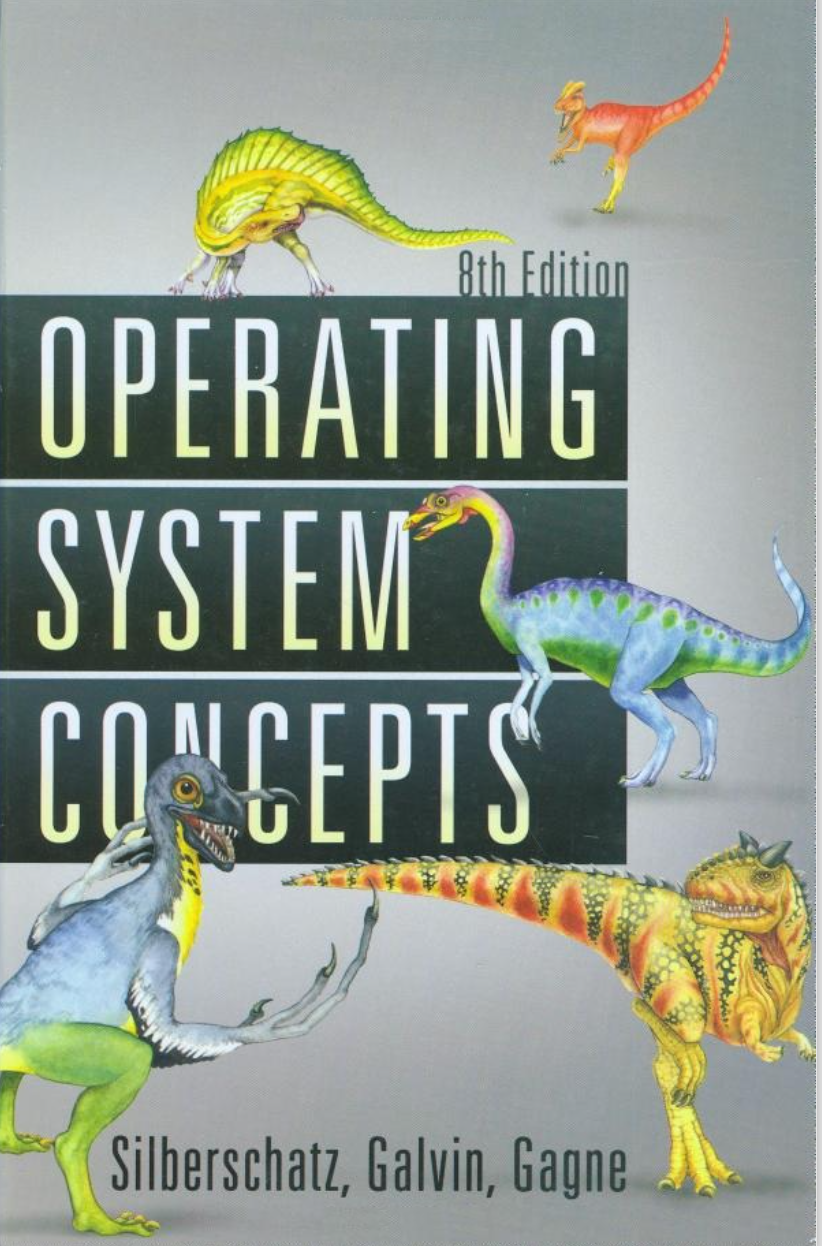
**T6Synopsis of Operating System for (Semester 2 - 2017)**

The whole course requires at least 10 sessions to be complete. (Each session is on a two-hour basis).

After each session of key chapters, there's also a set of Review Questions on textbook to help students check their understanding. Students should be able to complete the review questions in order to better understand similar questions which will be asked on final exam.

By the time students finish this class, they'll know how to apply most of algorithms to such questions as process state, critical sections, pre-emptive process, CPU scheduling, deadlock, page replacement, strategies of allocation and disk performance, and many other questions. They'll also know how to select the right and appropriate words and phrases to answer questions needed to explain in details.

**Corresponding textbook:**

****

Operating Systems Concepts (8th edition) by A. Silberschatz, P. B. Galvin  
and G. Gagne,, John Wiley& Sons, 2009

**Chapter 1, 2 Introduction (session 1)**

**Keywords**: user mode/supervisor/kernel mode, virtual machine, multiprogramming, spooling, timesharing, system calls, process, signals, file, file descriptor, shell

* What are the two functions of an operating systems?
* What is multiprogramming?
* Why is the process table needed in timesharing?

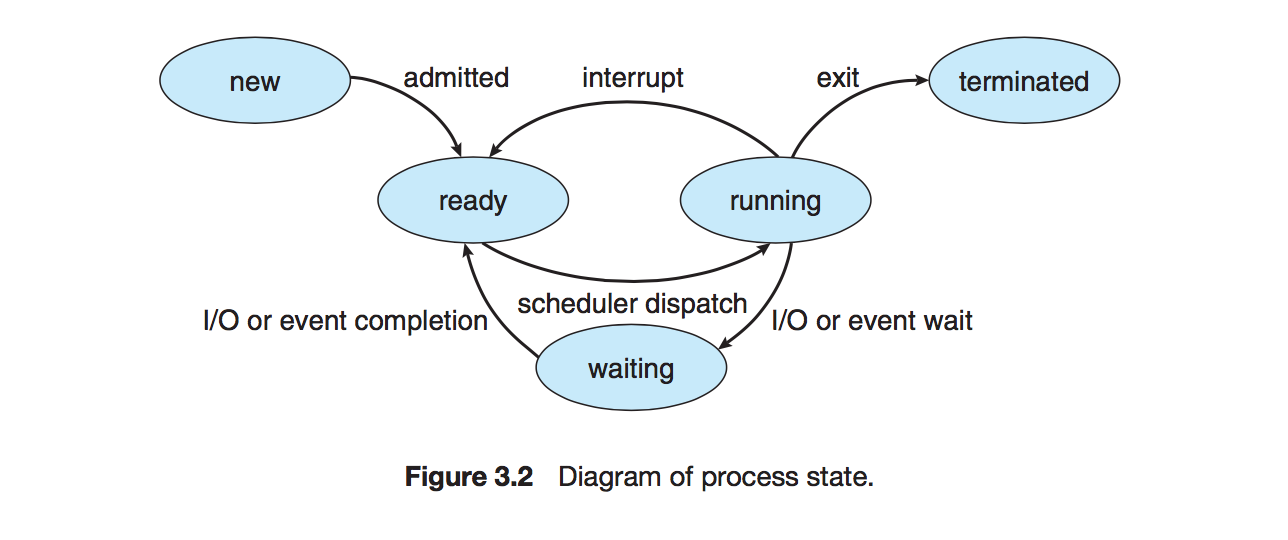
Textbook chapter 1 problems: 5, 6, 12 Textbook chapter 2 problems: 1, 2, 8, 18, 21

**82326669 for wifi password**

**process synchronization**

**Chapter 3-7 Processes Management (session 2, 3, 4, 5)**

**Keywords**: fork, process state, threads, race conditions, critical section, mutual exclusion, busy waiting, semaphore, atomic, signal & wait, mutex, monitor, condition variables, starvation, compute-bound, IO-bound, pre-emptive, interactive, response time, quantum, context switch, send, receive, notify, rendezvous, traps, deadlock, resources, resource allocation graph (RAG), deadlock prevention, deadlock avoidance, safe state



* **READY to RUNNING** – It is handled by the Process Scheduler using some predefined algorithm, such as FCFS, SJF, priority scheduling or round robin, to determine which process will get the CPU, when, and for how long.
* **RUNNING back to READY** – It is handled by the Process Scheduler according to some predefined time limit or other criterion, for example, priority interrupts or quantum expired.
* **RUNNING to WAITING** – It is handled by the Process Scheduler and is initiated by an instruction in the process such as a command to READ, WRITE or other I/O requests.
* **WAITING to READY** – It is handled by the Process Scheduler and is initiated by signal from I/O device manager that I/O request has been satisfied and job can continue.

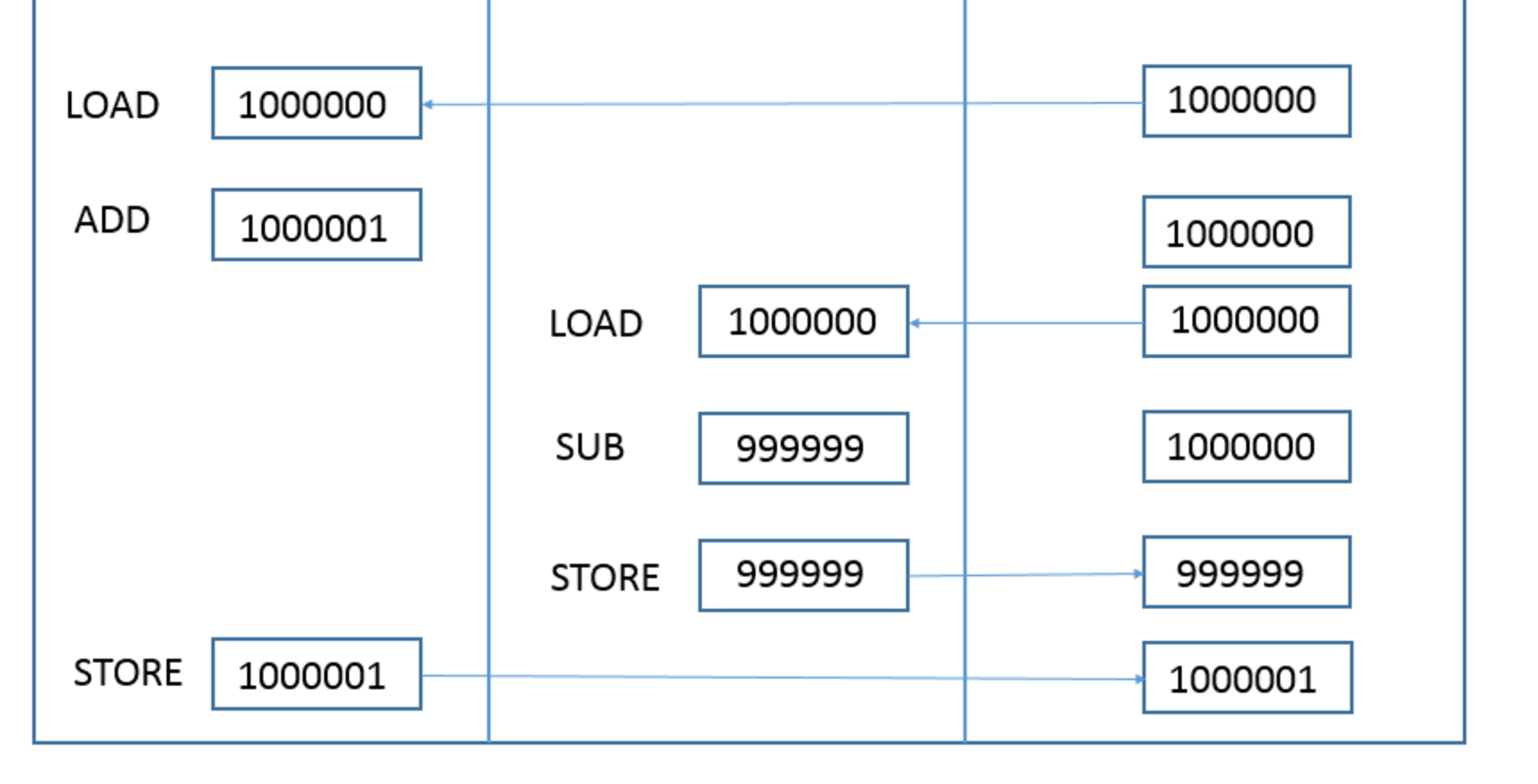
Refer to ppt

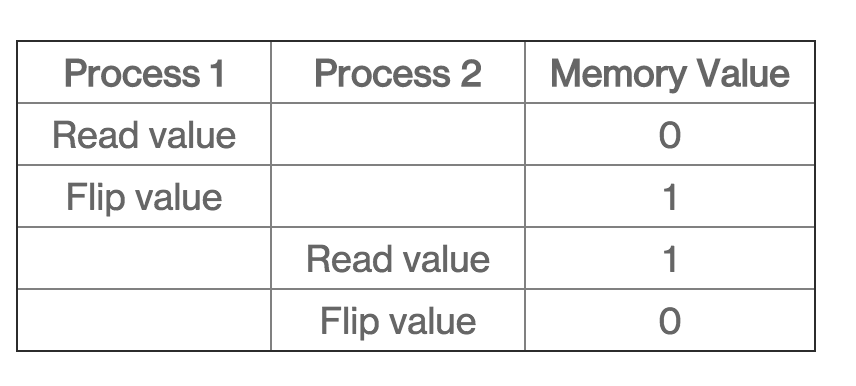
Race condition:

