

# Processes and Threads

Concepts & models, scheduling,  
and synchronization

Chapters 3,4,5,6

# Outline

- Concept of “**Process**” and “**thread**”
- Threads & Multithreaded systems
- Thread models
- Process (or thread) scheduling
- Cooperating processes
- Process (or thread) synchronization

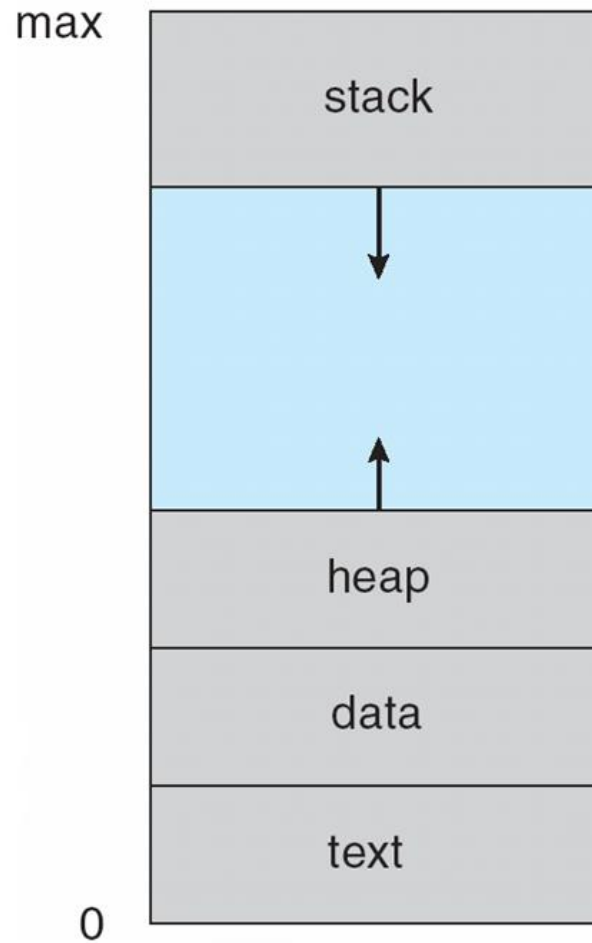
# Process Concept

- An operating system executes a variety of programs:
  - Batch system – jobs
  - Time-shared systems – user programs or tasks
- Short definition: A process is a “program in execution.”
  - A program is the code sitting in a file in this terminology.

# Process Concept

- Multiple parts
  - The program code, also called **text section**
  - Current activity including **program counter register** and **processor registers**
  - **Stack** containing temporary data
    - Function parameters, return addresses, local variables
  - **Data section** containing global variables
  - **Heap** containing memory dynamically allocated during run time (through `new` and `malloc()`).

# Process in Memory



Address space of a process (typically virtual)

# Process Concept

- An operating system involves *asynchronous*, and sometimes *parallel/concurrent*, activities.
- The notion of a software process is used in operating systems to express the management and control of such activities.
- A process is **the key concept** underlying modern operating system design.

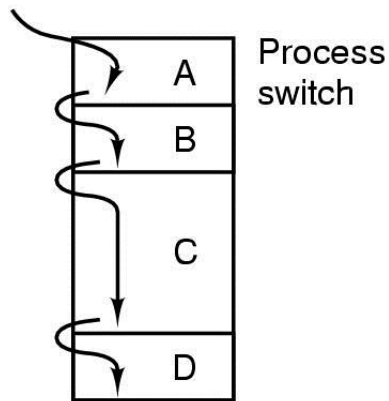
# Process Concept

- The concept of process simplifies the user's view of the computation model:
  - process execution uses a sequential model of computation, making the programming simpler.
  - Long definition: A process is an abstraction for a sequence of operations that implement a computation.
  - Multiple processes are executed concurrently with CPU (or CPU's) multiplexing among them.

# Processes

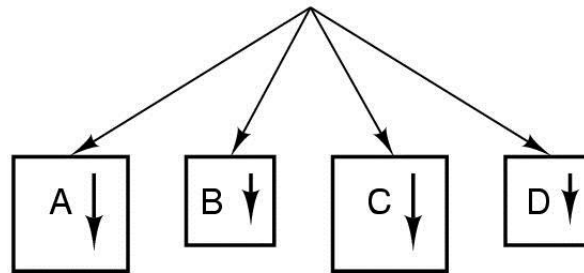
## The Process Model with one CPU

One program counter

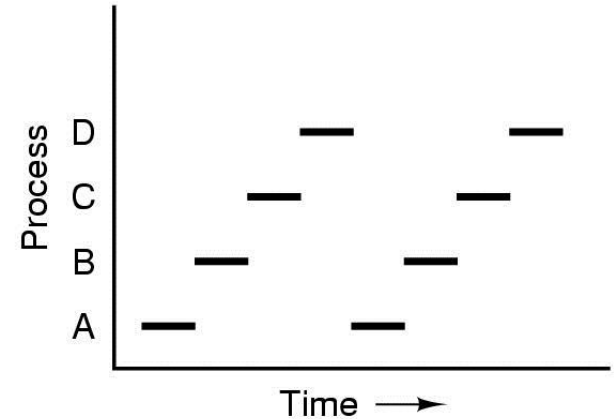


(a)

Four program counters



(b)



(c)

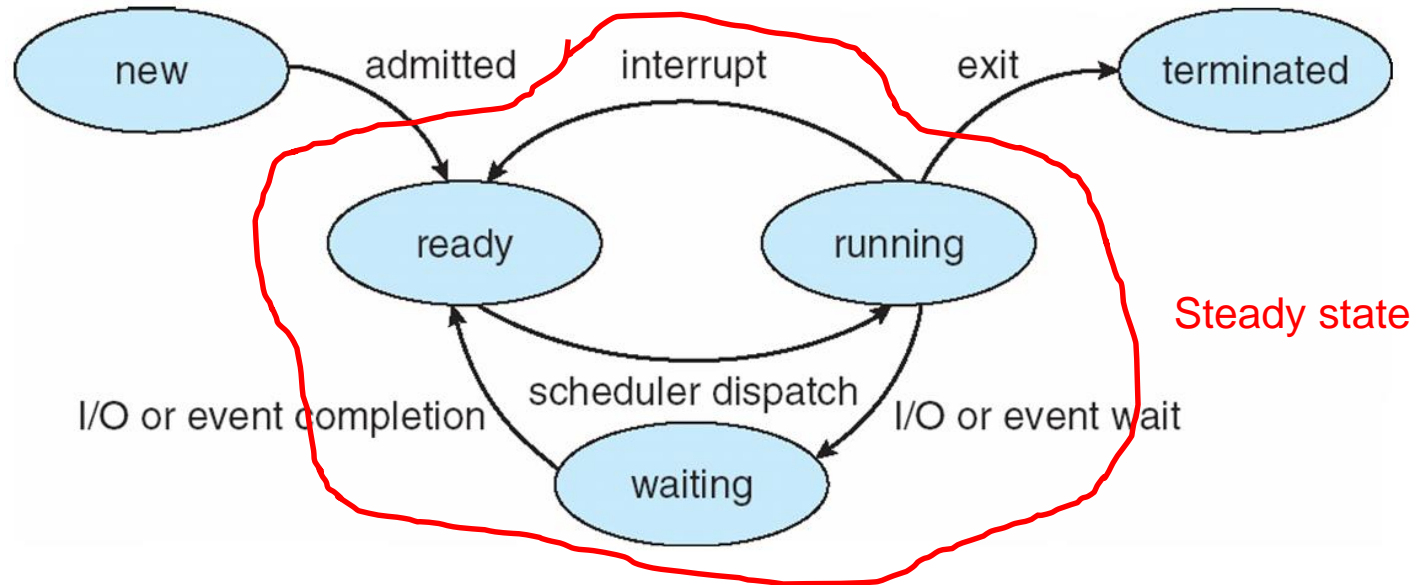
- a) Multiprogramming of four programs on a single CPU
- b) Conceptual model of 4 independent, concurrent, sequential processes
- c) Only one program active at any instant on a single CPU



# Process States

- As a process executes, it changes process state (status).
- The list of possible process states:
  - new: The process is being created. (transient state)
  - running: Instructions are being executed on the CPU.
  - blocked (waiting): The process is waiting for some event to occur (e.g., I/O request to complete) – It cannot do anything even if given the CPU.
  - ready: The process is ready to run and is waiting to be assigned to the CPU.
  - Terminated (dead): The process has finished execution. (transient state)

# Diagram of Process State Transitions



- Possible process states during their lifetime:
  - running
  - Blocked/waiting
  - ready
- Transitions between states shown

# Process Creation

Principal events that cause process creation (new process):

1. System initialization
2. Execution of a process creation system
3. User request to create a new process
4. Initiation of a batch job

# Process Creation

- Processes are created and deleted dynamically.
- **Process Creation**: Process which creates another process is called **parent** process.
- Created process is called **child** process.
- Parent process creates children processes, which, in turn create other processes, forming a tree of processes.

# Process Creation

- UNIX examples (for historical reasons)
  - **Fork()** system call creates a new process → clone of the parent process.
  - **Execve()** system call used after a **fork** to replace the process' memory space with a new program → overwrites the cloned parent process.
- On Win32, the combination of these two is done by a single system call
  - **CreateProcess()**

# UNIX Example

## more on fork()

```
int prntpid;
int chldpid;
if ((chldpid = fork()) == -1 ) {
    perror("can't create a new process");
    exit(1);
}
else if (chldpid == 0) { /* child executes */
    printf("chld: chldpid=%d, prntpid=%d \n",
        getpid(), getppid());
    exit(0);
}
else { /* parent process executes */
    printf("prnt: chldpid=%d, prntpid=%d\n",
        chldpid, getpid());
    exit(0);
}
```

# Process Creation

- Execution possibilities (different OS's implement differently):
  - the parent continues to execute concurrently with the child; or
  - parent waits until child has terminated.
- Address space possibilities:
  - child process is duplicate of the parent process; or
  - child process has a program loaded into it during creation.

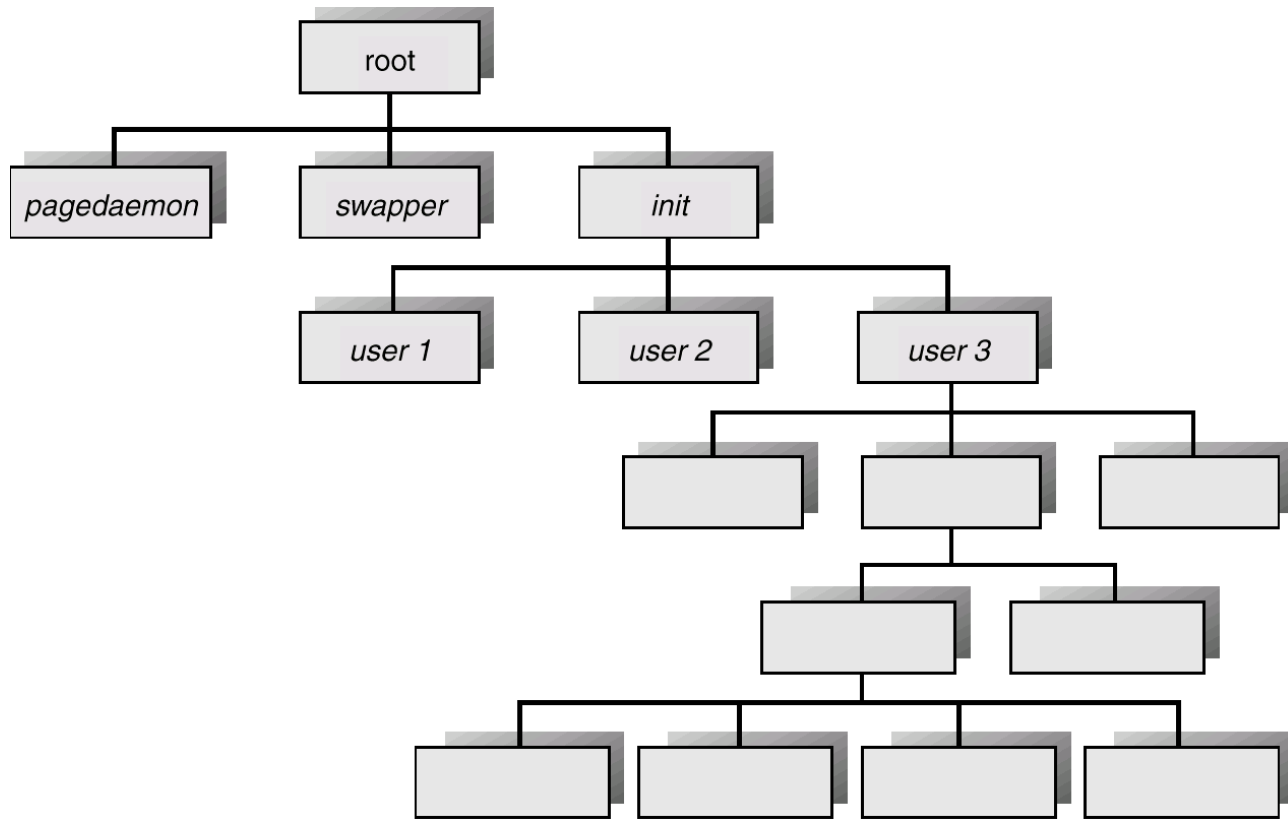
# Process Hierarchies

In many OS's processes exist in a hierarchy:

- Parent creates a child process, child processes can create their own child processes, etc.
- Forms a hierarchy
  - UNIX calls this a “process group” or process tree.



# Process Tree on a UNIX System



# Process Termination

Conditions which terminate processes

1. Normal exit (voluntary)
2. Error exit (voluntary)
3. Fatal error (involuntary)
4. Killed by another process (involuntary)

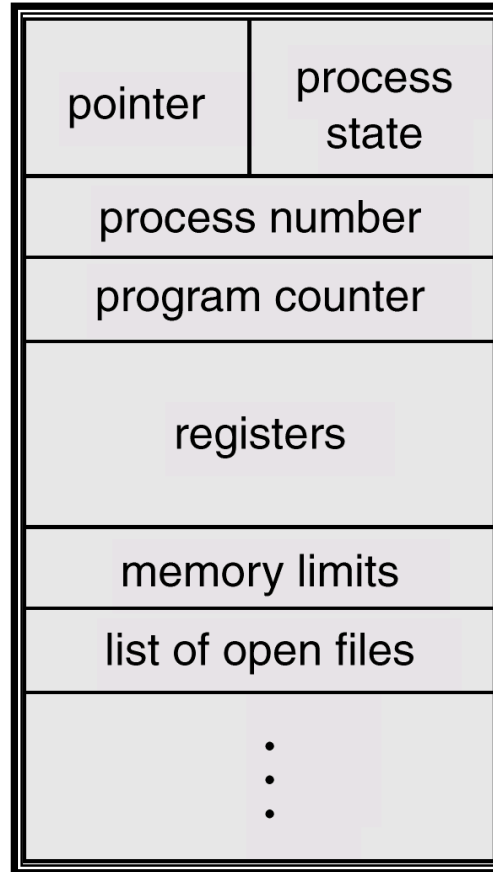
# Process Termination

- Process executes last statement and asks the operating system to execute system call (**exit**).
  - Output data from child to parent (via **wait**).
  - Process' resources are deallocated by operating system.

# Process Termination

- Parent may terminate execution of children processes (**abort** or **kill**).
  - Child has exceeded allocated resources.
  - Task assigned to child is no longer required.
  - Parent is exiting.
    - Operating system does not allow child to continue if its parent terminates.
    - Cascading termination.

# Implementation of Processes



- Through **process table** or **process control block (PCB)**
  - Similar structures for threads (i.e., thread table)

# Process table information

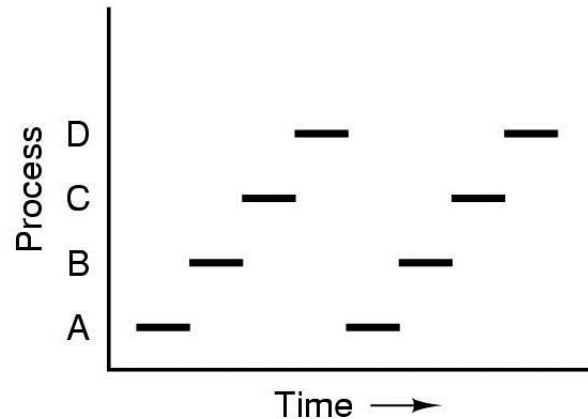
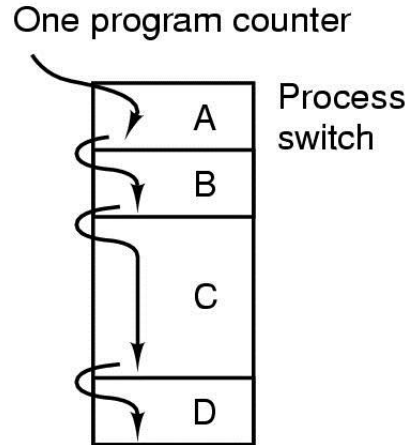
- What is included in the process table?
  - Information about the state of computation necessary for process management (e.g., the value of the PC, processor register contents, etc).
  - Information about memory management (e.g., page tables and other memory related information)
  - Information about file management (e.g., open file descriptors, etc).

# Implementation of Processes

Process management	Memory management	File management
Registers Program counter Program status word Stack pointer Process state 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	Pointer to text segment Pointer to data segment Pointer to stack segment	Root directory Working directory File descriptors User ID Group ID

- Fields of a process table entry

# Revisit multiplexing CPU



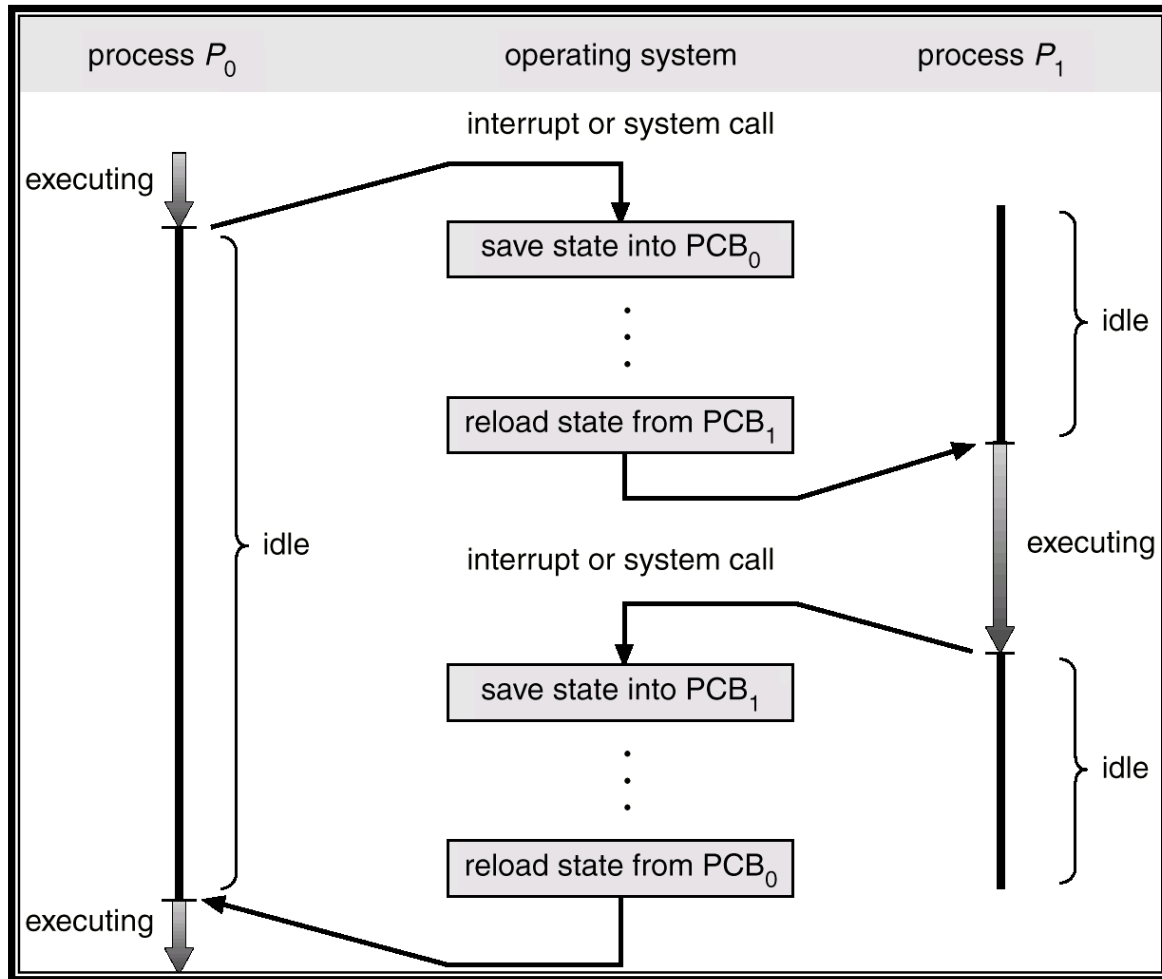
- One CPU is shared between multiple processes over time (multiplexed).
- This is accomplished via “context switch”



# Context Switch

- The act of switching CPU from one process to another is called **Context Switch**.
- Context Switch must do:
  - save PCB state of the old (switched out) process;
  - load PCB state of the new (switched in) process, which might include changing the current register set, advanced memory management techniques moving data from one memory segment to another;

# Context switch



# Context Switch

- Context Switch represents a pure overhead because the system does not do any useful work during switching (Context switching time 1-1000 microseconds).
- Context switch highly depends on hardware support.
- Context switch can become a bottleneck.

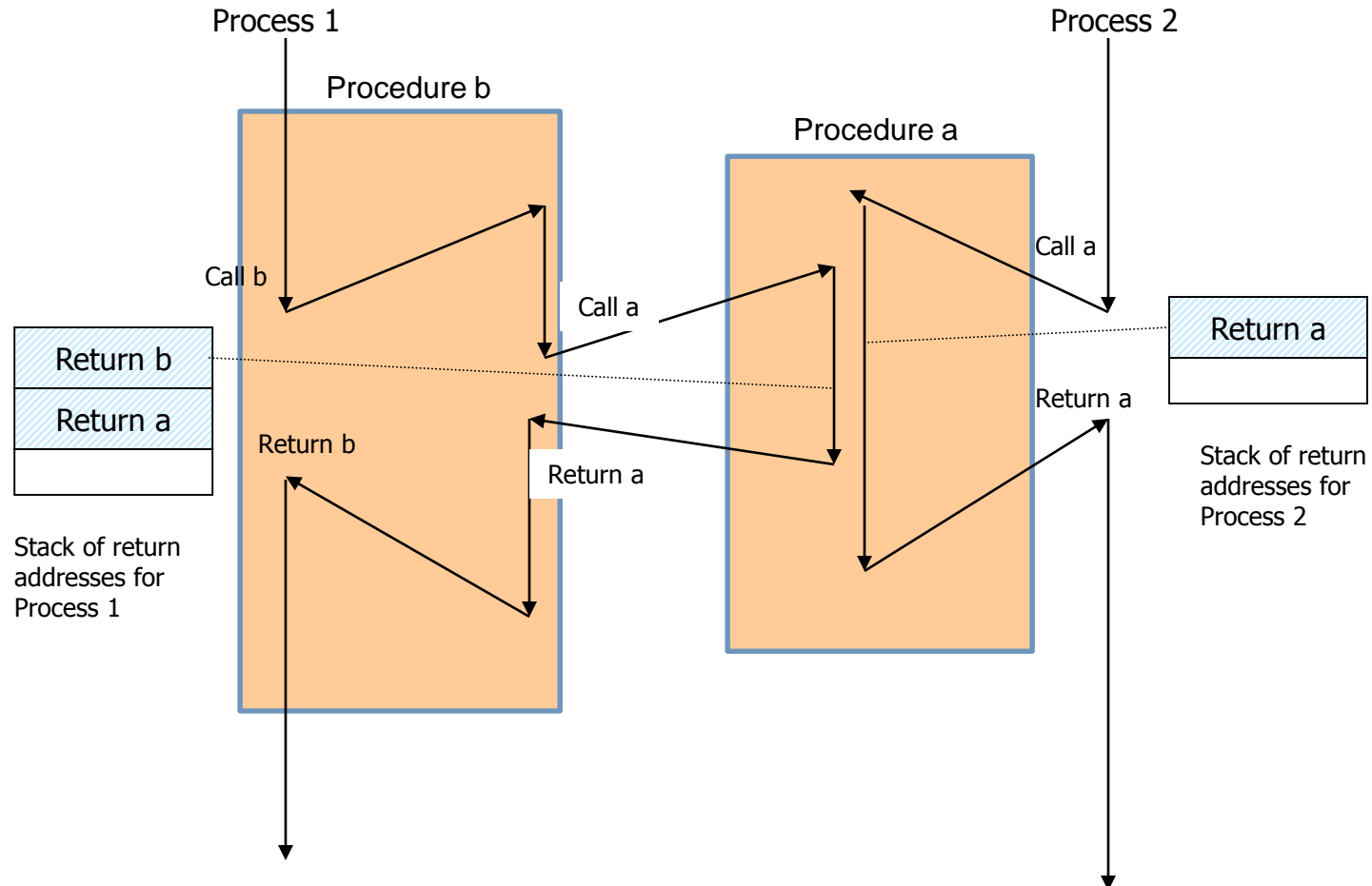
# A little history

- Reentrant Code
- Co-routines
- Processes
- Threads

# Reentrant Code

- Two or more processes may invoke and execute a common procedure concurrently.
- Such procedures must use reentrant code to store **process specific data** in storage locations that are local to the process.
- Reentrant codes cannot be self-modifying.

