University of Illinois at Urbana-Champaign
Department of Computer Science

Midterm Conflict Examination 1

CS 323 Operating System Design
Spring, 2003

5-6pm Monday, March 10th
DCL 1320

Print your name and ID number neatly in the space provided below; print your name at the upper right corner of every page.

Name:
Net ID:

This is a **closed book, closed notes, No calculator** exam.

Write your answers CLEARLY. The answer is wrong if the instructor and the TAs cannot read it.

Do all parts of all four problems in this booklet. This booklet should include this title page, 7 additional pages and 2 empty pages. Do your work inside this booklet, using the empty pages and backs of pages if needed. Problems are of various difficulty degree, hence if you don't know the answer immediately, progress to the next problem and come back to the unsolved problem later.

Problem	Score	Grader
1		
2		
3		
4		
Total		

1. Problem – Process Scheduling (25 Points)

Suppose we have 4 processes arrive for execution. The process burst time (duration), as well as the arrival time and priority of each process is indicated in the following table. Time is in milliseconds (ms).

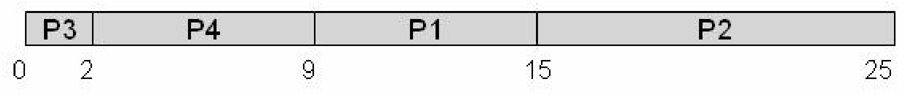
Process	Arrival Time	Burst Time (duration)	Priority
P_1	6	6	4
P_2	2	10	3
P_3	0	2	2
P_4	1	7	4

Table 1: The processes with their arrival time, process behavior and priority

1. (15 Points) Fill out the three Gantt charts below illustrating the scheduled execution of these processes using
   a) non-preemptive shortest job first scheduling algorithm
   b) Round Robin (quantum = 4ms)
   c) **Preemptive** priority (a smaller priority number implies a higher priority).

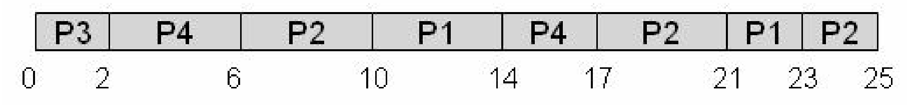
Note: CPU cannot stay idle if there are still unfinished processes.

Gantt Chart for non-preemptive shortest job first Algorithm



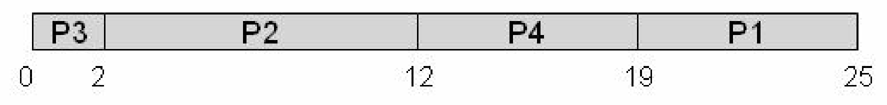
P_3 runs from time 0 to 2.
P_4 runs from time 2 to 9.
P_1 runs from time 9 to 15.
P_2 runs from time 15 to 25.

Gantt Chart for Round Robin Algorithm with Quantum 4



P_3 runs from time 0 to 2.
P_4 runs from time 2 to 6.
P_2 runs from time 6 to 10.
P_1 runs from time 10 to 14.
P_4 runs from time 14 to 17.
P_2 runs from time 17 to 21.
P_1 runs from time 21 to 23.
P_2 runs from time 23 to 25.

Gantt Chart for <u>Preemptive</u> Priority Algorithm



When scheduling processes of the same priority, using FIFO.

2. (10 Points) Calculate the average waiting for these processes with each of the scheduling algorithms in part (1).

Algorithm	**Average Waiting time**
Non-preemptive shortest job first	0+1+3+13 = 17/4 = 4.25
Round Robin	0+9+13+11 = 33/4 = 8.25
Preemptive Priority	0+0+11+13=24/4 = 6

2. Problem-Synchronization (20 points)

1. (10 points) Show how counting semaphores (i.e., semaphores that can hold an arbitrary value) can be implemented using only binary semaphores and ordinary machine instructions . You may use a pseudo-code to show the implementation. **Note: A busy-waiting implementation is acceptable. There is no need to implement a block/wakeup counting semaphore.**

