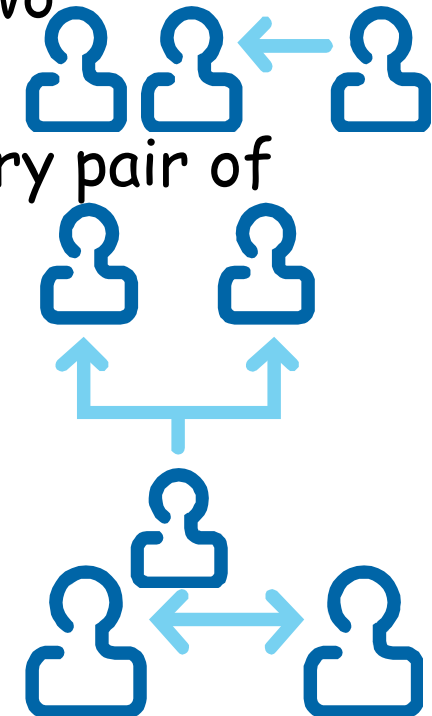
- Mechanism for processes to **communicate and to synchronize their actions**
- Message system - processes communicate with each other without resorting to shared variables
- IPC facility provides two operations:
 - **send(message)** - message size fixed or variable
 - **receive(message)**
- If P and Q wish to communicate, they need to:
 - establish a *communication link* between them
 - exchange messages via send/receive
- Implementation of communication link
 - physical (e.g., shared memory, hardware bus)
 - logical (e.g., logical properties)

send-receive
can help in
synchronization
e.g. Producer-
consumer relations –
more in coming
lecture(s)



Implementation Questions

- How are links established?
- Can a link be associated with more than two processes?
- How many links can there be between every pair of communicating processes?
- What is the capacity of a link?
- Is a link unidirectional or bi-directional?



Direct Communication

- Processes must name each other explicitly:
 - `send (P, message)` - send a message to process P
 - `receive(Q, message)` - receive a message from process Q



Indirect Communication

- Messages are directed and received from mailboxes (sometimes referred to as ports)
 - Each mailbox has a unique id
 - Processes can communicate only if they share a mailbox
- Operations
 - create a new mailbox
 - send and receive messages through mailbox
 - destroy a mailbox
- Primitives are defined as:
send(*A*, *message*) - send a message to mailbox *A*
receive(*A*, *message*) - receive a message from mailbox *A*



Examples of IPC Systems - POSIX

- **POSIX Shared Memory**

- Process first creates shared memory segment

```
segment id = shmget(IPC PRIVATE, size, S_IRUSR | S_IWUSR);
```

- Process wanting access to that shared memory must attach to it

```
shared memory = (char *) shmat(id, NULL, 0);
```

- Now the process could write to the shared memory

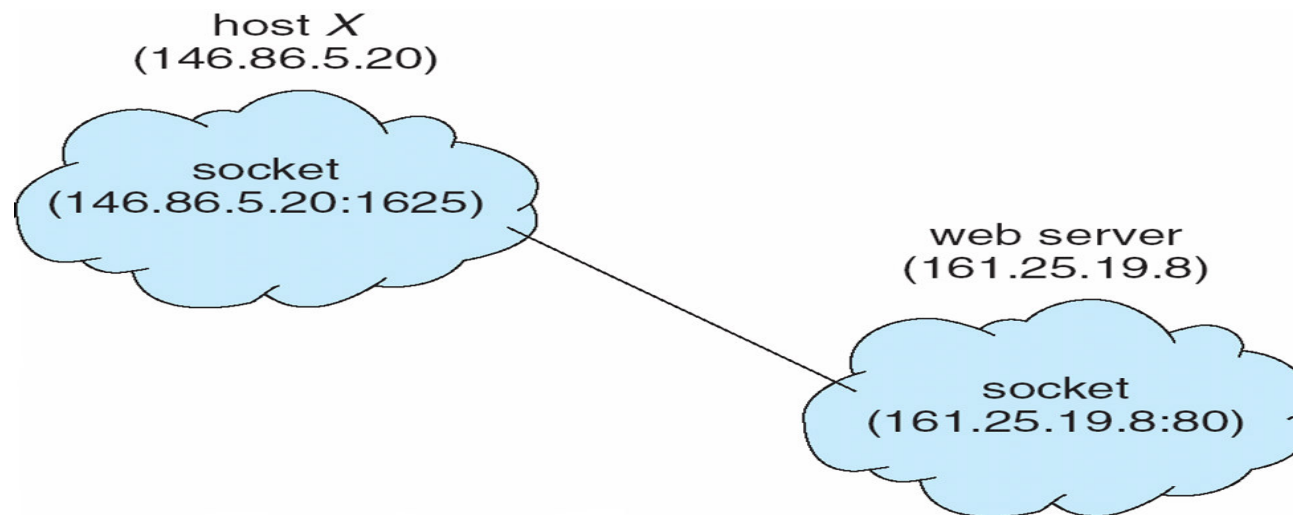
```
sprintf(shared memory, "Writing to shared memory");
```

