Processes

Reading: Chapter 3 the Textbook

What is a Process?

The program in execution.

A program is an inanimate entity; only when a processor breathes life into it does it become the active entity, we call a process.

A process is the unit of work in a modern time-sharing system

Programs and Processes

A process is different than a program.

Consider the following analogy

Scenario-1: A computer scientist is baking a birthday cake for his daughter

- Computer scientist - CPU
- Birthday cake recipe - program
- Ingredients - input data
- Activities: - processes
  - reading the recipe
  - fetching the ingradients
  - backing the cake

Programs and Processes

Scenario-2: Scientist's son comes running in crying, saying he has been stung by a bee.

- Scientist records where he was in the recipe
- Reach first aid book and materials
- Follow the first aid action (high priority job)
- On completion of aid, cake baking starts again from where it was left

- the state of running process saved
- Another process fetched
- Processor switched for new process
- Completion of high priority job & return back to the last one

A process is an activity of some kind, it has program, input, output and state.

Process States

A process goes through a series of discrete process states.

Running state:

- Process executing on CPU.
- Only one process at a time.

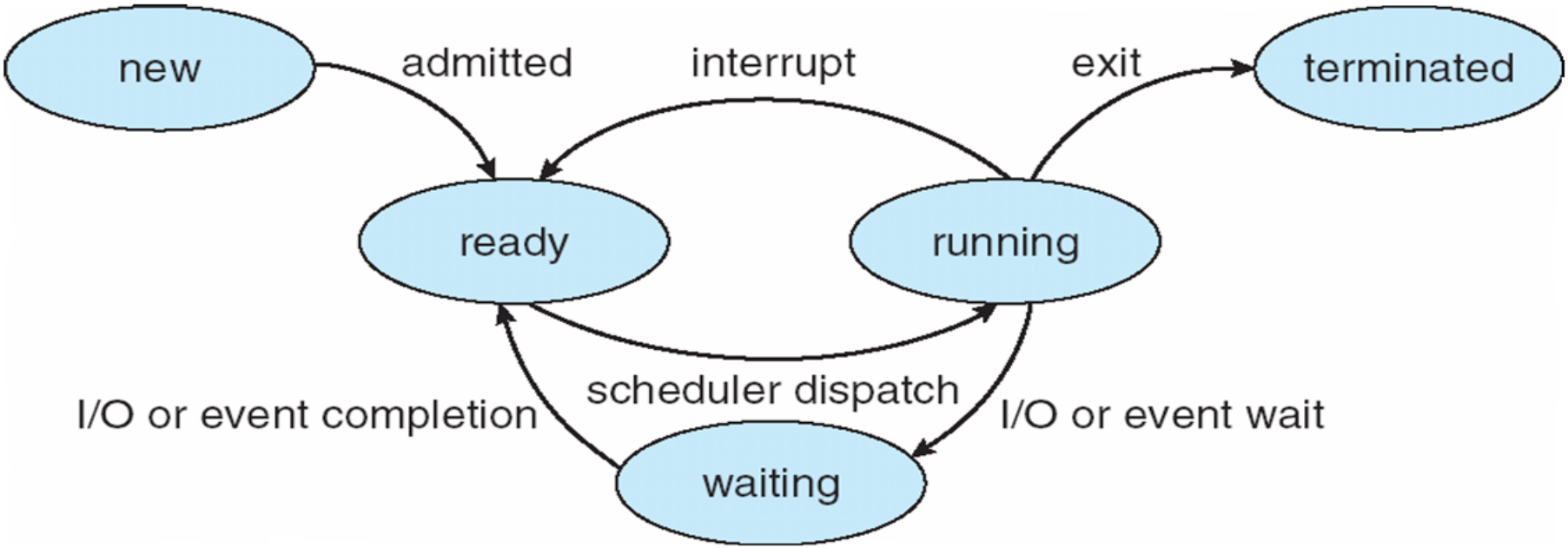
Ready state:

- Process that is not allowed to CPU but is ready to run.
- a list of processes ordered based on priority.

Waiting state:

- Process that is waiting for some event to happen (e. g. I/O completion events).
- a list of processes (no clear priority).

Process States



Process state transitions diagram

Process State Transitions

When a job is admitted to the system, a corresponding process is created and normally inserted at the back of the ready list.

When the CPU becomes available, the process is said to make a state transition from ready to running.

ready -> running.

To prevent any one process from monopolizing the CPU, OS specify a time period (quantum) for the process. When the quantum expire or interrupt to CPU makes state transition running to ready.

running -> ready.

When the process requires an I/O operation before quantum expire or interrupt, the process voluntarily relinquishes the CPU and changed to the wait state.

running -> wait.

When an I/O operation completes. The process make the transition from wait state to ready state.

wait -> *ready.*

When process finished execution it will terminated

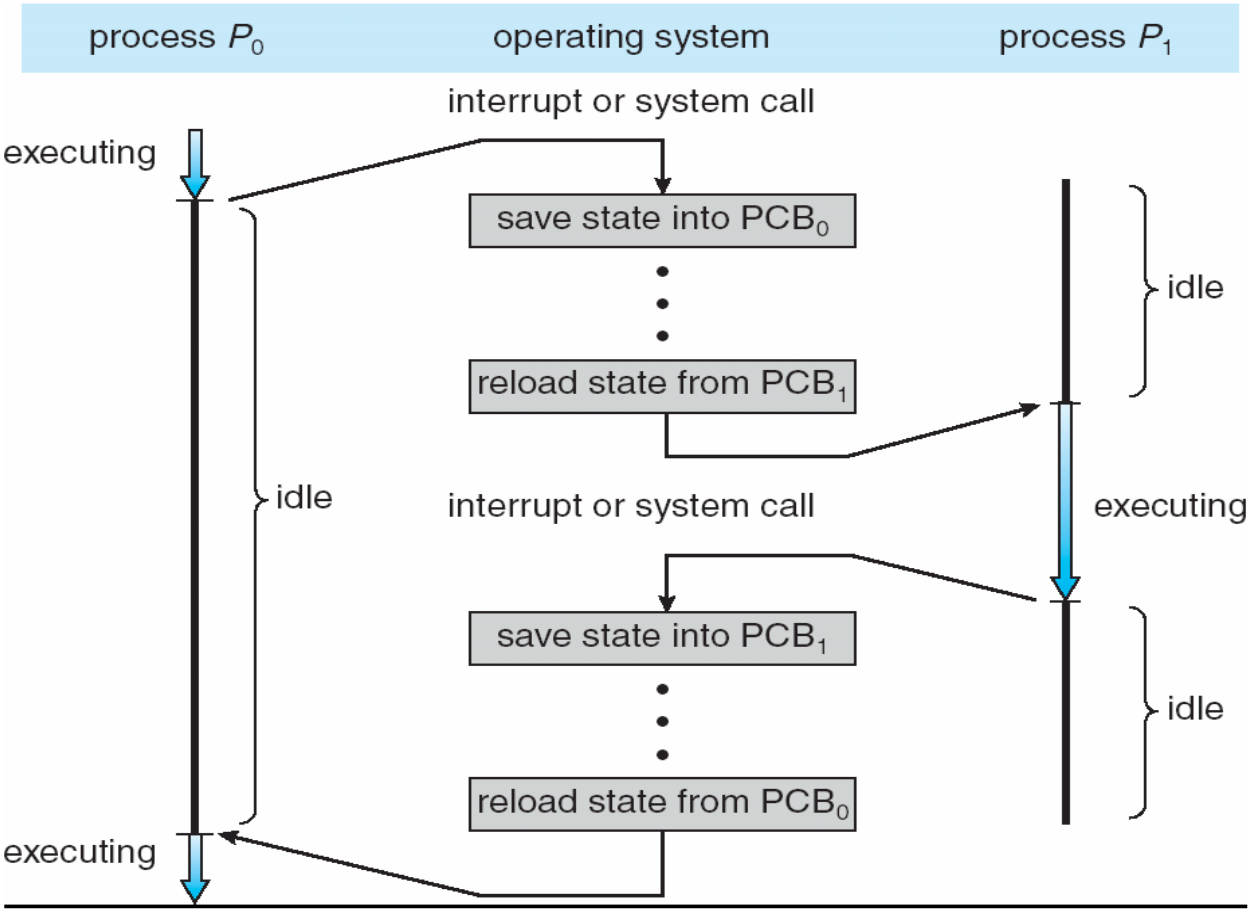
Process Control Block

Process must be saved when the process is switched from one state to another so that it can be restarted later as it had never been stopped - **Context-switch**.

The PCB is the data structure containing certain important information about the process -also called process table or processor descripter.

- *Process state*: running, ready, blocked.
- *Program counter*: Address of next instruction for the process.
- *Registers*: Stack pointer, accumulator, PSW etc.
- *Scheduling information*: Process priority, pointer to scheduling queue etc.
- *Memory-allocation*: value of base and limit register, page table, segment table etc.
- *Accounting information*: time limit, process numbers etc.
- *Status information:* list of I/O devices, list of open files etc.

Process Control Block



Operations on Processes

The processes in the system can execute concurrently, and they must be created and deleted dynamically. OS provide the mechanism for process *creation* and *termination*.

- Process Creation.
- Process Termination.

