

27. Interlude: Thread API

Operating System: Three Easy Pieces

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 (OUT).
- ◆ **attr**: Used to specify any attributes this thread might have.
 - Stack size, Scheduling priority, ... (IN)
- ◆ **start_routine**: the function this thread start running in (IN)
- ◆ **arg**: the argument to be passed to the function (start routine) (IN)
 - *a void pointer* allows us to pass in *any type of* argument.
- ◆ Returns 0 if went good (a error code otherwise: EAGAIN, EINVAL, EPERM)

Thread Creation (Cont.)

- If `start_routine` instead required another type argument, the declaration would look like this (example):

- ◆ An integer argument:

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

- ◆ Input is anything (usually a pointer to struct for multiple arguments or even internal returns), 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 we want to put the return value of the start routine (ouch!)
 - Because `pthread_join()` routine changes the value, you need to **pass in a pointer** to that value.
- ◆ Returns 0 if good, or EINVAL, ESRCH if err

Example: Waiting for Thread Completion

```
1  #include <stdio.h>
2  #include <pthread.h>
3  #include <assert.h>
4  #include <stdlib.h>
5
6  typedef struct __myarg_t {
7      int a;
8      int b;
9  } myarg_t;
10
11 typedef struct __myret_t {
12     int x;
13     int y;
14 } myret_t;
15
16 void *mythread(void *arg) {
17     myarg_t *m = (myarg_t *) arg;
18     printf("%d %d\n", m->a, m->b);
19     myret_t *r = malloc(sizeof(myret_t));
20     r->x = 1;
21     r->y = 2;
22     return (void *) r;
23 }
24
```

Example: Waiting for Thread Completion (Cont.)

```
25  int main(int argc, char *argv[]) {
26      int rc;
27      pthread_t p;
28      myret_t *m;
29
30      myarg_t args;
31      args.a = 10;
32      args.b = 20;
33      pthread_create(&p, NULL, mythread, &args);
34      pthread_join(p, (void **) &m); // this thread has been
                                     // waiting inside of the
                                     // pthread_join() routine.
35      printf("returned %d %d\n", m->x, m->y);
36      return 0;
37 }
```

Example: Dangerous code

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

```
1  void *mythread(void *arg) {
2      myarg_t *m = (myarg_t *) arg;
3      printf("%d %d\n", m->a, m->b);
4      myret_t r; // ALLOCATED ON STACK: BAD!
5      r.x = 1;
6      r.y = 2;
7      return (void *) &r;
8  }
```

- ◆ When the variable `r` returns, it is automatically **de-allocated**.
- ◆ Don't malloc here! (memory leak prone) [bad example before]
 - Better to be consistent a allocate and free in parent

Example: Simpler Argument Passing to a Thread

▣ Just passing in a single value

```
1  void *mythread(void *arg) {
2      int m = (int) arg;
3      printf("%d\n", m);
4      return (void *) (arg + 1);
5  }
6
7  int main(int argc, char *argv[]) {
8      pthread_t p;
9      int rc, m;
10     pthread_create(&p, NULL, mythread, (void *) 100);
11     pthread_join(p, (void **) &m);
12     printf("returned %d\n", m);
13     return 0;
14 }
```

From a practical perspective
using threads this way is pointless!
(just do a procedure call)

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** (i.e. unlocked value).

- ◆ 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 && "Error in mutex init");
```

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 && "Error in acquire");
}
```

▣ 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

Not a bad idea to define a wrapper: much cleaner code

```
1 #ifndef __MYTHREADS_h_
2 #define __MYTHREADS_h_
3
4 #include <pthread.h>
5 #include <assert.h>
6 #include <sched.h>
7
8 void
9 Pthread_mutex_lock(pthread_mutex_t *m)
10 {
11     int rc = pthread_mutex_lock(m);
12     assert(rc == 0);
13 }
14
15 void
16 Pthread_mutex_unlock(pthread_mutex_t *m)
17 {
18     int rc = pthread_mutex_unlock(m);
19     assert(rc == 0);
20 }
21
22 void
23 Pthread_create(pthread_t *thread, const pthread_attr_t *attr,
24               void *(*start_routine)(void*), void *arg)
25 {
26     int rc = pthread_create(thread, attr, start_routine, arg);
27     assert(rc == 0);
28 }
29
30 void
31 Pthread_join(pthread_t thread, void **value_ptr)
32 {
33     int rc = pthread_join(thread, value_ptr);
34     assert(rc == 0);
35 }
36
37 #endif // __MYTHREADS_h_
```

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 **pthread library**, by adding the `-pthread` flag.

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

- ◆ For more information,

```
man -k pthread
```

Thread API Use Guidelines

- ❑ Keep it simple
 - ◆ Tricky thread interactions lead to (hard to find) bugs
- ❑ Minimize thread interaction
 - ◆ Limits scalability
- ❑ Initialize mutex and cond vars
- ❑ Check always return codes
- ❑ Be careful how to pass arguments and get values:
 - ◆ A good practice is to allocate/free memory in the calling thread
 - ◆ Be careful with heap
- ❑ Each thread has his own stack
- ❑ Always use cond. variables to signal between threads
- ❑ **Read the man pages**

- Disclaimer: Disclaimer: This lecture slide set is used in AOS course at University of Cantabria by V.Puente. Was initially developed for Operating System course in Computer Science Dept. at Hanyang University. This lecture slide set is for OSTEP book written by Remzi and Andrea Arpaci-Dusseau (at University of Wisconsin)