CSE321

Take-Home Quiz 3.

Name: Umme Abira Azmary

ID: 20101539

Section: 06

Ans. to the ques. no - 1 :

Multithreading may not always provide better performance than a single-threaded solution while handling repetitive tasks. for example: the computation of factorial involves repetitive multiplication. As a result, splitting the computation into multithread may generate ambiguity in synchronization and degrade potential performance gain.

```
double factorial(int n){
    if (n<0){
        return -1;
    }
    double res = 1;
    for(int i=2; i<=n; i++){
        res *= i;
    }
    return result;
}
```

If this factorial code is running in multithreads, it will not showcase any better performance than a single-thread as most of the segments are serial types of segments.

Ans. to the ques. no- 02:

```
pid_t pid;
pid = fork(); ——→ f1.
if(pid == 0){
   fork(); ——→ f2
   thread create(...);
}
fork();
```

(a) In the above code segment, 6 unique processes are created. The parent creates 2 child process, child1 process creates 2 new process and child 2 process creates 1 new process.

So, parent as a process + 2 child of parent + 2 child of child 1 and 1 child of process 2 = 6 processes.

(b) 6 processes have 6 threads. Moreover, thread create(...) function is called twice.

So, Total thread = 6+2
= 8 threads.

Ans to the ques. no-03:

A single semaphore can be used to enforce mutual exclusion among n processes by ensuring that only one process enters critical section at a time. To explain, when the S of semaphore is restricted ~~to ine~~ between 0 to 1; which means it can increment upto 1 and decrement upto 0, this ensures only one process can enter its critical section at a given moment and all other processes will be block because of the value of S becomes $S \leq 0$.

Ans. to the ques. no-4:

To satisfy the bounded-waiting requirement while working with the compare and swap() instruction, a new variable needs to introduce to control the flow of bounded-waiting requirement:

```
while (compare_and_swap(&lock, 0, 1) != 0 && val != i) {
  }
  //critical section;
 lock = 0;
 val = val+1;
}
```

Here, val is tracking the posilion of and situation of those processes who are ready to enter critical section. Here, i is the waiting queue.

which ensures that the prior arrived processes enters the critical section, ~~handles~~ satisfying the bounary-waiting requirement.

Ans. to the ques. no - 05:

In this banking system, the amount is a shared data that is accessed by both withdraw() and deposit() function and so accessing both of these functions at a time causes the occurance of race condition. For example: Let's assume the account has a amount of 5000taka. Now, if the ~~withdraw~~ withdraw(200) and deposit(600) is called at a time, the withdraw() function reads the current amount 5000 and calculates the new current balance as 5000 - 200 = 4800taka.

As on the other hand, the function deposit() also reads the current amount 5000 and calculates the new current balance as 5000+600 = 5600taka

This occurance of race condition can be prevented by enforcing mutual exclusion so that only one function can access and modify the shared data (amount) at a time.