Solver: Ricardo Jack Liwongan

Email Address: rica0003@e.ntu.edu.sg

1. (a)
   (i) True. All I/O instructions are privileged instructions. (Slide 1.32)
   (ii) True. Preemptive means that the CPU can be taken away from running process at any time by OS. If the system use a nonpreemptive scheduling, it means that once the CPU has been allocated to a process, the process keeps the CPU until it releases the CPU either by terminating or requesting I/O/event wait, which means that any process can only transition from running to terminated or running to waiting. (Slide 3.4-3.5)
   (iii) True, shared memory is simpler because the system calls are required only to establish shared-memory regions. It is also faster than Message Passing because system calls are required to implement the message passing.
   (iv) False. Multi-programmed systems are designed to have the capability to run more than one program at the same time in one sinlge CPU. Several processes are kept in main memory at the same time, and the CPU is multiplexed among them with the purpose of improving CPU utilization.
   (v) False. Giving high priorities to jobs with intensive CPU computation will not improve the CPU utilization, it will just control the order of execution of the processes where processes with intensive CPU computation will execute first.

   (b)
   Turnaround time = time of completion - time of submission
   Waiting time= Turnaround time – CPU burst time

   (i) Non-preemptive Shortest Job First Scheduling

P1 CPU 2 units	P4 CPU 2 units	P3 CPU 1 units	P2 CPU 3 units	P5 CPU 3 units

0 2 4 5 8 11

P1: Turnaround time = 2 – 0 = 2 ms , waitting time = 2 - 2 = 0 ms
P2: Turnaround time = 8 – 2 = 6 ms , waitting time = 6 - 3 = 3 ms
P3: Turnaround time = 5 – 4 = 1 ms , waitting time = 1 - 1 = 0 ms
P4: Turnaround time = 4 – 0 = 4 ms , waitting time = 4 - 2 = 2 ms
P5: Turnaround time = 11 – 3 = 8 ms , waitting time = 8 - 3 = 5 ms

Average waiting time = $\frac{(0+3+0+2+5)ms}{5} = \frac{10\ ms}{5} = 2\ ms$

   (ii) Preemptive Priority based Scheduling

P1 CPU 2 units	P2 CPU 1 units	P5 CPU 3 units	P2 CPU 2 units	P4 CPU 2 units	P3 CPU 1 units

0 2 3 6 8 10 11

P1: Turnaround time = 2 – 0 = 2 ms , waitting time = 2 - 2 = 0 ms
P2: Turnaround time = 8 – 2 = 6 ms , waitting time = 6 - 3 = 3 ms
P3: Turnaround time = 11 – 4 = 7 ms , waitting time = 7 - 1 = 6 ms
P4: Turnaround time = 10 – 0 = 10 ms , waitting time = 10 - 2 = 8 ms
P5: Turnaround time = 6 – 3 = 3 ms , waitting time = 3 - 3 = 0 ms

Average waiting time = $\frac{(0+3+6+8+0)ms}{5} = \frac{17\ ms}{5} = 3.4\ ms$

(c)
Operating system will execute the load instructions for the base and limit registers. OS will then check whether the address being accessed by the program is larger than the base registers and smaller than the sum of (base and limit) registers. In this case, buffer overflow, where the program will overwrite parts of memory that it is not supposed to, will take place. As the program access memory address outside its allocated legal range, then a trap will be issued to OS then OS will handle the trap for this addressing error.

(d)
(i) Advantage: It allows some high priority processes to get more unit of CPU time and complete faster, which will allow some kind of priority scheduling with good response time for each process.
Disadvantage: Increase the memory complexity for the ready queue.
(ii) One of the alternatives to achieve the same effect is to modify the mechanism for the ready queue. Instead of using queue as the data structure for the ready queue, we can use array data structure. Then, when some processes are preemted(after its quantum time has ellapsed), the pointers will be moved to the position between the head and end of the ready queue, instead of being moved to the end of the queue. For other processes, the pointers is moved to the end of te ready queue.