# Reentrant Code



# Co-routines

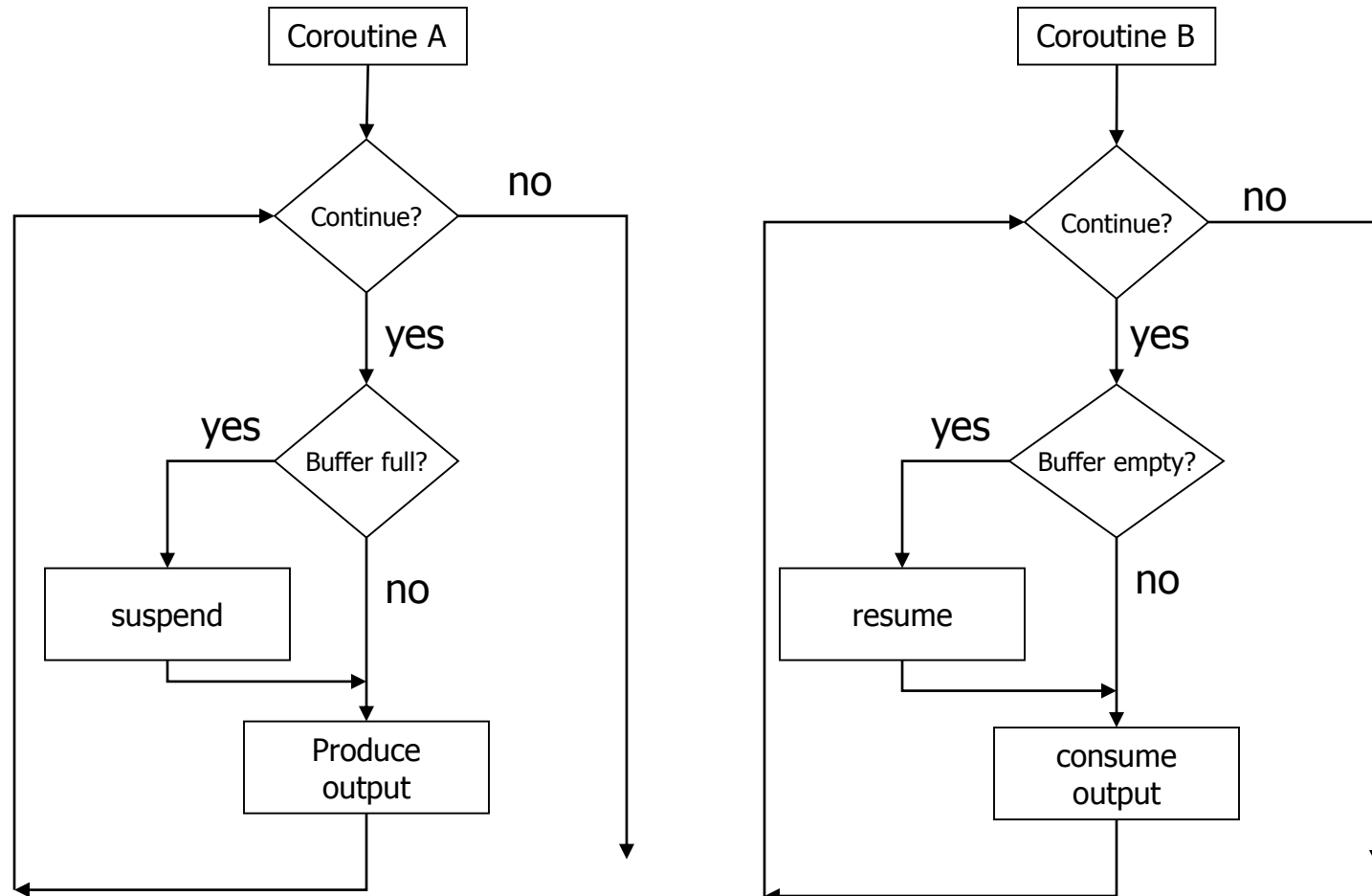
- **coroutines** are program components that generalize **subroutines (functions)** to allow multiple entry points (re-entrant code) and suspending and resuming of execution at certain locations
- Suspending and resuming done by explicit transfer of control through
  - Suspend
  - Resume
- Invented to cope with *asynchronous & cooperative* tasks.

# Co-routines

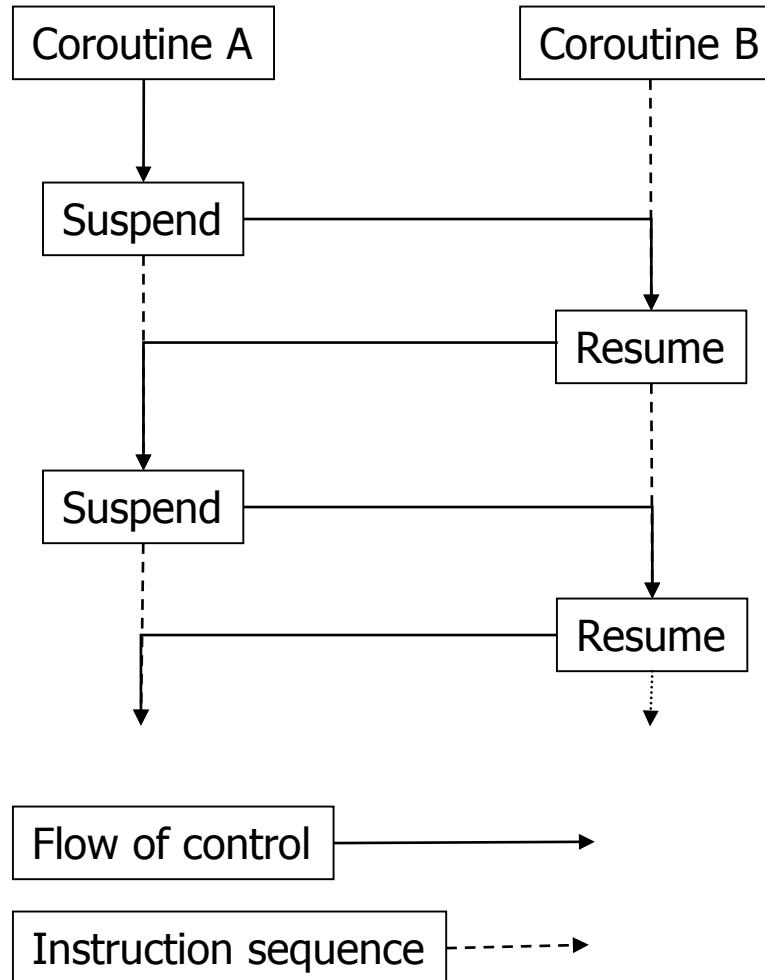
- Co-routines are *interdependent*.
- They structure a program that contains several concurrent sequential activities.
- The *thread of control* winds between co-routines in a program. (Hence, the modern term “thread”.)
- Co-routines were provided by such high level programming languages as Simula & Modula-2.
- Not available in more modern languages or C.
  - More modern languages (such as Java) provide thread support.



# Producer/Consumer Co-routine



# Producer/Consumer thread of Control



# Processes & Co-routines

- **Process**

- Each Process has an “abstract” locus of control.
- Any **exchanges of control** are done **automatically** and **implicitly**.
- May run in parallel on multiprocessor.

- **Co-routine**

- Co-routines specify concurrency of activities but they do not provide a mechanism with which to program parallelism on a multiprocessor.
- Co-routines require **explicit exchange of control**.

# Threads

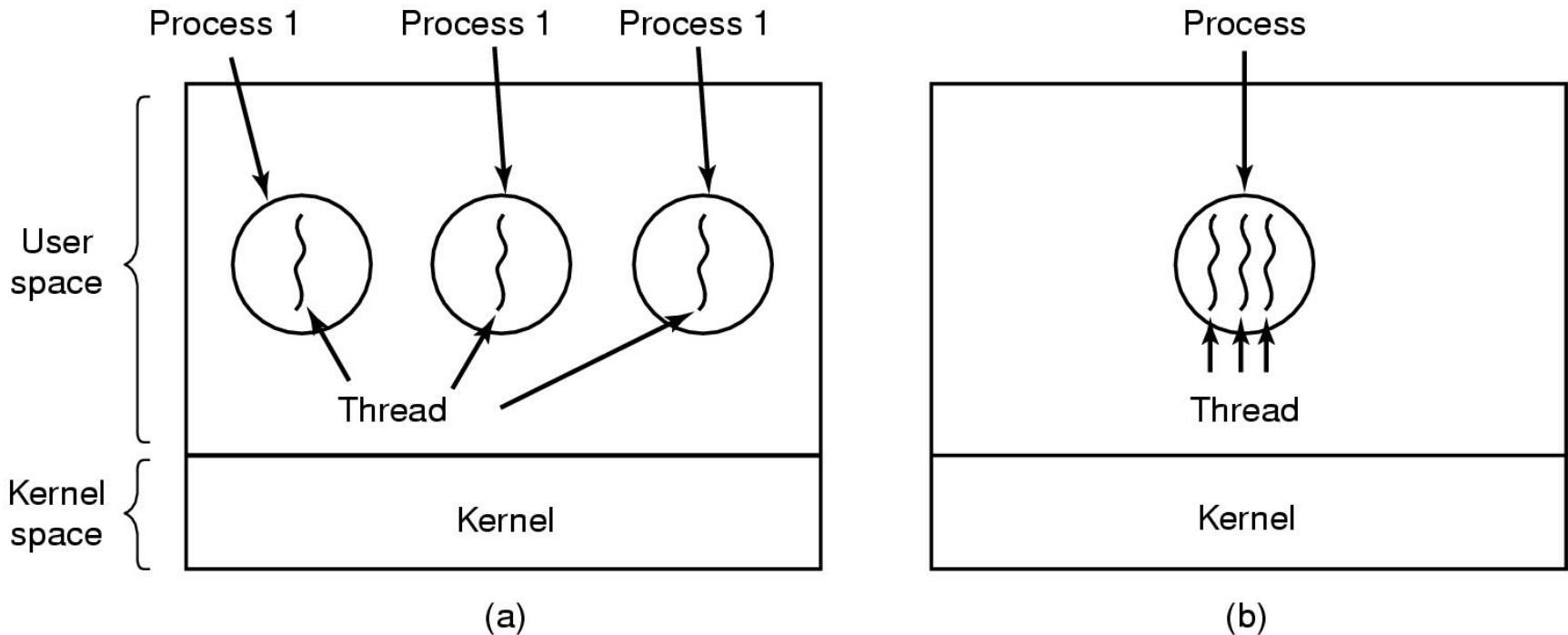
- **Threads** are the *modern analogue* of co-routines
  - Some degree of explicit control (**yield calls**)
  - Some degree of automation (**thread scheduling**)

# Threads in one Task (process)

- Processes do not share resources very well
  - shared variables/memory
  - open files
- Context switching of processes can be *very expensive*.
- **THREADS** (belonging to the same process) share some of the resources and, thus, have lower context switching overhead:
  - Shared information need not be saved and restored.

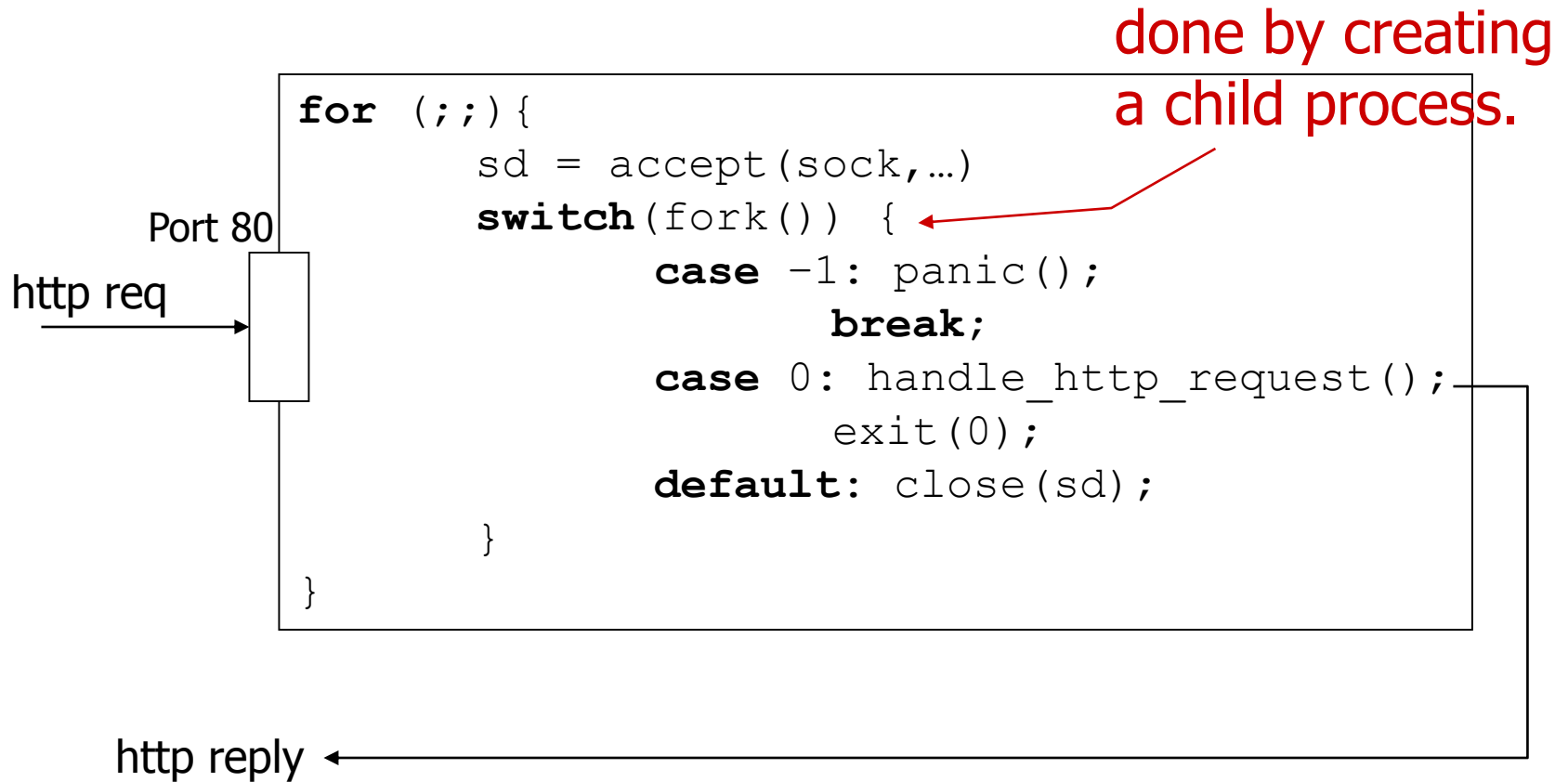
# Threads

## The Thread Model



- (a) Three processes each with one thread
- (b) One process with three threads

# Example: Vanilla web server



# Example: Vanilla web server

- Creating a new process for every http request would work, but would have a **big overhead**.
- Same thing could be accomplished by creating a *thread* to process the http request instead of creating a process.
- It would be more efficient.
- Example of this given further ahead.



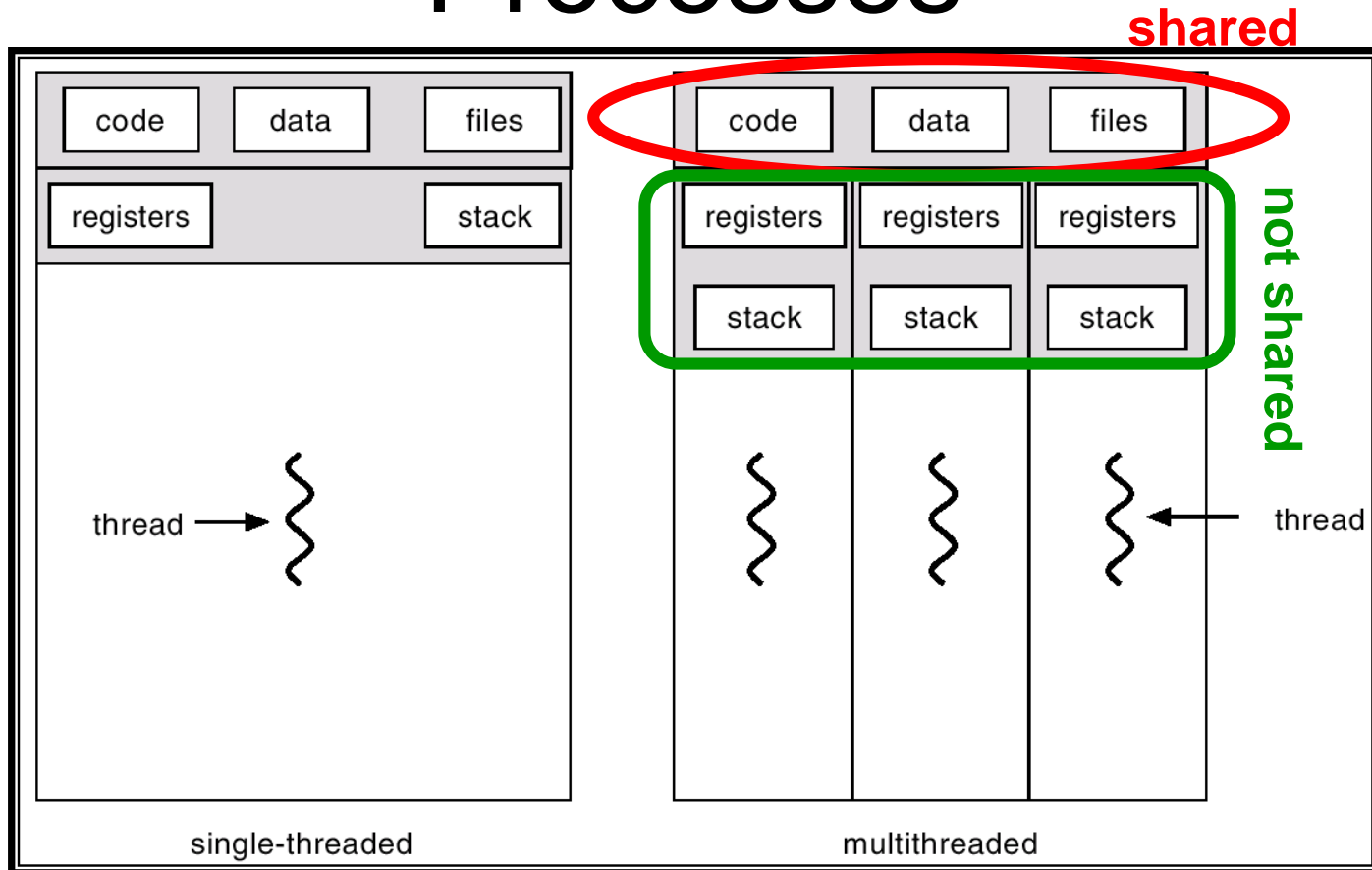
# Benefits of threads

- Responsiveness — because of low overhead of context switching
- Resource Sharing — Unlike processes, threads share a common address space.
- Economy — because of resource sharing.
- Utilization of Multiprocessor Architectures — easier to map multiple threads to multiple CPU's with shared memory.

# Threads within one process

- A thread **shares** with its peer threads its
  - code section,
  - data section and
  - OS resources such as
    - open files and
    - signals belonging to the process.
- A traditional process has one thread – sometimes called a **heavyweight** process.

# Single and Multithreaded Processes



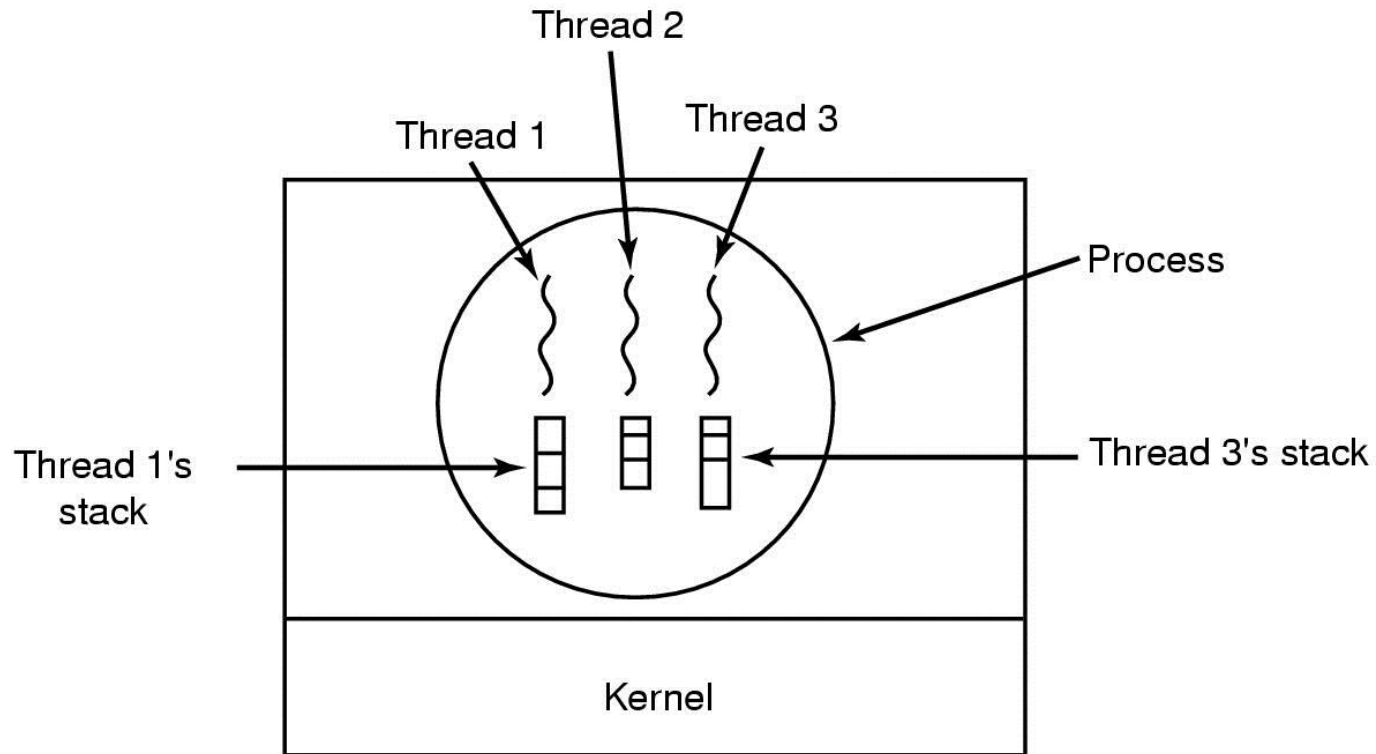
# The Thread Model

Items shared by all threads in a process

Items private to each thread

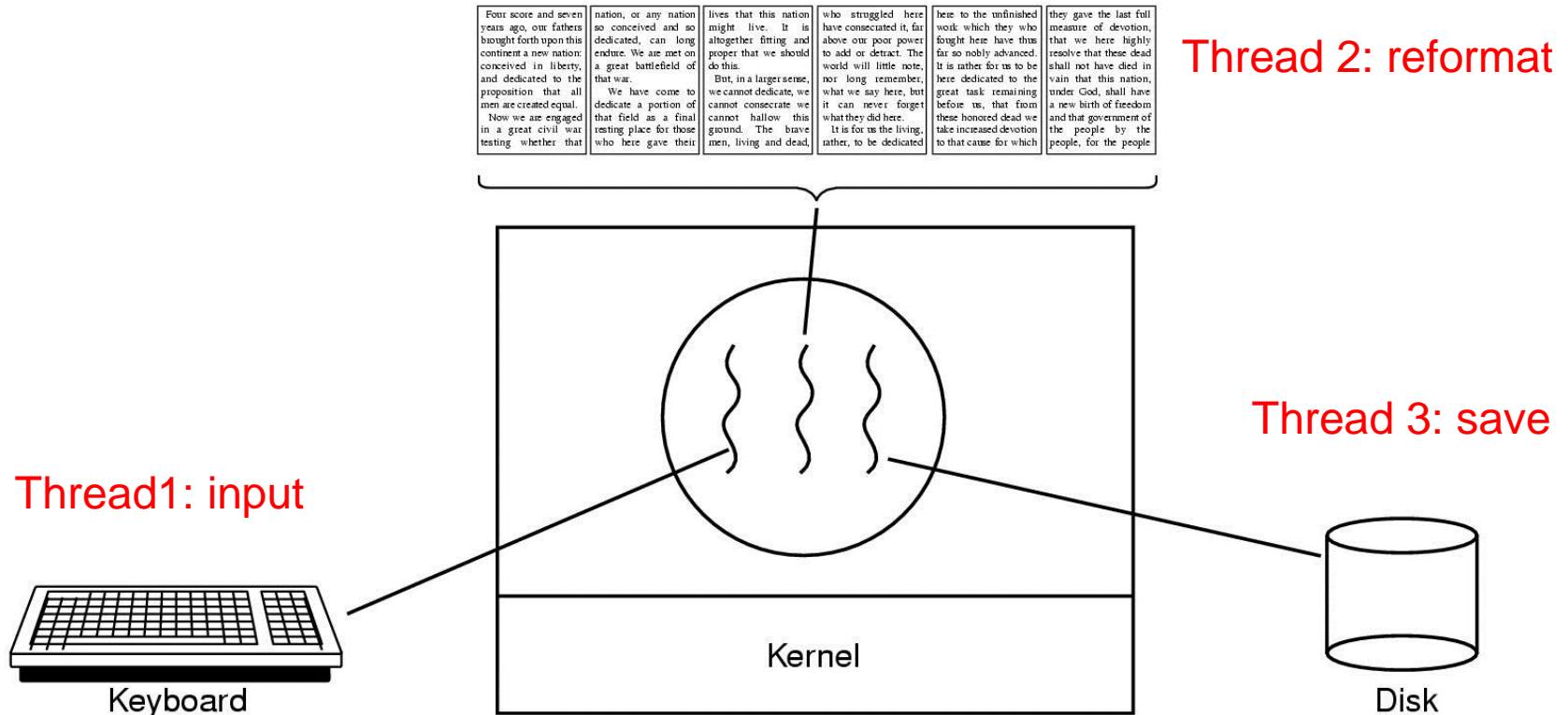
Per process items	Per thread items
Address space	Program counter
Global variables	Registers
Open files	Stack
Child processes	State
Pending alarms	
Signals and signal handlers	
Accounting information	

# The Thread Model



- **Each thread has its own stack** because it can have a different execution history with different function calls, etc.

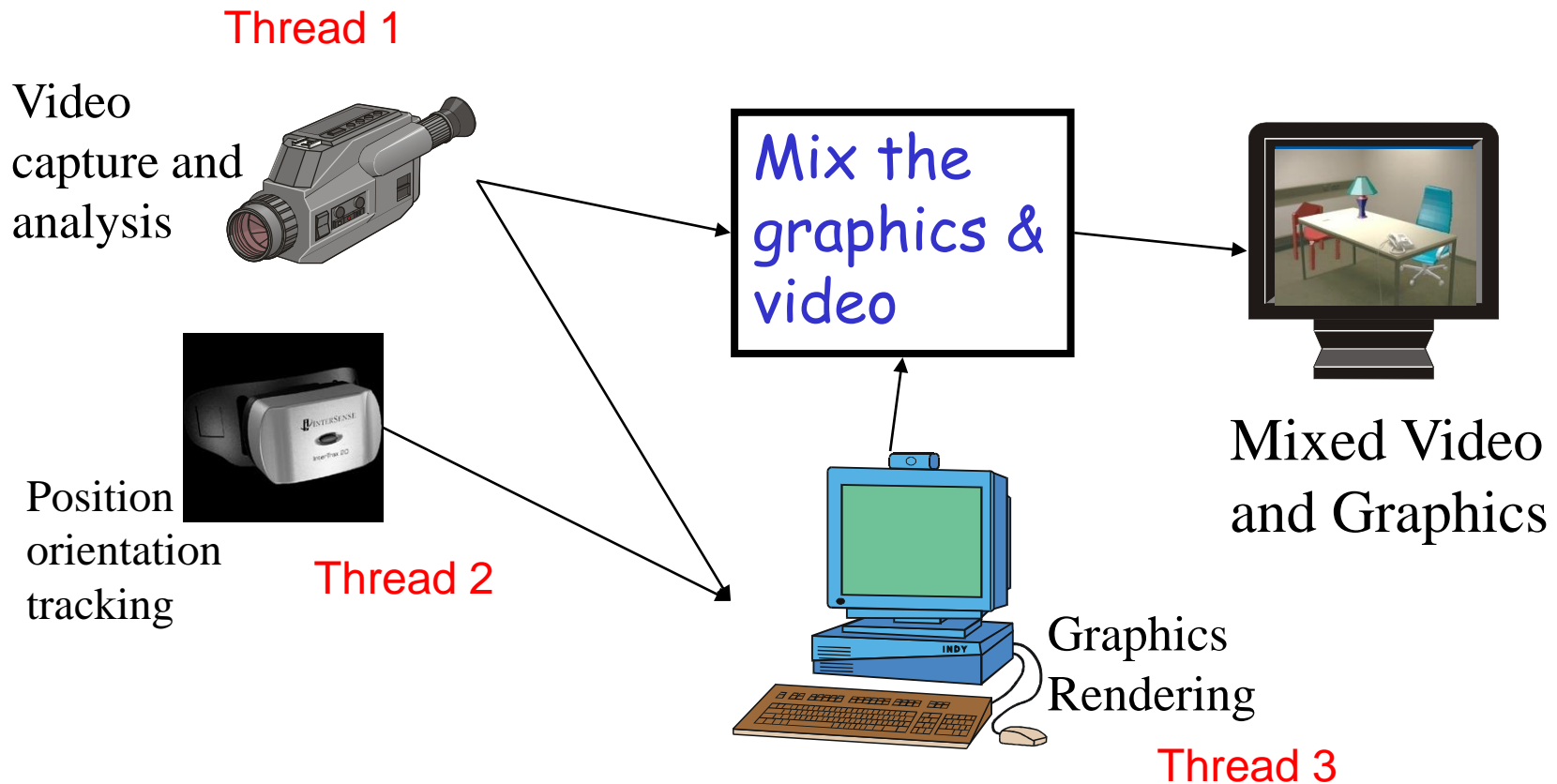
# Thread Usage



- A word processor with three threads: input, reformat, and save.
- Each can be implemented and thought of as a simple sequential computation.

