【OS】Day37

Class	Operating System: Three Easy Pieces
Date	@February 16, 2022

【Ch30】Condition Variables

There are many cases where a thread wishes to check whether a condition is true before continuing its execution.

For example, a parent thread might wish to check whether a child thread has completed before continuing.

```
void *child(void *arg) {
    printf("child\n");
    // XXX how to indicate we are done?
    return NULL;
}

int main(int argc, char *argv[]) {
    printf("parent: begin\n");
    pthread_t c;
    Pthread_create(&c, NULL, child, NULL); // create child
    // XXX how to wait for child?
    printf("parent: end\n");
    return 0;
}
```

Figure 30.1: **A Parent Waiting For Its Child**

What we would like to see here is the following output:

```
parent: begin
child
parent: end
```

We could let the parent thread spin wait for the child thread to finish, but this can be super inefficient:

```c
volatile int done = 0;

void *child(void *arg) {
    printf("child\n");
    done = 1;
    return NULL;
}

int main(int argc, char *argv[]) {
    printf("parent: begin\n");
    pthread_t c;
    Pthread_create(&c, NULL, child, NULL); // create child
    while (done == 0)
        ; // spin
    printf("parent: end\n");
    return 0;
}
```

Figure 30.2: **Parent Waiting For Child: Spin-based Approach**

30.1 Definition and Routines

To wait for a condition to become true, a thread can make use of what is known as a condition variable.

A condition variable is an explicit queue that threads can put themselves on when some state of execution(i.e. some condition) is not as desired(by waiting on the condition); some other thread, when it changes said state, can then wake one of those waiting threads and thus allow them to continue.

To declare such a condition variable, one simple writes something like this: `pthread_cond_t c;`, which declares `c` as a condition variable. A condition variable has two operations associated with it: `wait()` and `signal()`. The `wait()` call is executed when a thread wishes to put itself to sleep; the `signal()` call is executed when a thread has changed something in the program and thus wants to wake a sleeping thread waiting on this condition.

```c
pthread_cond_wait(pthread_cond_t *c, pthread_mutex_t *m);
pthread_cond_signal(pthread_cond_t *c);
```

`wait()` call also takes a `mutex` as a parameter; it assumes that this `mutex` is locked when `wait()` is called.

The responsibility of `wait()` is to release the lock and put the calling thread to sleep; when the thread wakes up, it must re-acquire the lock before returning to the caller. This complexity stems from the desire to prevent certain race conditions from occurring when a thread is trying to put itself to sleep.

```
1  int done  = 0;
2  pthread_mutex_t m = PTHREAD_MUTEX_INITIALIZER;
3  pthread_cond_t c  = PTHREAD_COND_INITIALIZER;
4
5  void thr_exit() {
6      Pthread_mutex_lock(&m);
7      done = 1;
8      Pthread_cond_signal(&c);
9      Pthread_mutex_unlock(&m);
10 }
11
12 void *child(void *arg) {
13     printf("child\n");
14     thr_exit();
15     return NULL;
16 }
17
18 void thr_join() {
19     Pthread_mutex_lock(&m);
20     while (done == 0)
21         Pthread_cond_wait(&c, &m);
22     Pthread_mutex_unlock(&m);
23 }
24
25 int main(int argc, char *argv[]) {
26     printf("parent: begin\n");
27     pthread_t p;
28     Pthread_create(&p, NULL, child, NULL);
29     thr_join();
30     printf("parent: end\n");
31     return 0;
32 }
```

Figure 30.3: **Parent Waiting For Child: Use A Condition Variable**

There are two cases to consider:

1. In the first, the parents creates the child thread but continues running itself and thus immediately calls into `thr_join()` to wait for the child thread to complete. In this case, it will acquire the lock, check if the child is done, and put itself to sleep by calling `wait()`.

The child will eventually run, print the message `“child”`, and call `thr_exit()` to wake the parent thread; this code just grabs the lock, sets the state variable done, and signals the parents thus waking it.

Finally, the parent will run, unlock the lock, and print the final message `“parent: end”`.

2. In the second case, the child runs immediately upon creation, sets done to 1, calls signal to wake a sleeping thread(but there is none, so it just returns), and is done. The parent then runs, calls `thr_join()`, sees that `done` is 1, and thus does not wait and returns.

Note: Just an if statement when deciding whether to wait on the condition will suffice our need here. It is always a good idea to use while.