Solver: Prem Adithya Suresh
Email: prem0008@e.ntu.edu.sg

1(a)(i) False. IO utilisation is higher in multiprogramming systems due to the presence of I/O device scheduling.

1(a)(ii) False. A switch from user mode to monitor mode will be required.

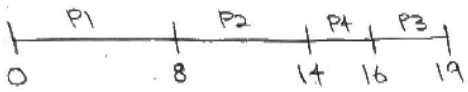
1(a)(iii) False. Non pre-emptive CPU scheduling happens when a process in the running state terminates or enters waiting state due to event or I/O wait. (Slide 3.5)

1(a)(iv) False. It selects from among the processes that are ready to execute (waiting for CPU only) and allocates the CPU to one of them. (Slide 2.14)

1(a)(v) True. Regardless of whether it is direct or indirect, a single sender can always communicate with many receivers. (Slide 2.24)

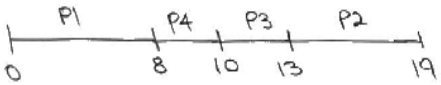
1(b)

FCFS



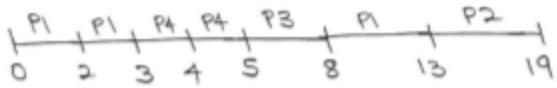
Average waiting time
$= [0 + (8-2) + (14-3) + (16-4)] \div 4$
$= 29 \div 4$
$= 7.25s$ //

SJF



Average waiting time
$= [0 + (8-3) + (10-4) + (13-2)] \div 4$
$= 22 \div 4$
$= 5.5s$ //

SRTF



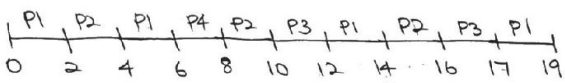
Average waiting time

$= [(13-0-8) + (19-2-6) + (8-4-3) + (5-3-2)] \div 4$

$= 17 \div 4$

$= 4.25s$

RR



Average waiting time

$= [(19-0-8) + (16-2-6) + (17-4-3) + (8-3-2)] \div 4$

$= 32 \div 4$

$= 8s$

1(c)
The fork–join model is a way of setting up and executing parallel programs, such that execution branches off in parallel at designated points in the program, and “join” at a subsequent point and resume sequential execution.

```
int main() {
        Thread t1, t2;
        int count = 0
        fork(&t1, &count, 1);
        fork(&t2, &count, 1);
        join(&t1);
        join(&t1);
        printf(count);
}
```

In this above code, the join statements are in the correct place and hence the final value of count will be 2. If the join statements were not where they are, the value of count printed will be 0.

2(a)(i)
True. Race condition can be prevented by disabling context switches, also known as atomic instructions. (Slide 4.6 – 4.7)

2(a)(ii)
False. Mutual exclusion is to ensure that no more than one process is in the critical section at any given time.

2(a)(iii)
True. Processes can always request for resources anytime, but deadlock avoidance algorithms will only grant a request if granting it will not put the system in an unsafe state.

2(a)(iv)
False. Only bounded waiting is violated. (Slide 4.24)

2(a)(v)
False. Blocking semaphores have context switch overhead and hence when critical sections are short, busy waiting semaphores are better. (Slide 4.28)

2(a)(vi)
True. Incorrect semaphore use can lead to starvation based on the queue disciplines used. (Slide 4.37)

2(b)

```
Entry Section: while(lock);
               lock = true;

        Critical section

Exit Section: lock = false;

        Remainder section
```

1. lock is initially false
2. P1 executes while(lock) and exits loop
3. Context switch occurs from P1 to P2
4. P2 executes while(lock) and exits loop
5. Context switch occurs from P2 to P1
6. P1 sets lock to true and then enters critical section
7. Context switch occurs from P1 to P2
8. P2 sets lock to true and then enters critical section

In this case both P1 and P2 are in the critical section, hence violating mutual exclusion. Therefore, a simple Boolean variable cannot ensure mutual exclusion.

2(c)

	Allocation	Need	Work	Finished
	A B C	A B C	A B C	
P1	1 2 1	2 1 5	~~2 2 1~~	
P2	1 2 0	1 2 0	~~2 0 0~~	
P3	1 0 2	3 1 1	~~3 3 2~~	P4
P4	~~+ + +~~	~~+ 2 +~~	~~1 3 4~~	P3
	1 3 2	1 0 0	5 5 4	P2

Request should not be granted as a sequence does not exist. P1 can never complete as 5 instances of C is never available.