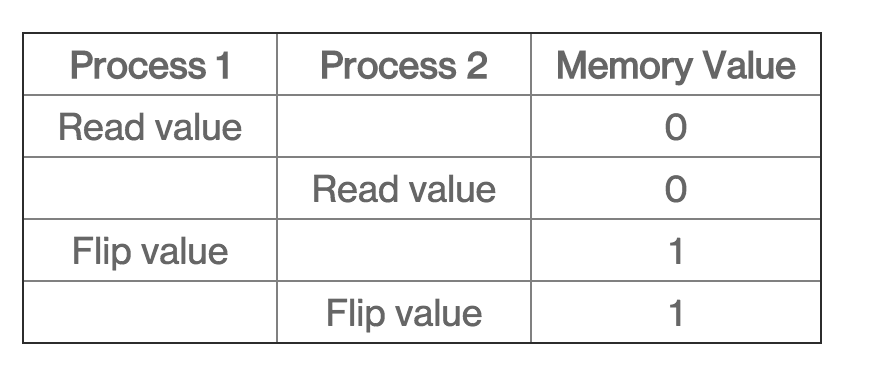
See java example:

A race condition occurs when two or more threads can access shared data and they try to change it at the same time. Because the thread scheduling algorithm can swap between threads at any time, you don't know the order in which the threads will attempt to access the shared data. Therefore, the result of the change in data is dependent on the thread scheduling algorithm, i.e. both threads are "racing" to access/change the data.

Problems often occur when one thread does a "check-then-act" (e.g. "check" if the value is X, then "act" to do something that depends on the value being X) and another thread does something to the value in between the "check" and the "act".







**There is a theoretical solution to the problem.**

Atomic:

An operation acting on shared memory is **atomic** if it completes in a single step relative to other threads.

Critical section:

In [concurrent programming](https://en.wikipedia.org/wiki/Concurrent_programming), concurrent accesses to shared resources can lead to unexpected or erroneous behavior, so parts of the program where the shared resource is accessed are protected. This protected section is the **critical section** or **critical region.** It cannot be executed by more than one process.

**Mutex:**  
  
Is a key to a toilet. One person can have the key - occupy the toilet - at the time. When finished, the person gives (frees) the key to the next person in the queue.  
  
Officially: "Mutexes are typically used to serialise access to a section of  re-entrant code that cannot be executed concurrently by more than one thread. A mutex object only allows one thread into a controlled section, forcing other threads which attempt to gain access to that section to wait until the first thread has exited from that section."  
Ref: Symbian Developer Library  
  
(A mutex is really a semaphore with value 1.)

Mutex obj;

pthread\_mutex\_lock(&obj) 

global\_data; 

pthread\_mutex\_unlock(&obj)

**Semaphore:**  
  
Is the number of free identical toilet keys. Example, say we have four toilets with identical locks and keys. The semaphore count - the count of keys - is set to 4 at beginning (all four toilets are free), then the count value is decremented as people are coming in. If all toilets are full, ie. there are no free keys left, the semaphore count is 0. Now, when eq. one person leaves the toilet, semaphore is increased to 1 (one free key), and given to the next person in the queue.  
  
Officially: "A semaphore restricts the number of simultaneous users/processes of a shared resource up to a maximum number. Threads can request access to the resource (decrementing/wait() vthe semaphore), and can signal() that they have finished using the resource (incrementing the semaphore)."  
Ref: Symbian Developer Library

sema(10) // ten threads/process have the concurrent access.   
  
sema\_lock(&sema\_obj)   
global\_data   
sema\_unlock(&sema\_obj)

Peterson’s algorithm



**package** philo;

/\*每个哲学家相当于一个线程\*/

**class** **Philosophers** **extends** **Thread**{

**private** **String** name;

**private** **Fork** fork;

**public** Philosophers (**String** **name**,**Fork** **fork**){

**super**(name);

**this**.name=name;

**this**.fork=fork;

}

**public** **void** run(){

**while**(**true**){

thinking();

fork.takeFork();

eating();

fork.putFork();

}

}

**public** **void** eating(){

**System**.***out***.println("I am Eating:"+name);

**try** {

*sleep*(1000);//模拟吃饭，占用一段时间资源

} **catch** (**InterruptedException** e) {

// **TODO** Auto-generated catch block

e.printStackTrace();

}

}

**public** **void** thinking(){

**System**.***out***.println("I am Thinking:"+name);

**try** {

*sleep*(1000);//模拟思考

} **catch** (**InterruptedException** e) {

// **TODO** Auto-generated catch block

e.printStackTrace();

}

}

}

**class** **Fork**{

/\*5只筷子，初始为都未被用\*/

**private** **boolean**[] used={**false**,**false**,**false**,**false**,**false**};

/\*只有当左右手的筷子都未被使用时，才允许获取筷子，且必须同时获取左右手筷子\*/

**public** **synchronized** **void** takeFork(){

**String** name = **Thread**.*currentThread*().getName();

**int** i = **Integer**.*parseInt*(name);

**while**(used[i]||used[(i+1)%5]){

**try** {

wait();//如果左右手有一只正被使用，等待

} **catch** (**InterruptedException** e) {

// **TODO** Auto-generated catch block

e.printStackTrace();

}

}

used[i]= **true**;

used[(i+1)%5]=**true**;

}

/\*必须同时释放左右手的筷子\*/

**public** **synchronized** **void** putFork(){

**String** name = **Thread**.*currentThread*().getName();

**int** i = **Integer**.*parseInt*(name);

used[i ]= **false**;

used[(i+1)%5]=**false**;

notifyAll();//唤醒其他线程

}

}

//测试

**public** **class** **Philosopher** {

**public** **static** **void** main(**String** [**]arg**s){

**Fork** fork = **ne**w **For**k();

**ne**w **Philosopher**s("0",fork).start();

**ne**w **Philosopher**s("1",fork).start();

**ne**w **Philosopher**s("2",fork).start();

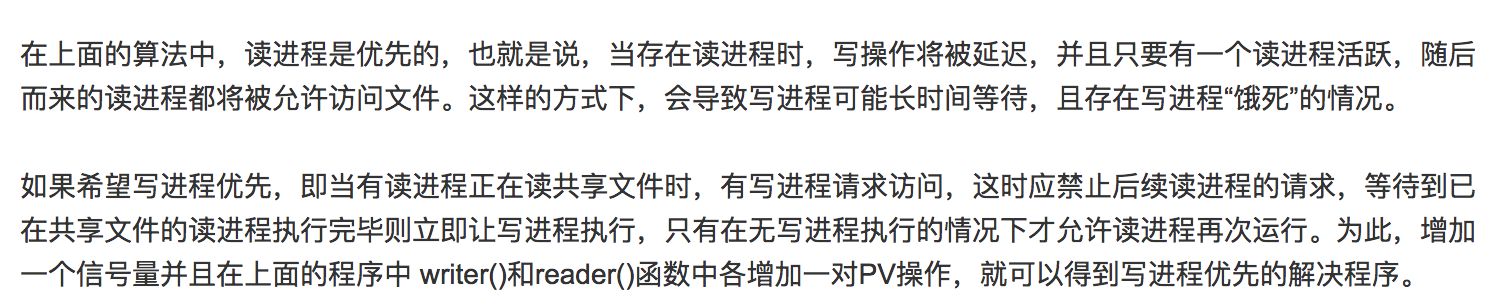
**ne**w **Philosopher**s("3",fork).start();

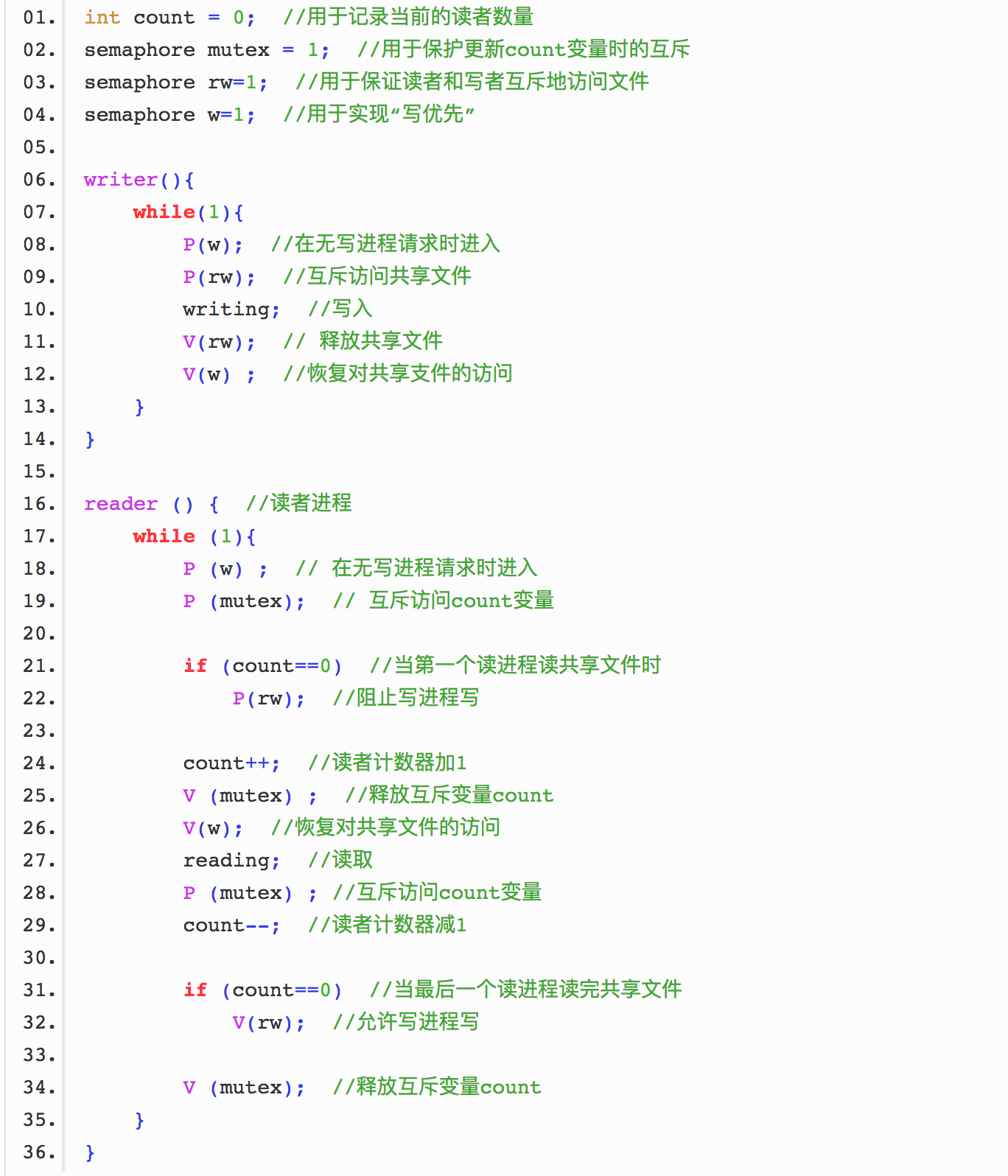
**ne**w **Philosopher**s("4",fork).start();

}

}



****

 ****

**Process Synchronization**

**What is a critical section?**

**What three requirements must a solution to the critical-section problem satisfy?**

**What does execute “atomically” mean?**

**Describe context switch**

In general, the operating system must save the state of the currently running process and restore the state of the process scheduled to be run next. Saving the state of a process typically includes the values of all the CPU registers in addition to memory allocation. Context switches must also perform many architecture-specific operations, including flushing data and instruction caches.

