

Multi-threading in C with Pthreads

ICS432 - Fall 2008 Concurrent and High-Performance Programming

Henri Casanova (henric@hawaii.edu)

Pthreads: POSIX Threads

- Pthreads is a standard set of C library functions for multithreaded programming
 - IEEE Portable Operating System Interface, POSIX, section 1003.1 standard, 1995
- Pthread Library (60+ functions)
 - Thread management: create, exit, detach, join, . . .
 - Thread cancellation
 - Mutex locks: init, destroy, lock, unlock, . . .
 - Condition variables: init, destroy, wait, timed wait, . . .
 - . . .
- Programs must include the file `pthread.h`
- Programs must be linked with the pthread library (`-lpthread`)

Pthreads Naming Convention

- Types: `pthread[_object]_t`
- Functions: `pthread[_object]_action`
- Constants/Macros: `PTHREAD_PURPOSE`
- Examples:
 - `pthread_t`: the type of a thread
 - `pthread_create()`: creates a thread
 - `pthread_mutex_t`: the type of a mutex lock
 - `pthread_mutex_lock()`: lock a mutex
 - `PTHREAD_CREATE_DETACHED`

pthread_self()

- Returns the thread identifier for the calling thread
 - At any point in its instruction stream a thread can figure out which thread it is
 - Convenient to be able to write code that says: "If you're thread 1 do this, otherwise do that"
 - However, the thread identifier is an opaque object (just a `pthread_t` value)
 - you must use `pthread_equal()` to test equality

```
pthread_t pthread_self(void);  
int pthread_equal(pthread_t id1, pthread_t id2);
```

pthread_create()

- Creates a new thread

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

- Returns 0 to indicate success, otherwise returns error code
- `thread`: output argument for the id of the new thread
- `attr`: input argument that specifies the attributes of the thread to be created (NULL = default attributes)
- `start_routine`: function to use as the start of the new thread
 - must have prototype: `void * foo(void*)`
- `arg`: argument to pass to the new thread routine
 - If the thread routine requires multiple arguments, they must be passed bundled up in an array or a structure

pthread_create() example

- Want to create a thread to compute the sum of the elements of an array
 - `void *do_work(void *arg);`
- Needs three arguments
 - the array, its size, where to store the sum
 - we need to bundle them in a structure

```
struct arguments {  
    double *array;  
    int size;  
    double *sum;  
}
```

pthread_create() example

```
int main(int argc, char *argv) {
    double array[100];
    double sum;
    pthread_t worker_thread;
    struct arguments *arg;

    arg = (struct arguments *)calloc(1,
                                     sizeof(struct arguments));
    arg->array = array;
    arg->size=100;
    arg->sum = &sum;

    if (pthread_create(&worker_thread, NULL,
                     do_work, (void *)arg)) {
        fprintf(stderr, "Error while creating thread\n");
        exit(1);
    }
    ...
}
```

pthread_create() example

```
void *do_work(void *arg) {
    struct arguments *argument;
    int i, size;
    double *array;
    double *sum;

    argument = (struct arguments*)arg;

    size = argument->size;
    array = argument->array;
    sum = argument->sum;

    *sum = 0;
    for (i=0;i<size;i++)
        *sum += array[i];

    return NULL;
}
```

Comments about the example

- The “main thread” continues its normal execution after creating the “child thread”
- **IMPORTANT:** If the main thread terminates, then all threads are killed!
 - We will see that there is a join() function
- Of course, memory is shared by the parent and the child (the array, the location of the sum)
 - nothing prevents the parent from doing something to it while the child is still executing
 - which may lead to a wrong computation
 - we will see that Pthreads provide locking mechanisms
- The bundling and unbundling of arguments is a bit tedious

Memory Management of Args

- The parent thread allocates memory for the arguments
- **Warning #1:** you don't want to free that memory before the child thread has a chance to read it
 - That would be a race condition
 - Better to let the child do the freeing
- **Warning #2:** if you create multiple threads you want to be careful there is no sharing of arguments, or that the sharing is safe
 - For instance, if you reuse the same data structure for all threads and modify its fields before each call to pthread_create(), some threads may not be able to read the arguments destined to them
 - Safest way: have a separate arg structure for each thread

pthread_exit()

- Terminates the calling thread

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

- The return value is made available to another thread calling a pthread_join() (see next slide)
- My previous example had the thread just return from function do_work()
 - In this case the call to pthread_exit() is implicit
 - The return value of the function serves as the argument to the (implicitly called) pthread_exit().

pthread_join()

