National University of Computer and Emerging Sciences, Lahore Campus

	Course:	Operating System	Course Code:	CS-205
	Program:	BS(Computer Science)	Semester:	Fall 2018
	Duration:	3 hour	Total Marks:	80
	Paper Date:	21st December, 2018	Weight:	45%
	Section:		Page(s):	6
	Exam:	Final	Roll No.	

Instructions/Notes: Answer questions on the question paper. Write answers clearly and precisely, if the answers are not easily readable then it will result in deduction of marks. Use extra sheet for rough work, **cutting and blotting on this sheet will result in deduction of marks**.

Question 1 (2 points): Two processes can communicate with each other using

1. Process IDs
2. Pipes
3. Process control blocks
4. Page tables

Question 2 (2 points): In order to copy the content of one file and write them to another file, multiple systems calls must be executed

1. True
2. False

Question 3 (2 points): Which of the following technique uses kernel memory space to do interprocess communication

1. Stack Segment
2. Message Queues
3. Anonymous/Ordinary Pipes
4. Shared Memory

Question 4 (2 points): Which of the following type of scheduling requires the use of a special hardware to improve the time in repeated context switching

1. Preemptive scheduling
2. Non-preemptive scheduling
3. Cooperative Scheduling
4. None of the above

Question 5 (2 points): Too much context switching

1. Provides a higher degree of multiprogramming
2. Results in degrading the performance
3. Is suitable for non-interactive processes
4. Helps avoiding starvation issues

Question 6 (2 points): Which of the following table in Linux keeps a copy of File Control Block (inode) of each opened file in memory

1. Mount table
2. Per-process open file table
3. System-wide open file table
4. Directory entry table

Question 7 (2 points): Which of the following phenomenon can reduce the number of page faults without degrading the performance

1. Pager
2. Better page replacement algorithm
3. Increased preemption of scheduler
4. Increasing the wait in demand paging

Question 8 (2 points): Dynamic loading and demand paging are both dynamic techniques. The difference between them is that the former is a programmer level decision and demand paging is system-level decision.

1. True
2. False

Question 9 (2 points): Generally, a page whose modified/dirty bit has been set to 1 should not be swapped out, if there are some unmodified/non-dirty pages available for swapping

1. True
2. False

Question 10 (2 points): The two common implementation methods of LRU page replacement algorithm are via (choose two)

1. Stack
2. Counting
3. Tree
4. Queue

Question 11 (2 points): The two primary jobs of a memory management unit (MMU) are (choose two)

1. Reduce fragmentation
2. Address translation
3. Memory protection
4. Increase cohesion

Question 12 (2 points): In Linux, the data structure that can be generally referred to as the process control block (PCB) is called

1. Job queue
2. task_struct
3. Process structure
4. Job Structure

Question 13 (5 points): On a virtual memory system we have a page size of 1 MB. If 1 byte is accessed in 1 nanoseconds, and 1 MB data is transferred from memory to the permanent storage in 1000 nanoseconds. The same 1000 nanoseconds are consumed to transfer 1 MB data from permanent storage to memory. Now calculate the time (in nanoseconds) of accessing a single byte in **worst case** scenario. Also mention the time for each step.

Question 14 (5 points): According to the same scenario as above, calculate the time for the best case scenario. Mention the time for each step and the accumulative time.

Question 15 (10 points): Implement a function `getByte`. The function is an imitation of an MMU. It fetches the data inside a byte from physical memory, given the logical address of the byte along with its process ID. You must use all the helper functions to complete the task.

```
#define PAGE_SIZE 123123 // the value does not matter
int* getPageTable(int PID); // takes the process ID as input and returns its page table.
boolean isPageValid(int pageNumber, int* pageTable); // takes the page number and page table
    as input and returns whether page is loaded into memory or not. Returns true if valid.
void handlePageFault(int pageNumber, int PID, int* pageTable); // takes the page number,
    process ID and page table as inputs and loads page into a frame and updates the page table
     accordingly.
byte readAddress(int frameNumber, int byteNumber); // takes the frame number and byte number
    as inputs and returns the value of the byte to be read.
```

```
byte getByte( int logicalAddress, int PID)//the address to be read and the process ID are the
    parameters of the function.
{











}
```

Question 16 (10 points): Three processes are running in parallel sharing variables i and j, which are initialized as $i = 1$ and $j = 1$. You have to synchronize the following processes such that the output on the console is the Fibonacci series. First few elements are as given; $1, 1, 2, 3, 5, \ldots$

NOTE: You can change the following code only by calling the **wait** and **signal** methods of semaphores. No other change in the code is allowed. You should use semaphores **judiciously**. Also, you have to mention the **initial values** of all the semaphores. You cannot add any statement which alters the values of variables i and j.