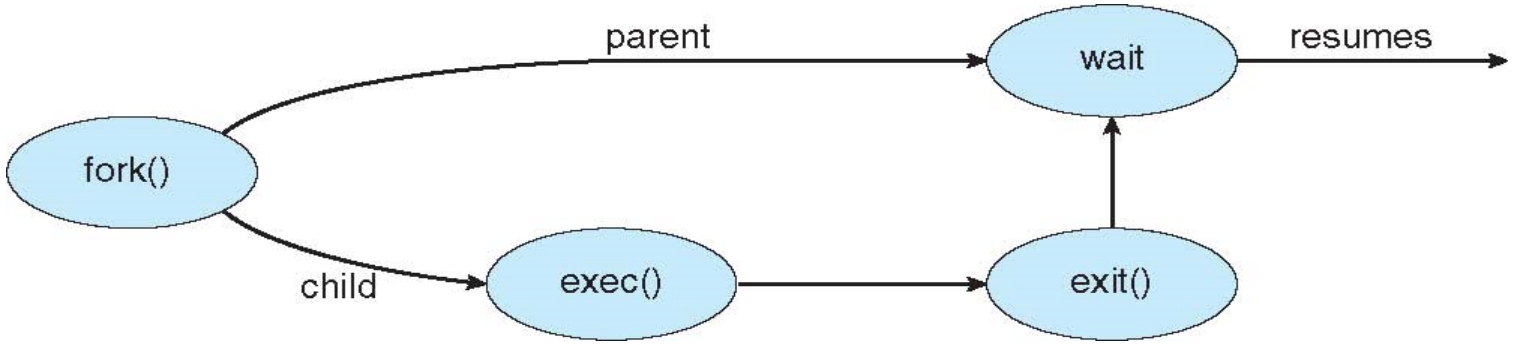
Process Creation

There are four principal events that cause the the process to be created:

- System initialization.
- Execution of a process creation system call.
- User request to create a new process.
- Initiation of a batch job.

A process may create several new processes forming tree of process during the course of execution. The creating process is called a parent process, where as the new processes are called children of that process.

UNIX examples: fork system call creates new process exec system call



Process Creation

Two ways to create a new process

- Build a new one from scratch
  - Load specified code and data into memory.
  - Create and initialize PCB.
  - Put processes on the ready list.
- Colon an existing one (e.g. Unix fork() syscall)
  - Stop the current process and save its state.
  - Make copy of code, data, stack, and PCB.
  - Add new process PCB to ready list.

Unix Process Creation Example

```
#include<stdio.h>
int main(int argc, char *argv[])
        {
        int pid;
                pid = fork();  /* create new process */
                if(pid < 0) {  /* error occurred */
                        fprintf(stderr, "fork failed");
                        exit(-1);
                                }
                else if(pid == 0){      /* child process */
                        execlp("/bin/ls", "ls", Null); }
                else {                                  /* parent process */
                        wait(Null);
                        printf("Child Complete");
                        exit(0);
                        }
                return 0;
        }
```

Process Termination

Process are terminated on the following conditions

1. Normal exit.
2. Error exit.
3. Fatal error.
4. Killed by another process.

Example:

In Unix the normal exit is done by calling a *exit* system call. The process return data (output) to its parent process via the *wait* system call. *kill* system call is used to kill other process.

Inter Processes Communication

How one process can pass the information to the another (cooperate each other)?

Process required cooperation for:

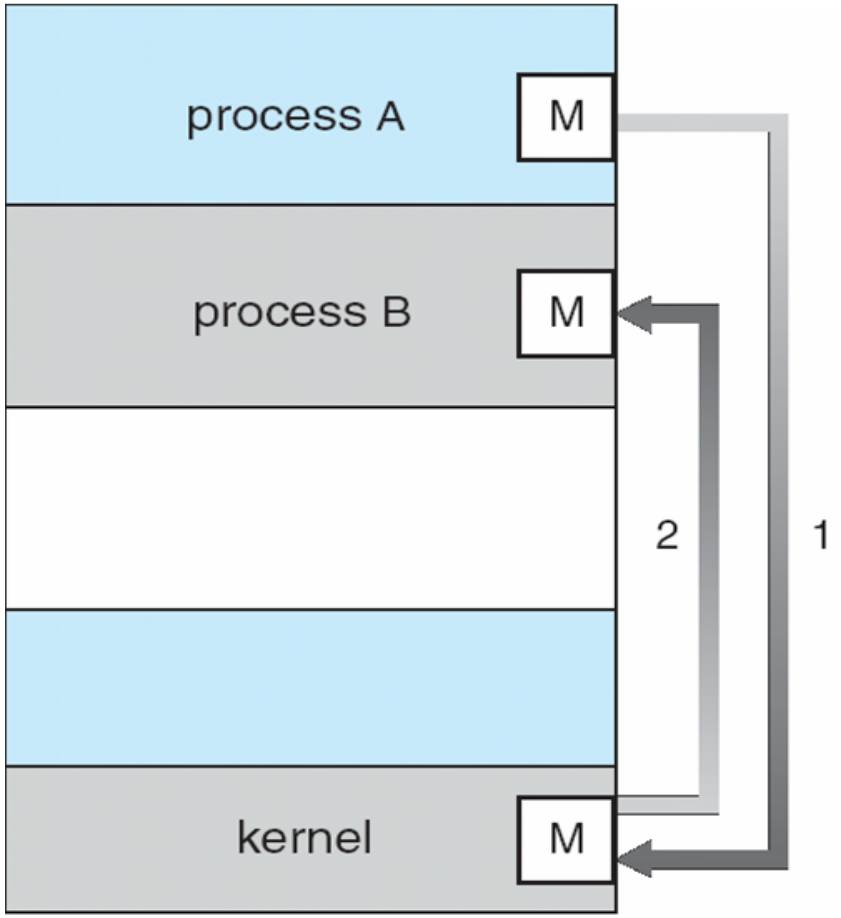
- Information Sharing
- Computation Speedup
- Modularity
- Convenience

Two models of IPC

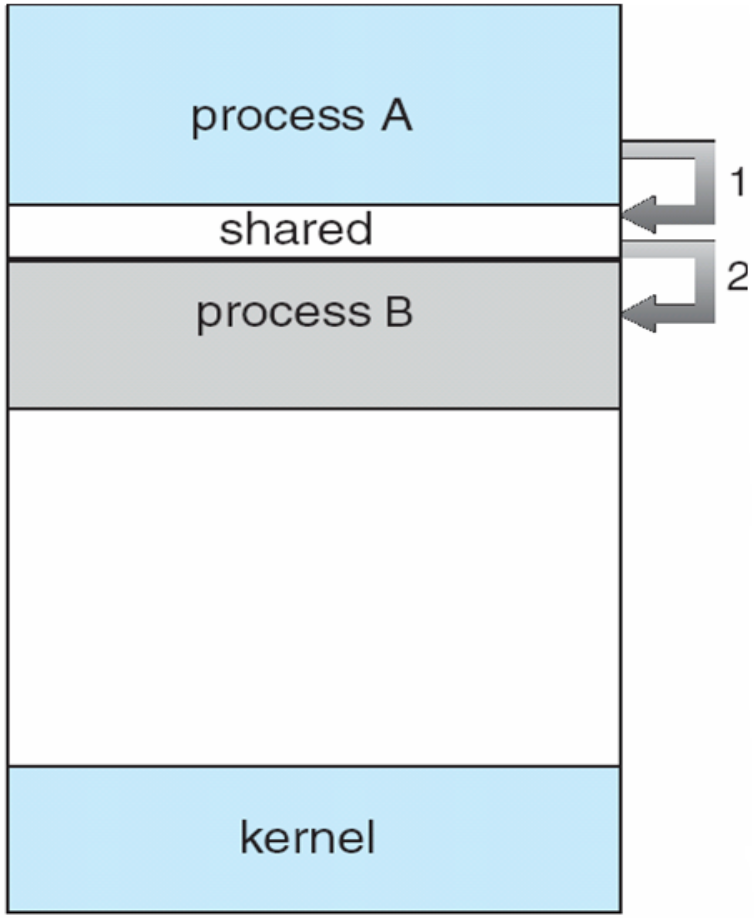
- Shared memory – exchange information by reading or writing data on shared areas.
- Message passing-processes communicate with each other without resorting to shared variables

Inter Processes Communication

Communications Models



(a) Message Passing



(b) Shared Memory

Shared memory: Example

Producer-Consumer Problem

Suppose one process, producer, is generating information that the second process, consumer, is using.

```
#define N = 100     /* number of slots in the buffer*/
 int count = 0;      /* number of item 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 */
  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 go to sleep*/
 item = remove_item(); /*take item out
                                  of buffer*/
 count = count -1; /*decrement count*/
 if (count == N-1)wakeup(producer);
       /*was buffer full ?*/
 consume_item();     /*print item*/
}
}
```

What happened when consumer try to consume the item before producer produce it?

Message Passing

IPC facility provides two operations:

- **send**(*message*) – message size fixed or variable
- **receive**(*message*)

If *P* and *Q* wish to communicate, they need to establish a *communication link* between them and exchange messages via send/receive

send (P, message) – send a message to process P
receive(Q, message) – receive a message from process Q

Used for distributed environment

Allows processes to communicate as well as synchronize the activities

Home Work

HW #2:

1. Q. No. 3.1, 3.5, 3.6, 3.11, & 3.12 from the Textbook.
2. What are disadvantages of too much multiprogramming?
3. List the definitions of process.
4. For each of the following transitions between the process states, indicate whether the transition is possible. If it is possible, give an example of one thing that would cause it.
    a) Running -> Ready
    b) Running -> Wait
    c) Wait -> Running

Multithreading

Reading: Chapter 4 of Textbook

What is Thread?

