CME213/ME339
Lecture 13

Eric Darve
Erich Elsen
Austin Gibbons

Department of Mechanical Engineering
Institute for Computational and Mathematical Engineering
Stanford University

Spring 2012

What is a thread?

Threads on a CPU are not the same on a GPU (CUDA)

- CUDA threads : 100,000s of threads each with 100s of instructions in overhead
- PThread threads: 100s of threads each with 100,000s of instructions in overhead

So why should you care?

CPU threads

- Versatile - created with the idea that different threads would perform independent tasks
- Hardware - million dollar clusters
- Software - PThreads, OpenMP, MPI, Map-Reduce are ubiquitously used

Pthreads - overview

- Problem: hardware vendors would create their own threading mechanisms.
- Solution: POSIX Threads: an IEEE standard which defines an API for the C programming language.
- They require much less overhead than a process.

Pthreads - Resources

Threads maintain their own:

- program control
- registers
- stack
- scheduling and synchronization information

and they share system resources such as the file system, and heap memory.

To use pthreads in your program:

```
#include <pthread.h>
g++ yourprogram.cc -lpthread
```

Pthreads - What does it look like?

```c
#include <stdio.h>
#include <pthread.h>

void *printID(void *input) {
    int id = *(int *) input;
    printf("Hello from thread %d!\n", id);
}

int main() {
    int numThreads = 8;
    pthread_t thread[numThreads];
    for(int i = 0; i < numThreads; i++) {
// launch a thread (pthread_t, attr, function, arguments)
        pthread_create(&thread[i], NULL, printID, (void *) &i);
    }
}
```

Quiz #1

What happens in the last slide? We wanted each thread to output its unique id from 0 to 7.

```
Hello from thread 2!
Hello from thread 5!
Hello from thread 2!
Hello from thread 4!
Hello from thread 6!
Hello from thread 7!
Hello from thread 8!
Hello from thread 8!
```

Quiz #1

```c
#include <stdio.h>
#include <pthread.h>

void *printID(void *input) {
    int id = *(int *) input;
    printf("Hello from thread %d!\n", id);
}

int main() {
    int numThreads = 8;
    pthread_t thread[numThreads];
    for(int i = 0; i < numThreads; i++) {
// launch a thread (pthread_t, attr, function, arguments)
        pthread_create(&thread[i], NULL, printID, (void *) &i);
    }
}
```

Pthreads - Dangers

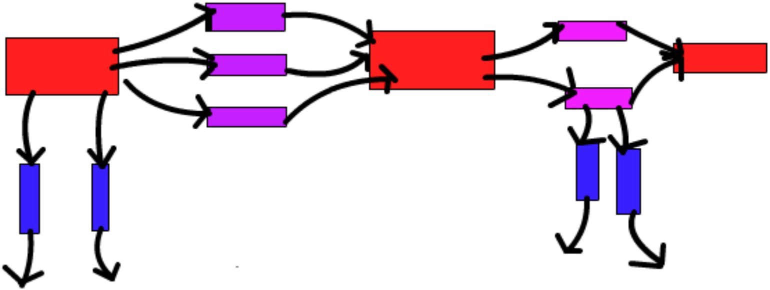
- Data Races, Synchronization, and Communication can be tricky to get right in complicated programs.
- Hard to detect data races - your program is often non-deterministic.
- valgrind, gdb, code serialization can still be useful.

Pthreads- Solutions

Pthreads supports a large suite of synchronization through its API:

- mutexes (locks)
- barriers
- conditional variables
- join / detach

Pthreads - create, join, detach



The main thread (red) *creates* purple and blue threads. It *detaches* the blue threads, so it does not need to monitor them. It *joins* the purple threads, so it must wait until all of them return to the main thread.