- When done a process can detach the shared memory from its address space

```
shmdt(shared memory);
```

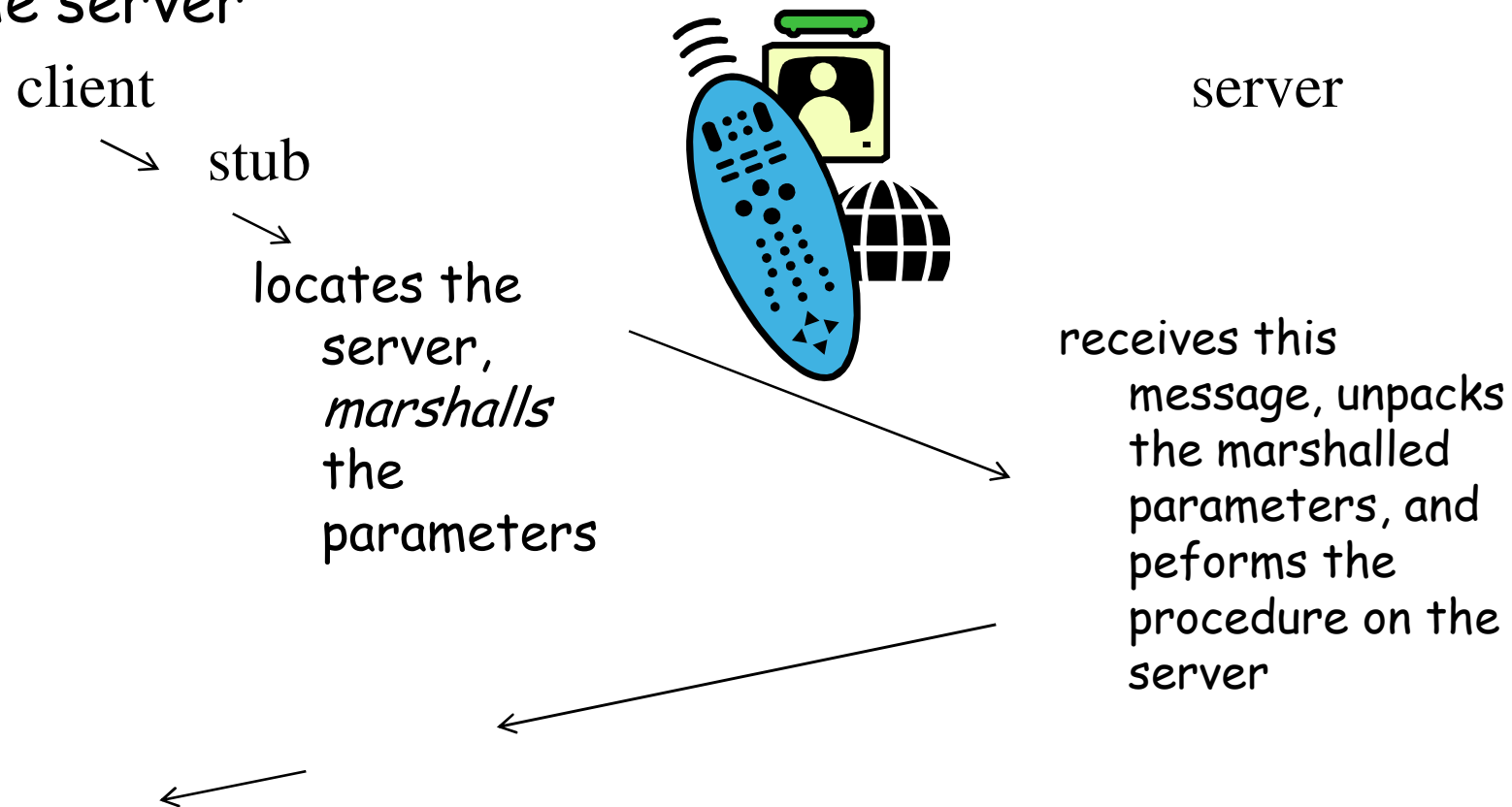

Examples IPC: Sockets

- Using sockets: defined as *endpoints for communication*
- Concatenation of IP address and port
 - The socket **161.25.19.8:1625** refers to port **1625** on host **161.25.19.8**
- Communication consists between a pair of sockets