Threads, like process, are a mechanism to allow a program to do more than one thing at a time.

Conceptually, a thread (also called *lightweight process*) exists within a process (heavyweight process). Threads are a finer-grained unit of execution than processes

The term *multithreading* is used to describe the situation of allowing the multiple threads in same process.

Multithreading

All threads share the same address space, global variables, set of open file, child processes, alarms and signals etc.

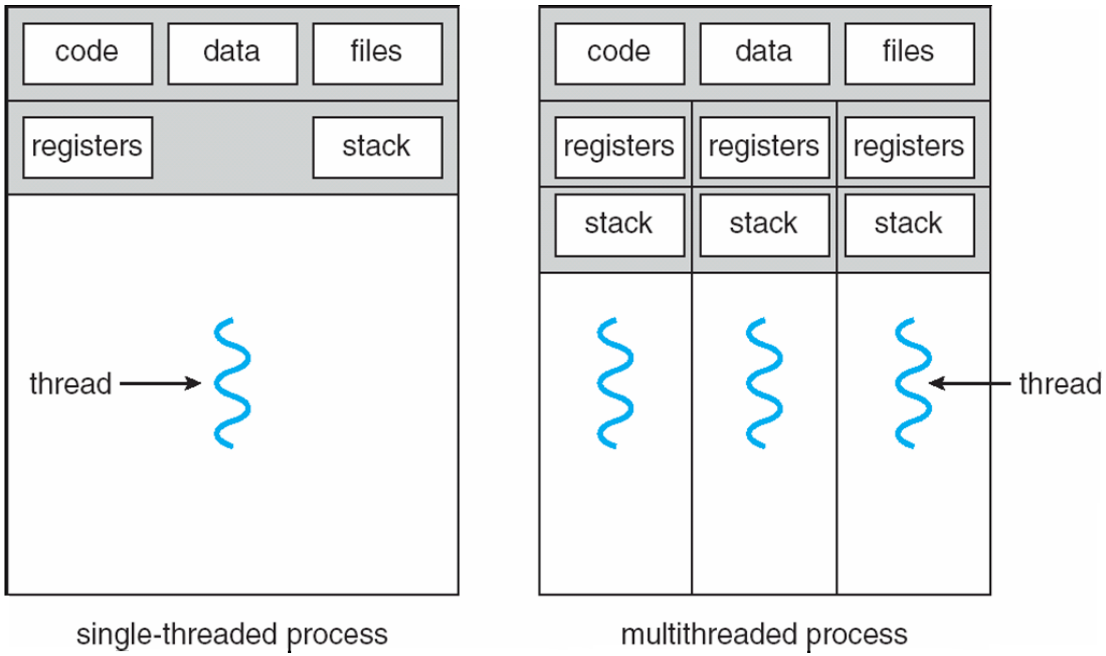


Figure (b): three threads are actually part of the same job and are actively and closely cooperating with each other.

Each thread maintain its own stack.

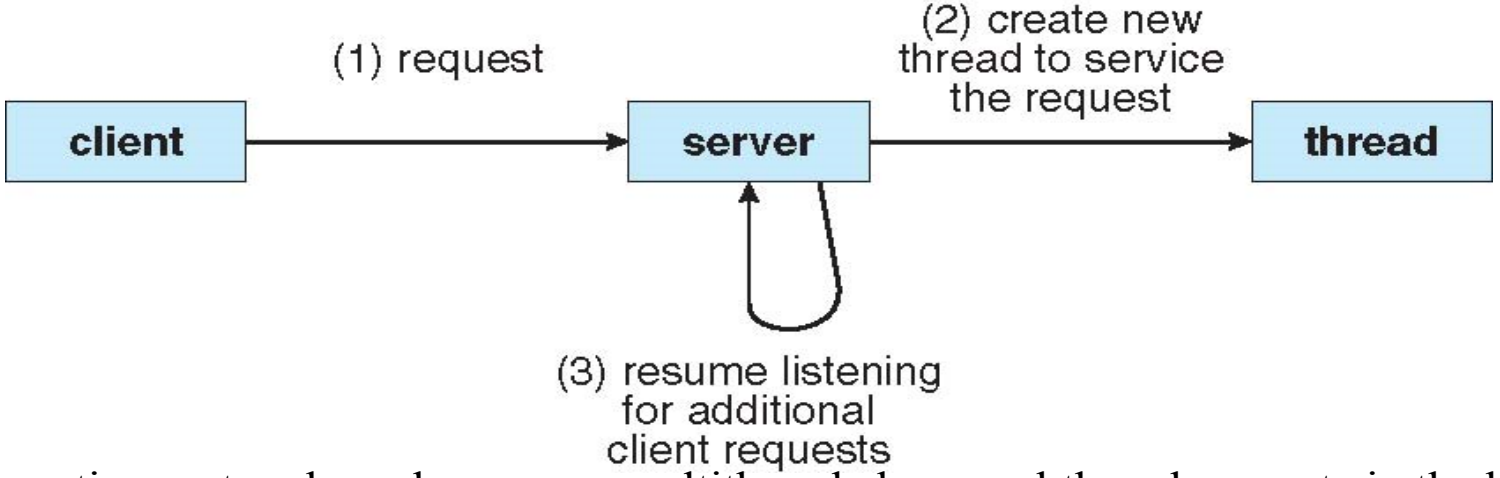
Multithreading Benefits

In modern operating system multithreading has several benefits as :

- Responsiveness – program can run continuously even if the part of it is blocked
- Resource Sharing – Share the code and data
- Economy – economic in resources and context switching
- Scalability – supports multiprocessor architecture that increase concurrency

Threads also play a vital role in remote procedure call (RPC) systems.

When a server receives a message, it services the message using a separate thread. This allows the server to service several concurrent requests



Most operating system kernels are now multithreaded; several threads operate in the kernel, and each thread performs a specific task, such as managing devices or interrupt handling.

Users and Kernel Threads

User Threads:

Thread management done by user-level threads library.

- Implemented as a library
- Library provides support for thread creation, scheduling and management with no support from the kernel.
- Fast to create
- If kernel is single threaded, blocking system calls will cause the entire process to block.

Examples: POSIX Pthreads, Win32 API threads, Java threads

Kernel Threads:

Supported by the Kernel

- Kernel performs thread creation, scheduling and management in kernel space.
- Slower to create and manage
- Blocking system calls are no problem
- Most OS's support these threads

Examples: Windows XP/2000, Solaris, Linux, Tru64 UNIX, Mac OS X

Multithreading Programs
Pthreads Example

Program demonstrate the basic Pthreads API for constructing multithreading program

```
#include <pthread.h>
#include <stdio.h>

int sum; /* this data is shared by the thread(s) */
void *runner(void *param); /* the thread */

int main(int argc, char *argv[])
{
  pthread_t tid; /* the thread identifier */
  pthread_attr_t attr; /* set of thread attributes */

  if (argc != 2) {
    fprintf(stderr,"usage: a.out <integer value>\n");
    return -1;
  }
  if (atoi(argv[1]) < 0) {
    fprintf(stderr,"%d must be >= 0\n",atoi(argv[1]));
    return -1;
  }
```

```
  /* get the default attributes */
  pthread_attr_init(&attr);
  /* create the thread */
  pthread_create(&tid,&attr,runner,argv[1]);
  /* wait for the thread to exit */
  pthread_join(tid,NULL);

  printf("sum = %d\n",sum);
}

/* The thread will begin control in this function */
void *runner(void *param)
{
  int i, upper = atoi(param);
  sum = 0;

  for (i = 1; i <= upper; i++)
    sum += i;

  pthread_exit(0);
}
```

This program calculate the summation of nonnegative integer in separate thread

Multithreading Issues

- In Unix system when *fork()* called by a program, it copy all threads parent to the child.

  What happens if the thread of parent blocked? Required two different version of fork.
- Threads share the many data structure.

  What happens if one thread closes/cancel a file while another is still reading from it?
- In Unix Signal handling problem, deliver signal to single thread to which the signal applies or to all threads of the process
- How many threads to be created when concurrent request serviced by new threads and which one in active in the system? Required thread pools
- How to manage Thread specific data? – Allows each thread to have its own copy of data
- When thread to be communicate each other, which one to be activated and communicate to kernel to maintain the correct number kernel threads? - Scheduler activations

Designing complexity. Need complex scheduling operations

Home works

HW #3:

1. Q. 4.1, 4.2, 4.3, 4.4, 4.5, 4.6, 4.9, 4.10, 4.11, 4.12 of the Textbook
2. What are the two differences between the kernel level threads and user level threads? Which one has a better performance?
3. List the differences between processes and threads.
4. What resources are used when a thread is created? How do they differ from those used when a process is created?

