Assignment 3 Review

The following code uses two threads to find a digit in a range of numbers. Write and check the running time. Check your output with your peer. Discuss with your peer to make it more efficient.

```cpp
#include <iostream>
#include <thread>

using namespace std;

int flag = 1;
long range = 10000000000;

int search(long first, long end, long x) {
    for (long i=first; i<=end; i++){
        if (i==x){
            cout<<"Find it"<<endl;
            return(i);
        }
    }
}

int main(int argc, char const *argv[]) {
    long first, end;
    long x = 10;
    thread t1(search, 1, (range/2), x);
    thread t2(search, (range/2), range, x);

    t1.join();
    t2.join();

    return 0;
}
```

Synchronization Definitions

**Race Condition**
A race condition occurs when processes have concurrent access to shared data and the final result depends on the particular order in which concurrent accesses occur. Race conditions can result in corrupted values of shared data.

**Critical Section**
A critical section is a section of code where shared data may be manipulated and a possible race condition may occur. The critical-section problem is to design a protocol whereby processes can synchronize their activity to cooperatively share data.

**Solution Requirement**
A solution to the critical section problem must satisfy the following three requirements: (1) mutual exclusion, (2) progress, and (3) bounded waiting. Mutual exclusion ensures that only one process at a time is active in its critical section. Progress ensures that programs will cooperatively determine what process will next enter its critical section. Bounded waiting limits how much time a program will wait before it can enter its critical section.

**Software Solutions**
Software solutions to the critical-section problem, such as Peterson’s solution, do not work well on modern computer architectures.

**Hardware Solutions**
Hardware support for the critical-section problem includes memory barriers; hardware instructions, such as the compare-and-swap instruction; and atomic variables.

**Mutex**
A mutex lock provides mutual exclusion by requiring that a process acquire a lock before entering a critical section and release the lock on exiting the critical section.

**Semaphores**
Semaphores, like mutex locks, can be used to provide mutual exclusion. However, whereas a mutex lock has a binary value that indicates if the lock is available or not, a semaphore has an integer value and can, therefore, be used to solve a variety of synchronization problems.

**Monitor**
A monitor is an abstract data type that provides a high-level form of process synchronization. A monitor uses condition variables that allow processes to wait for certain conditions to become true and to signal one another when conditions have been set to true.

**Drawback**
Solutions to the critical-section problem may suffer from liveness problems, including deadlock.

**Semaphores**
A semaphore $S$ is an integer variable that, apart from initialization, is accessed only through two standard atomic operations: `wait()` and `signal()`. The `wait()` operation was originally termed P (from the Dutch *proberen*, “to test”); `signal()` was originally called V (from *verhogen*, “to increment”).

The definition of `wait()` is as follows:
wait(S) {
    while (S <= 0) ;
    // busy wait
    S--;
}

The definition of signal() is as follows:
signal(S) {
    S++;
}

All modifications to the integer value of the semaphore in the wait() and signal() operations must be executed atomically. That is, when one process modifies the semaphore value, no other process can simultaneously modify that same semaphore value. In addition, in the case of wait(S), the testing of the integer value of S (S ≤ 0), as well as its possible modification (S--), must be executed without interruption.

Semaphore Usage

The binary semaphores behave similarly to mutex locks. In fact, on systems that do not provide mutex locks, binary semaphores can be used instead of providing mutual exclusion.

The counting semaphores can be used to control access to a given resource consisting of a finite number of instances. The semaphore is initialized to the number of resources available. Each process that wishes to use a resource performs a wait() operation on the semaphore (thereby decrementing the count). When a process releases a resource, it performs a signal() operation (incrementing the count). When the count for the semaphore goes to 0, all resources are being used. After that, processes that wish to use a resource will block until the count becomes greater than 0.

We can also use semaphores to solve various synchronization problems. For example, consider two concurrently running processes: P1 with a statement S1 and P2 with a statement S2. Suppose we require that S2 be executed only after S1 has completed. We can implement this scheme readily by letting P1 and P2 share a common semaphore synch, initialized to 0. In process P1, we insert the statements

\[ S1; \]
\[ \text{signal(synch);} \]

In process P2, we insert the statements

\[ \text{wait(synch);} \]
\[ S2; \]

Because synch is initialized to 0, P2 will execute S2 only after P1 has invoked signal(synch), which is after statement S1 has been executed.

Example 1 - Using a semaphore to synchronize 15 threads

In this example, we use condition_variable class. A condition variable is an object able to block the calling thread until notified to resume. It uses a unique_lock (over a mutex) to lock
the thread when one of its wait functions is called. The thread remains blocked until woken
up by another thread that calls a notification function on the same condition_variable object.

```cpp
#include <iostream>
#include <thread>
#include <mutex>
#include <condition_variable>

using namespace std;

mutex mtx;            // mutex for critical section
condition_variable cv; // condition variable for critical section
bool ready = false;       // Tell threads to run
int current = 0;       // current count

/* Prints the thread id / max number of threads */
void print_num(int num, int max) {
    unique_lock<mutex> lck(mtx);
    while(num != current || !ready){
        cv.wait(lck);
    }
    current++;
    cout << "Thread: ";
    cout << num + 1 << " / " << max;
    cout << " current count is: ";
    cout << current << endl;

    /* Notify next threads to check if it is their turn */
    cv.notify_all();
}

/* Changes ready to true, and begins the threads printing */
void run(){
    unique_lock<mutex> lck(mtx);
    ready = true;
    cv.notify_all();
}

int main (){

    int threadnum = 15;
    thread threads[15];

    /* spawn threadnum threads */
    for (int id = 0; id < threadnum; id++)
```
threads[id] = thread(print_num, id, threadnum);

cout << "\nRunning " << threadnum << " in parallel: \n" << endl;

run(); // Allows threads to run

/* Merge all threads to the main thread */
for(int id = 0; id < threadnum; id++)
    threads[id].join();

cout << "\nCompleted semaphore example!\n";
cout << endl;

return 0;
}