COMPSCI 630 Systems **Spring 2017**

Lecture 8: Memory Allocation

Lecturer: Emery Berger *Scribe(s): Rahul Handa,Peter Jay*

8.1 Discussion about Determinism and Non Determinism

We start by continuing our discussion about DThreads from the paper reviewed last week. (At this point prof.Berger found an innovative use for the old speakers). We discussed what exactly is meant by determinism, the idea that a program repeatedly run with the same inputs will produce the same results. This is in contrast to non-determinism which produces different results on subsequent executions. Sources of non-determinism include hardware conditions and variations in input such as GUI events.

8.1.1 Threads

Threads and thread scheduling are another source of non determinism. (insert the single and multicore thread diagram here, if we want to). A sub discussion lead to the two kinds of bugs that a program faces :

- HeisenBug (non-deterministic) - A bug that may or may not surface with every execution; these are harder to debug but users like them because they may ?go away?.
- Bohr Bug (deterministic) - A bug that occurs with every execution; programmers like these bugs because they are easy to debug however users hate them because they do not resolve themselves.

8.1.2 Single core v/s Multi Core

The execution of threads on both architectures, Single core is a nice world where execution of threads is kind of deterministic and threads are interrupted only by quantum expiration or explicit yield where as in a multicore environment thread scheduling faces problems like interruption by other cores/self, cache(Different addresses for every core) and which thread reaches the required resource first.

8.2 Mutex and Race example

We study a couple of programs to understand how race condition affects determinism and how we can use locks(mutual execution) to counter that.

```
Const int c =1000;
Int x[c] ;
//multiple threads
for (i=0;i<1000000;i++){
      for(j=0;j<1000;j++){
x[j]++;  }
}
```

We never get the same result twice in C++ for this program. Hence locks are used to make such a program deterministic. Use something like :

```
ptmutex.acquire();
 for(j=0;j<1000;j++){
x[j]++;  }
}
ptmutex.release();
```

This executes one thread at a time.But another issue brought up by this approach is the semantic impact , i.e., output depends on which lock acquires the lock first. Monitors and semaphores are used to deal with this. (Professor uses the dirty banking transaction example to explain this , which he very much hates.)

8.3 Detecting a Race

There are static and dynamic race detectors , major difference is the former being at compile time while latter is at runtime and it causes 10x-400x slowdown of a program.A controversial question come up: **Do race conditions come under bugs?**

8.4 How DThreads enforce determinism?

Dthreads are used to force execution to behave in a deterministic way. It works by partitioning all threads into separate process with individual memory spaces. Normally threads share memory but dthreads operate in parallel and make memory access to their own private memory set. Eventually there is a synchronization phase where all dthreads update the main memory in a deterministic way. This eliminates non-deterministic output but may be costly as it trades space for predictability. Dthreads also eliminates false sharing which is when multiples thread wish to access variables that unfortunately share the same cache line. Once a thread gets a hold of one of the variables the CPU invalidates updates made to the other variables which can result in a massive performance decrease. Dthreads solve this issue because they operate in private address spaces.