CPSC 457 T03/T04

Week 4 Day 2

Xining Chen

Agenda

- pthread
- pthread mutex
- pthread condition variables
- semaphores
- pthread barrier
  - Re-cycling threads using barrier
- pthread spinlock
- Strange syntax
- Assignment 3

Thread → pthread

#include <pthread.h>

	thread	**pthread**
create	thread t1 = thread(func);	pthread_t t1; pthread_create(&t1, NULL, func, NULL);
join	t1.join()	pthread_join(t1, NULL);
join w/ return value		void *ret_join; pthread_join(t1, &ret_join);
exit		pthread_exit(0);

Thread → pthread

<thread>

```
thread t1 = thread(func, param1, param2, ...);
```

* pack input parameters into a structure.

```
struct Param {
    long param1;
    long param2;
    int id;
};
```

Param p;

<pthread.h>

```
pthread_t threadPool[32];

pthread_create(&threadPool[i], NULL, func, p);
```

Mutex → pthread_mutex

#include <pthread.h>

	mutex	**pthread_mutex**
declare	mutex myMutex;	pthread_mutex_t myMutex;
initialize		pthread_mutex_init(&myMutex,0);
lock	myMutex.lock();	pthread_mutex_lock(&myMutex);
unlock	myMutex.unlock();	pthread_mutex_unlock(&myMutex);
destroy		pthread_mutex_destroy(&myMutex);

Demo 1: syncFind2.cpp
Demo 2: threadRead.cpp

Condition Variable → pthread_cond

#include <pthread.h>

	condition variable	pthread condition variable
declare	condition_variable cv;	pthread_cond_t cv;
create		pthread_cond_init(&cv, NULL);
wait	unique_lock <mutex> lck(myMutex); cv.wait(lck);	pthread_cond_wait(&cv, &myMutex); * after returning, mutex is automatically re-acquired. **The condition must be rechecked.
signal 1	cv.notify_one();	pthread_cond_signal(&cv); * must be followed by pthread_mutex_unlock() if blocked thread uses the same mutex
signal 2	cv.notify_all();	
destroy		

Demo: condExp1.cpp

Semaphores

#include <semaphore.h>

	semaphores
instantiate	sem_t mySem;
initialize private semaphore	sem_init(&mySem, **0**, value); * value represents the initial value of the semaphore. Should be some integer value.
initialize shared semaphore	sem_init(&mySem, **1**, value);
increment by 1	sem_post(&mySem);
decrement by 1	sem_wait(&mySem); * if mySem value is 0, then calling thread gets blocked. * if mySem value > 0, then decrement value by 1
non-blocking decrement	sem_try_wait(&mySem);
destroy	sem_destroy(&mySem);

Demo: semaphore1.cpp, semaphore2.cpp

pthread barrier

	pthread barrier
instantiate	pthread_barrier_t myBarrier;
initialize	pthread_barrier_init(&myBarrier, NULL, N_THREADS);
barrier wait	pthread_barrier_wait(&myBarrier);
destroy	pthread_barrier_destroy(&myBarrier);

Demo 1: barrierExp1.cpp
Demo 2 (Pavol's lecture): https://repl.it/@pfederl/barrier-fork-join-with-pthreadbarrier

Reusing threads w/ pthread barrier

Demo: barrierExp2.cpp

Reusing threads with mutexes

- https://repl.it/@emmynex2007/Thread-Reuse

- Source: Emmanuel Onu

pthread spinlock

- A spin lock pools its lock condition repeatedly until that condition becomes true.
- Used when the expected wait time for a lock is small.
- Main difference vs mutex: a thread waiting to acquire a spin lock will keep trying to acquire the lock without sleeping and will consume processor resources until it finally acquires the lock.