2. (a)
(i) True. Solution to the critical section problem should not depend on the order of the process executions that enter the critical sections. It should be able to handle processes with different speed of executions, no matter which process is faster, it should still be able to work. Therefore, it should be independent of te speed of process executions.
(ii) False. Atomic instructions only do a limited set of operations, and often they are not enough to synthesize/execute more complicated operations efficiently. For example, manipulating data using atomic instruction generally requires global memory accesses, which take hundreds of clock cycles.
(iii) False. Deadlock may occur in any system that can run multiple processes/ tasks concurently and has multiple resources.
(iv) True. Preempting a philosopher will violate the no-preemption deadlock condition. As no-preemption is a necessary condition for deadlock, deadlock will not occur.
(v) True. The necessary condition for deadlock is that there at least one cycle in the resource allocation graph.(Slide 5.10)

(b)
Total number of instance of the resource = 10

Process	Max	Allocation
P	8	0
Q	4	0
R	9	0

- Request r1, $Request_R = 2$, $Need_R = 9$, Available = 10
  $Request_R \le Need_R \Rightarrow$ *true.* $Request_R \le Available \Rightarrow true$
  Pretend to allocate the requested resource:

Available = 10 - 2 = 8, $\text{Allocation}_R = 0 + 2 = 2$, $\text{Need}_R = 9 - 2 = 7$
Run the safety algorithm:

Process	Allocation	Need
P	0	8
Q	0	4
R	2	7

Available = 8

Executing safety algorithm shows that there exist a safe sequence of the processes. Sequence<P, Q, R> satisfies safety requirement. The request r1 is granted by the system and the resource is allocated to process R.

- Request r2, $\text{Request}_P = 4$, $\text{Need}_P = 8$, Available = 8
  $\text{Request}_P \le \text{Need}_P \Rightarrow$ *true.* $\text{Request}_P \le$ *Available* $\Rightarrow$ *true*
  Pretend to allocate the requested resource:
  Available = 8 - 4 = 4, $\text{Allocation}_P = 0 + 4 = 4$, $\text{Need}_P = 8 - 4 = 4$
  Run the safety algorithm:

Process	Allocation	Need
P	4	4
Q	0	4
R	2	7

Available = 4

Executing safety algorithm shows that there exist a safe sequence of the processes. Sequence<P, Q, R> satisfies safety requirement. The request r2 is granted by the system and the resource is allocated to process P.

- Request r3, $\text{Request}_Q = 2$, $\text{Need}_Q = 4$, Available = 4
  $\text{Request}_Q \le \text{Need}_Q \Rightarrow$ *true.* $\text{Request}_Q \le$ *Available* $\Rightarrow$ *true*
  Pretend to allocate the requested resource:
  Available = 4 - 2 = 2, $\text{Allocation}_Q = 0 + 2 = 2$, $\text{Need}_Q = 4 - 2 = 2$
  Run the safety algorithm:

Process	Allocation	Need
P	4	4
Q	2	2
R	2	7

Available = 2

Executing safety algorithm shows that there exist a safe sequence of the processes. Sequence<Q, P, R> satisfies safety requirement. The request r3 is granted by the system and the resource is allocated to process Q.

- Request r4, $\text{Request}_P = 2$, $\text{Need}_P = 4$, Available = 2
  $\text{Request}_P \le \text{Need}_P \Rightarrow$ *true.* $\text{Request}_P \le$ *Available* $\Rightarrow$ *true*
  Pretend to allocate the requested resource:
  Available = 2 - 2 = 0, $\text{Allocation}_P = 4 + 2 = 6$, $\text{Need}_P = 4 - 2 = 2$
  Run the safety algorithm:

Process	Allocation	Need
P	6	2
Q	2	2
R	2	7

Available = 0

Executing safety algorithm shows that there is no safe sequence of the processes. Therefore, the request r4 is not granted by the system and the old resource-allocation state is restored: $Need_P = 4$, Available = 2.