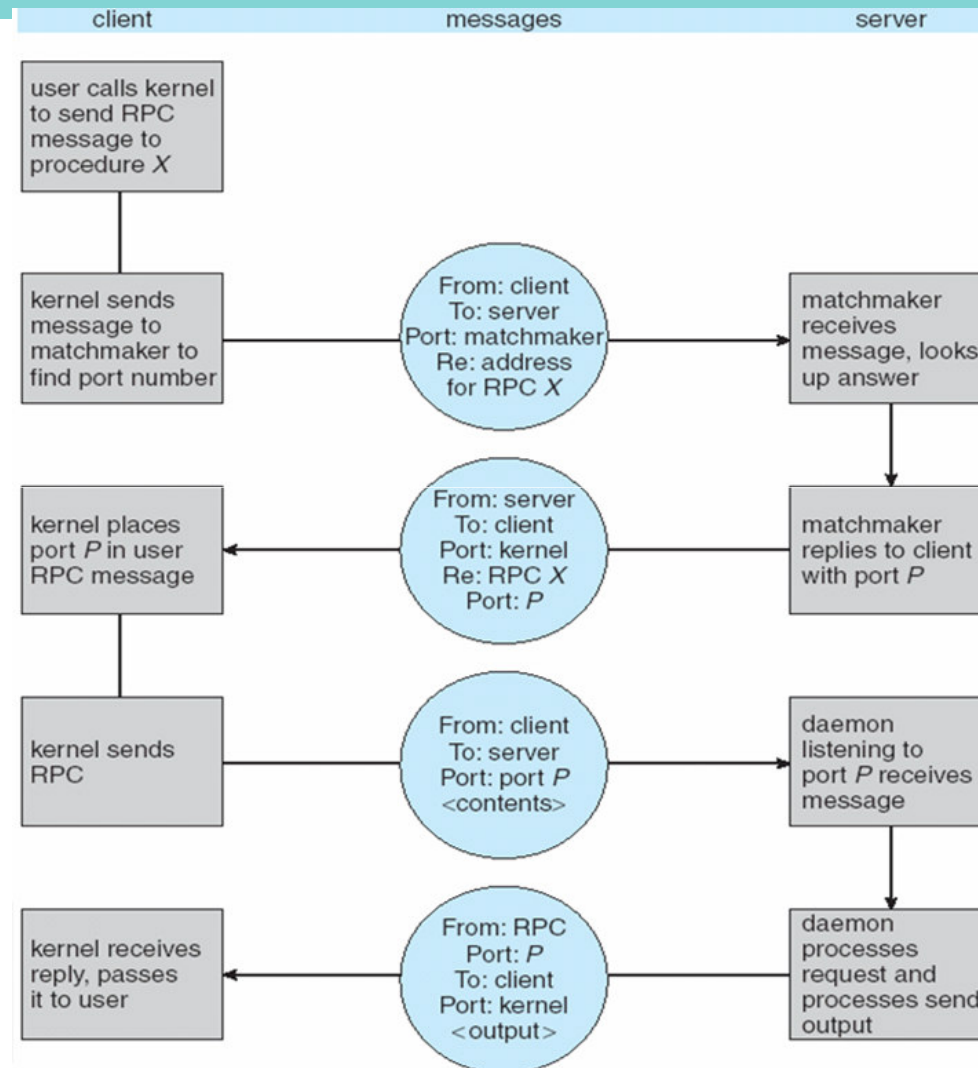


Examples IPC: Remote Procedure Calls (eg Java Remote Method Invocation)

- Remote procedure call (RPC) abstracts procedure calls between processes on networked systems (or VMs)
