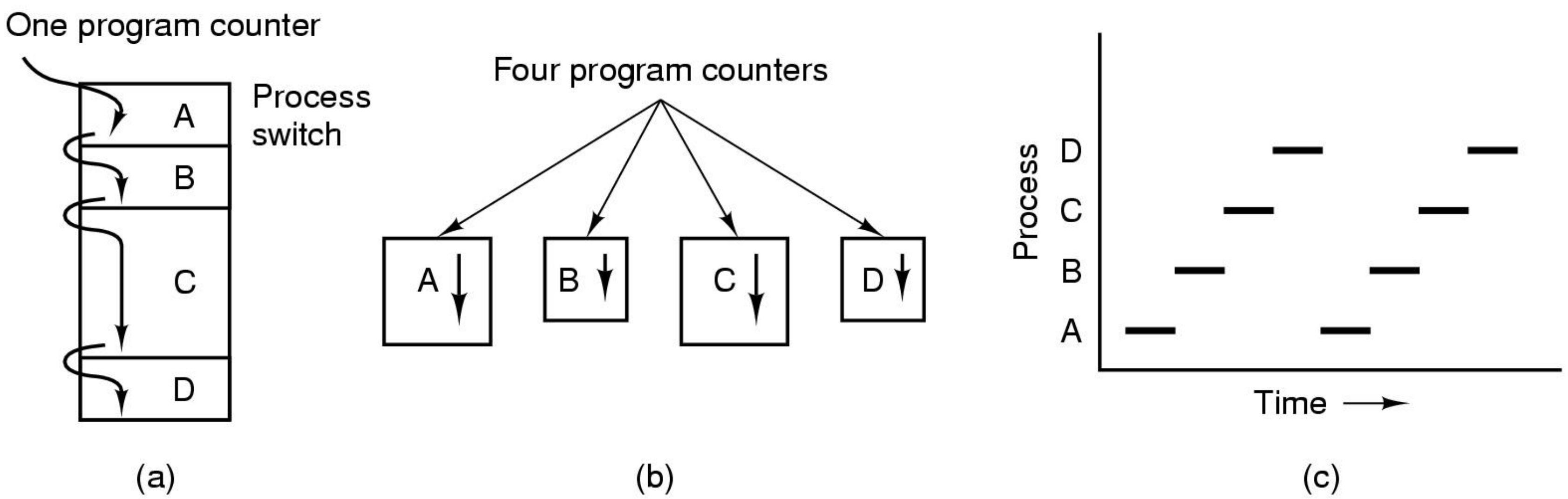
Chapter 2

Processes and Threads

2.1 Processes
2.2 Threads
2.3 Scheduling
2.4 Interprocess communication

2.1 Processes

Processes
The Process Model



- Multiprogramming of four programs
- Conceptual model of 4 independent, sequential processes
- Only one program active at any instant

Processes
Process Concept

- An operating system executes a variety of programs:
  - Batch system – jobs
  - Time-shared systems – user programs or tasks
- Process – a program in execution; process execution must progress in sequential fashion
- A process resources includes:
  - Address space (text segment, data segment)
  - CPU (virtual)
    - program counter
    - registers
    - stack
  - Other resource (open files, child processes…)

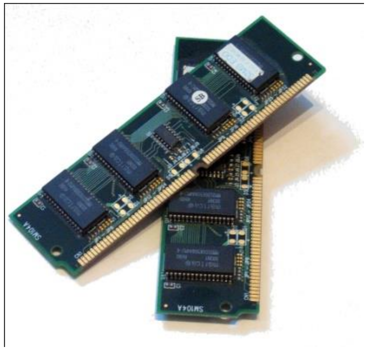
Processes

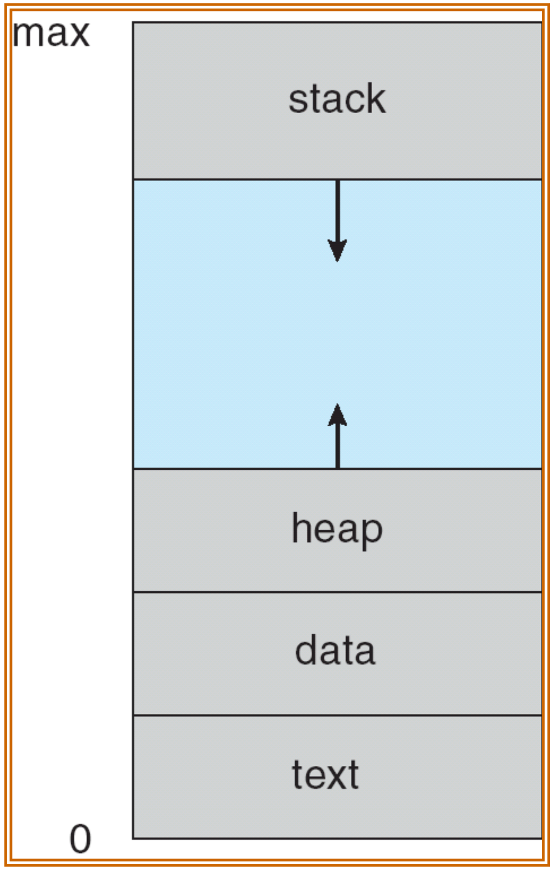
Process in Memory

IT_client.exe

Word.exe

VMWare.exe





Load to memory

Sequential instructions is executed by CPU

PROCESS

Processes
Process Creation (1)

Principal events that cause process creation

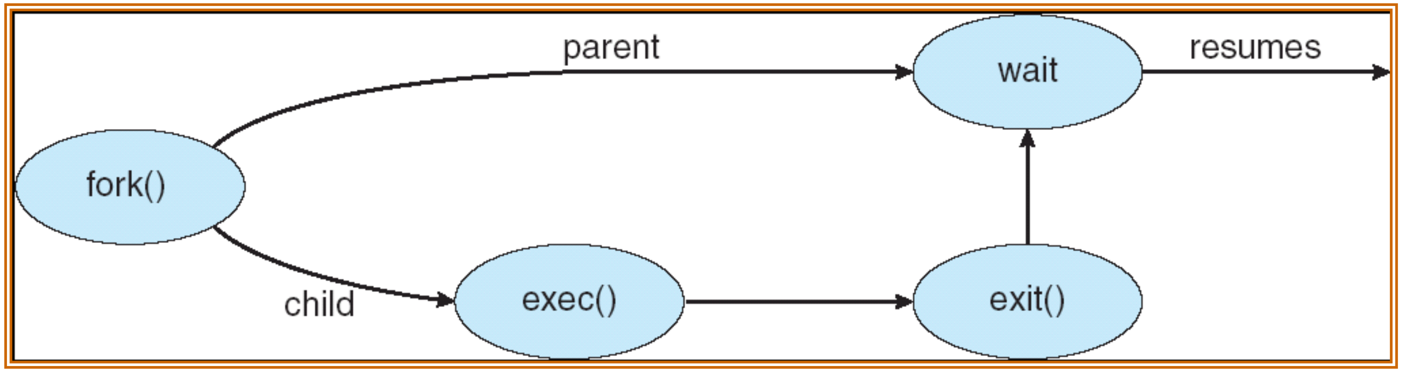
1. System initialization
2. Execution of a process creation system Call
3. User request to create a new process
4. Initiation of a batch job

Processes
Process Creation (2)

- Address space
  - Child duplicate of parent
  - Child has a program loaded into it
- UNIX examples
  - **fork** system call creates new process
  - **exec** system call used after a **fork** to replace the process' memory space with a new program

Processes

Process Creation (3) : Example



Processes
Process Termination

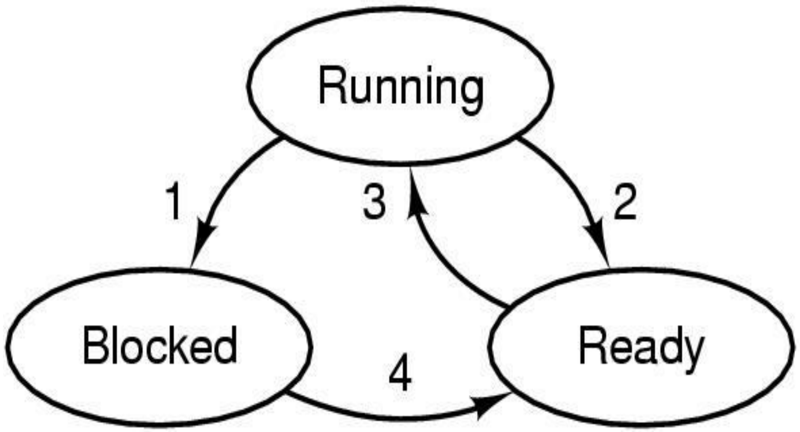
Conditions which terminate processes

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

Processes
Process Hierarchies

- Parent creates a child process, child processes can create its own process
- Forms a hierarchy
  - UNIX calls this a "process group"
- Windows has no concept of process hierarchy
  - all processes are created equal

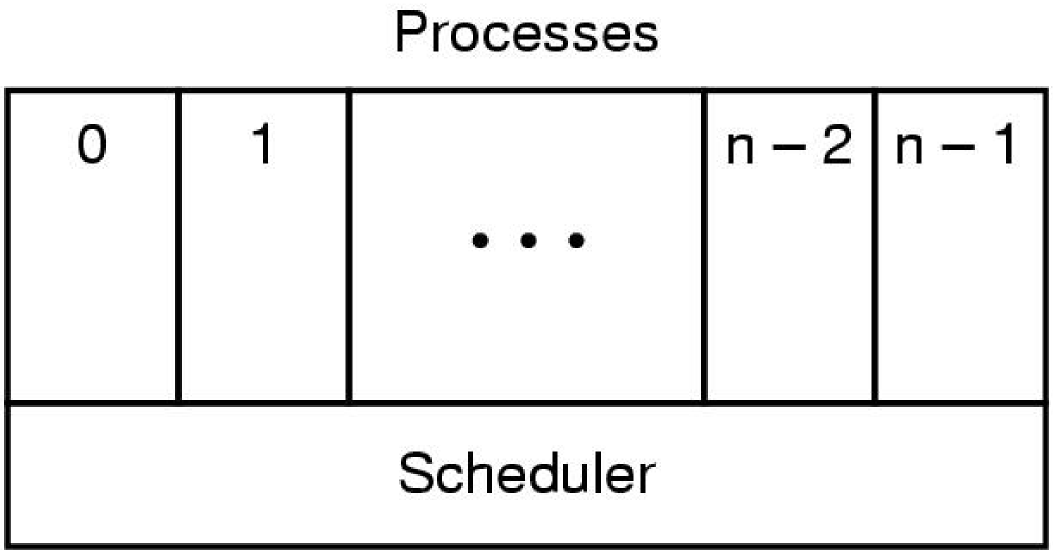
Processes
Process States (1)



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

- Possible process states
  - running
  - blocked
  - ready
- Transitions between states shown

