

Feedback on Quiz # 2 (Summative)

The **Quiz # 2** was composed of **10 questions**, which were randomly selected from a Question Bank of 30 questions. Answers and feedback comments for all of the questions are given below:

Q1.1	Which of the following statements are true?
(a)	A context switch can be triggered by an I/O request
(b)	Parent and child processes share the same process id
(c)	Child process can modify variables in the parent process
(d)	I/O bound processes have low CPU utilisation
Feedback: <ul style="list-style-type: none"> a) A running process can be blocked by an I/O request until the response from I/O device is ready. To do this, the state of the process should be saved and another process should be scheduled for execution because the CPU is free. In other words, a context switch is triggered. b) Parent and child processes have different process ids because they are different processes c) Each process has its own copy of the address space including a stack that contains variables. Hence, each process can only modify its own stack and variables. d) I/O bound processes execute I/O actions most of the time so the CPU has nothing to do as it waits for these I/O actions to complete. Hence, CPU utilisation is low. 	

Q1.2	Which of the following statements are true?
(a)	A context switch can be caused by an interrupt
(b)	Within the child process, the fork system call returns the process ID of the child process
(c)	A child process can not modify variables in the parent process
(d)	I/O bound processes have high CPU utilisation
Feedback: <ul style="list-style-type: none"> a) On a preemptive multitasking system, the scheduler may switch out processes that are still runnable. To prevent other processes from being starved of CPU time, pre-emptive schedulers often configure a timer interrupt to fire when a process exceeds its time slice. This interrupt ensures that the scheduler will gain control to perform a context switch. b) On success within the child process, the fork system call returns 0. c) Each process has its own copy of the address space including a stack that contains variables. Hence, each process can only modify its own stack and variables. d) I/O bound processes execute I/O actions most of the time so the CPU has nothing to do as it waits for these I/O actions to complete. Hence, CPU utilisation is low. 	

Q1.3	Which of the following statements are true?
(a)	A context switch can be caused by a page fault
(b)	Within the parent process, the fork system call returns the process ID of the child process
(c)	The whole process must reside in the memory at all times in order to execute

(d)	I/O bound processes have high CPU utilisation
Feedback: <ul style="list-style-type: none"> a) When a process tries to access the part of its address space that resides on a disk a page fault occurs and the process is suspended. As it is suspended, its state should be saved and another process should run, i.e. a context switch is performed. b) Within the parent process, the fork system call returns the process ID of the child process. c) Virtual address space makes it possible to execute processes that are not fully loaded into main memory: part of the process may reside on disk until it is accessed. d) I/O bound processes execute I/O actions most of the time so the CPU has nothing to do as it waits for these I/O actions to complete. Hence, CPU utilisation is low. 	

Q2.1	<p>A section of memory containing five memory locations is represented below, with each memory location containing an integer. Memory locations #2-#4 (highlighted in blue) are reserved for a stack. The first integer pushed to the stack was 7 then 13 and lastly 300.</p> <table border="1"> <thead> <tr> <th>#</th><th>Integer held in Memory Location</th></tr> </thead> <tbody> <tr> <td>5</td><td>42</td></tr> <tr> <td>4</td><td>300</td></tr> <tr> <td>3</td><td>13</td></tr> <tr> <td>2</td><td>7</td></tr> <tr> <td>1</td><td>85</td></tr> </tbody> </table> <p>What would this section of memory look like after three items are popped and then the number 50 is pushed to the stack?</p>	#	Integer held in Memory Location	5	42	4	300	3	13	2	7	1	85
#	Integer held in Memory Location												
5	42												
4	300												
3	13												
2	7												
1	85												
(a)	<table border="1"> <thead> <tr> <th>#</th><th>Integer held in Memory Location</th></tr> </thead> <tbody> <tr> <td>5</td><td>42</td></tr> <tr> <td>4</td><td>300</td></tr> <tr> <td>3</td><td>13</td></tr> <tr> <td>2</td><td>50</td></tr> <tr> <td>1</td><td>85</td></tr> </tbody> </table>	#	Integer held in Memory Location	5	42	4	300	3	13	2	50	1	85
#	Integer held in Memory Location												
5	42												
4	300												
3	13												
2	50												
1	85												
(b)													

	<table><tr><th>#</th><th>Integer held in Memory Location</th></tr><tr><td>5</td><td>42</td></tr><tr><td>4</td><td></td></tr><tr><td>3</td><td></td></tr><tr><td>2</td><td>50</td></tr><tr><td>1</td><td>85</td></tr></table>	#	Integer held in Memory Location	5	42	4		3		2	50	1	85
#	Integer held in Memory Location												
5	42												
4													
3													
2	50												
1	85												
(c)	<table><tr><th>#</th><th>Integer held in Memory Location</th></tr><tr><td>5</td><td>42</td></tr><tr><td>4</td><td>300</td></tr><tr><td>3</td><td>13</td></tr><tr><td>2</td><td>7</td></tr><tr><td>1</td><td>50</td></tr></table>	#	Integer held in Memory Location	5	42	4	300	3	13	2	7	1	50
#	Integer held in Memory Location												
5	42												
4	300												
3	13												
2	7												
1	50												
(d)	<table><tr><th>#</th><th>Integer held in Memory Location</th></tr><tr><td>5</td><td>42</td></tr><tr><td>4</td><td></td></tr><tr><td>3</td><td></td></tr><tr><td>2</td><td></td></tr><tr><td>1</td><td>50</td></tr></table>	#	Integer held in Memory Location	5	42	4		3		2		1	50
#	Integer held in Memory Location												
5	42												
4													
3													
2													
1	50												
Feedback: As 300 was the most recent addition to the stack, the stack pointer starts pointing at memory location #5.													