- Causes the calling thread to wait for another thread to terminate

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

- thread: input parameter, id of the thread to wait on
- value_ptr: output parameter, value given to pthread_exit() by the terminating thread (which happens to always be a void *)
- returns 0 to indicate success, error code otherwise
- multiple simultaneous calls for the same thread are not allowed

pthread_kill()

- Causes the termination of a thread

```
int pthread_kill(  
    pthread_t thread,  
    int sig);
```

- **thread**: input parameter, id of the thread to terminate
- **sig**: signal number
- returns 0 to indicate success, error code otherwise

pthread_join() example

```
int main(int argc, char *argv) {  
    double array[100];  
    double sum;  
    pthread_t worker_thread;  
    struct arguments *arg;  
    void *return_value;  
  
    arg = (struct arguments *)calloc(1, sizeof(struct arguments));  
    arg->array = array;  
    arg->size=100;  
    arg->sum = &sum;  
    if (pthread_create(&worker_thread, NULL,  
        do_work, (void *)arg)) {  
        fprintf(stderr, "Error while creating thread\n");  
        exit(1);  
    }  
    ...  
    if (pthread_join(worker_thread, &return_value)) {  
        fprintf(stderr, "Error while waiting for thread\n");  
        exit(1);  
    }  
}
```

pthread_join() Warning

- This is a common “bug” that first-time pthread programmers encounter
- Without the call to pthread_join() the previous program may end immediately, with the main thread reaching the end of main() and exiting, thus killing all other threads perhaps even before they have had a chance to execute

pthread_join() Warning

- When creating multiple threads be careful to store the handle of each thread in a separate variable
 - Typically one has an array of thread handles
- That way you’ll be able to call pthread_join() for each thread
- Also, note that the following code is sequential!

```
for (i=0; i < num_threads; i++) {  
    pthread_create(&(threads[i]),...)  
    pthread_join(threads[i],...)  
}
```

Thread Attributes

- One of the parameters to pthread_create() is a **thread attribute**
- In all our previous examples we have set it to NULL
- But it can be very useful and provides a simple way to set options:
 - Initialize an attribute
 - Set its value with some Pthread API call
 - Pass it to Pthread API functions like pthread_create()

pthread_attr_init()

- Initialized the thread attribute object to the default values

```
int pthread_attr_init(  
    pthread_attr_t *attr);
```

- Return 0 to indicate success, error code otherwise
- **attr**: pointer to a thread attribute

Detached Thread

- One option when creating a thread is whether it is joinable or detached
 - **Joinable**: another thread can call join on it
 - By default a thread is joinable
 - **Detached**: no thread can call join on it
- Let's look at the function that allows to set the "detached state"

pthread_attr_setdetachstate()

- Sets the detach state attribute

```
int pthread_attr_setdetachstate(  
    pthread_attr_t *attr,  
    int detachstate);
```

- returns 0 to indicate success, error code otherwise
- attr: input parameter, thread attribute
- detachstate: can be either
 - PTHREAD_CREATE_DETACHED
 - PTHREAD_CREATE_JOINABLE (default)

Detach State

- Detached threads have all resources freed when they terminate
- Joinable threads have state information about the thread kept even after they finish
 - To allow for a thread to join a finished thread
 - So-called "no rush to join"
- So, if you know that you will not need to join a thread, create it in a detached state so that you save resources
- This is lean-and-mean C, as opposed to hand-holding Java, and every little saving is important

Creating a Detached Thread

```
#include <pthread.h>  
#define NUM_THREAD 25  
  
void *thread_routine (void *arg) {  
    printf("Thread %d, my TID is %u\n",  
        (int)arg, pthread_self());  
    pthread_exit(0);  
}
```

Creating a Detached Thread

```
int main() {  
    pthread_attr_t attr;  
    pthread_t tids[NUM_THREADS];  
    int x;  
  
    pthread_attr_init(&attr);  
    pthread_attr_setdetachstate(&attr, PTHREAD_CREATE_DETACHED);  
    pthread_create(&tids[x],  
                 &attr,  
                 thread_routine,  
                 (void *)x);  
  
    . . . // should take a while otherwise  
    . . . // the child may not have time to run  
}
```

Mutual Exclusion and Pthreads

- Pthreads provide a simple mutual exclusion lock
- Lock creation

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

- returns 0 on success, an error code otherwise
- mutex: output parameter, lock
- attr: input, lock attributes
 - NULL: default
 - There are functions to set the attribute (look at the man pages if you're interested)