# Other Thread Usage examples

- Graphics applications: Augmented Reality



# Operations on threads

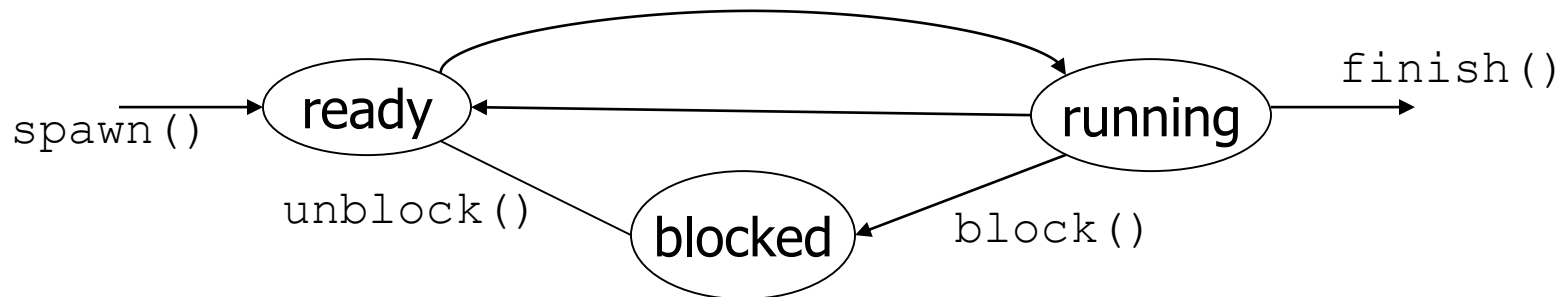
Similar to process operations:

- Thread creation  
int thread\_id = spawn(); (or fork())
- Thread blocking  
block();
- Thread unblocking  
unblock(int thread\_id);
- Thread termination  
finish();



# Thread states

- Threads go through states similar to processes: ready, blocked, running, terminated
- Unlike processes, threads are considered cooperating, therefore, there is no protection among threads.



# Threads

- In a multiple threaded process, while one server thread is blocked and waiting, a second thread in the same task can run.
  - Cooperation of multiple threads in same job confers higher throughput and improved performance.
  - Applications that require sharing a common buffer (i.e., producer-consumer) benefit from thread utilization.

# Threads

- Extensive sharing makes CPU switching among peer threads and creation of thread inexpensive compared to heavyweight processes.
- Thread context switch still requires a register set switch, but **no memory management related work!** → Less overhead; more efficient.

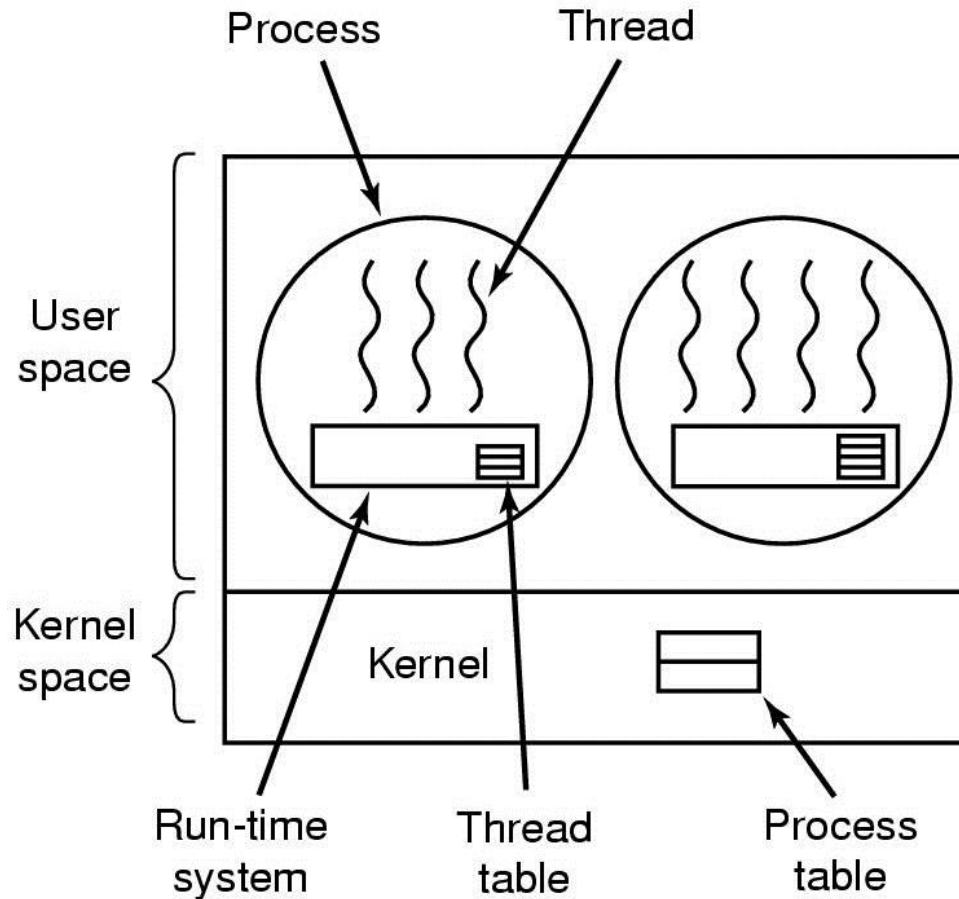
# Thread models

- Types of thread implementations:
  - **Kernel-supported** threads (Mach and OS/2).
  - **User-level threads**; supported above the kernel, via a set of library calls at the user level (Project Andrew from CMU).
  - **Hybrid** approach implements both user-level and kernel-supported threads (Solaris 2).

# User Threads

- Thread management done by user-level threads library
- Kernel does not know about threads (only processes)
- Examples
  - POSIX *Pthreads*
  - Mach *C-threads*
  - Solaris *threads*

# User Threads



A user-level threads package

# User Threads

- Reduces context switch times by eliminating kernel overhead
- No need to involve the kernel → context switch done in the user space
- Flexibility: Allows user scheduling of processes
- Need new primitives for processes to coordinate user with system  
example: may need **synchronization** primitives

# User-level Threads

- User-level threads are implemented
  - in user space
  - using user-level libraries, not system calls.
    - i.e., they don't go through the kernel.
  - threads do not depend on calls to OS and **do not cause interrupts or traps** to the kernel.
  - User-level threads require a stack and a program counter.



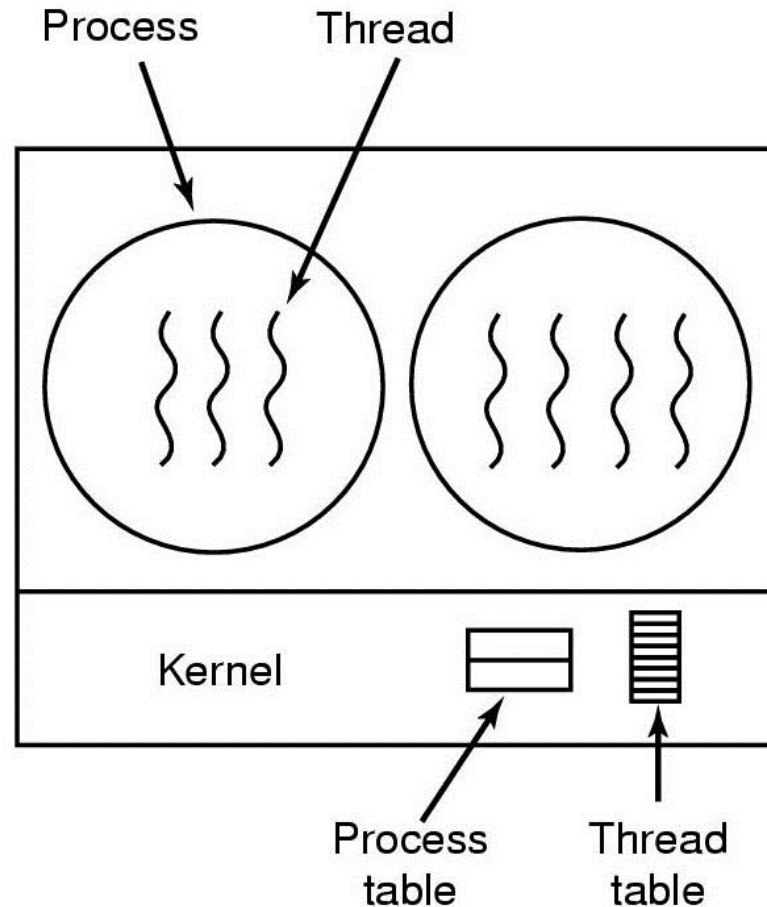
# Disadvantages of User-level Threads

- If kernel is single threaded, then any user-level thread can block the entire task executing a single system call.
  - For example an I/O request must involve the kernel. To the kernel this will look like an I/O request from one user process. → process is blocked → all threads in this process will be blocked.
- No concurrency for I/O operations

# Kernel Threads

- Supported by the Kernel
- Many examples from real OS's
  - Windows
  - Solaris
  - Tru64 UNIX
  - BeOS
  - Linux

# Kernel Threads



- A threads package managed by the kernel

# Kernel (system) threads

- Done by going to the kernel to do context switching
- OS does the scheduling of threads
- Lightweight threads – switch between threads but not between memories (i.e., heavyweight processes)
- Allows user programs to continue after starting I/O and/or making blocking system calls.

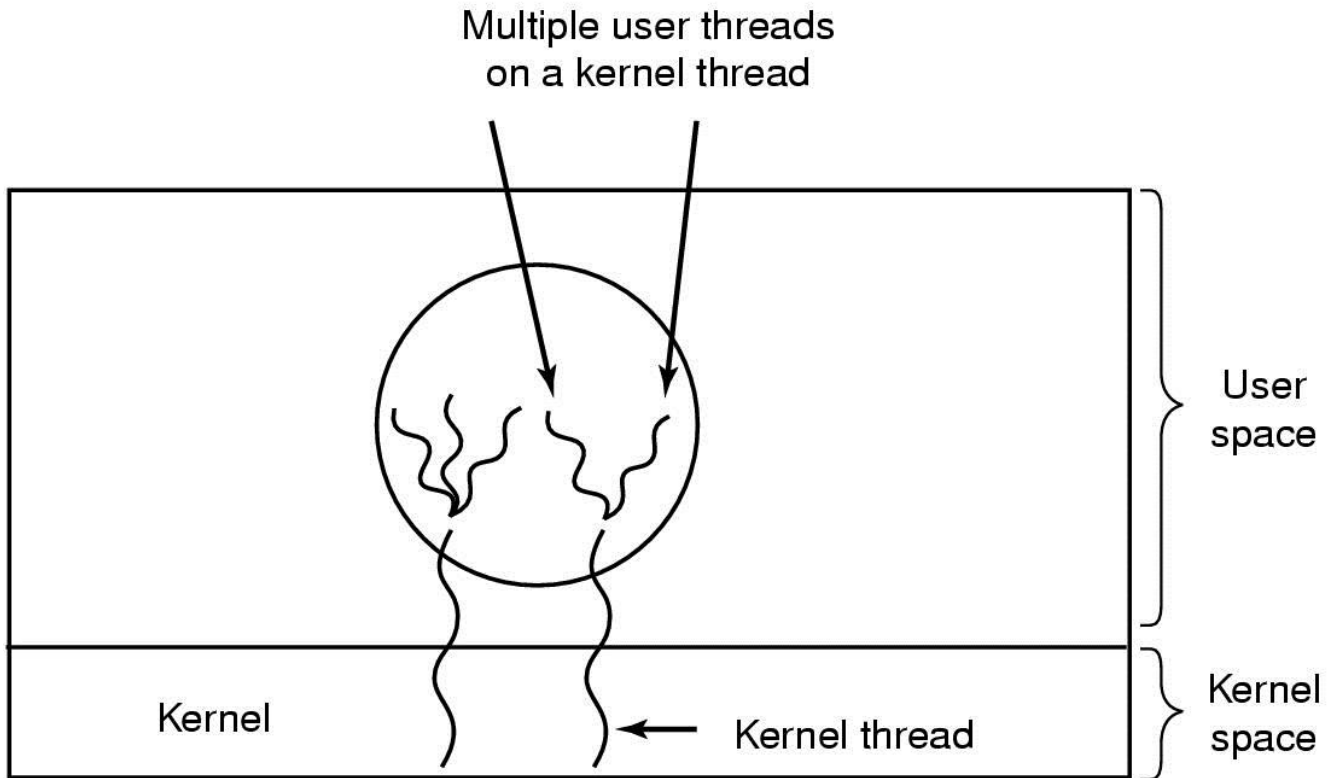
# Kernel (system) threads

- User cannot explicitly schedule threads.
- The user can control some of the scheduling by passing parameters to the kernel (e.g., indicating priorities, etc. that will affect the scheduling).
- Allows parallel processing
  - Kernel knows about possibly multiple CPU's.
  - Kernel can share the ready queue between the CPU's.

# Kernel threads

- Disadvantages
  - Less control over scheduling than user-level threads
  - Heavier than user threads (system calls to kernel are involved → inherently more expensive)
  - Cannot change the thread model.

# Hybrid Implementations



- Multiplexing user-level threads onto kernel-level threads

# Multithreading Models

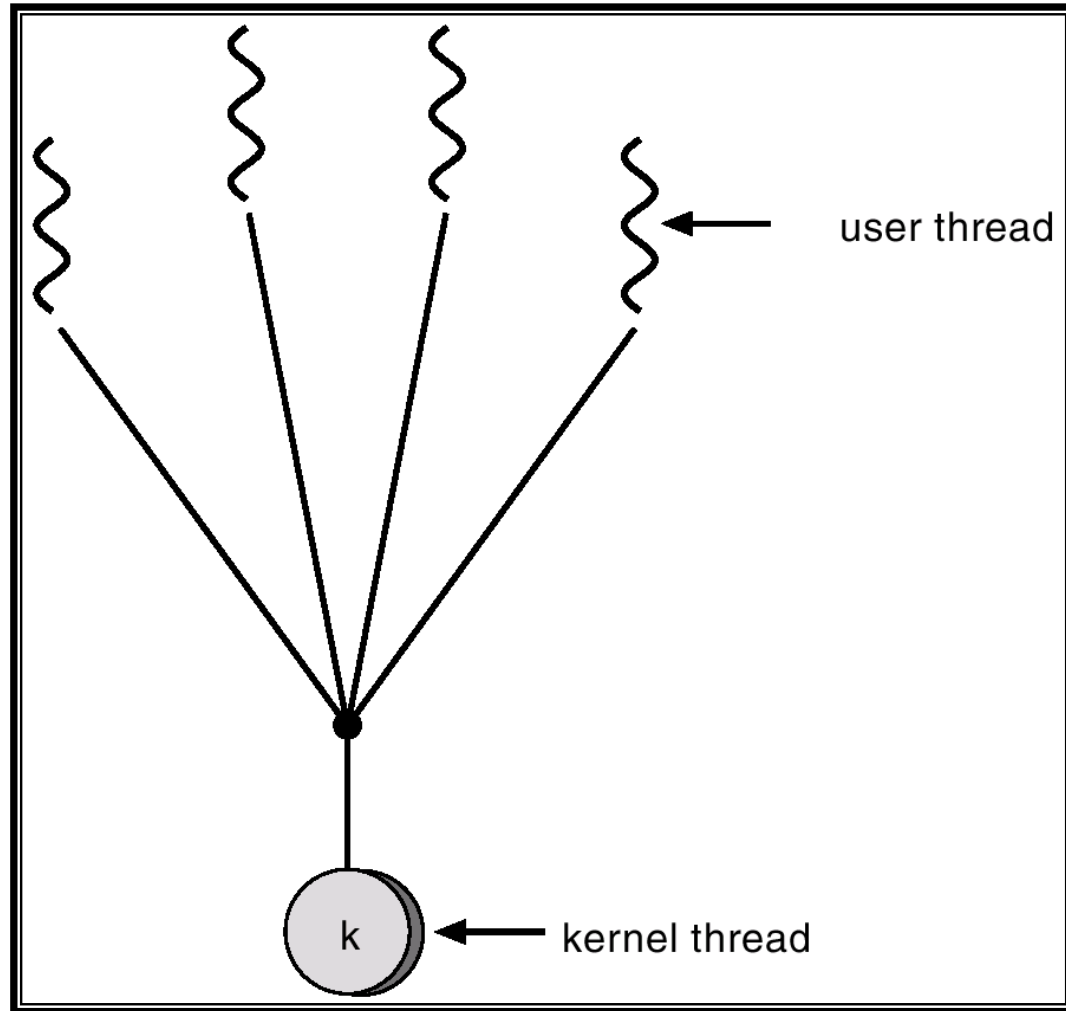
- Many-to-One
- One-to-One
- Many-to-Many



# Many-to-One

- Many user-level threads mapped to single kernel thread.
- Used on systems that do not support kernel threads.

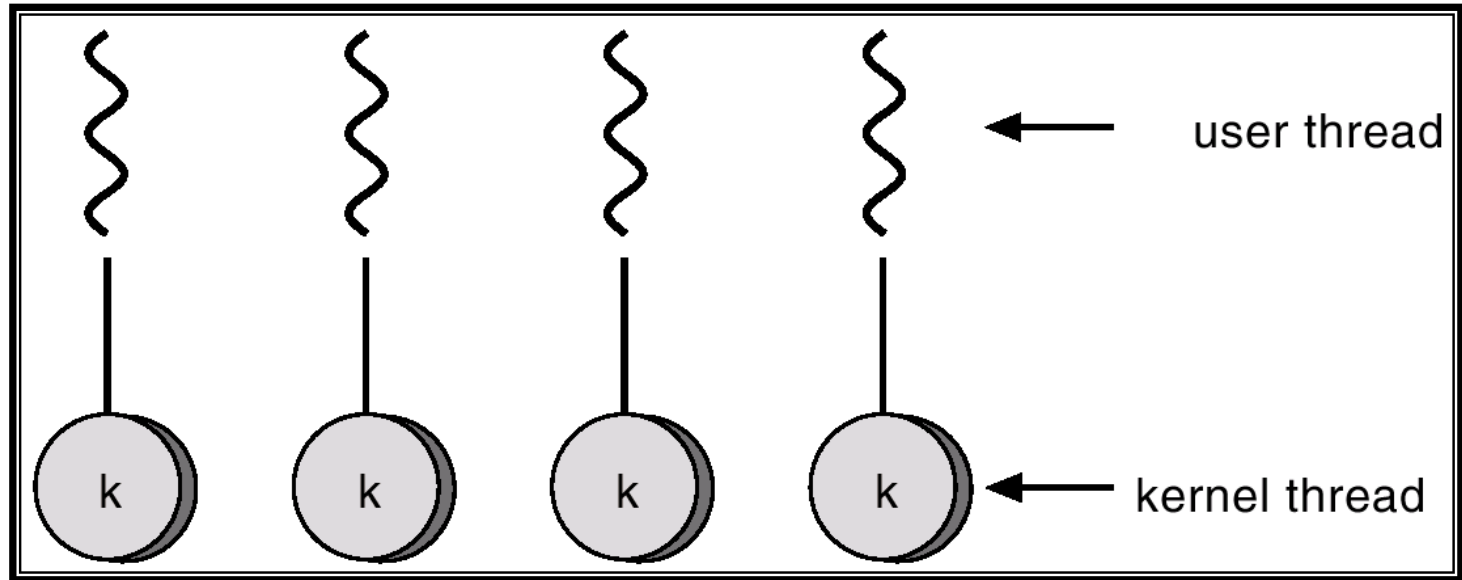
# Many-to-One Model



# One-to-One

- Each user-level thread maps to kernel thread.
- Examples
  - Windows 95/98/NT/2000
  - OS/2

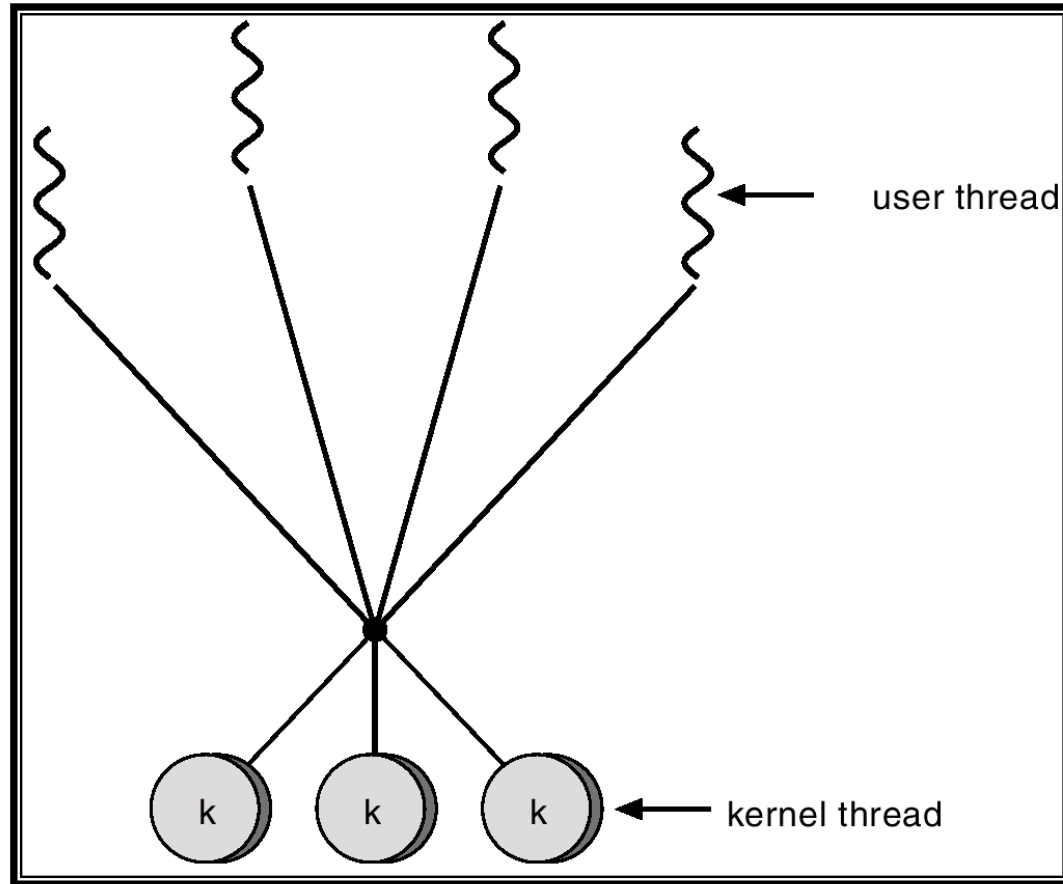
# One-to-one Model



# Many-to-Many Model

- Allows many user level threads to be mapped to many kernel threads.
- Allows the operating system to create a sufficient number of kernel threads.
- Solaris 2
- Windows NT/2000 with the ThreadFiber package

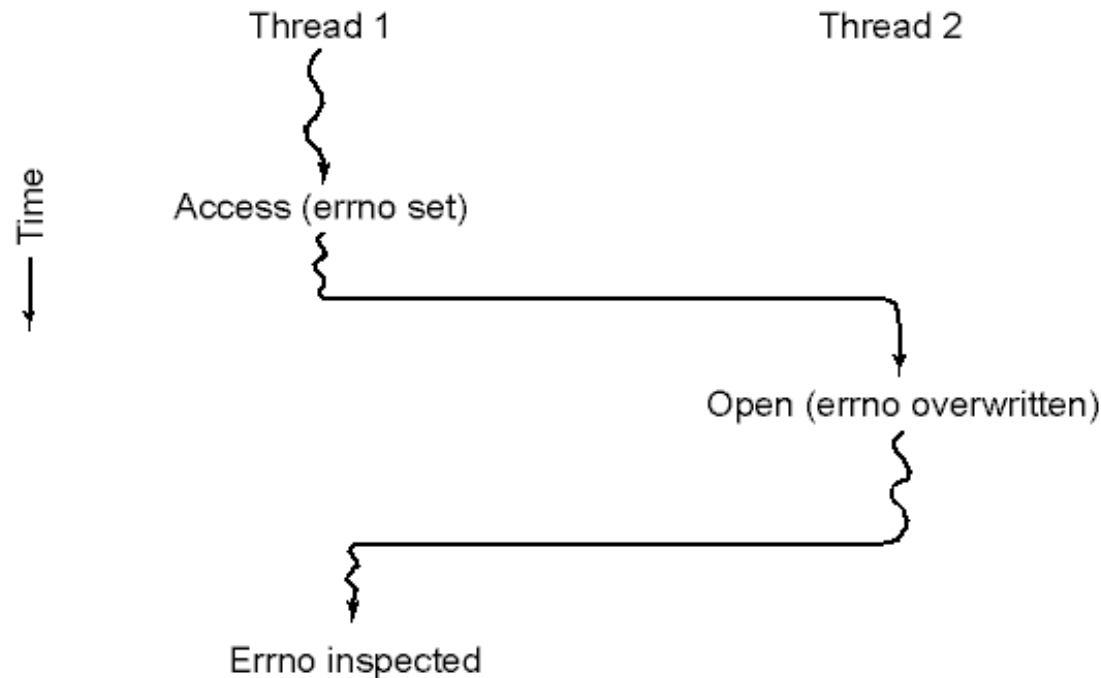
# Many-to-Many Model



# Threading Issues

- Semantics of `fork()` and `exec()` system calls.
  - when a process is spawned, do we create clones of all the threads in the parent process?
- Global variables
  - If each thread is to have a global variable, how is this handled? (ex: `errno`; see next slide)
  - Similarly issues with `malloc()`.
- Signal handling – which thread should catch the signals.
  - Especially important in user-threads. Kernel doesn't know about multiple threads.

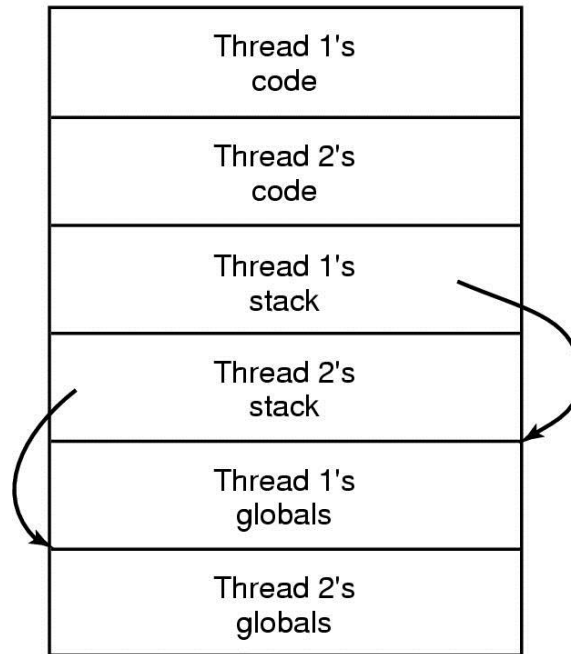
# Making Single-Threaded Code Multithreaded



- Conflicts between threads over the use of a global variable
- If one thread causes an error and gets suspended, next thread will think an error occurred in it.