Initial values:

Process 1	Process 2	Process 3
`while(true)`	`while(true)`	`while(true)`
`{`	`{`	`{`
`j = i + j;`	`i = i + j;`	`cout << j<<",";`
		`cout << i<<",";`
`}`	`}`	
		`}`

Question 17 (10 points): Get the physical byte stored in a file which exists in a file system that uses single indexed table. Parameters are the logical address of the byte, and the file ID.

```
#define BLOCK_SIZE  xxxx; // tells how many bytes are there in one block
int getIndexBlockNumber(int fileID);// takes the file ID and returns the block where index
    table is stored.
int* loadIndexFromBlock(int blockNumber);// takes the block number and loads the index table
    in memory and returns its pointer.
byte* loadBytesFromBlock(int blockNumber); // takes the block number and loads raw bytes in
    that block in memory, and returns its address.
```

```
int getByte(int logicalByteNumber, int fileID)
{

}
```

Question 18 (10 points): Given five memory partitions of size 100KB, 500KB, 200KB, 300KB, and 600KB, as shown in Table 1. How would each of the first-fit, best-fit, and worst-fit algorithms place processes of 212KB, 417KB, 112KB, and 426KB, as shown in Table 2? Also fill in the blank in the end.
To answer the question fill out Tables 3, 4 and 5. When you fill a hole you might create another one with smaller size. In the last column the remaining size of the hole should be written. The new hole should also be used for filling, if needed.

Hole Number	Size
1	100 KB
2	500 KB
3	200 KB
4	300 KB
5	600 KB

Table 1: List of holes in memory as maintained by OS. Their order in the list is same as their position in the memory, and their position in the list of holes.

Process Number	Size
1	212 KB
2	417 KB
3	112 KB
4	426 KB

Table 2: List of process with their numbers and sizes. Small number shows that the process arrived earlier. So the order here represents order of their arrival.

- The ______________________________ algorithm accommodates all processes, but other algorithms do not. **(1)**

Process #	Process Size	Hole# where Process is loaded	Remaining Size of the hole
1	212 KB	2	500-212 = 288 KB
2	417 KB		
3	112 KB		
4	421 KB		

Table 3: Allocation of memory holes to processes according to First-Fit Algorithm.

Process #	Process Size	Hole# where Process is loaded	Remaining Size of the hole
1	212 KB	4	300-212 = 88 KB
2	417 KB		
3	112 KB		
4	426 KB		

Table 4: Allocation of memory holes to processes according to Best-Fit Algorithm.

Process #	Process Size	Hole# where Process is loaded	Remaining Size of the hole
1	212 KB	5	600-212 = 388 KB
2	417 KB		
3	112 KB		
4	426 KB		

Table 5: Allocation of memory holes to processes according to Worst-Fit Algorithm.

Question 19 (6 points): Tell the output of the following code. Assume that each instruction runs in the order. Meaning instructions written on smaller line numbers will necessarily execute before the instructions written on bigger line numbers.

```
1         int main(void)
2         {
3         int pid=0;
4         pid = fork();
5         x = 1;
6         if ( pid == 0)
7         {
8                 printf("%d,", x);
9                 pid = fork();
10                x++;
11                if ( pid == 0)
12                {
13                        printf("%d,", x);
14                        x++;
15                        pid = fork();
16                        if ( pid > 0)
17                        {
18                                x++;
19                                printf("%d,", x);
20                                pid = fork();
21                                if ( pid > 0)
22                                {
23                                        printf("%d,", x);
24                                        pid = fork();
25                                        if ( pid == 0)
26                                        {
27                                                x++;
28                                                printf("%d,", x);
29                                        }
30                                }
31                        }
32                }
33
34        }
35        else if (pid > 0)
36        {
37                printf("%d,", x);
38        }
39        return 0;
40        }
```