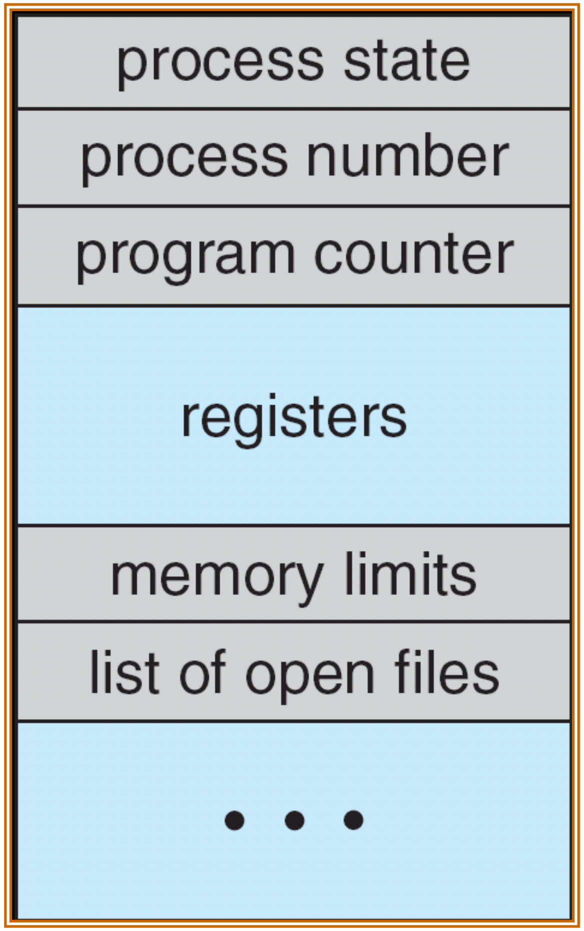
Processes
Process States (2)



- Lowest layer of process-structured OS
  - handles interrupts, scheduling
- Above that layer are sequential processes

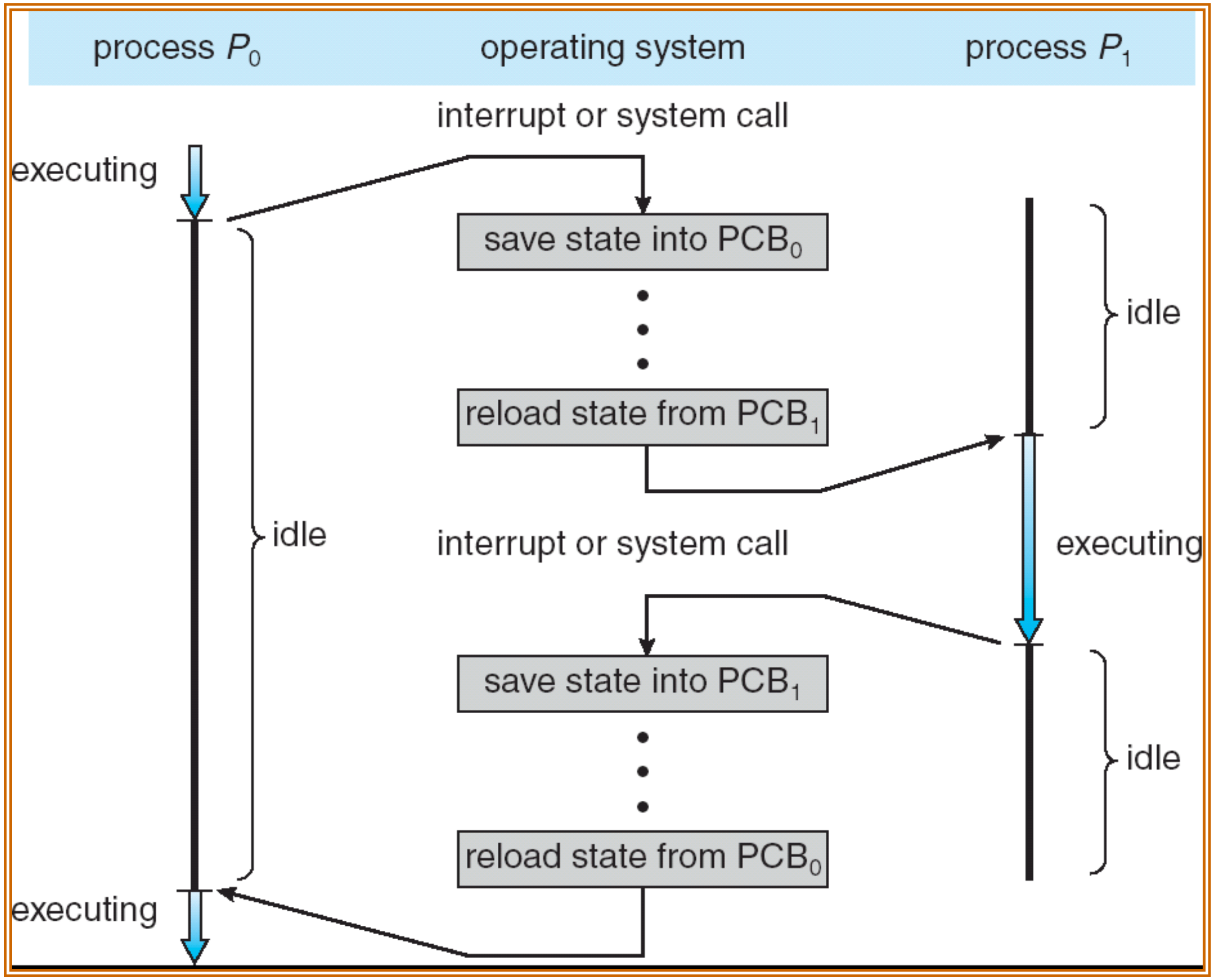
Implementation of Processes

Process Control Block (PCB)



Implementation of Processes

context switch



Processes

Implementation of Processes (1)