- **Stubs** - client-side proxy for the actual procedure on the server



Execution of RPC



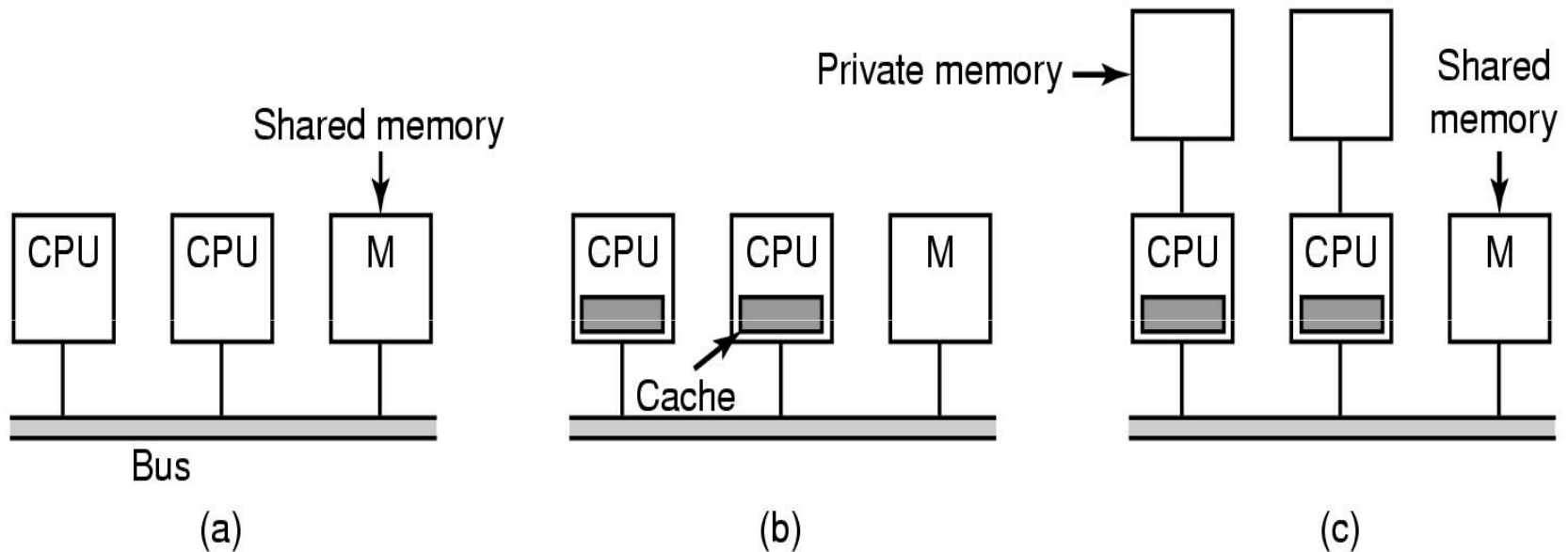
Multiprocessor Systems: Some closer look