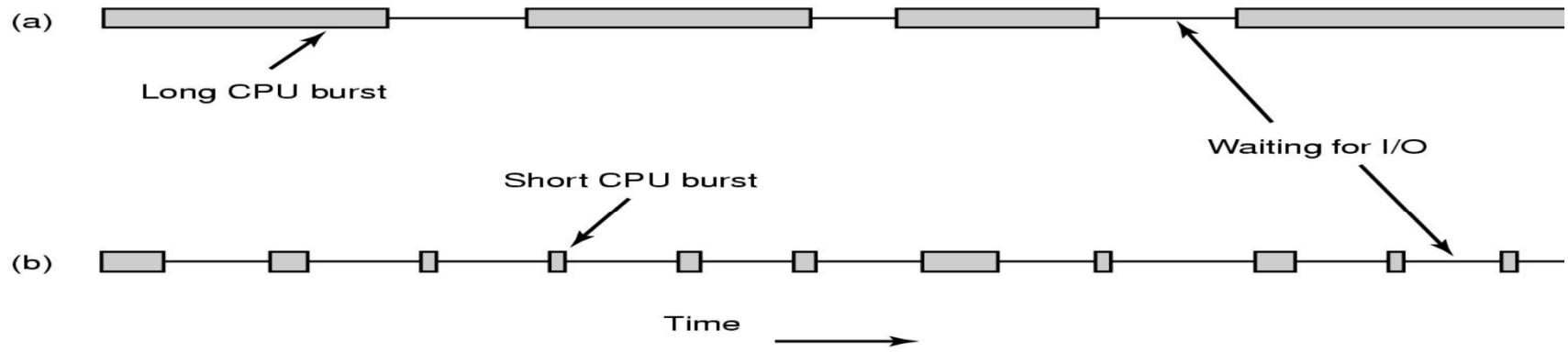
Process Scheduling

Reading: Chapter 5 of textbook

Which process is given control of the CPU and how long?

By switching the processor among the processes, the OS can make the computer more productive – basic objective for multiprogrammed operating systems

CPU-I/O Burst



Process execution consists of a cycle of CPU execution and I/O wait.

Process execution begins with a CPU burst that is followed by I/O burst, then another CPU burst, then another I/O burst.....

CPU-bound: Processes that use CPU until the quantum expire.

I/O-bound: Processes that use CPU briefly and generate I/O request.

CPU-bound processes have a long CPU-burst while I/O-bound processes have short CPU burst.

Key idea: when I/O bound process wants to run, it should get a chance quickly.

When to Schedule

1. When a new process is created.
2. When a process terminates.
3. When a process blocks on I/O, on semaphore, waiting for child termination etc.
4. When an I/O interrupt occurs.
5. When quantum expires.

Once a process has been given the CPU, it runs until blocks for I/O or termination is known as *nonpreemptive*, otherwise it is *preemptive*.

Dispatcher vs. Scheduler

Dispatcher

- Low level mechanism.
- Responsibility: Context switch
  - Save execution state of old process in PCB.
  - Load execution state of new process from PCB to registers.
  - Change the scheduling state of the process (running, ready, blocked)
  - Switch from kernel to user mode.

Scheduler

- Higher-level policy.
- Responsibility: Which process to run next.

Scheduling Criteria

The scheduler is to identify the process whose selection will results the best possible system performance.

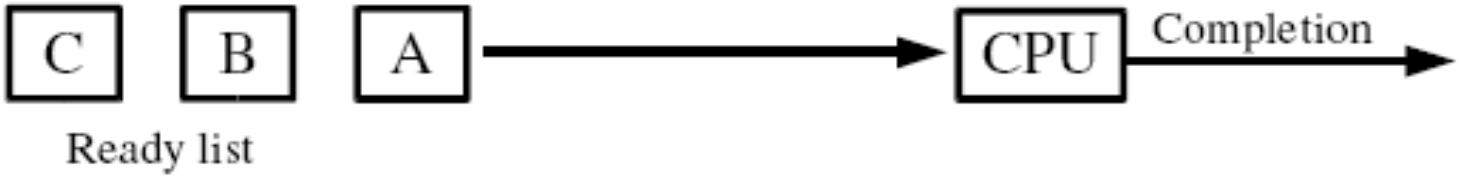
Criteria for comparing scheduling algorithms:

CPU Utilization	*Throughput*
Turnaround Time	*Waiting Time*
Response Time	*Predictability*
Fairness	*Priorities*

The scheduling policy determine the important of each criteria.
The scheduling algorithms should designed to optimizes maximum possible criteria.

Scheduling Algorithms

First-Come First-Serve (FCFS)



Processes are scheduled in the order they are received.

Once the process has the CPU, it runs to completion -Nonpreemptive.

Easily implemented, by managing a simple queue or by storing time the process was received.

Fair to all processes.

Problems:

- No guarantee of good response time.
- Large average waiting time.
- Not applicable for interactive system.

Scheduling Algorithms
Shortest Job First (SJF)

SJF selects the process with shortest expected processing time. In case of the tie FCFS scheduling is used. The processing times are known in advanced.

The decision policies are based on the CPU burst time.

Advantages:

Reduces the average waiting time over FCFS.

Favors shorts jobs at the cost of long jobs.

Problems:

Estimation of run time to completion. Accuracy?

Not applicable in timesharing system.

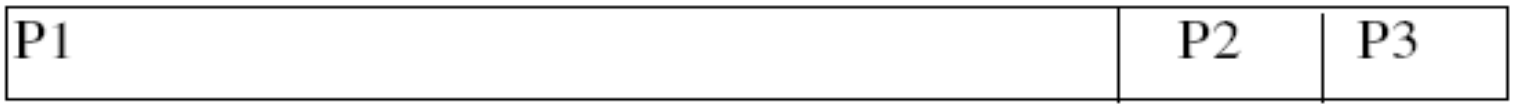
Optimal for minimizing queueing time, but impossible to implement. Tries to predict the process to schedule based on previous history

Scheduling Algorithms
SJF -Performance

Scenario: Consider the following set of processes that arrive at time 0, with length of CPU-bust time in milliseconds.

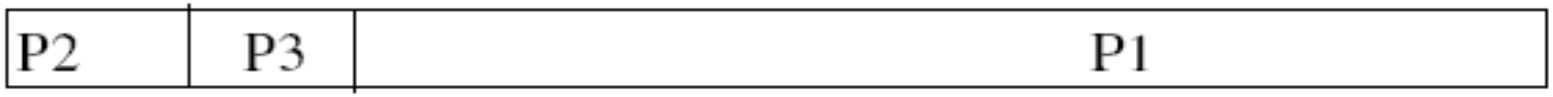
Processes	Burst time
P1	24
P2	3
P3	3

if the processes arrive in the order P1, P2, P3 and are served in FCFS order,



The average waiting time is (0 + 24 + 27)/3 = 17.

if the processes are served in SJF



The average waiting time is (6 + 0 + 3)/3 = 3.

Scheduling Algorithms

Shortest-Remaining-Time-First (SRTF)

Preemptive version of SJF.

Any time a new process enters the pool of processes to be scheduled, the scheduler compares the expected value for its remaining processing time with that of the process currently scheduled. If the new process's time is less, the currently scheduled process is preempted.

Merits:

Low average waiting time than SJF.

Useful in timesharing.

Demerits:

Very high overhead than SJF.

Requires additional computation.

Favours short jobs, longs jobs can be victims of starvation.

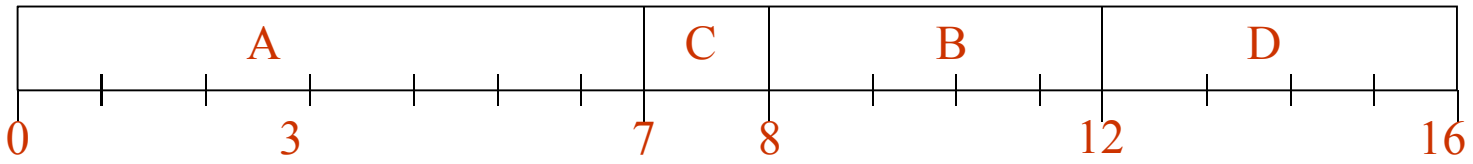
Scheduling Algorithms

SRTF-Performance

Scenario: Consider the following four processes with the length of CPU-burst time given in milliseconds:

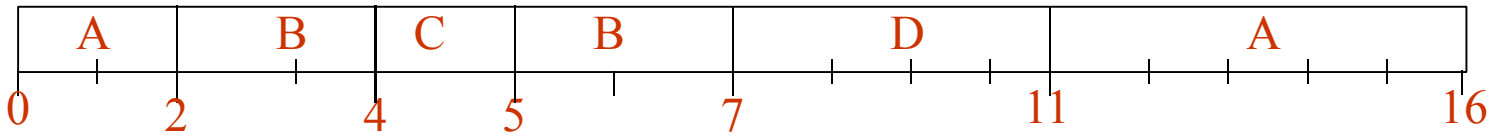
Processes	Arrival Time	Burst Time
A	0.0	7
B	2.0	4
C	4.0	1
D	5.0	4

SJF:



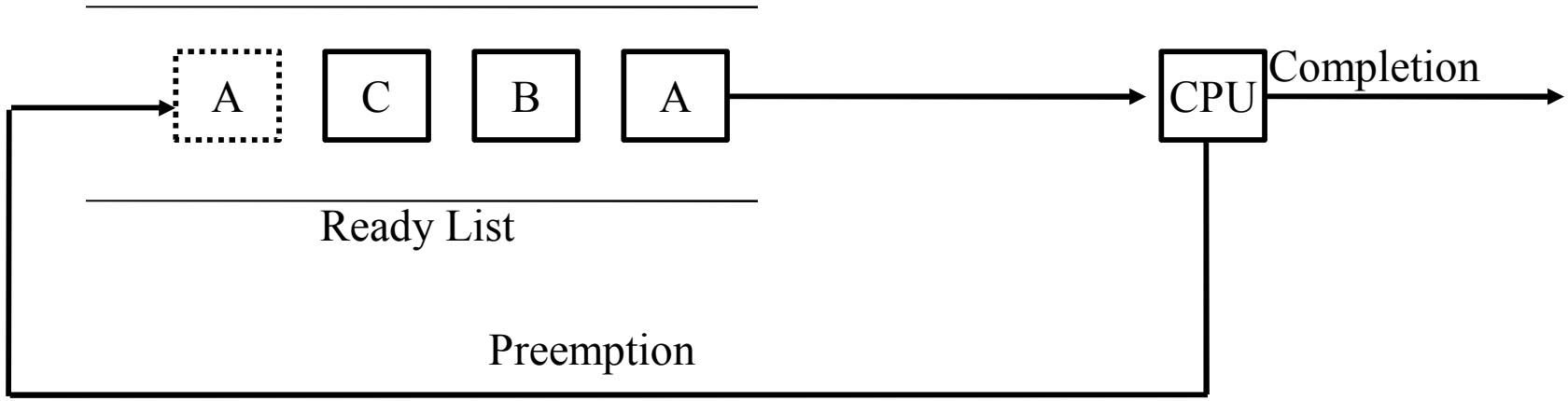
Average waiting time = (0 + 6 + 3 + 7)/4 = 4

SRTF:



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

Scheduling Algorithms
Round-Robin (RR)



Preemptive FCFS.

