Lecture 13 Appendix: More PThreads (Mar 8, 2005) Yap

March 24, 2005

1 Review of POSIX Threads

- We review pthreads so that you can do homework 4.
- The original notes on pthreads are found in Lecture 7. BUT PLEASE RELOAD THAT CHAPTER, AS WE HAVE REVISED/EXTENDED THE PTHREAD MATERIAL.
- The 3 main classes of objects in pthreads are (1) Pthreads, (2) Mutexes, (3) Conditional Variables. There are associated attribute objects for each. Hence we have 6 main classes of objects overall.
- For each kind of object, you have operations to create and to destroy them.

  The creation operation is sometimes called **create** and sometimes called **init**.

  Do not forget to destroy the objects before you exit the program because they represent system resources which are tied up otherwise.

  In the following, the create and destroy operations will be taken for granted.
- Mutex variables are easiest to understand: they guard critical sections to ensure mutual exclusion.

  The two interesting actions on a mutex variable are **lock** and **unlock**.

  Each mutex variable is associated with a queue. When a thread fails to lock a mutex (because someone else is currently holding the lock), that thread is put on the queue. When that lock holder unlocks (releases the lock), one thread from the queue (if non-empty) will be woken up.
- Conditional Variables (CondVars) is the trickiest: it is always used in conjunction with an associated mutex variable.

- The interesting operations on CondVars are **wait**, **signal** and **broadcast**.

  Each CondVar is associated with a queue. When a thread waits on that CondVar, it is put on the queue. When someone signals the CondVar, one thread from the queue (if non-empty) will be woken up. When someone broadcasts the CondVar, all thread from the queue will be woken up.

- Example of Use of Conditional Variables: two procedures called dec() and inc():

```
pthread_mutex_t count_lock;
pthread_cond_t count_nonzero;
unsigned count;

dec()
   { pthread_mutex_lock(&count_lock);

     while (count == 0)
        pthread_cond_wait(&count_nonzero, &count_lock);
     count = count - 1;
     pthread_mutex_unlock(&count_lock);
   }

inc()
   { pthread_mutex_lock(&count_lock);
     if (count == 0)
       pthread_cond_signal(&count_nonzero);
     count = count + 1;
     pthread_mutex_unlock(&count_lock);
    }
```

  Source: David Marshall (www.cs.cf.ac.uk/Dave/C/node31.html)

- Discussion of above code:

  Why the while loop?

  Shouldn't we increment the count variable *before* the if-statement, and change the if-statement to test if count is 1?

- Sharing of Thread Code.

  If more than one thread will run a piece of code, you may need to give each thread its own stack to avoid them interfering with each other!

  For details on how to do this, please read the (updated) Lecture 7.

- Some Suggestions from Chien-I, our grader:

  A simple example:
  http://www.cs.cf.ac.uk/Dave/C/node29.html#SECTION00294000000000000000

  Some of you might use Cygwin to simulate the UNIX system, however, some feature of pthread is NOT availabe in Cygwin, here is a reference:

  http://rustam.uwp.edu/support/faq.html

  Here section 16 is totally about thread management.

  Since `pthread_attr_setstackaddr` is NOT implemented in Cygwin, every thread shared same stack and symbol table. Therefore, if you use "i" as a local variable, it might be changed by other threads.

  To illustrate this, we list two programs below: `th_wrong.c` and `th_array.c`.

  Try compiling and running these programs. The first program is wrong and the second program fixes this by introducing an array of local variables.

```
/************************
/ th_wrong.c
/
/ There is only one i variable,
/ so when Thread 2 initialize i in (*)
/ i in Thread 1 also changed,
/ and will start from 0 again.
/ Note ii was also wrong in this
/ program.
/
/************************/

#include <pthread.h>
#include <stdio.h>
#include <sys/types.h>

void *loop(void *arg)
{
int i,ii;
        ii=*(int *)arg; // ii: thread index, also changed by other threads!
        for(i=0;i<10;i++)    ////////////////////////// (*)
                printf("Thread %d : loop %d\n",ii,i);
printf("Thread %d terminated!\n", ii);
        pthread_exit(NULL);
}

int main(void){
        pthread_t a[10];
```

```
        int i,ret;
        for(i=0;i<10;i++){
                ret=pthread_create(&(a[i]), NULL, loop, (void *)(&i));
                printf("Create %d : %d\n",i ,ret);
// Sleep(1);
        }
        for(i=0;i<10;i++){
                pthread_join(a[i],NULL);
        }
        return 0;
}


/*************************
/ th_array.c
/
/ In this program, the bug
/ was fixed by using i[10]
/ instead of i
/
/*************************/

#include <pthread.h>
#include <stdio.h>
#include <sys/types.h>

void *loop(void *arg)
{
int i[10];
        for(i[*(int *)arg]=0;i[*(int *)arg]<10;i[*(int *)arg]++)
                printf("Thread %d : loop %d\n",*(int *)arg, i[*(int *)arg]);
printf("Thread %d terminate!\n",*(int *)arg);
        pthread_exit(NULL);
}

int main(void){
        pthread_t a[10];
        int i,ret,index[10];
        for(i=0;i<10;i++){
index[i]=i;
                ret=pthread_create(&(a[i]), NULL, loop, (void *)(&index[i]));
                printf("Create %d : %d\n",i ,ret);
        }
        for(i=0;i<10;i++){
                pthread_join(a[i],NULL);
        }
```

```
        return 0;
}
```

- PRAGMATICS OF THREAD PROGRAMMING:

    1. It is suggested that `#include <pthread.h>` is the very first one in your include files. This is to ensure that you use only "thread safe" code for other include files.
    2. In Unix (solaris, linux but not cygwin), when you link your program, be sure to include `-lpthread` to your gcc compiler.
    3. When you have more than one thread running the same thread routine, remember that all automatic variables are NOT safe, unless you have thread-specific stacks.