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

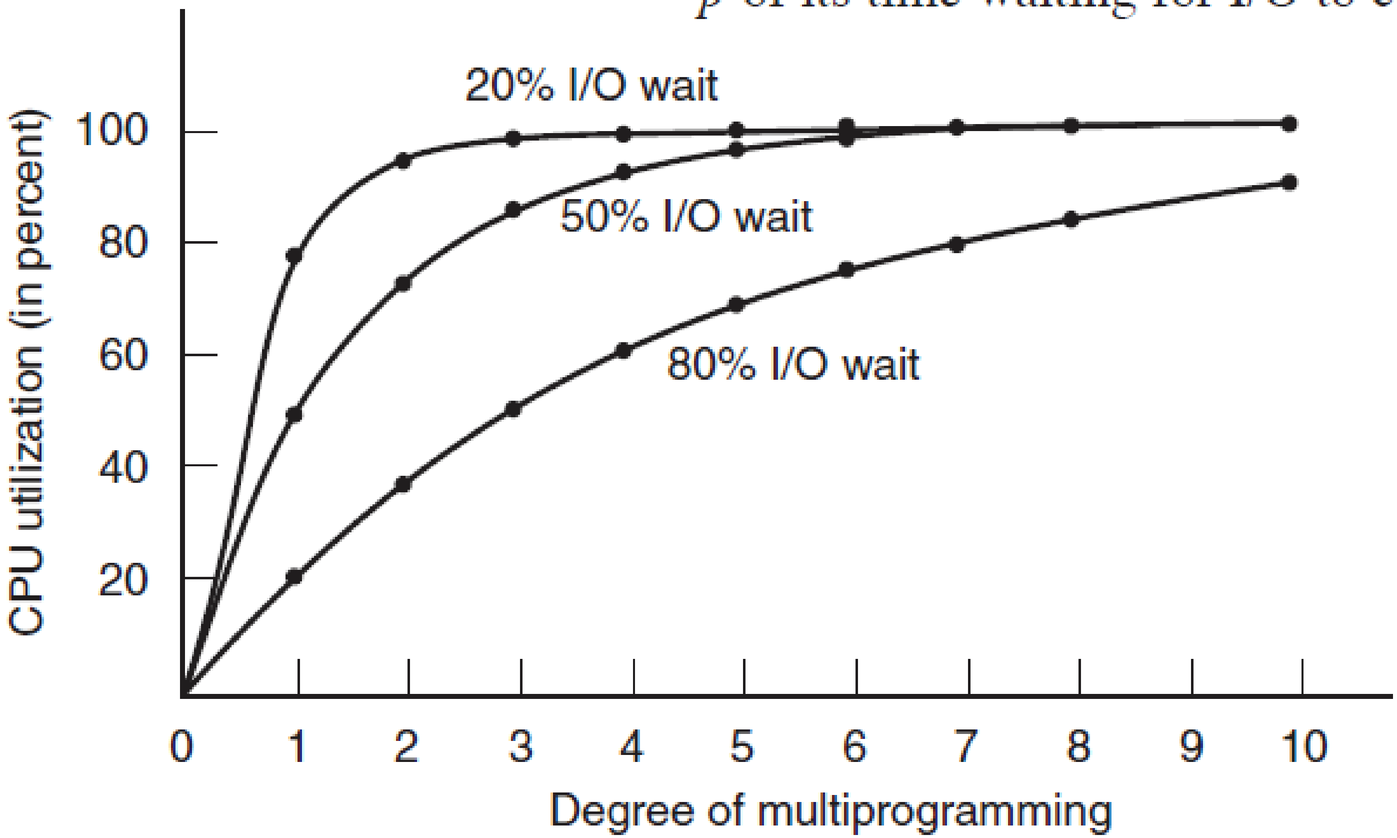
Fields of a process table entry

Modeling Multiprogramming

$$\text{CPU utilization} = 1 - p^n$$

n processes in memory at once

p of its time waiting for I/O to complete

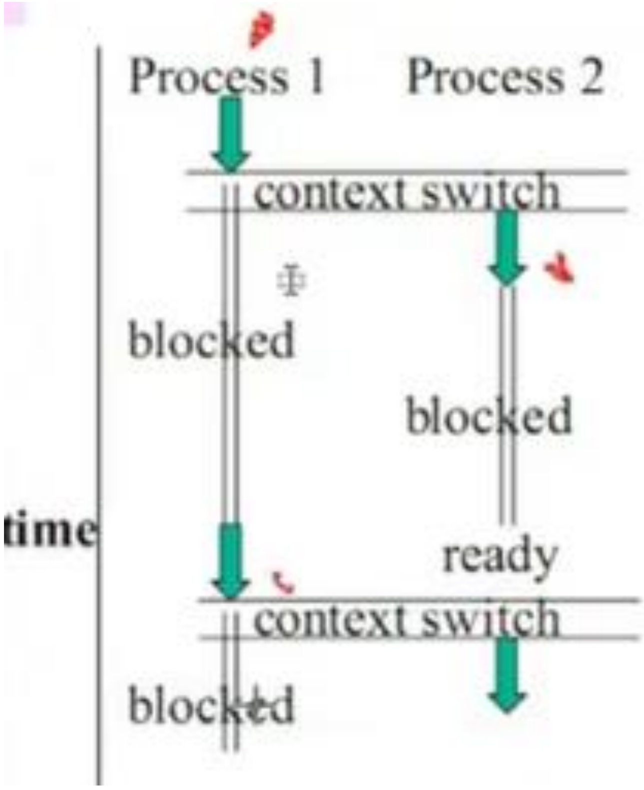


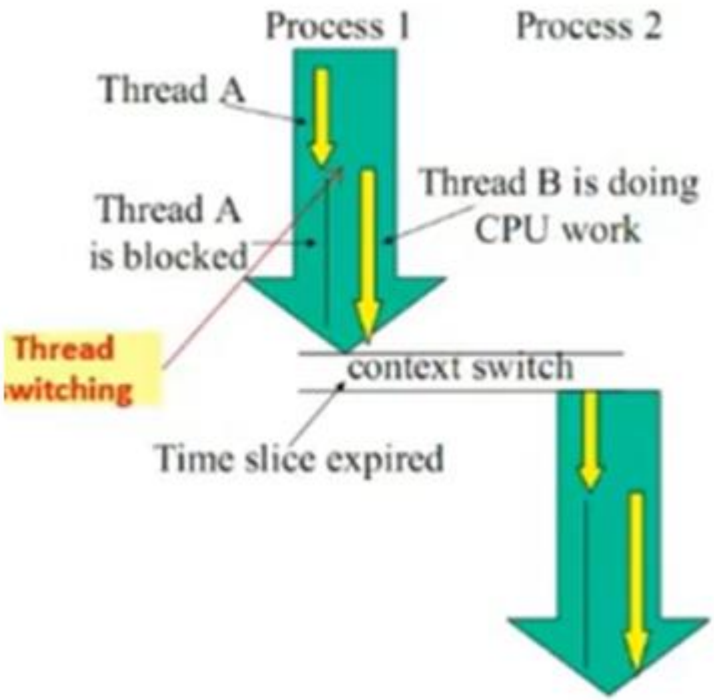
CPU utilization as a function of the number of processes in memory

2.2 Threads

Threads

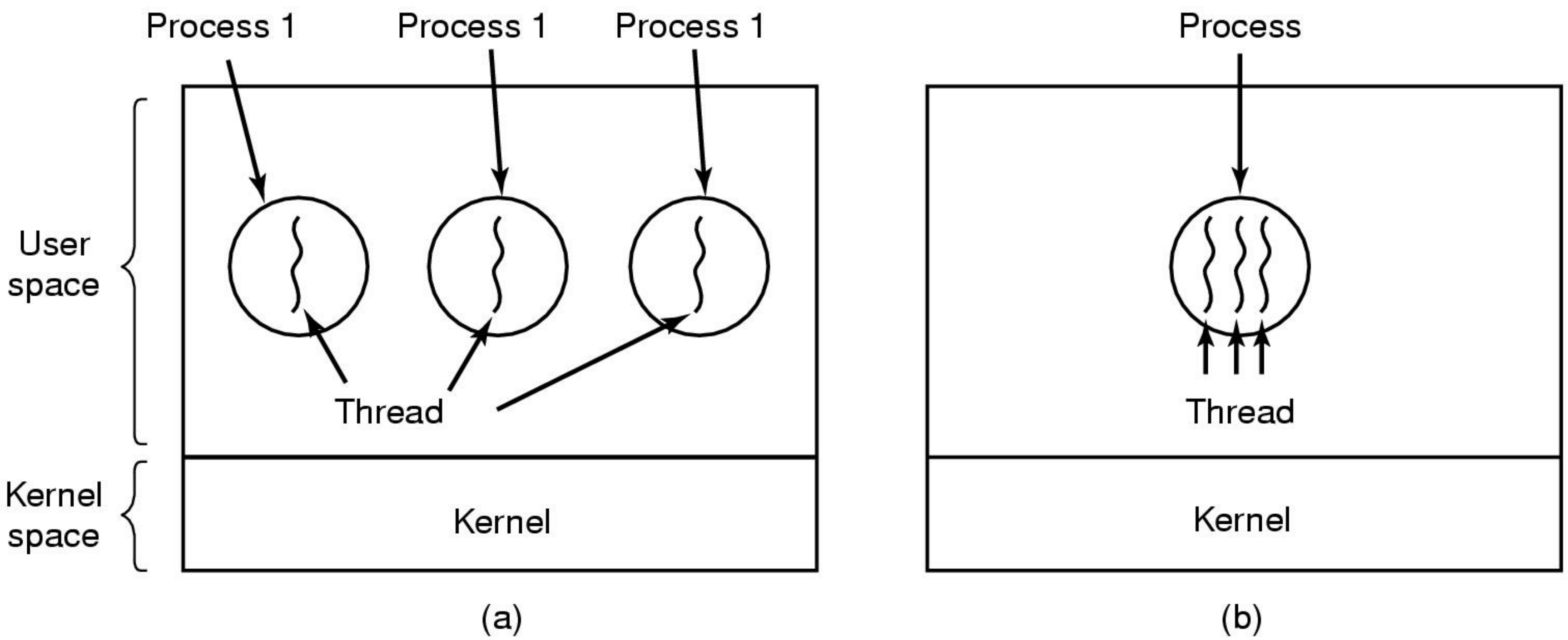
A thread is an unit execution in a process (a function, mini-process or light-process)





Threads

The Thread Model



(a) Three processes each with one thread (Unix) => Multiprogramming
(b) One process with three threads (Window/Linux) => Multithreading

Threads
Process with single thread

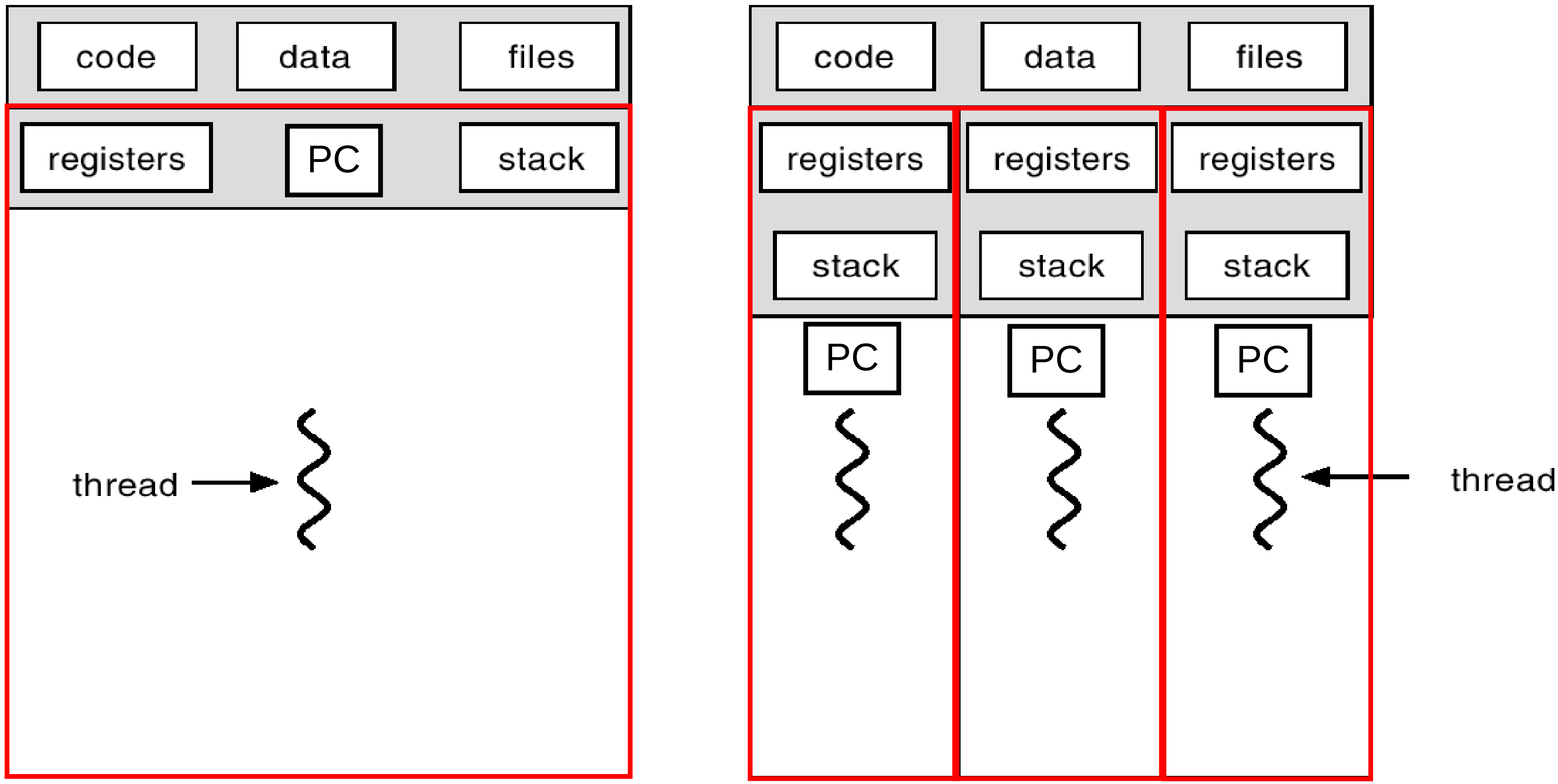
- A process:
  - Address space (text section, data section)
  - Single thread of execution
    - program counter
    - registers
    - Stack
  - Other resource (open files, child processes…)

Threads
Process with multiple threads

Multiple threads of execution in the same environment of process

- Address space (text section, data section)
- Multiple threads of execution, each thread has private set:
  - program counter
  - registers
  - stack
- Other resource (open files, child processes…)