# Making Single-Threaded Code Multithreaded



## One possible solution:

- Threads can have private global variables.
- Complicates the threads implementation.

# Processes and Threads

## Scheduling

# Contents

- Scheduling objectives
- Scheduling levels
- Batch systems
- Interactive systems
- Real-time systems

# Process Scheduling Objectives

- **Objective of multiprogramming:** maximal CPU utilization (always have a process running);
- **Objective of time-sharing:** switch CPU among processes frequently enough so that users can interact with a program which it is running;

# Scheduling Objectives

- Enforcement of fairness
- Enforcement of priorities
- Make best use of equipment
- Give preference to resources holding key resources
- Give preference to processes exhibiting good behavior
- Degrade gracefully under heavy loads

# Performance Terms

- **Fairness**
- **CPU and resource utilization** -- keep resources as busy as possible
- **Throughput** -- # of processes that complete execution in unit time
- **Turnaround time** -- amount of time to execute a particular process from the time it entered the system

# Performance Terms

- **Waiting time** -- amount of time process has been waiting in ready queue
- **Response time** -- amount of time from when a request was first submitted until first response is produced

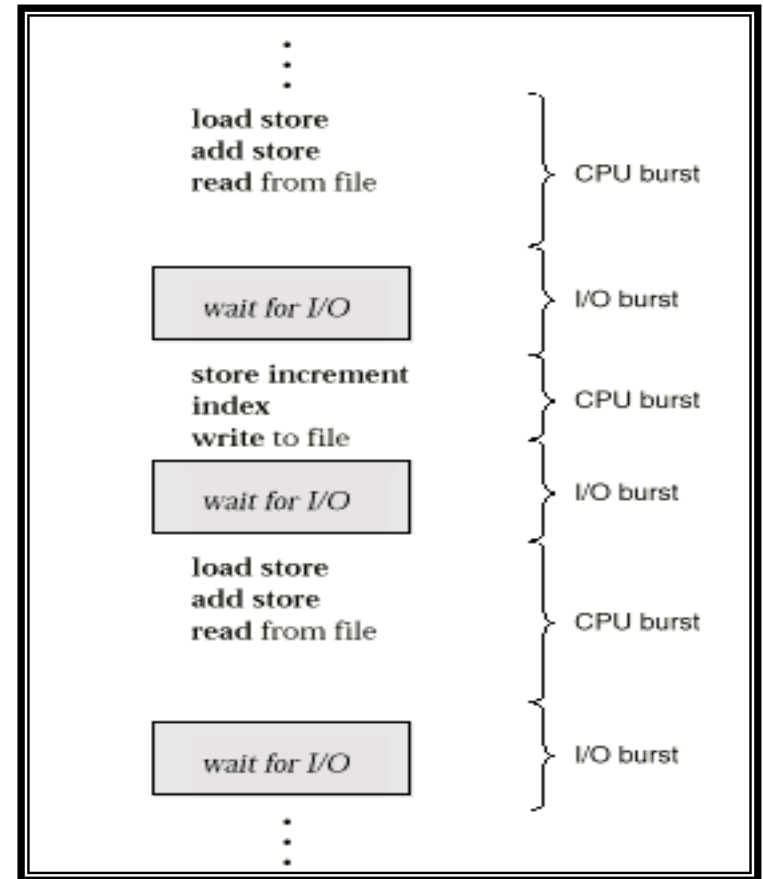
# Performance Criteria

- Max CPU utilization
- Max throughput
- Min turnaround time
- Min waiting time
- Min response time



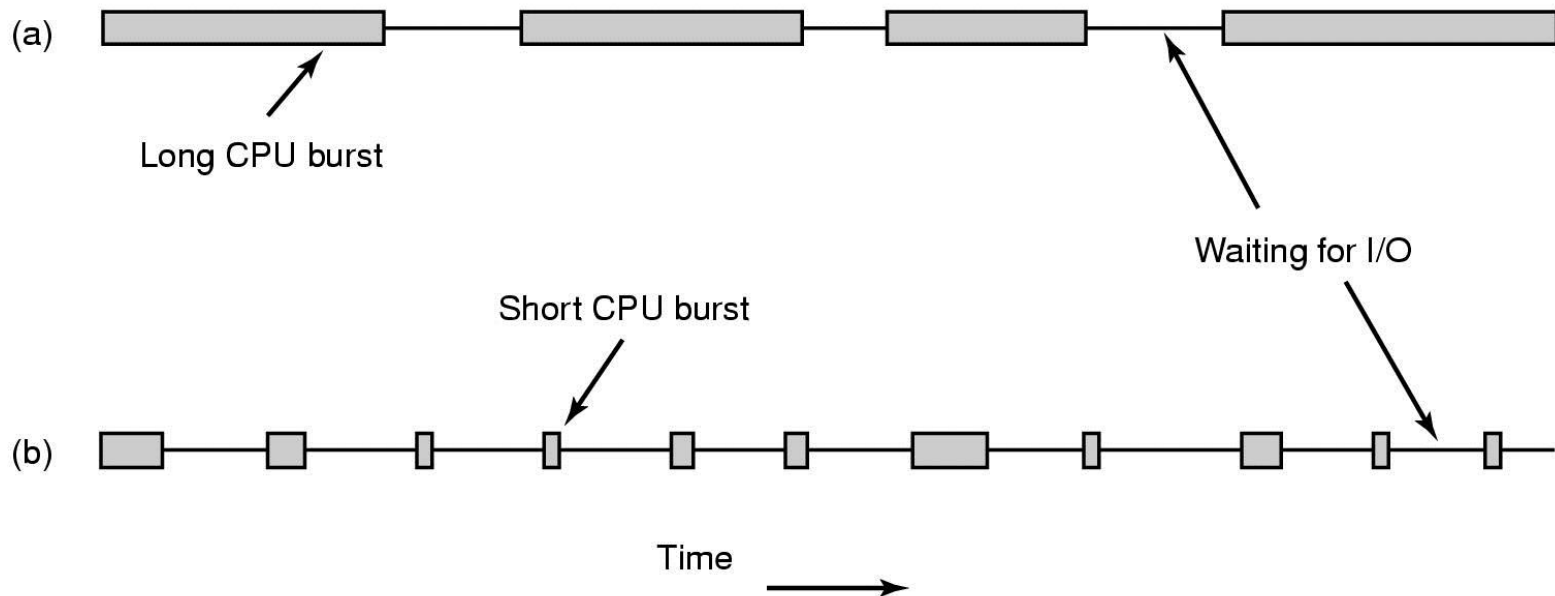
# Program behavior characteristics

- *CPU-I/O Burst Cycle*
  - Process execution consists of a *alternating cycles* of CPU execution and I/O wait during their lifetime.

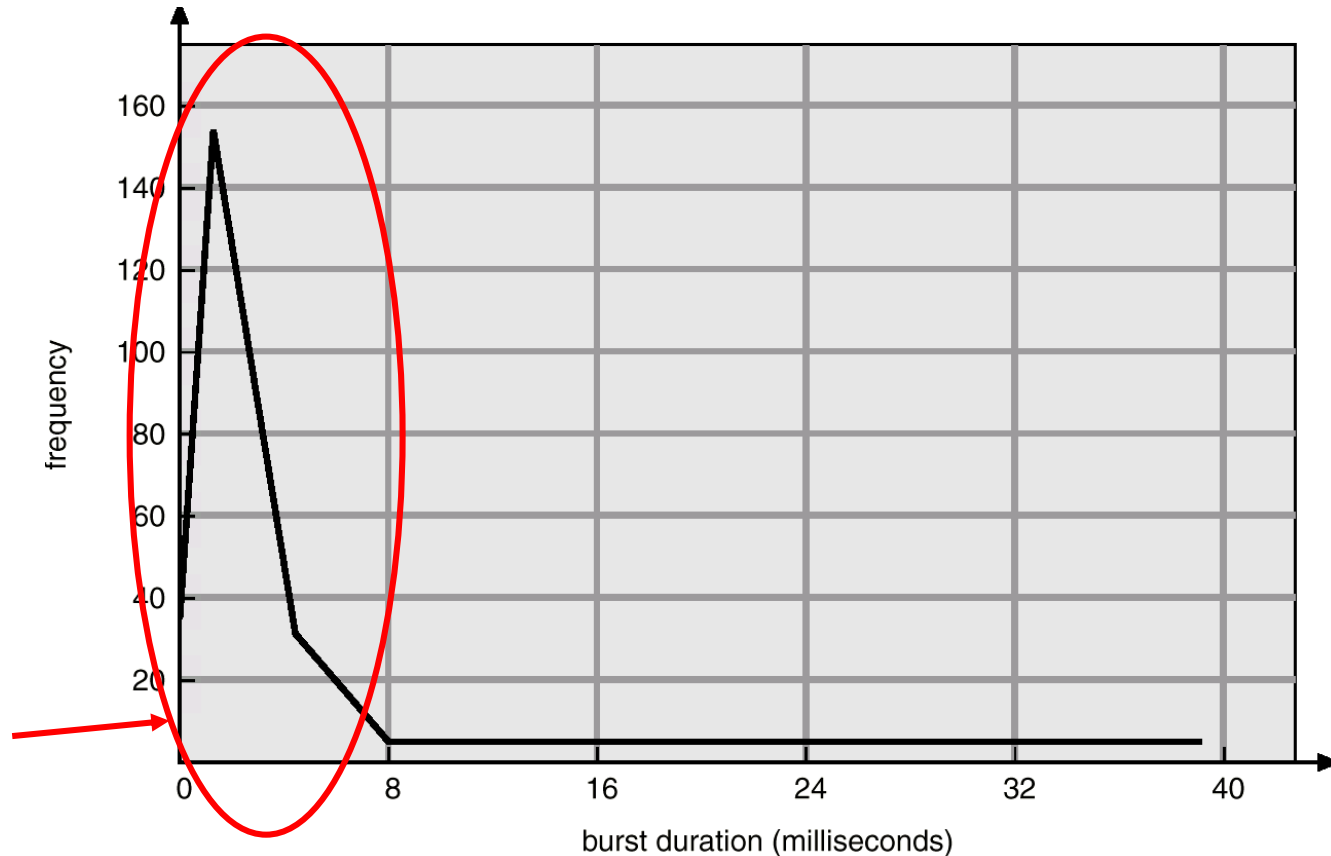


# CPU/I/O cycles

- Some processes have long CPU bursts between I/O requests
  - a CPU-bound process
- And some have short CPU bursts
  - an I/O bound process



# Histogram of CPU Burst Times



Most  
programs  
have  
CPU  
burst  
length in  
this  
range

# Scheduling Algorithm Goals

## **All systems**

Fairness - giving each process a fair share of the CPU

Policy enforcement - seeing that stated policy is carried out

Balance - keeping all parts of the system busy

## **Batch systems**

Throughput - maximize jobs per hour

Turnaround time - minimize time between submission and termination

CPU utilization - keep the CPU busy all the time

## **Interactive systems**

Response time - respond to requests quickly

Proportionality - meet users' expectations

## **Real-time systems**

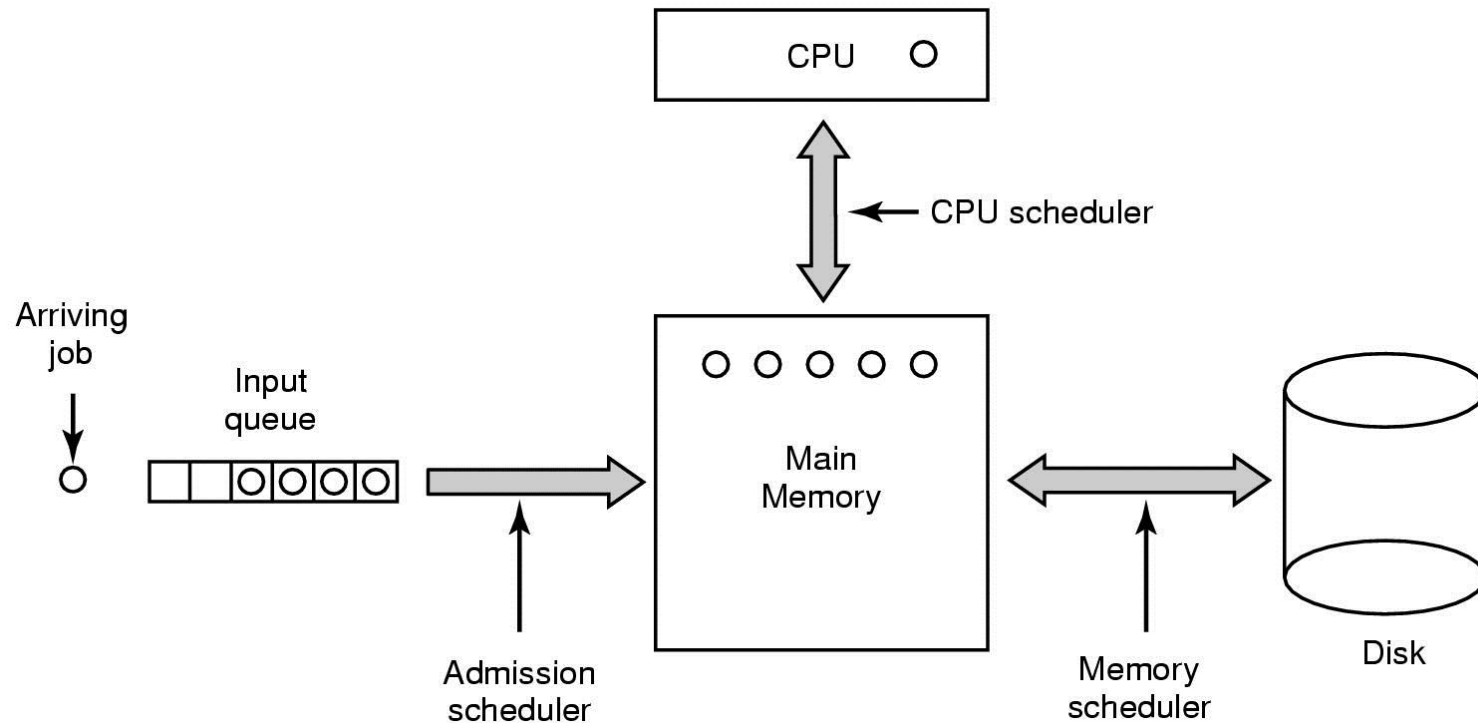
Meeting deadlines - avoid losing data

Predictability - avoid quality degradation in multimedia systems

# Scheduling Levels

- High level scheduling (admission scheduling) (Typically in batch systems.)
- Intermediate level scheduling (memory scheduling) (tries to balance system load)
- Low level scheduling (CPU scheduling)

# Scheduling in Batch Systems



Three level scheduling

# High Level Scheduling

- Also called long term scheduling or admission scheduling. (Typically in batch systems.)
- Which jobs should be allowed to enter the system and compete for the
  - CPU and other resources.
- This scheduler selects jobs from the input queue
- The goal is to select a “good” mix of CPU-bound and I/O-bound jobs so that the system is well-utilized.
- Alternate selection criteria: select short jobs for admission with higher priority.

# Intermediate Scheduling

- Also called medium term scheduling or **memory scheduling**
- Scheduling for jobs admitted to the system.
  - Temporarily suspend, or
  - Resume to smooth fluctuations in system load
  - e.g., too many processes and not enough memory → swap some processes out to disk (hence the name “memory scheduling”)



# Low-level Scheduling

- Also called CPU scheduling, short term scheduling or dispatching
- Assigning a CPU to a ready process
  - Many algorithms we will see will concentrate on this.

# CPU Scheduler

- CPU scheduler selects one of the processes from the ready queue and allocates CPU to one of them;
- **Ready queue**: FIFO queue, tree queue, or unordered linked list, or priority queue;
- Elements in the ready queue are PCBs (**Process Control Block**) of processes

# Kinds of Scheduling

- Non-preemptive scheduling
- Preemptive scheduling

# Non-preemptive Scheduling

- Once the CPU is allocated to a process, the process keeps the CPU until
  1. the process exits (i.e., finishes), or
  2. it blocks (i.e., it switches to the wait state)
- Does not require any special hardware (e.g., Timer) unlike preemptive scheduling

# Preemptive Scheduling

- A process can involuntarily relinquish its CPU because a higher priority process is becoming ready to run
- Interrupt request due to
  - Timer, or
  - Device event: Key stroke

# Preemptive Scheduling Issues

- Preemptive scheduling has problems of interference between processes through shared data in an inconsistent state
- Mechanisms required to coordinate access to shared data (**synchronization**)
- Possible starvation

# Dispatcher

- Dispatcher gives control of the CPU to the process, scheduled by the short-term scheduler. Its functions:
  1. switching context,
  2. switching to user mode,
  3. jumping to the proper location in the user program
- Dispatcher must be fast
- Dispatch latency = time to stop process and start another one

# Scheduling in batch systems

- Mostly mainframe computers doing data processing.
- Can have batch processes on interactive systems also (e.g., using **cron** or **at** on Unix)
  - But they have to optimize their scheduling more for interactive environment.



# First Come First Serve (FCFS)

- Policy: process that requests the CPU FIRST is allocated the CPU FIRST
- FCFS is Non-preemptive scheduling
- Implementation: FIFO queues
  - Process entering the ready queue is inserted to the tail of the queue
  - The process selected from the ready queue is taken from the head of the queue

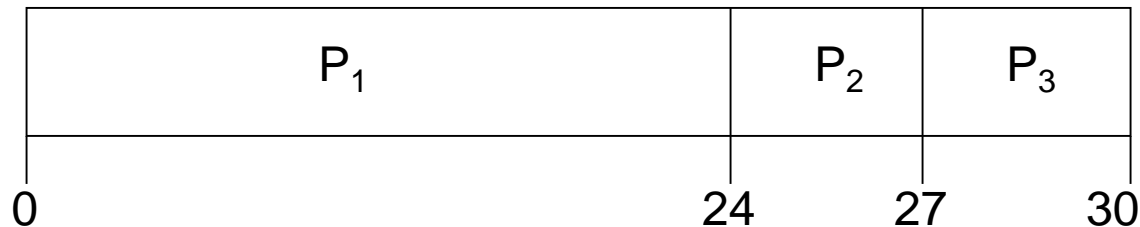
# First Come First Serve (FCFS)

- Performance metric: average waiting time
- Given parameters:
  - Burst time (in ms),
  - Arrival time and
  - Order
- Visualization of schedules - use Gantt charts
- Metric: average waiting time

# FCFS Scheduling

Example:	Process	Burst Time
	P1	24
	P2	3
	P3	3

- Suppose that the processes arrive at time=0 in the order: P1, P2, P3  
The Gantt Chart for the schedule is:



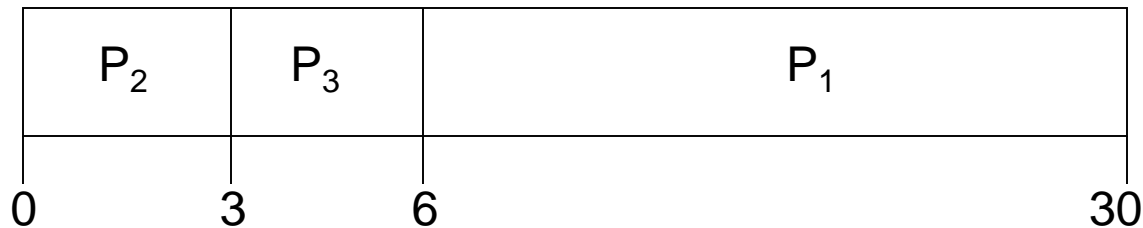
- Waiting time for P1 = 0; P2 = 24; P3 = 27
- Average waiting time:  $(0 + 24 + 27)/3 = 17$

# FCFS Scheduling

Suppose that the processes arrive at time=0 in the order

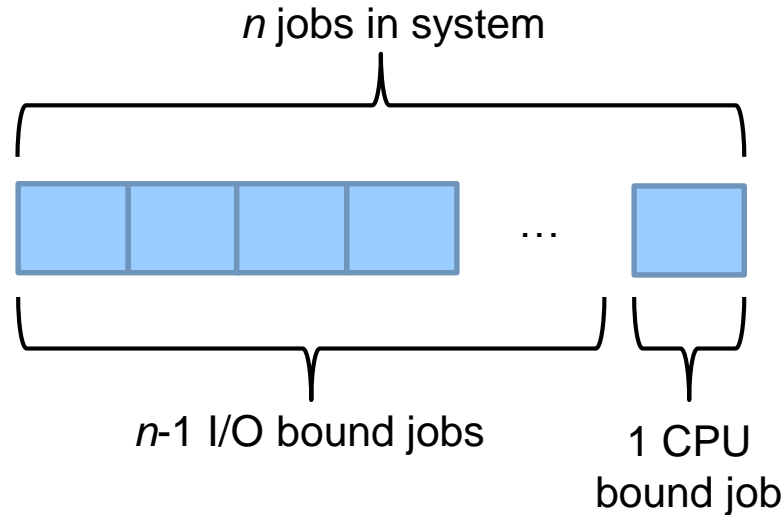
$P_2, P_3, P_1$ .

- The Gantt chart for the schedule is:



- Waiting time for  $P_1 = 6$ ;  $P_2 = 0$ ;  $P_3 = 3$
- Average waiting time:  $(6 + 0 + 3)/3 = 3$
- Much better than previous case.
- *Convoy effect*: short process behind long process

# Convoy effect in FCFS



- Remember FCFS is **non-preemptive**
  - I/O bound jobs pass quickly through the ready queue and suspend themselves waiting for I/O
  - CPU bound job arrives at head of queue and executes until burst is complete
  - I/O bound jobs rejoin ready queue and wait for CPU bound job to complete
- **I/O devices idle until CPU bound job completes** ← **underutilized I/O devices**
- When CPU bound job completes, other processes rush to wait on I/O again
- **CPU becomes idle** ← **underutilized CPU**

# FCFS Performance

- Result: **convoy effect, low CPU and I/O device utilization**
  - CPU bound process holds CPU while I/O bound processes wait in the ready queue → during this I/O devices are idle.
  - When I/O bound processes get hold of CPU, their CPU bursts are short → during this time CPU is idle.

# Shortest-Job-First (SJF) Scheduling

- Associate with each process the length of its next CPU burst. Use these lengths to schedule the process with the shortest time.
- Normally, we don't know ahead of time the job lengths.
- Used mostly as a benchmark (baseline for comparing other methods).

# Shortest-Job-First (SJF) Scheduling

- Two versions:
  - **nonpreemptive** – once CPU given to the process it cannot be preempted until it completes its CPU burst.
  - **Preemptive** – if a new process arrives with CPU burst length less than remaining time of current executing process, preempt. This scheme is also known as the Shortest-Remaining-Time-First (SRTF).
- **SJF is optimal** – gives minimum average waiting time for a given set of processes.



# Example of Non-Preemptive SJF

<u>Process</u>	<u>Arrival Time</u>	<u>Burst Time</u>
$P_1$	0.0	7
$P_2$	2.0	4
$P_3$	4.0	1
$P_4$	5.0	4

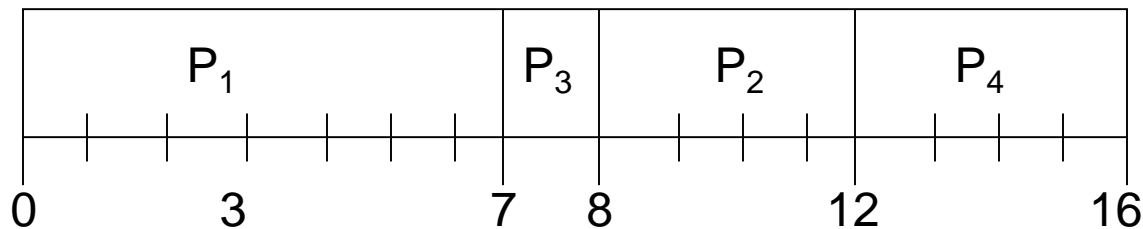
0 wait time

6 wait time

3 wait time

7 wait time

- SJF (non-preemptive)



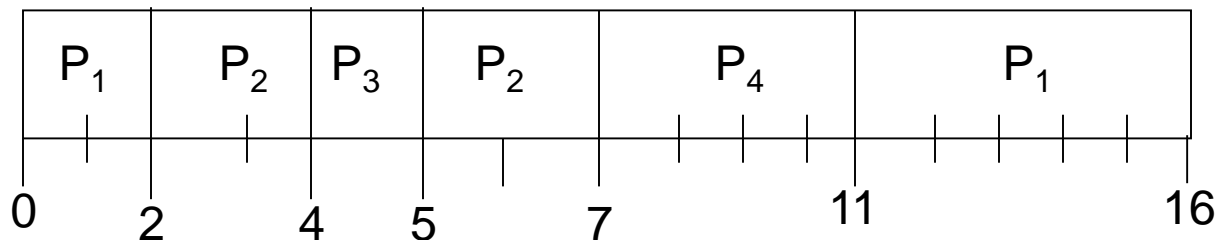
Arrives at 2,  
has to wait  
until 8

- Average waiting time =  $(0 + 6 + 3 + 7)/4 = 4$

# Example of Preemptive SJF (SRTF)

<u>Process</u>	<u>Arrival Time</u>	<u>Burst Time</u>
$P_1$	0.0	7
$P_2$	2.0	4
$P_3$	4.0	1
$P_4$	5.0	4

- SJF (preemptive)



- Average waiting time =  $(9 + 1 + 0 + 2)/4 = 3$

# SJF - Preemptive Vs. Non-preemptive

Process (Arrival Order):	P1	P2	P3	P4
Burst Time	8	4	9	5
Arrival Time:	0	1	2	3



$$\text{Average WT} := ((10-1) + (1-1) + (17-2) + (5-3))/4 = 6.5 \text{ ms}$$



$$\text{Average WT} := (0 + (8-1) + (12-3) + (17-2))/4 = 7.75 \text{ ms}$$

# Determining Length of Next CPU Burst

One possibility: guess the CPU burst based on history

- Can only estimate the length.
- Can be done by using the length of previous CPU bursts, using exponential averaging.
  1.  $t_n$  = actual length of  $n^{th}$  CPU burst
  2.  $\tau_{n+1}$  = predicted value for the next CPU burst
  3.  $\alpha, 0 \leq \alpha \leq 1$
  4. Define:  
$$\tau_{n+1} = \alpha t_n + (1 - \alpha)\tau_n.$$

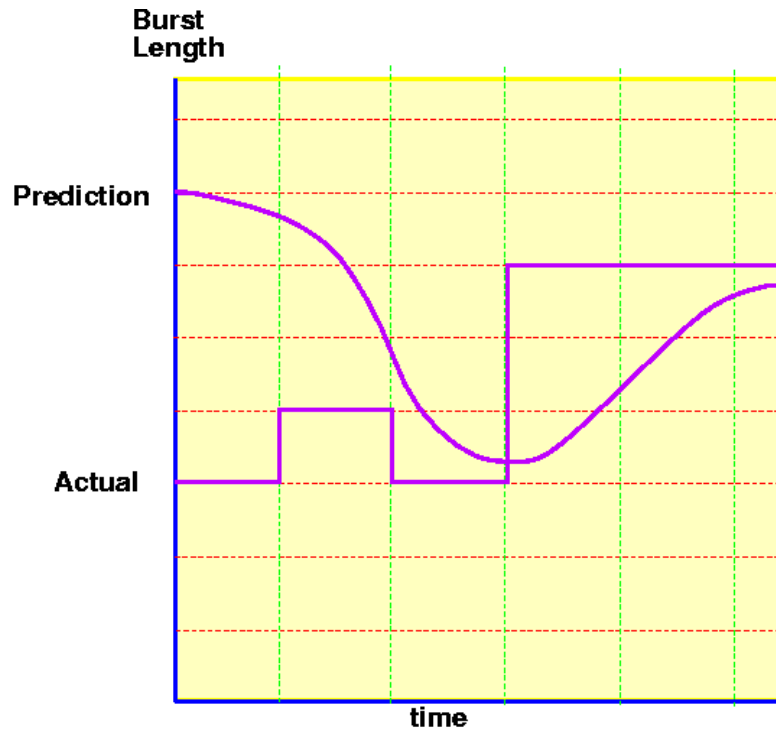
# Examples of Exponential Averaging

- $\alpha = 0$ 
  - $\tau_{n+1} = \tau_n$
  - Recent history does not count.
- $\alpha = 1$ 
  - $\tau_{n+1} = t_n$
  - Only the actual last CPU burst counts.

