ECE 350 S23 Midterm Exam Solutions

(1.1)

Improvements Answers will vary, but my immediate suggestions are:

(1) terminate the loop if we do find the process ID (i.e. after `found = true;` insert a `break;` statement) because otherwise we are just wasting time; and

(2) remember the last generated value – instead of starting at 2 all the time, if the last process ID generated was 981 then just start at 981 and count up. It's much less likely that the next one is taken than if we start at 2, because they'll most likely all be clustered at the low numbers so we can avoid constantly checking the same process IDs.

Predictabiity You can argue for both yes and do, as long as your answer is supported by your argument. If you argue no, it's likely because knowing the value of the process ID is not very useful – knowing this arbitrary number is not super helpful, particularly because you are unlikely to be able to get it to be exactly a number that you need it to be.

However, if you argue yes, then it's because this is a covert way for different processes to communicate even if they "should not" – in principle the next process ID can be manipulated by programs by starting or killing processes and that is a problem if you need total security.

(1.2) I'd say reject this solution. Although this does prevent-low-priority threads from starving, the reason for rejection is because this solution makes it possible for medium-priority threads to starve. High priority threads will proceed with an 80% chance when a thread is unblocked; low priority threads with a 20% chance. But if there are enough high- and low-priority threads that are being unblocked by this scheme, a medium-priority thread will never be selected. So this solution will not be acceptable.

(1.3) The primary motivation here is security: any system service that runs at a higher level of privilege is potentially an attack vector and reducing the risk is reasonable. However, it does mean that these processes run more slowly – thanks to system calls and other overhead and all the limitations that go with being user-level processes.

(1.4) A simple way to implement sleep is to set up a timer until the designated sleep time, block the thread, and when the timer expires, unblock the thread.

(2)

You could come up with a different scheme than what I have, but the important thing is it has to work in an atomic manner without using locks.

1. Data Structure A linked list with just a head pointer is sufficient here.

(If you have a tail pointer it gets more difficult to update the list and tail pointer)

2. Add PCB

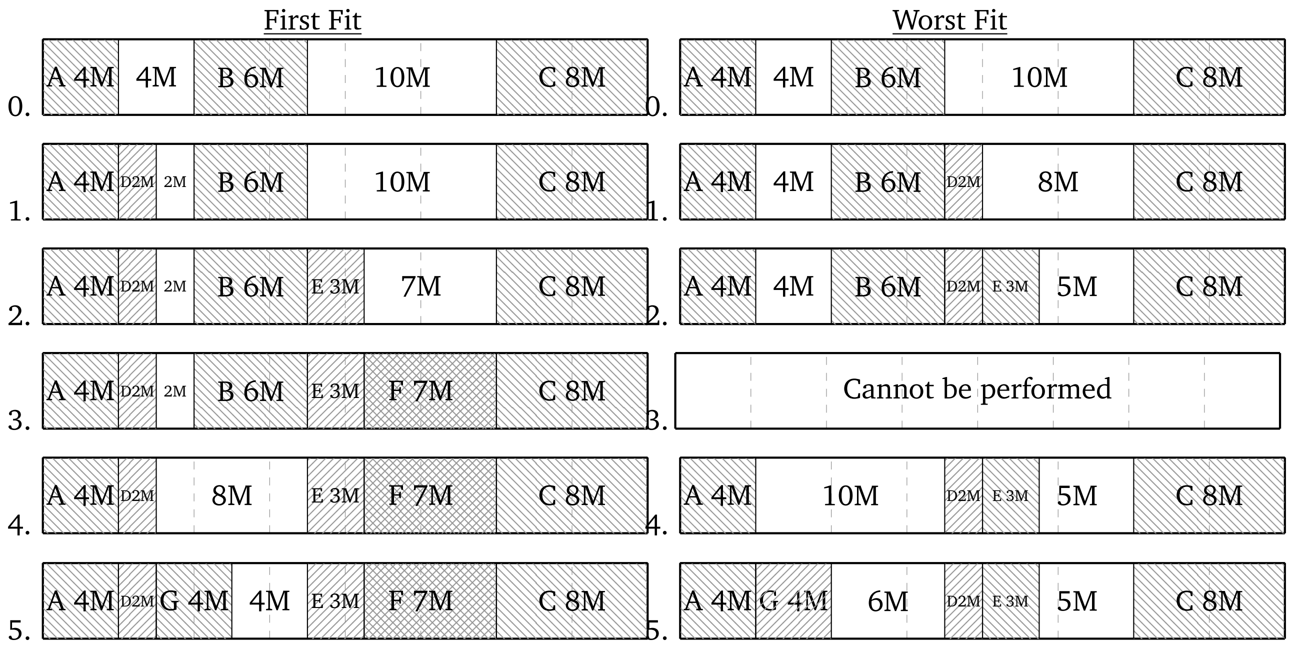
```
Set current pointer to the start of the list
Iterate to the end of the list
while true
  Compare and swap the current->next pointer from NULL to the new PCB
  if successful break out of loop
  else advance the current pointer to the new last node of the list
end while
```

