ECE 459 W20 Midterm Solutions

J. Zarnett

March 3, 2020

(1)

(a) No. Single threaded means no parallelism, but concurrency is still possible. An asynchronous operation, like handling a signal, can still result in a race condition.

(b) On this one you could argue either yes or no.

Yes: if we can guess what the user might do with high accuracy then most of the time it will be correct but there will be some latency when we are wrong. You can also run several options in parallel and only show the one the user actually chose.

No: unless we are absolutely perfect (which is impossible) in guessing what the user will do, there will be latency spikes that will make the users upset. You could also argue that there are too many possibilities to make good guesses in the first place. You can also argue that most games don't support being able to try many actions concurrently or roll back state so precisely.

(c) Yes. If CPU4 has `x` in its cache exclusively, CPU2 can write to that location with a statement like `x = 0`. CPU2 does not need to read `x` to execute this statement so it can just issue the write even though another CPU held it exclusively before that.

(d) Any answer between 10 000 and 100 000 is correct here.

(e) Helgrind might report a lock ordering problem here: if money is sent from account A to account B the first time, that's viewed as "correct". If at some later time, money is transferred from B to A, that will be reported as a lock ordering problem by helgrind. Fix by: using a consistent lock ordering, such as based on account numbers.

(2)

```
typedef bool (*test_fn)( );
int failed_tests = 0;
int next;
int total_tests = 0;
pthread_mutex_t l1;
pthread_mutex_t l2;

void init( )  {
    pthread_mutex_init( &l1, NULL );
    pthread_mutex_init( &l2, NULL );
    next = 0;
}

void* executor( void* arg ) {
    test_fn* tests = (test_fn*) arg;
    bool done = false;
    int n;
    while( !done ) {
      pthread_mutex_lock( &l1 );
      n = next;
      next++;
      pthread_mutex_unlock( &l1 );

      if ( n >= total_tests ) {
        return; /* Or pthread_exit */
      }

      test_fn f = tests[n];
      bool success = f();
      if ( !success ) {
        pthread_mutex_lock( &l2 );
        failed_tests++;
        pthread_mutex_unlock( &l2 );
      }
    }
}
```

```
void cleanup() {
    pthread_mutex_destroy( &l1 );
    pthread_mutex_destroy( &l2 );
}

int run_tests_parallel( test_fn * tests, int
    num_tests, int threads ) {
    init();
    total_tests = num_tests;
    pthread_t t[threads];

    for ( int i = 0; i < threads; i++ ) {
        pthread_create( &t[i], NULL, executor,
            tests );
    }
    for ( int j = 0; j < threads; j++ ) {
        pthread_join( t[j], NULL );
    }
    cleanup();
    return failed_tests;
}
```

You have LOTS of options about how to complete this; you can use atomics to modify the shared variables instead; you can use a semaphore as a counter; you can use locks around shared variables; whatever you like. The atomics would be better! You could also combine locks `l1` and `l2`, but that's not as good of a solution.

However, it is not okay to divide the array up and have each thread do a subsection of the array. That isn't the thread pool pattern that the question asked for.

(3)

```
double gr(int n) {
    if (n >= 2) {
        double f, g;

        #pragma omp task default(none) shared(f, n)
        f = gr(n-1);

        #pragma omp task default(none) shared(g, n)
        g = gr(n-1);

        #pragma omp taskwait
        return ((f+5)/g)/2;
    }
    else {
        return 2.0;
    }
}

int main(int argc, char *argv[])
{
    if (argc != 2) {
        printf("%s:␣requires␣an␣argument␣>␣10\n", argv[0]);
        return EXIT_FAILURE;
    }
    int n = atoi(argv[1]);
    if (n <= 10) {
        printf("%s:␣requires␣an␣argument␣>␣10\n", argv[0]);
        return EXIT_FAILURE;
    }
    #pragma omp parallel
    {
        #pragma omp single
        {
            printf("gr(%d)␣=␣%lf\n", n, gr(n));
        }
    }
    return EXIT_SUCCESS;
}
```

(4)

The `mfence` instruction prevents reorderings that would defeat the purpose of the critical section. The one in `lock()` prevents statements inside the section from being moved before the lock is acquired and the one in `unlock()` prevents statements from being moved out. To give a specific example, the statement `x = 0` that is inside the critical section could moved immediately after the unlock statement without the fence.

(5) There are 1000 requests in groups of 10 (= 100 groups), 1 out of every 10 will be slow; 100 slow requests.

Part 1:

1. Worst case scenario: every group contains one slow request. Therefore the total time is $100 \times 0.5s = 50s$ to complete.

2. Best case scenario: all the slow requests are packed into as few groups as possible. There are 100 slow requests so that's 10 groups that take 500 ms and the remaining 90 groups take 50 ms. So $10 \times 0.5s + 90 \times 0.05s = 5s + 4.5s = 9.5s$

Part2, after the change:

3. Consider this scenario as explained by a student Shuhao Sun (thanks for the correction!) showing 9950 ms to complete:

- 800 fast puzzles = 80 batches of 10 fast puzzles = 4000ms
- 80 slow puzzles = 8 batches of 10 slow puzzles = 4000ms
- 9 slow puzzles + 10 fast puzzles = 500ms now, we have 90 fast puzzles left + 11 slow puzzles left
- 90 fast puzzles = 9 batches of 10 fast puzzles = 450ms
- 10 slow puzzles = 1 batch of 10 slow puzzles = 500ms
- 1 slow puzzle = 1 batch of 1 slow puzzle = 500ms

4. The best case scenario time doesn't change under this scenario; it's still best to have as many slow requests as possible running concurrently so the calculation is the same.

Part 3: the average case after the change is $(9.95 + 9.5)/2$; the average for original strategy is $(50 + 9.5)/2 = 29.75$. Expected speedup is $29.75/9.725 = 3.06$.