When an item is popped from the stack, it is read and the stack pointer is decremented – but it is not removed from memory. After three pops, the stack pointer is pointing at memory location #2.

When an item is pushed onto the stack, it is placed at the memory location that the stack pointer is currently pointing to, and the stack pointer is then incremented. After 50 is pushed onto the stack, it replaces the integer at memory location #2.

Q2.2 A section of memory containing six memory locations is represented below, with each memory location containing an integer. Memory locations #2-#5 (highlighted in blue) are reserved for a stack. The first integer pushed to the stack was 7 then 13 then 300 and lastly 42.

#	Integer held in Memory Location
6	999
5	42
4	300
3	13
2	7
1	85

What would this section of memory look like after four items are popped and then the numbers 50 and 51 (in that order) are pushed to the stack?

(a)

#	Integer held in Memory Location
6	999
5	42
4	300
3	51
2	50
1	85

(b)

	#	Integer held in Memory Location
	6	999
	5	
	4	
	3	51
	2	50
	1	85

(c)

#	Integer held in Memory Location
6	999
5	42
4	300
3	13
2	51
1	50

(d)

#	Integer held in Memory Location
6	999
5	
4	
3	
2	51
1	50

Feedback:

As 42 was the most recent addition to the stack, the stack pointer starts pointing at memory location #6.

When an item is popped from the stack, it is read and the stack pointer is decremented – but it is not removed from memory. After four pops, the stack pointer is pointing at memory location #2.

When an item is pushed onto the stack, it is placed at the memory location that the stack pointer is currently pointing to, and the stack pointer is then incremented. After 50 is pushed onto the stack, it replaces the integer at memory location #2 and then the stack pointer is pointing at memory location #3. After 51 is pushed onto the stack, it replaces the integer at memory location #3.

Q2.3 A section of memory containing five memory locations is represented below, with each memory location containing an integer. Memory locations #2-#4 (highlighted in blue) are reserved for a stack. The first integer pushed to the stack was 7 and then 13. The stack pointer is currently pointing at memory location #4.

#	Integer held in Memory Location
5	42
4	
3	13
2	7
1	85

What would this section of memory look like after the number 50 is pushed to the stack and then we pop the stack twice?

(a)

#	Integer held in Memory Location
5	42
4	50
3	13
2	7
1	85

(b)	<table> <tr> <th>#</th><th>Integer held in Memory Location</th></tr> <tr> <td>5</td><td>42</td></tr> <tr> <td>4</td><td></td></tr> <tr> <td>3</td><td></td></tr> <tr> <td>2</td><td>7</td></tr> <tr> <td>1</td><td>85</td></tr> </table>	#	Integer held in Memory Location	5	42	4		3		2	7	1	85
#	Integer held in Memory Location												
5	42												
4													
3													
2	7												
1	85												
(c)	<table> <tr> <th>#</th><th>Integer held in Memory Location</th></tr> <tr> <td>5</td><td></td></tr> <tr> <td>4</td><td></td></tr> <tr> <td>3</td><td>13</td></tr> <tr> <td>2</td><td>7</td></tr> <tr> <td>1</td><td>85</td></tr> </table>	#	Integer held in Memory Location	5		4		3	13	2	7	1	85
#	Integer held in Memory Location												
5													
4													
3	13												
2	7												
1	85												
(d)	<table> <tr> <th>#</th><th>Integer held in Memory Location</th></tr> <tr> <td>5</td><td>42</td></tr> <tr> <td>4</td><td>50</td></tr> <tr> <td>3</td><td></td></tr> <tr> <td>2</td><td></td></tr> <tr> <td>1</td><td>85</td></tr> </table>	#	Integer held in Memory Location	5	42	4	50	3		2		1	85
#	Integer held in Memory Location												
5	42												
4	50												
3													
2													
1	85												
Feedback: When an item is pushed onto the stack, it is placed at the memory location that the stack pointer is currently pointing to, and the stack pointer is then incremented. After 50 is pushed onto the stack, it													

replaces the integer at memory location #4, and then the stack pointer is pointing at memory location #5.

When an item is popped from the stack, it is read and the stack pointer is decremented – but it is not removed from memory.

Q3.1	Which of the following statements about memory are true?
(a)	Main memory does not persist after the machine has been switched off
(b)	Secondary storage does not persist after the machine has been switched off
(c)	If the CPU is loading data, the cache is the first location checked
(d)	ROM memory can only be modified infrequently
(e)	Memory is an array of bytes
Feedback: See Slides #9 and #10 of Week 4.2 - Elements of an OS Memory is an array of bytes where each byte is addressable. Main memory does not persist after the machine has been switched off, whereas secondary storage does. The cache is a faster storage system where data that has been used a lot is placed, and when the CPU looks for data the cache is checked first. ROM memory cannot be modified, and EEPROM memory can only be changed infrequently.	

Q3.2	Which of the following statements about memory are true?
(a)	Main memory persists after the machine has been switched off
(b)	Secondary storage persists after the machine has been switched off
(c)	The cache is a faster memory where data that has been used more frequently is placed.
(d)	EEPROM memory can only be modified infrequently
(e)	Memory is exclusively made up of registers
Feedback: See Slides #9 and #10 of Week 4.2 - Elements of an OS Memory is an array of bytes where each byte is addressable. Main memory does not persist after the machine has been switched off, whereas secondary storage does. The cache is a faster storage system where data that has been used a lot is placed, and when the CPU looks for data the cache is checked first. ROM memory cannot be modified, and EEPROM memory can only be changed infrequently.	