Two possible solutions are given below. The left column uses block/wakeup and the right column uses busy waiting.

```
Class semaphore{
  Mutex mutex;  //a binary mutex
  Mutex wait; // not needed for the right solution
  int value = N; // the counter
```

```
Semaphore::P() {
{
   mutex.down();

   while ( value <= 0 ) {
      mutex.up();
      wait.down();
      mutex.down();
   }

   value -= 1;
   mutex.up();
}

Semaphore::V()
{
   mutex.down();
   value += 1;

   if ( value <= 0 ) {
      wait.up();
   }

   mutex.up();
}
```

```
Semaphore::P()
{
   mutex.down();

   while ( value <= 0 ) {
      mutex.up();
      while ( value <= 0 ) ;
      mutex.down();
   }

   value -= 1;
   mutex.up();
}

Semaphore::V()
{
   mutex.down();
   value += 1;
   mutex.up();
}
```

2. (10points) Consider the following producer and consumer code. Is the code correct? If yes, explain what is happening in each line of the code. If no, specify: what is wrong, which line is wrong and why it is wrong. The number at the very beginning of each line is the line number.

```
1. typedef int semaphore;
2. semaphore mutex=1;
3. semaphore empty = N;
4. semaphore full = 0;
5.
6. void producer(void)
7.  { int item;
8.
9.   while (TRUE) {
10.      item = produce_item ();
11.      down(&mutex);
12.      insert_item(item);
13.      down(&empty);
14.      up(&full);
15.   }
16. }
17. void consumer(void)
18.  { int item;
19.
20.   while (TRUE) {
21.      down(&mutex);
22.      item= remove_item();
23.      down(&full);
24.      up(&empty);
25.      consume_item(item):
26.   }
27.  }
```

There are four problems in the code. These are identified, together with appropriate fixes, below.

Line 10/11: Before the *down* on *mutex*, the producer should also *down* the *empty* semaphore. This will block the producer if the number of empty slots is 0 (that is, there is no space left in the shared buffer). Note that this must be done before the *down* on *mutex* to avoid the potential for deadlock.

Line 13. Doing a *down* on the *empty* semaphore at this point makes no sense. The producer should instead release the mutual exclusion it holds by doing an *up* on *mutex*.

Line 20/21. This is an analogous problem to that between lines 10 and 11. The consumer must, before doing *down* on the *mutex* semaphore, also *down* the *full* semaphore. This will block it if the number of full slots is 0 (all slots are empty, in other words).

Line 23. The *down* on the *full* semaphore is not correct. Instead, an *up* on the *mutex* semaphore should happen here, indicating the consumer is leaving the critical section.

3 Problem-Deadlock (30 points)

Consider two resource types A and B. Furthermore, consider the snapshot of a system in the following table. Allocation matrix (column) lists the number of resources allocated for each process; while the Max. Request matrix (column) lists the maximum number resources each process request.

	Allocation		**Max. Request**		**Available**		**Need**	
	A	B	A	B	A	B	A	B
P_0	2	0	2	5	3	Y	0	5
P_1	5	2	10	2			5	0
P_2	0	4	5	4			5	0
P_3	1	1	4	1			3	0
P_4	0	0	5	9			5	9

1. (5 points) Determine the Need Matrix (column) for resource types A and B and fill them in then above table.

.

2. (10points) Suppose Y=8, is it a safe state? If it is safe, provide a safe sequence for executing all processes (In order to get a full credit, show your work how you find the safe sequence). If it is not safe, explain it.

The state is a safe one. There are multiple correct sequences. One possibility is shown below. The columns show how many of A and B is available after that process successfully executes. Essentially the constraints are that either P_0 or P_3 must run first (constraint by availability of A). After either of these processes successfully execute all other processes can run in any order with the exception of P_4 which must wait until one more instance of B is released.

	A	B
	3	8
P_0	5	8
P_1	10	10
P_2	10	14
P_3	11	15
P_4	11	15

3.

3. (15 points) What is the smallest value of Y for which this is a safe state? Provide a safe sequence for executing all processes for the situation with the smallest value of Y for which this is a safe state. For your convenience, the table in the previous page is **copied** below:

	Allocation		**Max. Request**		**Available**		**Need**	
	A	B	A	B	A	B	A	B
P_0	2	0	2	5			0	5
P_1	5	2	10	2			5	0
P_2	0	4	5	4	3	Y	5	0
P_3	1	1	4	1			3	0
P_4	0	0	5	9			5	9

The need matrix is as above. We compute minimum Y by first assuming Y=0 and finding a process that we can execute. If one is not available, we pick the process with the minimum B requirement. We repeat the algorithm until all processes have executed. (Note: multiple correct sequences may exist).