# Predicting the Length of a SJF Job

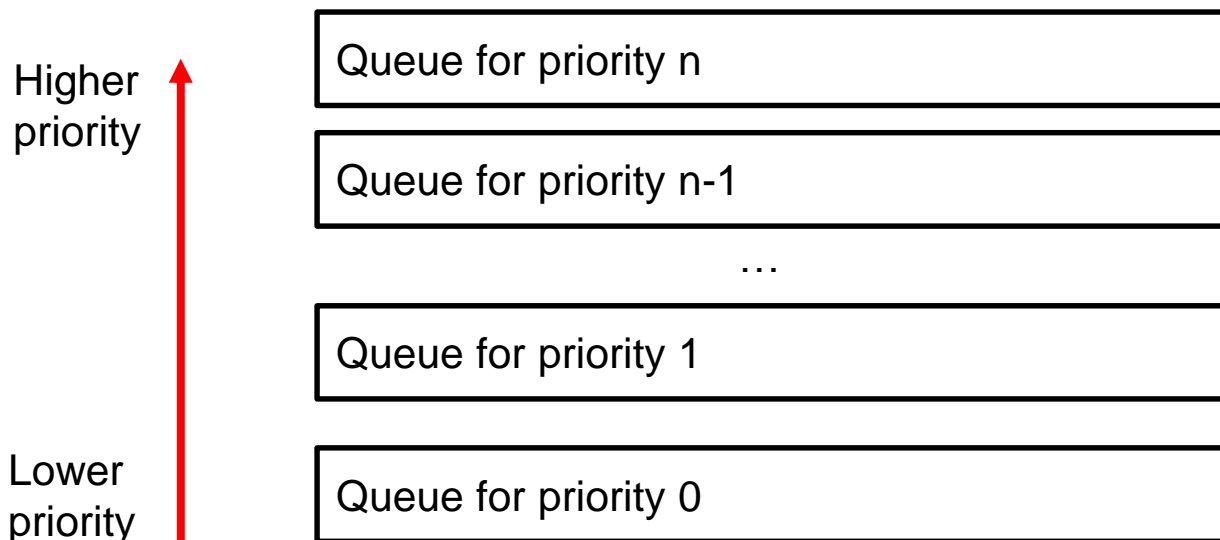
- Predicted value  $\tau_{n+1} = \alpha t_n + (1 - \alpha) \tau_n$
- Expanding the equation, we get
$$\tau_{n+1} = \alpha t_n + (1 - \alpha) \alpha t_{n-1} + \dots + (1 - \alpha)^j \alpha t_{n-j} + \dots + (1 - \alpha)^{n+1} \tau_0$$
- $0 \leq \alpha \leq 1$ . both  $\alpha$  and  $1 - \alpha$  are less than or equal to 1. Each term adds less to current value
- For  $\alpha = 0.5$  recent history and past history are equally weighted
- Version of SJF with preemption is called shortest remaining time first

# Exponential Average



# Priority Scheduling

- Can be used both in batch as well as interactive systems.
- Each job is assigned a priority
- FIFO within each priority level





# Priority Scheduling

- Select highest priority job over lower ones
- Starvation possible.
- Priority may be based on:
  - Cost to user
  - Importance of user
  - Aging (we'll talk more about this)
  - Percentage of CPU time used in last X hours

# Priority Scheduling - Example

Process (Arrival Order)	P1	P2	P3	P4	P5
Burst Time	10	1	2	1	5
Arrival Time	0	0	0	0	0
Priority	3	1	3	4	2

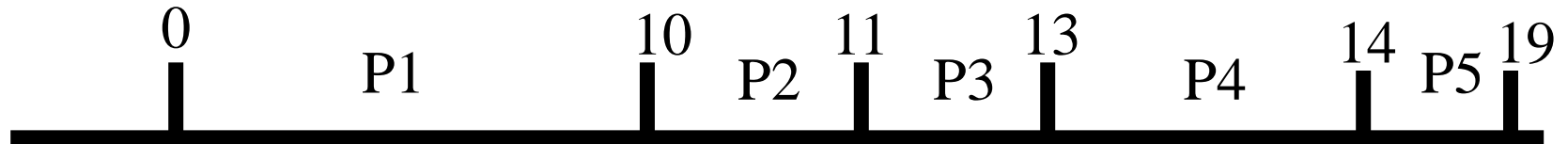


$$\text{Average WT} := (0 + 1 + 6 + 16 + 18) / 5 = 8.2 \text{ ms}$$

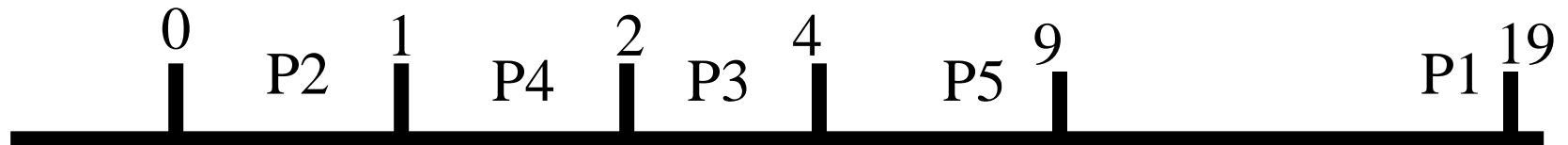
P1 and P3 have same priority → FCFS

**Low number means high priority**

# Priority Scheduling - Comparison



Average WT :=  $(0+10 + 11 + 13 + 14)/5 = 9.6$  ms FCFS



Average WT :=  $(0+1 + 2 + 4 + 9)/5 = 3.2$  ms SJF

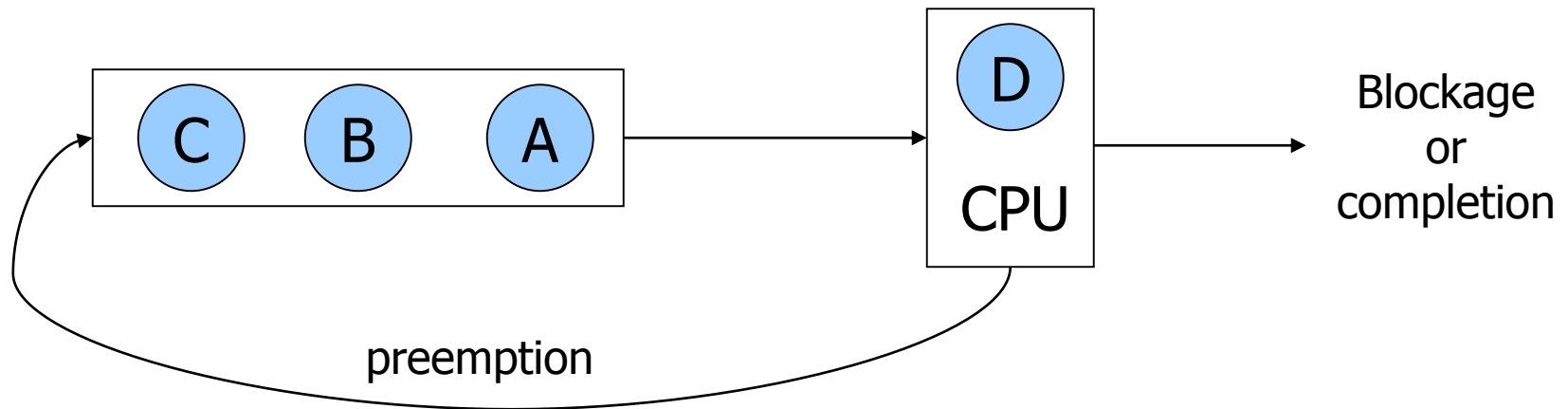
No preemption

# Priority Scheduling

- Low priority processes may end up waiting forever
  - ➔ Indefinite postponement or **starvation** a problem.
- Solution?  
Aging...
  - As the process ages, increase its priority over time.
  - Eventually, it has highest priority and gets scheduled.

# Round Robin (RR)

- Used in interactive systems.
- Goal: minimize response time.
- Each process gets a small unit of CPU time (**time quantum or time slice**), usually 10-100 milliseconds. After this time has elapsed, the process is preempted and added to the end of the ready queue.



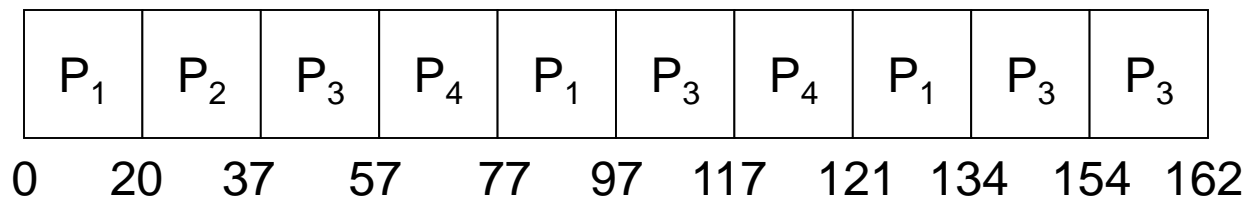
# Round Robin (RR)

- If there are  $n$  processes in the ready queue and the time quantum is  $q$ , then each process gets  $1/n$  of the CPU time in chunks of at most  $q$  time units at once.
- No process waits more than  $(n - 1)q$  time units.
- Short CPU burst processes get their chance and exit the queue quickly when they are done. (interactive tasks such as typing a character)
- Long CPU burst processes turnaround time usually becomes longer.

## Example: RR with Time Quantum $q = 20$

<u>Process</u>	<u>Burst Time</u>
$P_1$	53
$P_2$	17
$P_3$	68
$P_4$	24

- The Gantt chart is:



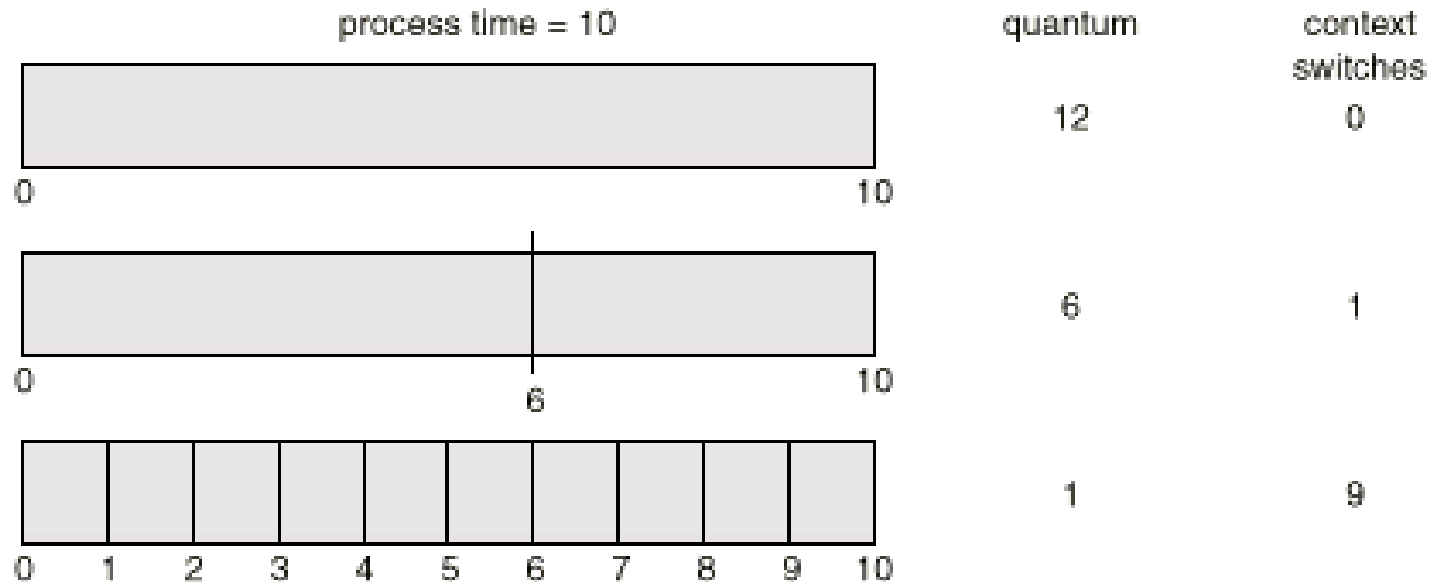
- Typically, higher average turnaround than SJF, but better *response*.

# Round Robin (RR)

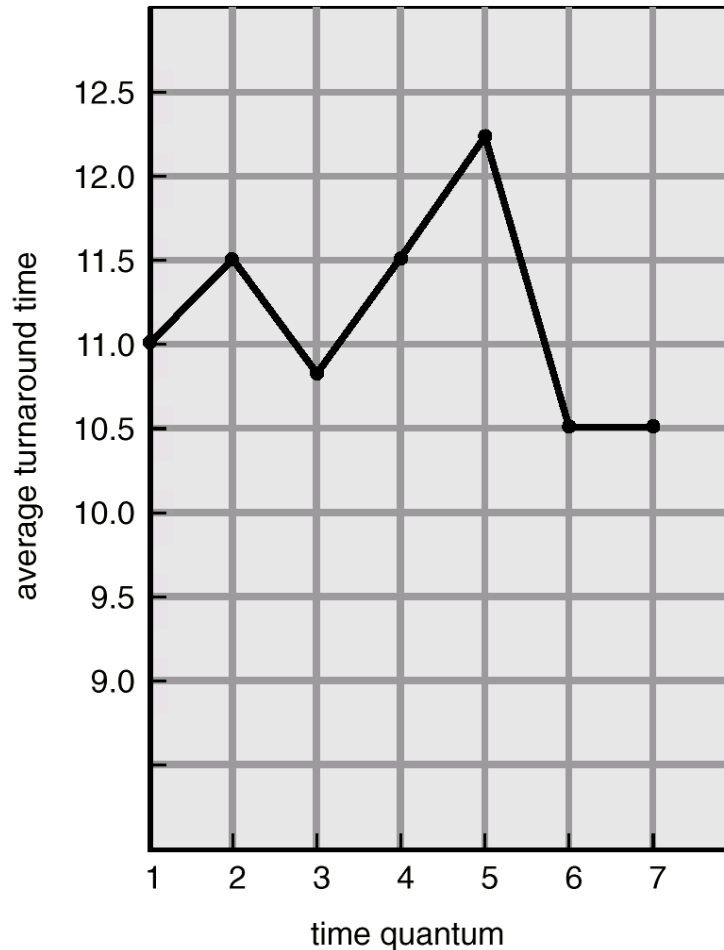
- Performance
  - $q$  large  $\Rightarrow$  FIFO
  - $q$  small  $\Rightarrow$   $q$  must be large with respect to context switch, otherwise overhead is too high.



# How a Smaller Time Quantum Increases Context Switches



# Turnaround Time Varies With The Time Quantum



process	time
$P_1$	6
$P_2$	3
$P_3$	1
$P_4$	7

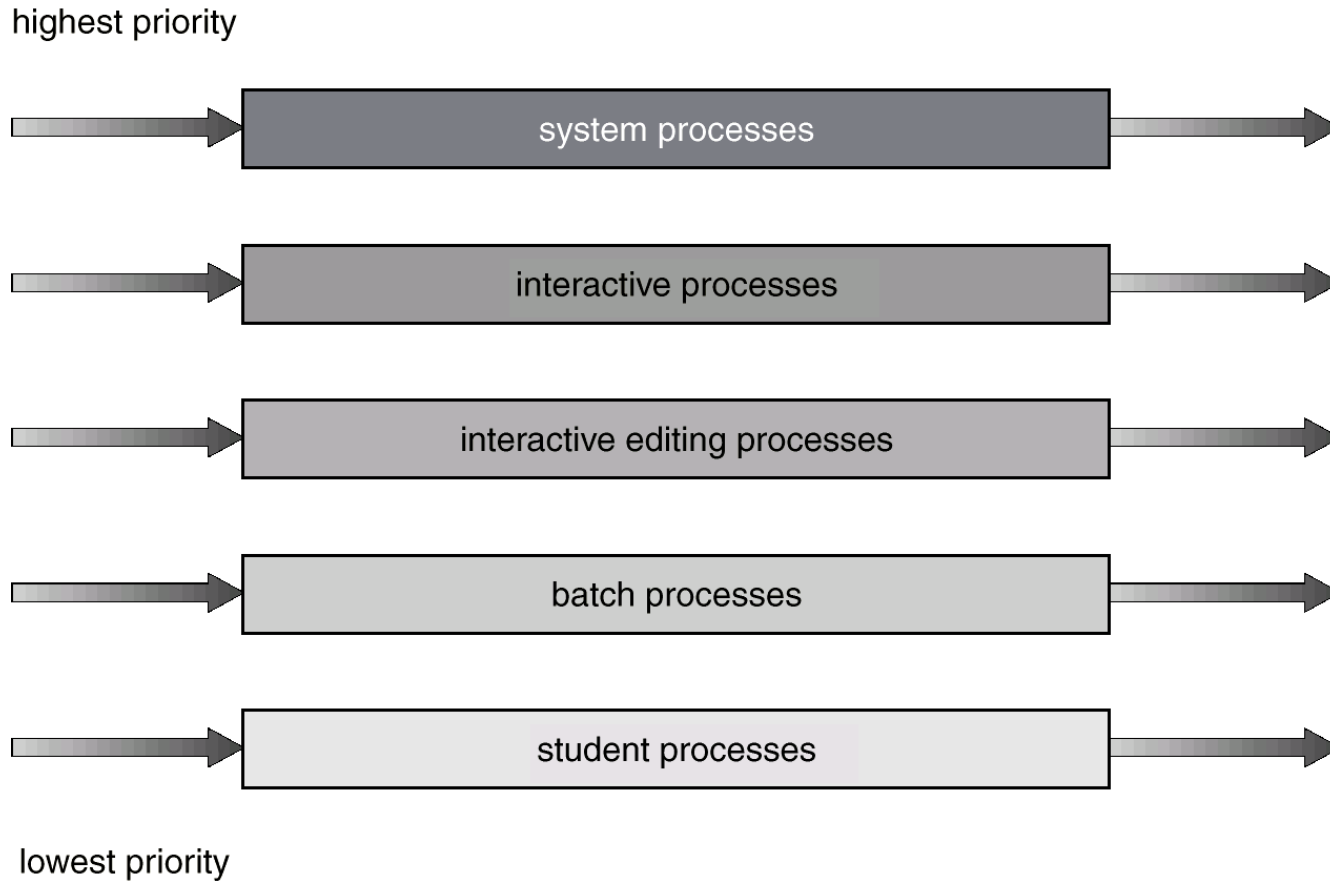
# Discussion

- Processes change behavior
  - E.g., foreground vs. background
- How can we adapt scheduling?
  - Have multiple queues

# Multilevel Queue

- Ready queue is partitioned into separate queues:  
foreground (interactive) background (batch)
- Each queue has its own scheduling algorithm,  
foreground – RR  
background – FCFS
- **Processes stay in the queue they enter**
- Scheduling must be done between the queues.
  - Fixed priority scheduling; i.e., serve all from foreground then from background. Possibility of starvation.
  - Time slice – each queue gets a certain amount of CPU time which it can schedule amongst its processes; i.e.,  
80% to foreground in RR  
20% to background in FCFS

# Multilevel Queue Scheduling

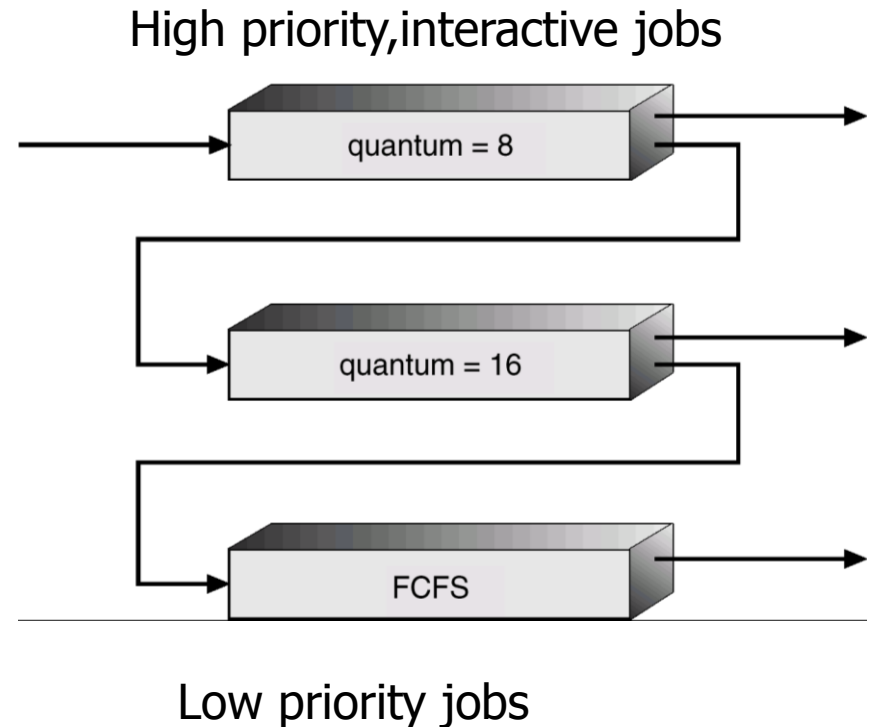


# Multilevel Feedback Queue

- A process can move (up and down) between the various queues; aging can be implemented this way.
- Multilevel-feedback-queue scheduler defined by the following parameters:
  - number of queues
  - scheduling algorithms for each queue
  - method used to determine when to upgrade a process
  - method used to determine when to demote a process
  - method used to determine which queue a process will enter when that process needs service

# Multi-level Feedback Queues

- Each queue represents jobs with similar CPU usage.
- Requirement to assign to each queue permanently is relaxed.  
→ processes can migrate to other queues.
- Processes using too much CPU move to lower priority queues.
- Processes that age move to higher priority queues.



# Multi-level Feedback Queues

- Example:  $queue_i$  has time-slice  $t \times 2^i$
- If job in  $queue_i$  doesn't block by end of time-slice it is moved to  $queue_{i+1}$
- Lowest priority queue is FIFO
- **Starvation**: aging may move process to higher priority queue



# Lottery Scheduling

- Each process/thread given a lottery ticket.
- When scheduler wants to decide who to schedule, pick one ticket at random.
  - The process holding that ticket number gets the CPU for a fixed amount of time (e.g., 20ms).

# Lottery Scheduling

- If everyone gets the same number of tickets, then everyone has fair share.
  - 5 processes, each holding 1 ticket: each has a 20% chance of getting scheduled.
- One can increase the odds of a thread getting scheduled (e.g., because it is high priority) by giving more tickets to higher priority processes.
  - 1 process holds 2 tickets, and 4 processes hold 1 ticket each, then
    - Process holding 2 tickets has  $2/6=33\%$  chance of getting scheduled.
    - Other 4 processes each have  $1/6=16.7\%$  chance of getting scheduled.
- Also, newly arriving thread after given a ticket, immediately has a chance of getting scheduled.

# Fair share scheduling

- Instead of fair share of CPU time for each process/thread, it tries to allocate fair share of time for each user.
- Works out better if there are imbalances in the number of processes/user.
- Example: User A has processes A, B, C and user B has one process D,
- Allocate 50% CPU time per user. A possible schedule would be:
  - A, D, B, D, C, D, A, D, ...
- instead of
  - A, B, C, D, A, B, C, D, ...
  - in which case, user A would have gotten 75% of the CPU time.

# Multiple-processor Scheduling

- Homogeneous processors permit load sharing
- To keep all CPUs busy, have one ready queue accessed by each CPU
- Self-scheduled -- each CPU dispatches a job from the ready queue
  - Critical section problems in accessing common queue: shared data structure maintenance problems and synchronization).
- Master-slave -- one CPU schedules the other CPUs.  
extension of M/S results in → Asymmetric scheduling
- Asymmetric -- one CPU runs the kernel and the others the user applications
  - System call interface: send a message to kernel CPU.

# Discussion

- On a parallel processor, its inefficient to have one process using a CPU to wait for the results from another process on a CPU.
- How do we reduce the waiting time in parallel processing?
  - Small number of processors (2-3) can share multi-level queue.
  - Larger number of processors (50-100)
    - Context switch is more expensive
    - Message passing to all the processes (if there is a delay, then all processes can be delayed).

# Gang Scheduling

- A collection of processes belonging to one job
  - Not clear if multiple processors can execute large number of processes efficiently.
- All the processes are run at the same time. If one process is preempted, all the processes of the gang are preempted
  - Implies a M/S relationship with some other processors.

# Gang Scheduling

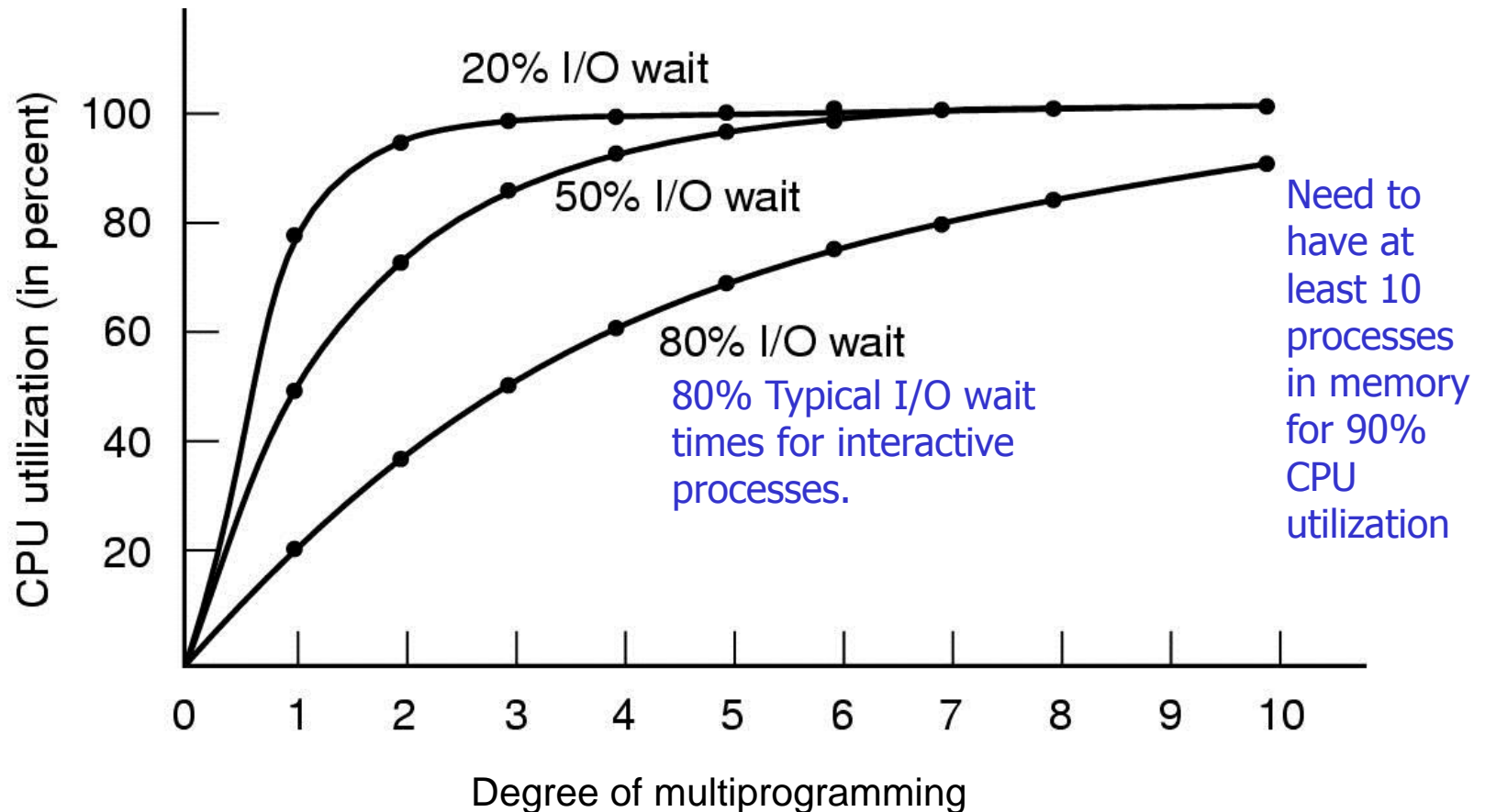
- Helps to eliminate the time a process spends waiting for other processes in its parallel computation
  - Especially if other processes will eventually wait for preempted process.
  - Reduces extra overhead for individual management of processes.