Pthreads - What does it look like?

```c
int main() {
    int numThreads = 8;
    pthread_t thread[numThreads];
    int *tD = malloc(numThreads*sizeof(int));
    for(int i = 0; i < numThreads; i++) {
      tD[i] = i;
      pthread_create(&thread[i], NULL, printID, (void *) &tD[i]);
    }

    for(int i = 0; i < numThreads; i++) {
//waits for a thread to return (pthread_t, return value)
        pthread_join(thread[i], NULL);
    }

    return 0;
}
```

Pthreads - What does it look like?

```
int main() {
    int numThreads = 8;
    pthread_t thread[numThreads];
    int *tD = malloc(numThreads*sizeof(int));
    for(int i = 0; i < numThreads; i++) {
      tD[i] = i;
      pthread_create(&thread[i], NULL, printID, (void *) &tD[i]);
    }

    for(int i = 0; i < numThreads; i++) {
//waits for a thread to return (pthread_t, return value)
        pthread_detach(thread[i]);
    }

    return 0;
}
```

Quiz #2

What happens in the last slide? We wanted each thread to output its unique id from 0 to 7.

```
Hello from thread 1!
Hello from thread 3!
Hello from thread 0!
```

Pthreads - What does it look like?

```
int main() {
    int numThreads = 8;
    pthread_t thread[numThreads];
    int *tD = malloc(numThreads*sizeof(int));
    for(int i = 0; i < numThreads; i++) {
      tD[i] = i;
      pthread_create(&thread[i], NULL, printID, (void *) &tD[i]);
    }

    for(int i = 0; i < numThreads; i++) {
//waits for a thread to return (pthread_t, return value)
        pthread_detach(thread[i]);
    }

    return 0;
}
```

There is also synchronization while threads are alive:

- Barriers are a place where all threads must arrive before continuing, like syncthreads() in CUDA
- Mutexes are locks. They allow atomic access to a region of code, called a critical section.
- Conditional variables are a way to ensure variables are what you expect. They also allow a way to send and receive messages.

Pthreads - Optimizations

Using synchronization in your program incurs a trade-off: synchronization serializes your code, but finer granularity requires more synchronization instructions and memory accesses.

Furthermore, implementing fine grained schemes, as is common with locks, is programmatically much more challenging than a naive implementation.

```c
pthread_mutex_t a, b;
int main() {
//initialize the mutexes
    pthread_mutex_init(&a, NULL);
    pthread_mutex_init(&b, NULL);

//create the threads...
}

void *foo(void *id) {
    pthread_mutex_lock(&a);
    pthread_mutex_lock(&b);
    // critical section
    pthread_mutex_unlock(&a);
    pthread_mutex_unlock(&b);
}

void *bar(void *id) {
    pthread_mutex_lock(&b);
    pthread_mutex_lock(&a);
    // critical section
    pthread_mutex_unlock(&b);
    pthread_mutex_unlock(&a);
}
```

Quiz 3

What can go wrong on the last slide?

```
pthread_mutex_t a, b;
int main() {
//initialize the mutexes
    pthread_mutex_init(&a, NULL);
    pthread_mutex_init(&b, NULL);

//create the threads...
}

void *foo(void *id) {
    pthread_mutex_lock(&a);
    pthread_mutex_lock(&b);
    // critical section
    pthread_mutex_unlock(&a);
    pthread_mutex_unlock(&b);
}

void *bar(void *id) {
    pthread_mutex_lock(&b);
    pthread_mutex_lock(&a);
    // critical section
    pthread_mutex_unlock(&b);
    pthread_mutex_unlock(&a);
}
```

Quiz 3

Deadlock — two different threads are waiting on a resource that the other thread holds.

In complicated systems, not uncommon, can be hard to detect (non-deterministic).

There are good deadlock detection, prevention, and avoidance tools available.

Pthreads - Analysis

The theoretical speedup is
$Time_{parallel} = Time_{serial} + Time_{parallelizable}/p$

where p is the number of processors.

