

CIT 5950 Recitation 5 - Processes and Threads

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

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+