Pthread: Locking

- Locking a lock
 - If the lock is already locked, then the calling thread is blocked
 - If the lock is not locked, then the calling thread acquires it
- ```
int pthread_mutex_lock(
 pthread_mutex_t *mutex);
```
- returns 0 on success, an error code otherwise
  - **mutex**: input parameter, lock

## Pthread: Locking

- Just checking
    - Returns instead of locking
- ```
int pthread_mutex_trylock(  
    pthread_mutex_t *mutex);
```
- returns 0 on success, EBUSY if the lock is locked, an error code otherwise
 - **mutex**: input parameter, lock

Synchronizing pthreads

- Releasing a lock
- ```
int pthread_mutex_unlock(
 pthread_mutex_t *mutex);
```
- returns 0 on success, an error code otherwise
  - **mutex**: input parameter, lock
- Pthreads implement exactly the concept of locks as it was described in the previous lecture notes

## Cleaning up memory

- Releasing memory for a mutex attribute

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

- Releasing memory for a mutex

```
int pthread_mutexattr_destroy(
 pthread_mutexattr_t *mutex);
```

## Condition Variables

- Pthreads also provide condition variables as they were described in the previous lecture notes
  - Condition variables are of the type `pthread_cond_t`
  - They are used in conjunction with mutex locks
- Let's look at the API's functions

## pthread\_cond\_init()

- Creating a condition variable

```
int pthread_cond_init(
 pthread_cond_t *cond,
 const pthread_condattr_t *attr);
```

- returns 0 on success, an error code otherwise
- **cond**: output parameter, condition
- **attr**: input parameter, attributes (default = NULL)

## pthread\_cond\_wait()

- Waiting on a condition variable

```
int pthread_cond_wait(
 pthread_cond_t *cond,
 pthread_mutex_t *mutex);
```

- returns 0 on success, an error code otherwise
- `cond`: input parameter, condition
- `mutex`: input parameter, associated mutex

## pthread\_cond\_signal()

- Signaling a condition variable

```
int pthread_cond_signal(
 pthread_cond_t *cond);
```

- returns 0 on success, an error code otherwise
- `cond`: input parameter, condition
- “Wakes up” one thread out of the possibly many threads waiting for the condition
  - The thread is chosen non-deterministically

## pthread\_cond\_broadcast()

- Signaling a condition variable

```
int pthread_cond_broadcast(
 pthread_cond_t *cond);
```

- returns 0 on success, an error code otherwise
- `cond`: input parameter, condition
- “Wakes up” ALL threads waiting for the condition
  - May be useful in some applications

## Condition Variable: example

- Say I want to have multiple threads wait until a counter reaches a maximum value and be awakened when it happens

```
pthread_mutex_lock(&lock);
while (count < MAX_COUNT) {
 pthread_cond_wait(&cond, &lock);
}
```

```
pthread_mutex_unlock(&lock)
```

- Locking the lock so that we can read the value of count without the possibility of a race condition
- Calling `pthread_cond_wait()` in a while loop to avoid “spurious wakes ups”
- When going to sleep the `pthread_cond_wait()` function **implicitly releases the lock**
- When waking up the `pthread_cond_wait()` function **implicitly acquires the lock**
- The lock is unlocked after exiting from the loop

## pthread\_cond\_timedwait()

- Waiting on a condition variable with a timeout

```
int pthread_cond_timedwait(
 pthread_cond_t *cond,
 pthread_mutex_t *mutex,
 const struct timespec *delay);
```

- returns 0 on success, an error code otherwise
- `cond`: input parameter, condition
- `mutex`: input parameter, associated mutex
- `delay`: input parameter, timeout (same fields as the one used for `gettimeofday`)

## Putting a Pthread to Sleep

- To make a Pthread thread sleep you should use the `usleep()` function

```
#include <unistd.h>
```

```
int usleep(usecond_t microseconds);
```

- Do not use the `sleep()` function as it may not be safe to use it in a multi-threaded program

## PThreads: Conclusion

- Pthreads implement everything we talked about in the previous set of lecture notes almost directly
- There are many other useful functions, some which you may even need for your programming assignment
  - e.g., Read/Write locks
- You can find more information all over
  - Man pages
  - PThread Tutorial:  
<http://www.llnl.gov/computing/tutorials/pthreads/>

## Beyond Locks and Cond. Vars

- At this point, we have everything we need to write concurrent programs
  - **Locks** for efficient mutual exclusion
  - **Condition variables** for non-wasteful thread synchronization and/or communication
- In the next lecture we'll see another abstraction for thread synchronization and mutual exclusion: Semaphores