Threads
Single and Multithreaded Processes



single-threaded
multithreaded

Threads

Items shared and Items private

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	

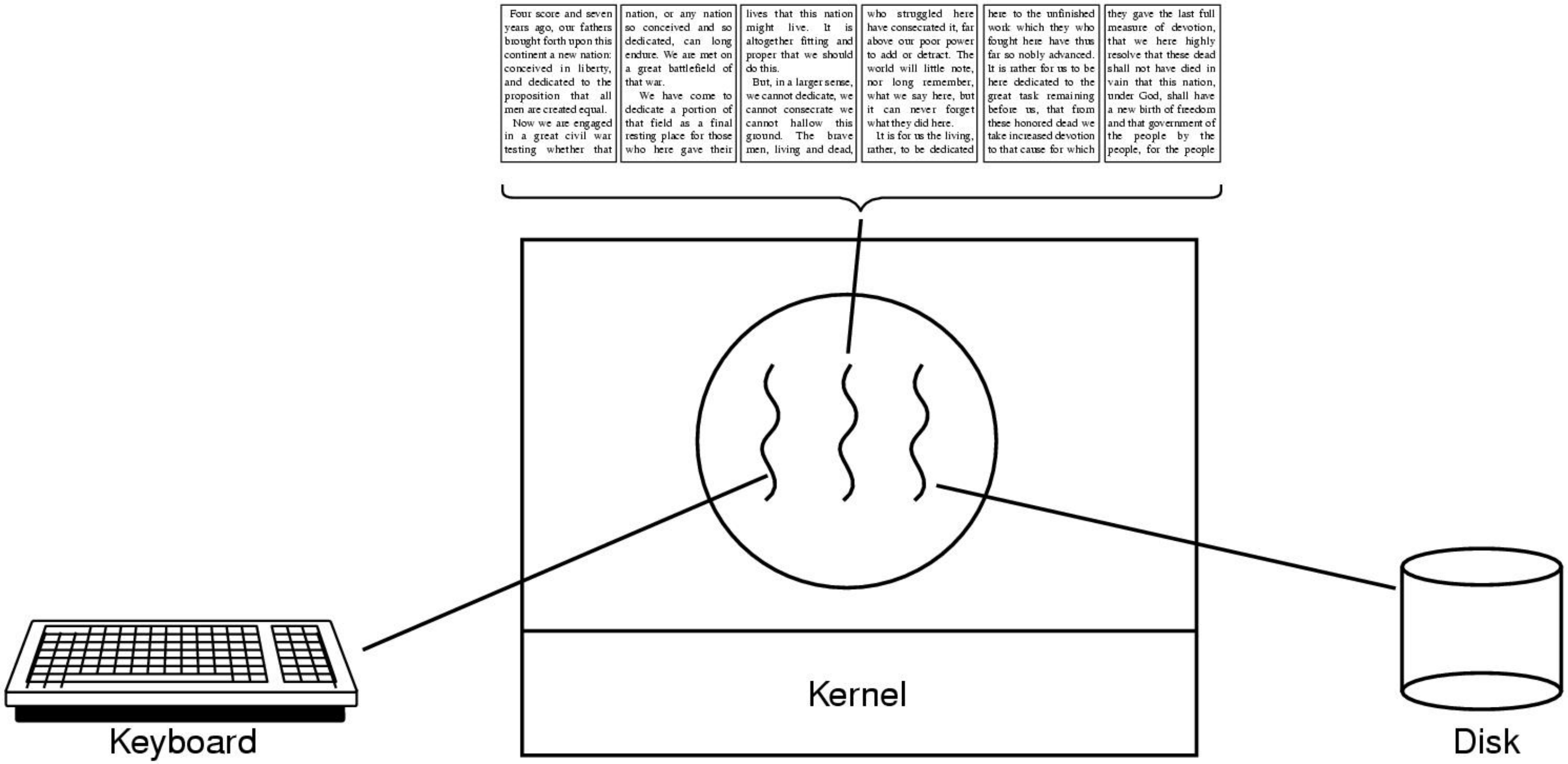
- Items shared by all threads in a process
- Items private to each thread

Threads
Benefits

- Responsiveness
- Resource Sharing
- Economy
- Utilization of Multiprocessor Architectures

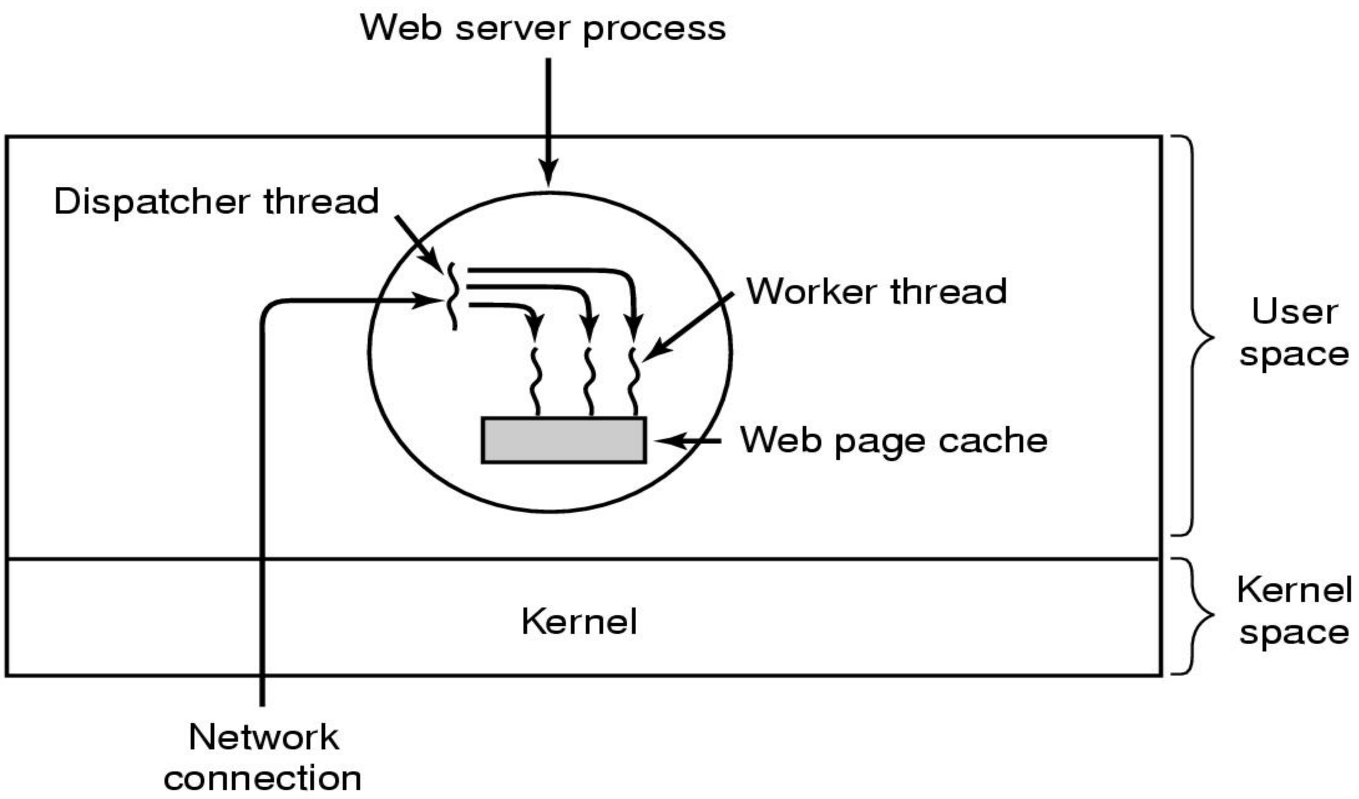
Threads

Thread Usage (1)



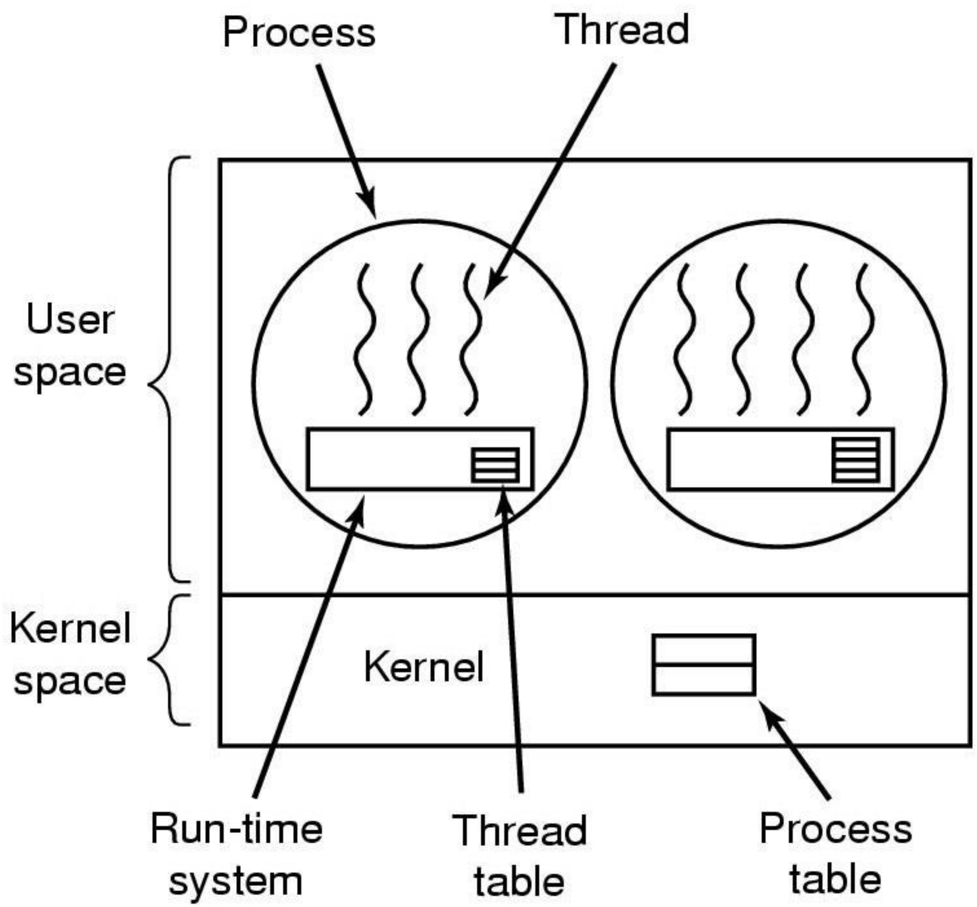
A word processor with three threads - Self-study

Threads
Thread Usage (2)



A multithreaded Web server - Self-study

Threads
Implementing Threads in User Space (1)



A user-level threads package

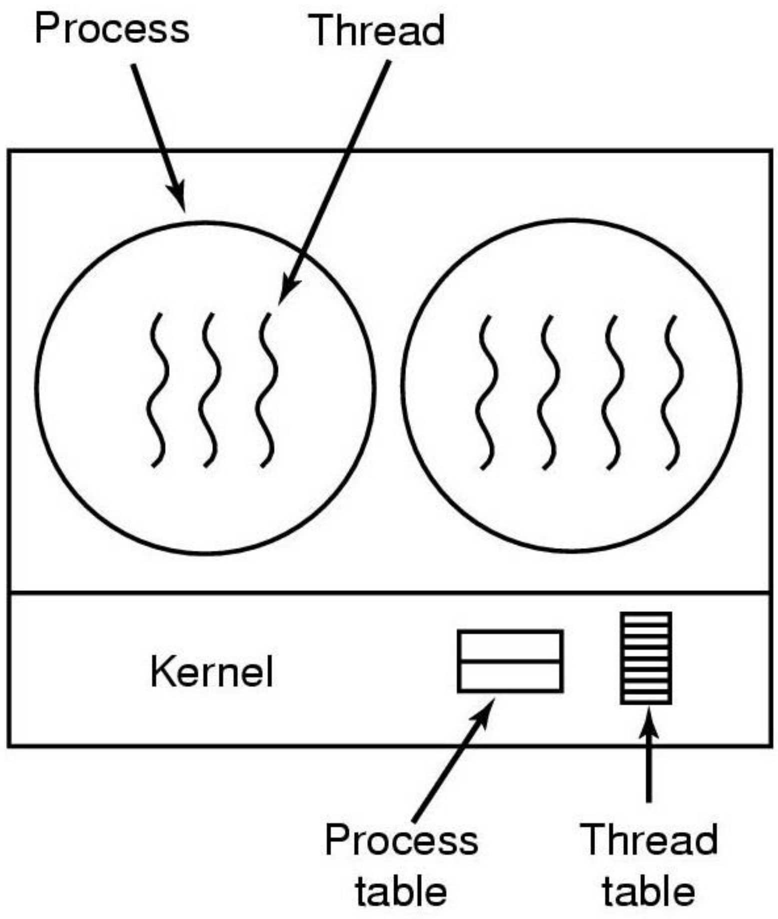
Threads

Implementing Threads in User Space (2)