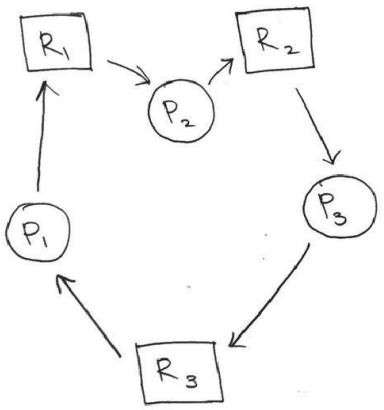
2(d)
Progress will not be violated as at any point in time turn can only be 0 or 1 allowing one of the processes to enter the critical section.
Bounded waiting will not be violated as the respective flags are set to false at the exit section.

1. P0 executes turn = 1.
2. Context switch occurs from P0 to P1.
3. P1 executes turn = 0;
4. P1 executes flag[1] = true.
5. P1 enters the critical section since flag[0] is initialised to false.
6. Context switch occurs from P1 to P0.
7. P0 executes flag[0] = true.
8. P0 enters the critical section as turn = 0.

In the scenario described above, both processes can enter the critical section hence violating mutual exclusion.

Note: This question is similar to slide 4.18 but the order of the variables is flipped.

2(e)



In the above diagram a cycle is formed leading to a deadlock since every resource is of a single instance. Keeping in mind that we are unable to break the deadlock conditions of mutual exclusion and no pre-emption in this system, a possible deadlock prevention approach here is to limit the number of processes that can use the resources to a maximum of two. This will break the deadlock condition of circular wait. However, the potential downside to this approach is the possibility of starvation as a third process may never get to acquire these resources.

3(a)(i)
True. The addresses are identical as binding is done at load time. (Slide CR.16)

3(a)(ii)
False. If allocated memory is larger than requested memory, internal fragmentation occurs.

3(a)(iii)
True. Number of pages in a page table is determined by the size of the logical address space.

3(a)(iv)
False. Whether the array is stored as row-major order or column-major order affects which elements are contiguous. Accessing a 2-D array by columns when elements are arranged in row-major order will lead to non-contiguous data access and weaker spatial locality.

3(b)(i) 2^y

3(b)(ii) 2^{x+y}

3(b)(iii) 2^x

3(b)(iv) Size of first level page table = $2^x * 2^2 = 2^{(x+2)}$ bytes

Number of entries of outer level page table = $2^{(x+2)} / 2^y = 2^{(x+2-y)}$

3(c)(i) 649

3(c)(ii) Addressing error

3(d)(i)

F1	0,0	0,0	0,1	0,1	0,0	0,0	1,0	1,0
F2		1,0	1,0	1,0	4,0	4,0	4,0	0,0
F3				2,0	2,0	2,1	2,0	2,0

Reference string: 0, 1, 0, 2, 4, 2, 1, 0

3(d)(ii)
When all the bits are set, the hand cycles through the whole queue, giving each page a second chance. In this case it degenerates to FIFO replacement.

3(e)
If the physical memory is large, more pages can be kept in the memory therefore reducing the number of page faults.

Increasing the degree of multiprogramming improve CPU utilisation to an extent. However, if the degree of multiprogramming is too high, it can result in thrashing hence negatively affecting the performance of the system.

The access time of secondary storage affects the performance greatly as reading in a new page is the longest component of the page fault time.

4(a)(i)
"/usr/peter/documents/os/lecture_slides.ppt"

4(a)(ii)
rw- r-- ---

4(a)(iii)
When the block size is very small, files may consist of many blocks leading to a low data rate. However, it will ensure a better disk space utilisation by minimising internal fragmentation at the last block.

4(a)(iv)
In this configuration there will be two pairs of mirrored hard disks allowing for an overall capacity of 400GB.

4(b)(i)
Contiguous allocation. This method supports random access and does not require additional overhead for storage (as compared to indexed allocation for index blocks/address mapping). Since data is updated infrequently, the higher cost of updating is not as crucial.

4(b)(ii)
Indexed allocation. This method supports random access, dynamic storage allocation and indexing allows for easy updating.

4(c)
The linked list method only requires keeping a pointer to the first free block on the disk therefore not taking up much space, however it is inefficient as the free blocks must be accessed sequentially. The bit-map method is more efficient as it allows for quick random access and deleting a block is as easy as flipping a bit. However, bit maps occupy more space since keeping track of each block takes up 1 bit of space. Moreover, to remain efficient the whole bit map needs to be kept in memory.

4(d)
SSTF: 90, 100, 120, 160, 75, 60, 40, 20
LOOK: 90, 100, 120, 160, 75, 60, 40, 20
The requests will be serviced in the same order.