(Amdahl's law)

With pthreads, there is a quick and dirty analysis:

`top` - watch your performance

austingibbons — ssh — 80×24

```
top - 02:10:56 up 2 days, 12:44, 15 users,  load average: 2.84, 2.09, 1.90
Tasks: 270 total,   1 running, 260 sleeping,   0 stopped,   9 zombie
Cpu(s): 96.9%us,  3.1%sy,  0.0%ni,  0.0%id,  0.0%wa,  0.0%hi,  0.0%si,  0.0%st
Mem:  33014000k total, 26702056k used,  6311944k free,   522716k buffers
Swap: 10485756k total,    24000k used, 10461756k free,  8355760k cached

  PID USER      PR  NI  VIRT  RES  SHR S %CPU %MEM    TIME+  COMMAND
19748 gibbons4  20   0 79720 1088  856 S  588  0.0   0:39.22 a.out
 1542 tenhoeve  20   0 2124m 1.1g  44m S  108  3.5  20:32.72 MATLAB
28907 lanemc    20   0 14.5g  13g  59m S  100 43.1  76:19.62 MATLAB
32196 smeylan   20   0 1100m 368m  63m S    2  1.1 817:35.54 MATLAB
   46 root      20   0     0    0    0 S    1  0.0   2:02.77 kworker/7:1
  573 root      39  19     0    0    0 S    1  0.0 132:13.38 kipmi0
   44 root      20   0     0    0    0 S    0  0.0   2:47.56 kworker/5:1
 1796 root      20   0 49600 9980  964 S    0  0.0   2:33.29 perl
 9408 bodwyer   20   0  267m  64m  22m S    0  0.2   1:13.82 Mathematica
18394 gibbons4  20   0 13380 1308  864 R    0  0.0   0:01.18 top
23871 jnam0712  20   0  303m  15m 9812 S    0  0.0   0:30.72 gnome-terminal
24021 jnam0712  20   0  478m  29m  15m S    0  0.1   0:08.39 emacs
26120 kimyi     20   0  479m  28m  15m S    0  0.1   0:33.74 emacs
    1 root      20   0 24272 2088 1260 S    0  0.0   0:13.41 init
    2 root      20   0     0    0    0 S    0  0.0   0:00.00 kthreadd
    3 root      20   0     0    0    0 S    0  0.0   0:44.78 ksoftirqd/0
    4 root      20   0     0    0    0 S    0  0.0   0:00.00 kworker/0:0
```

If you are getting close to 100% cpu utilization, more threads won't do anything for you, and you'll have to find your speedup elsewhere — buying new hardware, optimizing your code, augmenting your dataset.

(This is similarly useful for OpenMP)

Pthreads - Attributes

Thread attributes can specify properties about the threads (such as whether it is joinable) and how the thread interacts with the system (such as scheduling information).

Mutexes, barriers, conditional variables all have attributes specifying similar roles. These can be very useful if you have some knowledge about how they are being used.

General use does not require modifying these attributes. Often, the attributes are specified once at the beginning of a program and apply to the entire process (rather than specific threads).

Pthreads - Models

There is a large community that performs research in scheduling algorithms and thread control

- Producer-Consumer: some threads create work, some threads perform work
- Master-Slave: some threads administrative work, some threads perform work
- Thread-pool: all threads can perform jobs taken from a collective resource

Pthreads - Fairness

In addition to having units of task for threads to work on, you need to decide an order. For example:

- First-in-First-out: whomever most recently requested to run gets to run
- Round-robin: threads take turns working each receiving a specific time interval

This leads to the ideas of *fairness* and *starvation*. Both your theoretical and empirical scheduling algorithms should have some rule governing which threads get what *priority* in running.

Using these different models and scheduling algorithms depends on both your operating system and your implementation. There are many thread libraries using pthreads to implement these different schema.

Pthreads - Optimizations

The typical scenario for developing in Pthreads uses

- 4–128 threads
- A private local cache
- A shared cache
- Shared memory

Understanding the memory behavior can have a huge impact on your program. In particular, because data exists in multiple local caches, a *cache coherency* schema is needed to ensure that the data is accurate at all times.

While too large a topic to fully investigate, it is important to remember that *writing to the same memory location* can cause "conflicts" in your program, which will slow your program down.

PThreads - Optimizations

Pthreads specific optimizations require investigating a lot of topics:

- manual analysis of the synchronization (such as allowing race conditions in your code!)
- manually controlling the scheduling
- manually adjust the stack

PThreads - Takeaway

Soon we will discuss OpenMP.

Underneath the hood, OpenMP relies on Pthreads. When you learn about the different parameters, tunables, optimizations, and options in OpenMP, think about how threads are working internally.