**Define the operations wait (S) and signal (S)**

**Show that, if the wait() and signal() semaphore operations are not executed atomically, then mutual exclusion may be violated.**

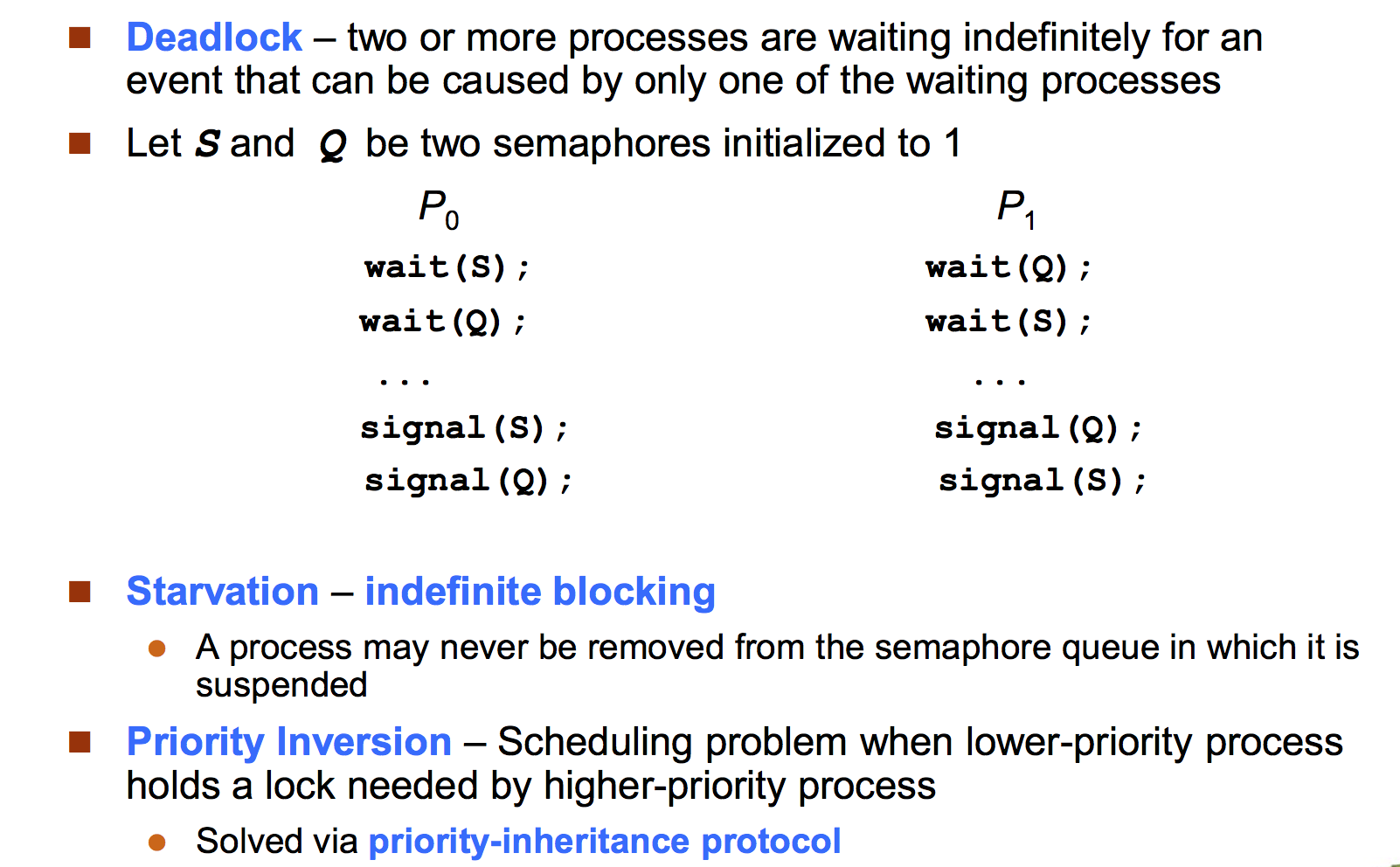
A wait operation atomically decrements the value associated with a semaphore. If two wait operations are executed on a semaphore when its value is 1, if the two operations are not performed atomically, then it is possible that both operations might proceed to decrement the semaphore value thereby violating mutual exclusion.

**What is the meaning of the term *busy waiting*? What other kinds of waiting are there in an operating system? Can busy waiting be avoided altogether? Explain your answer.**

*Busy waiting* means that a process is waiting for a condition to be satisfied in a tight loop without relinquish the processor. Alternatively, a process could wait by relinquishing the processor, and block on a condition and wait to be awakened at some appropriate time in the future. Busy waiting can be avoided but incurs the overhead associated with putting a process to sleep and having to wake it up when the appropriate program state is reached (Context switches).

**Explain why spinlocks are not appropriate for single-processor systems yet are often used in multiprocessor systems.**

Spinlocks are not appropriate for single-processor systems because the condition that would break a process out of the spinlock could be obtained only by executing a different process. If the process is not relinquishing the processor, other processes do not get the opportunity to set the program condition required for the first process to make progress. In a multiprocessor system, other processes execute on other processors and thereby modify the program state in order to release the first process from the spinlock.

****

**Three requirements for critical sections**

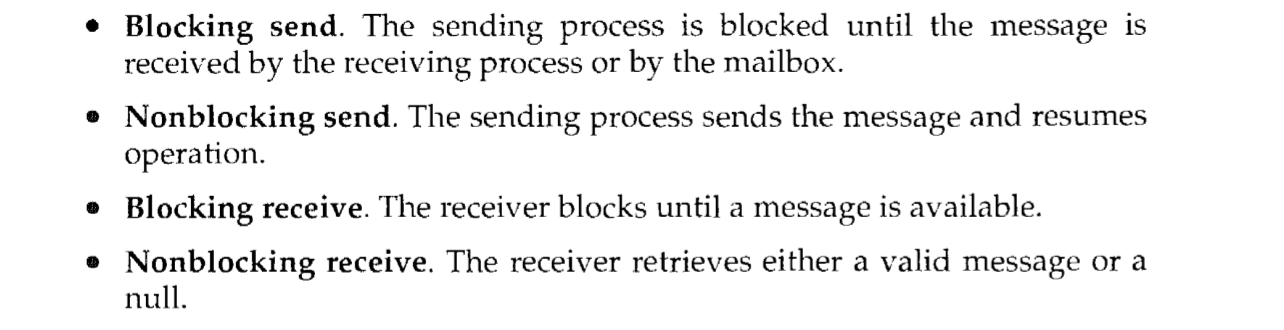
1. Mutual Exclusion. If process Pi is executing in its critical section (CS), then no other process can execute in its CS.

2. Progress. If no process is executing in its CS and there exist some processes that wish to enter their CS, then the selection of the process that will enter the CS next cannot be postponed indefinitely.

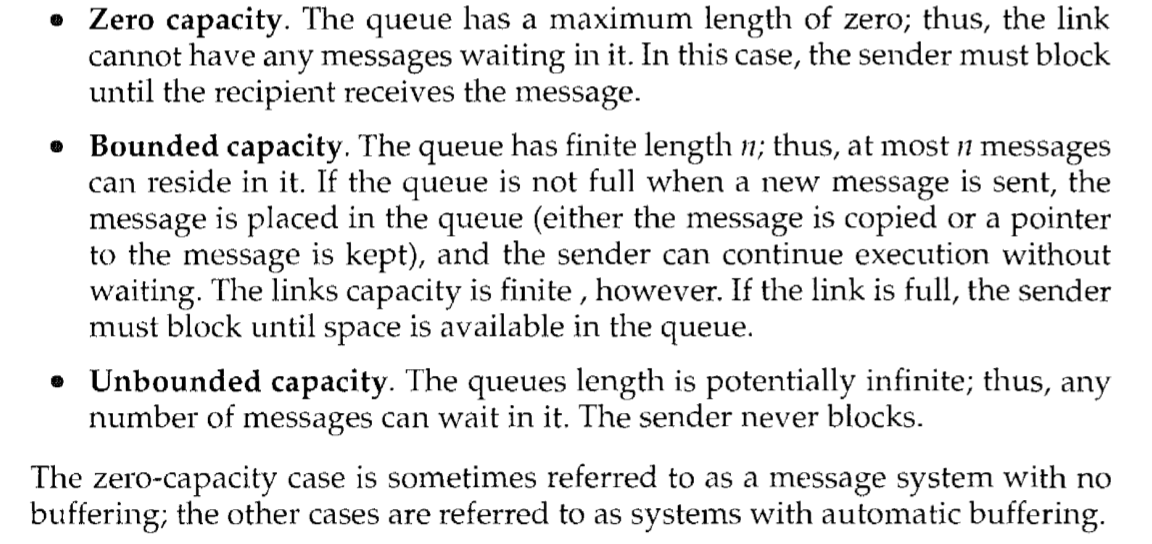
3. Bounded Waiting. There exists a bound on the number of times that other processes are allowed to enter their CS after a process has made a request to enter its CS and before that request is granted.

**Describe Process Communication**

Synchronous:

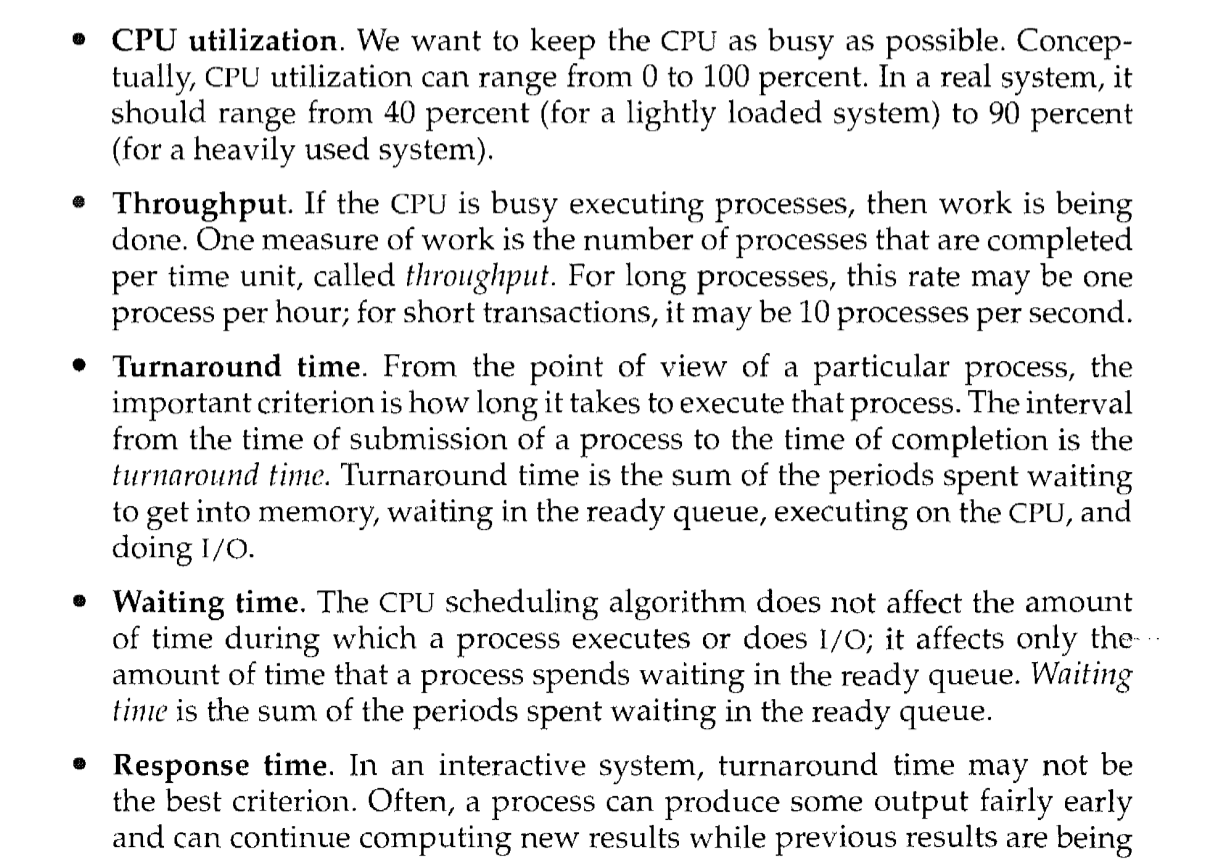
****

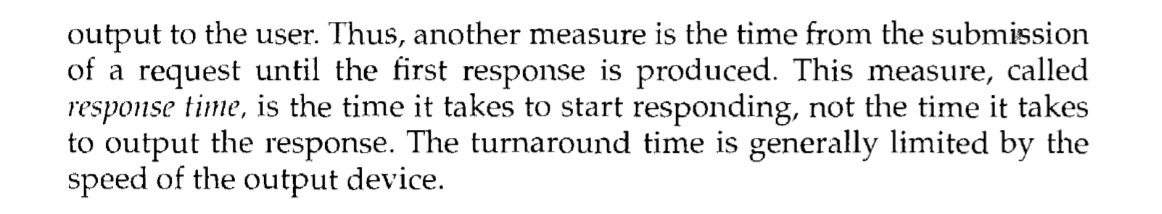
**Describe buffering in process communication**