- Request r5, $Request_R = 1$, $Need_R = 7$, Available = 2
  $Request_R \le Need_R \Rightarrow$ *true*. $Request_R \le$ *Available* $\Rightarrow$ *true*
  Pretend to allocate the requested resource:
  Available = 2 - 1 = 1, $Allocation_R = 2 + 1 = 3$, $Need_R = 7 - 1 = 6$
  Run the safety algorithm:

Process	Allocation	Need
P	4	4
Q	2	2
R	3	6

Available = 1

Executing safety algorithm shows that there is no safe sequence of the processes. Therefore, the request r5 is not granted by the system and the old resource-allocation state is restored: $Need_R = 7$, Available = 2.

- Request r6, $Request_Q = 2$, $Need_Q = 2$, Available = 2
  $Request_Q \le Need_Q \Rightarrow$ *true*. $Request_Q \le$ *Available* $\Rightarrow$ *true*
  Pretend to allocate the requested resource:
  Available = 2 - 2 = 0, $Allocation_Q = 2 + 2 = 4$, $Need_P = 2 - 2 = 0$
  Run the safety algorithm:

Process	Allocation	Need
P	4	4
Q	4	0
R	2	7

Available = 0

Executing safety algorithm shows that there exist a safe sequence of the processes. Sequence<Q, P, R> satisfies safety requirement. The request r3 is granted by the system and the resource is allocated to process Q.

(c)
Shared Resources: Buffer B1 and B2
Semaphore: b1empty = 1, b1full = 0, b2empty = 1, b2full=0

Process R	Process M	Process P

while(1){
wait(b1empty);
get one data item from the device;
put the data to **B1**;
signal(b1full);
}
	while(1){
	wait(b1empty);
	get one data item from **B1**;
	signal(b1full);
	...
	process data item;
	...
	wait(b2empty);
	put the result to **B2**;
	signal(b2full);
	}
		while(1){
		wait(b2full);
		get result from **B2**;
		signal(b2empty);
		print the result;
		}

(d)

(i) Possible scenarios when deadlock occurs:

- When full = 0, empty = n, mutex = 1

Producer process	Consumer process
	wait(mutex); //mutex become 0
	wait(full); //locked as full = 0
wait(mutex); //locked as mutex = 0	

- When full = n, empty = 0, mutex = 1

Producer process	Consumer process
wait(mutex); //mutex become 0	
wait(empty); //locked as full = 0	
	wait(mutex); //locked as mutex = 0

(ii) One way to solve this problem is to modify the code for the consumer and producer processes:

Producer process:

```
...
while(1){
...
wait(empty);
wait(mutex);
...
}
```

Consumer process:

```
...
while(1){
...
wait(full);
wait(mutex);
...
}
```

3. (a)

(i) True. In load time address binding absolute address format, where logical address(address generated by the CPU) = physical address, is used in process image.(Slide 6.7)

(ii) True. Since each program is broken into segments of different sizes, then as program leaves the system, it creates holes of varying size in the memory outside the segments→ external fragmentation.

(iii) False. If the hit ratio is too small then the effective memory access time may not be reduced, that is when TLBs Lookup(ε) ≥ Memory cycle time(μ) x Hit ratio(α)

(iv) True. If a code is not reentrant, it contains global functions or variables. These codes cannot be shared among process, because different processes may not share the same variables.

(v) False. When page thrashing occurs then increasing the degree of multiprogramming will reduce the CPU utilization as each process is busy bringing pages in and out (I/O).

