Operating Systems

27. Interlude: Thread API

Thread Creation

- How to create and control threads?

```
#include <pthread.h>

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

  - thread: Used to interact with this thread.
  - attr: Used to specify any attributes this thread might have.
    - Stack size, Scheduling priority, …
  - start_routine: the function this thread start running in.
  - arg: the argument to be passed to the function (start routine)
    - *a void pointer* allows us to pass in *any type of* argument.

Thread Creation (Cont.)

- If `start_routine` instead requires another type argument, the declaration would look like this:
  - An integer argument:

```
int
pthread_create(…, // first two args are the same
                     void*  (*start_routine)(int),
                     int    arg);
```

  - Return an integer:

```
int
pthread_create(…, // first two args are the same
                     int  (*start_routine)(void*),
                     void*    arg);
```

Example: Creating a Thread

```
#include <pthread.h>

typedef struct __myarg_t {
        int a;
        int b;
} myarg_t;

void *mythread(void *arg) {
        myarg_t *m = (myarg_t *) arg;
        printf(“%d %d\n”, m->a, m->b);
        return NULL;
}

int main(int argc, char *argv[]) {
        pthread_t p;
        int rc;

        myarg_t args;
        args.a = 10;
        args.b = 20;
        rc = pthread_create(&p, NULL, mythread, &args);
        …
}
```

Wait for a thread to complete

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

- `thread`: Specify which thread *to wait for.*
- `value_ptr`: A pointer to the return value
  - Because `pthread_join()` routine changes the value, you need to pass the pointer to that value.

Example: Waiting for Thread Completion

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

typedef struct __myarg_t {
    int a;
    int b;
} myarg_t;

typedef struct __myret_t {
    int x;
    int y;
} myret_t;

void *mythread(void *arg) {
    myarg_t *m = (myarg_t *) arg;
    printf(“%d %d\n”, m->a, m->b);
    myret_t *r = malloc(sizeof(myret_t));
    r->x = 1;
    r->y = 2;
    return (void *) r;
}

```

Example: Waiting for Thread Completion (Cont.)

```c
int main(int argc, char *argv[]) {
    int rc;
    pthread_t p;
    myret_t *m;

    myarg_t args;
    args.a = 10;
    args.b = 20;
    pthread_create(&p, NULL, mythread, &args);
    pthread_join(p, (void **) &m);  // this thread has been
                                     // waiting inside of the
                                     // pthread_join() routine.
    printf(“returned %d %d\n”, m->x, m->y);
    return 0;
}
```

Example: Dangerous code

- Be careful with how values are returned from a thread.

```
void *mythread(void *arg) {
    myarg_t *m = (myarg_t *) arg;
    printf(“%d %d\n”, m->a, m->b);
    myret_t r; // ALLOCATED ON STACK: BAD!
    r.x = 1;
    r.y = 2;
    return (void *) &r;
}
```

  - When the variable `r` returns, it is automatically de-allocated.

Example: Simpler Argument Passing to a Thread

- Just passing in a single value

```
void *mythread(void *arg) {
    int m = (int) arg;
    printf(“%d\n”, m);
    return (void *) (arg + 1);
}

int main(int argc, char *argv[]) {
    pthread_t p;
    int rc, m;
    pthread_create(&p, NULL, mythread, (void *) 100);
    pthread_join(p, (void **) &m);
    printf(“returned %d\n”, m);
    return 0;
}
```

Locks

- Provide mutual exclusion to a critical section
  - Interface

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

  - Usage (w/o *lock initialization* and *error check*)

```
pthread_mutex_t lock;
pthread_mutex_lock(&lock);
x = x + 1; // or whatever your critical section is
pthread_mutex_unlock(&lock);
```

    - No other thread holds the lock → the thread will acquire the lock and enter the critical section.
    - If another thread hold the lock → the thread will not return from the call until it has acquired the lock.

Locks (Cont.)

- All locks must be properly initialized.
  - One way: using `PTHREAD_MUTEX_INITIALIZER`

```
pthread_mutex_t lock = PTHREAD_MUTEX_INITIALIZER;
```

  - The dynamic way: using `pthread_mutex_init()`

```
int rc = pthread_mutex_init(&lock, NULL);
assert(rc == 0); // always check success!
```

Locks (Cont.)

- Check errors code when calling lock and unlock
  - An example wrapper

```
// Use this to keep your code clean but check for failures
// Only use if exiting program is OK upon failure
void Pthread_mutex_lock(pthread_mutex_t *mutex) {
    int rc = pthread_mutex_lock(mutex);
    assert(rc == 0);
}
```

- These two calls are used in lock acquisition

```
int pthread_mutex_trylock(pthread_mutex_t *mutex);
int pthread_mutex_timelock(pthread_mutex_t *mutex,
                           struct timespec *abs_timeout);
```

  - trylock: return failure if the lock is already held
  - timelock: return after a timeout

Locks (Cont.)

- These two calls are also used in lock acquisition

```
int pthread_mutex_trylock(pthread_mutex_t *mutex);
int pthread_mutex_timelock(pthread_mutex_t *mutex,
                           struct timespec *abs_timeout);
```

  - `trylock`: return failure if the lock is already held
  - `timelock`: return after a timeout or after acquiring the lock

Condition Variables

- **Condition variables** are useful when some kind of signaling must take place between threads.

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

- `pthread_cond_wait`:
    - Put the calling thread to sleep.
    - Wait for some other thread to signal it.
- `pthread_cond_signal`:
    - Unblock at least one of the threads that are blocked on the condition variable

Condition Variables (Cont.)

- A thread calling wait routine:

```
pthread_mutex_t lock = PTHREAD_MUTEX_INITIALIZER;
pthread_cond_t init = PTHREAD_COND_INITIALIZER;

pthread_mutex_lock(&lock);
while (initialized == 0)
        pthread_cond_wait(&init, &lock);
pthread_mutex_unlock(&lock);
```

  - The wait call releases the lock when putting said caller to sleep.
  - Before returning after being woken, the wait call re-acquire the lock.

- A thread calling signal routine:

```
pthread_mutex_lock(&lock);
initialized = 1;
pthread_cond_signal(&init);
pthread_mutex_unlock(&lock);
```

Condition Variables (Cont.)

- The waiting thread **re-checks** the condition in a while loop, instead of a simple if statement.

```
pthread_mutex_t lock = PTHREAD_MUTEX_INITIALIZER;
pthread_cond_t init = PTHREAD_COND_INITIALIZER;

pthread_mutex_lock(&lock);
while (initialized == 0)
        pthread_cond_wait(&init, &lock);
pthread_mutex_unlock(&lock);
```

  - Without rechecking, the waiting thread will continue thinking that the condition has changed *even though it has not*.

Condition Variables (Cont.)

- Don’t ever to this.
  - A thread calling wait routine:

```
while(initialized == 0)
          ; // spin
```

  - A thread calling signal routine:

```
initialized = 1;
```

  - It performs poorly in many cases. → just wastes CPU cycles.
  - It is error prone.

Compiling and Running

- To compile them, you must include the header `pthread.h`
  - Explicitly link with the pthreads library, by adding the `-pthread` flag.

```
prompt> gcc -o main main.c -Wall -pthread
```

  - For more information,

```
man -k pthread
```