# Modeling Multiprogramming

- Suppose a process spends  $p$  fraction of its time waiting for I/O to complete.
- With  $n$  processes in memory, all  $n$  of them waiting for I/O is  $p^n$ .
  - If all  $n$  processes are waiting for I/O simultaneously, then CPU would be idle. Therefore, the probability that CPU would be idle with  $n$  processes is  $p^n$ .
  - And the probability that the CPU will be doing useful computation at any given moment would be  $1 - p^n$ .
- Thus, CPU utilization =  $1 - p^n$ .



# Modeling Multiprogramming



- CPU utilization as a function of number of processes in memory

# Discussion

- In all the previous scheduling schemes, the scheduler is completely unaware of the wall clock time? How do we schedule to meet real-time situations then?
  - Use **real-time scheduling**.

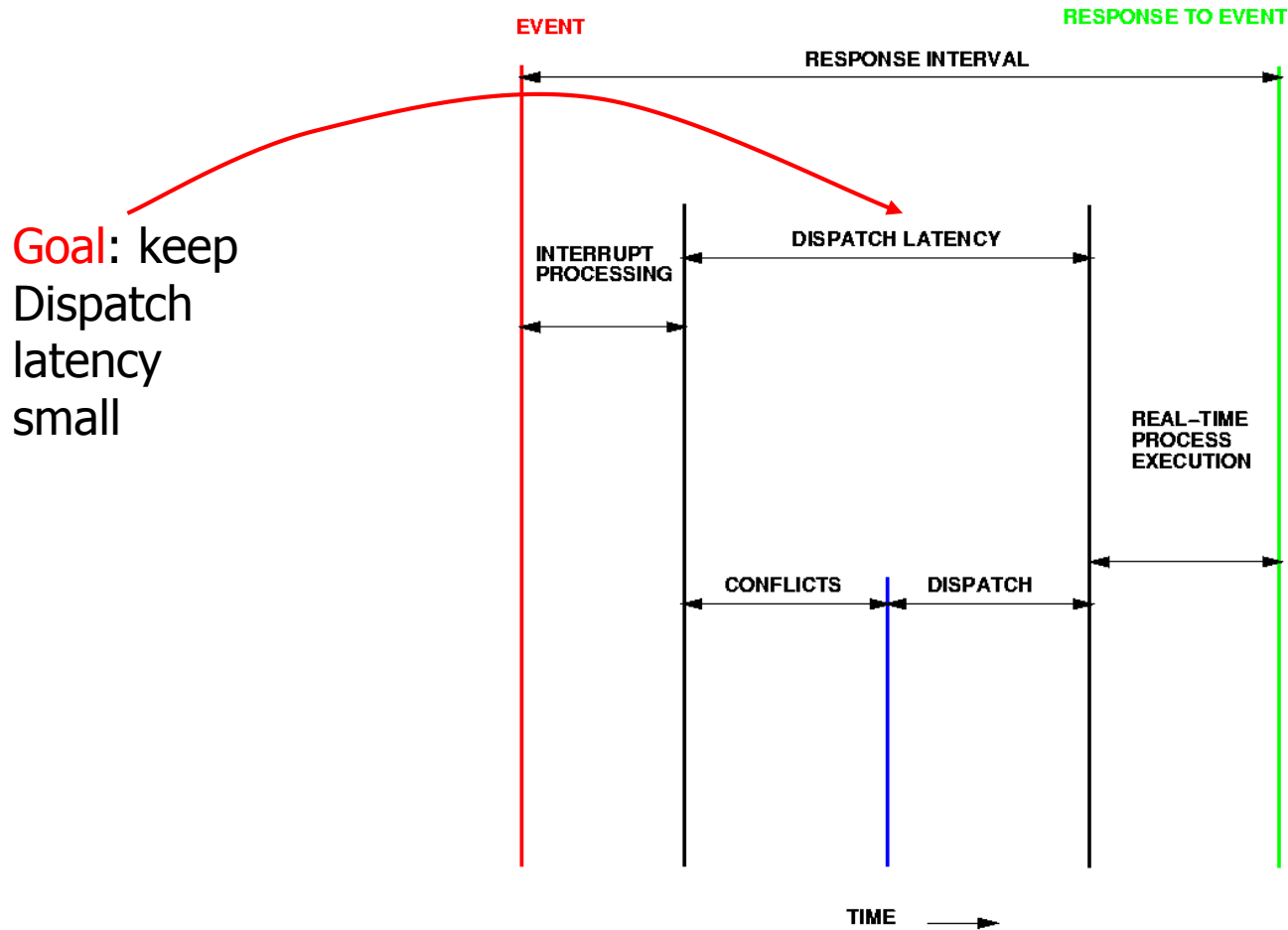
# Real-time Scheduling

- Lots of solutions possible.
- These depend on type of applications.
- Different types of real-time scheduling
  - **Hard real-time**: produce output (or complete processing) within set time bounds.
    - Examples: flight control systems, etc.
  - **Soft real-time**: try to do things as fast as possible, at interactive rates.
    - Examples: interactive systems, video compression, etc.

# Real-time Scheduling

- Periodic schedulers
  - Tasks repeat at periodic rates.
- Demand-driven schedulers
  - Interrupt-driven
- Deadline schedulers
  - Deadlines by which tasks need to be completed.

# Latency in Dispatching



# Dispatch Latency

- **Problem:** keep dispatch latency small. OS may enforce process to wait either for a system call to complete or for an I/O block to take place
  - This adds to dispatch latency

# Dispatch Latency

- **Solution:** need preemptable system calls
- Insert preemption points (can be placed at safe location where kernel structures are not modified)
- Make the kernel preemptable (all kernel structures must be protected through the use of various **synchronization** mechanisms)  
Example: Solaris2

# Priority Inversion and Inheritance

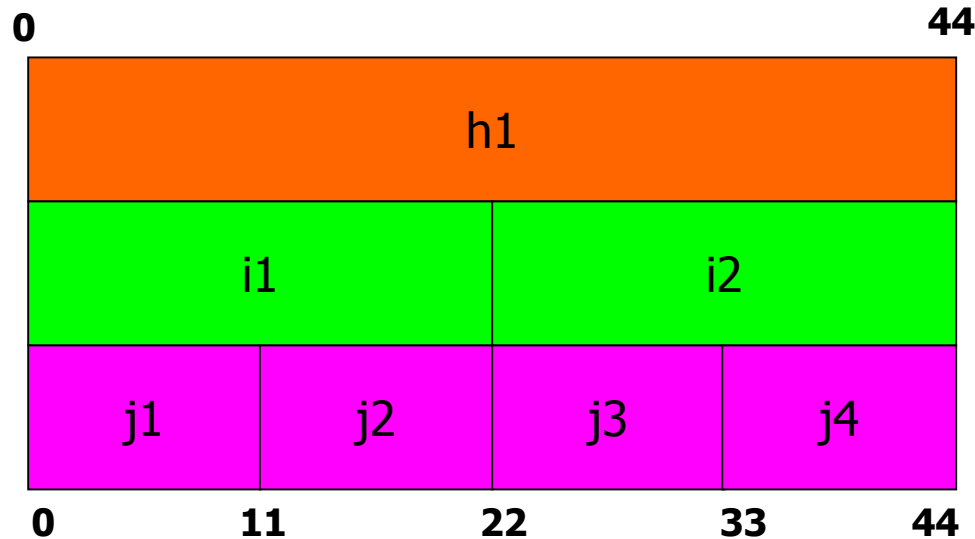
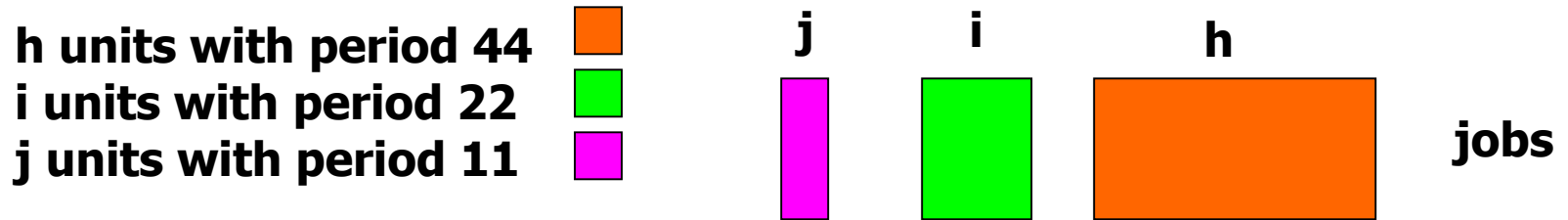
- Problem:
  - A higher priority process needs to read or modify kernel data
  - The data is being accessed by another, lower priority process
    - E.g., lower priority process has a lock
  - The result is that the higher priority process waits! → thus, **priority inversion**.



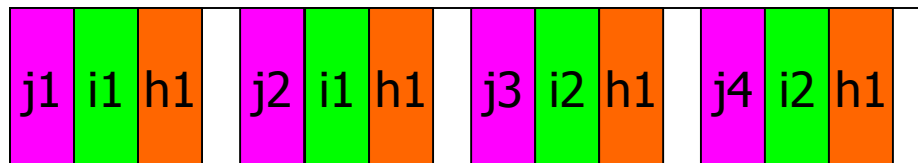
# Priority Inversion and Inheritance

- Solution: priority inheritance
- Goal is to have the lower priority processes do their task and get out of the way ASAP.
  - All processes accessing a resource needed by a high-priority process inherit high-priority until finished with the resource
  - When complete, their priority reverses to its natural value

# Periodic Scheduler



Periods and levels



Gant chart

# Scheduling in Real-Time Systems

- Question: What are the conditions under which periodic processes can be scheduled?

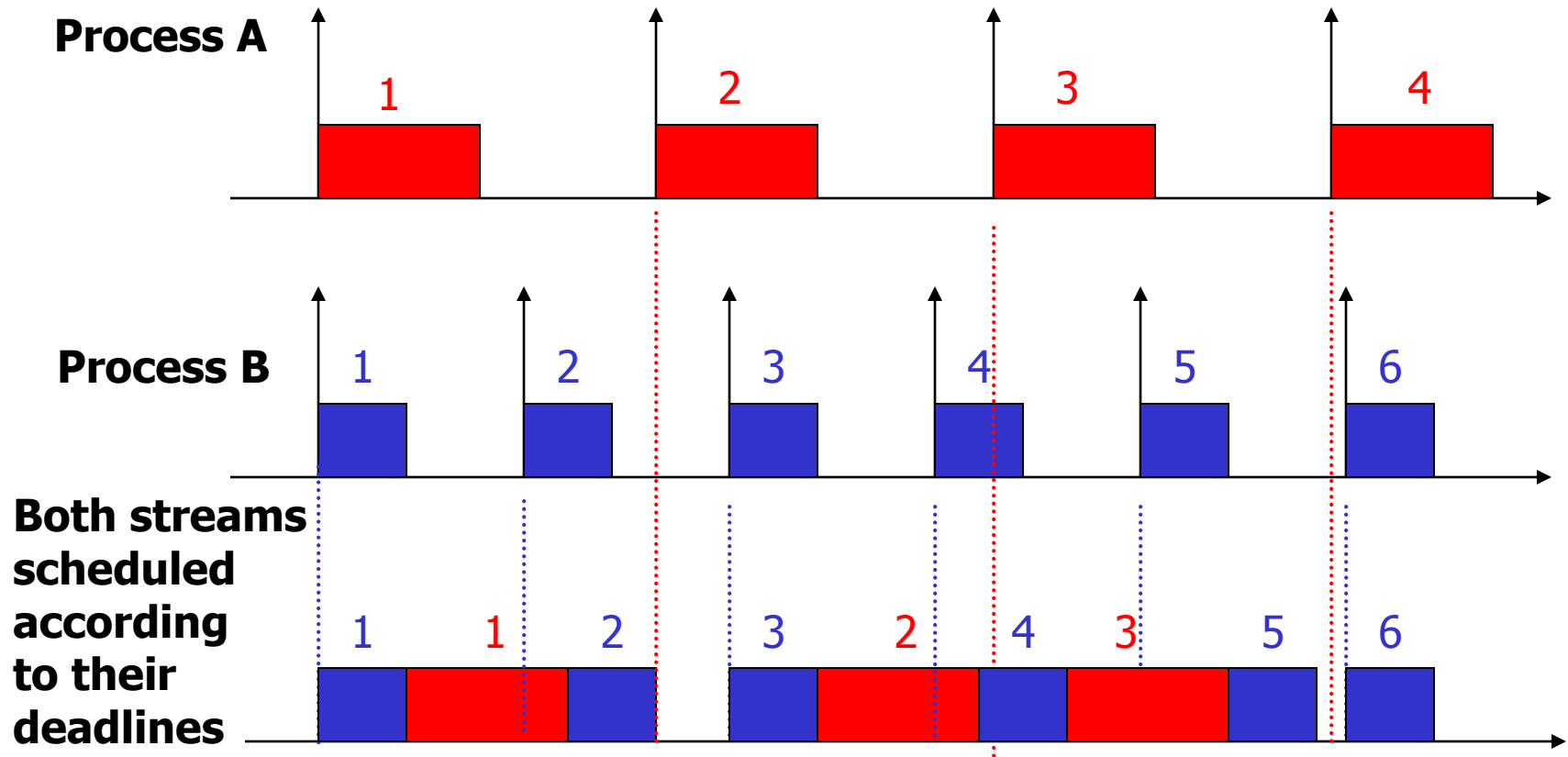
Schedulable real-time system

- Given
  - $m$  periodic events
  - event  $i$  occurs within period  $P_i$  and requires  $C_i$  seconds of processing
- Then the load can only be handled if

$$\sum_{i=1}^m \frac{C_i}{P_i} \leq 1$$

# Deadline Scheduling

Process priority determined by process deadline



# Deadline Scheduling

- **Result by Liu and Layland (1973):** if any schedule will work, earliest deadline first will schedule tasks
- Citation: “Scheduling Algorithms for Multiprogramming in a Hard-Real-Time Environment,” C. L. Liu and James W. Layland, Journal of the ACM (JACM), v. 20, no. 1, pp. 46-61, 1973.

# Summary

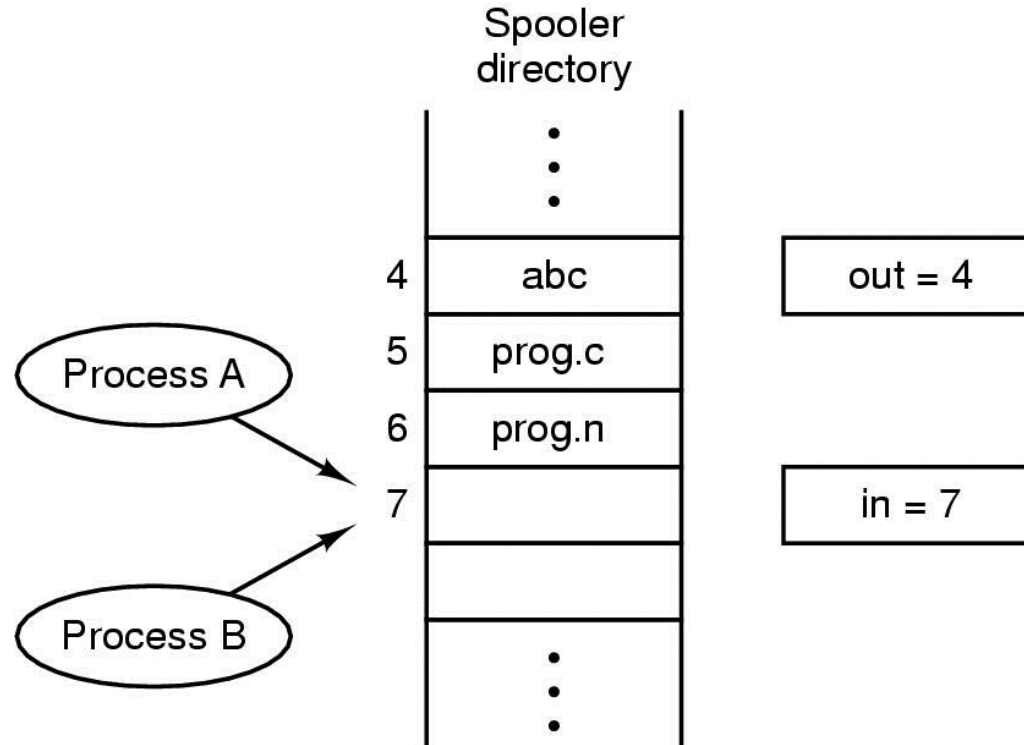
- Many systems use multi-queue or multi-level feed back queues
- Average response time and variance important
- Can prioritize jobs according to various attributes
- Preemption helps fast response, but may cause starvation, missed deadlines or increased overhead

# Process/thread Synchronization

An introduction

# Interprocess Communication

## Race Conditions



- Two processes want to access shared memory at same time



# Cooperating Processes

- Cooperating processes may be communicating with each other via **shared variables or resources**.
- The actual execution of the instructions of a thread may become interleaved with other instructions of other threads due to preemptive scheduling.
- In a multiprocessor system, the concurrent threads may actually each execute on an independent processor.

# High level language code

- Shared variable `x`

Thread 1	Thread 2
<code>x++</code>	<code>x++</code>

- Concurrent threads, operating on shared variable `x`.
- At high level language: we want increments to be atomic operations
- When both threads are done
  - we expect `x` to be incremented by 2 regardless of order in which increment is performed.

# Low level translated code

- `x++` could be implemented as

```
register1 ← x
register1 ← register1 + 1
x ← register1
```
- It takes 3 low level instructions to accomplish the increment.
- If the increment is not done atomically, computations can end up wrong.

# Interference and Shared Variables

Strictly sequential	<b>Process 1</b>	<b>Register R</b>	<b>Process 2</b>	<b>Register R</b>	<b>M</b>
	Load M into R	1	Noop	0	1
	Add 1 to R	2	Noop	0	1
	Store R in M	2	Noop	0	2
	Noop	2	Load M into R	2	2
	Noop	2	Add 1 to R	3	2
interleaved	<b>Process 1</b>	<b>Register R</b>	<b>Process 2</b>	<b>Register R</b>	<b>M</b>
	Load M into R	1	Noop	0	1
	Noop	1	Load M into R	1	1
	Add 1 to R	2	Noop	1	1
	Store R in M	2	Noop	1	2
	Noop	2	Add 1 to R	2	2
	Noop	2	Store R in M	2	2

# Producer-Consumer Problem

- Concurrent execution that requires cooperation among processes needs mechanisms to allow:
  - communication with each other and synchronization of their actions.
- Producer-Consumer problem is a paradigm for cooperating processes (and an example you will come across often).

# Producer-Consumer Problem

- For concurrent processing we need a **buffer** of items that can be filled by the producer and emptied by the consumer.
- Communication occurs through shared memory.
- **buffer** is the shared data between the two processes/threads.

# Producer-Consumer Problem

- Buffer can be of finite size or infinite size.
- Unbounded-buffer producer-consumer problem - places no practical limits on buffer size, The consumer may wait for a new item (when buffer is empty), but producer never waits (there is always space in the buffer).
- Bounded-buffer producer-consumer problem - assumes fixed buffer size. The consumer must wait for a new item when buffer is empty and the producer must wait when the buffer is full.

# Background

- A solution to shared-memory producer-consumer problem is not simple.
  - Let us create a variable *counter*, initialized to 0 and incremented when a new item is added and decremented when an item is consumed.
  - Let us create two pointers into buffer *in* and *out* where the produced item is put and from which consumed item is taken, respectively.



# Bounded-Buffer

- Shared data

```
#define BUFFER_SIZE N
typedef struct {
    . . ./* whatever information is produced/consumed */
} item;
item buffer[BUFFER_SIZE];
int in = 0; /* pointer to where next item produced goes */
int out = 0; /* pointer from which next item is consumed */
int counter = 0; /* number of items in the buffer */
/* counter == 0 means buffer is empty
   counter == BUFFER_SIZE means buffer is full */
```

# Bounded-Buffer

- Producer process

```
item nextProduced;
```

```
while (TRUE) {
```

```
    while (counter == BUFFER_SIZE)
```

```
        {};
```

```
    buffer[in] = nextProduced;
```

```
    in = (in + 1) % BUFFER_SIZE; /* wrap around */
```

```
    counter++;
```

```
}
```

This is called a “busy wait”  
or “spinlock”

C.S.

there's our shared variable: counter

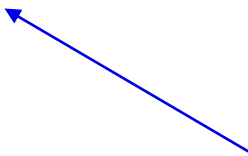
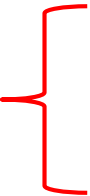
# Bounded-Buffer

- Consumer process

```
item nextConsumed;
```

```
while (TRUE) {  
    while (counter == 0)  
        {}; /* do nothing; buffer is empty */  
    nextConsumed = buffer[out];  
    out = (out + 1) % BUFFER_SIZE;  
    counter--;  
}
```

C.S.



there's our shared variable: counter

# Bounded Buffer

- **counter** is shared variable between producer and consumer → both of them modify it.
- **counter** counts the number of full buffer locations
- The statements  
`counter++;`  
`counter--;`  
must be performed *atomically*.
- Atomic operation means an operation that completes in its entirety without interruption (or if interrupted other processes/threads cannot operate on the shared variable).

# Bounded Buffer

- The statement “**counter++**” may be implemented in machine language as:

```
register1 = counter  
register1 = register1 + 1  
counter = register1
```

- The statement “**counter--**” may be implemented as:

```
register2 = counter  
register2 = register2 - 1  
counter = register2
```

# Bounded Buffer

- If both the producer and consumer attempt to update the buffer concurrently, the assembly language statements may get interleaved.
- Similar situation to previous example.
- Interleaving depends upon how the producer and consumer processes are scheduled.

# Bounded Buffer

- Consider this execution interleaving with “counter = 5” initially:

S0: producer execute	<code>register1 = counter</code>	{register1 = 5}
S1: producer execute	<code>register1 = register1 + 1</code>	{register1 = 6}
S2: consumer execute	<code>register2 = counter</code>	{register2 = 5}
S3: consumer execute	<code>register2 = register2 - 1</code>	{register2 = 4}
S4: producer execute	<code>counter = register1</code>	{counter = 6}
S5: consumer execute	<code>counter = register2</code>	{counter = 4}
- The correct result should be count = 5 after one atomic increment and one atomic decrement.
  - But we end up with counter = 4 instead.

# Race Condition

- **Race condition**: The situation where several processes access – and manipulate **shared data concurrently**. The final value of the shared data depends upon which process finishes last.
- To prevent race conditions, concurrent processes must be **synchronized**.



# Critical Section Problem

- Consider system of  $n$  processes  $\{p_0, p_1, \dots, p_{n-1}\}$
- Each process has a **critical section** segment of code
  - Process may be changing common variables, updating table, writing file, etc.
  - When one process in critical section, no other may be in its critical section
- **Critical section problem** is to design a protocol to solve this
- Each process must ask permission to enter critical section in **entry section**, may follow critical section with **exit section**, then **remainder section**

# Critical Section Problems

- General structure of a C.S. process is:

do {

ENTER CRITICAL SECTION

critical section

LEAVE CRITICAL SECTION

remainder section

} while (TRUE);

This could be  
a very complex  
operation:  
e.g., updating a  
database.

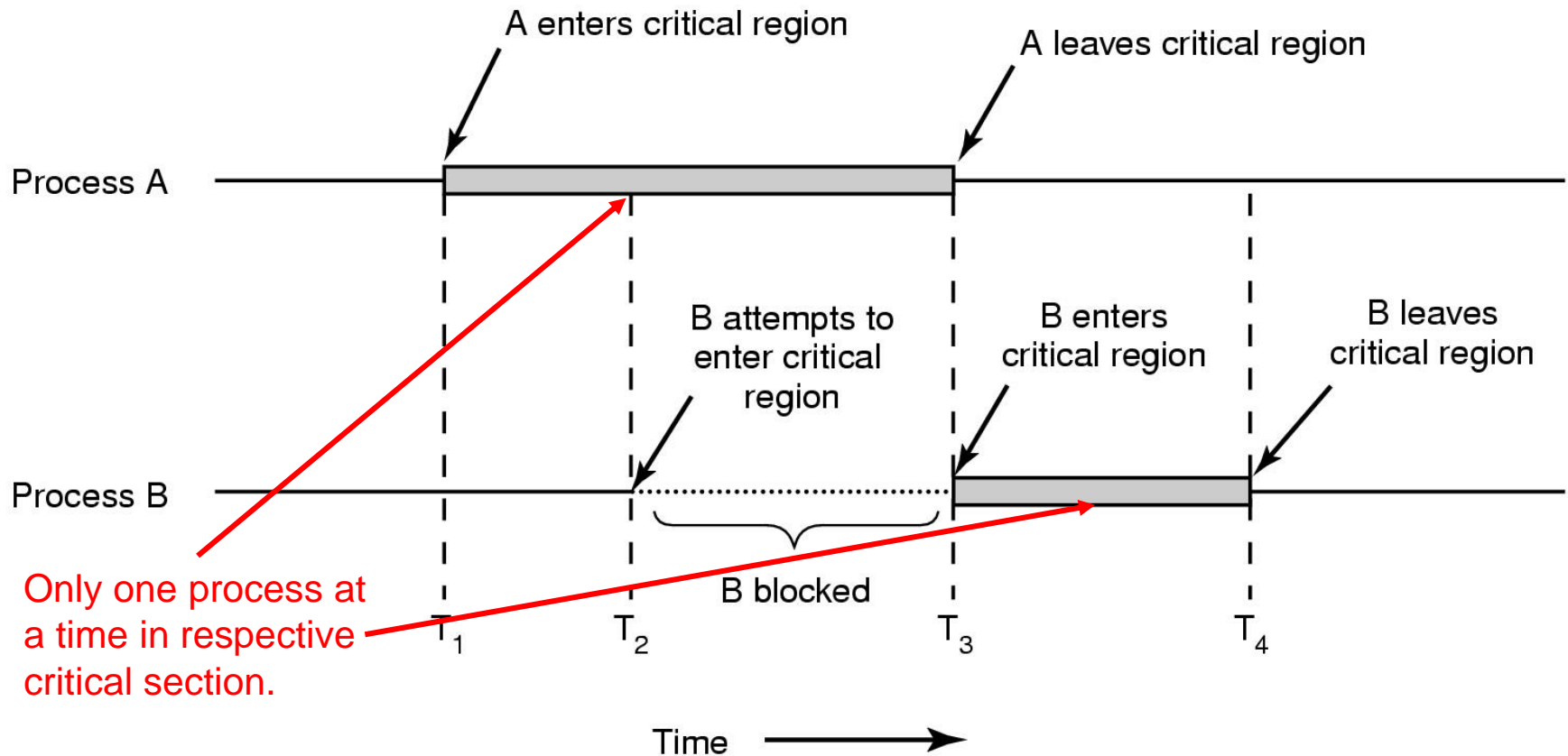
any other computation

- Processes may share some common variables to synchronize their actions.