Textbook chapter 5 problems: 3, 6, 7, 10, 26, 29, 41

**CPU scheduling**

****

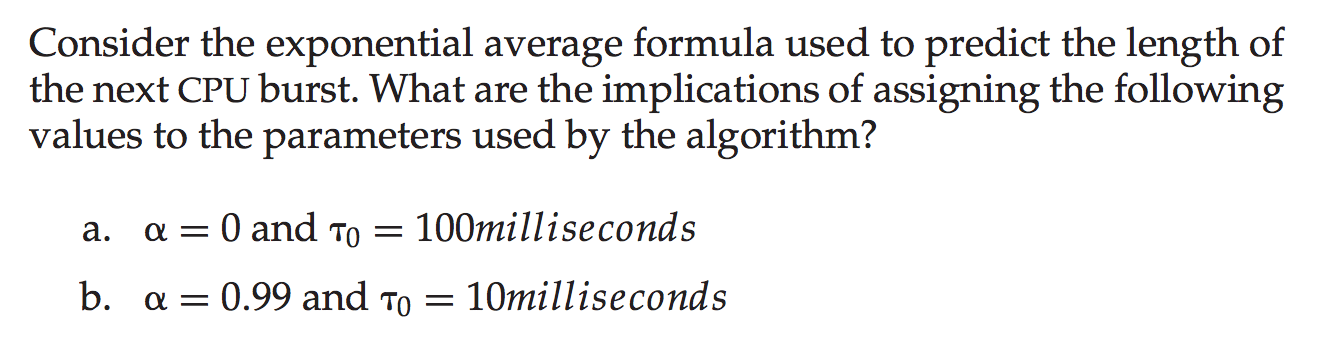
****

**Discuss how the following pairs of scheduling criteria conflict in certain settings.**

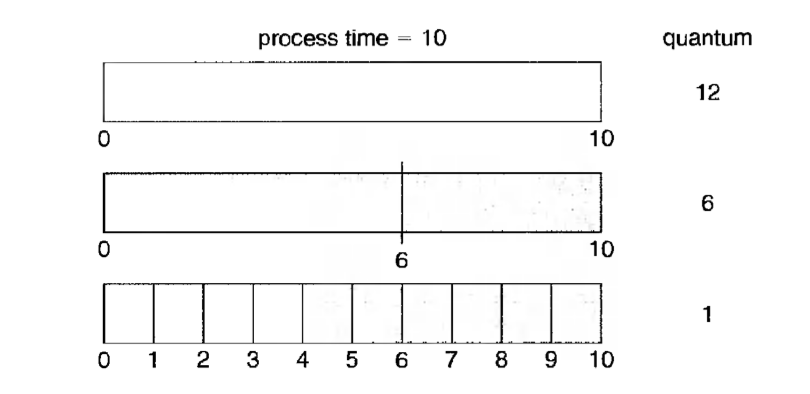
* **CPU utilization and response time**
* **Average turnaround time and maximum waiting time**

• CPU utilization and response time: CPU utilization is increased if the overheads associated with context switching is minimized. The context switching overheads could be lowered by performing context switches infrequently. This could however result in increasing the response time for processes.

• Average turnaround time and maximum waiting time: Average turnaround time is minimized by executing the shortest tasks first. Such a scheduling policy could however starve long-running tasks and thereby increase their waiting time.



**RR**

****

An I/O bound program would typically have what kind of CPU burst?

How should the time quantum be related to the CPU burst times?

Why is it important for the scheduler to distinguish I/O-bound programs from CPU-bound programs?

I/O-bound programs have the property of performing only a small amount of computation before performing IO. Such programs typically do not use up their entire CPU quantum. CPU-bound programs, on the other hand, use their entire quantum without performing any blocking IO operations. Consequently, one could make better use of the computer’s resources by giving higher priority to I/O-bound programs and allow them to execute ahead of the CPU-bound programs.

Exercises

## First-Come, First Serve

* non-preemptive scheduling management
* ready queue is managed as a FIFO queue
* example: 3 jobs arrive at time 0 in the following order (batch processing):

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| Process | Burst Time | Arrival | Start | Wait | Finish | TA |
| 1 | 24 | 0 | 0 | 0 | 24 | 24 |
| 2 | 3 | 0 | 24 | 24 | 27 | 27 |
| 3 | 3 | 0 | 27 | 27 | 30 | 30 |

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| Process | Burst Time | Arrival | Start | Wait | Finish | TA |
| 2 | 3 | 0 | 0 | 0 | 3 | 3 |
| 3 | 3 | 0 | 3 | 3 | 6 | 6 |
| 1 | 24 | 0 | 6 | 6 | 30 | 30 |

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| Process | Burst Time | Arrival | Start | Wait | Finish | TA |
| 1 | 12 | 0 | 0 | 0 | 12 | 12 |
| 2 | 6 | 1 | 12 | 11 | 18 | 17 |
| 3 | 9 | 4 | 18 | 14 | 27 | 23 |

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| Process | Burst Time | Arrival | Start | Wait | Finish | TA |
| 1 | 10 | 0 | 0 | 0 | 10 | 10 |
| 2 | 29 | 0 | 10 | 10 | 39 | 39 |
| 3 | 3 | 0 | 39 | 39 | 42 | 42 |
| 4 | 7 | 0 | 42 | 42 | 49 | 49 |
| 5 | 12 | 0 | 49 | 49 | 61 | 61 |

## Shortest Job First (SJF)

* associate with each process the length of its next CPU burst
* schedule the process with the shortest time
* two schemes
  + non-preemptive: once scheduled, a process continues until the end of its CPU burst
  + preemptive: preempt if a new process arrives with a CPU burst of less length than the remaining time of the currently executing process; known as the *Shortest Remaining Time First* (SRTF) algorithm
* SJF is provably optimal; it yields a minimum average waiting time for any set of processes
* however, we cannot always predict the future (i.e., we do not know the next burst length)
* we can only estimate its length
* an estimate can be formed by using the length of its previous CPU bursts:

*Tn* = actual length of the nth CPU burst

ψn = predicted value of nth CPU burst

0 <= *w* <= 1

ψ*n*+1 = *w* \* *Tn* + (1-*w*) \* ψn

## SJF (non-preemptive) examples

* example 1:

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| Process | Burst Time | Arrival | Start | Wait | Finish | TA |
| 1 | 6 | 0 | 3 | 3 | 9 | 9 |
| 2 | 8 | 0 | 16 | 16 | 24 | 24 |
| 3 | 7 | 0 | 9 | 9 | 16 | 16 |
| 4 | 3 | 0 | 0 | 0 | 3 | 3 |

* example 2:

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| Process | Burst Time | Arrival | Start | Wait | Finish | TA |
| 1 | 7 | 0 | 0 | 0 | 7 | 7 |
| 2 | 4 | 2 | 8 | 6 | 12 | 10 |
| 3 | 1 | 4 | 7 | 3 | 8 | 4 |
| 4 | 4 | 5 | 12 | 7 | 16 | 11 |

* example 3:

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| Process | Burst Time | Arrival | Start | Wait | Finish | TA |
| 1 | 10 | 0 | 10 | 10 | 20 | 20 |
| 2 | 29 | 0 | 32 | 32 | 61 | 61 |
| 3 | 3 | 0 | 0 | 0 | 3 | 3 |
| 4 | 7 | 0 | 3 | 3 | 10 | 10 |
| 5 | 12 | 0 | 20 | 20 | 32 | 32 |

## SJF (preemptive) examples

* example 1:

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| Process | Burst Time | Arrival | Start | Wait | Finish | TA |
| 1 | 8 | 0 | 0 | 9 | 17 | 17 |
| 2 | 4 | 1 | 1 | 0 | 5 | 4 |
| 3 | 9 | 2 | 17 | 15 | 26 | 24 |
| 4 | 5 | 3 | 5 | 2 | 10 | 7 |

* example 2:

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| Process | Burst Time | Arrival | Start | Wait | Finish | TA |
| 1 | 7 | 0 | 0 | 9 | 16 | 16 |
| 2 | 4 | 2 | 2 | 1 | 7 | 5 |
| 3 | 1 | 4 | 4 | 0 | 5 | 1 |
| 4 | 4 | 5 | 7 | 2 | 11 | 6 |

## Priority Scheduling

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| Process | Burst Time | Priority | Arrival | Start | Wait | Finish | TA |
| 1 | 10 | 3 | 0 | 6 | 6 | 16 | 16 |
| 2 | 1 | 1 | 0 | 0 | 0 | 1 | 1 |
| 3 | 2 | 4 | 0 | 16 | 16 | 18 | 18 |
| 4 | 1 | 5 | 0 | 18 | 18 | 19 | 19 |
| 5 | 5 | 2 | 0 | 1 | 1 | 6 | 6 |

## Round Robin

* time sharing (preemptive) scheduler where each process is given access to the CPU for 1 time quantum (slice) (e.g., 20 milliseconds)
* a process may block itself before its time slice expires
* if it uses its entire time slice, it is then preempted and put at the end of the ready queue
* the ready queue is managed as a FIFO queue and treated as a circular
* if there are *n* processes on the ready queue and the time quantum is *q*, then each process gets 1/*n* time on the CPU in chunks of at most *q* time units
* no process waits for more than (*n*-1)*q* time units
* the choice of how big to make the time slice (*q*) is extremely important
  + if *q* is very large, Round Robin degenerates into FCFS
  + if *q* is very small, the context switch overhead defeats the benefits
* example 1 (*q* = 20):

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| Process | Burst Time | Arrival | Start | Wait | Finish | TA |
| 1 | 53 | 0 | 0 | ? | 134 | 134 |
| 2 | 17 | 0 | 20 | ? | 37 | 37 |
| 3 | 68 | 0 | 37 | ? | 162 | 162 |
| 4 | 24 | 0 | 57 | ? | 121 | 121 |

* example 2 (*q* = 4):

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| Process | Burst Time | Arrival | Start | Wait | Finish | TA |
| 1 | 24 | 0 | 0 | 6 | 30 | 30 |
| 2 | 3 | 0 | 4 | 4 | 7 | 7 |
| 3 | 3 | 0 | 7 | 7 | 10 | 10 |

* example 3 (*q* = 10):

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| Process | Burst Time | Arrival | Start | Wait | Finish | TA |
| 1 | 10 | 0 | 0 | 0 | 10 | 10 |
| 2 | 29 | 0 | 10 | 32 | 61 | 61 |
| 3 | 3 | 0 | 20 | 20 | 23 | 23 |
| 4 | 7 | 0 | 23 | 23 | 30 | 30 |
| 5 | 12 | 0 | 30 | 40 | 52 | 52 |

Answers

Each scheduling algorithm favours particular criteria:

* *CPU utilization* (maximize)
* *throughput*: number of processes which complete execution per time unit (maximize)
* *turnaround time* (TA): total amount of time to execute a particular process (minimize)
* *waiting time*: amount of time a process has been waiting in the ready queue (minimize)
* *response time*: amount of time it takes from when a request is submitted to when the response is produced (minimize); does not include the time for a response to be output

Some work is being done to minimize response time variance, to promote predictability.