- Thread library, (run-time system) in user space
    - thread_create
    - thread_exit
    - thread_wait
    - thread_yield (to voluntarily give up the CPU)
- Thread control block (TCB) (Thread Table) stores states of user thread (program counter, registers, stack)
- Kernel does not know the present of user thread

Threads

Implementing Threads in the Kernel (1)



A threads package managed by the kernel

Threads

1. When should we implement the threads in the user space?
2. When should we implement the threads in the kernel space?

2.3 Interprocess Communication

Cooperating Processes

- **Independent** process cannot affect or be affected by the execution of another process
- **Cooperating** process can affect or be affected by the execution of another process
- Advantages of process cooperation
  - Information sharing
  - Computation speed-up
  - Modularity
  - Convenience

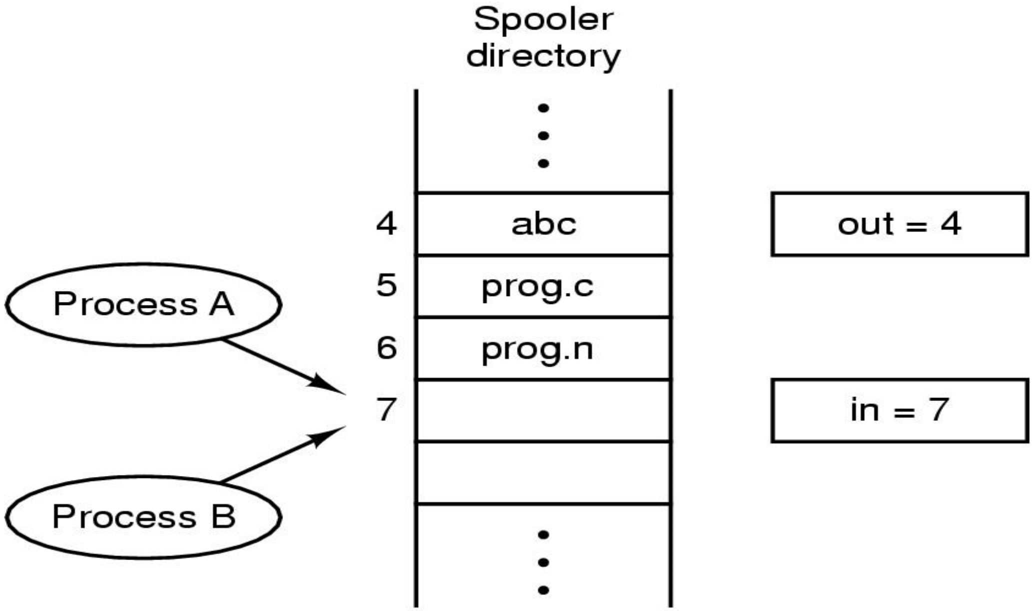
Problem of shared data

- Concurrent access to shared data may result in data inconsistency
- Maintaining data consistency requires mechanisms to ensure the orderly execution of cooperating processes
- Need of mechanism for processes to communicate and to synchronize their actions

ATM Bank account Internet banking

Race Conditions

- Two processes want to access shared memory at same time and the final result depends who runs precisely, are called **race condition**
- **Mutual exclusion** is the way to prohibit more than one process from accessing to shared data at the same time



Critical Regions (1)

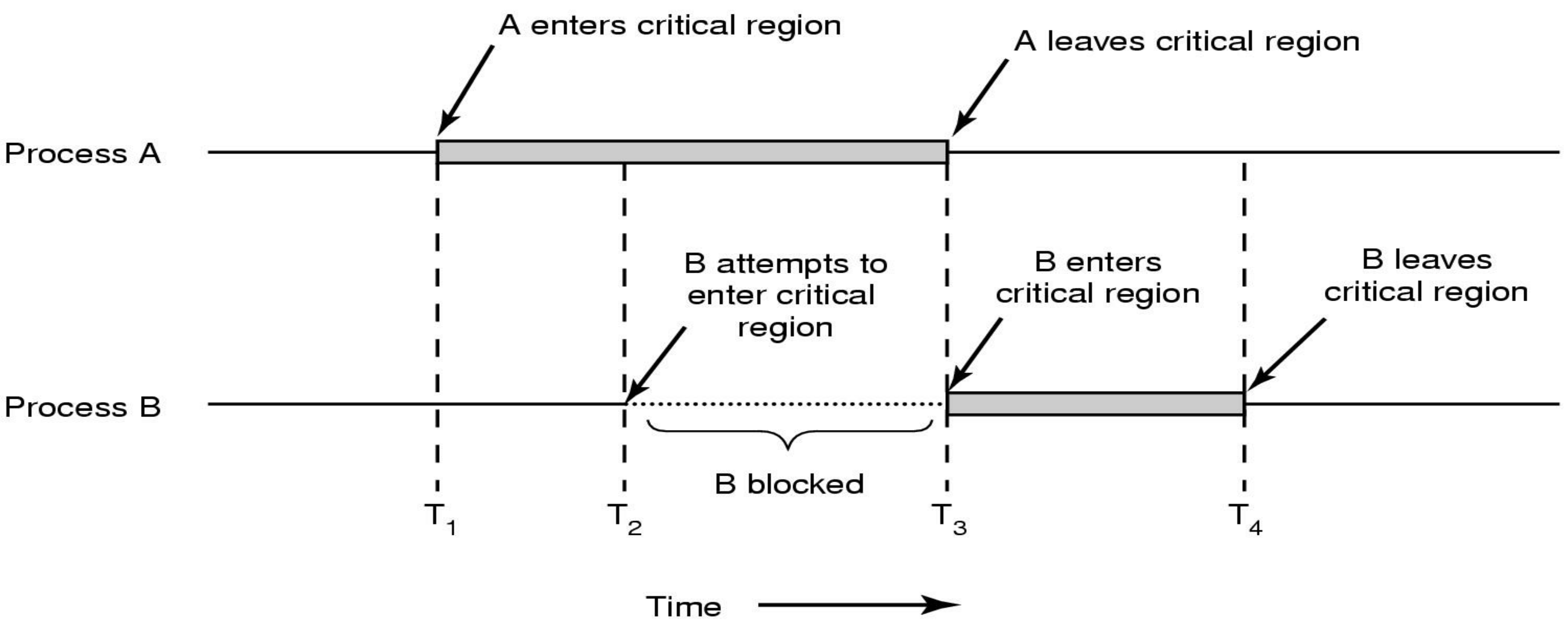
The Part of the program where the shared memory is accessed is called **Critical Regions (Critical Section)**

Four conditions to provide mutual exclusion

1. No two processes simultaneously in critical region
2. No assumptions made about speeds or numbers of CPUs
3. No process running outside its critical region may block another process
4. No process must wait forever to enter its critical region

Critical Regions (2)

Mutual exclusion using critical regions (Example)



Solution: Mutual exclusion with Busy waiting

- Software proposal
  - Lock Variables
  - Strict Alternation
  - Peterson's Solution
- Hardware proposal
  - Disabling Interrupts
  - The TSL Instruction

Mutual exclusion with Busy waiting

Software Proposal 1: Lock Variables

```
int lock = 0
```

P0

```
NonCS;

while (lock == 1); // wait
lock = 1;

CS;

lock = 0;

NonCS;
```

P1

```
NonCS;

while (lock == 1); // wait
lock = 1;

CS;

lock = 0;

NonCS;
```

Mutual exclusion with Busy waiting

Software Proposal 1: Event

int lock = 0

P0

NonCS;

```
while (lock == 1); // wait
lock = 1;
```

CS;

```
lock = 0;
```

NonCS;

P1

NonCS;

```
while (lock == 1); // wait
lock = 1;
```

CS;

```
lock = 0;
```

NonCS;

Mutual exclusion with Busy waiting

Software Proposal 2: Strict Alternation

```
int turn = 1
```

P0

```
NonCS;

while (turn !=0); // wait

CS;

turn = 1;

NonCS;
```

P1

```
NonCS;

while (turn != 1); // wait

CS;

turn = 0;

NonCS;
```

Mutual exclusion with Busy waiting
Software Proposal 2: Strict Alternation

- Only 2 processes
- Responsibility Mutual Exclusion
  – One variable "*turn*“, one process “*turn*” come in CS at the moment.

Mutual exclusion with Busy waiting

Software Proposal 3: Peterson's Solution

- int turn;
- int interest[2] = FALSE;

```
Pi                NonCS;

    j = 1 - i;
    interest[i] = TRUE;
    turn = j;
    while (turn==j && interest[j]==TRUE);

                  CS;

    interest[i] = FALSE;

                  NonCS;
```

Mutual exclusion with Busy waiting

Software Proposal 3: Peterson's Solution

Pj

NonCS;

```
i = 1 - j;
interest[j] = TRUE;
turn = i;
while (turn==i && interest[i]==TRUE);
```

CS;

```
interest[j] = FALSE;
```

NonCS;

Mutual exclusion with Busy waiting

Comment for Software Proposal 3: Peterson's Solution

- Satisfy 3 conditions:
  - Mutual Exclusion
    - Pi can enter CS when *interes[j] == F, or turn == i*
    - If both want to come back, because *turn* can only receive value 0 or 1, so one process enter CS
  - Progress
    - Using 2 variables distinct *interest[i]* ==> opposing cannot lock
  - Bounded Wait: both *interest[i]* and *turn* change value
- Not extend into N processes

Mutual exclusion with Busy waiting

Comment for Busy-Waiting solutions

- Don't need system's support
- Hard to extend
- Solution 1 is better when *atomicity* is supported

Busy waiting – Hardware Proposal

- Software proposal
  - Lock Variables
  - Strict Alternation
  - Peterson's Solution
- Hardware proposal
  - Disabling Interrupts
  - The TSL Instruction

Busy waiting – Hardware Proposal 1: Disabling Interrupt

NonCS;

$\boxed{\text{Disable Interrupt;}}$

CS;

$\boxed{\text{Enable Interrupt;}}$

NonCS;

- Disable Interrupt: prohibit all interrupts, including spin interrupt
- Enable Interrupt: permit interrupt

Hardware proposal 1: Disable Interrupt

- Not be careful
  - If process is locked in CS?
    - System Halt
  - Permit process use command privileges
    - Danger!
- System with N CPUs?
  - Don't ensure Mutual Exclusion

Hardware proposal 1: TSL Instruction

- CPU support primitive Test and Set Lock
  - Return a variable's current value, set variable to true value
  - Cannot divide up to perform (Atomic)

```
TSL (boolean &target)
{
        TSL = target;
        target = TRUE;
}
```

Applied TSL

int lock = 0

Pi

NonCS;

```
while (TSL(lock)); // wait
```

CS;

```
lock = 0;
```

NonCS;

Comment for hardware solutions in Busy-Waiting

- Necessary hardware mechanism's support
  - Not easy with n-CPUs system
- Easily extend to N processes

Comment for hardware solutions in Busy-Waiting

- Using CPU not effectively
  - Constantly test condition when wait for coming in CS
- Overcome
  - Lock processes that not enough condition to come in CS, concede CPU to other process
    - Using Scheduler
    - Wait and See...