Q3.3	Which of the following statements about memory are true?
(a)	Main memory persists after the machine has been switched off
(b)	Secondary storage does not persist after the machine has been switched off
(c)	The cache is a faster memory where data that has been used more frequently is placed.
(d)	ROM is type of memory that cannot be modified
(e)	Each byte of memory is addressable
Feedback: See Slides #9 and #10 of Week 4.2 - Elements of an OS Memory is an array of bytes where each byte is addressable. Main memory does not persist after the machine has been switched off, whereas secondary storage does. The cache is a faster storage system where data that has been used a lot is placed, and when the CPU looks for data the cache is checked first. ROM memory cannot be modified, and EEPROM memory can only be changed infrequently.	

Q4.1	Calculate the 8-bit result of the following binary expression: (0011 1001 XOR 0111 1010) + (1100 0101 AND 0111 1011)
(a)	1000 0100
(b)	1011 1100
(c)	1000 0011
(d)	1011 1011
Feedback: First we perform the XOR ("exclusive-or") bitwise: <pre> 0011 1001 XOR 0111 1010 = 0100 0011 </pre> Then we perform the AND bitwise: <pre> 1100 0101 AND 0111 1011 = 0100 0001 </pre> And finally we add the two results: <pre> 0100 0011 + 0100 0001 = 1000 0100 </pre>	

Q4.2	Calculate the 8-bit result of the following binary expression: (0011 1001 OR 0111 1010) XOR (0100 0101 + 0111 1011)
(a)	1011 1011
(b)	1111 1011
(c)	1000 0011
(d)	1100 0011
Feedback: First we perform the OR bitwise: <pre> 0011 1001 OR 0111 1010 = 0111 1011 </pre> Then we perform the addition: <pre> 0100 0101 + 0111 1011 = 1100 0000 </pre> And finally we XOR (“exclusive-or”) the results bitwise: <pre> 0111 1011 XOR 1100 0000 = 1011 1011 </pre>	

Q4.3	Calculate the 8-bit result of the following binary expression: (0011 1001 + 0111 1010) AND (1100 0101 XOR 0111 1011)
(a)	1011 0010
(b)	1011 0011
(c)	0111 0001
(d)	0011 1000
Feedback: First we perform the addition: <pre> 0011 1001 + 0111 1010 = 1011 0011 </pre> Then we perform the XOR (“exclusive-or”) bitwise: <pre> 1100 0101 </pre>	

```
XOR 0111 1011
    = 1011 1110
```

And finally we AND the two results bitwise:

```
    1011 0011
AND 1011 1110
    = 1011 0010
```

Q5.1	Consider the following infix expression: (7 + (2 - 9 * 4)) Suppose that we are using the Shunting-Yard algorithm to convert the expression from infix to RPN (postfix notation). What will be the maximum number of symbols that will appear on the stack AT ONE TIME during the conversion of this expression? Note: The parenthesis () are also considered as symbols.
(a)	5 symbols
(b)	1 symbol
(c)	2 symbols
(d)	3 symbols
(e)	4 symbols

Feedback:

The expression: (7 + (2 - 9 * 4))

Will be converted to RPN using the following steps:

Step#	OUTPUT	STACK	INPUT
1	Empty	Empty	(7 + (2 - 9 * 4))
2	Empty	(7 + (2 - 9 * 4))
3	7	(+ (2 - 9 * 4))
4	7	(+	(2 - 9 * 4))
5	7	(+ (2 - 9 * 4))
6	7 2	(+ (- 9 * 4))
7	7 2	(+ (-	9 * 4))
8	7 2 9	(+ (-	* 4))
9	7 2 9	(+ (- *	4))
10	7 2 9 4	(+ (- *))
11	7 2 9 4 * -	(+)
12	7 2 9 4 * - +	Empty	Empty

We can see that the maximum number of symbols on the stack is **5** (during steps 9 & 10).

Q5.2 Consider the following infix expression:
 $4 + ((8 - 5) * 7)$
 Suppose that we are using the Shunting-Yard algorithm to convert the expression from infix to RPN (postfix notation). What will be the maximum number of symbols that will appear on the stack AT ONE TIME during the conversion of this expression?
Note: The parenthesis () are also considered as symbols.

(a) 5 symbols

(b) 1 symbol

(c) 2 symbols

(d) 3 symbols

(e) **4 symbols**

Feedback:

The expression: $4 + ((8 - 5) * 7)$

Will be converted to RPN using the following steps:

Step#	OUTPUT	STACK	INPUT
1	Empty	Empty	$4 + ((8 - 5) * 7)$
2	4	Empty	$+ ((8 - 5) * 7)$
3	4	+	$((8 - 5) * 7)$
4	4	+ ($(8 - 5) * 7)$
5	4	+ (($8 - 5) * 7)$
6	4 8	+ (($- 5) * 7)$
7	4 8	+ ((-	$5) * 7)$
8	4 8 5	+ ((-	$) * 7)$
9	4 8 5 -	+ ($* 7)$
10	4 8 5 -	+ (*	$7)$
11	4 8 5 - 7	+ (*)
12	4 8 5 - 7 *	+	Empty
13	4 8 5 - 7 * +	Empty	Empty

We can see that the maximum number of symbols on the stack is **4** (during steps 7 & 8).