3. Remove PCB

```
while true
 compare and swap list head from current value to head->next
 if successful break out of the loop
 else update head to the new value
end while
```

4. Cancellation One option is iterate over the list until the next is the target and then do a compare-and-exchange so `current->next = current->next-next` which would effectively remove it from the list. But you could also just mark the process as cancelled and when it gets to the head of the list, after it has been dequeued you don't run it but run another process instead.

(3.1)


First Fit
Worst Fit
0. A 4M | 4M | B 6M | 10M | C 8M
0. A 4M | 4M | B 6M | 10M | C 8M
1. A 4M | D2M | 2M | B 6M | 10M | C 8M
1. A 4M | 4M | B 6M | D2M | 8M | C 8M
2. A 4M | D2M | 2M | B 6M | E 3M | 7M | C 8M
2. A 4M | 4M | B 6M | D2M | E 3M | 5M | C 8M
3. A 4M | D2M | 2M | B 6M | E 3M | F 7M | C 8M
3. Cannot be performed
4. A 4M | D2M | 8M | E 3M | F 7M | C 8M
4. A 4M | 10M | D2M | E 3M | 5M | C 8M
5. A 4M | D2M | G 4M | 4M | E 3M | F 7M | C 8M
5. A 4M | G 4M | 6M | D2M | E 3M | 5M | C 8M

(3.2)

```
1, 2, 3, 4, 1, 2, 5, 1, 2, 3, 4, 5
```

Page 0	P1	P2	Fault?	Total Faults
1	–	–	Y	1
1	2	–	Y	2
1	2	3	Y	3
4	2	3	Y	4
4	1	3	Y	5
4	1	2	Y	6
5	1	2	Y	7
5	1	2	N	7
5	1	2	N	7
5	3	2	Y	8
5	3	4	Y	9
5	3	4	N	9

Page 0	P1	P2	P3	Fault?	Total Faults
1	–	–	–	Y	1
1	2	–	–	Y	2
1	2	3	–	Y	3
1	2	3	4	Y	4
1	2	3	4	N	4
1	2	3	4	N	4
5	2	3	4	Y	5
5	1	3	4	Y	6
5	1	2	4	Y	7
5	1	2	3	Y	8
4	1	2	3	Y	9
4	5	2	3	Y	10

(3.3) Separate instruction and data caches are beneficial because many threads operate on the basis of doing the operations on sequential elements of data. In that sense, it makes sense for the instructions to remain in cache since they'll be accessed again soon (temporal locality) and for data to move in and out as needed (spatial locality).

(3.4)

Part 1. The intention was that segment numbers are the first 8 bits, but that's hard to identify from the way the question is written – you'd have to know that by looking at the addresses in the table and noticing that the first two digits are probably it. Sorry about that.

Still, there's at least one bit for the segment since the table shows segments 0 and 1. So as long as you said that the number of bits for the segment is at least 1, the offset is 12 bits, and the total number of bits across all three segments adds up to 32, you got the mark here.

Part 2.

Virtual Address	**Physical Address**	**Evaluation**
0x010029AA	0x79AA	OK
0x00001F0C	N/A	Page Fault
0x010049E0	N/A	Segmentation Fault
0x000030D7	0x40D7	OK