pthread spin lock

	pthread spin lock
Instantiate	pthread_spinlock_t slock;
Initialize a **not shareable** spin lock	pthread_spin_init(&slock, PTHREAD_PROCESS_PRIVATE);
Initialize a **shareable** spin lock	pthread_spin_init(&slock, PTHREAD_PROCESS_SHARED);
lock	pthread_spin_lock(&slock)
unlock	pthread_spin_unlock(&slock)
destroy	pthread_spin_destroy(&slock)

Demo: spinlock.cpp

Strange syntax

- void pointers
  - syntax: void *

Assignment 3

- Parallelizing getSmallestDivisor()

```
int64_t getSmallestDivisor(int64_t n)
{
  if( n <= 3) return 0; // 2 and 3 are primes
  if( n % 2 == 0) return 2; // handle multiples of 2
  if( n % 3 == 0) return 3; // handle multiples of 3
  int64_t i = 5;
  int64_t max = sqrt(n);
  while( i <= max) {
    if (n % i == 0) return i;
    if (n % (i+2) == 0) return i + 2;
    i += 6;
  }
  return 0;
}
```

Assignment 3

- Parallelizing getSmallestDivisor()

```
int64_t getSmallestDivisor(int64_t n)
{
  if( n <= 3) return 0; // 2 and 3 are primes
  if( n % 2 == 0) return 2; // handle multiples of 2
  if( n % 3 == 0) return 3; // handle multiples of 3
  int64_t i = 5;
  int64_t max = sqrt(n);
  while( i <= max) {
    if (n % i == 0) return i;
    if (n % (i+2) == 0) return i + 2;
    i += 6;
  }
  return 0;
}
```

Assignment 3

- Parallelizing getSmallestDivisor()

```
int64_t getSmallestDivisor(int64_t n)
{
  if( n <= 3) return 0; // 2 and 3 are primes
  if( n % 2 == 0) return 2; // handle multiples of 2
  if( n % 3 == 0) return 3; // handle multiples of 3
  int64_t i = 5;
  int64_t max = sqrt(n);
  while( i <= max) {
    if (n % i == 0) return i;
    if (n % (i+2) == 0) return i + 2;
    i += 6;
  }
  return 0;
}
```

Option 1:
1) Calculate the start interval for each thread normally based on even distribution.
2) Calculate the end interval of each thread by continuously incrementing the start by 6 until it is equal or greater than the next start.
3) If the end interval is greater than the next start, subtract 6.

https://repl.it/@emmynex2007/A3-interval-Calc

Assignment 3

- Parallelizing getSmallestDivisor()

```
int64_t getSmallestDivisor(int64_t n)
{
  if( n <= 3) return 0; // 2 and 3 are primes
  if( n % 2 == 0) return 2; // handle multiples of 2
  if( n % 3 == 0) return 3; // handle multiples of 3
  int64_t i = 5;
  int64_t max = sqrt(n);
  while( i <= max) {
    if (n % i == 0) return i;
    if (n % (i+2) == 0) return i + 2;
    i += 6;
  }
  return 0;
}
```

Option 2: use correct formula to find lower and upper bounds. Handle first and last thread individually.

i = 5 + m*6

What is m?

Assignment 3

- Parallelizing getSmallestDivisor()

```
int64_t getSmallestDivisor(int64_t n)
{
  if( n <= 3) return 0; // 2 and 3 are primes
  if( n % 2 == 0) return 2; // handle multiples of 2
  if( n % 3 == 0) return 3; // handle multiples of 3
  int64_t i = 5;
  int64_t max = sqrt(n);
  while( i <= max) {
    if (n % i == 0) return i;
    if (n % (i+2) == 0) return i + 2;
    i += 6;
  }
  return 0;
}
```

Option 2: use correct formula to find lower and upper bounds. Handle first and last thread individually.

i = 5 + m*6

What is m?

$$m = \frac{\sqrt{n}}{6 * nThreads}$$

Use thread id to determine each thread's lower and upper bounds.

Assignment 3

- Things to consider:
  - Only use multi-thread when the number is “large”

Demo – my results