Multiprocessors

Rough definition:

A computer system in which two or more CPUs sharing full access to a common RAM

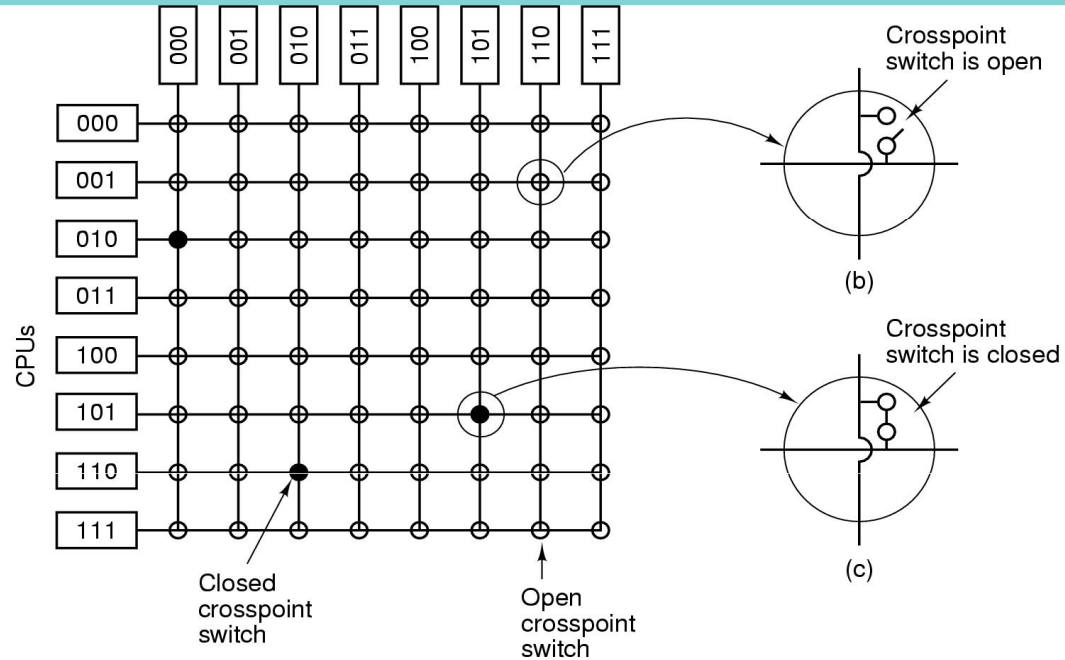
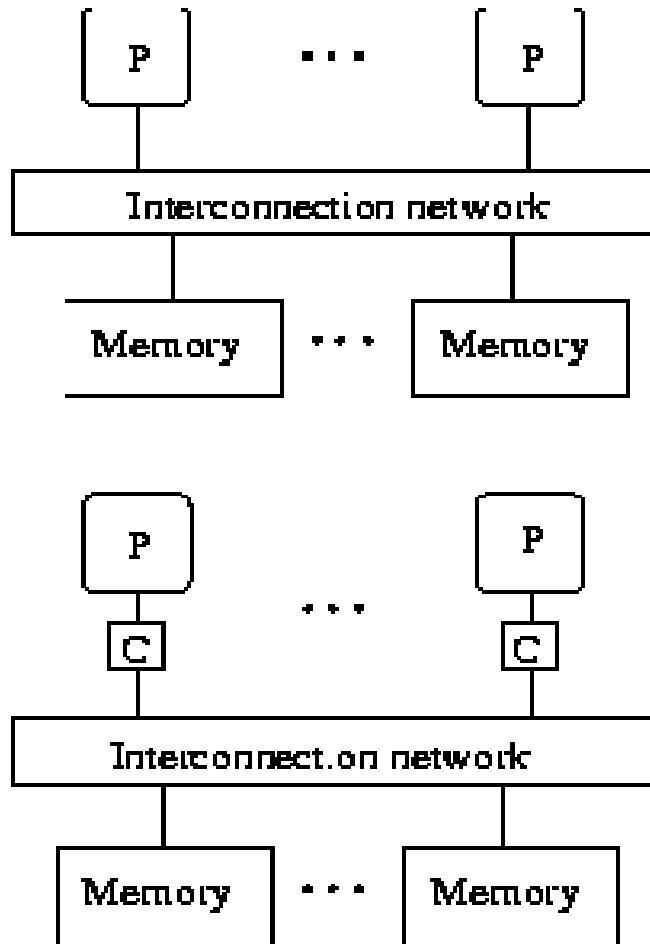
Multiprocessor/Multicore Hardware (ex.1)