Q5.3	Consider the following infix expression: (6 * 2 + (9 - 5)) Suppose that we are using the Shunting-Yard algorithm to convert the expression from infix to RPN (postfix notation). What will be the maximum number of symbols that will appear on the stack AT ONE TIME during the conversion of this expression? Note: The parenthesis () are also considered as symbols.
(a)	5 symbols
(b)	1 symbol
(c)	2 symbols
(d)	3 symbols
(e)	4 symbols

Feedback:


The expression: (6 * 2 + (9 - 5))


Will be converted to RPN using the following steps:

Step#	OUTPUT	STACK	INPUT
1	Empty	Empty	(6 * 2 + (9 - 5))
2	Empty	(6 * 2 + (9 - 5))
3	6	(* 2 + (9 - 5))
4	6	(*	2 + (9 - 5))
5	6 2	(*	+ (9 - 5))
6	6 2 *	(+	(9 - 5))
7	6 2 *	(+ (9 - 5))
8	6 2 * 9	(+ (- 5))
9	6 2 * 9	(+ (-	5))
10	6 2 * 9 5	(+ (-))
11	6 2 * 9 5 -	(+)
12	6 2 * 9 5 - +	Empty	Empty

We can see that the maximum number of symbols on the stack is 4 (during steps 9 & 10).

Q6.1	<p>Consider a processor with 2 levels of memory, Level 1 Cache and Level 2 Main Memory (RAM), represented in the figure below. Level 1 can store 128KB of data and has an access time of 12ns and Level 2 has an access time of 0.22μs and has a capacity of 256MB.</p> <p>Assume that, if a byte to be accessed is in level 1, then the processor accesses it directly. If it is in level 2 (RAM), then the byte is first transferred to level 1 and then accessed by the processor. For simplicity, we ignore the time required for the processor to determine whether the byte is in level 1 or level 2.</p> <p>We can define hit-ratio H, as the fraction of all memory accesses that are found in the faster memory (the cache memory). The hit-ratio H of cache is 97%. What is the average time to access a byte in such a two-level memory subsystem?</p> <div data-bbox="404 749 1313 831" data-label="Diagram"> <pre> graph LR CPU[CPU] --- CM[Cache Memory] CM --- MM[Main Memory] </pre> </div>
(a)	12.0066 ns
(b)	19.12 ns
(c)	6.612 ns
(d)	18.60 ns
<p>Feedback:</p> <p>The average access time is calculated as:</p> $T_{Avg} = \text{Hit Rate} * \text{Cache access time} + \text{Miss Rate} * \text{Lower-level access time}$ $T_{Avg} = H \times T_1 + (1 - H) \times (T_1 + T_2) \quad \text{OR} \quad T_{Avg} = T_1 + (1 - H) \times T_2$ <p>We know that $H = 0.97$, $T_1 = 12 \text{ ns}$ and $T_2 = 0.22 \mu\text{s} = 220 \text{ ns}$</p> $T_{Avg} = 0.97 \times 12 \text{ ns} + (1 - 0.97) \times (12 \text{ ns} + 220 \text{ ns})$ $= 11.64 \text{ ns} + 0.03 \times 232 \text{ ns} = \mathbf{18.60 \text{ ns}}$	

Q6.2	<p>Consider a processor with 2 levels of memory, Level 1 Cache and Level 2 Main Memory (RAM), represented in the figure below. Level 1 can store 64KB of data and has an access time of 9ns and Level 2 has an access time of 0.24μs and has a capacity of 128MB.</p> <p>Assume that if a byte to be accessed is in level 1, then the processor accesses it directly. If it is in level 2 (RAM), then the byte is first transferred to level 1 and then accessed by the processor. For simplicity, we ignore the time required for the processor to determine whether the byte is in level 1 or level 2.</p> <p>We can define hit-ratio H, as the fraction of all memory accesses that are found in the faster memory (the cache memory). The hit-ratio H of cache is 94%. What is the average time to access a byte in such a two-level memory subsystem?</p> <div style="text-align: center;">  <pre> graph LR CPU[CPU] --- CM[Cache Memory] CM --- MM[Main Memory] </pre> </div>
(a)	9.0144 ns
(b)	24.75 ns
(c)	14.409 ns
(d)	23.40 ns
<p>Feedback:</p> <p>The average access time is calculated as:</p> $T_{Avg} = \text{Hit Rate} \times \text{Cache access time} + \text{Miss Rate} \times \text{Lower-level access time}$ $T_{Avg} = H \times T_1 + (1 - H) \times (T_1 + T_2) \quad \text{OR} \quad T_{Avg} = T_1 + (1 - H) \times T_2$ <p>We know that $H = 0.94$, $T_1 = 9 \text{ ns}$ and $T_2 = 0.24 \mu\text{s} = 240 \text{ ns}$</p> $T_{Avg} = 0.94 \times 9 \text{ ns} + (1 - 0.94) \times (9 \text{ ns} + 240 \text{ ns})$ $= 8.46 \text{ ns} + 0.06 \times 249 \text{ ns} = \mathbf{23.40 \text{ ns}}$	

Q6.3	<p>Consider a processor with 2 levels of memory, Level 1 Cache and Level 2 Main Memory (RAM), represented in the figure below. Level 1 can store 256KB of data and has an access time of 15ns and Level 2 has an access time of 0.27μs and has a capacity of 512MB.</p> <p>Assume that if a byte to be accessed is in level 1, then the processor accesses it directly. If it is in level 2 (RAM), then the byte is first transferred to level 1 and then accessed by the processor. For simplicity, we ignore the time required for the processor to determine whether the byte is in level 1 or level 2.</p> <p>We can define hit-ratio H, as the fraction of all memory accesses that are found in the faster memory (the cache memory). The hit-ratio H of cache is 87%. What is the average time to access a byte in such a two-level memory subsystem?</p> <div style="text-align: center;">  <pre> graph LR CPU[CPU] --- CM[Cache Memory] CM --- MM[Main Memory] </pre> </div>
(a)	15.0351 ns
(b)	33.48 ns
(c)	35.115 ns
(d)	50.10 ns
<p>Feedback:</p> <p>The average access time is calculated as:</p> $T_{Avg} = \text{Hit Rate} \times \text{Cache access time} + \text{Miss Rate} \times \text{Lower-level access time}$ $T_{Avg} = H \times T_1 + (1 - H) \times (T_1 + T_2) \quad \text{OR} \quad T_{Avg} = T_1 + (1 - H) \times T_2$ <p>We know that $H = 0.87$, $T_1 = 15 \text{ ns}$ and $T_2 = 0.27 \mu\text{s} = 270 \text{ ns}$</p> $T_{Avg} = 0.87 \times 15 \text{ ns} + (1 - 0.87) \times (15 \text{ ns} + 270 \text{ ns})$ $= 13.05 \text{ ns} + 0.13 \times 285 \text{ ns} = \mathbf{50.10 \text{ ns}}$	

