Spring 2015 COMP 3511 Operating Systems

Midterm Examination Solutions

Date: March 31, 2015 (Tuesday) Time: 19:00 - 20:30

Name: _____ Student ID: _____

Email: _____ Lecture Section: _____

Note: Please write your name, student ID, and email address on this page. Read the following instructions carefully.

1. This is a **Closed-Book Exam**.
2. This examination solution consists of 4 questions and 9 pages (including this page).
3. You have 90 minutes to complete this exam.
4. Answer all questions within the space provided on the examination paper. You may use back of the pages for your rough work. Please be concise! This is NOT an essay contest.
5. Please read each question very carefully and answer the question clearly to the point.
6. Make sure that your answers are neatly written and legible.
7. Show all the steps used in deriving your answer, wherever appropriate.

Question	Points	Score
1	10	
2	20	
3	40	
4	30	
Total	100	

1. [10 points] Please answer the following questions in a few sentences.

   1.1. What do we refer as *policy* and *mechanism* in OS design? Please explain briefly why we need to separate them. (3 points)

   Answer: The *policy* defines what needs to be done, while the *mechanism* specifies how it is actually implemented (2 points). This separation is necessary for achieving flexibility so to minimize the changes needed when either the policy or mechanism is changed (1 point).

   1.2. What is a *dual mode operation* in an operating system? Please briefly explain why OS provides a dual mode operation. (3 points)

   Answer: Kernel and user modes (1 point). This provides a basic form of protection in that certain instructions could only be executed when the CPU is executing in kernel mode, and it prevents users from accessing critical resources directly so to avoid misuse. (2 points)

   1.3. What is *emulation*? What is *virtualization*? (4 points)

   Answer: *Emulation* is used when the source CPU type is different from the target CPU type, which allows applications complied for a source CPU to run on a target CPU (2 points). *Virtualization* is a technology that allows an operating system to run as an application within another operating system (2 points).

2. [20 points] Process and Thread

   2.1. Can you describe three elements contained in a Thread Control Block (TCB). (3 points)

   Answer: Thread ID, scheduling information, program counters, stack, registers, thread local storage. (1 point each)

   2.2. In a multi-threaded process, in which a process can have multiple threads, please name two ways that a signal can be delivered. (2 points)

   Answer: Deliver to a particular thread, to all threads, selected threads, or arrange a special thread responsible for receiving all signals. (1 point each)

   2.3. Can you please name two differences between an *ordinary pipe* and a *name pipe* in Unix? (2 points)

   Answer: an *ordinary pipe* is unidirectional and a name pipe is bi-directional; a *name pipe* allows multiple processes to communicate with it while an ordinary pipe only allows communications between a parent and a child (two processes). (1 point each)

   2.4. What is a *zombie process* in Unix? How does UNIX handle a zombie process? (3 points)

   Answer: A process that has terminated, but whose parent has not yet called `wait()`. This is known as a *zombie process*. (2 points) The Unix does not do anything as the parent is expected to call `wait()` later. (1 point)

2.5. What is the total number of processes in the following code? Please elaborate. (5 points)

```
#include<sys/types.h>
#include<stdio.h>
#include<unistd.h>

int main(){
   if(!fork()){
      if(fork())
         fork();
   }
   else
      fork();
}
```

Answer: There are 5 processes. Under all the branches, each "`fork()`" would create only one more process. And there are 4 "`fork()`", hence the total number of processes is 5.

2.6. In the following program, explain what will be output at Line A and B. Please justify your answer. (5 points)

```
#include<sys/types.h>
#include<stdio.h>
#include<unistd.h>

int number1 = 5;
int number2 = 10;

int main()
{
    pid_t pid;
    int tmp;
    pid = fork();
    if(pid == 0){
           tmp = number1;
           number1 = number2;
           number2 = tmp;
           printf("CHILD: number1 = %d\n", number1);/*Line A*/
    }
    else{
           wait(NULL);
           printf("PARENT: number2 = %d\n", number2);/*Line B*/
           exit(0);
    }
}
```

Answer: `CHILD: number1 = 10`

`PARENT: number2 = 10`

In the child process, the value of `number1` and `number2` are swapped, hence the value of `number1` is 10. And since all the variables of the parent process are copied to the child process and they are independent, in parent process the value of `number2` is still 10.

