Lecture 7: DEADLOCKS II; POSIX THREADS (Feb 8, 2005) Yap

March 11, 2005

1 ADMIN

- Hw1 graded and handed back.
- READING: start to read up on deadlocks (CHAPTER 4)
- GENERAL HINT ABOUT READING: at first, read the assigned chapter lightly, to prepare for class. But after my lectures, you should reread the chapter but this time focusing on the topics that I actually lectured on. HENCE attendence in lectures is very important to know where the emphasis is.
- We will be getting a grader this week – watch out for announcements.

2 Review

- Q: State the 4 necessary conditions for a deadlock to occur

  A:

  – Process will WAIT until requested resource is granted

  – Process require EXCLUSIVE use of resources.

  – Resources are NON-PREEMPTABLE

  – CIRCULARITY OF wait among 2 or more processes

3 DEADLOCKS (contd)

- Execution Trajectories: Figure 3-8 [p.176] of text.
- Banker's Algorithm.

  SIMPLE EXAMPLE (1 resource)

	Has	Max
A	0	6
B	0	5
C	0	5
D	0	7
Free	10	

	Has	Max
A	1	6
B	1	5
C	2	5
D	4	7
Free	2	

	Has	Max
A	1	6
B	2	5
C	2	5
D	4	7
Free	1	

Initially, Safe, Unsafe.

- Generalize to multiple resources.

4 POSIX THREADS

The book's treatment of threads in chapter 2 is somewhat confusing because it goes into issues of threading before clearly explaining what it is. To gain a better understanding of this important concept, we are going to look at **POSIX threads** (or "pthreads"). To ensure portability of code, you should try to program only pthreads. Note: POSIX=portable operation system interface (a registered trademark of IEEE)

- Is pthreads available in cygwin?

  A: Yes! The library is found in C:/cygwin/lib/libpthread.a, and the include file in C:/cygwin/usr/include/pthread.h.

- Tutorials? A: There are various tutorials to be found on the web. One such tutorial (which we henceforth call "pthreadTutor") may be found at Lawrence Livermore National Lab (LLNL), at the URL `www.llnl.gov/computing/tutorials/workshops/workshop/pthreads/MAIN.html`

- Books? A: See "Threads Primer: A guide to multithreaded programming" by Bil Lewis and Daniel J. Berg, Sunsoft Press, 1996.

- What are some models for writing threaded programs?

  Your program must break up its task into work that can be done by independent concurrent threads.

  (1) Manager/worker: manager thread assigns work to other worker threads. Typically, manager handles all inputs and parcels out work to workers. (can have static or dynamic worker pool) [Tanenbaum] illustrates this with web server application.

  (2) Pipeline: line an automobile assembly line.

  (3) Peer: the manager thread is like a worker thread after the initial setup.

- How can I use pthread?

  It is only defined for the C language.

  Your C program must include "pthread.h".

Please make sure that your `#include <pthread.h>` statement is the very first of all your include files. REASON: other include files may have "thread-safe" features which are turned on only after pthread.h has been included.

When you compile and link, be sure to tell your linker to use the pthread library. E.g.,

```
gcc -lpthread mythreadprog.c
```

- What is the pthread API?

  There are three main types of objects: threads, mutex variables, condition variables.

  But associated with each main type of objects are corresponding **attribute objects**. Thus we have objects for (resp.) thread attributes, mutex attributes and condition attributes. Some people call them **opaque objects**.

  The API has over 60 subroutines divided into 3 classes of calls:

  (1) Thread management: create, detach, join, etc. set/query thread attributes (joinable, scheduling, etc).

  (2) Mutexes: create, destroy, lock, unlocking mutexes. set/query mutex attributes.

  (3) Condition variables: communication between threads that share a mutex. Create/destroy/wait/signal. Also set/query condition variable attributes.

- Names in the pthread module can be bewildering at first, so remember the following name prefix conventions:

```
pthread_... are threads themselves and misc.subroutines
pthread_attr_ are thread attribute objects
pthread_mutex_ are mutexes
pthread_mutex_attr are mutex attribute objects
pthread_cond_ are condition vars
pthread_cond_attr are condition attribute objects
pthread_key_ are thread-specific data keys
```

  In other words, every method in this API will begin with the prefix `pthread_`.