Each process is assigned a time interval (quantum), after the specified quantum, the running process is preempted and a new process is allowed to run.

Preempted process is placed at the back of the ready list.

Advantages:

- Fair allocation of CPU across the process.
- Used in timesharing system.
- Low average waiting time when process lengths very widely.

Scheduling Algorithms
RR-Performance

- Poor average waiting time when process lengths are identical.

  Imagine 10 processes each requiring 10 msec burst time and 1msec quantum is assigned.

  RR: All complete after about 100 times.

  FCFS is better! (About 20% time wastages in context-switching).
- Performance depends on quantum size.

Quantum size:

If the quantum is very large, each process is given as much time as needs for completion; RR degenerate to FCFS policy.

If quantum is very small, system busy at just switching from one process to another process, the overhead of context-switching causes the system efficiency degrading.

Optimal quantum size?

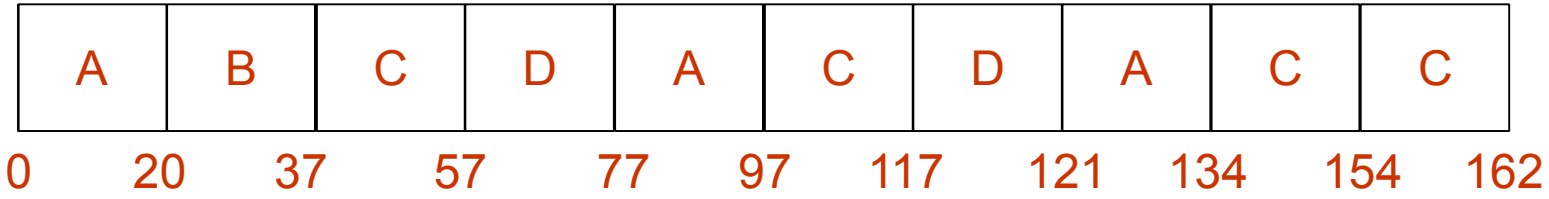
Key idea: 80% of the CPU bursts should be shorter than the quantum.

20-50 msec reasonable for many general processes.

Scheduling Algorithms
Example of RR with Quantum = 20

Process	Burst Time
A	53
B	17
C	68
D	24

- The Gantt chart is:



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

Scheduling Algorithms
Priority

Each process is assigned a priority value, and runnable process with the highest priority is allowed to run.

FCFS or RR can be used in case of tie.

To prevent high-priority process from running indefinitely, the scheduler may decrease the priority of the currently running process at each clock tick.

Assigning Priority

Static:

Some processes have higher priority than others.

Problem: Starvation.

Dynamic:

Priority chosen by system.

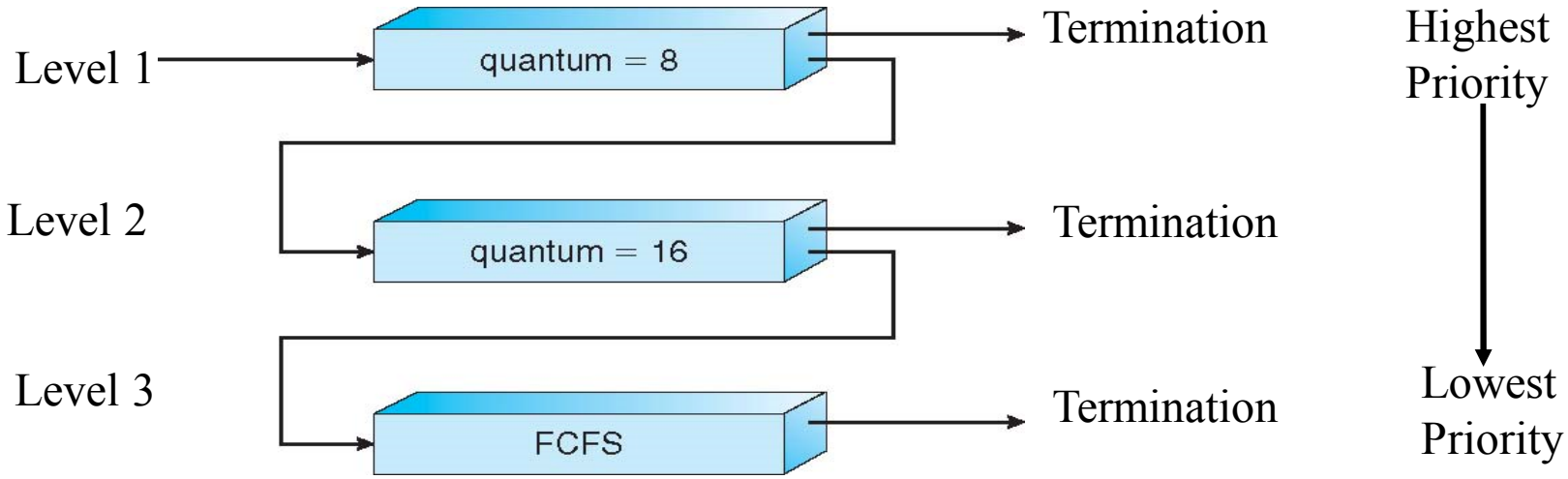
Decrease priority of CPU-bound processes.

Increase priority of I/O-bound processes.

Many different policies possible…….

E. g.: priority = (time waiting + processing time)/processing time.

Scheduling Algorithms
Multilevel-Feedback-Queues (MFQ)



MFQ implements multilevel queues having different priority to each level (here lower level higher priority), and allows a process to move between the queues. If the process use too much CPU time, it will be moved to a lower-priority queue.

Each lower-priority queue larger the quantum size.

Each queue may have its own scheduling algorithm

This leaves the I/O-bound and interactive processes in the high priority queue.

Scheduling Algorithms
MFQ-Example

Consider a MFQ scheduler with three queues numbered 1 to 3, with quantum size 8, 16 and 32 msec respectively.

The scheduler execute all process in queue 1, only when queue 1 is empty it execute process in queue 2 and process in queue 3 will execute only if queue 1 and queue 2 are empty.

A process first enters in queue 1 and execute for 8 msec. If it does not finish, it moves to the tail of queue 2.

If queue 1 is empty the processes of queue 2 start to execute in FCFS manner with 16 msec quantum. If it still does not complete, it is preempted and move to the queue 3.

If the process blocks before using its entire quantum, it is moved to the next higher level queue.

Home Works

HW#5

Question No 5.1 – 5.10 of Textbook

Process Synchronization

Reading: Chapter 6 of Textbook

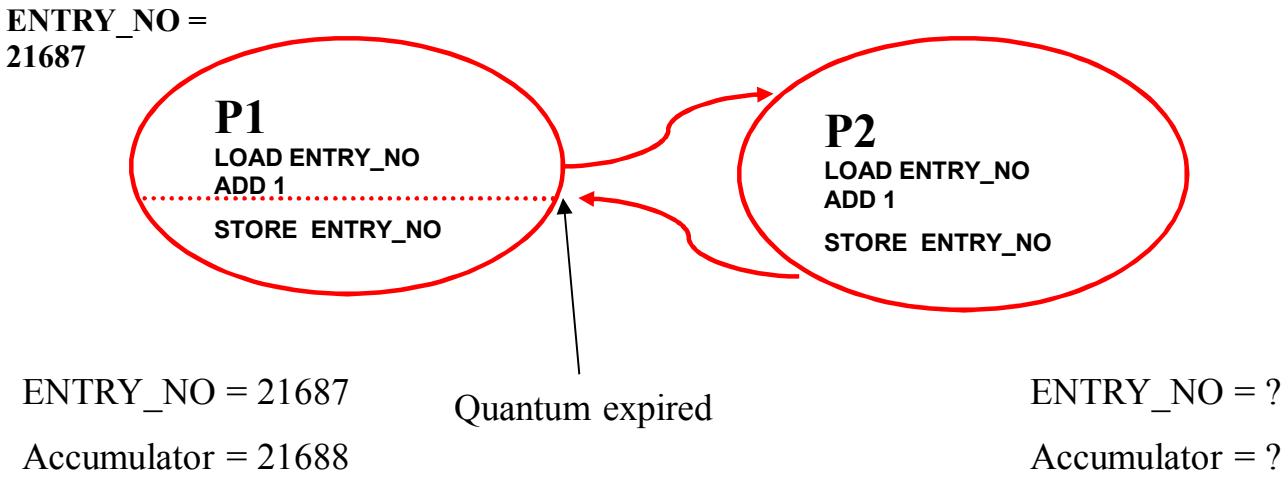
- How to make sure two or more processes do not get into each other's way when engaging in shared resources?
- How to maintain the proper sequence when dependencies are presents?

