CSC 447: Parallel Programming for Multi-Core and Cluster Systems

Shared Memory Programming Using POSIX Threads

Instructor: Haidar M. Harmanani

Spring 2021

What are Pthreads?

- IEEE POSIX 1003.1c standard
- `pthreads` routines be grouped in the following categories
  - *Thread Management*: Routines to create, terminate, and manage the threads.
  - *Mutexes*: Routines for synchronization
  - *Condition Variables*: Routines for communications between threads that share a mutex.
  - *Synchronization*: Routines for the management of read/write locks and barriers.
- All identifiers in the threads' library begin with `pthread_`

Preliminaries

- All major thread libraries on Unix systems are Pthreads-compatible
- Include `pthread.h` in the main file
- Compile program with `-lpthread`
  - `gcc -o test test.c -lpthread`
  - may not report compilation errors otherwise but calls will fail
  - The MacOS has dropped the need for the inclusion of `-lpthread`
  - Check your OS's requirement!
- Good idea to check return values on common functions

The Pthreads API

Routine Prefix	Functional Group
pthread_	Threads themselves and miscellaneous subroutines
pthread_attr_	Thread attributes objects
pthread_mutex_	Mutexes
pthread_mutexattr_	Mutex attributes objects.
pthread_cond_	Condition variables
pthread_condattr_	Condition attributes objects
pthread_key_	Thread-specific data keys
pthread_rwlock_	Read/write locks
pthread_barrier_	Synchronization barriers

Creating Threads

- Identify portions of code to thread
- Encapsulate code into function
  - If code is already a function, a driver function may need to be written to coordinate work of multiple threads
- Use **`pthread_create()`** call to assign thread(s) spawn a thread that runs the function

pthread_create

- **`int pthread_create(tid, attr, function, arg);`**
  - **`pthread_t *tid`**
    - Handle of created thread
  - **`const pthread_attr_t *attr`**
    - attributes of thread to be created
    - You can specify a thread attributes object, or NULL for the default values.
  - **`void *(*function)(void *)`**
    - The C routine that the thread will execute once it is created
  - **`void *arg`**
    - single argument to function
    - NULL may be used if no argument is to be passed.

Example: pthread_create

```
pthread_create(&threads[t], NULL, HelloWorld, (void *) t)
```

- Thread handle returned via `pthread_t` structure
  - Specify NULL to use default attributes
- Single argument sent to function
  - If no arguments to function, specify NULL
- Check error codes!

EAGAIN - insufficient resources to create thread
EINVAL - invalid attribute

What is the Outcome of the following code?

```
#include <stdio.h>
#include <pthread.h>
void *hello ()
{
   printf("Hello Thread\n");
}

main() {
      pthread_t tid;
      pthread_create(&tid, NULL, hello, NULL);
}
```

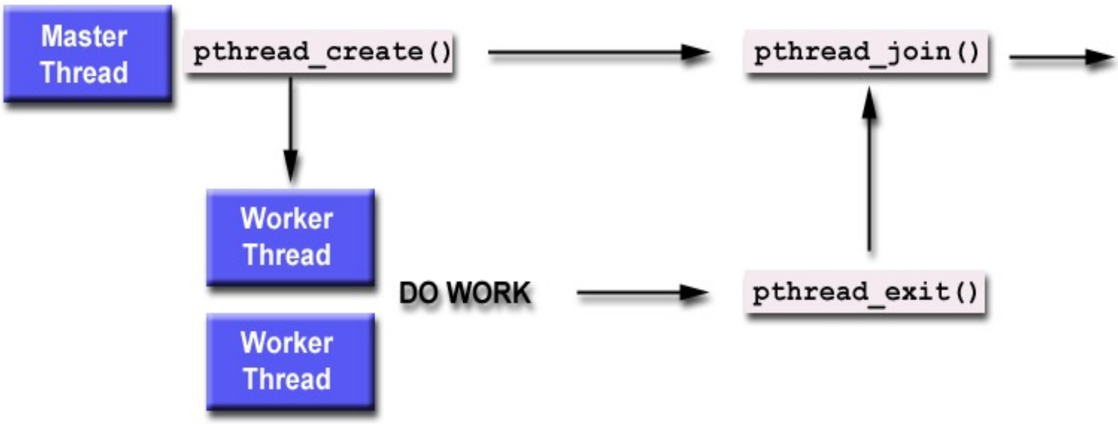
Example: Thread Creation

- The outcome is not what we would expect!
- In fact nothing is printed on screen.
- Why?