- Attribute Objects. This is an interesting concept that must be mastered if you want to use pthreads. We focus on thread attributes at first.

  A thread attribute object stores the following information (the default values are in parenthesis):
  Scope (PTHREAD_SCOPE_PROCESS)
  Detached State (PTHREAD_CREATE_JOINABLE)

Stack Address (NULL)
Stack Size (NULL)
Priority (NULL)

When we create a thread, one of the arguments is an attribute object. The attributes are thereby transferred to the thread.

- HOW TO USE ATTRIBUTE OBJECTS:

  1. You must create an attribute object before using it. Do:

     ```
     // declare attr to be attribute variable
     pthread_attr_t   attr;

     // initialize attr
     if (0 != phtread_attr_init(pthread_attr_t *attr))
          perror("initializing pthread attribute");
     ```

     This returns a 0 if successful, else an error number is returned. Now `attr` is an initial object with all the default values.

  2. You can subsequently change the attribute fields. E.g., to set the detached state, do

     `phtread_attr_setdetachstate(pthread_attr_t *attr, int state)`

     where `state=PTHREAD_CREATE_DETACHED` or `state=PTHREAD_CREATE_JOINABLE`.

  3. To see the current value of the detached state, use

     `phtread_attr_getdetachstate(const pthread_attr_t *attr, int *state)`

  4. If you subsequently change the attribute object, this has no effect on the thread which was created earlier with this object.

  5. REMARKS ON STACK: The default is no stack – the stack address and stack size pointers are both NULL in the pthread attribute object.

     When do you need a thread-specific stack? If the thread code is run by more than one thread, you should consider giving your thread its own stack! *BE SURE THAT EACH THREAD HAS A DIFFERENT STACK ADDRESS.*

     To get the thread-specific stack you need to set the stack size and stack address attributes.

  6. SETTING STACK SIZE: You might think that this is easy. But there is a catch: if you set it smaller than some constant `PTHREAD_STACK_MIN`, there will be an error (presumably the default stack size of 0 will be used). Also, I found that `PTHREAD_STACK_MIN`, was not defined in my environment. So I had to define my own value and verify that it is big enough. The following seems good:

```
#define PTHREAD_STACK_MIN 10000
size_t my_stacksize;

if (0!= pthread_attr_setstacksize(&attr, (size_t)PTHREAD_STACK_MIN))
     perror("set stack size");
pthread_attr_getstacksize(&attr, &my_stacksize);
        printf("my stack size = %d \n", (int)my_stacksize);
```

7. SETTING A STACK ADDRESS: Well, you first need to allocate the space with the standard system call "malloc".

   ```
   void * stackaddr;               // stack address pointer
   stackaddr = (void *)malloc((size_t)PTHREAD_STACK_MIN);

   pthread_attr_setstackaddr(&attr, &my_stacksize);
   ```

8. SCOPE NOTE: The scope of the thread can be "process level" (default) or "system level". A process-scope thread is not visible to the kernel, and a system-scope is. E.g., the system does not schedule a process-level thread.

We also say the process is **unbound** when it has has process-scope, and **bound** otherwise. This terminology arise from the fact that bound processes are identified with a LWP (**light weight process**) which are kernel objects capabable of being scheduled.

- OK, I am ready, how do I create a pthread?

  Initially, there is a single **main thread** associated with the main() program. All other threads must be explicitly created by the programmer using: Here is the exact prototype for the call:

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

  Suppose we call: `pthread_create(&thread, &attr, start_routine, (void *)&arg)`.

  A successful call returns a 0. The new thread ID is pointed to by the `thread` argument. The caller can use this ID to perform other operations.

  The `attr` parameter is used to set thread attributes. You can use a "thread attribute object" or NULL to accept the defaults.

  The `start_routine` is a C routine that the thread will execute when it is created. The argument to `start_routine` is a pointer to a void; this argument is the last argument of `pthread_create()`.

The last `arg` is passed to the `start_routine` by reference (recase as a pointer of type void).

- HOW TO TERMINATE THREADS?