Process synchronization – synchronize the process actions

Race Conditions

Scenario: Suppose two user working in a computer, there is a mechanism to monitor continuously the total number of lines each users have entered, each user entry is recorded globally, different process monitoring different users .

Each process : LOAD ENTRY_NO
ADD 1
STORE ENTRY_NO



Situation where two or more processes are reading or writing some shared data and the final result depends on who run precisely, are called race conditions

Mutual Exclusion

Possibilities of race:

- many concurrent process read the same data.
- one process reading and another process writing same data.
- two or more process writing same data.

Solution: *prohibiting more than one process from reading writing the same data at the same time- Mutual Exclusion.*

Mutual Exclusion:

Some way of making sure that if one process is using a shared variables or files, the other process will be excluded from doing the same thing.

Critical Section Problem

Problem: How to avoid race?

Fact: The part of the time, process is busy doing internal computations and other things that do not lead to the race condition.

Code executed by the process can be grouped into sections, some of which require access to shared resources, and other that do not.

The sections of the code, common to all cooperating processes, that require access to shared resources are called critical section.

Critical Section Problem

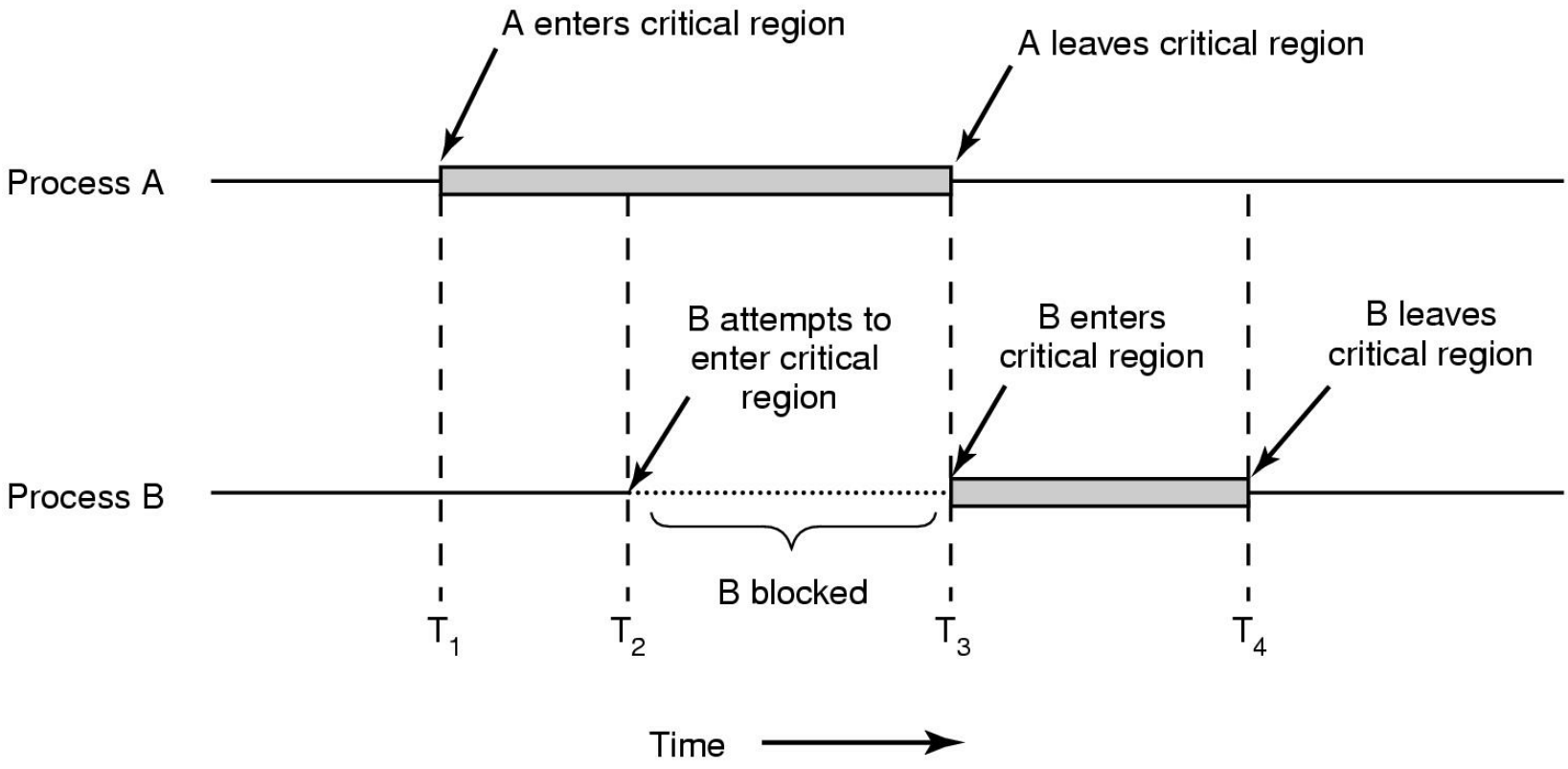
General structure of process

```
while(true){
        entry_section
                critical_section
        exit_section
                reminder_section
        }
```

Solution to Critical Section Problem (CSP) must satisfy the following conditions:

1. No two processes may be simultaneously inside their critical sections **(mutual exclusion).**
2. No process running outside its critical section may block other process to enter into Critical Section.
3. No process should have to wait forever to enter its Critical Section.

Critical Section Problem



Mutual exclusion in critical sections

Solution to CSP: Lock Variables

A single, shared (lock) variable (flag), initially 0. When a process wants to enter its CR, it first test the lock. If the lock is 0, the process set it to 1 and enters the CR. If the lock is already 1, the process just waits until it becomes 0.

```
While(lock =!0);
 {
        lock = 1;
        critical section
        lock = 0;
        remainder section
}
```

Advantages: seems no problems.

Problems:

Suppose that one process reads the lock and sees that it is 0, (due to quantum expire) before it can set lock to 1, another process scheduled, enter the CR, set lock to 1 and can have two process at CR (violates mutual exclusion).

Solution to CSP: Strict Alternation

Processes share a common integer variable turn. If turn == i then process P_i is allowed to execute in its CS, if turn == j then process P_j is allowed to execute.

```
While (true){
while(turn!=i); /*loop */
critical_section();
turn = j;
noncritical_section();
}
```

Process P_i

```
While (true){
while(turn!=j); /*loop*/
critical_section();
turn = i;
noncritical_section();
}
```

Process P_j

Advantages: Ensures that only one process at a time can be in its CS.

Problems: strict alternation of processes in the execution of the CS.

What happens if process i just finished CS and again need to enter CS and the process j is still busy at non-CS work? (violate condition 2)

Solution to CSP: Peterson's Algorithm

The two processes share two variables: int **turn**; Boolean **ready[2]**
The variable **turn** indicates whose turn it is to enter the critical section
The **ready** array is used to indicate if a process is ready to enter the critical section.
ready[i] = true $\Rightarrow$ P_i ready to enter its critical section
Algorithm for Process P_i

```
do {
        ready[i] = true; /* Pi is ready for CS*/
        turn = j;              /* set turn to other*/
        while (ready[j] ==true and turn == j) ; /* null statement*/
        critical section
        ready[i] = false;
        remainder section
} while (1);
```

Advantages: Preserves all conditions.

Problems: difficult to program for n-processes system and less efficient.

Solve the critical-section problem for two processes only

Solution to CSP

Hardware Solution

hardware support for critical sections problems

- Use special hardware instruction that reads the contents of the memory word (shared variable) into the register and then stores nonzero value at the memory address or instruction that swap the content of two memory locations.
- Modern machines provide special atomic (non-interruptable) hardware instructions: TestAndSet() – used in uniprocessor and Swap()

```
boolean TestAndSet (boolean *target)
    {
        boolean rv = *target;
        *target = TRUE;
        return rv:
    }
```

```
void Swap (boolean *a, boolean *b)
    {
        boolean temp = *a;
        *a = *b;
        *b = temp:
    }
```

Solution to CSP
Hardware Solution

Shared boolean variable lock, initialized to FALSE

```
do {
    while ( TestAndSet (&lock ))
            ;   // do nothing
        //    critical section
    lock = FALSE;
        //      remainder section
} while (TRUE);
```

Shared Boolean variable lock, initialized to FALSE;
Each process has a local Boolean variable key

```
do {
    key = TRUE;
    while ( key == TRUE)
        Swap (&lock, &key );
     //    critical section
    lock = FALSE;
    //      remainder section
} while (TRUE);
```

Advantages: *Preserves all condition, easier programming task and improve system efficiency.*

Problems: *difficulty in hardware design.*

Busy Waiting Alternate?

Busy waiting:

When a process want to enter its CS, it checks to see if the entry is allowed, if it is not, the process just sits in a

tight loop waiting until it is.

Waste of CPU time for NOTHING!

Possibility of *Sleep* and *Wakeup* pair instead of waiting.

Sleep causes to caller to block, until another process wakes it up.

Sleep and Wakeup
Producer-Consumer Problem

- Two process share a common, fixed sized buffer.
- Suppose one process, producer, is generating information that the second process, consumer, is using.
- Their speed may be mismatched, if producer insert item rapidly, the buffer will full and go to the sleep until consumer consumes some item, while consumer consumes rapidly, the buffer will empty and go to sleep until producer put something in the buffer.