3. [40 points] Scheduling

3.1. What is *starvation* in CPU scheduling? In general, what type of scheduling algorithms can result in starvation? Please briefly justify your answer. (3 points)

Answer: *Starvation* occurs when a process is ready to run but is stuck waiting indefinitely for the CPU (1 point). A priority-based scheduling algorithm (such as SJF, SRTF) can suffer from starvation in that a low-priority process might wait for excessively long time if there are always high-priority processes on the ready queue (2 points).

3.2. In a symmetric multiprocessing or SMP system, what do we mean by *process affinity*? What is the difference between *hard affinity* and *soft affinity*? (4 points)

Answer: *Processor affinity* implies that a process has an affinity for the processor on which it is currently running (2 points). *Hard affinity* enforces a process to run a subset of processor(s), while *soft affinity* attempts to keep a process running on the same processors, but not guaranteeing it (2 points).

3.3. For a given set of processes, considering a Round-Robin scheduling and a FCFS scheduling algorithm, which scheduling algorithm yields smaller average waiting time? Please briefly justify your answer (suppose the context switching time can be ignored) (3 points)

Answer: It depends on the CPU burst time of each process and the time quantum of the round-robin scheduling algorithm. If the first arrived process has a long CPU burst time, then the round-robin scheduling yields smaller average waiting time, since it effectively prevents shorter jobs from getting stuck behind longer jobs. However, if the CPU burst time of each process is identical to the time quantum of round-robin scheduling, the average waiting time of these two scheduling should be the same. Sometimes FCFS scheduling yields smaller average waiting time, when the time quantum of round-robin scheduling is too large or processes with shorter CPU burst time arrive first. Thus it depends on different situation. (3 points)

3.4. Exponential averaging algorithm (10 points)

Consider the exponential average formula used to predict the length of the next CPU burst. The formula is as follows:

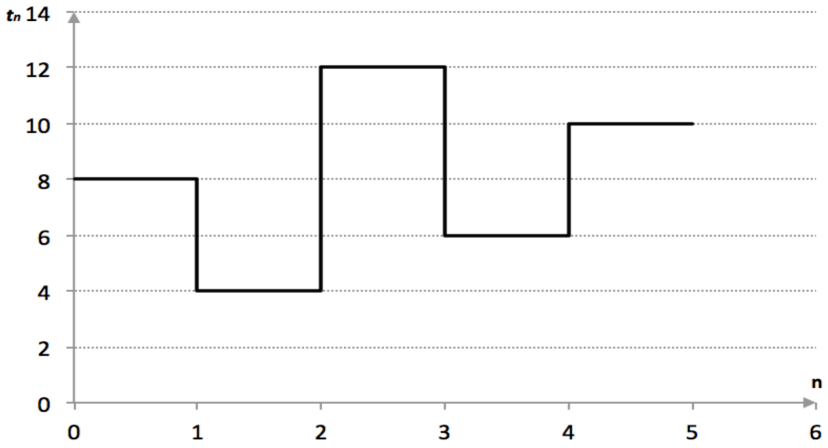
i. $t_n = acutal\ length\ of\ n^{\text{th}}\ CPU\ burst$

ii. $\tau_{n+1} = predicted\ value\ for\ the\ next\ CPU\ burst$

iii. $\alpha,\ 0 \le \alpha \le 1$

iv. Define: $\tau_{n+1} = \alpha \cdot t_n + (1-\alpha) \cdot \tau_n$

a) Suppose $\alpha = 0.5$, $\tau_0 = 8$, given the below graph describing t_n, please calculate the predicted CPU burst time at $n = 1, 2, 3, 4, 5$. (5 points)



