

GEORGIA INSTITUTE OF TECHNOLOGY

College of Computing

CS3431 — Operating Systems and Data Management

Winter 1999

CS3431 Handout #2
Corrected Appendix 1 to Project 1

Issued: January 14, 1999

The “Introduction to pthreads” appendix to Project 1 has a number of bugs. Here’s a replacement version. Changes are marked with a box (“”) in the left margin.

Appendix 1: Introduction to pthreads

This appendix gives a quick introduction to the POSIX standard “pthreads” threads interface for the purposes of this assignment. Although full documentation is available on-line (start with `man threads` on a Sun machine running Solaris, e.g. `acme`), I found the documentation a little difficult to follow initially.

Sun’s Solaris operating system provides a thread system with both a POSIX-compatible (“pthreads”) interface as well as a Solaris-specific interface. The interfaces are compatible, so you are welcome to use one or the other. I found the pthreads interface a little simpler and was also attracted by the fact that, since it’s a standard, I could run the same code on my (Linux-equipped) laptop.

The rest of this introduction describes the thread system in four pieces (1) which files you need to include, (2) how to create threads and wait for them to finish, (3) how to synchronize with mutexes and (4) how to synchronize with more elaborate constructs built on “condition variables”.

1. Files: All multithreaded programs should include the following two lines (define `_REENTRANT` before any include files):

```
#define _REENTRANT
 #include <pthread.h>
```

2. Create and Join: There is a `pthread_create()` call to create a new thread and a `pthread_join()` call to wait for an existing thread to finish. The `pthread_create()` call takes a pointer to an ID (for subsequent reference to the new thread), an optional pointer to an “attributes” structure you can ignore, a pointer to the thread code (obviously), and an optional pointer to arguments to be passed to the new thread. It returns 0 or an error code.

The `pthread_join()` call takes a thread ID and an optional pointer to a place to put the thread’s return value. The idea is that the caller wants to wait for the indicated thread to exit.

Here’s an example of the use of `pthread_create()` and `pthread_join()` (with arguments and return values unused):

```
static void *reader(void *unused_argument)
{
    /* [...] */
}

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

    pthread_create(&reader_id, NULL, reader, NULL);
    /* [...] */
    pthread_join(reader_id, NULL);
}
```

3. Mutexes: Mutual-Exclusion locks or mutexes are supported by pthreads. There are *init*, *lock* and *unlock* calls. The *init* call initializes the mutex to be unlocked and also allows you to apply another bunch of “attributes” that you can ignore. Here’s a simple example:

```
static pthread_mutex_t amutex;

static void *reader(void *unused_argument)
{
    /* [...] */
    pthread_mutex_lock(&amutex);
    /* [...] critical section protected by amutex [...] */
    pthread_mutex_unlock(&amutex);
    /* [...] */
}

int main(int argc, char *argv[])
{
    pthread_mutex_init(&amutex, NULL);
    /* [...] */
}
```

4. Condition Variables: Condition variables allow you to synthesize arbitrarily complex synchronization schemes by associating a user-defined condition with a queue of blocked threads. There are four calls: *init*, *wait* and then two flavors of *signal*. Here's an example:

```
static pthread_mutex_t amutex;
static pthread_cond_t acondition;

static void producer(void)
{
    pthread_mutex_lock(&amutex);
    while (is_my_bounded_buffer_full())      /* check user-defined-condition */
 pthread_cond_wait(&acondition, &amutex); /* block while waiting */
    /* [... critical section protected by amutex ...] */
    pthread_mutex_unlock(&amutex);
}

static void consumer(void)
{
    pthread_mutex_lock(&amutex);
    /* [... critical section protected by amutex ...] */
    pthread_cond_signal(&acondition);      /* tell producer to check again */
    pthread_mutex_unlock(&amutex);
}

int main(int argc, char *argv[])
{
 pthread_mutex_init(&amutex, NULL);
 pthread_cond_init(&acondition, NULL);
    /* [...] */
}
```

Notes:

- A condition variable is always associated with some mutex: calls to *wait*, *signal* or *broadcast* should be made with the mutex locked.
- Just because your call to *wait* returns, that doesn't mean that your user-defined condition has necessarily become true — you have to check your condition again. Therefore, calls to *wait* are always in a loop.
- The other version of *signal* is *broadcast*. The difference is that *signal* wakes up one waiting thread while *broadcast* wakes up all possible waiting threads. If in doubt, use *broadcast*.