	A	B	B Need
	3	Y	
P_3	4	Y+1	0
P_0	6	Y+1	5
P_1	11	Y+3	0
P_2	11	Y+7	0
P_4	11	Y+7	9

We can now compute Y in the following manner. We look at the first row and see that to execute P_3 we need 0 B, thus Y=0 (from the row above). Next, to execute P_0, Y+1=5, or Y=4. We proceed like this and find that Y must be at least 4 to allow this sequence to execute.

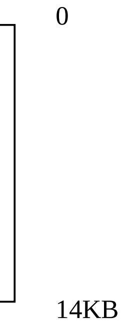
The above is a safe sequence.

4 Problem – Memory Management (25 Points)

In most systems, a process can allocate and de-allocate memory from its heap using *malloc()* and *free()*. A heap is just a chunk of memory. Suppose the heap is managed with linked list. Each node in the list is either allocated or free. **And the list kept sorted by address in increasing order.** When *malloc()* is called, the list is searched for a free segment that is big enough (depending on the allocation algorithm), that segment is **divided** into an allocated segment (at the beginning) and a free segment. When *free()* is called, the corresponding segment should **merge** with its neighboring segments, if they are also free.

Now suppose a process has a heap of 14KB, which is initially empty. During its execution, the process issues the following memory allocate/de-allocate calls (pi is just a pointer):

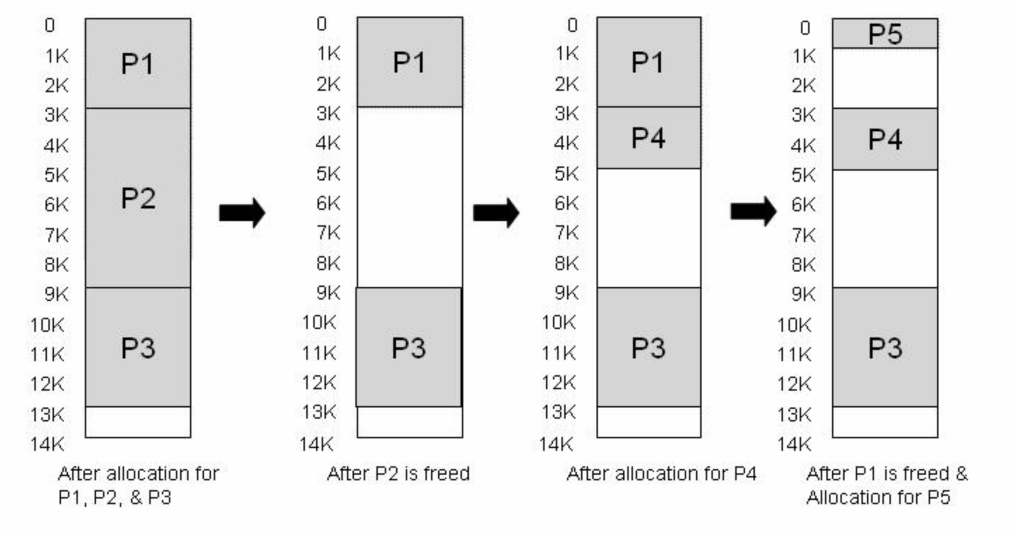
p1 = malloc(3KB)
p2 = malloc(6KB)
p3 = malloc(4KB)
free(p2)
p4 = malloc(2KB)
free(p1)
p5 = malloc(1KB)



Show what the heap is like after the above calls, using (1) best-fit, (2)first fit and (3)worst-fit algorithms respectively. For simplicity, assume the memory begins at address 0, and ignore the memory used by the linked list itself.

