CSE 333 Section 9 - pthreads

Welcome back to section! We're glad that you're here :)

Process

- A process has a virtual address space. Each process is started with a single thread, but can create additional threads.

Threads

- A thread contains a sequential execution of a program.
- Contained within a process.
- Threads of the same process share a memory/address space: see the same heap and globals, but each thread has its own stack.

POSIX threads (Pthreads)

- The POSIX standard provides APIs for dealing with threads.
- Part of the standard C/C++ libraries, declared in `pthread.h`.
- **Compile and link with** `-pthread`.
- Core pthread functions: `pthread_create, pthread_exit, pthread_join`
- ```
  int pthread_create( pthread_t *thread,
                      const pthread_attr_t *attr,
                      void *(*start_routine) (void *),
                      void *arg);
  ```
  `thread:` output parameter.
  `attr:` used to set thread attributes. Use `NULL` as default.
  `start_routine:` function pointer to a C routine that the thread will execute once it is created.
  `arg:` a single argument that may be passed to `start_routine`. `NULL` may be used if no argument is to be passed.
  Overall, it creates a new thread and calls `start_routine` with `arg` as its parameter. Returns 0 if successful; otherwise, returns an error number.
- `int pthread_join( pthread_t thread, void **retval);`
  Synchronization between threads. It waits for the thread specified by `thread` to terminate. If that thread has already terminated, then it returns immediately. If `retval` is non-`NULL`, then `retval` acts an output parameter and the address passed to `pthread_exit` by the finished thread is stored in it. For this course we can just set `retval` to `NULL`. It returns 0 if successful; otherwise, returns an error number.
- `void pthread_exit(void *retval);`
  It terminates the calling thread and allows the user to specify an optional termination status parameter, `retval`. For this course we can just set `retval` to `NULL`.

Mutex

- Protect shared data from being simultaneously accessed by multiple threads.
- `pthread_mutex_init, pthread_mutex_lock, pthread_mutex_unlock, pthread_mutex_destroy`
  ```

- `int pthread_mutex_init(pthread_mutex_t *mutex, const pthread_mutexattr_t *attr);`
  `mutex:` initializes the mutex referenced by `mutex`.
  `attr:` use `NULL` for the default values.
- `int pthread_mutex_destroy(pthread_mutex_t *mutex);`
  Destroys the mutex object referenced by `mutex`.
- `int pthread_mutex_lock(pthread_mutex_t *mutex);`
  `int pthread_mutex_unlock(pthread_mutex_t *mutex);`
  Use these to let a single thread access/modify shared data while blocking the other threads.

Pthread Example

```
// pthread example. This program dispatches some threads to run
// thread_main. thread_main does two separate things: (1) casts the
// argument to an int and prints it out; (2) updates sum_total.

#include <pthread.h>
#include <iostream>

using std::cout;
using std::cerr;
using std::endl;

const int NUM_THREADS = 50;
const int LOOP_NUM = 10000;

static int sum_total = 0;
static pthread_mutex_t sum_lock;

void *thread_main(void *arg) {
  int *num = reinterpret_cast<int*>(arg);
  cout << "[cthread: " << *num << "] " << endl;

  for (int i = 0; i < LOOP_NUM; i++) {
    pthread_mutex_lock(&sum_lock);
    sum_total++;
    pthread_mutex_unlock(&sum_lock);
  }

  delete num;
  return NULL;
}
```

```
int main(int argc, char** argv) {
  pthread_t thds[NUM_THREADS];
  pthread_mutex_init(&sum_lock, NULL);

  for (int i = 0; i < NUM_THREADS; i++) {
    int *num = new int(i);
    if (pthread_create(&thds[i], NULL, &thread_main, num) != 0) {
      /*report error*/
    }
  }

  for (int i = 0; i < NUM_THREADS; i++) {
    if (pthread_join(thds[i], NULL) != 0) { /*report error*/ }
  }

  cout << "Total: " << sum_total << endl;

  pthread_mutex_destroy(&sum_lock);
  return 0;
}
```

Question

If we have

```
    MyClass onTheStack;
    pthread_t child;
    pthread_create(&child, nullptr, foo, &onTheStack);
```

onTheStack is on the parents stack. However, each thread has its own stack. Can we still access onTheStack from the child? Why or why not?

Yes, we can still access onTheStack from the new thread. Threads share an address space. When a new thread is created, it gets a new stack. This stack is put somewhere inside the address space, but doesn't overwrite the stack of any other thread. Since all the stacks are in the same address space, they can be accessed by any other thread in the process.

Exercise

1) Write code to print "Woof!" "Meow!" and "Ssss!" from three different child threads:

```
// See sec9p1.cc for a compilable file
void* woof(void* data) {
      std::cout << "Woof!" << std::endl;
}

void* meow(void* data) { … }

void* ssss(void* data) { … }

int main() {
      int err;
      pthread_t dog, cat, snake;

      // create a woof thread
      if ( (err = pthread_create(&dog, nullptr,
            woof, nullptr)) != 0) {
            std::cerr << "Error making dog: " << strerror(err)
                  << std::endl;
      }

      // create the meow and ssss threads
      if ( (err = pthread_create(&cat, nullptr,
            meow, nullptr)) != 0) {
            std::cerr << "Error making cat: " << strerror(err)
                  << std::endl;
      }

      if ( (err = pthread_create(&snake, nullptr,
            ssss, nullptr)) != 0) {
            std::cerr << "Error making snake: " << strerror(err)
                  << std::endl;
      }

      // exit without stopping any running threads
      pthread_exit(nullptr);
}
```

What are some possible outputs of this program?

“Woof!\nMeow!\nSsss!\n”
“Ssss!\nMeow!\nWoof!\n”
“Woof!Meow!Ssss!\n\n\n”
“Meow!Woof!\n\nSsss!\n”
… and many more

The end lines aren’t guaranteed to come directly after the strings, since `<< std::endl` is a separate operation from `<< “Woof!”`. However, since each thread executes sequentially, each new line will only come after the related string is printed. Additionally, the code won’t crash or do anything horribly wrong, since putting to std::cin from multiple threads at the same time is guaranteed to be thread safe.

2) Calculating primes is slow. Use 10 threads to calculate the first 1,000 primes. Then, print them out in ascending order:

```
// See sec9p2.cc for a compilable file
#define NTHREAD 10
struct Bounds {
      int lo;
      int hi;
      Bounds(int lo, int hi): lo(lo), hi(hi) {}
};

bool isPrime(int num) { … }

void* getPrimes(void* data) {
      Bounds* b = reinterpret_cast<Bounds*>(data);
      // setup a way to store the primes we find in order
      std::vector<int>* primes = new std::vector<int>();

      // calculate primes
      for (int i = b->lo; i < b->hi; i++) {
            if (isPrime(i)) {
                  primes->push_back(i);
            }
      }

      // ???
      Return reinterpret_cast<void*>(primes);
}
```

// continued on next page

```
int main() {
      // make space to store our threads and data
      std::vector<std::unique_ptr<Bounds>> bounds;
      pthread_t threads[NTHREAD];

      // create and run our threads
      int err;
      for (int i = 0; i < NTHREAD; i++) {
            // for simplicity, we arbitrarily give every thread 100
            // numbers to calculate
            int lo = i * 100;
            int hi = (i + 1) * 100;
            bounds.push_back(std::unique_ptr<Bounds>(
                  new Bounds(lo, hi)));

            if ((err = pthread_create(&threads[i], nullptr,
                  getPrimes, bounds.back().get())) != 0) {

                  std::cout << "Thread create err on i = " <<
                        i << std::endl;
                  std::cout << strerror(err) << std::endl;
                  return -1;
            }
      }

      // wait for each thread to finish and get its data
      for (int i = 0; i < NTHREAD; i++) {
            // wait for thread, storing its return value
            std::vector<int>* out;
            err = pthread_join(threads[i],
                  reintrepret_cast<void**>(&out));
            if (err != 0) {
                  std::cout << std::endl <<
                        "Error while joining thread " << i <<
                        ", " << strerror(err) << std::endl;
                  continue;
            }

            // print the data
            for (int prime : *out) {
                  Std::cout << prime << ", ";
            }
            delete out;
```

```
        }
        // extra newline to prevent weird terminal wrapping
        std::cout << std::endl;
        return 0;
}
```

Boost Library

- Very useful for dealing with strings (HW 4!), such as trimming, pattern matching, splitting, replacing, etc.
- `#include <boost/algorithm/string.hpp>.`
- Examples: `boost::split, boost::to_upper, etc.`
- **View boost libraries online.**