Sleep and Wakeup
Producer-Consumer Problem

```
#define N = 100     /* number of slots in the buffer*/
 int count = 0;      /* number of item 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 */
  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 go to sleep*/
 item = remove_item(); /*take item out
                                of buffer*/
 count = count -1; /*decrement count*/
 if (count == N-1)wakeup(producer);
       /*was buffer full ?*/
 consume_item();      /*print item*/
}
}
```

Sleep and Wakeup
Producer-Consumer Problem

Problem:

-leads to race as in spooler directory.

-What happen if buffer is empty, the consumer just reads count and quantum is expired, the producer inserts an item in the buffer, increments count and wake up consumer. The consumer not yet asleep, so the wakeup signal is lost, the consumer has the count value 0 from the last read so go to the sleep. Producer keeps on producing and fill the buffer and go to sleep, both will sleep forever.

Think: If we were able to save the wakeup signal that was lost.....

Semaphores

E. W. Dijkstra (1965) suggested using an integer variable to count the number of wakeups, called a semaphore.
It could have the value 0, indicating no wakeups ware saved, or some positive value if one or more wakeups were pending.

Operations: *Down* and *Up* (originally he proposed *P* and *V* in Dutch and sometimes known as wait and signal)
Down: checks if the value greater than 0
yes- decrement the value (i.e. Uses one stored wakeup) and continues.
No- process is put to sleep without completing down.
- Checking value, changing it, and possibly going to sleep, is all done as single action.
Up: increments the value; if one or more processes were sleeping, unable to complete earlier down operation, one of them is chosen and is allowed to complete its down.

Semaphores

```
typedef int semaphore S;
void down(S)
{ if(S> 0)  S--;
            else        sleep();
}
void up(S)
{if(one or more processes are sleeping on S)
            one of these process is proceed;
              else     S++;
}
while(TRUE){
down(mutex)
critical_region();
up(mutex);
noncritical_region();
}
```

Semaphores

Producer-Consumer using Semaphore

```
#define N 100                           /*number of slots in buffer*/
typedef int semaphore;                  /*defining semaphore*/
semaphore mutex = 1;                    /* control access to the CR */
semaphore empty = N                     /*counts empty buffer slots*/
semaphore full = 0;                     /counts full buffer slots*/

void producer(void)
{
int item;
while(TRUE){                            /*repeat forever */
                item = produce_item();      /*generate something */
                down(empty);                /*decrement empty count*/
                down(mutex);                /*enter CR */
                insert_item();              /* put new item in buffer*/
                up(mutex);                  /* leave CR*/
                up(full);                   /*increment count of full slots*/
        }
}
```

Semaphores

Producer-Consumer using Semaphore

```
void consumer(void)
{
int item;
while(TRUE){                        /*repeat forever*/
        down(full);                         /*decrement full count */
        down(mutex);                        /*enter CR*/
        item = remove_item();               /*take item from buffer*/
        up(mutex);                          /*leave CR*/
        up(empty);                          /*increment count of empty slots*/
        consume_item();             /*print item*/
}
}
```

Use of Semaphore

1. *To deal with n-process critical-section problem.*
The n processes share a semaphore, (e. g. mutex) initialized to 1.
2. To solve the various synchronizations problems.
For example two concurrently running processes: P1 with statement S1 and P2 with statement S2, suppose, it required that S2 must be executed after S1 has completed. This problem can be implemented by using a common semaphore, synch, initialized to 0.

```
P1:   S1;
      up(synch);
P2:   down(synch);
      S2;
```

Criticality Using Semaphores

All process share a common semaphore variable mutex, initialize to 1. Each process must execute down(mutex) before entering CR, and up(mutex) afterward. *What happens when this sequence is not observed?*

1. When a process interchange the order of *down* and *up* operation: causes the multiple processes in CR simultaneously.
2. When a process replace *up* by *down*: causes the dead lock.
3. When a process omits *down* or *up* or both: violated mutual exclusion and deadlock occurs.

A subtle error is capable to bring whole system grinding halt!!

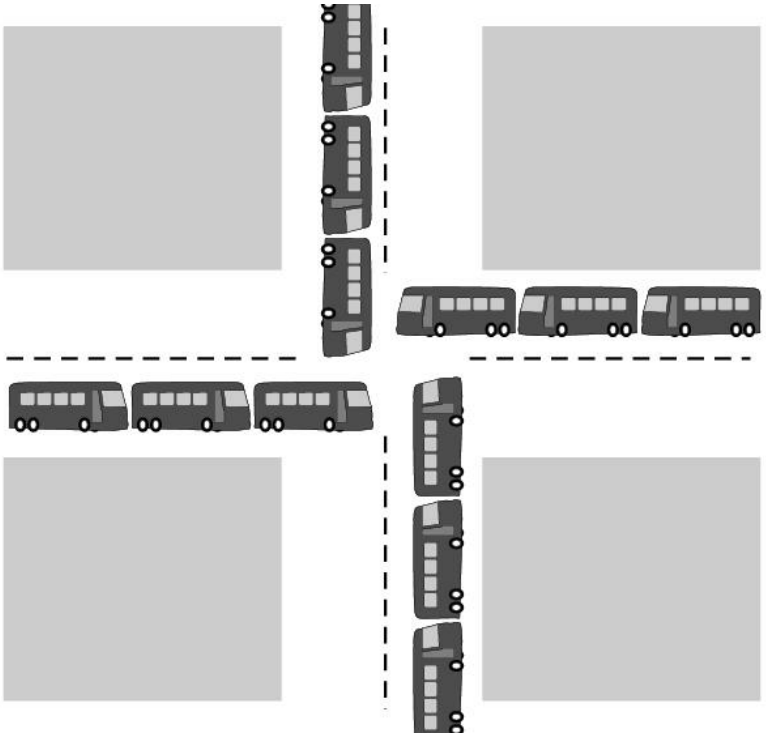
Home Works

HW #6

1. Question 6.1, 6.2, 6.11, 6.12 of Textbook
2. What is the meaning of busy waiting? What others kinds of waiting are in OS? Compare each types on their applicability and relative merits.
3. Show the Peterson's algorithm preserve mutual exclusion, indefinite postponement and dead lock.

Deadlock

A process in a multiprogramming system is said to be in dead lock if it is waiting for a particular event that will never occur.



Resource Deadlock

A process request a resource before using it, and release after using it.

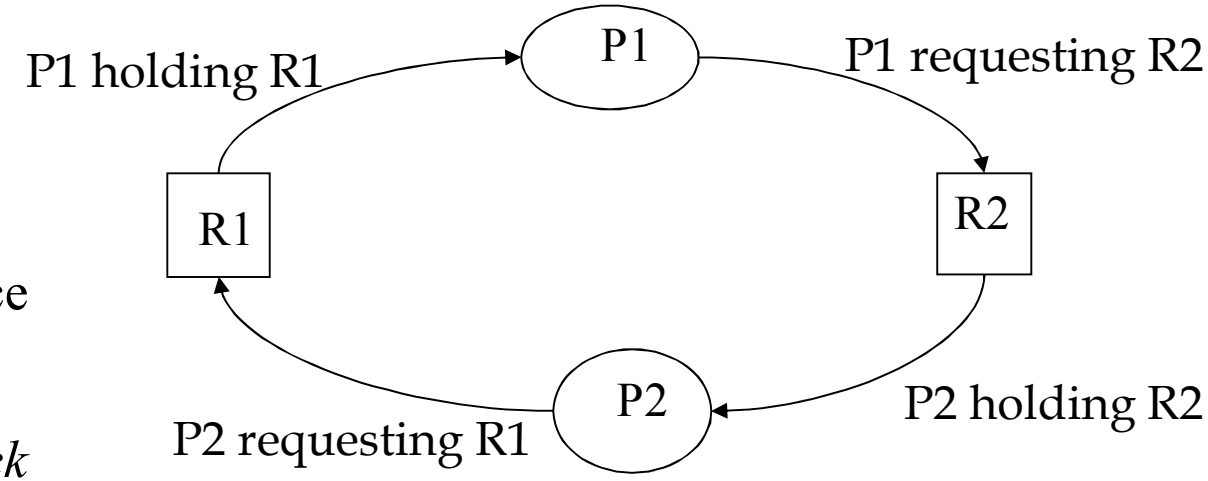
1. Request the resource.
2. Use the resource.
3. Release the resource.

If the resource is not available when it is requested, the requesting process is force to wait.

Most deadlocks in OS developed because of the normal contention for dedicated resources.

Process P1 holds resource R1 and needs resource R2 to continue; Process P2 holds resource R2 and needs resource R1 to continue – deadlock.

Key: Circular wait - deadlock



Conditions for Deadlock

1. *Mutual Exclusion:* Process claims exclusive control of resources they require.
2. *Hold and Wait:* Processes hold resources already allocated to them while waiting for additional resources.
3. *No preemption:* Resources previously granted can not be forcibly taken away from the process.
4. *Circular wait:* Each process holds one or more resources that are requested by next process in the chain.

A deadlock situation can arise if all four conditions hold simultaneously in the system.

Handling Deadlocks

Deadlock handling strategies:

- We can use a protocol to *prevent* or *avoid* deadlocks, ensuring that the system never enter a deadlock state.
- We can allow the system to enter a deadlock state, *detect* it, and *recover*.
- We can ignore the problem all to gather, and pretended that deadlock never occur in the system.