## CPU Scheduling Algorithms

* First-Come, First Serve (FCFS or FIFO) (non-preemptive)
* Priority (e.g., Shortest Job First (SJF; non-preemptive)

or Shortest Remaining Time First (SRTF; preemptive))

* Round Robin (preemptive)
* Multi-level Queue
* Multi-level Feedback Queue

## First-Come, First Serve

* non-preemptive scheduling management
* ready queue is managed as a FIFO queue
* example: 3 jobs arrive at time 0 in the following order (batch processing):

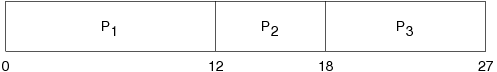
|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| Process | Burst Time | Arrival | Start | Wait | Finish | TA |
| 1 | 24 | 0 | 0 | 0 | 24 | 24 |
| 2 | 3 | 0 | 24 | 24 | 27 | 27 |
| 3 | 3 | 0 | 27 | 27 | 30 | 30 |

* Gantt chart:   
    
     
    
  (regenerated from [OSC8] p. 189)  
  (regenerated from [OSCJ8] p. 199)
* average waiting time: (0+24+27)/3 = 17
* average turnaround time: (24+27+30)/3 = 27
* consider arrival order: 2, 3, 1

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| Process | Burst Time | Arrival | Start | Wait | Finish | TA |
| 2 | 3 | 0 | 0 | 0 | 3 | 3 |
| 3 | 3 | 0 | 3 | 3 | 6 | 6 |
| 1 | 24 | 0 | 6 | 6 | 30 | 30 |

* Gantt chart:   
    
     
    
  (regenerated from [OSC9] p. 189)  
  (regenerated from [OSCJ8] p. 199)
* average waiting time: (0+3+6)/3 = 3
* average turnaround time: (3+6+30) = 13
* another example:

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| Process | Burst Time | Arrival | Start | Wait | Finish | TA |
| 1 | 12 | 0 | 0 | 0 | 12 | 12 |
| 2 | 6 | 1 | 12 | 11 | 18 | 17 |
| 3 | 9 | 4 | 18 | 14 | 27 | 23 |

* Gantt chart:   
    
  
* average waiting time: (0+11+14)/3 = 8.33
* average turnaround time: (12+17+23) = 52/3 = 17.33
* another example:

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| Process | Burst Time | Arrival | Start | Wait | Finish | TA |
| 1 | 10 | 0 | 0 | 0 | 10 | 10 |
| 2 | 29 | 0 | 10 | 10 | 39 | 39 |
| 3 | 3 | 0 | 39 | 39 | 42 | 42 |
| 4 | 7 | 0 | 42 | 42 | 49 | 49 |
| 5 | 12 | 0 | 49 | 49 | 61 | 61 |

* Gantt chart:   
    
     
    
  (regenerated from [OSC8] p. 214)  
  (regenerated from [OSCJ8] p. 229)
* average waiting time: (0+10+39+42+49)/5 = 28
* average turnaround time: (10+39+42+49+61)/5 = 40.2

## Priority Scheduling

* associate a priority with each process, allocate the CPU to the process with the highest priority
* any 2 processes with the same priority are handled FCFS
* SJF is a version of priority scheduling where the priority is defined using the predicted CPU burst length
* priorities are usually numeric over a range
* high numbers may indicate low priority (system dependent)
* internal (process-based) priorities: time limits, memory requirements, resources needed, burst ratio
* external (often political) priorities: importance, source (e.g., faculty, student)
* priority scheduling can be non-preemptive or preemptive
* problem: *starvation* --- low priority processes may never execute because they are waiting indefinitely for the CPU
* a solution: *aging* --- increase the priority of a process as time progresses
* nice in UNIX executes a utility with an altered scheduling priority
* renice in UNIX alters the priority of running processes

## Shortest Job First (SJF)

* associate with each process the length of its next CPU burst
* schedule the process with the shortest time
* two schemes
  + non-preemptive: once scheduled, a process continues until the end of its CPU burst
  + preemptive: preempt if a new process arrives with a CPU burst of less length than the remaining time of the currently executing process; known as the *Shortest Remaining Time First* (SRTF) algorithm
* SJF is provably optimal; it yields a minimum average waiting time for any set of processes
* however, we cannot always predict the future (i.e., we do not know the next burst length)
* we can only estimate its length
* an estimate can be formed by using the length of its previous CPU bursts:

*Tn* = actual length of the nth CPU burst

ψn = predicted value of nth CPU burst

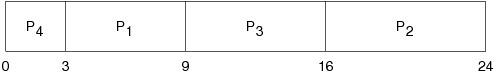
0 <= *w* <= 1

ψ*n*+1 = *w* \* *Tn* + (1-*w*) \* ψn

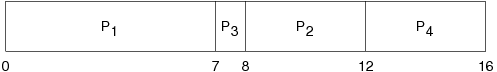
## SJF (non-preemptive) examples

* example 1:

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| Process | Burst Time | Arrival | Start | Wait | Finish | TA |
| 1 | 6 | 0 | 3 | 3 | 9 | 9 |
| 2 | 8 | 0 | 16 | 16 | 24 | 24 |
| 3 | 7 | 0 | 9 | 9 | 16 | 16 |
| 4 | 3 | 0 | 0 | 0 | 3 | 3 |

* Gantt chart:   
    
     
    
  (regenerated from [OSC8] p. 190)  
  (regenerated from [OSCJ8] p. 200)
* average waiting time: (3+16+9+0)/4 = 7
* average turnaround time: (9+24+16+3)/4 = 13
* example 2:

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| Process | Burst Time | Arrival | Start | Wait | Finish | TA |
| 1 | 7 | 0 | 0 | 0 | 7 | 7 |
| 2 | 4 | 2 | 8 | 6 | 12 | 10 |
| 3 | 1 | 4 | 7 | 3 | 8 | 4 |
| 4 | 4 | 5 | 12 | 7 | 16 | 11 |

* Gantt chart:   
    
  
* average waiting time: (0+6+3+7)/4 = 4
* average turnaround time: (7+4+10+11)/4 = 8
* example 3:

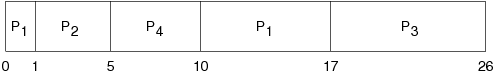
|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| Process | Burst Time | Arrival | Start | Wait | Finish | TA |
| 1 | 10 | 0 | 10 | 10 | 20 | 20 |
| 2 | 29 | 0 | 32 | 32 | 61 | 61 |
| 3 | 3 | 0 | 0 | 0 | 3 | 3 |
| 4 | 7 | 0 | 3 | 3 | 10 | 10 |
| 5 | 12 | 0 | 20 | 20 | 32 | 32 |

* Gantt chart:   
    
     
    
  (regenerated from [OSC8] p. 214)  
  (regenerated from [OSCJ8] p. 229)
* average waiting time: (10+32+0+3+20)/5 = 13
* average turnaround time: (10+39+42+49+61)/5 = 25.2

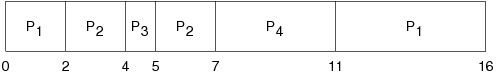
## SRTF (preemptive) examples

* example 1:

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| Process | Burst Time | Arrival | Start | Wait | Finish | TA |
| 1 | 8 | 0 | 0 | 9 | 17 | 17 |
| 2 | 4 | 1 | 1 | 0 | 5 | 4 |
| 3 | 9 | 2 | 17 | 15 | 26 | 24 |
| 4 | 5 | 3 | 5 | 2 | 10 | 7 |

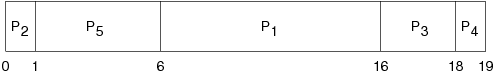
* Gantt chart:   
    
     
    
  (regenerated from [OSC8] p. 192)  
  (regenerated from [OSCJ8] p. 202)
* average waiting time: (9+0+15+2)/4 = 6.5
* average turnaround time: (17+4+24+7)/4 = 13
* example 2:

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| Process | Burst Time | Arrival | Start | Wait | Finish | TA |
| 1 | 7 | 0 | 0 | 9 | 16 | 16 |
| 2 | 4 | 2 | 2 | 1 | 7 | 5 |
| 3 | 1 | 4 | 4 | 0 | 5 | 1 |
| 4 | 4 | 5 | 7 | 2 | 11 | 6 |

* Gantt chart:   
    
  
* average waiting time: (9+1+0+2)/4 = 3
* average turnaround time: (16+5+1+6)/4 = 7

## Priority Scheduling example

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| Process | Burst Time | Priority | Arrival | Start | Wait | Finish | TA |
| 1 | 10 | 3 | 0 | 6 | 6 | 16 | 16 |
| 2 | 1 | 1 | 0 | 0 | 0 | 1 | 1 |
| 3 | 2 | 4 | 0 | 16 | 16 | 18 | 18 |
| 4 | 1 | 5 | 0 | 18 | 18 | 19 | 19 |
| 5 | 5 | 2 | 0 | 1 | 1 | 6 | 6 |

Gantt chart:   
  
   
  
(regenerated from [OSC8] p. 193)  
(regenerated from [OSCJ8] p. 203)

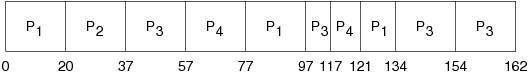
average waiting time: (6+0+16+18+1)/5 = 8.2

average turnaround time: (1+6+16+18+19)/5 = 12

## Round Robin

* time sharing (preemptive) scheduler where each process is given access to the CPU for 1 time quantum (slice) (e.g., 20 milliseconds)
* a process may block itself before its time slice expires
* if it uses its entire time slice, it is then preempted and put at the end of the ready queue
* the ready queue is managed as a FIFO queue and treated as a circular
* if there are *n* processes on the ready queue and the time quantum is *q*, then each process gets 1/*n* time on the CPU in chunks of at most *q* time units
* no process waits for more than (*n*-1)*q* time units
* the choice of how big to make the time slice (*q*) is extremely important
  + if *q* is very large, Round Robin degenerates into FCFS
  + if *q* is very small, the context switch overhead defeats the benefits
* example 1 (*q* = 20):

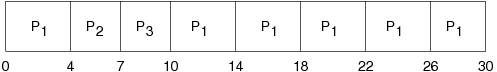
|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| Process | Burst Time | Arrival | Start | Wait | Finish | TA |
| 1 | 53 | 0 | 0 | ? | 134 | 134 |
| 2 | 17 | 0 | 20 | ? | 37 | 37 |
| 3 | 68 | 0 | 37 | ? | 162 | 162 |
| 4 | 24 | 0 | 57 | ? | 121 | 121 |

* Gantt chart:   
    
  
* waiting times:
* p1: (77-20) + (121-97) = 81
* p2: (20-0) = 20
* p3: (37-0) + (97-57) + (134-117) = 94
* p4: (57-0) + (117-77) = 97
* average waiting time: (81+20+94+97)/4 = 73

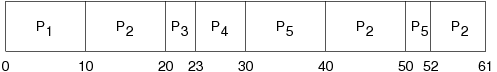
[adelaideneocs@gmail.com](mailto:adelaideneocs@gmail.com)

* example 2 (*q* = 4):

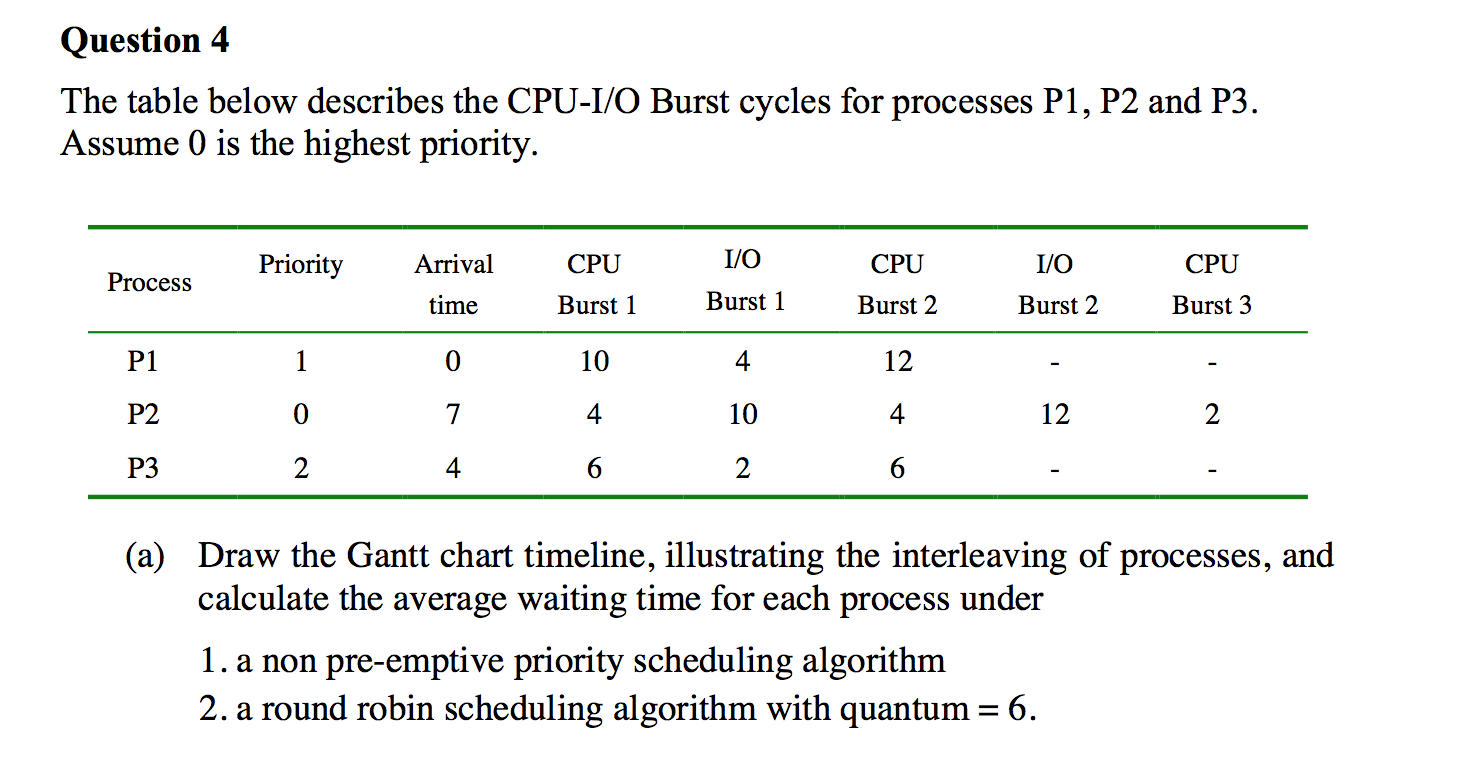
|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| Process | Burst Time | Arrival | Start | Wait | Finish | TA |
| 1 | 24 | 0 | 0 | 6 | 30 | 30 |
| 2 | 3 | 0 | 4 | 4 | 7 | 7 |
| 3 | 3 | 0 | 7 | 7 | 10 | 10 |