Answer: According to the formula, we have:

$\tau_1 = 0.5*8 + 0.5*8 = 8$
$\tau_2 = 0.5*4 + 0.5*8 = 6$
$\tau_3 = 0.5*12 + 0.5*6 = 9$
$\tau_4 = 0.5*6 + 0.5*9 = 7.5$
$\tau_5 = 0.5*10 + 0.5*7.5 = 8.75$

b) What would happen if α value is too small? What's the disadvantage? You can simply explain the case $\alpha = 0$. (5 points)

Answer: When α is 0, the estimated value does not change according to the real time value. This will not reflect the changing values of real CPU burst length, making some scheduling algorithm less accurate.

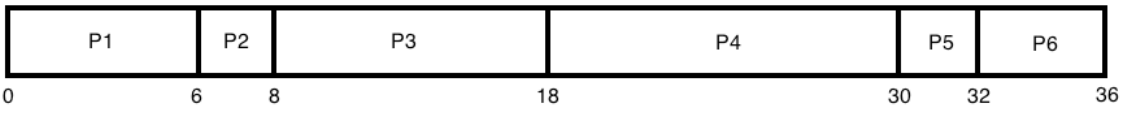
3.5. Given the following set of processes, with arrival time and length of the CPU-burst time given in milliseconds:

Process	Arrival Time (ms)	Burst Time (ms)
P1	0	6
P2	1	2
P3	5	10
P4	8	12
P5	11	2
P6	15	4

For each of the following scheduling algorithms, construct the Gantt chart depicting the sequence of process execution and calculate the *average waiting time* of each algorithm. (20 points)

a) FCFS (first-come, first-served) (5 points)

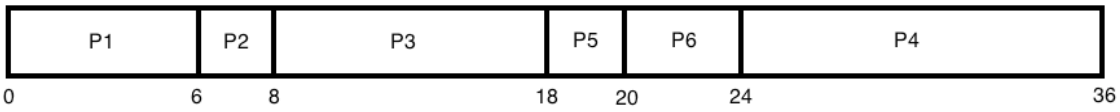
Answer:



Average waiting time = (0+5+3+10+19+17)/6 = 54/6 =9ms

b) SJF (shortest job first) non-preemptive (5 points)

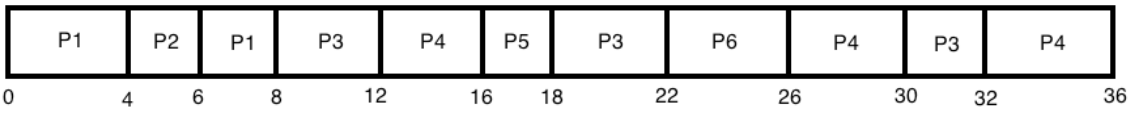
Answer:



Average waiting time = (0+5+3+16+7+5)/6 = 36/6 =6ms

c) RR (round robin) with quantum of 4 milliseconds (5 points)

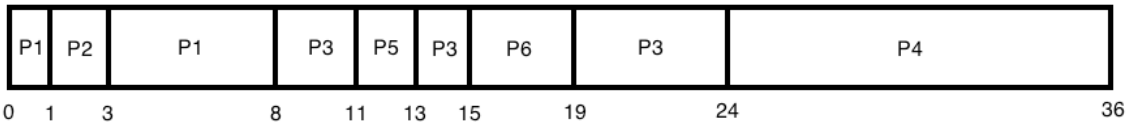
Answer:



Average waiting time = (2+3+17+16+5+7)/6 = 50/6 =8.33ms

d) SRTF (shortest remaining time first, i.e., preemptive SJF) (5 points)

Answer:



Average waiting time = (2+0+9+16+0+0)/6 = 27/6 =4.5ms

4. [30 points] Process Synchronization

4.1. What is the advantage of a *Spinlock*? (2 points)

Answer: When the critical section code is short, spinlock does not involve context switch, which saves the time (1 point), esp. in a multiprocessor (or SMP) system (1 point).