Q7.1	<p>Consider the following program in C that uses system call <code>fork</code>. What output to the terminal is possible after this program has completed execution? You can assume that calls to <code>printf</code> are atomic, i.e. different outputs can not interleave.</p> <p>Note: Please scroll down to see all the options.</p>
------	---

```

1  int main(void) {
2      fork();
3      fork();
4
5      printf("Hello\n");
6      printf("World\n");
7
8      wait(NULL);
9      return 0;
10 }
11
12
13

```

a)

```

1  Hello
2  World
3  Hello
4  World
5  Hello
6  Hello
7  World
8  World

```

b)

1	Hello
2	World

c)

1	Hello
2	World
3	Hello
4	Hello
5	World
6	World
7	Hello
8	World

d)

1	Hello
2	World
3	World
4	Hello
5	Hello
6	World
7	Hello
8	World

e)

1	Hello
2	World
3	Hello
4	World
5	Hello
6	World
7	Hello
8	World

Feedback:

As each call to `fork()` produces an extra process, after line 3 we will have 2 processes. Each of these processes will execute `fork()` on line 4 which will lead to 4 processes in total. Then, each of these 4 processes will first output "Hello\n" and then output "World\n". Remember, that processes are executed concurrently. For example, a process's time slice can run out just after line 6 and another process can be given a chance to run which would result in two "Hello" being printed.

We can consider "Hello\n" to be "(" and "World\n" to be ")". Then all the sequences the program can produce are well-balanced bracket sequences of length 8. That is, at any given time the program can not have printed more "World"s than "Hello"s. The option in d) corresponds to `()))(())` which is not well-balanced.

Q7.2	Consider the following program in C that uses system call <code>fork</code> . What output to the terminal is possible after this program has completed execution? You can assume that calls to <code>printf</code> are atomic, i.e. different outputs can not interleave.
------	---

```
1  int main(void) {
2
3      fork();
4      fork();
5
6      printf("Hello\n");
7      printf("World\n");
8
9      wait(NULL);
10     return 0;
11 }
12
13
```

a)

1	Hello
2	World
3	Hello
4	World
5	World
6	Hello
7	Hello
8	World

b)

1	Hello
2	World
3	Hello
4	World

c)

1	Hello
2	World
3	Hello
4	Hello
5	World
6	World
7	Hello
8	World

d)

1	Hello
2	World
3	World
4	Hello
5	Hello
6	World
7	Hello
8	World

e)

1	Hello
2	World
3	Hello
4	World
5	Hello
6	World
7	Hello
8	World

Feedback:

As each call to `fork()` produces an extra process, after line 3 we will have 2 processes. Each of these processes will execute `fork()` on line 4 which will lead to 4 processes in total. Then, each of these 4 processes will first output “Hello\n” and then output “World\n”. Remember, that processes are executed concurrently. For example, a process’s time slice can run out just after line 6 and another process can be given a chance to run which would result in two “Hello” being printed.

We can consider “Hello\n” to be “(” and “World\n” to be “)”. Then all the sequences the program can produce are well-balanced bracket sequences of length 8. That is, at any given time the program can not have printed more “World”s than “Hello”s.

The option in a) corresponds to `()()()` and the option in d) corresponds to `()()()` which are not well-balanced.

Q7.3	Consider the following program in C that uses system call <code>fork</code> . What output to the terminal is possible after this program has completed execution? You can assume that calls to <code>printf</code> are atomic, i.e. different outputs can not interleave.
------	---

```
1  int main(void) {  
2  
3      fork();  
4      fork();  
5  
6      printf("Hello\n");  
7      printf("World\n");  
8  
9      wait(NULL);  
10     return 0;  
11 }  
12  
13
```

a)

```
1  Hello  
2  World  
3  Hello  
4  World  
5  Hello  
6  World  
7  Hello  
8  World
```

b)

1	Hello
2	World
3	Hello
4	World
5	Hello
6	World
7	World
8	Hello

c)

1	Hello
2	World
3	Hello
4	World

d)

1	Hello
2	World
3	Hello
4	Hello
5	World
6	World
7	Hello
8	World

e)

1	Hello
2	World
3	World
4	Hello
5	Hello
6	World
7	Hello
8	World

Feedback:

As each call to `fork()` produces an extra process, after line 3 we will have 2 processes. Each of these processes will execute `fork()` on line 4 which will lead to 4 processes in total. Then, each of these 4 processes will first output "Hello\n" and then output "World\n". Remember, that processes are executed concurrently. For example, a process's time slice can run out just after line 6 and another process can be given a chance to run which would result in two "Hello" being printed.