Synchronous solution

- Sleep & Wakeup
  - Semaphore
  - Message passing

"Sleep & Wake up" solution

```
if not Sleep();

CS;

Wakeup(somebody);
```

- Give up CPU when not come in CS
- When CS is empty, will be waken up to come in CS
- Need support of OS
  - Because of changing status of process

"Sleep & Wake up" solution: Idea

- OS support 2 primitive:
  - Sleep(): System call receives blocked status
  - WakeUp(P): P process receive ready status
- Application
  - After checking condition, coming in CS or calling Sleep() depend on result of checking
  - Process that using CS before, will wake up processes blocked before

Apply Sleep() and Wakeup()

- int busy;
- int blocked;

```
if (busy) {
                blocked = blocked + 1;
                Sleep();
        }
else busy = 1;
```

CS;

```
busy = 0;
        if(blocked) {           WakeUp(P);
                                blocked = blocked - 1;
                    }
```

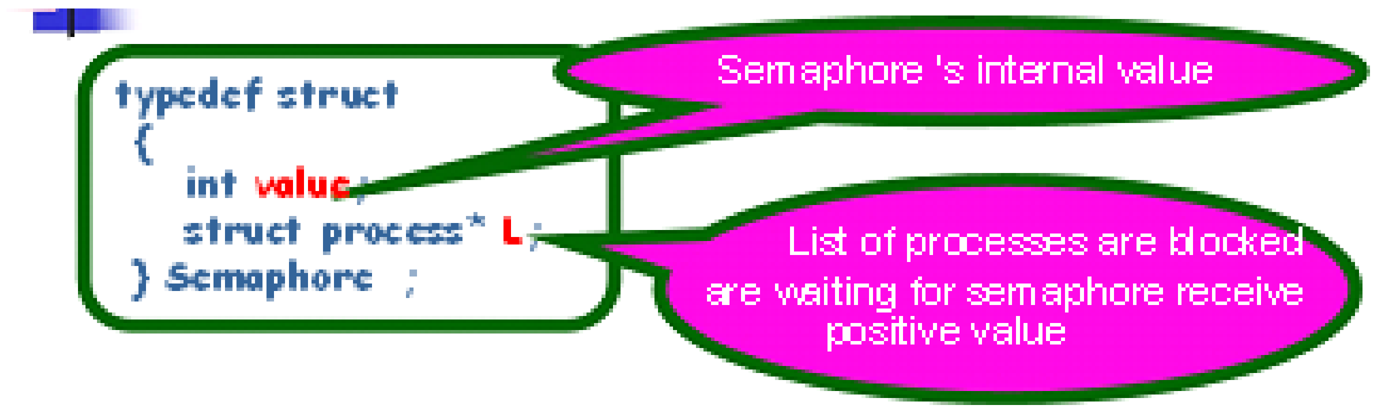
Problem with Sleep & WakeUp

- Reason:
  - Checking condition and giving up CPU can be broken
  - Lock variable is not protected

Semaphore

- Suggested by Dijkstra, 1965
- Properties: Semaphore s;
  - Unique value
  - Manipulate with 2 primitives:
    - Down(s)
    - Up(s)
  - Down and Up primitives excuted cannot divide up

Install Semaphore (Sleep & Wakeup)



- Semaphore: similar to resource
  - Processes "request" semaphore: call Down(s)
    - If Down(s) is not finished: resource is not allocated
      - Blocked, insert to s.L
- Need OS's support
  - Sleep() & Wakeup()

Install Semaphore (Sleep & Wakeup)

```
Down (S)
{
  S.value --;
   if S.value < 0
   {
     Add(P,S.L);
      Sleep();
   }
}
```

```
Up(S)
{
  S.value ++;
  if S.value ≤ 0
   {
     Remove(P,S.L);
      Wakeup(P);
    }
}
```

Using Semaphore

Semaphore s = 1

P_i

Down (s)

CS;

Up(s)

Semaphore s = 0

P_1 :

Job1;

Up(s)

P_2:

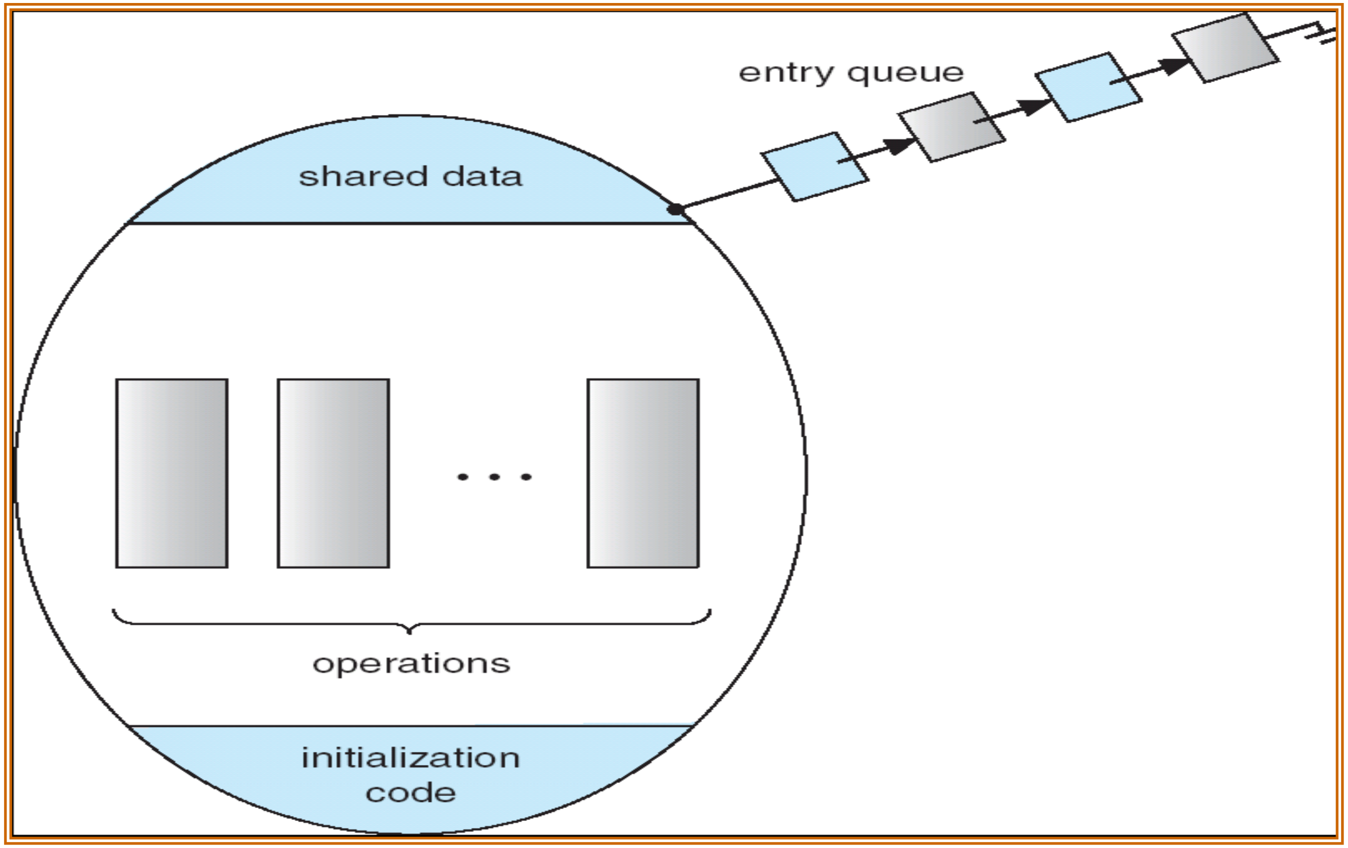
Down (s);

Job2;

Monitor

- Hoare (1974) & Brinch (1975)
- Synchronous mechanism is provided by programming language
  - Support with functions, such as Semaphore
  - Easier for using and detecting than Semaphore
    - Ensure Mutual Exclusion automatically
    - Using condition variable to perform Synchronization

Monitor: structure



Monitor: structure

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

Using Monitor

```
Monitor      M
<resource type> RC;
Function   AccessMutual
                CS; // access RC
```

```
Pi
 M.AccessMutual(); //CS
```

```
Monitor   M
Condition  c;
Function   F1
         Job1;
         Signal(c);
Function F2
         Wait(c);
         Job2;
```

```
P1 :
M.F1();
```

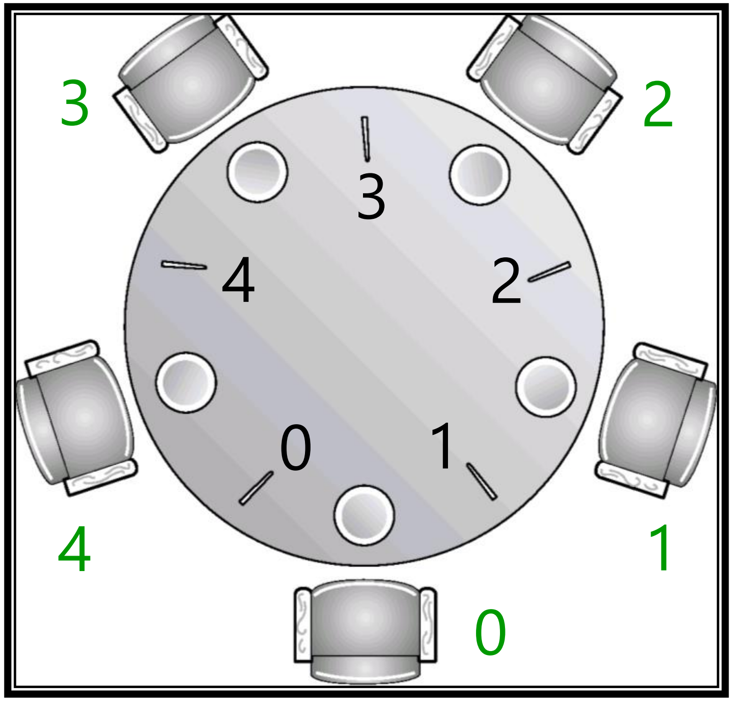
```
P2:
M.F2();
```

Message Passing

- Processes must name each other explicitly:
  - **send** (*P*, *message*) – send a message to process P
  - **receive**(*Q*, *message*) – receive a message from process Q
- Properties of communication link
  - Links are established automatically
  - A link is associated with exactly one pair of communicating processes
  - Between each pair there exists exactly one link
  - The link may be unidirectional, but is usually bi-directional

Classical Problems of Synchronization

- Bounded-Buffer Problem
  (Producer-Consumer Problem)
- Readers and Writers Problem
- Dining-Philosophers
  Problem