Bus-based multiprocessors

Multiprocessor/Multicore Hardware (ex.2)

UMA (uniform memory access)



Not/hardly scalable

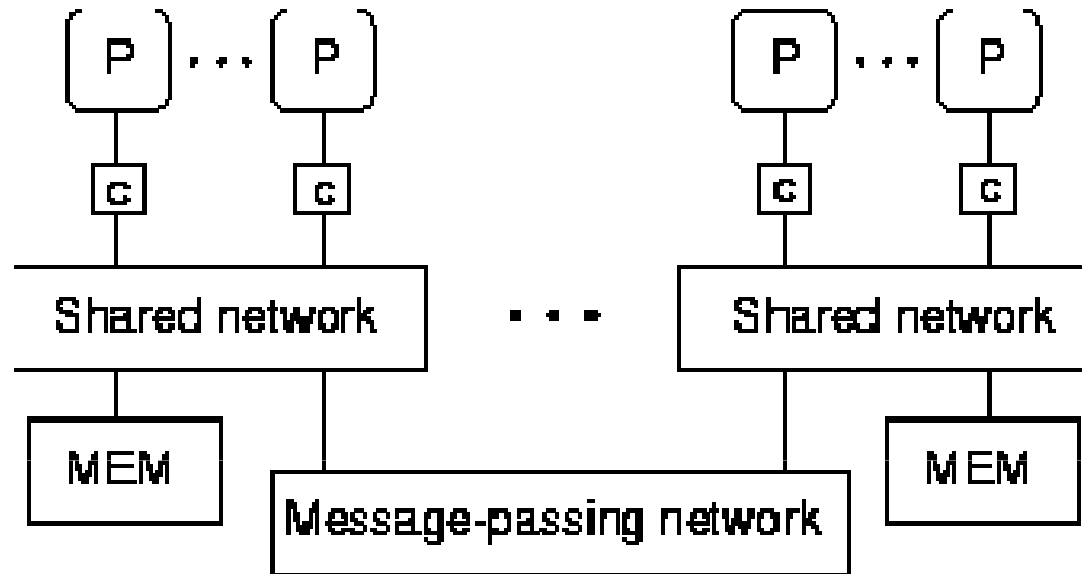
- Bus-based architectures -> saturation
- Crossbars too expensive (wiring constraints)

Possible solutions

- Reduce network traffic by caching
- Clustering -> non-uniform memory latency behaviour (NUMA).

Multiprocessor/Multicore Hardware (ex.3)

NUMA (non-uniform memory access)