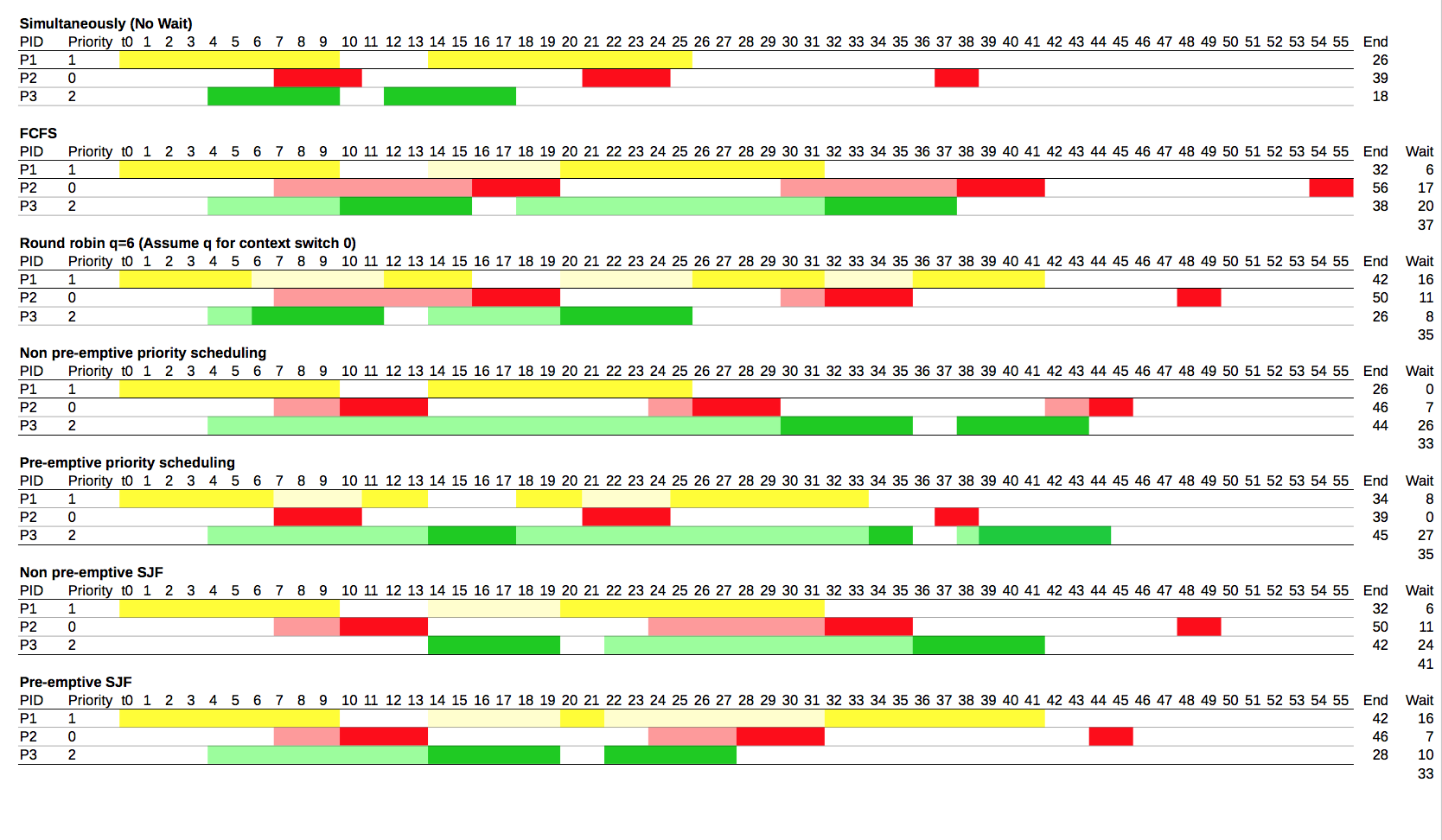
* Gantt chart:   
    
     
    
  (regenerated from [OSC8] p. 194) (regenerated from [OSCJ8] p. 204)
* average waiting time: (6+4+7)/3 = 5.67
* average turnaround time: (30+7+10)/3 = 15.67
* example 3 (*q* = 10):

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| Process | Burst Time | Arrival | Start | Wait | Finish | TA |
| 1 | 10 | 0 | 0 | 0 | 10 | 10 |
| 2 | 29 | 0 | 10 | 32 | 61 | 61 |
| 3 | 3 | 0 | 20 | 20 | 23 | 23 |
| 4 | 7 | 0 | 23 | 23 | 30 | 30 |
| 5 | 12 | 0 | 30 | 40 | 52 | 52 |

* Gantt chart:   
    
     
    
  (regenerated from [OSC8] p. 214)  
  (regenerated from [OSCJ8] p. 229)
* average waiting time: (0+32+20+23+40)/5 = 23
* average turnaround time: (10+39+42+49+61)/5 = 35.2

## 



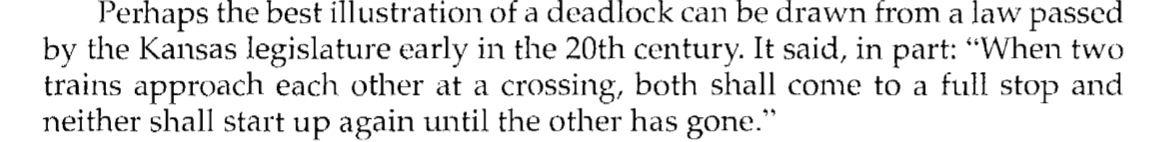


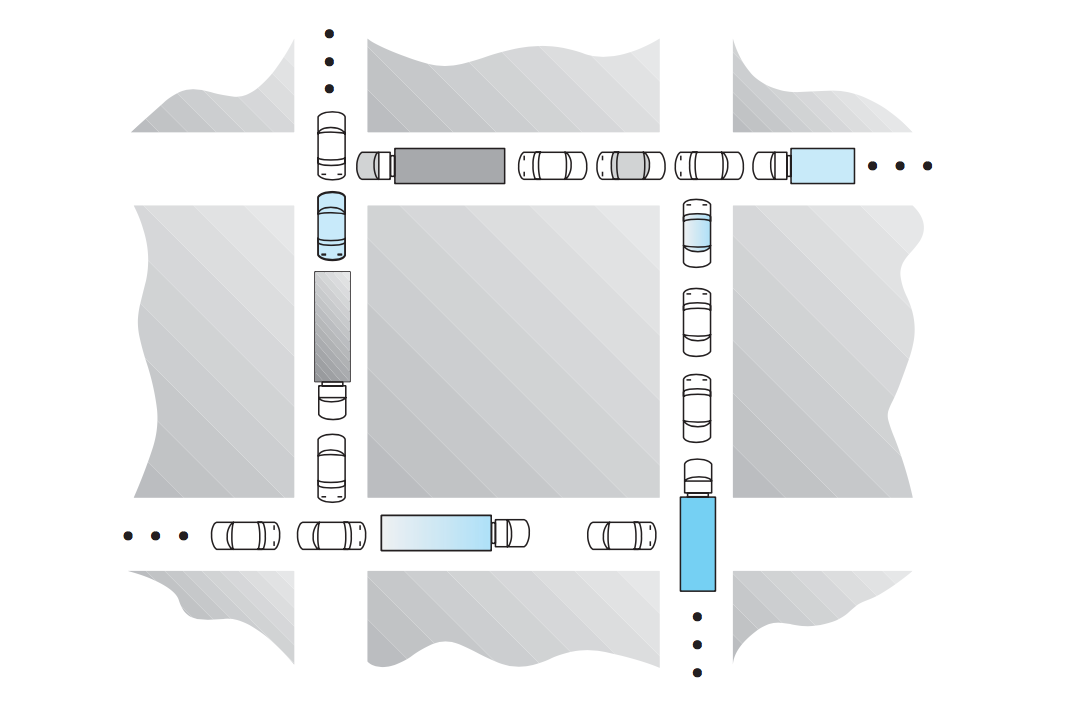
<http://perugini.cps.udayton.edu/teaching/courses/cps346/lecture_notes/scheduling.html>

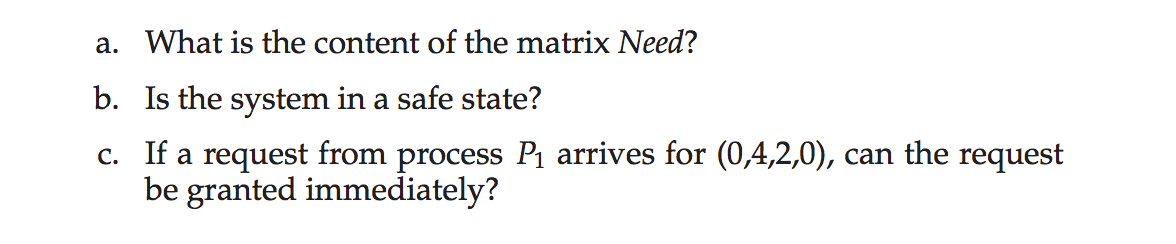
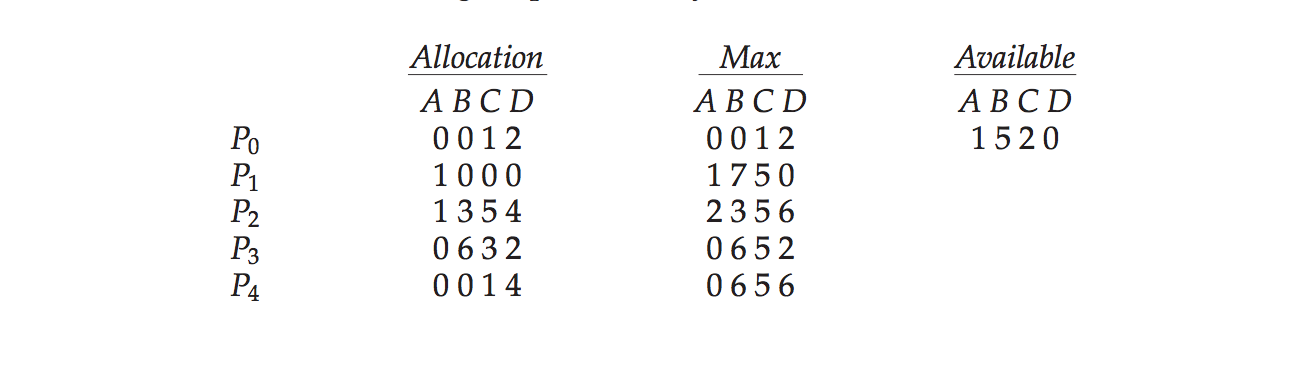
Textbook chapter 6 problems: 2, 4, 6, 11, 13, 16, 17, 19, 21