Example: Thread Creation

- The outcome is not what we would expect!
- In fact nothing is printed on screen.
- Why?
    - Main thread is the process and when the process ends, all threads are cancelled, too.
    - Thus, if the `pthread_create` call returns before the OS has had the time to set up the thread and begin execution, the thread will die a premature death when the process ends.

pthread_join



Waiting for a Thread

int pthread_join(tid, val_ptr);

- `pthread_join` will block until the thread associated with the `pthread_t` handle has terminated.
  - There is no single function that can join multiple threads.
- The second parameter returns a pointer to a value from the thread being joined.
- `pthread_join()` can be used to wait for one thread to terminate.

`pthread_t tid`
- handle of *joinable* thread

`void **val_ptr`
- exit value returned by joined thread

A Better Hello Threads…

```c
#include <pthread.h>
#include <stdio.h>
#include <stdlib.h>

#define NUM_THREADS 8
void* hello(void* threadID) {
    long id = (long) threadID;
    printf("Hello World, this is thread %ld\n", id);
    return NULL;
}

int main(int argc, char argv[]) {
    long t;
    pthread_t thread_handles[NUM_THREADS];
    for(t=0 ; t<NUM_THREADS; t++)
        pthread_create(&thread_handles[t], NULL, hello, (void *) t);
    printf("Hello World, this is the main thread\n");
    for(t=0; t<NUM_THREADS; t++)
        pthread_join(thread_handles[t], NULL);
    return 0;
}
```


Sample Execution Runs

```
yoda:~ haidar$ ./a.out
Hello World, this is thread 0
Hello World, this is thread 1
Hello World, this is thread 2
Hello World, this is thread 3
Hello World, this is thread 4
Hello World, this is thread 5
Hello World, this is the main thread
Hello World, this is thread 7
Hello World, this is thread 6
```

```
yoda:~ haidar$ ./a.out
Hello World, this is thread 0
Hello World, this is thread 1
Hello World, this is thread 2
Hello World, this is thread 3
Hello World, this is thread 4
Hello World, this is the main thread
Hello World, this is thread 5
Hello World, this is thread 7
Hello World, this is thread 6
```

Thread States

- pthreads threads have two states
  – *joinable* and *detached*
- A detached thread when you know you won't want to wait for it with **`pthread_join()`**
- Threads are joinable by default
  – Resources are kept until **`pthread_join`**
  – Can be reset with attributes or API call
- Detached threads cannot be joined
  – Resources can be reclaimed at termination
  – Cannot reset to be *joinable*

Example: Multiple Threads with Joins

```
#include <stdio.h>
#include <pthread.h>
#define NUM_THREADS 4
void *hello () {
    printf("Hello Thread\n");
}
main() {
    pthread_t tid[NUM_THREADS];
    for (int i = 0; i < NUM_THREADS; i++)
       pthread_create(&tid[i], NULL, hello, NULL);

    for (int i = 0; i < NUM_THREADS; i++)
       pthread_join(tid[i], NULL);
}
```

Avoiding Data Races

- Scope variables to be local to threads
  – Variables declared within threaded functions
  – Allocate on thread's stack
  – Thread Local Storage (TLS)
- Control shared access with critical regions
  – Mutual exclusion and synchronization
  – Lock, semaphore, condition variable, critical section, mutex…

pthread's Mutex

- Simple, flexible, and efficient
- Enables correct programming structures for avoiding race conditions
- `Mutex` variables must be declared with type **`pthread_mutex_t`**, and must be initialized before they can be used
- Attributes are set using **`pthread_mutexattr_t`**
- The `mutex` is initially unlocked.

