Lecture 15 - Reentrant/Thread-Safe and Assignments

ECE 459: Programming for Performance

Jon Eyolfson

University of Waterloo

February 6, 2012

Previous Lecture

Difference between reentrant and thread-safe functions

Reentrancy

- Has nothing to do with threads, it assumes a **single thread**
- Reentrant means the execution can context switch at any point in in a function, call the **same function** and **complete** before returning to the original function call
- Result does not depend on where the context switch happens

Thread-safety

- Result does not depend on any interleaving of threads from concurrency or parallelism
- In other words, you do not get unexpected results

Another Definition of Thread-Safe Functions

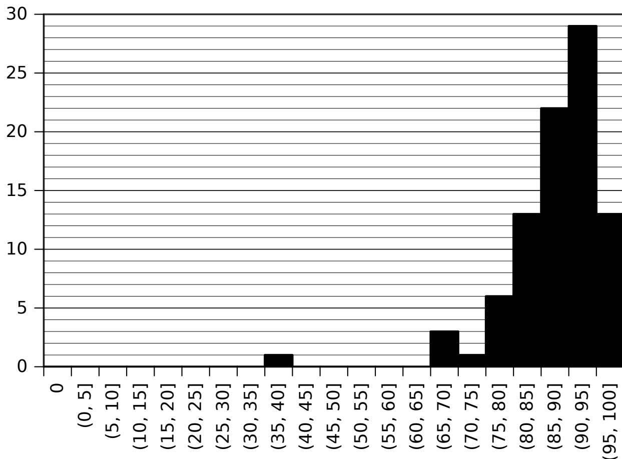
A function whose effect, when called by two or more threads, is guaranteed to be as if the threads each executed the function one after another in an undefined order, even if the actual execution is interleaved

Good Example of a Midterm Question

```
void swap(int *x, int *y) {
    int t;
    t = *x;
    *x = *y;
    *y = t;
}
```

- Is the following code thread-safe?
- Write some expected results for running two calls in parallel
- Argue these expected results always hold, or show an example where they do not

Grade Distribution



Average: ~89

Common Mistakes (1)

Casting unsigned long to void*

- Okay if you put down in comments you are assuming 64-bit
- Generally, this is unsafe, and should be avoided

Didn't handle uneven job distribution

- What if `iterations` does not divide evenly by the number of threads?
- Okay if you stated in comments that this will not affect the results much
- You should, however, have considered it

Common Mistakes (2)

Failed to destroy the mutex

- You need to call `pthread_mutex_destroy` to free the resources associated with the mutex

Failed to free memory

- Okay if you stated in comments that this is cleaned up in Linux
- Again, you should, however, have considered it
- Write down in the comments who is responsible for freeing memory
- Use `valgrind` to help detect memory leaks

Solution with a Mutex (1)

```
static pthread_mutex_t mutex = PTHREAD_MUTEX_INITIALIZER;

void* run(void* arg)
{
    unsigned long int private_count =
        montecarlo(iterations/num_threads);
    pthread_mutex_lock(&mutex);
    count += private_count;
    pthread_mutex_unlock(&mutex);
    return NULL;
}
```

Solution with a Mutex (2)

```
pthread_t* threads =
    malloc(sizeof(pthread_t)*num_threads);
int i;
for (i = 0; i < num_threads; ++i) {
    pthread_create(&threads[i], NULL, &run, NULL);
}
for (i = 0; i < num_threads; ++i) {
    pthread_join(threads[i], NULL);
}
free(threads);
pthread_mutex_destroy(&mutex);

// Do the reminding number of iterations we couldn't
// evenly divide into threads
count += montecarlo(iterations % num_threads);
```

Solution with a Return Value (1)

```
void* run(void* arg)
{
    // The joining thread is responsible for freeing this
    // memory
    unsigned long int* ret =
        malloc(sizeof(unsigned long int));
    *ret = montecarlo(iterations/num_threads);
    return (void*) ret;
}
```

Solution with a Return Value (2)

```
// I'm assuming i > t, so we don't have to worry about
// that type of job allocation
pthread_t* threads =
    malloc(sizeof(pthread_t)*(num_threads - 1));
int i;
for (i = 0; i < (num_threads - 1); ++i) {
    pthread_create(&threads[i], NULL, &run, NULL);
}
// Do this thread's share and reminding number of
// iterations we couldn't evenly divide into threads
count += montecarlo(iterations/num_threads
                        + iterations % num_threads);

for (i = 0; i < (num_threads - 1); ++i) {
    void* ret;
    pthread_join(threads[i], &ret);
    count += *((unsigned long int*)ret);
    // Free the memory from the thread
    free(ret);
}
free(threads);
```

Benchmarking - Sequential and Parallel Physical Cores

- Make sure your results are consistent (nothing else is running)
- Follow the 10 second guideline (60 second runs are no fun)
- Since we are assuming 100% parallel, the runtime should decrease by a factor of `physicalcores`
- Results should be close to predicted, therefore our assumption holds (could estimate P in Amdahl's law and find it's 0.99)
- Overhead of threading (create, joining, mutex?) is insignificant for this program

Benchmarking - Parallel Virtual CPUs vs Virtual CPUs + 1

- Hyperthreading results were weird, slower the majority of the time
- Table 4 should be slower than Table 3 (or Table 3)
- It's better to have a number of threads that match the number of virtual CPUs then an unbalanced number
- Difference because, if it's uneven, one thread will constantly be context switching between virtual CPUs
- Worse case for 9 threads on 8 virtual CPUs: 8 threads complete, each doing a ninth of the work in parallel, last ninth of the work runs only on one CPU

Assignment 2

- Go over the assignment
- Identify anything that may be a problem
- Take away points
  - You are much better at parallelizing code than your compiler
  - OpenMP fine-grained task parallelism isn't that great

Summary

- Clearer explanation of **reentrant** and **thread-safe** (hopefully)
- Comfortable programming Pthreads / preventing data races
- Always remember to free resources (the earlier the better)
- Usually the best number of threads is the same as the number of physical cores or virtual CPUs (unbalanced threads are bad)