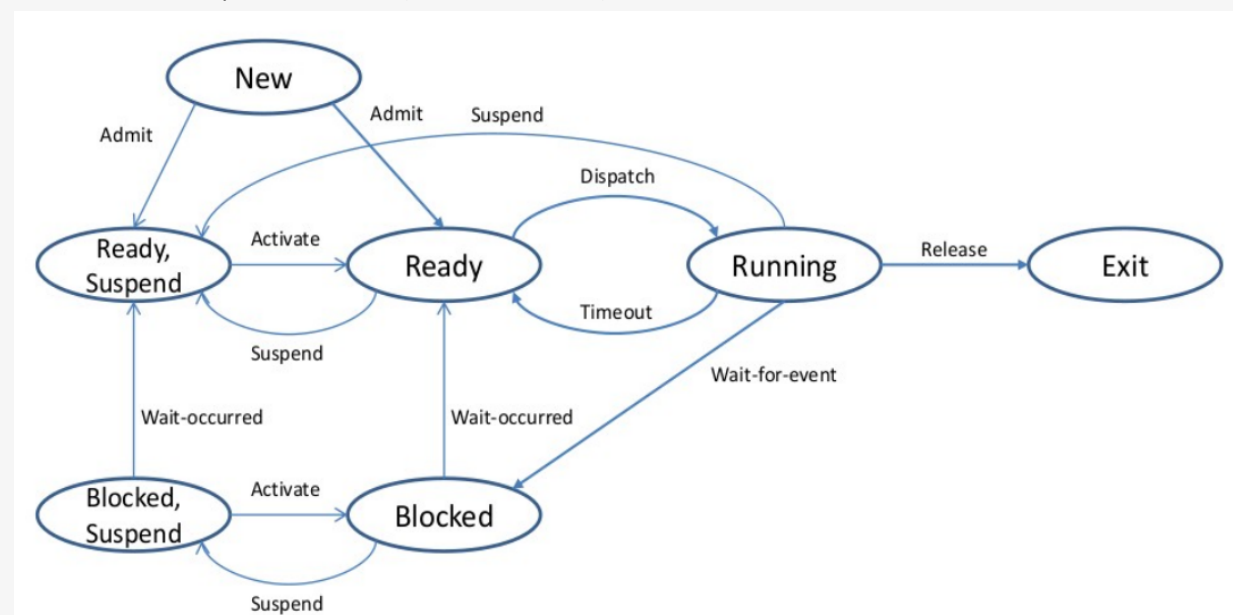
We can consider "Hello\n" to be "(" and "World\n" to be ")". Then all the sequences the program can produce are well-balanced bracket sequences of length 8. That is, at any given time the program can not have printed more "World"s than "Hello"s.

The option in b) corresponds to `()()()()` and the option in e) corresponds to `()()(())` which are not well-balanced.

Q8.1	Recall the 7-state process model and identify the state transitions that are not possible:
(a)	A running process makes an I/O request and after some time (while the I/O request is in progress), it is moved to the secondary storage by the medium term scheduler.
(b)	A ready process starts executing and consumes its time-slice, the process is then directly moved to the swap area by the medium term scheduler.
(c)	A running process consumes its time-slice, moves to the blocked state and then becomes ready once the I/O device sends an interrupt.
(d)	A new process is admitted to the ready queue, but as the system is under memory pressure, this process is moved to the swap area.

Feedback:

Recall the 7-state process model (as shown below)



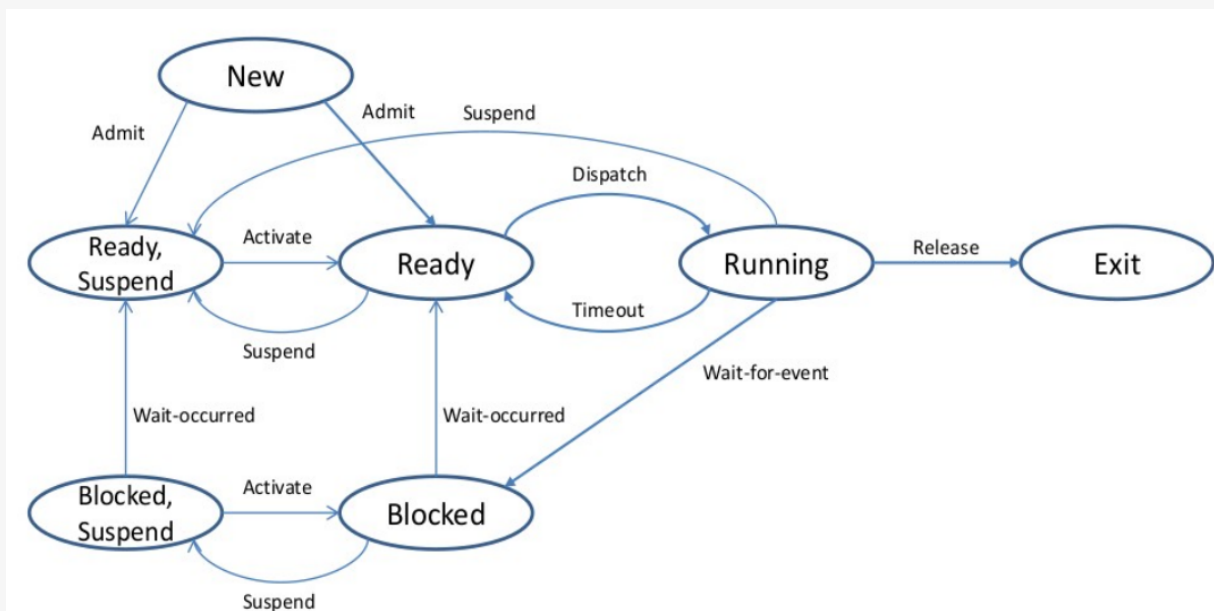
Here are the explanations for the above options:

- When a "Running" process makes an I/O request, it is moved to the "Blocked" state, and if needed the medium term scheduler can move it to the "Blocked, Suspend" state on the swap space.
- A "Ready" process starts executing, and once it completes its time-slice, it is moved to the "Ready" state. The process is not moved to the "Ready, Suspend" state on time-outs.
- A "Running" process, once it completes its time-slice, is moved to the "Ready" state not the "Blocked" state. The process is not waiting for any I/O operation, hence it does not depend on the I/O device sending an interrupt.
- A new process, once admitted to the "Ready" queue, can either start "Running" if selected by the scheduler or it may be moved to the swap area and placed in "Ready, Suspend" state.