(b)

Advantage: Reduce the amount of memory consumed for page table

Disadvantage: It increases search time for an entry as the page table is sorted by physical address but lookups occur on logical address

(c)

$654321_{10} = 10011111101111110001_2$

(i) Page Table in single-level paging

Offset = $\log_2 512 = 9$ bits

p = 20 – 9 = 11 bits

Page table entry that should be used = $10011111101_2 = 1277_{10}$

(ii) Outer page-table in two-level paging

Offset = $\log_2 512$ = 9 bits

p2 = $\log_2$(page table size/ size of page table entry) = $\log_2(512/4) = \log_2(128)$ = 7 bits

p1 = 20 – 9 – 7 = 4 bits

Outer page table entry that should be used = $1001_2 = 9_{10}$

(d)

1) Trap to OS to bring in the page. The state of interrupted process is saved.

2) OS uses the page table entry to locate the page in backing store.

3) OS finds a victim frame (i.e., using free frame list & page replacement algorithm), and schedules a disk operation to read the desired page into the newly allocated frame.

4) Change the page table entry of the victim page to invalid

5) When the disk read is complete, page table entry is modified (change the valid-invalid bit into valid and set the page table entry to be the victim frame number).

6) The state of interrupted process is resumed, and the instruction is now re-started.

Major overhead:

- Servicing the page fault interrupt
- Executing page replacement algorithm
- Read in a new page (longest component)
- Restart the process

4. (a)

(i) False. For naming characteristic, tree-structured directories only allows files under different directories to have the same name. (Slide 8.16)

(ii) True. Using symbolic link, we need to resolve the link by following the path name to locate the real file/entry. (Slide 8.19)

(iii) False. Only one user is able to list the content of the directory because execute permission is essential to execute any command on the directory. Hence, if a user has no execute permission, he can't execute any command on the directory even if he has the read/write permission. (Slide 8.23)

(iv) True. In all file allocation methods, each file is divided into blocks of certain size, the file may not fully occupy the last allocated block and creates internal fragmentation.

(v) False. Device drivers are part of device-driver layer which is a layer below the kernel I/O subsystem layer. Device drivers hide differences among I/O controllers from kernel, while the kernel I/O subsystem provides generic services.

(b)

(i) Contiguous allocation

Operation: read block 6→write block 5(with data read from block 6)→read block 7→write block 6→read block 8 →write block 7→read block 9→ write block 8

Total read operations = 4

Total write operations = 4

(ii) Linked allocation

Operation: read block 0→ read block 1→ read block 2→ read block 3→ read block 4→ read block 5→write to block 4 (to link block 4 and block 6)

Total read operations = 6

Total write operations = 1

(iii) Indexed allocation

Operation: read the index block→ write to index block(change pointer number 5)→ write to index block(change pointer number 6)→ write to index block(change pointer number 7)→ write to index block(change pointer number 8)

Total read operations = 1

Total write operations = 4

(c)

(i)

SCAN

|60- 75| + |45 - 60| + |20 - 45| + |0 - 20| + |95 - 0| + |125 - 95| + |150 - 125| + |185 - 150|

= 250

C-SCAN

|60- 75| + |45 - 60| + |20 - 45| + |0 - 20| + |199 - 0| + |185 - 199| + |150 - 185| + |125 - 150| + |95 - 125| = 378

SCAN will result in less head movement

(ii)

SCAN algorithm is also called the elevator algorithm since the disk arm behaves like an elevator in a building, first servicing all the requests going up and then servicing all requests as it is going down. One the other hand, C-SCAN scheduling algorithm treats the disk cylinder as a circular list, it will immediately return to the beginning of the disk without servicing any requests during the returning trip.

In the SCAN algorithm, and with the scenario where the requests are in one end of the disk and the head on the other end, the time consumed for the head to satisfy the requests is high in such scenario. On the other hand, given the same scenario, the C-SCAN algorithm will be a good solution where once the edge of the disk is reached, the head returns to the opposite edge without dealing with any requests. Hence, it will reduce the maximum delay that may be experienced by a request as compared to SCAN algorithm.

For reporting of errors and errata, please visit pypdiscuss.appspot.com
Thank you and all the best for your exams! ☺