National University of Computer and Emerging Sciences, Lahore Campus

Course Name:	Operating Systems		Course Code:	CS-205
Program:	BS (CS)		Semester:	Spring 2018
Duration:	Three hours		Total Marks:	60
Paper Date:	18-May-2018		Weight	
Section:	ALL		Page(s):	7
Exam Type:	Final			

Student : Name:_____ Roll No._____ Section:_____

Question 1 (10 marks)

Consider a variation of the Indexed allocation method: The start block has N pointers. The first $N-1$ pointers point to data blocks, while the last pointer points to another index block. The second index block holds addresses of only data blocks; no more index blocks.

Now write a C/C++ function for address translation in such a variation. Use the following prototype:

```
int translate(int start, int log)
```

The function takes the physical address of the start block, and a logical block number, and computes/returns the corresponding physical block number.

Assume you have a system call to read a disk block into an array:

```
void read(int blockNo, int* array)
```

```
int translate(int start, int log) {
     int a[N];
     read(start, a);
     if (log < N-1)
          return a[log];

     int rem = log - (N-1);     // remaining
     read(a[N-1], a);
     return a[rem];
}
```

Question 2(5+5 marks)

a) Show execution of the LRU page replacement algorithm on the following page reference string:

    1   2   3   1   4   5   3   2   1

Assume there are only three frames in the RAM. (Please note that the number of boxes below maybe less or more depending upon the question. It, certainly, does not mean you have to utilize exactly the given number of boxes.)

1	1	1	1	1	3	3	3	
	2	2	4	4	4	2	2	
		3	3	5	5	5	1	

b) Reorder the following steps in handling a page fault so that it is correct:

1. The OS restarts the interrupted instruction
2. CPU executes a memory referencing instruction
3. The OS swaps out the victim page
4. The OS loads the desired page
5. MMU looks up the TLB to find invalid entry

Enter the correct order of steps in the box below:

2   5   3   4   1

Question 3 (5+5)

a) Consider a paging system with a page size of 256. Assume a process running in this system has the following page table:

50	10	90
0	1	2

Now translate the following logical addresses into the corresponding physical addresses:

i) 700   ii) 450

i)	ii)
page = 700 / 256 = 2	page = 450 / 256 = 1
frame = tbl[2] = 90	frame = tbl[1] = 10
offset = 700 mod 256 = 188	offset = 450 mod 256 = 194
phy add = 90 * 256 + 188	phy add = 10 * 256 + 194
= 23,228	= 2,754

b) Consider a system with a memory access time of 100 ns, and a page-fault service time of 7 milliseconds (including the memory access time). Assuming a page fault rate of 10%, compute the effective-access-time. (help: 1 sec = 1000 ms = 10^9 ns)

EAT = 10/100 * 7,000,000 + 90/100 * 100

        = 700,090 ns

Question 4 (10 marks)

Consider the following code for a simple Stack:

```
class Stack {
private:
    int* a;    // array for stack
    int max;   // max size of array
    int top;   // stack top
public:
    Stack(int m) {
        a = new int[m];  max = m;   top = 0;
    }
    void push(int x) {
        while (top == max)
            ;     // if stack is full then wait
        a[top] = x;
        ++top;
    }
    int pop() {
        while (top == 0)
            ;     // if stack is empty then wait
        int tmp = top;
        --top;
        return a[tmp];
    }
};
```

You can see from the code that a process blocks if it calls push() when the stack is full, or it calls pop() when the stack is empty, the same behavior should be present in the answer. Assuming that the functions push and pop can execute concurrently, synchronize the code using semaphores. Also eliminate the busy waiting.

```cpp
Semaphore full = 0;
Semaphore empty = MAX;
Semaphore mutex = 1;

class Stack {
private:
    int* a;    // array for stack
    int max;   // max size of array
    int top;   // stack top
public:
    Stack(int m) {
        a = new int[m];
        max = m;
        top = 0;
    }
    void push(int x) {
        wait(empty);
        wait(mutex);
        a[top] = x;
        ++top;
        signal(mutex);
        signal(full);
    }
    int pop() {
        wait(full);
        wait(mutex);
        int tmp = top;
        --top;
        signal(mutex);
        signal(empty);
        return a[tmp];
    }
};
```

Question 5 (5+5 points)

a)   Give all possible outputs for the following program:

```cpp
int main() {
    int x = 1;
    ++x;  // 2
    cout << x << endl;
    pid_t pid = fork();
    if (pid == 0) {
        ++x;
        cout << x << endl;
    }
    else if (pid > 0) {
        x = x + 3;
        cout << x << endl;
    }
    ++x;
    cout << x << endl;
    return 0;
}
```

2	2	2	2	2	2
3	5	3	5	3	5
5	3	5	3	4	6
4	4	6	6	5	3
6	6	4	4	6	4

b)   Depending upon what technique will be suited best for the requirements given below, write "*shared memory*" or "*message passing*" in front of each of the following phrases:

   i.   matrix multiplication   (shared)

   ii.   multiple programs running one after another   (message)

   iii.   email   (message)

   iv.   people meeting in a room   (shared)

   v.   the IPC method that requires synchronization   (shared)

Question 6 (10 points)
You have to write the code of the round-robin scheduling algorithm. You will implement the function **RoundRobin** whose skeleton is provided below. The function is called whenever a dispatch occurs, and it returns the process to be executed. It has one parameter called **inProc**. It is the pointer to the PCB of the process which was in execution before dispatch.

Below given is the definition of the process control block. Also, you have an object of type **Queue** globally defined as **readyQueue**. It stores the pointers to the Process Control Blocks which are ready to run. Use **readyQueue** and its functions as defined below to implement the function **RoundRobin**.

```
struct PCB{
        int PID; // PID of the process
        int remainingTime;// the number of cycles remaining for the process to complete.
...  // other elements
};

class Queue {
        int enqueu(PCB* proc); // adds an element of type PCB* into the queue
        PCB* dequeu(); // removes an element of type PCB* from the queue
        PCB* operator [] (const int index); // to enable the class to work like an array.
        int size(); //returns the number of elements inside the queue.
};
Queue readyQueue;
```

PCB* **RoundRobin**(PCB* inProc) {

```
        if (inProc->remainingTime > 0)
             readyQueue.enqueu(inProc);

        return readyQueue.dequeu();
```