Q8.2	Recall the 7-state process model and identify the state transitions that are not possible:
(a)	A ready process makes an I/O request and after some time, it is moved to the secondary storage by the medium term scheduler.
(b)	A running process consumes its time-slice, and is moved to the swap area by the medium term scheduler, after passing through the ready state.
(c)	A running process consumes its time-slice, moves to the ready state and then becomes blocked once its I/O request reaches the concerned device.
(d)	A process waiting for an I/O request is moved to the swap area as the system is under memory pressure. Once the I/O request completes, the process can change to another state while it is still in the swap area.

Feedback:

Recall the 7-state process model (as shown below)



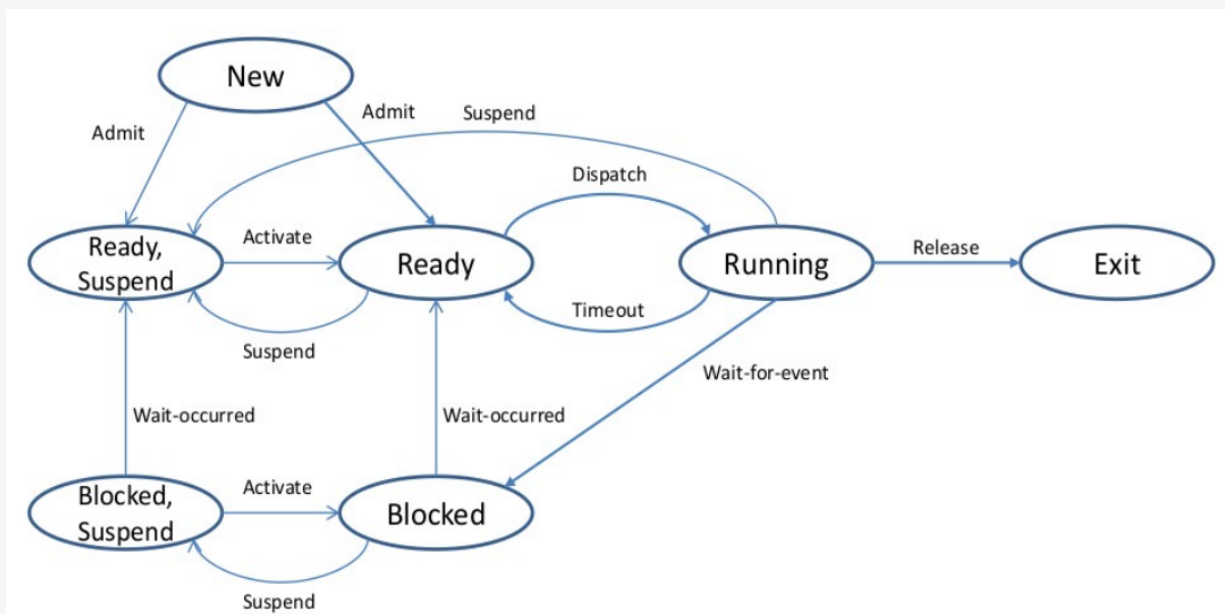
Here are the explanations for the above options:

- A "Ready" process cannot make an I/O request.
- A "Running" process, once it completes its time-slice, is moved to the "Ready" state. The process can then be moved to the "Ready, Suspend" by the medium term scheduler.
- A "Running" process, once it completes its time-slice, is moved to the "Ready" state. The process cannot move to the "Blocked" state, without running again.
- A process waiting for an I/O request in "Blocked" state can be moved to the "Blocked, Suspend" state, while the I/O is still in progress. On completion of the I/O request, the process can be moved to the "Ready, Suspend" state while it is still in the swap area.

Q8.3	Recall the 7-state process model and identify the state transitions that are not possible:
(a)	A process currently on the swap area and waiting for an I/O request, starts running as soon as the I/O request is completed.
(b)	A running process consumes its time-slice, and is moved to the ready state. Assuming that the system is under high demand, the process releases the CPU and exits.
(c)	A new process can be suspended immediately by the medium term scheduler, even though the process hasn't executed anything yet.
(d)	A process waiting for an I/O request is moved to the swap area as the system is under memory pressure. Once the I/O request completes, the process can change to another state while it is still in the swap area.

Feedback:

Recall the 7-state process model (as shown below)



Here are the explanations for the above options:

- A process currently on the swap area and waiting for the I/O request completion is in "Blocked, Suspend" state. It cannot start running, without going through the "Ready, Suspend" and "Ready" states.
- A "Running" process, once it completes its time-slice, is moved to the "Ready" state. The process cannot "Exit" the system, without going through the "Running" state, even if the system is under high demand.
- A "New" process can be placed in the "Ready, Suspend" state, if the system is under memory pressure. Once the system is able to, the medium term scheduler will bring the process in memory and place it in the "Ready" state.
- A process waiting for an I/O request in "Blocked" state can be moved to the "Blocked, Suspend" state, while the I/O is still in progress. On completion of the I/O request, the process can be moved to the "Ready, Suspend" state while it is still in the swap area.