Deadlock prevention

Fact: *If any one of the four necessary conditions is denied, a deadlock can not occur.*

Denying Mutual Exclusion:

Sharable resources do not require mutually exclusive access such as read only shared file.

Problem: Some resources are strictly nonsharable, mutually exclusive control required.

We can not prevent deadlock by denying the mutual exclusion

Denying Hold and Wait:

Resources grant on *all or none* basis;

If all resources needed for processing are available then granted and allowed to process.

If complete set of resources is not available, the process must wait set available.

While waiting, the process should not hold any resources.

Problem: Low resource utilization.
Starvation is possible.

Deadlock prevention

Denying No-preemption:

When a process holding resources is denied a request for additional resources, that process must release its held resources and if necessary, request them again together with additional resources.

Problem: When process releases resources the process may loose all its works to that point .

Indefinite postponement or starvation.

Denying Circular Wait:

All resources are uniquely numbered, and processes must request resources in linear ascending order.

The only ascending order prevents the circular.

Problem: Difficult to maintain the resource order; dynamic update in addition of new resources.

Indefinite postponement or starvation.

Deadlock Avoidance

Avoiding deadlock by careful resource allocation.

Decide whether granting a resource is safe or not, and only make the allocation when it is safe.

Need extra information in advanced, maximum number of resources of each type that a process may need.

The deadlock-avoidance algorithm dynamically examines the resource-allocation state to ensure that there can never be a circular-wait condition.

Deadlock Avoidance

Safe and Unsafe States

A state is said to be safe if it is not deadlocked and there is a some scheduling order in which every process can run to completion.

	Has	Max
A	3	9
B	2	4
C	2	7

Free: 3

(a)

	Has	Max
A	3	9
B	4	4
C	2	7

Free: 1

(b)

	Has	Max
A	3	9
B	0	–
C	2	7

Free: 5

(c)

	Has	Max
A	3	9
B	0	–
C	7	7

Free: 0

(d)

	Has	Max
A	3	9
B	0	–
C	0	–

Free: 7

(e)

State in (a) is safe

Deadlock Avoidance

Safe and Unsafe States

	Has	Max
A	3	9
B	2	4
C	2	7

Free: 3

(a)

	Has	Max
A	4	9
B	2	4
C	2	7

Free: 2

(b)

	Has	Max
A	4	9
B	4	4
C	2	7

Free: 0

(c)

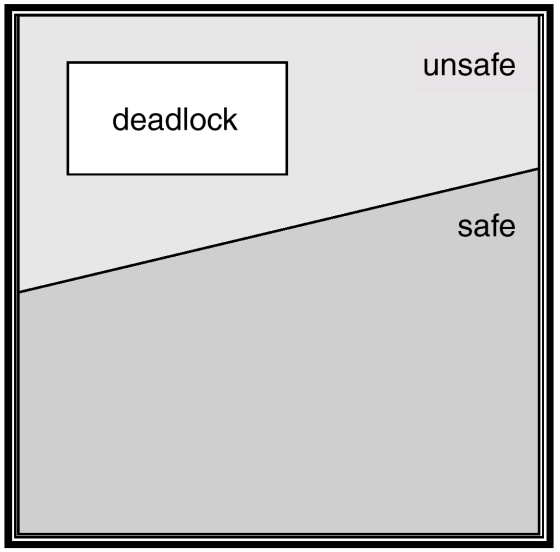
	Has	Max
A	4	9
B	—	—
C	2	7

Free: 4

(d)

State in (b) is not safe.

In a safe state, system can guarantee that all processes will finish – no deadlock occur; from an unsafe state, no such guarantee can be given – deadlock may occur.



Deadlock Avoidance

Banker's Algorithm

Models on the way of banking system to ensure that the bank never allocates its available cash such that it can no longer satisfy the needs of all its customers.

When a process request the set of resources, the system determine whether the allocation of these resources will left the system in safe state, if it will the resources are allocated, otherwise process must wait until some other process release enough resources.

Data structures: Available, Max, Allocation & Need.

need = max – allocation.

Deadlock Avoidance

Banker's Algorithm

```
finish_i = False;
for each process
  if (need_i <= available && finish_i
    = false)
          continue_i;
   available = available +
    allocation;
  finish_i = True;
for each process
if (finish_i = True)
     system is in safe state.
```

```
if (request_i <= need_i)
    if (request_i <= available)
     {  available = available- request_i;
     allocation_i = allocation_i + request_i;
     need_i = need_i–request_i;
     }
    else
     wait;
else
    error;
```

Deadlock Avoidance

Banker's Algorithm

	allo.	max
A	0	6
B	0	5
C	0	4
D	0	7

Available = 10

	allo.	max
A	1	6
B	1	5
C	2	4
D	4	7

Available = 2

	allo.	max
A	1	6
B	2	5
C	2	4
D	4	7

Available = 1

Which one is unsafe?

Deadlock Avoidance

Problem with Banker's Algorithm

Algorithms requires fixed number of resources, some processes dynamically changes the number of resources.

Algorithms requires the number of resources in advanced, it is very difficult to predict the resources in advanced.

Algorithms predict all process returns within finite time, but the system does not guarantee it.

Deadlock Detection and Recovery

Instead of trying to prevent or avoid deadlock,
system allow to deadlock to happen,

and recover from deadlock

when it does occur.

Hence, the mechanism for deadlock detection and recovery from deadlock required.

Deadlock Detection

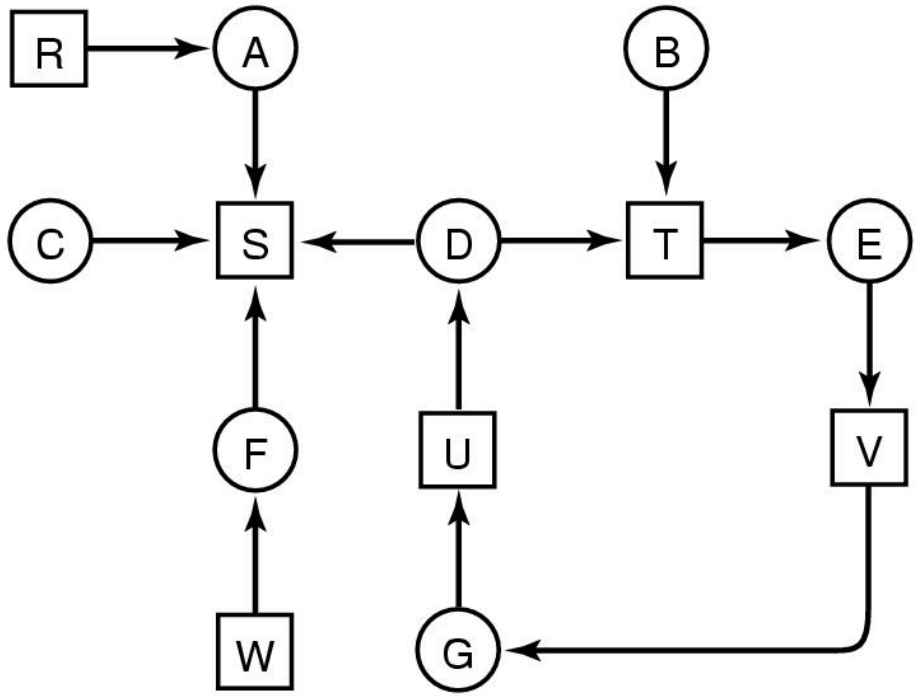
Consider the following scenario:

a system with 7 processes (A – G), and 6 resources (R – W) are in following state.

1. Process A holds R and wants S.
2. Process B holds nothing but wants T.
3. Process C holds nothing but wants S.
4. Process D holds U and wants S and T.
5. Process E holds T and wants V.
6. Process F holds W and wants S.
7. Process G holds V and wants U.

Is there any deadlock situation?

Deadlock Detection



Resource graph

How to detect the cycle in directed graph?

Deadlock Detection Algorithm

```
List L;
Boolean cycle = False;
for each node
    L = Φ                    /* initially empty list*/
    add node to L;
    for each node in L
    for each arc
     if (arc[i] = true)
          add corresponding node to L;
          if (it has already in list)
               cycle = true;
               print all nodes between these nodes;
               exit;
      else if (no such arc)
          remove node from L;
          backtrack;
```

Recovery from Deadlock

What do next if the deadlock detection algorithm succeed?
– recover from deadlock.

By Resource Preemption

Preempt some resources temporarily from a processes and give these resources to other processes until the deadlock cycle is broken.

Problem:

Depends on the resources.

Need extra manipulation to suspend the process.

Difficult and sometime impossible.

Recovery from Deadlock

By Process Termination

Eliminating deadlock by killing one or more process in cycle or process not in cycle.

Problem:

If the process was in the midst of updating file, terminating it will leave file in incorrect state.

Key idea: we should terminate those processes the termination of which incur the minimum cost.

Ostrich Algorithm

Fact: there is no good way of dealing with deadlock.

Ignore the problem altogether.

For most operating systems, deadlock is a rare occurrence;
So the problem of deadlock is ignored, like an ostrich sticking its head in the sand and hoping
the problem will go away.

Home Works

HW #7

1. Question no 7.1 – 7.10 of Textbook
2. Distinguish indefinite postponement and Deadlock.

2. State the conditions necessary for deadlock to exist. Give reason, why all conditions are necessary.

3. A system has two processes and three identical resources. Each process needs a maximum of two resources. Is deadlock possible? Explain your answer.

4. Show that four necessary conditions hold in traffic deadlock example. State a simple rule that will avoid deadlock in traffic system.