# Critical Section Problem

- Problem – How to ensure *Mutual Exclusion* in critical section?
- That's when one process is executing in its critical section, no other process is allowed to execute in its critical section? That is, how to synchronize?

# Critical Sections



Mutual exclusion using critical regions

# Mutual exclusion

- Assumption: In normal situations (i.e., computation at the machine instruction level), the Read/Write cycle on a bus (at the HW level) provides the mutual exclusion.
  - When doing write, the bus will prevent any other CPU from accessing memory (bus arbitration unit).
  - This assumption is used for the solution to critical section problem as we will see.

# Properties of a Correct Solution to Critical-Section Problem

1. **Mutual Exclusion.** If process  $P_i$  is executing in its critical section, then no other processes can be executing in their corresponding critical sections.
2. **Progress.** If no process is executing in its critical section and there exist some processes that wish to enter their critical section, then the selection of the processes that will enter the critical section next **cannot be postponed indefinitely**.
  - Only processes that are not executing in the “remainder sections” (i.e., doing other work) can participate in deciding who gets into critical sections.

# Properties of a Correct Solution to Critical-Section Problem

3. **Bounded Waiting.** A bound must exist on the number of times that other processes are allowed to enter their critical sections after a process has made a request to enter its critical section and before that request is granted.

- ☐ Guarantee that a process will enter C.S. within a bounded amount of time.
- ☐ Assume that each process executes at a nonzero speed.
- ☐ No assumption concerning relative speed of the processes.

# Initial Attempts to Solve Problem

Start with *software solutions* to synchronizing only *two processes*,  $P_0$  and  $P_1$

History of attempts to solve the problem

- Approach 1: Turn Mutual Exclusion
- Approach 2: Other Flag Mutual Exclusion
- Approach 3: Two Flag Mutual Exclusion
- Approach 4: Two Flag and Turn Mutual Exclusion (aka *Peterson's solution*)
- In all examples to follow for these approaches, we have *two processes*:  $P_0$  and  $P_1$ .



# Approach 1: Turn Mutual Exclusion

- Shared variables:
  - **int turn;**  
initially **turn = 0**
  - **turn = i**  $\Rightarrow P_i$  can enter its critical section
- Process  $P_i$ 
  - do {**
    - while (turn != i) { /\* no-op \*/; // busy wait**  
critical section
    - turn = j;**  
remainder section
  - } while (true);**
- Satisfies mutual exclusion, but not progress
  - $P_0$  and  $P_1$  take turns. If speed difference between two processes is large, this causes unnecessary waiting.

# Approach 2: Other Flag Mutual Exclusion

- Shared variables
  - **boolean flag[2];**  
initially **flag [0] = flag [1] = false.**
  - **flag[i] = true**  $\Rightarrow P_i$  ready to enter its critical section
- Process  $P_i$ 
  - do {**
    - while (flag[j]) { /\* noop \*/; // busy wait**
    - flag[i] = true;**  
critical section
    - flag [i] = false;**  
remainder section
  - } while (true);**
- Does not satisfy mutual exclusion.
  - Initially both flags can be false, and both processes will proceed into their critical regions. By the time a process sets flag to true it would be too late.

# Approach 3: Two Flag Mutual Exclusion

- Shared variables
  - **boolean flag[2];**  
initially **flag [0] = flag [1] = false.**
  - **flag[i] = true**  $\Rightarrow P_i$  ready to enter its critical section
- Same as other flag M.E., but flag set before entering the busy wait while loop.
- Process  $P_i$ 
  - do {**
    - flag[i] = true;**
    - while (flag[j]) { /\* no-op \*/; // busy wait**  
critical section
    - flag [i] = false;**  
remainder section
  - } while (true);**
- Satisfies mutual exclusion, but not progress requirement.
  - Can block indefinitely:  
They can both set the flags to true before the loop and wait on each other indefinitely.

## Approach 4: Combine turn and two flag approaches (Peterson's solution)

- Combined shared variables of algorithms 1 and 3.
- Process  $P_i$ 

```
do {  
    flag [i]= true;  
    turn = j;  
    while (flag[j] and turn == j) { /* no-op */; // busy wait  
        critical section  
    flag [i] = false;  
    remainder section  
} while (true);
```
- Meets all three requirements; solves the critical-section problem for **two processes**.
- In case there is a race condition (both set the flag = true at the same time), turn breaks the tie.

# Bounded Buffer Producer-Consumer Solution using Petersen's solution

- Producer process

```
item nextProduced;
int turn; // initialize to 0
Boolean flag[2]; // initialize to false
#define producer 0
#define consumer 1

while (TRUE) {
    while (counter == BUFFER_SIZE); /* do nothing; buffer is full */
    flag [producer]= true; // The 3 statements in blue
    turn = consumer; // are ENTER CRITICAL SECTION
    while (flag[consumer] and turn == consumer); // busy wait
    buffer[in] = nextProduced;
    in = (in + 1) % BUFFER_SIZE; /* wrap around */
    counter++; // this is the critical section
    flag [producer] = false; // exit critical section
}
```

# Bounded Buffer Producer-Consumer Solution using Petersen's solution

- Consumer process

```
item nextConsumed;
int turn;
Boolean flag[2];
#define producer 0
#define consumer 1

while (TRUE) {
    while (counter == 0); /* do nothing; buffer is full */
    flag [consumer]= true; // The 3 statements in blue
    turn = producer; // are ENTER CRITICAL SECTION
    while (flag[producer] and turn == producer); // busy wait
    nextConsumed = buffer[out];
    out = (out + 1) % BUFFER_SIZE; /* wrap around */
    counter--; // this is the critical section
    flag [consumer] = false; // exit critical section
}
```

# n-process synchronization

- There is a solution called “baker’s algorithm”
- We won’t go into details.
- The solution relies on each process getting “tickets” as in a bakery and using this “ticket number” to order who accesses critical section in what order, but still **one-at-a-time**.

# Hardware Solutions

- The earlier mutual exclusion examples depend on the memory hardware having an **atomic write**. If multiple reads and writes could occur to the same memory location at the same time, they would not work.
- Processors with caches but no cache coherency cannot use the solutions provided.



# Hardware Solutions

- In general, it is IMPOSSIBLE to build mutual exclusion without a primitive that provides some form of mutual exclusion *at the hardware level*.

# Synchronization Hardware

- Thus, many systems provide hardware support for implementing the critical section code.
- All solutions we will see are based on idea of **locking**
  - Protecting critical regions via locks
- Uniprocessors – could disable interrupts
  - Currently running code would execute without preemption
  - Generally too inefficient on multiprocessor systems
    - Operating systems using this not broadly scalable

# Solution to Critical-section Problem Using Locks

- General structure of the critical section will look as follows:

```
do {  
    acquire lock  
        critical section  
    release lock  
        remainder section  
} while (TRUE);
```

# Hardware Solutions

- Modern machines provide special atomic hardware instructions
  - **Atomic** = non-interruptible
- Two solutions (and their related derivatives) are popular:
  - Either test memory word and set value
    - Test-and-set (e.g., Motorola 68000 TAS instruction)
  - Or swap contents of two memory words
    - Swap (e.g., Pentium XCHG instruction)
  - In general, *not trivial* to implement in *multi-processor* architectures. ➔ Beyond the scope of this class.

# Test-and-Set

- Hardware provides a test-and-set instruction that is guaranteed by hardware to be atomic and has the following semantics:

Example Machine instruction	Semantics (meaning) in C syntax
TAS in 68000	<pre>boolean test_and_set (boolean *target) { // all done atomically   boolean rv = *target;   *target = TRUE;   return rv; }</pre>

1. Executed atomically
2. Returns the original value of passed parameter
3. Set the new value of passed parameter to “TRUE”.

# Solution using test\_and\_set()

- Shared Boolean variable lock, initialized to FALSE
- Solution:

```
do {  
    while (test_and_set(&lock)) // spinlock  
        ; /* do nothing */      // trying to acquire lock  
    /* critical section */  
    lock = false;                // release lock  
    /* remainder section */  
  
} while (true);
```

# Swap

- An alternative instruction is swap, provided by hardware that works atomically in hardware.

Example Machine instruction	Semantics (meaning) in C syntax
XCHG in Pentium	<pre>void swap(byte *x, *y); // All done atomically {     byte temp = *x;     *x = *y;     *y = temp; }</pre>

# Variations on the theme compare\_and\_swap Instruction

Definition:

```
int compare_and_swap(int *value, int expected, int new_value)
{
    int temp = *value;

    if (*value == expected)
        *value = new_value;
    return temp;
}
```

1. Executed atomically
2. Returns the original value of passed parameter “value”
3. Set the variable “value” the value of the passed parameter “new\_value” but only if “value” == “expected”. That is, the swap takes place only under this condition.



# Solution using compare\_and\_swap

- Shared integer “lock” initialized to 0;
- Solution:

```
do {  
    while (compare_and_swap(&lock, 0, 1) != 0)  
        ; /* do nothing */  
    /* critical section */  
    lock = 0; // release lock  
    /* remainder section */  
} while (true);
```

# Mutex Locks

- Previous solutions are complicated and generally inaccessible to application programmers
- OS designers build software tools to solve critical section problem
- Simplest is **mutex** lock
- Protect a critical section by first **acquire()** a lock then **release()** the lock
  - Boolean variable indicating if lock is available or not
- Calls to **acquire()** and **release()** must be atomic
  - Usually implemented on top of hardware atomic instructions

# Usage of hardware primitives

- Example implementation
- Example: “lock” C++ class using test-and-set.
  - Provides mutual exclusion.

# A Lock Class that uses TestAndSet

## Lock class declaration:

```
class Lock {  
    byte state; // internal state var  
    public:  
        Lock();  
        void acquire();  
        void release();  
        int isFree();  
};
```

# A Lock Class that uses TestAndSet

## Example usage of Lock class:

```
Lock a;  
a.acquire(); // acquire the lock  
// Do Critical Section using lock a  
a.release(); // release the lock
```

# Lock Class Implementation using TestAndSet()

```
inline Lock::Lock() { state = 0; }
inline void Lock::acquire()
    { while (test_and_set(state)); /* spin-lock */ }
inline void Lock::release() { state = 0; }
inline int Lock::isFree()
    { return (state == 0); }
```



This test is atomic!

# Busy wait disadvantages

- Previous solutions to synchronization problems we saw (e.g., bounded-buffer producer-consumer problem) used “**busy-wait**” to implement the waiting on a condition: while loop waiting on some condition to change.
- This is wasteful, because the while loop needs to execute on the CPU and, therefore, wastes precious processor cycles, doing nothing useful.
  - There are cases where busy-wait is better: **if the wait is expected to be short**
  - Tradeoff in this case: overhead of context-switch going through kernel vs. wasting CPU cycles.

# Alternatives to busy waiting

- One possible solution: `sleep()` and `wakeup()` system calls.
- Process/thread executing `sleep()` would be suspended and be put in “blocked” state until some other process/thread wakes it up.
- `wakeup(pid)` call wakes up process/thread with `pid`: meaning it is put in ready state and put in the ready queue to be scheduled for the CPU.
- Problem: it can cause race conditions.



# Sleep and Wakeup

```
#define N 100                                /* number of slots in the buffer */
int count = 0;                               /* number of items in the buffer */

void producer(void)
{
    int item;

    while (TRUE) {                            /* repeat forever */
        item = produce_item();                /* generate next item */
        if (count == N) sleep();              /* if buffer is full, go to sleep */
        insert_item(item);                    /* put item in buffer */
        count = count + 1;                    /* increment count of items in buffer */
        if (count == 1) wakeup(consumer);    /* was buffer empty? */
    }
}

void consumer(void)
{
    int item;

    while (TRUE) {                            /* repeat forever */
        if (count == 0) sleep();              /* if buffer is empty, got to sleep */
        item = remove_item();                 /* take item out of buffer */
        count = count - 1;                    /* decrement count of items in buffer */
        if (count == N - 1) wakeup(producer); /* was buffer full? */
        consume_item(item);                   /* print item */
    }
}
```

- Producer-consumer problem with fatal race condition:  
because of unconstrained access to count.

# Race condition

1. Buffer is empty.
2. Consumer has loaded count into a register getting ready to test against 0
3. At that moment, scheduler decides to stop running consumer and schedules producer to run on the CPU.
4. Producer inserts an item into buffer, increments count, tests against 1, and issues wakeup(consumer).
5. When consumer is scheduled to run on CPU again, it is not logically asleep yet.
  - It continues with the previous value of count (=0).
  - Consumer tests this value against 0 and issues a sleep() command.
6. But the wakeup call issued by the producer is lost!

# Solution: Semaphores

- Invented by Dijkstra in 1965, they are meant to keep track of the wakeup's and sleeps so they don't get lost.
- Various versions:
  - Original: P() and V() operations (from the initials of the dutch words for test/**P**roberen and signal/**V**erhogen)
  - wait() and signal() (your textbook's notation)

# Semaphore

- **Goal** is to make it easier to program synchronization (SW synchronization primitive).
  - Want to have the same effect from a SW view as the acquire and release.
- A semaphore count represents the number of abstract resources.
- The P (**wait**) operation is used to acquire a resource and decrements count.
  - If a P operation is performed on a count that is 0, then the process must wait for a V or release of a resource.
- The V (**signal**) operation is used to release a resource and increments count.
- P and V occur **atomically**: i.e. indivisibly.

# Busy Wait Semaphore Class

Uses test\_and\_set primitive

```
class Semaphore {  
    Lock mutex;           // Mutual exclusion.  
    int count;            // Resource count.  
public:  
    Semaphore(int num);    // constructor  
    void P();              // wait  
    void V();              // signal  
};  
static inline Semaphore::Semaphore(int num)  
    { count = num; }
```

# Busy Wait Semaphore P() Defn

```
void Semaphore::P()  
{  
    while (count == 0)  
        { /* busy-wait */ };  
    mutex.acquire(); // M.E. lock acquire  
    count--;  
    mutex.release();  
}
```

count is no longer= 0  
so I can decrement

Mutex protects the  
updating of count.

- For semaphores to work, the modification of the semaphore variable, count must be done atomically.
- In this implementation, this is accomplished through the use of lock, which uses the hardware provided atomic operation test-and-set.

# Busy Wait Semaphore V() Defn

```
void  
Semaphore::V()  
{  
    mutex.acquire();  
    count++;  
    mutex.release();  
}
```

# Two Types of Semaphores

- **Counting semaphore** – integer value can range over an unrestricted domain.
- **Binary semaphore** – integer value can range only between 0 and 1; can be simpler to implement. These are also called **mutexes** (because they are used mostly to implement mutual exclusion).



# Busy Wait Semaphore Issues

- Busy waits waste CPU cycles.
- Instead of a busy wait, we can implement a **blocking semaphore**.
- Instead of going into a busy wait loop, the semaphore blocks (yields) so another process/thread can be scheduled to use the CPU.

# Busy Wait Semaphore Issues

- The queue/block/resume implementation of Semaphore has less of a busy wait problem.
  - The processes/threads can be put on a queue for the semaphore!
  - When a release operation occurs, look at the queue to see who gets semaphore.
  - A mutex is used to make sure that two processes don't change count at the same time.
  - Two possible ways to implement:
    1. If P() decrements before is suspends → count can be  $< 0$ .
    2. P() suspends when count == 0, then decrements when woken up → count can never go negative.
  - For negative counts → value gives number of processes/threads in the queue.

# Busy Wait Semaphore Issues

- If process is to be blocked, enqueue (**in the semaphore queue**) PCB of process/thread and call scheduler to run a different process.
  - Thread in Nachos project 1 instead of process.
  - Yield() in Nachos will do this.
- Comment: An interrupt while mutex is held will result in a long delay. So, turn off interrupts during critical section.

# Blocking semaphore

```
Class Semaphore {  
    int count;    // semaphore value  
    ProcessList queue; // queue/list of  
                    // process PCB's for  
                    // this semaphore  
    lock mutex;  
Public:  
    Semaphore(int num);  
    void P();  
    void V();  
};
```

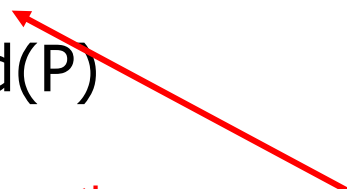
# Blocking semaphore

```
Void Semaphore::P() {  
    lock.acquire();           // for mutual exclusion  
    count--; // decremented before suspend  
    if (count < 0) {  
        // put process on semaphore queue  
        queue.append(this); // this=current  
                             // process  
        lock.release();  
        block(); // suspend/block current process  
                 // or equivalently sleep()  
    }  
    else  
        lock.release();  
}
```

In uniprocessor system lock.acquire() would just turn off interrupts; no need for Spinlock.

# Blocking Semaphore

```
Void Semaphore::V() {  
    lock.acquire();           // for mutual exclusion  
    count++;  
    if (count <= 0) {  
        // wake up some process waiting  
        P = queue.remove()  
        ReadyQueue.append(P)  
    }  
    lock.release();  
}
```



Move the process at the head of L to ReadyQueue

# Semaphores class in nachos (java version)

Let's see how semaphores are implemented in Nachos. (some details deleted)  
This version does not allow semaphore count to go negative.

```
class Semaphore {  
    public String name;        // useful for debugging  
    private int value;         // semaphore value, always >= 0  
    private List queue;        // threads waiting in P() for the value to be > 0  
  
    // constructor  
    public Semaphore(String debugName, int initialValue) {  
        name = debugName;  
        value = initialValue;  
        queue = new List();  
    }  
    // methods in the next couple of slides
```

# Implementation P()

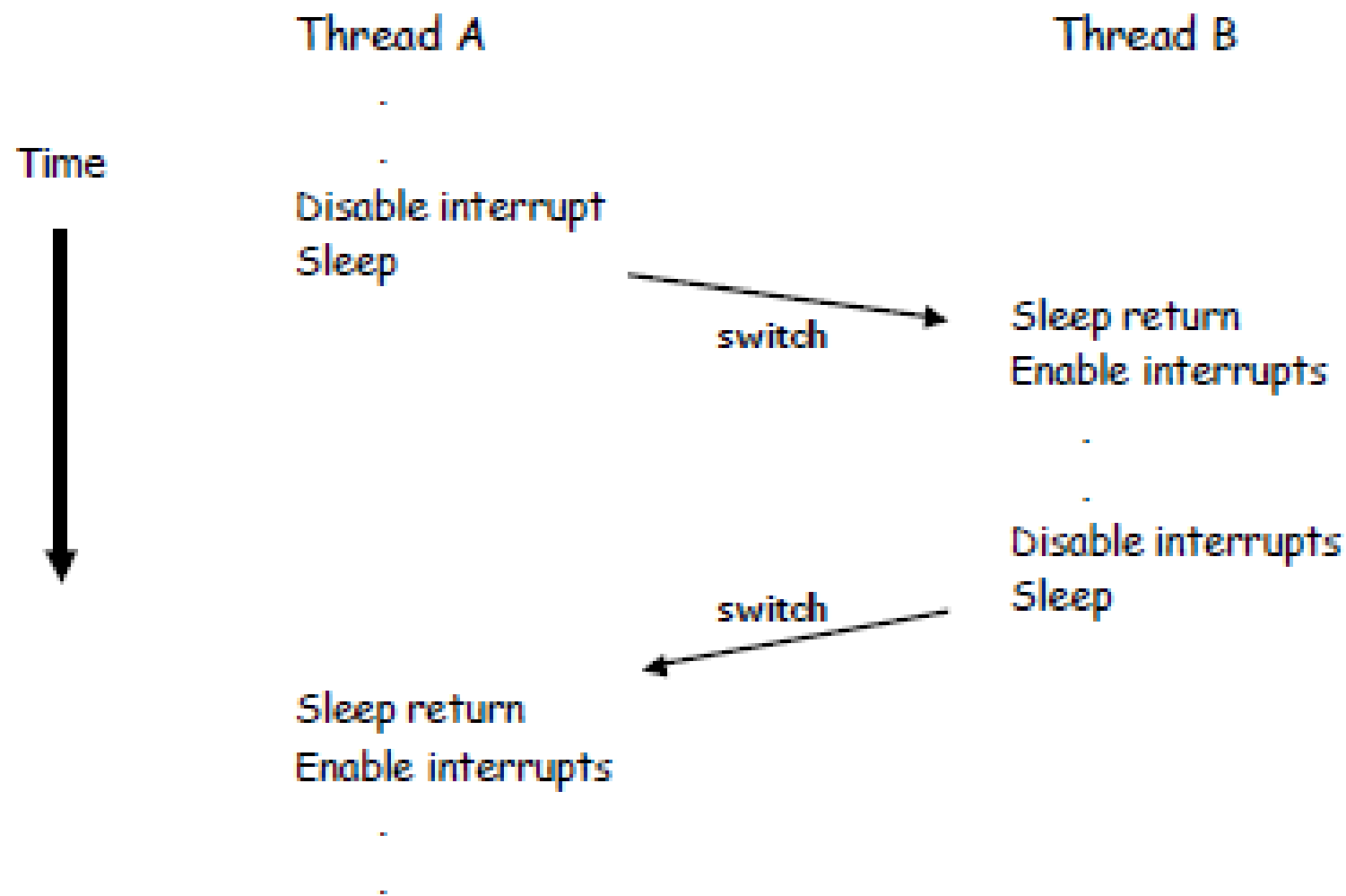
// Checking the value and decrementing (i.e., the entire P() operation)  
// must be done atomically, so we need to disable interrupts  
// before checking the value.

```
public void P() {  
    int oldLevel = Interrupt.setLevel(Interrupt.IntOff); // disable interrupts  
  
    while (value == 0) { // semaphore not available  
        queue.append(NachosThread.currentThread()); // so go to sleep  
        NachosThread.currentThread().sleep();  
        // Thread::Sleep() assumes interrupts are off  
        // next scheduled thread is responsible for enabling interrupts  
    }  
  
    // here we don't need lock or mutex because we have disabled interrupts  
    value-- // semaphore available, therefore decrement (consume value)  
    Interrupt.setLevel(oldLevel); // re-enable interrupts  
}
```



# Implementation V()

```
public void V() {  
    NachosThread thread;  
    // like P(), V() also must be done atomically  
    int oldLevel = Interrupt.setLevel(Interrupt.IntOff);  
    // remove next waiting process from this semaphore's queue  
    // scheduler.readyToRun() assumes the interrupts are disabled when called.  
  
    thread = (NachosThread)queue.remove();  
    if (thread != null) // make thread ready, consuming the V immediately  
        Scheduler.readyToRun(thread); // put the thread into ready queue  
    value++; // increment value  
  
    Interrupt.setLevel(oldLevel);  
}
```



# Deadlock and Starvation

- Implementation of semaphore with waiting queue can cause **deadlock**.
- **Deadlock** – two or more processes are waiting indefinitely for an event that can be caused by only one of the waiting processes.

# Deadlock and Starvation

Example:

P0

P1

share two semaphores S and Q

Semaphore S(1), Q(1);

S.P(); // S==0 -----> Q.P(); //Q==0

Q.P(); // Q== -1 <-----

-----> S.P(); // S== -1

// P0 blocked

// P1 blocked

**DEADLOCK**

S.V();

Q.V();

Q.V();

S.V();

# Deadlock and Starvation

- **Starvation** – indefinite blocking
  - A process may never get the chance to be removed from the semaphore queue in which it is suspended
- **Priority Inversion** – Scheduling problem when lower-priority process holds a lock needed by higher-priority process
  - Solved via **priority-inheritance protocol**

# More on deadlocks...

...In chapter 7

# Classical Problems of Synchronization

- Bounded-Buffer Problem
- Readers and Writers Problem
- Dining-Philosophers Problem

# Bounded-Buffer producer-consumer solution using semaphores

- Shared data



Buffer of size n

Three semaphores are used in the solution:

Semaphore full(0), empty(n), mutex(1);

Initially: full = 0, empty = n, mutex = 1

full: # of full cells

empty: # of empty cells; as items are put into buffer, empty gets decremented and full gets incremented.



# Bounded-Buffer Problem Producer Process

```
Producer() {  
    do {  
        item = produce_item();  
        empty.P();           // wait if buffer is full  
                             // i.e., if #empty cells == 0  
        mutex.P();          // mutual exclusion to protect buffer  
        insert_item(item);  // insert item into buffer  
        mutex.V();          // end M.E.  
        full.V();           // increment full  
    } while (true);  
}
```

# Bounded-Buffer Problem

## Consumer Process

```
Consumer() {  
    do {  
        full.P();           // wait if buffer is empty  
                           // i.e. if # full cells = 0  
        mutex.P();         // start M.E.  
        item = remove_item();  
        mutex.V();         // end M.E.  
        empty.V();         // increment empty.  
        consume_item(item);  
    } while (true);  
}
```

# Readers-Writers Problem

- Shared data among concurrent processes
- Some want to do only reads (**readers**)
  - If two access the data simultaneously, no damage will result; they will both get the same value.
- Others want to update as well as read (**writers**)
  - More than one cannot access data because they may do a write operation.
- Writers should have exclusive access to data.
  - So, if there are any readers accessing data (even for read-only), writer must wait.

# Readers-Writers Problem

- Shared data

**Semaphore mutex(1), wrt(1);**

**int readcount = 0;**

Initially:

**mutex = 1, wrt = 1, readcount = 0**




Only 1 writer allowed access

# Readers-Writers Problem

## Writer Process

```
wrt.P();      startWrite()  
            ...  
            writing is performed  
            ...  
wrt.V();      endWrite()
```



M.E. for  
writing  
process.

# Readers-Writers Problem

## Reader Process

Readcount  
protected by  
mutex:  
only 1 reader  
at a time  
updates  
readcount

```
mutex.P();  
readcount++;  
if (readcount == 1)  
    wrt.P();  
mutex.V();
```

startRead()

*n* readers

- If first reader, wait on wrt semaphore.
- If there is a writer already accessing data, reader must wait.
- If a writer wants to access data while a reader has access, writer must also wait.

...  
reading is performed

```
mutex.P();  
readcount--;  
if (readcount == 0)  
    wrt.V();  
mutex.V();
```

endRead()

If there are no readers left, any waiting writer can proceed.