Q9.1	<p>Consider the algorithm below:</p> <pre> i=1 sum=0 val=0 for i=1 .. n sum = sum + i for i=1 ..n for j=1 .. 10*n for k=1 .. 100 val = val + i*j*k </pre> <p>Using big-O notation, what is the complexity of this algorithm?</p>
(a)	O(n²)
(b)	O(10*n ²)
(c)	O(n+n ²)
(d)	O(n+10n ²)
(e)	O(n ³)
(f)	O(n)
<p>Feedback:</p> <p>The first loop is O(n)</p> <p>The second loop is O(n²) - notice that the innermost loop is not dependent on the value of n.</p> <p>We are not concerned with constants in this analysis so a loop that goes from 1 .. 10*n is regarded as O(n)</p> <p>If we have independent loops, we take the one with the greatest complexity: So O(n)+O(n²) reduces to O(n²) - because the more complex component dominates as n increases.</p> <p>If there are nested loops then we multiply the complexity - so a loop O(n) within a loop O(n) will have a combined complexity of O(n*n) which is written O(n²)</p>	

Q9.2	<p>Consider the algorithm below:</p> <pre> i=1 sum=0 val=0 for i=1 .. n sum = sum + i for i=1 ..n for j=1 .. 100*n for k=1 .. 50 val = val + i*j*k </pre> <p>Using big-O notation, what is the complexity of this algorithm?</p>
(a)	O(n²)
(b)	O(100*n ²)
(c)	O(n+n ²)
(d)	O(n ³)
(e)	O(n+n ³)
(f)	O(n)
<p>Feedback:</p> <p>The first loop is O(n)</p> <p>The second loop is O(n²). There is a third nested loop within this but it does not vary with n.</p> <p>We are not concerned with constants in this analysis so a loop that goes from 1 .. 100*n is regarded as O(n)</p> <p>If we have independent loops, we take the one with the greatest complexity: So O(n)+O(n²) reduces to O(n²) - because the more complex component dominates as n increases.</p> <p>If there are nested loops then we multiply the complexity - so a loop O(n) within a loop O(n) will have a combined complexity of O(n*n) which is written O(n²)</p>	

Q9.3	<p>Consider the algorithm below:</p> <pre> i=1 sum=0 val=0 for i=1 .. n sum = sum + i for i=1 ..n for j=1 .. 2*n for k=1 .. 500 val = val + i*j*k </pre> <p>Using big-O notation, what is the complexity of this algorithm?</p>
(a)	O(n²)
(b)	O(n+n ²)
(c)	O(2*n ²)
(d)	O(n ³)
(e)	O(n+n ³)
(f)	O(n)
<p>Feedback:</p> <p>The first loop is O(n)</p> <p>The second loop is O(n²) - there is a third nested loop but this is of constant time because it does not depend on the value of n.</p> <p>We are not concerned with constants in this analysis so a loop that goes from 1 .. 2*n is regarded as O(n)</p> <p>If we have independent loops, we take the one with the greatest complexity: So O(n)+O(n²) reduces to O(n²) - because the more complex component dominates as n increases.</p> <p>If there are nested loops then we multiply the complexity - so a loop O(n) within a loop O(n) will have a combined complexity of O(n*n) which is written O(n²)</p>	

Q10.1	A program you have developed has a fixed startup time and a main algorithm which you know is $O(n^3)$. You have tested the program with 1 data item, and it takes 100 ms to execute. With 1000 data items it takes 350 ms to run. Estimate how long (in ms) this algorithm will take for 3000 data items?
(a)	6850 ms
(b)	2350 ms
(c)	850 ms
(d)	4750 ms
(e)	9450 ms
Feedback: With 1 item it takes 100 ms. This is approx. the fixed cost of the algorithm. With $n=1000$ it takes 350 ms: 100 ms startup time + 250 ms to process the 1000 items. The main algorithm is $O(n^3)$. Therefore, if we have 3 times as much data, we can estimate that it will take $3^3 = 27$ times as long to execute that part of the program + the fixed startup time. Therefore, it will take $100 + 27 \cdot 250 = 100 + 6750 = 6850$ ms	

Q10.2	A program you have developed has a fixed startup time and a main algorithm which you know is $O(n^4)$. You have tested the program with 1 data item, and it takes 50 ms to execute. With 500 data items it takes 270 ms to run. Estimate how long (in ms) this algorithm will take for 1500 data items?
(a)	17870 ms
(b)	5990 ms
(c)	710 ms
(d)	11930 ms
(e)	21870 ms
Feedback: With 1 item it takes 50 ms. This is approx. the fixed cost of the algorithm. With $n=500$ it takes 270 ms: 50 ms startup time + 220 ms to process the 500 items. The main algorithm is $O(n^4)$. Therefore, if we have 3 times as much data, we can estimate that it will take $3^4 = 81$ times as long to execute that part of the program + the fixed startup time. Therefore, it will take $50 + 81 \cdot 220 = 50 + 17820 = 17870$ ms	

Q10.3	A program you have developed has a fixed startup time and a main algorithm which you know is $O(n^5)$. You have tested the program with 1 data item, and it takes 200 ms to execute. With 2000 data items it takes 460 ms to run. Estimate how long (in ms) this algorithm will take for 4000 data items?
(a)	8520 ms
(b)	4360 ms
(c)	1240 ms
(d)	6440 ms
(e)	14720 ms
Feedback: With 1 item it takes 200 ms. This is approx. the fixed cost of the algorithm. With $n=2000$ it takes 460 ms: 200 ms startup time + 260 ms to process the 2000 items. The main algorithm is $O(n^5)$. Therefore, if we have 2 times as much data, we can estimate that it will take $2^5 = 32$ times as long to execute that part of the program + the fixed startup time. Therefore, it will take $200 + 32 \cdot 260 = 100 + 8320 = 8520$ ms	