4.2. What do we refer as a *Critical Section*? What is the fundamental problem for a Critical Section that demands a solution? (4 points)

Answer: The *Critical Section (CS)* is a segment of code (can be short or long), in which a process may change shared variables such as updating a table, writing a file and so on (2 points). The fundamental problem is the existence of a race condition, in which the instructions can be interleaved arbitrarily during their execution due to the concurrent access and manipulation of share data from multiple processes, and the outcomes are not deterministic (2 points).

4.3. Under what scenario that a *reader-writer lock* is more efficient than a *semaphore*? Please briefly justify your answer (4 points).

Answer: a *reader-writer lock* is used to protect shared data that are accessed frequently by multiple threads, when they often access the data in a read-only manner (2 points). This is more efficient than semaphores, because it allows multiple threads to read the shared data concurrently, whereas semaphores always serialize access to the shared data (2 points).

4.4. Please explain the similarities and differences between `x.wait()` and `x.signal()` operations on conditional variables inside a monitor and `wait()` and `signal()` operations on semaphores without busy waiting. (5 points) (Hint: each has a queue associated)

Answer: They are similar in that both `wait()` and `x.wait()` can suspend a process (by putting it on a waiting queue associated either with the semaphore or with the conditional variable), and `signal()` and `x.signal()` can wake up a process from the queue. (2 points).

They are different in 1) `wait()` and `signal()` involves in updating the value of a semaphore, and `x.wait()` and `x.signal()` do not change values of any variable(s); (1 point) 2) suspending a process in `wait()` or removing a process from a queue in `signal()` needs to satisfy certain condition or depends on the value of the semaphore; (1 point) 3) `x.signal()` can have no effect if the queue associated with the conditional variable is empty, while `signal()` increment the semaphore value which will have an effect. (1 point)

4.5. The following codes implement a monitor and conditional variables using semaphores. (15 points)

```
// For a monitor, we have:

semaphore mutex; //(initially set to 1)
semaphore next; //(initially set to 0)
int next_count = 0;

//Each procedure F will be replaced by:

wait(mutex);
   ...
   body of F
   ...
if (next_count >0)
   signal(next);
else
   signal(mutex);
```

```
//For each conditional variable x, we have:

semaphore x_sem; //(initially set to 0)
int x_count = 0;


//The implementation of operation x.wait()

x_count++;
if (next_count > 0)
   signal(next);
else
   signal(mutex);
wait(x_sem);
x_count--;


//The implementation of operation x.signal()

if (x_count > 0) {
   next_count++;
   signal(x_sem);
   wait(next);
   next_count--;
}
```

a) Why is `mutex` needed? (1 point)

Answer: This provides mutual exclusion for accessing the monitor **`F`**.

b) What is the purpose of using the semaphore `next`? Why is `next` initialized to zero instead of one? (2 points)

Answer: next is used as the entry queue of the monitor **`F`** (1 point), whose initialization to zero ensures that a process can join the entry queue immediately when the monitor is not available. (1 point)

c) When `next_count > 0`, why do we `signal(next)`, instead of `signal(mutex)`? (2 points)

Answer: This allows one process on the entry queue to enter the monitor (1 point), still need to lock the monitor (by `mutex`) (1 point)

d) When do we `signal(mutex)`? (2 points)

Answer: This is done only when the entry queue is empty, or `next_count=0`.

e) What is the usage of the semaphore `x_sem`? (2 points)

Answer: `x_sem` defines the conditional variable x and its associated queue (1 point each)

f) What does `signa(x_sem)` do? (2 points)

Answer: It waits up one process if any on the conditional variable `x`.

g) What does `wait(next)` do following `signal(x_sem)`? (2 points)

Answer: The process gives up the monitor after signal operation, so to wait on the entry queue

h) Why do we need to do `x_count--` immediately following `wait(x_sem)`? (2 points)

Answer: This ensures that after the process waits up, `x_count` is decremented (or updated).