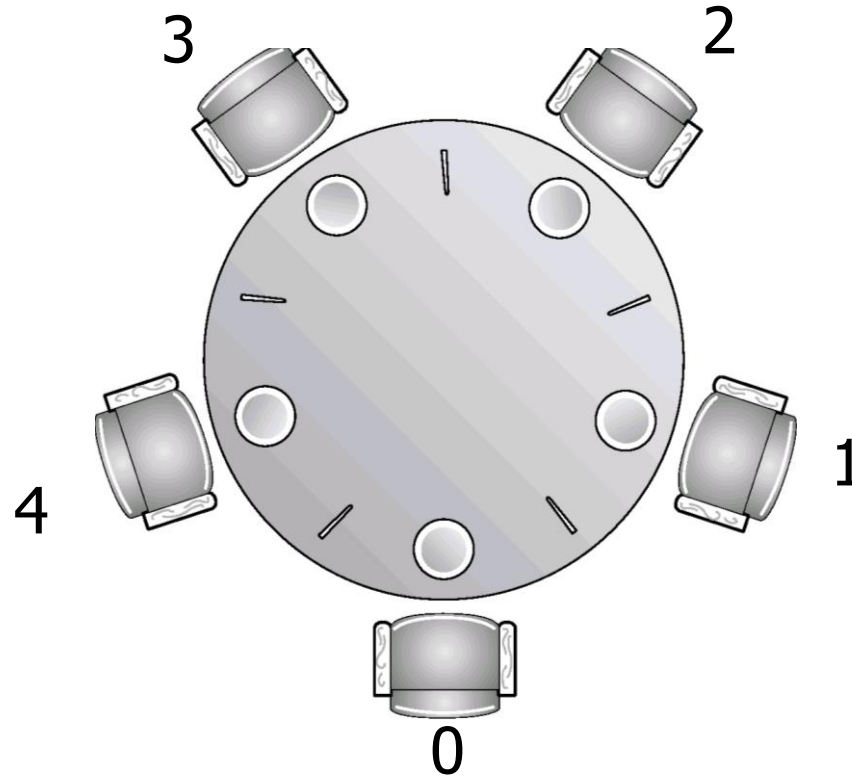
Either a writer or a reader  
can go next.

# Readers-Writers Problem

## Variations

- ***First*** variation – no reader kept waiting unless writer has permission to use shared object
- ***Second*** variation – once writer is ready, it performs the write ASAP
- Both may have starvation leading to even more variations
- Problem is solved on some systems by kernel providing reader-writer locks

# Dining-Philosophers Problem

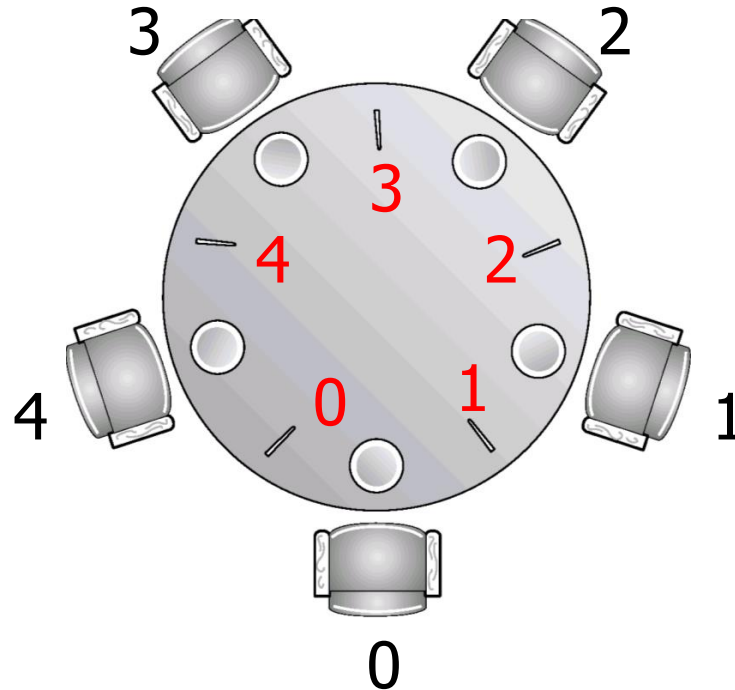


Possibility  
of deadlocks  
exists.

- Five philosophers who think and eat.
- Each philosopher can eat only if he can pick up 2 chopsticks.



# Dining-Philosophers Problem



- Shared data  
**`semaphore chopstick[5];`**  
Initially all values are 1
- For philosopher  $i$ , chopstick  $i$  is the left chopstick and chopstick  $(i+1)\%n$  is the right chopstick.

# Dining-Philosophers Problem

- A **non-solution** to the dining philosophers problem.
- Philosopher  $i$ :

```
do {  
    chopstick[i].P();           // pick up left chopstick  
    chopstick[(i+1) % 5].P()    // pick up right chopstick  
    eat();  
    chopstick[i].V();           // put down left chopstick  
    chopstick[(i+1) % 5].V();    // put down right chopstick  
    think();  
} while (true);
```

- Guarantees Mutual Exclusion, but...
- It is **not a correct** solution because the possibility of deadlock exists:
  - If all philosophers pick up their left fork, none of them can pick up the right fork  
→ deadlock.

# Dining-Philosophers Handling Deadlock

- Deadlock handling
  - Allow at most 4 philosophers to be sitting simultaneously at the table. → There will be
  - Allow a philosopher to pick up the forks only if both are available (picking must be done in a critical section).
  - Use an asymmetric solution -- an odd-numbered philosopher picks up first the left chopstick and then the right chopstick. Even-numbered philosopher picks up first the right chopstick and then the left chopstick.

# The Sleeping Barber Problem



# The Sleeping Barber Problem

- One barber + one barber chair
- $n$  chairs for waiting customers
- Rules:
  1. If there are no customers barber sits on barber chair and goes to sleep
  2. When a customer arrives, he wakes up sleeping barber
  3. If customers arrive and barber is already cutting a customer's hair, they sit down if there are available waiting chairs.
  4. If waiting chairs are full, they leave.
- Problem: program barber and customers to avoid race conditions.
- Similar to queuing situations in real applications (the  $n$  chairs are the queue).

# The Sleeping Barber Problem Solution

```
#define CHAIRS 5                /* # chairs for waiting customers */       $n = 5$ 

typedef int semaphore;          /* use your imagination */

semaphore customers = 0;        /* # of customers waiting for service */
semaphore barbers = 0;         /* # of barbers waiting for customers */
semaphore mutex = 1;           /* for mutual exclusion */
int waiting = 0;               /* customers are waiting (not being cut) */

void barber(void)
{
    while (TRUE) {
        down(&customers);      /* go to sleep if # of customers is 0 */
        down(&mutex);          /* acquire access to 'waiting' */
        waiting = waiting - 1; /* decrement count of waiting customers */
        up(&barbers);          /* one barber is now ready to cut hair */
        up(&mutex);            /* release 'waiting' */
        cut_hair();            /* cut hair (outside critical region) */
    }
}

void customer(void)
{
    down(&mutex);               /* enter critical region */
    if (waiting < CHAIRS) {     /* if there are no free chairs, leave */
        waiting = waiting + 1; /* increment count of waiting customers */
        up(&customers);        /* wake up barber if necessary */
        up(&mutex);            /* release access to 'waiting' */
        down(&barbers);        /* go to sleep if # of free barbers is 0 */
        get_haircut();         /* be seated and be serviced */
    } else {
        up(&mutex);            /* shop is full; do not wait */
    }
}
```

# Lock and Condition Variables

- Semaphores are a **huge step up**.
- But problem with **semaphore** is that they are **dual purpose**: used for
  - *mutex* and
  - *scheduling*
- This makes the code hard to read, and hard to get right

# Lock and Condition Variables

- Idea in monitors is to separate these concerns: use
  - locks for *mutual exclusion* and
  - condition variables (like a scheduling semaphore) for *scheduling* constraints



# Monitors

- High-level synchronization construct that allows the safe sharing of an abstract data type among concurrent processes.

```
monitor monitor-name
{
    shared variable declarations
    procedure body P1 (...) {
        ...
    }
    procedure body P2 (...) {
        ...
    }
    procedure body Pn (...) {
        ...
    }
    {
        initialization code
    }
}
```

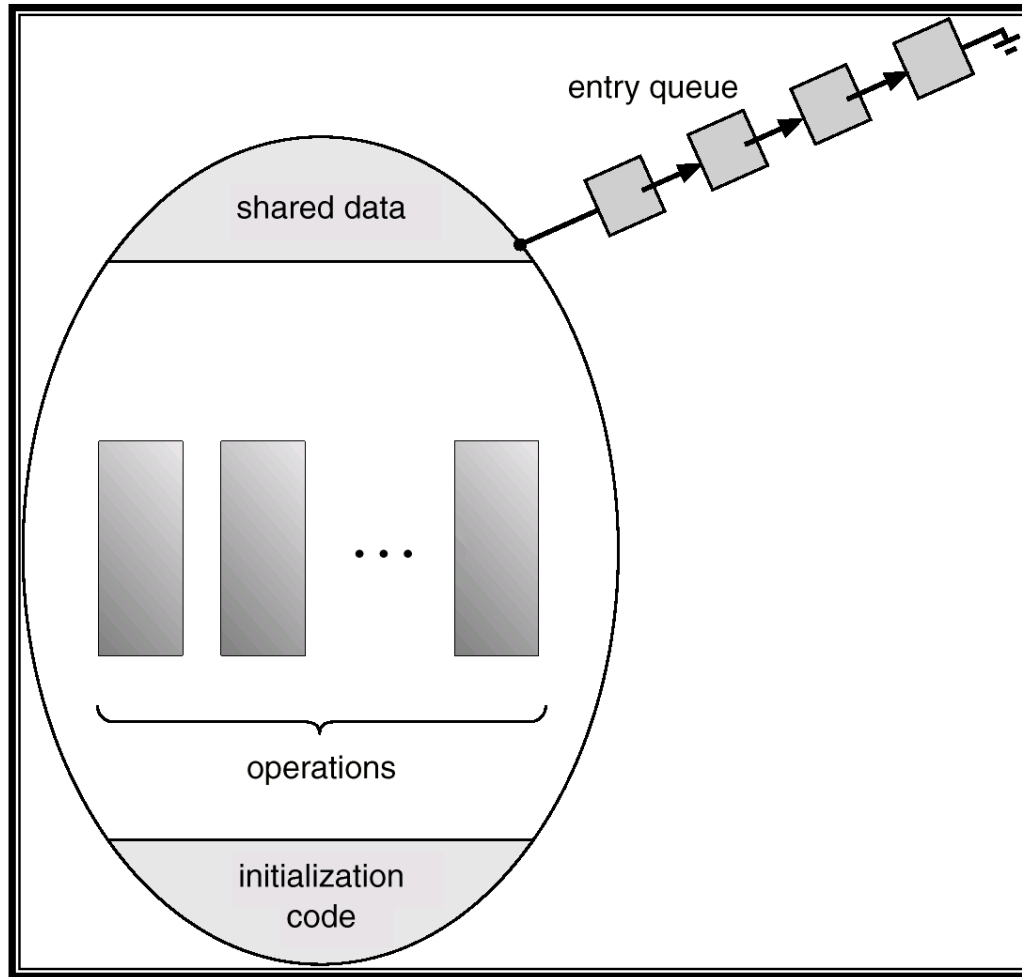
**Similar to C++ or Java class with data and ops on data**

**Can be implemented using semaphores at the low level.**

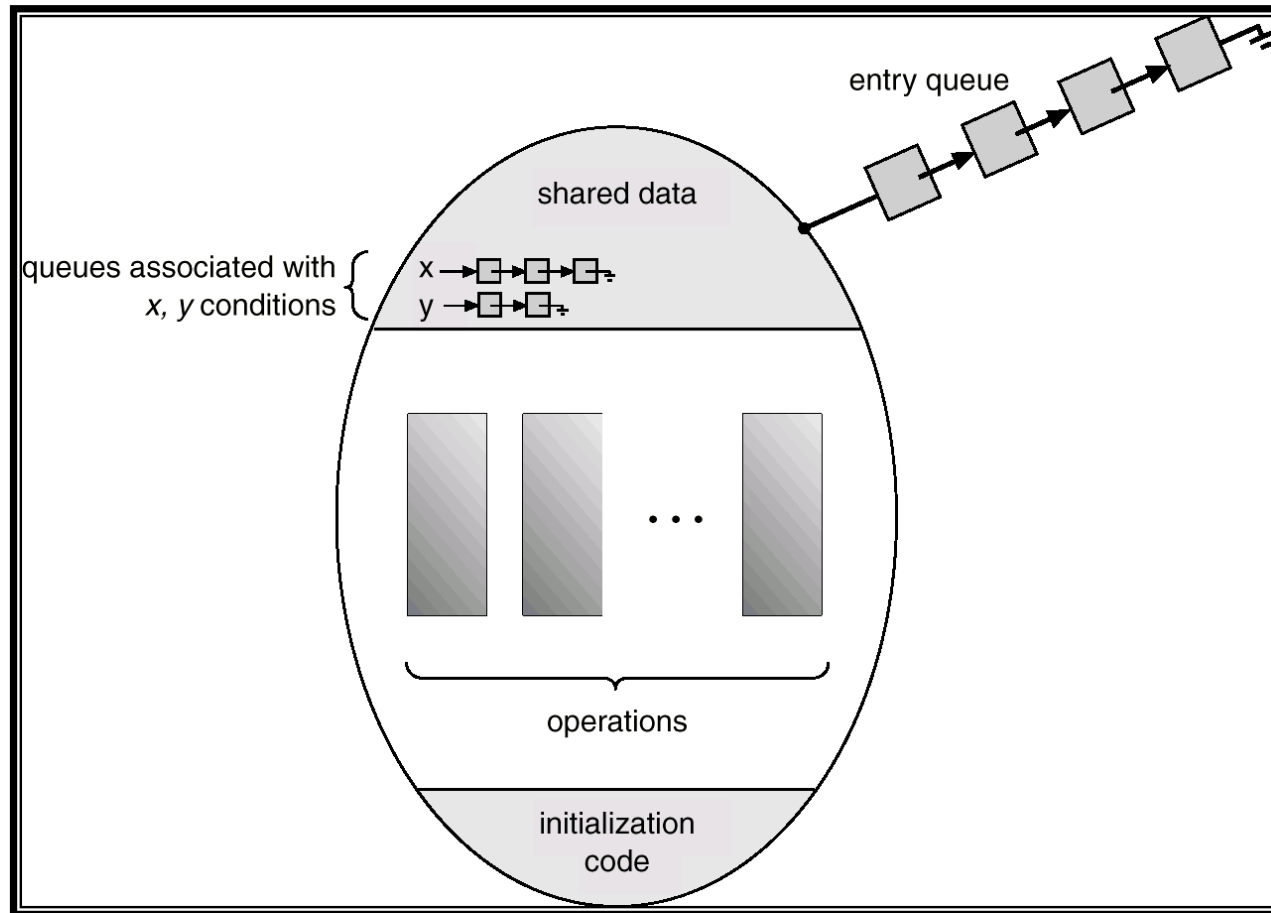
**Monitor guarantees that only one process enters monitor**

**Monitors have entry queues**

# Schematic View of a Monitor



# Monitor With Condition Variables



# Lock and Condition Variables

- Monitor: a lock and zero or more condition variables for managing concurrent access to shared data
- Supported by Posix.1c pthreads, Ada, Java, Modula II

**POSIX** := Portable Operating System Interface on UNIX

# Lock and Condition Variables

- Programming Languages that support this concept of a monitor are Ada95, Java
- For languages without language support for monitors (e.g. C, C++) operating systems like Nachos, WinNT, OS/2, or Solaris, support **locks and condition variables as library functions** that are linked into an application

Gives programmer flexibility  
To extend synch possibilities

# Condition Variables

- To allow a process to wait within the monitor, a **condition variable** must be declared, as

**condition x, y;**

- Condition variable can only be used with the operations **wait** and **signal**.

- The operation

**x.wait();**

means that the process invoking this operation is suspended until another process invokes

**x.signal();**

- The **x.signal** operation resumes exactly one suspended process. If no process is suspended, then the **signal** operation has no effect.

# Condition Variables

- Condition variables were introduced because a process may want to wait for different reasons.
  - Example: in bounded-buffer producer/consumer process, producer waits if the buffer is full and consumer waits if the buffer is empty.
  - We can use full and empty condition variables.
- Condition variables do not have a “truth value”: it does not have any stored value accessible to the program.
  - In this sense it is not a traditional variable.
- Condition variables are represented by **queues** of processes/threads.

# Monitors example

```
monitor ProducerConsumer
  condition full, empty;
  integer count;
  procedure insert(item: integer);
  begin
    if count = N then wait(full);
    insert_item(item);
    count := count + 1;
    if count = 1 then signal(empty)
  end;
  function remove: integer;
  begin
    if count = 0 then wait(empty);
    remove = remove_item;
    count := count - 1;
    if count = N - 1 then signal(full)
  end;
  count := 0;
end monitor;
```

```
procedure producer;
begin
  while true do
    begin
      item = produce_item;
      ProducerConsumer.insert(item)
    end
  end;
procedure consumer;
begin
  while true do
    begin
      item = ProducerConsumer.remove;
      consume_item(item)
    end
  end;
```

- Outline of producer-consumer problem with monitors
  - only one monitor procedure active at one time
  - buffer has  $N$  slots



# Mesa vs. Hoare Monitors

- Hoare semantics (ref: CACM paper by C.A.R. Hoare, vol 17, no 10, 1974) – original paper on monitors.
- Need to be careful about the precise definition of signal and wait.
  - Different implementations (and semantics) exist.
    - Hoare semantics
    - Mesa semantics

# Mesa vs. Hoare Monitors

- Hoare-style: (some textbooks)
  - Signaller gives up lock & CPU to waiter; waiter runs immediately with no intervening thread being scheduled.
  - Signalled waiter gives lock, processor back to signaller when exits critical section or if it waits again
  - Harder to implement.

# Mesa vs. Hoare Monitors

- Mesa-style: (Nachos, Java, Ada, ...)
  - Signaller keeps lock, processor
  - Waiter simply put on ready queue , with no special priority (in other words, waiter may have to wait for a lock )
  - Simpler to implement: just put the waiter in the queue and continue.
  - *Mesa* was a programming language developed in the late 1970s at the Xerox PARC.

# Extensions

- To have more control over how processes are picked from queues, one can extend the `wait()` with an integer priority argument:
  - `x.wait(p)`: means the same as regular `wait()` except the processes are awakened in increasing order of `p`.
  - This means the condition variables used this way are not fair.
- In addition to `wait()` and `signal()` one can have `broadcast()` operation also.
  - In this case, the signal to awaken is sent to all the waiting processes
  - One of them will get a hold of the lock and succeed.

# Condition variables in Nachos (C++) using semaphores

```
class Condition {
public:
    Condition();                // initialize condition to
                                // "no one waiting"
    ~Condition();               // deallocate the condition

    void Wait(Lock *conditionLock);    // these are the 3 operations on
    // condition variables; releasing the
    // lock and going to sleep are
    // *atomic* in Wait()
    void Signal(Lock *conditionLock);  // conditionLock must be held by
    void Broadcast(Lock *conditionLock); // Wake up all threads waiting on this condition
private:
    char* name;
    List<Semaphore *> *waitQueue;
};
```

# Condition variable constructor & destructor

```
Condition::Condition()
```

```
{
```

```
    // we will use semaphores to implement it.
```

```
    waitQueue = new List<Semaphore *>;
```

```
}
```

```
Condition::~~Condition()
```

```
{
```

```
    delete waitQueue;
```

```
}
```

# Condition var wait()

```
// Atomically release monitor lock and go to sleep.
// Our implementation uses semaphores to implement this, by
// allocating a semaphore for each waiting thread. The signaler
// will up() this semaphore, so there is no chance the waiter
// will miss the signal, even though the lock is released before
// calling down().
//
// Note: we assume Mesa-style semantics, which means that the
// waiter must re-acquire the monitor lock when waking up.
//
// "conditionLock" -- lock protecting the use of this condition
//-----
void Condition::Wait(Lock* conditionLock)
{
    Semaphore *waiter;

    waiter = new Semaphore("condition", 0); // create a semaphore to keep
                                           // track of this waiting thread

    waitQueue->Append(waiter); // append the semaphore to cond var queue
    conditionLock->Release();   // release the lock before possibly blocking
    waiter->down();              // semaphore->down() might put this thread to sleep.
    conditionLock->Acquire();    // after waking up re-acquire the lock to enter monitor.
    delete waiter;
}
```

# Condition var signal()

```
// Condition::Signal
// Wake up a thread waiting on this condition, if any.
//
// Note: we assume Mesa-style semantics, which means that the
// signaler doesn't give up control immediately to the thread
// being woken up (unlike Hoare-style).
//
// Also note: we assume the caller holds the monitor lock.
// This allows us to access waitQueue without disabling interrupts.
//
// "conditionLock" -- lock protecting the use of this condition

void Condition::Signal(Lock* conditionLock)
{
    Semaphore *waiter;
    if (!waitQueue->IsEmpty())
    {
        waiter = waitQueue->Remove();           // remove semaphore at head of queue
        waiter->up();                           // wake up the waiting semaphore.
    }
}
```



# Condition var broadcast()

```
//-----  
// Condition::Broadcast  
// Wake up all threads waiting on this condition, if any.  
//  
// "conditionLock" -- lock protecting the use of this condition  
//-----
```

```
void Condition::Broadcast(Lock* conditionLock)  
{  
    while (!waitQueue->IsEmpty())  
    {  
        Signal(conditionLock);  
    }  
}
```

# Comparison of semaphores to condition variables

- Semaphores are **commutative** and condition variables are not.
- If a semaphore does a  $V()$  and there is no waiting thread, a later  $P()$  done by a thread will not block and continue.
- Condition variable  $\text{signal}()$  is lost if there are no waiting threads. A later  $\text{wait}()$  will block the thread.

# Java monitors

- Java is “thread safe”
  - Java thread synchronization is built into the language (monitors).
  - Threads are declared “synchronized”
  - The compiler takes care of correct synchronization.

# Java monitors

- `wait()`
- `notify()` corresponding to `signal()`
  - The notify call is used in **one-at-a-time situations**. Ex: waiting on an empty FIFO queue until an entry is inserted. When one entry comes in, one waiting thread, the next in FIFO order, can return from its wait call and get the element.
- `notifyall()` corresponding to `broadcast()`
  - A synchronized method can call `notifyAll()` which will notify all threads waiting on a particular object. Then all the threads can start up, **one at a time** (because of the critical section) and retry their operation
- `notifyall()` provides an easy way to let all waiting threads retry their operation when it isn't easy to figure out which particular thread of several may need to wake up in response.

# Inter-Process Communication (IPC)

- All previous methods used shared memory approach to communicate with each other.
- Processes can also communicate using message passing instead of shared memory.

# Inter-Process Communication (IPC)

- Messages can be lost over the network.
- Receiver sends back acknowledgement.
  - If sender does not receive an acknowledgement, it resends message.
  - Acknowledgements can also be lost.
  - Solution: number the messages. When receiver gets a message with same message number, it's duplicate message.

# Message Passing

```
#define N 100                                /* number of slots in the buffer */

void producer(void)
{
    int item;
    message m;                                /* message buffer */

    while (TRUE) {
        item = produce_item( );              /* generate something to put in buffer */
        receive(consumer, &m);               /* wait for an empty to arrive */
        build_message(&m, item);             /* construct a message to send */
        send(consumer, &m);                  /* send item to consumer */
    }
}

void consumer(void)
{
    int item, i;
    message m;

    for (i = 0; i < N; i++) send(producer, &m); /* send N empties */
    while (TRUE) {
        receive(producer, &m);               /* get message containing item */
        item = extract_item(&m);             /* extract item from message */
        send(producer, &m);                 /* send back empty reply */
        consume_item(item);                 /* do something with the item */
    }
}
```

← Equivalent of shared buffer slots (N)

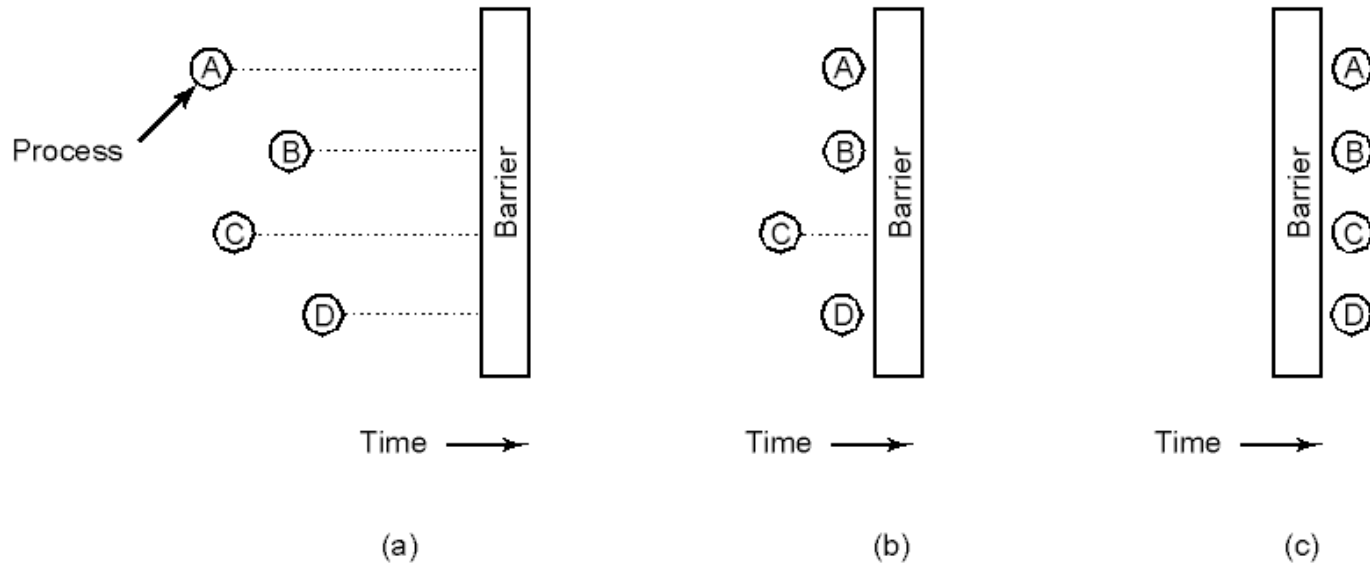
The producer-consumer problem with N messages

# Barriers

- Barriers are synchronization primitives that ensure that some processes do not outrun others – if a process reaches a barrier, it has to wait until every process reaches the barrier
- When a process reaches a barrier, it acquires a lock and increments a counter that tracks the number of processes that have reached the barrier – it then spins on a value that gets set by the last arriving process
- Must also make sure that every process leaves the spinning state before one of the processes reaches the next barrier



# Barriers



- Use of a barrier
  - processes approaching a barrier (a)
  - all processes but one blocked at barrier (b)
  - last process arrives, all are let through (c)

# Barrier Implementation using spinlock

```
barrier.lock.acquire();    // acquire lock for barrier to protect "counter"
if (barrier.counter == 0) { // first process arriving
    barrier.flag = 0;      // don't let other processes proceed
}
mycount = (barrier.counter++);
barrier.lock.release();
if (mycount == p) {       // if p processes/threads have arrived
    barrier.counter = 0;   // let them go thru by resetting counter
    barrier.flag = 1;      // and flag
} else {
    while (barrier.flag == 0) { /* busy-wait */ };
}
```

# Barriers

- Applied to tasks divided into phases and no process may proceed to next phase until all processes are ready.
  - Place a barrier at the end of each phase.
- Java implements barriers

# Summary Synchronization

- SW solution: Peterson's algorithm
- HW aided solutions
  - Locks
  - Semaphores
  - Monitors
    - Locks and condition variables
- Message Passing methods
- Barriers