**Deadlock**

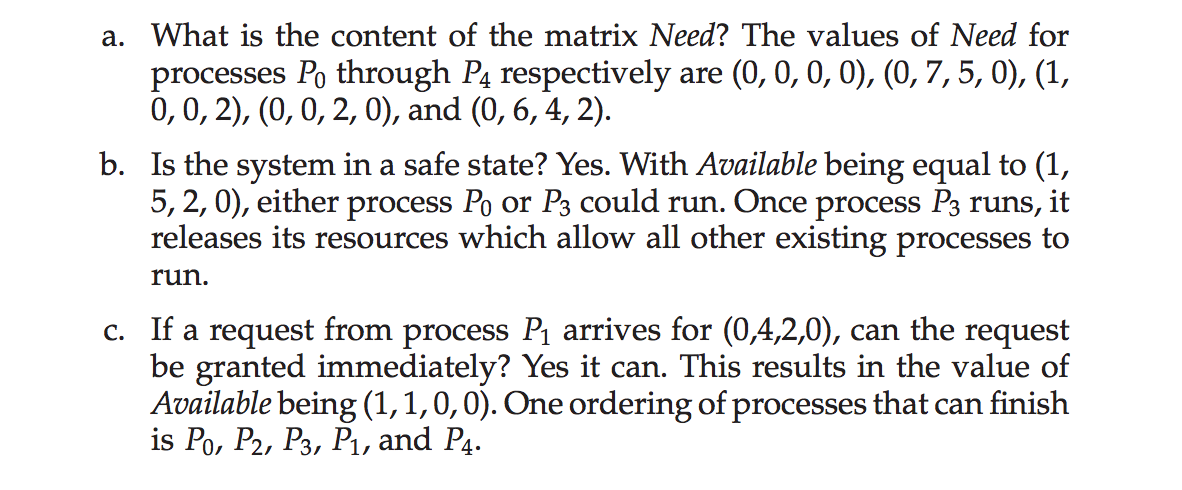
Give an example of a deadlock that could occur in the physical world.

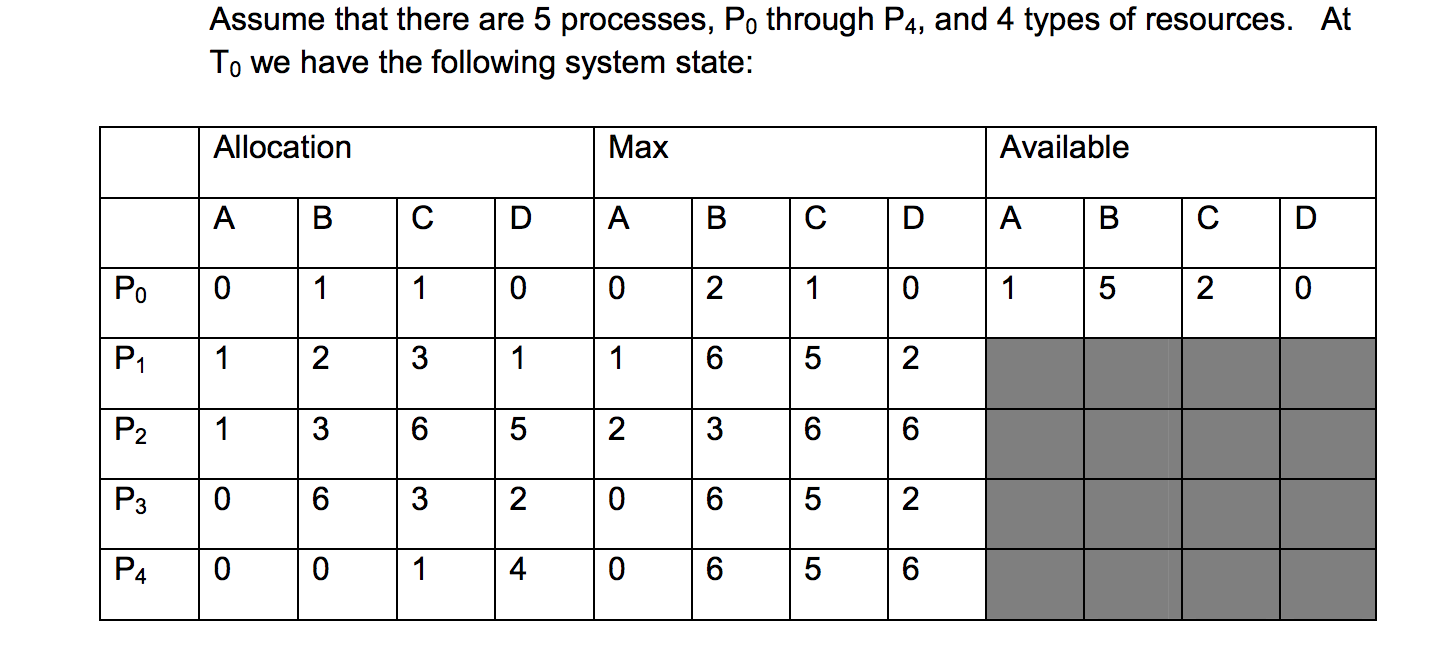


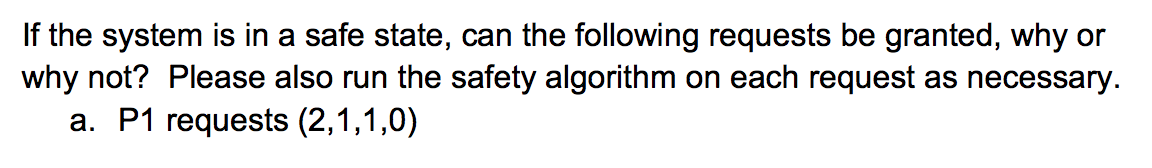


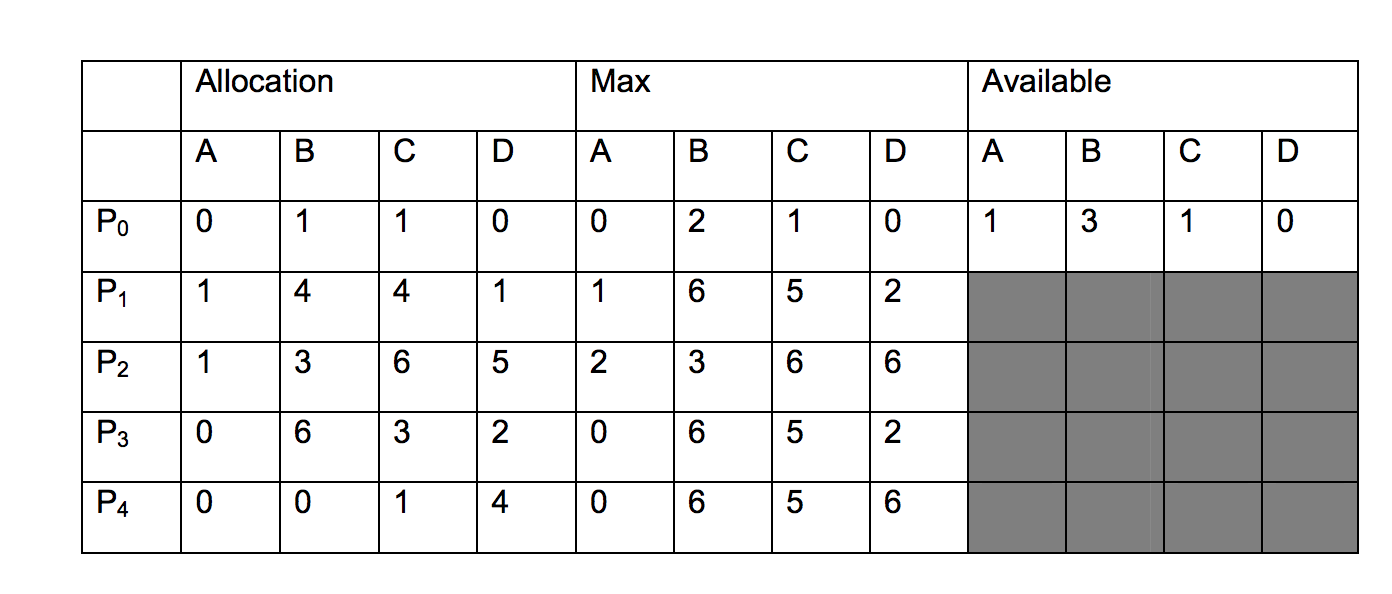


answer:









The run-time mapping from virtual to physical addresses is done by a hardware device called the MMU

Contiguous allocation

first fit, best fit, worst fit

First fit: Allocate the *first* hole that is big enough. Searching can start either at the beginning of the set of holes or at the location where the previous first-fit search ended. We can stop searching as soon as we find a free hole that is large enough.

Best fit: Allocate the *smallest* hole that is big enough. We must search the entire list, unless the list is ordered by size. This strategy produces the smallest leftover hole.

Worst fit: Allocate the *largest* hole. Again, we must search the entire list, unless it is sorted by size. This strategy produces the largest leftover hole, which may be more useful than the smaller leftover hole from a best-fit approach.

Simulations have shown that both first fit and best fit are better than worst fit in terms of decreasing time and storage utilization. Neither first fit nor best fit is clearly better than the other in terms of storage utilization, but first fit is generally faster.

Given five memory partitions of 100 KB, 500 KB, 200 KB, 300 KB, and 600 KB (in order), how would each of the first-fit, best-fit, and worst-fit algorithms place processes of 212 KB, 417 KB, 112 KB, and 426 KB (in order)? Which algorithm makes the most efficient use of memory?

* 1. Best-fit:
  2. 212K is put in 300K partition
  3. 417K is put in 500K partition
  4. 112K is put in 200K partition
  5. 426K is put in 600K partition
  6. Worst-fit:
  7. 212K is put in 600K partition
  8. 417K is put in 500K partition
  9. 112K is put in 388K partition
  10. 426K must wait

In this example, Best-fit turns out to be the best.

Paging

What happens on a 32-bit machine with 256MB of RAM and 4KB pages?

For each configuration (a-c), state how many bits are needed for each of the following:

• Virtual address

• Physical address

• Virtual page number

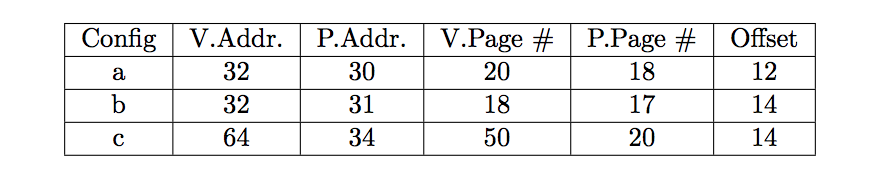
• Physical page number

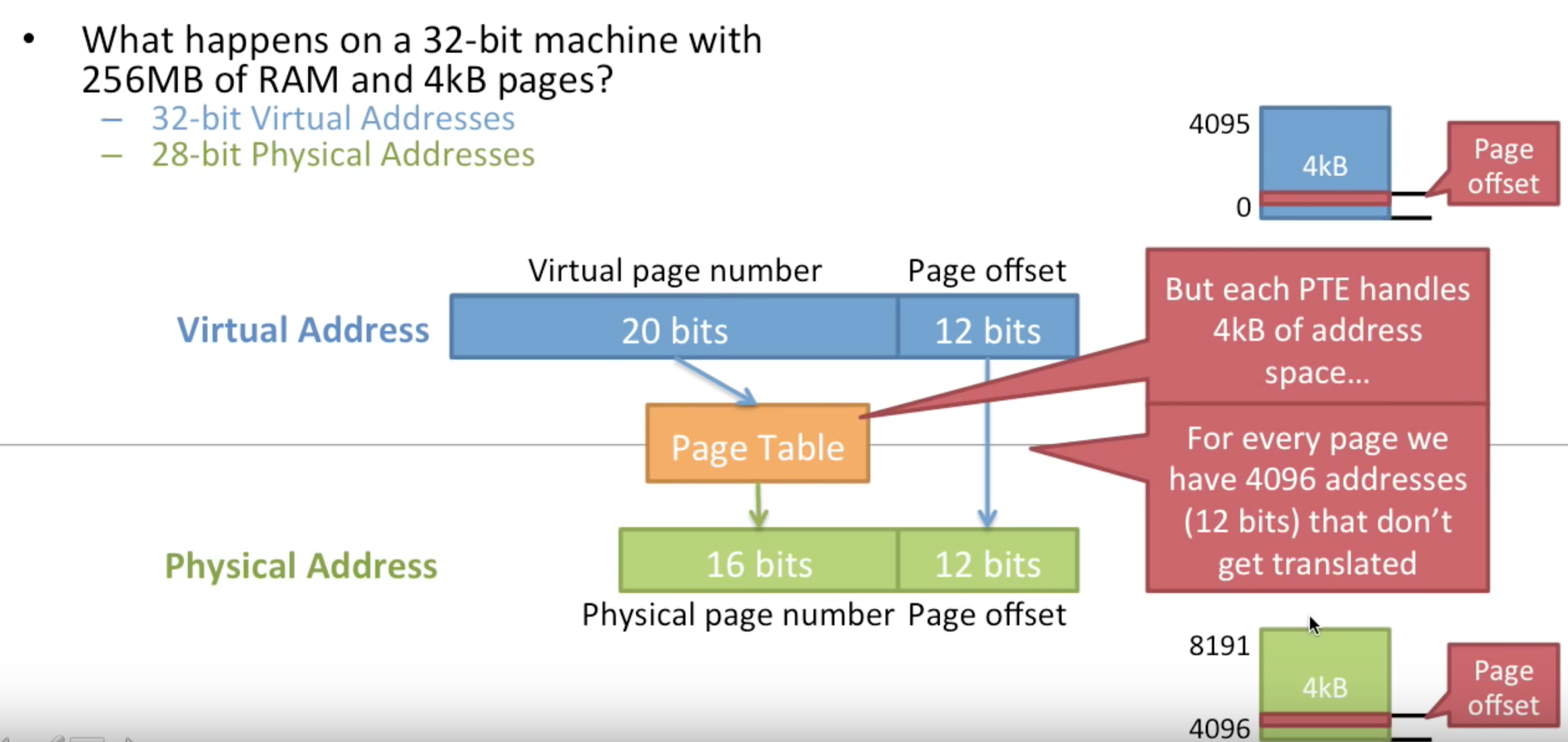
• Offset

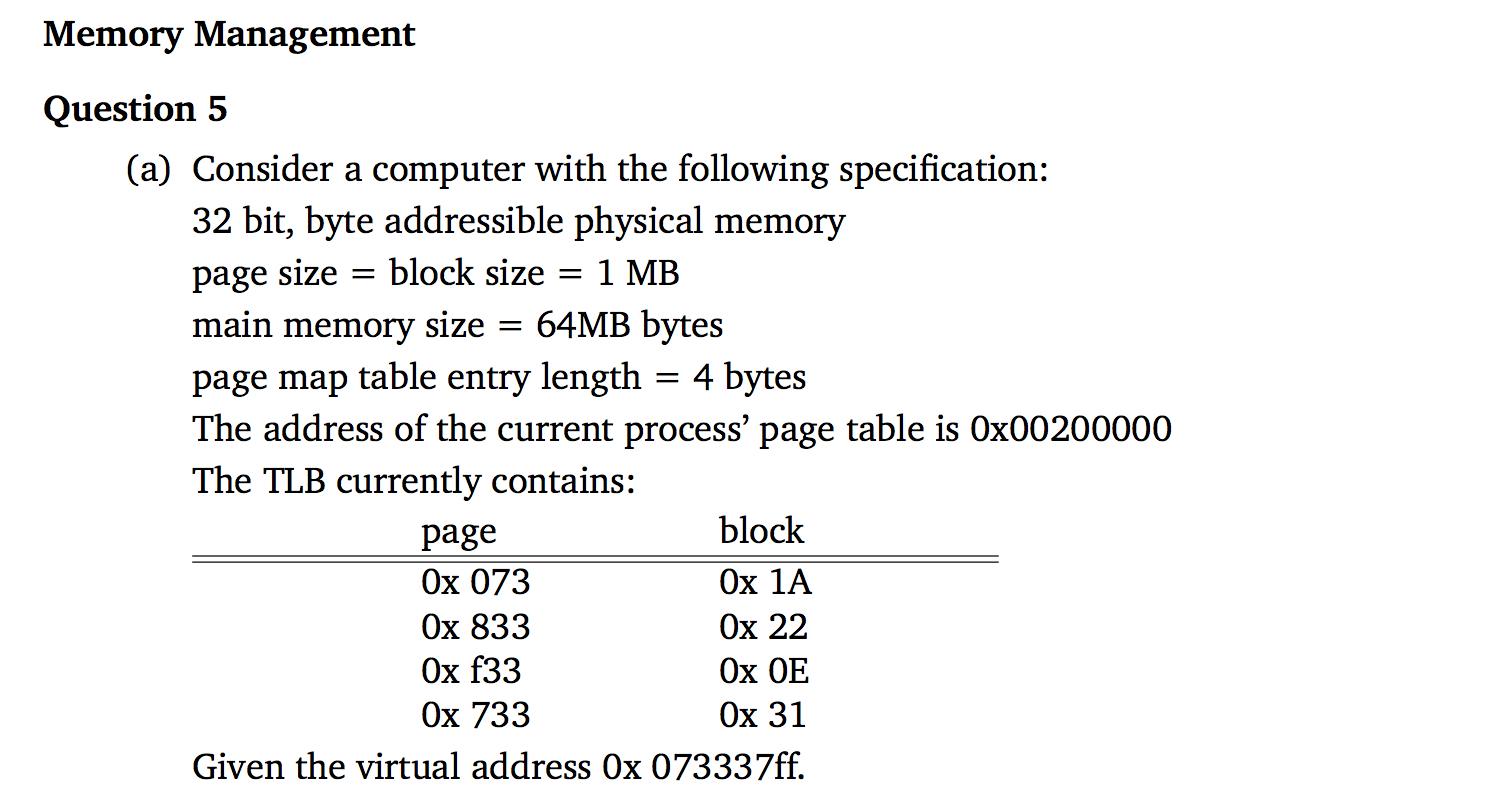
a. 32-bit operating system, 4-KB pages, 1 GB of RAM

b. 32-bit operating system, 16-KB pages, 2 GB of RAM

c. 64-bit operating system, 16-KB pages, 16 GB of RAM







* What is the page number and offset of this virtual address (VA)?
* How many blocks are in the main memory?
* What is the physical address?

TLB

Consider a paging system with the page table stored in memory.

* 1. If a memory reference takes 200 nanoseconds, how long does a  paged memory reference take?
  2. If we add associative registers, and 75 percent of all page-table references are found in the associative registers, what is the effective memory reference time? (Assume that finding a page-table entry in the associative registers takes zero time, if the entry is there.)

**Answer:**

* 1. 400 nanoseconds; 200 nanoseconds to access the page table and 200 nanoseconds to access the word in memory.
  2. Effective access time = 0.75 × (200 nanoseconds) + 0.25 × (400 nanoseconds) = 250 nanoseconds.

Segmentation

## Key Differences Between Paging and Segmentation

1. The basic difference between paging and segmentation is that a page is always of **fixed block size** whereas, a segment is of **variable size**.
2. Paging may lead to **internal fragmentation** as the page is of fixed block size, but it may happen that the process does not acquire the entire block size which will generate the internal fragment in memory. The segmentation may lead to **external fragmentation** as the memory is filled with the variable sized blocks.

Consider the following segment table:

Segment Base Length

Segment Base Length

0 219 600

1 2300 14

2 90 100

3 1327 580

4 1952 96

What are the physical addresses for the following logical addresses?

a. 0, 430 b. 1, 10 c. 2, 500 d. 3, 400 e. 4, 112

**Answer:**

* 219+430=649
* 2300+10=2310
* illegal reference, trap to operating system
* 1327 + 400 = 1727
* illegal reference, trap to operating system



**Chapters 8-9 Memory (session 6, 7, 8)**

**Keywords:** relocation, allocation, free list, first fit, next fit, best fit, worst fit, quick fit, page, page frame, MMU, page fault, page table, multilevel page table, TLB, inverted page table, page replacement, NRU, FIFO, LRU, second chance, NFU, again, working set, thrashing, prepaging, fragmentation, segment, segment table.

**Memory management**

1. What is swapping?
2. What is compaction? Why use it?
3. What is a frame?
4. What is contained in the page table?
5. How much fragmentation occurs with paging? Which type?
6. Why are segmentation and paging sometimes combined into one scheme?
7. What is a page fault?
8. What is page replacement? Ideally, what criteria we use to replace pages?
9. Does LRU require extra hardware?
10. What is mean by locality?
11. Describe the concept of pre-paging and explain its advantages relative to pure demand  paging

Textbook chapter 8 problems: 1, 3, 4, 9, 11, 12, 17, 20, 23, 25, 28 Textbook chapter 9 problems: 1, 3, 4, 6, 8, 12, 14, 15, 18, 19, 21, 22, 31

**Chapter 10-12 File systems (session 9)**

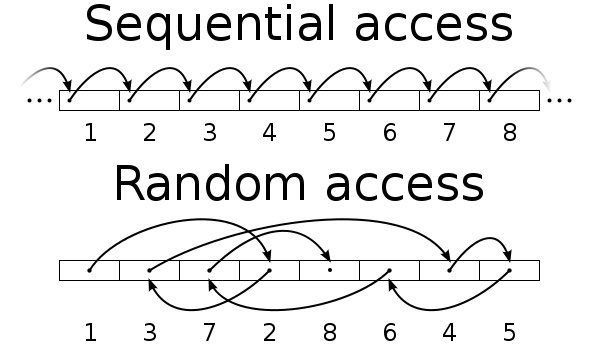
**Keywords**: file extension, file structure, file types, file attributes, sequential access, random access, file operations, directories, absolute path, relative path, working directory, contiguous allocation, linked list allocation, indexed, i-node, hard link, symbolic link, reliability, FS consistency, data integrity, denial of service, intruders, malware, authentication, passwords, encryption, protection domain, rights, protection matrix, access control list, capabilities, generic rights

**File system**

**Os-09**

"Random read" means you can get any part of the file in any order. So for example, you can read the middle part before the start.

"Sequential read" means you must first read the first part of the file, before reading second, then third etc. figure below explains.

[](http://kb.sandisk.com/euf/assets/images/faqs/8150/id8150_Random_vs_sequential_access.png)

1. What is a file? 2. Can a direct access file be read sequentially? Please, explain 3. What does OPEN do? 4. List ways to share files between directories in operating systems. 5. Rank the allocation methods on speed.

6. Why do some systems keep track of the type of file, while others leave it to the user or simply do not implement multiple file types? Which system is better?

7. Consider a system that supports the strategies of contiguous, linked and indexed allocation. What criteria should be used in deciding which strategy is best utilized for a particular file?

Textbook chapter 11 problems: 5, 7, 9, 12, 14, 16 Textbook chapter 12 problems: 3, 4, 6, 10, 13, 15, 16, 19 Textbook chapter 13 problems: 5, 9, 10

**Chapters 13 I/O Management (session 10)**

**Keywords**: block device, character device, device-independent, device controller, memory mapped I/O, DMA, buffering, spooling, device driver, standard driver interface, major device number, minor device number, disk partitions, subpartitions, RAM disk, disk scheduling, elevator algorithm, bad block, caching

**I/O**

1. What is memory-mapped I/O?
2. Explain what DMA is and why it is used
3. Although DMA does not use the CPU, the maximum transfer rate is still limited. Consider  reading a block from disk. Name three factors that might ultimately limit the rate of transfer.
4. Disk controllers have internal buffers and they are getting larger with each model. Why?
5. What is a spool?

**Disk**

1. Disk requests come in to the driver from cylinders 10,22,20,2,40,6 and 38 in that order. The arm is initially at cylinder 20 and a seek takes 5 msec per cylinder moved. How much seek time is needed for (a) FCFS, (b) SSF and (c) elevator algorithm which is moving upwards
2. How do bad blocks impact on disk performance?
3. In MINIX, device drivers are implemented in user space, but some of its work needs to be done in supervisor mode. Explain how this is implemented.
4. List and briefly described the six operations that are supported by a MINIX block device driver.
5. Compare the performance of C-SCAN and SCAN scheduling, assuming a uniform distribution of requests. Consider the average response time (the time between the arrival of a request and the completion of that request’s service), the variation in response time, and the effective bandwidth. How does performance depend on the relative sizes of seek time and rotational latency?