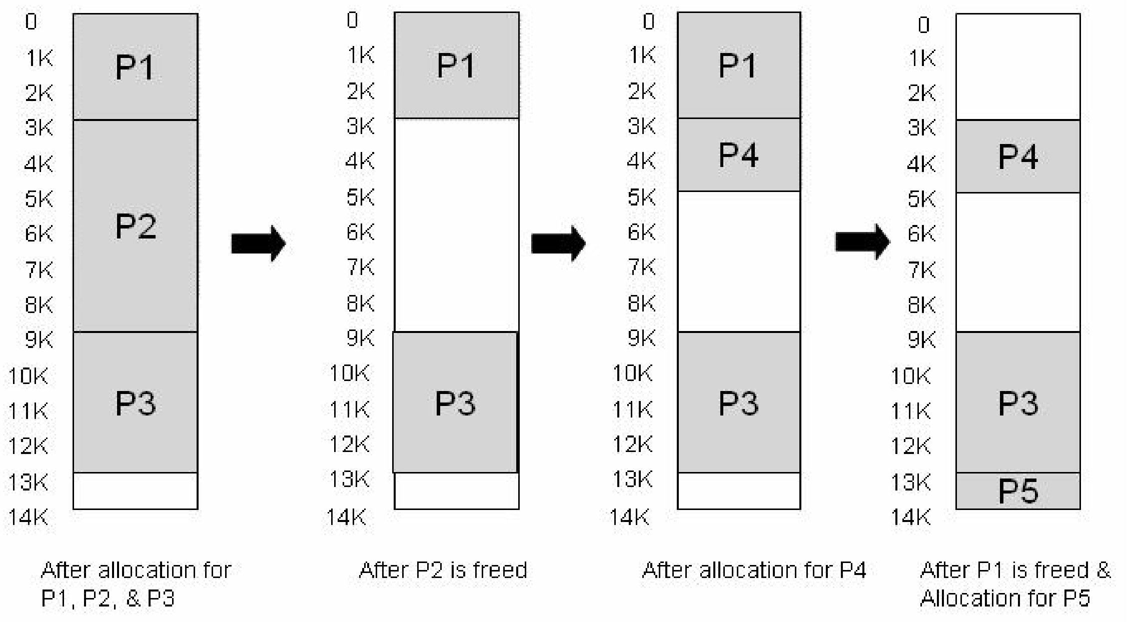
You should show your answer using a box similar to the above one and mark those segments that are allocated (for example, shaded area is allocated). To get full credits, show your steps.

1. **First-fit (5 points)**

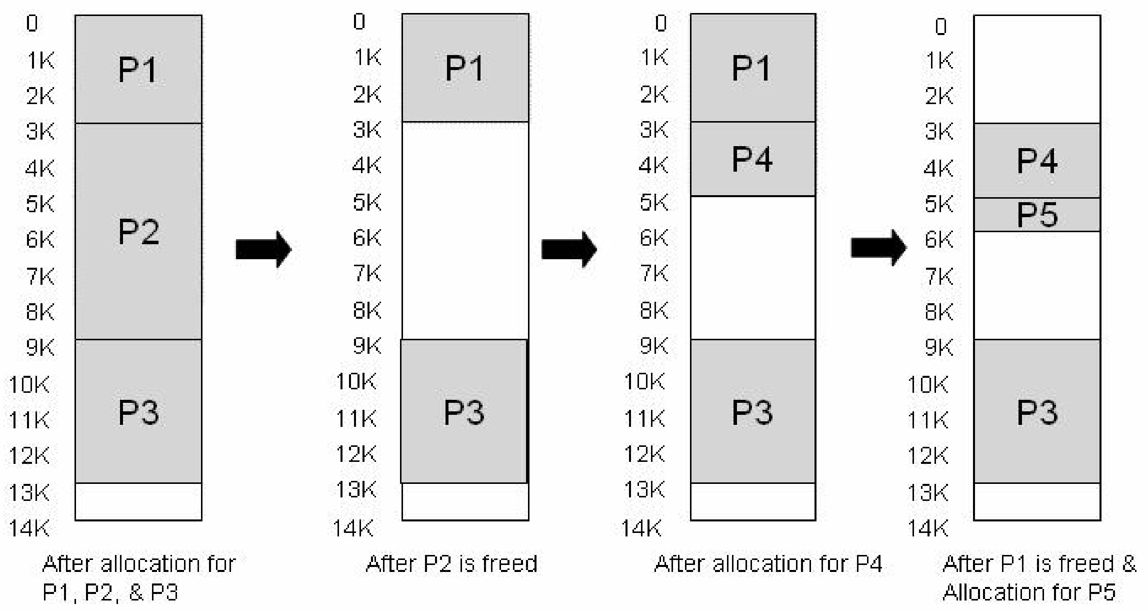


2.

2. Best-fit (10 points)



3. Worst-fit (10 points)


0
1K
2K
3K
4K
5K
6K
7K
8K
9K
10K
11K
12K
13K
14K
P1
P2
P3
P4
P5
After allocation for P1, P2, & P3
After P2 is freed
After allocation for P4
After P1 is freed & Allocation for P5