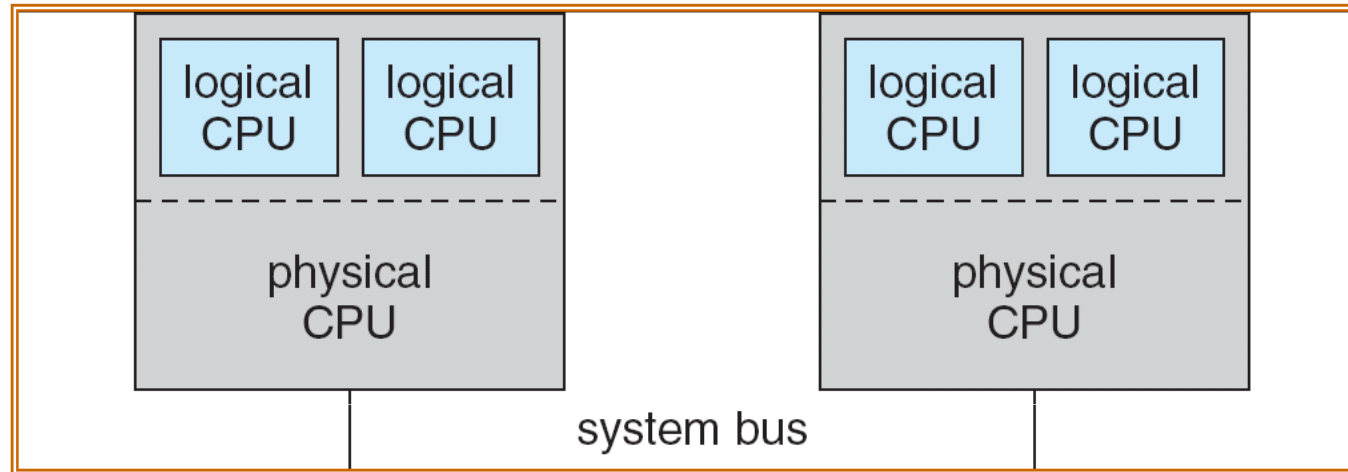


will discuss MMU
and cache-coherence
in the coming lecture(s)



- Single address space visible to all CPUs
- Access to remote memory slower than to local
- *Cache-controller/MMU* determines whether a reference is local or remote
- When caching is involved, it's called *CC-NUMA* (cache coherent NUMA)

Hyperthreading

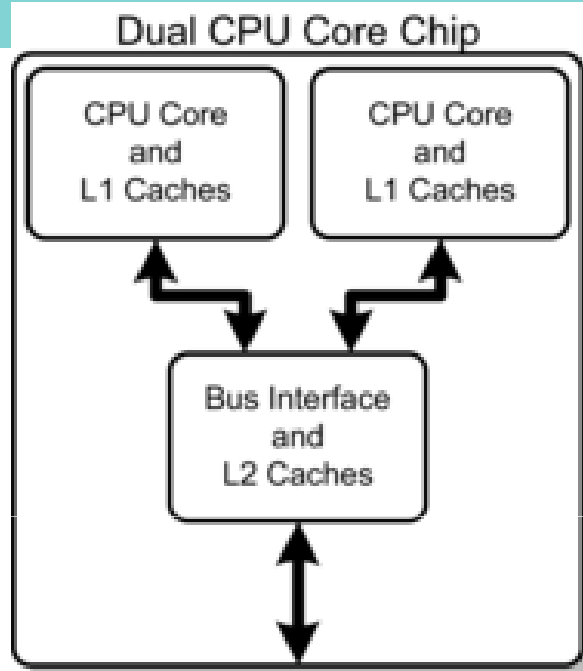


HW gives **image** of multiple processors per processor
OS can be oblivious; but will benefit from knowing that
it runs on such a HW (to be discussed in scheduling
context)



- logical CPU = state only (not execution unit)
- when processing stalls (eg cache miss, other data-dependency),
hyperthreaded processor can execute the other thread

On multicores



Intel Core 2 dual core processor, with CPU-local Level 1 caches+ shared, on-die Level 2 cache.

Contemporary graphics cards can have 128 (SIMD-like) cores

Reason for multicores: physical limitations can cause significant heat dissipation by high clock rate; instead, parallelize within the same chip!

In addition to operating system (OS) support, adjustments to existing software are required to maximize utilization of the computing resources provided by multi-core processors.

Virtual machine approach again in focus.

On multicores (cont)

- Also possible (figure from www.microsoft.com/licensing/highlights/multicore.mspx)

