Worksheet 5.6: Assignment 2

Consider the following code that implements the multithreaded program of Assignment 2:

```
int gData[MAX_SIZE]; //Array that holds the data
int gThreadCount; // Number of threads
int gDoneThreadCount; // Number of threads that are done so far
int gThreadProd[MAX_THREADS]; // The product of each thread
sem_t sem_1, sem_2;
#define NUM_LIMIT 9973

int main(int argc, char *argv[]){
1     pthread_t tid[MAX_THREADS];
2     pthread_attr_t attr[MAX_THREADS];
3     int indices[MAX_THREADS][3], i, arraySize, prod;

// Code for parsing command-line arguments and generating input

4    CalculateIndices (arraySize, gThreadCount, indices);
5    InitSharedVars ();
6    sem_init (&sem_1, 0, X1);
7    sem_init (&sem_2, 0, X2);
8    for (i=0; i<gThreadCount; i++)
9            pthread_create(tid+i, attr+i, Child, indices[i]);

10   sem_wait(X3);
11   prod = ComputeTotalProduct();
12   for (i=0; i<gThreadCount; i++)
13           pthread_cancel(tid[i]);
14   printf ("Product is %d\n", prod);
}

void* Child (void *param) {
1            int threadNum = ((int*)param)[0];
2            int start = ((int*)param)[1];
3            int end = ((int*)param)[2];

4            gThreadProd[threadNum] = 1;
5            for (int i=start; i<=end; i++) {
6                if (gData[i] == 0) {
7                        gThreadProd[threadNum] = 0;
8                        gThreadDone[threadNum] = true;
9                        sem_post(&sem_2);
10                       pthread_exit(0);
                 }
11               gThreadProd[threadNum] *= gData[i];
12               gThreadProd[threadNum] %= NUM_LIMIT;
             }
13           gThreadDone[threadNum] = true;

14           sem_wait(&sem_1);
15           gDoneThreadCount++;
16           if (gDoneThreadCount == gThreadCount)
17                sem_post(&sem_2);
18           sem_post(&sem_1);

}
```

1. Replace X1, X2, X3 in the above code with the right values or expressions.

   X1: 1

   X2: 0

   X3: &sem_2

2. Trace the consequences of changing sem_2 on Line 9 of the child to sem_1. Clearly indicate if the program will work correctly or not.

   If there is a zero in the input, the parent will never be notified and the program will not terminate. Multiple children could be updating shared variable gDoneThreadCount at the same time, which means that the shared variable may have an incorrect value.

3. Suppose that you have two separate programs for solving the above problem. One program does sequential search without dividing the array and another program divides the array into equal divisions and creates a thread to process each division **as in the above code**. The sequential program terminates as soon as a zero is found (if there is a zero).
   The search part of the sequential program takes 80% of the program's execution time if the input array has no zero. The other 20% is spent in parsing the command-line arguments and generating the input data, which is something that both programs do sequentially. Given that the input array is divided into 8 divisions, compute the speedup achieved by the multithreaded program relative to the sequential program in each of the two cases below. Assume **ideal parallelism** on a system with at least 9 cores. Ignore all kinds of overhead, and assume that all threads start immediately.

   First Case: The parent waits for all children (no semaphores used), and there is **no zero** in the input.

   Speedup = sequential time / parallel time = 1 / (0.2 + 0.8/8) = 1 / (0.2 + 0.1) = 10 / 3

   Second Case: Semaphores are used as in the above code, and there is a zero right at the beginning of the 8th division.

   Speedup = sequential time / parallel time = (0.2 + 0.8 x 7/8) / (0.2 + 0) = 0.9 / 0.2 = 4.5

4. If we replace Lines 8-13 in the Parent with the following:

```
for(i=0; i<gThreadCount; i++)
{
    pthread_create(tid+i, attr+i, Child, indices[i]);
    pthread_join(tid+i);
}
prod = ComputeTotalProduct();
```

   1. Will the program still compute the product correctly? [✓ YES] NO

   2. Assuming that there are no zeros in the input, will the execution time in this case be the same as Method 1, Method 2, Method 3 or as the sequential algorithm? First circle the right answer, and then Justify your answer.

      [✓ **Sequential: Don't divide the array and just scan it sequentially**]

      Method 1: Parent waits for all children

      Method 2: Parent checks on children in a busy-waiting loop

      Method 3: Parent waits on a semaphore as in the above code

      Justification: **(Limit: 2 Lines)**

      The parent launches one thread at a time and waits for it to complete before launching the next thread. This is equivalent to sequential processing of the divisions.

5. Modify the above code to solve the following search problem. Given an input array and a key, divide the array into K equal divisions and check if the key appears **exactly once** in each division. The program creates a thread to search each division as in the above code (gThreadCount = K). If the key appears once in **each** division, the program prints "YES"; otherwise (there is at least one division in which the key either does not appear or appears more than once), it prints "NO". Assume that the key is already stored in a global variable named gKey.
   Your program must meet the following specs:
   - There is a global array gKeyOccur of size K that holds the number of occurrences of the key in each division.
   - Each child searches its division and updates its entry in gKeyOccur. A child notifies the parent as soon as it either completes searching the entire division or determines that there is no need to search any further (the child must avoid doing unnecessary search in its division).
   - Each child must update gDoneThreadCount when it is done.
   - It is the parent's job, not the child's job to determine if the overall outcome is positive or negative and if all threads are done (unlike the above code). Each child is responsible only for the outcome of its own division.
   - The parent must print the result **as soon as it is known** and then cancel all threads and terminate cleanly. You will lose points for any delay in printing or termination.
   - The parent cannot waste CPU cycles while the child threads are searching. It must be in the waiting state until it receives a notification from a child. After processing that child's result, the parent must go into the waiting state again, waiting for the next child's notification, and so on.
   - If the hardware has x cores and x<k, only x threads can be active concurrently and the remaining k-x threads must be in the waiting state until one of the x threads is done. The point is minimizing context-switching overhead.

   Use the minimum number of shared variables and semaphores and minimize the time spent in critical sections. Give the initial value for every shared variable or semaphore that you use.
   You can reuse any lines from the given code by simply specifying their numbers; you don't have to rewrite any unchanged code.

```
sem notify=0, mutex=1, limit=X

void * Child(void* param) {
    wait (limit)

    int threadNum = ((int*)param)[0];
    int start = ((int*)param)[1];
    int end = ((int*)param)[2];

    for(int i=start; i<=end; i++) {
            if (gData[i] == gKey) {
                    gKeyOcurr[threadNum]++
                    if (gKeyOcurr[threadNum] > 1) {
                            gThreadDone[threadNum] = true;
                            post(&limit);
                            post(&notify);
                            pthread_exit(0)
                    }
            }
    }
    gThreadDone[threadNum] = true;

    sem_wait(&mutex);
    gDoneThreadCount++;
    sem_post(&mutex);
    sem_post(&limit);
    sem_post(&notify);
}
```

Parent

```
    Lines 1-9

    done = false
    while(!done) {
            wait(&notify);
            wait(&limit);
            res = CheckOccur();
            if (result == false || gDoneThreadCount == gThreadCount)
                    done = true;
            post(&limit);
    }
    Print result

    CheckOccur() {
            for (i =0; i< gThreadCnt; i++)
                    if (gKeyOccurr[i] > 1 || gThreadDone [i] == true && gKeyOccur[i] == 0)
                            return false;
            return true;
    }
```