2.4 Scheduling

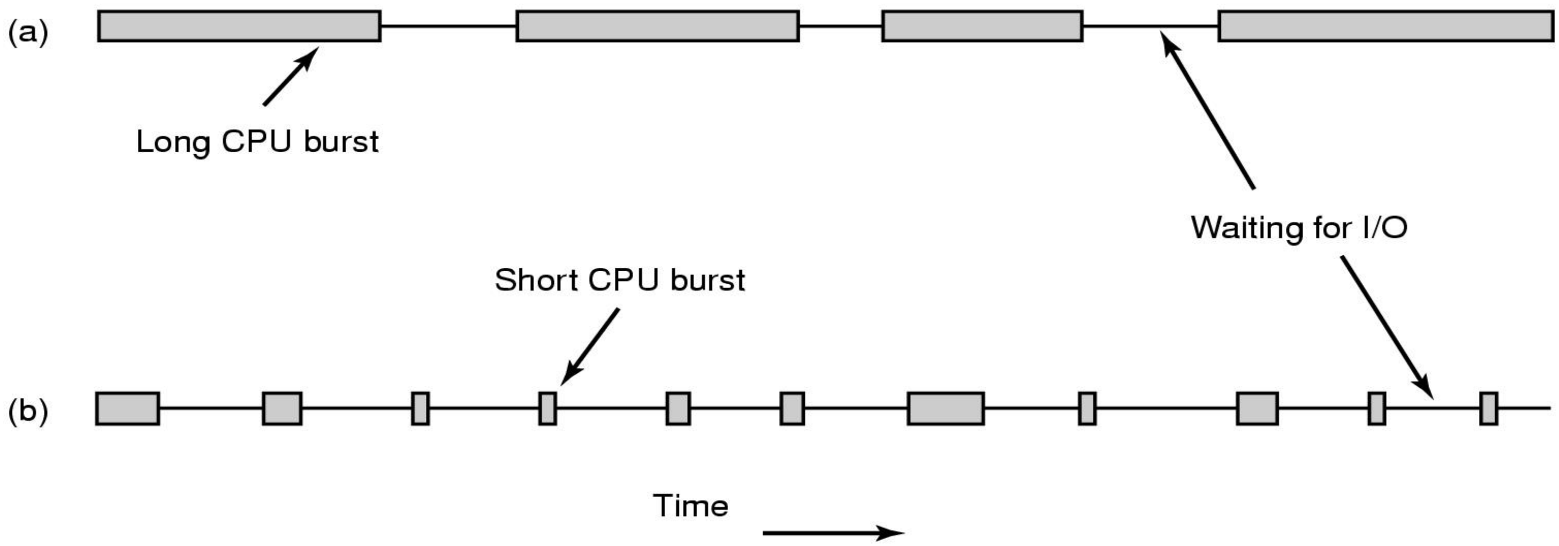
Scheduling

Introduction to Scheduling (1)

- Maximum CPU utilization obtained with multiprogramming
- CPU–I/O Burst Cycle – Process execution consists of a *cycle* of CPU execution and I/O wait
- CPU burst distribution

Scheduling

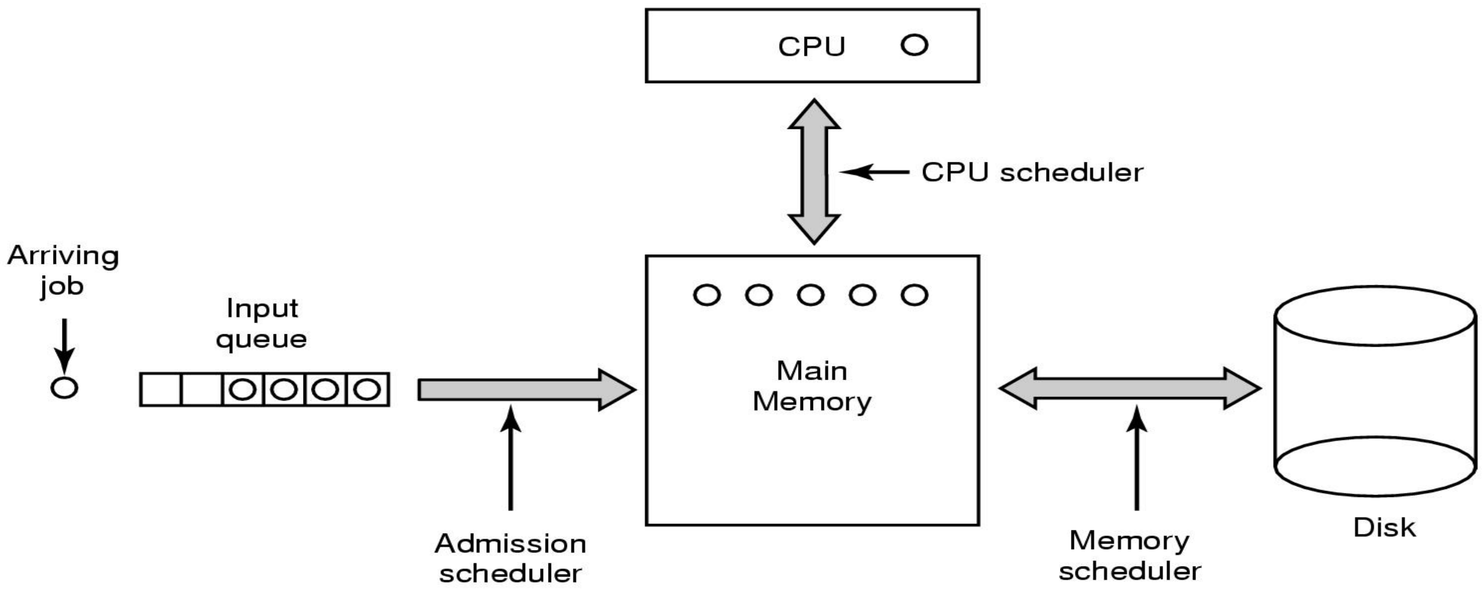
Introduction to Scheduling (2)



- Bursts of CPU usage alternate with periods of I/O wait
  - a CPU-bound process
  - an I/O-bound process

Scheduling

Introduction to Scheduling (3)



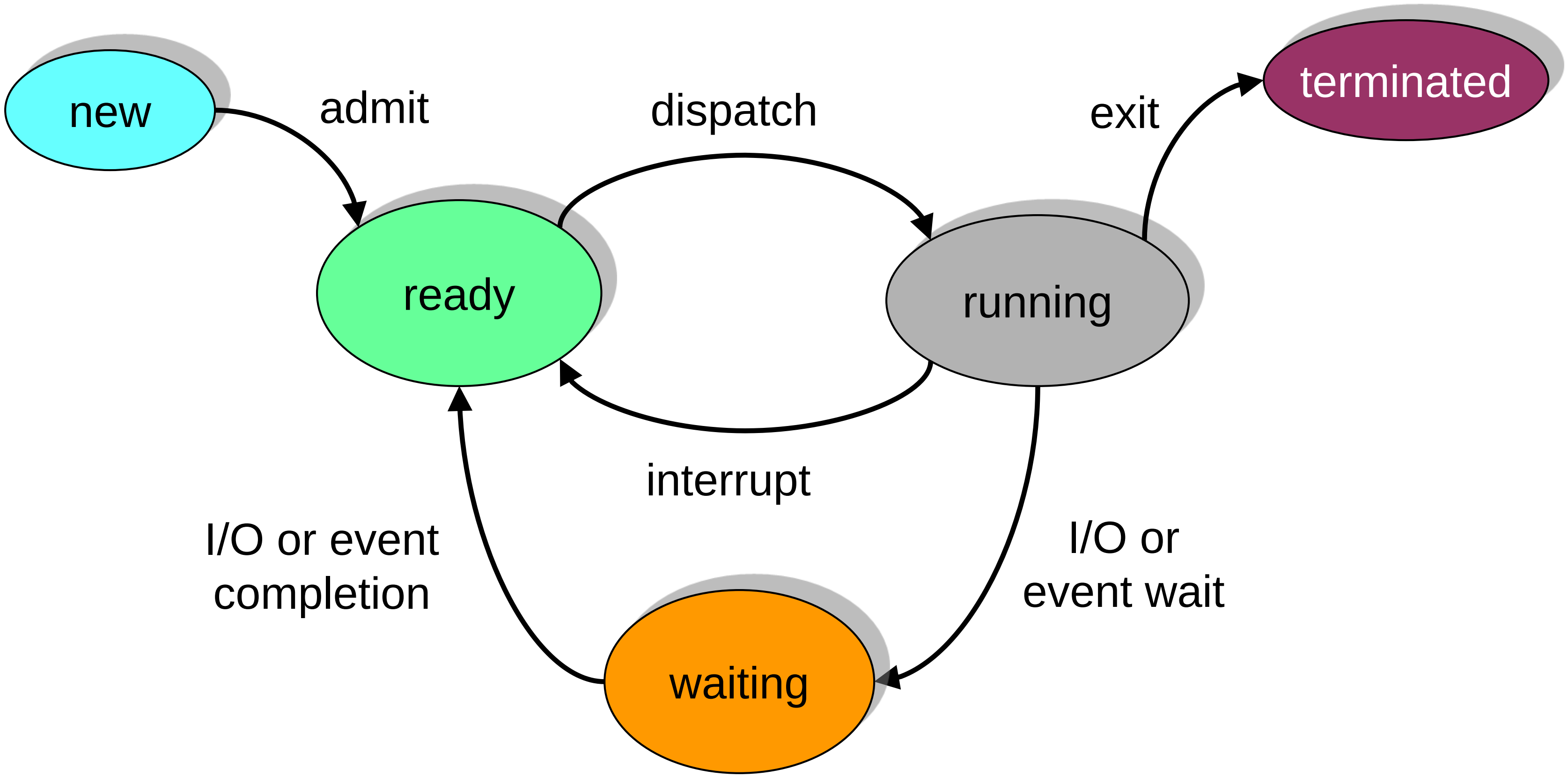
Three level scheduling

Scheduling
Introduction to Scheduling (4)

- Selects from among the processes in memory that are ready to execute, and allocates the CPU to one of them
- CPU scheduling decisions may take place when a process:
  1. Switches from running to waiting state
  2. Switches from running to ready state
  3. Switches from waiting or new process is created to ready
  4. Terminates
- *Nonpreemptive* scheduling algorithm picks process and let it run until it blocks or until it voluntarily releases the CPU
- *preemptive* scheduling algorithm picks process and let it run for a maximum of fix time

Scheduling

Introduction to Scheduling (5)



Scheduling
Introduction to Scheduling (6)

Scheduling Criteria

- CPU utilization – keep the CPU as busy as possible
- Throughput – # of processes that complete their execution per time unit
- Turnaround time – amount of time to execute a particular process
- Waiting time – amount of time a process has been waiting in the ready queue
- Response time – amount of time it takes from when a request was submitted until the first response is produced, **not** output (for time-sharing environment)

Scheduling

Introduction to Scheduling (7)

Optimization Criteria

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

Scheduling

Introduction to Scheduling (8)

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

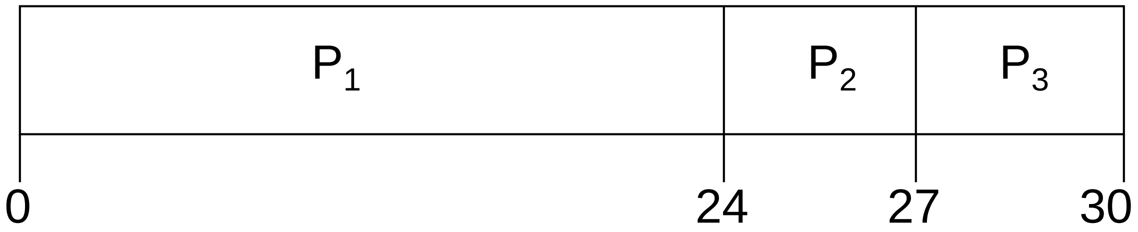
Meeting deadlines - avoid losing data
Predictability - avoid quality degradation in multimedia systems

Scheduling
Scheduling in Batch Systems (1)

First-Come, First-Served (FCFS) Scheduling

Process	Burst Time
P_1	24
P_2	3
P_3	3

- Suppose that the processes arrive in the order: P_1 , P_2 , P_3
The Gantt Chart for the schedule is:



- Waiting time for $P_1 = 0$; $P_2 = 24$; $P_3 = 27$
- Average waiting time: $(0 + 24 + 27)/3 = 17$

Scheduling

Scheduling in Batch Systems (2)

FCFS Scheduling (Cont.)

