

333 Section 10 - Concurrency and pthreads

Welcome back to section! :)

Process and Threads

- A process has a virtual address space. Each process is started with a single thread but can create additional threads.
- A thread contains a sequential execution of a program and is contained within a process.
- Threads of the same process share a memory/address space: use the same heap, globals, and code but each thread has its own stack.

POSIX threads (pthreads) API

- Part of the standard C/C++ libraries and declared in `pthread.h`.
- **Must compile and link with** `-pthread`.

```
int pthread_create(pthread_t *thread, const pthread_attr_t *attr,  
                  void *(*start_routine) (void *), void *arg);
```

→ `thread`: Output parameter for thread identifier

→ `attr`: Used to set thread attributes. Use `NULL/nullptr` for defaults.

→ `start_routine`: Pointer to a function that the thread will execute upon creation.

→ `arg`: A single argument that may be passed to `start_routine`. `NULL/nullptr` may be used if no argument is to be passed.

★ Creates a new thread and calls `start_routine(arg)`.

★ Returns 0 if successful and an error number otherwise.

```
int pthread_join(pthread_t thread, void **retval);
```

★ Called by parent thread to wait for the termination of the thread specified by `thread`. If `retval` is non-`NULL`, then `retval` acts as an output parameter and the address passed to `pthread_exit` by the finished thread is stored in it.

★ Returns 0 if successful and an error number otherwise.

```
void pthread_exit(void *retval);
```

★ Terminates the calling thread with an optional termination status parameter, `retval`, which can just be set to `NULL/nullptr`.

POSIX mutual exclusion (mutex) API

- Restrict access to sections of code in order to protect shared data from being simultaneously accessed by multiple threads.

```
int pthread_mutex_init(pthread_mutex_t *mutex,  
                       const pthread_mutexattr_t *attr);
```

★ Initializes the mutex referenced by `mutex` with attributes specified by `attr` (use `NULL/nullptr` for default attributes).

```
int pthread_mutex_destroy(pthread_mutex_t *mutex);
```

★ Destroys (*i.e.* uninitialized) the mutex object referenced by `mutex`.

```
int pthread_mutex_lock(pthread_mutex_t *mutex);
```

- ★ Attempts to acquire the mutex object referenced by `mutex` and blocks if it's currently held by another thread. Should be placed at the start of your critical section of code.

```
int pthread_mutex_unlock(pthread_mutex_t *mutex);
```

- ★ Releases the mutex object referenced by `mutex`. Should be placed at the end of your critical section of code.

Exercise 1

Imagine we have:

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

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

- List some reasons why it's better to use multiple threads within the same process rather than multiple processes running the same program.
- What benefits could there be to using multiple processes instead of multiple threads?
- Which registers will for sure be different between two threads that are executing different functions?
- How does the OS distinguish the threads?

Exercise 2

Consider the following multithreaded C program:

```
int g = 0;
void *worker(void *ignore) {
    for (int k = 1; k <= 3; k++) {
        g = g + k;
    }
    printf("g = %d\n", g);
    return NULL;
}

int main() {
    pthread_t t1, t2;
    int ignore;
    ignore = pthread_create(&t1, NULL, &worker, NULL);
    ignore = pthread_create(&t2, NULL, &worker, NULL);
    pthread_join(t1, NULL);
    pthread_join(t2, NULL);
    return EXIT_SUCCESS;
}
```

a) Give three different possible outputs (there are many)

b) What are the possible final values of the global variable 'g'? (circle all possible)

0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15+

Exercise 3

It's payday! It's time for UW to pay each of the 333 TAs their monthly salary. Each of the TA's bank account is inside the `bank_accounts[]` array and the person who is in charge of paying the TAs is a 333 student and decided to use `pthread`s to pay the TAs by adding 1000 into each bank account. Here is the program the student wrote:

```
// Assume all necessary libraries and header files are included
const int NUM_TAS = 10;

static int bank_accounts[NUM_TAS];
static pthread_mutex_t sum_lock;

void *thread_main(void *arg) {
    int *TA_index = reinterpret_cast<int*>(arg);

    pthread_mutex_lock(&sum_lock);
    bank_accounts[*TA_index] += 1000;
    pthread_mutex_unlock(&sum_lock);

    delete TA_index;
    return nullptr;
}

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

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

    for (int i = 0; i < NUM_TAS; i++) {
        cout << bank_accounts[i] << endl;
    }

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

(see next page)

Exercise 4

Calculating primes is slow.

In C++, use 10 threads to calculate the primes less than 1,000.

Then, print them out in ascending order:

```
#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

    // calculate primes

    // ???
    return
}
```

// continued on next page

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

    // create and run our threads
    int err;
    for (int i = 0; i < NTHREAD; i++) {

    }

    // wait for each thread to finish and get its data
    for (int i = 0; i < NTHREAD; i++) {
        // wait for thread, storing its return value

        // print the data

    }

    return 0;
}
```