【OS】Day31

Class	Operating System: Three Easy Pieces
Date	@February 2, 2022

【Ch27】Thread API

27.1 Thread Creation

The first thing we have to be able to do to write a multi-threaded program is to create new threads, and thus some kind of thread creation interface must exist.

In POSIX, it is easy:

```
#include <pthread.h>

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

There are four arguments in this declaration: `thread`, `attr`, `start_routine`, and `arg`.

1. The first, thread, is a pointer to a structure of type `pthread_t`.

   We'll use this structure to interact with this thread, and thus we need to pass it to `pthread_create()` in order to initialize it.

2. The second argument, `attr`, is used to specify any attributes this thread might have.

   Some examples include setting the stack size of perhaps information about the scheduling priority of the thread.

3. The third argument is the most complex, but is really just asking: *which function should this thread start running in?*

   This one tells us the following is expected: a function name(`start_routine`), which is passed a single argument of type `void *`, and which returns a value of type `void *`.

4. The fourth argument, `arg`, is exactly the argument to be passed to the function where the thread begins execution.

   Why do we need these void pointers?

   Having a void pointer as an argument to the function `start_routine` allows us to pass in any type of argument; having it as a return value allows the thread to return any type of result.

Let's look at the figure below.

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

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

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

int main(int argc, char *argv[]) {
    pthread_t p;
    myarg_t args = { 10, 20 };

    int rc = pthread_create(&p, NULL, mythread, &args);
    ...
}
```

Figure 27.1: **Creating a Thread**

Here we create a thread that is passed two arguments, packaged into a single type we define ourselves(`myarg_t`).

The thread, once created, can simply cast its argument to the type it expects and thus unpack the arguments as desired.

27.2 Thread Completion

What happens if we want to wait for a thread to complete?

We must call the routine `pthread_join()`

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

This routine takes two arguments.

1. The first is of type `pthread_t`, and is used to specify which thread to wait for.

   This variable is initialized by the thread creation routine(when we pass a pointer to it as an argument to `pthread_create()`).

2. The second argument is a pointer to the return value we expect to get back.

   Because the routine can return anything, it is defined to return a pointer to void.

Because the `pthread_join()` routine changes the value of the passed in argument, we need to pass in a pointer to that value, not just the value itself.

Let's look at another example.

```
1  typedef struct { int a; int b; } myarg_t;
2  typedef struct { int x; int y; } myret_t;
3
4  void *mythread(void *arg) {
5      myret_t *rvals = Malloc(sizeof(myret_t));
6      rvals->x = 1;
7      rvals->y = 2;
8      return (void *) rvals;
9  }
10
11 int main(int argc, char *argv[]) {
12     pthread_t p;
13     myret_t *rvals;
14     myarg_t args = { 10, 20 };
15     Pthread_create(&p, NULL, mythread, &args);
16     Pthread_join(p, (void **) &rvals);
17     printf("returned %d %d\n", rvals->x, rvals->y);
18     free(rvals);
19     return 0;
20 }
```

Figure 27.2: **Waiting for Thread Completion**

In the code, a single thread is again created, and passed a couple of arguments via the `myarg_t` structure.

To return values, the `myret_t` type is used. Once the thread is finished running, the main thread, which has been waiting inside of the `pthread_join()` routine, then returns and we can access the values returned from the thread.

Often times we don't have to do all of this painful packing and unpacking of arguments.

For example, if we just create a thread with no arguments, we can pass NULL in as an argument when the thread is created.

Similarly, we can pass NULL into `pthread_join()` if we don't care about the return value.

Second, if we are just passing in a single value(e.g. a `long long int`), we don't have to package it up as an argument.

The figure below shows and example.

```
void *mythread(void *arg) {
    long long int value = (long long int) arg;
    printf("%lld\n", value);
    return (void *) (value + 1);
}

int main(int argc, char *argv[]) {
    pthread_t p;
    long long int rvalue;
    Pthread_create(&p, NULL, mythread, (void *) 100);
    Pthread_join(p, (void **) &rvalue);
    printf("returned %lld\n", rvalue);
    return 0;
}
```

Figure 27.3: **Simpler Argument Passing to a Thread**

27.3 Locks

The next most useful set of functions provided by the POSIX threads library are those for providing mutual exclusion to a critical section via locks. The most basic pair of routines to use for this purpose is provided by the following:

```
int pthread_mutex_lock(pthread_mutext_t *mutex);
int pthread_mutex_unlock(pthbread_mutex *mutex);
```

When we have a region of code that is a critical section, and thus needs to be protected to ensure correct operation, locks are quite useful.

We can imagine what the code looks like:

```
pthread_mutex_t lock;
pthread_mutex_lock(&lock);
x = x + 1;
pthread_mutex_unlock(&lock);
```

The intent of the code is as follows: if no other thread holds the lock when `pthread_mutex_lock()` is called, the thread will acquire the lock and enter the critical section.

If another thread does indeed hold the lock, the thread trying to grab the lock will not return from the call until it has acquired the lock(implying that the thread holding the lock has released it via the unlock call).

Unfortunately, this code is broken, in two important ways. The first problem is a lack of proper initialization.

All locks must be properly initialized in order to guarantee that they have the correct values to begin with and thus work as desired when lock and unlock are called.

With POSIX threads, there are two ways to initialize locks. One way to do this is to use `PTHREAD_MUTEXT_INITIALIZER`, as follows:

```
pthread_mutex_t lock = PTHREAD_MUTEX_INITIALIZER;
```

Doing so sets the lock to the default values and thus makes the lock usable.

The dynamic way to do it(i.e. at run time) is to make a call to `pthread_mutex_init()`, as follows:

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

The first argument is the address of the lock itself, the second argument is the attributes of the lock.

The second problem with the code above is that it fails to check error codes when calling lock and unlock.

If our code doesn't properly check error codes, the failure will happen silently, which in this case could allow multiple threads into a critical section.

Minimally, use wrappers, which assert that the routine succeeded, as shown below:

```
// Keeps code clean; only use if exit() OK upon failure
void Pthread_mutex_lock(pthread_mutex_t *mutex) {
    int rc = pthread_mutex_lock(mutex);
    assert(rc == 0);
}
```

Figure 27.4: **An Example Wrapper**