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.

Exercise 1

How do threads and processes compare in these categories? Circle the correct answer.

	Multiple Threads	Multiple Processes
Memory / Address Space	Shared / Separate	Shared / Separate
Stack	Shared / Separate	Shared / Separate
Heap	Shared / Separate	Shared / Separate
Communication	Easy / Difficult	Easy / Difficult
Synchronization	Easy / Difficult / Not Applicable	Easy / Difficult / Not Applicable
Context-switch “Weight”	Light / Heavy	Light / Heavy
Crash Tolerance (what happens to the others when one dies?)	Tolerant / Not Tolerant	Tolerant / Not Tolerant

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);
```

Analogy – Parent: "Go do this {function}"
Parameters:

- `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.

Behavior:

- 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);
```

Analogy – Parent: "I'll wait for you to finish and report back your result"
Behavior:

- Called by the 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);
```

Analogy – Parent: "I changed my mind, you can stop now"
Behavior:

- 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);
```

Analogy – "Initialize the lock"
Behavior – 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);
```

Analogy – "Grab the lock or wait until I can get it"
Behavior – Destroys (*i.e.* uninitializes) the mutex object referenced by `mutex`.

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

Analogy – "Someone else can use the lock now"
Behavior – 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);
```

Analogy – "Someone else can use the lock now"
Behavior – Releases the `mutex` object referenced by `mutex`. Should be placed at the end of your critical section of code.

Exercise 2

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 pthreads 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 = static_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)

a) Does the program increase the TAs' bank accounts correctly? Why or why not?

b) Could we implement this program using processes instead of threads? Why would or why wouldn't we want to do this?

c) Assume that all the problems, if any, are now fixed. The student discovers that the program they wrote is kinda slow even though it's a multithreaded program. Why might it be the case? And how would you fix that?

Exercise 3

Journaling Activity! Take about 5 minutes with yourself or your neighbor to journal about your experiences with homework and exercises this quarter.

Guiding Questions:

- What were a couple main themes or takeaways from each homework?
- What lessons (e.g., concepts, tools, habits, etc.) did you learn from each homework?
- If you could change something about a certain homework or exercise what would it be?
    - Let us know in the course evals :)