Initializing `mutex` Variables

- Two ways:
  - Statically, when it is declared:
    - pthread_mutex_t mymutex = PTHREAD_MUTEX_INITIALIZER;
  - Dynamically, with the pthread_mutex_init() routine.
    - Permits setting mutex object attributes, *attr*.

pthread_mutex_init

```
int pthread_mutex_init( mutex, attr );
```

`pthread_mutex_t *mutex`

– mutex to be initialized

`const pthread_mutexattr_t *attr`

– attributes to be given to mutex

- The Pthreads standard defines three optional mutex attributes:
  - Protocol: Specifies the protocol used to prevent priority inversions for a mutex.
  - Prioceiling: Specifies the priority ceiling of a mutex.
  - Process-shared: Specifies the process sharing of a mutex.

Alternate Initialization

- Can also use the static initializer PTHREAD_MUTEX_INITIALIZER

```
pthread_mutex_t mtx1 = PTHREAD_MUTEX_INITIALIZER;
```

  – Uses default attributes

- Programmer must always pay attention to mutex scope
  – Must be visible to threads

pthread_mutex_lock

```
int pthread_mutex_lock( mutex );
```

`pthread_mutex_t *mutex`

- mutex to attempt to lock

- Used by a thread to acquire a lock on the specified mutex variable
  – If mutex is locked by another thread, calling thread is blocked
- Mutex is held by calling thread until unlocked
  – Mutex lock/unlock must be paired or deadlock occurs

`EINVAL` **- mutex is invalid**
`EDEADLK` **- calling thread already owns mutex**

pthread_mutex_trylock

- Attempt to lock a mutex.
- If the mutex is already locked, the routine will return immediately with a "busy" error code.
- This routine may be useful in preventing deadlock conditions, as in a priority-inversion situation.

pthread_mutex_unlock

```
int pthread_mutex_unlock( mutex );
```

`pthread_mutex_t *mutex`
– mutex to be unlocked

`EINVAL` **- mutex is invalid**
`EPERM` **- calling thread does not own mutex**

Freeing **`mutex`** Objects and Attributes

- Used to free a **`mutex`** object which is no longer needed
- **`pthread_mutexattr_init()`** and **`pthread_mutexattr_destroy()`**
  - Create and destroy mutex attribute objects respectively
- **`pthread_mutex_destroy()`**
  - Used to free a mutex object which is no longer needed.

More on Mutexes

Acquiring and Releasing Mutexes

```
int pthread_mutex_lock(              // Lock a mutex
  pthread_mutex_t *mutex);
int pthread_mutex_unlock(            // Unlock a mutex
  pthread_mutex_t *mutex);
int pthread_mutex_trylock(           // Nonblocking lock
  pthread_mutex_t *mutex);
```

Arguments:

Each function takes the address of a mutex variable.

Return value:

0 if successful. Error code from `<errno.h>` otherwise.

Notes:

The `pthread_mutex_trylock()` routine attempts to acquire a mutex but will not block. This routine returns the POSIX Threads constant `EBUSY` if the mutex is locked.

More on Mutexes

Dynamically Allocated Mutexes

```
pthread_mutex_t *lock;                     // Declare a pointer to a lock
lock=(pthread_mutex_lock_t *) malloc(sizeof(pthread_mutex_t));
pthread_mutex_init(lock, NULL);
    /*
     * Code that uses this lock.
     */
pthread_mutex_destroy(lock);
free(lock);
```


Thread Function: Semaphore / Mutex

Semaphore

```
void *sum_sem(void *vargp)
{
    int myid = *((int *)vargp);
    size_t start = myid * nelems_per_thread;
    size_t end = start + nelems_per_thread;
    size_t i;

    for (i = start; i < end; i++) {
        sem_wait(&semaphore);
        global_sum += i;
        sem_post(&semaphore);
    }
    return NULL;
}
```

Mutex

```
pthread_mutex_lock(&mutex);
global_sum += i;
pthread_mutex_unlock(&mutex);
```

Semaphore / Mutex Performance

- Terrible Performance
  - 2.5 seconds ➔ ~10 minutes
- Mutex 3X faster than semaphore
- Clearly, neither is successful