Suppose that the processes arrive 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**

Scheduling
Scheduling in Batch Systems (3)

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
- Two schemes:
  - nonpreemptive – once CPU given to the process it cannot be preempted until 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 know as the Shortest-Remaining-Time-First (SRTF)
- SJF is optimal – gives minimum average waiting time for a given set of processes

Scheduling

Scheduling in Batch Systems (4)

An example of shortest job first scheduling

Scheduling
Scheduling in Interactive Systems (1)

(a)

(b)

- Round Robin Scheduling
  - list of runnable processes (a)
  - list of runnable processes after B uses up its quantum (b)

Scheduling

Scheduling in Interactive Systems (2)

Round Robin (RR)`

- Each process gets a small unit of CPU time (*time quantum*), usually 10-100 milliseconds. After this time has elapsed, the process is preempted and added to the end of the ready queue.
- 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.
- Performance
  - q large $\Rightarrow$ FIFO
  - q small $\Rightarrow$ q must be large with respect to context switch, otherwise overhead is too high

Scheduling

Scheduling in Interactive Systems (3)

Example of RR with Time Quantum = 20

Process	Burst Time
P_1	53
P_2	17
P_3	68
P_4	24

- The Gantt chart is:

P_1	P_2	P_3	P_4	P_1	P_3	P_4	P_1	P_3	P_3

0 20 37 57 77 97 117 121 134 154 162

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

Scheduling

Scheduling in Interactive Systems (3)

Example of RR with Time Quantum = 20

Process	Burst Time
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*

Scheduling
Scheduling in Interactive Systems (4)

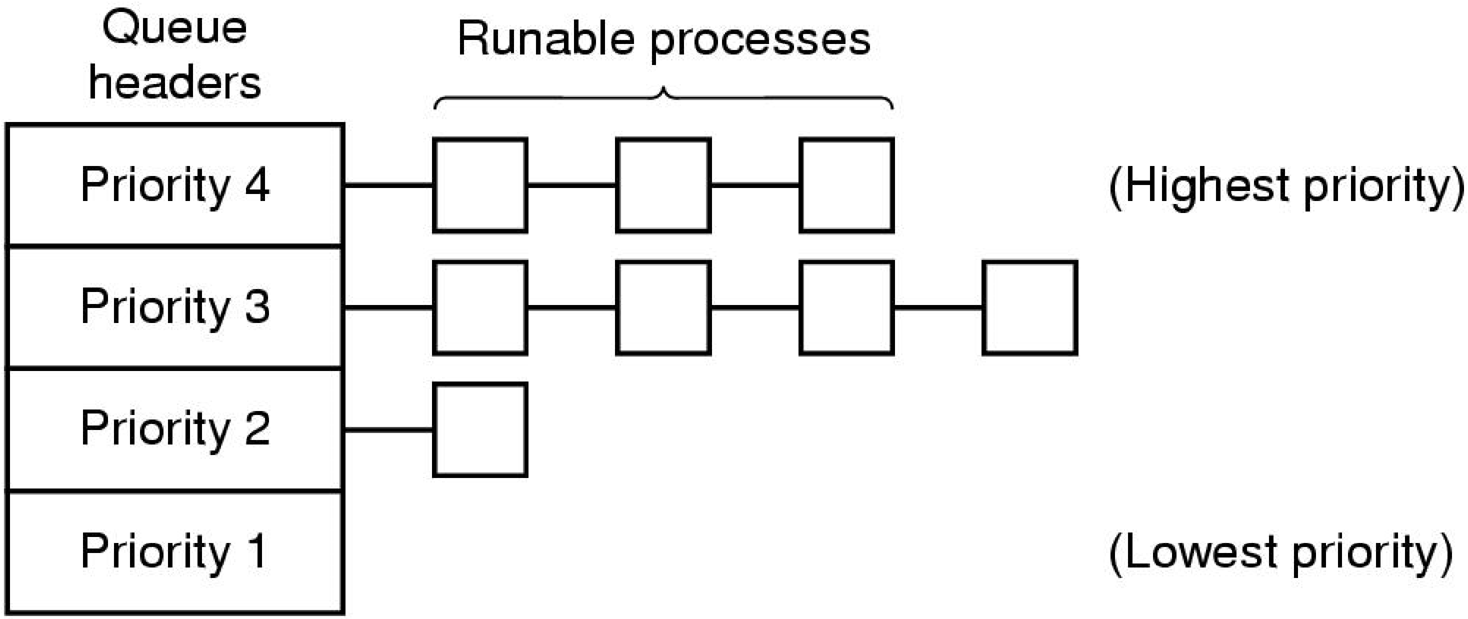
Priority Scheduling: A priority number (integer) is associated with each process

- The CPU is allocated to the process with the highest priority
- Preemptive
- nonpreemptive

- SJF is a priority scheduling where priority is the predicted next CPU burst time
- Problem ≡ Starvation – low priority processes may never execute
- Solution ≡ Aging – as time progresses increase the priority of the process

Scheduling
Scheduling in Interactive Systems (5)

A scheduling algorithm with four priority classes



Scheduling
Scheduling in Real-Time Systems (1)

- *Hard real-time* systems – required to complete a critical task within a guaranteed amount of time
- *Soft real-time* computing – requires that critical processes receive priority over less fortunate ones

Scheduling

Scheduling in Real-Time Systems(2)

Schedulable real-time system

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

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

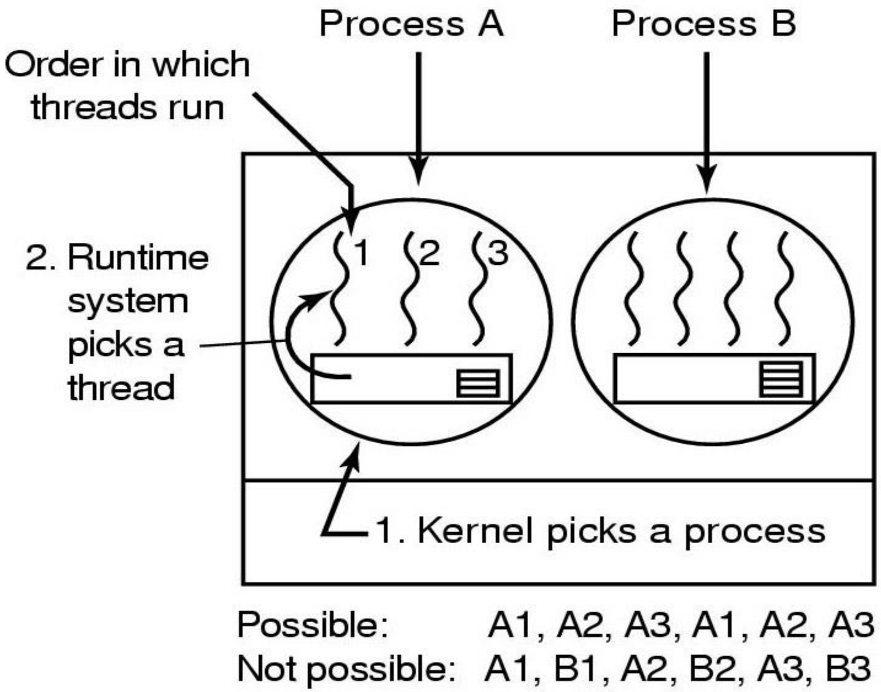
Scheduling
Policy versus Mechanism

- Separate what is allowed to be done with how it is done
  - a process knows which of its children threads are important and need priority
- Scheduling algorithm parameterized
  - mechanism in the kernel
- Parameters filled in by user processes
  - policy set by user process

Scheduling
Thread Scheduling (1)

- Local Scheduling – How the threads library decides which thread to put onto an available
- Global Scheduling – How the kernel decides which kernel thread to run next

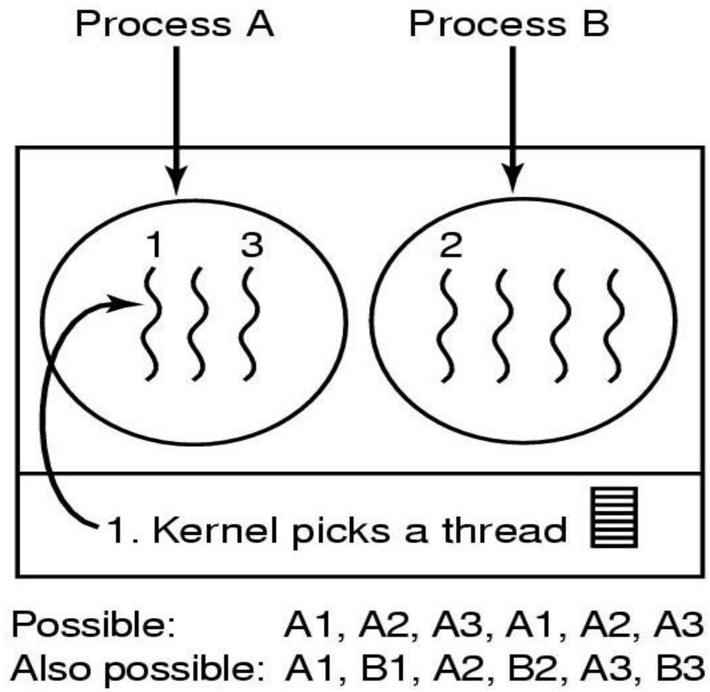
Scheduling
Thread Scheduling (2)



Possible scheduling of user-level threads

- 50-msec process quantum
- threads run 5 msec/CPU burst

Scheduling
Thread Scheduling (3)



Possible scheduling of kernel-level threads

- 50-msec process quantum
- threads run 5 msec/CPU burst