  This can happen in several ways:

  – the main (initial) thread returns

  – the thread calls `pthread_exit`

  – it is canceled by another thread via `pthread_cancel`

  – the process is terminated by a call to exit or exec

  Note that the thread_function turns a `void *`. In particular, the routine can call `int pthread_exit(void *status)` to exit, so status contains the return value.

  ```
  int pthread\_exit(void *status)
  ```

  If the main thread finishes using `pthread_exit()` before its descendent threads, the other threads continue running; if `pthread_exit()` is not used, the descendents are terminated automatically.

  The termination "status" is stored as a void pointer which any thread may join. To understand this remark, we must know that every thread will be defined to have either the DETACHED or the JOINABLE state (but not both). If DETACHED, all its resources and exit status are discarded when the thread terminates. If JOINABLE, this information will be retained until it is "joined" to another thread (and thereby available to this thread). The default state is JOINABLE.

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

  CLEANUP: `pthread_exit()` does not close files...

  SUGGESTION: Use `pthread_exit()` to exit from all threads, esp. main().

- How about a sample code?

  ```
  #include <pthread.h>
  #include <stdio.h>
  #define NUM_THREADS 5

  void *PrintHello(void *threadid) {
     printf("\n%d: hello world!\n", threadid);
     pthread_exit(NULL);
  }

  int main (int argc, char * argv[]) {
  ```

```
   pthread_t threads[NUM_THREADS];
   int rc, t;
   for (t=0; t<NUM_THREADS; t++) {
      printf("Creating thread %d\n", t);
      rc = pthread_create( &threads[t], NULL, PrintHello, (void *)t);
      if (rc){
         printf("ERROR; return code of pthread_create() is %d\n", rc);
 exit(-1);
      }
   }
   pthread_exit(NULL);
}
```

- Mutexes. Let us consider next the mutex synchronization objects. Each mutex is essentially an int variable that are access in a very stylized way: either **lock** or **unlock**. Let L and U be the number of lock and unlock operations on a mutex variable. If $L > U$, the mutex is in the "lock state", and exactly $L - U$ of the processes that performed the lock operation will be blocked. Any "unlock" operation will unblock one of these processes.

  ```
  int
  pthread_mutex_init ( pthread_mutex_t *mut, const pthread_mutexattr_t *attr);
      -- creates a mutex variable
  int
  pthread_mutex_lock ( pthread_mutex_t *mut);
      -- locks a mutex variable
  int
  pthread_mutex_unlock ( pthread_mutex_t *mut);
      -- unlocks a mutex variable
  int
  pthread_mutex_trylock ( pthread_mutex_t *mut);
      -- either acquires a mutex variable or returns EBUSY.
  int
  pthread_mutex_destroy ( pthread_mutex_t *mut);
  ```

- Condition Variables. Condition variables allows you to wait for some predicate to hold on the condition variables.

  It is always used in conjunction with an associated mutex variable.

  There are two main operations on a cond.variable: wait and signal.

  USAGE: this usage is very stylized (in some sense, the concept of cond.variables is not a primitive concept). Let "cond" and "mut" be the condition var. and associated mutex var.
  – HOW TO WAIT: thread locks "mut" then then waits on "cond" AND "mut". When wait returns, it should unlock "mut". [REMARK: while

waiting, ”mut” is actually unlocked by the wait call; it is locked again upon return from wait call.]
– HOW TO SIGNAL: thread locks ”mut” and then signals ”cond”. When done, it unlocks ”mut”.

```
1. Initialisation:
int
pthread_cond_init (pthread_cond_t *cond, pthread_condattr_t *attr);

-- as usual, *attr can be NULL.
2. Waiting:
int
pthread_cond_wait (pthread_cond_t *cond, pthread_mutex_t *mut);
-- caller thread always blocks.
-- NOTE that a mutex variable (mut) is required.
-- You MUST lock "mut" before this call, and unlock "mut" after.
-- This call is equivalent to:
a. pthread_mutex_unlock (mut);
b. block_on_cond (cond);
c. pthread_mutex_lock (mut);
-- Note that c. requires the mutex lock to be acquired.
3.  Signalling:
int
pthread_cond_signal (pthread_cond_t *cond);
-- this wakes up (at least?) one thread waiting on cond.
["At least" seems wrong: I think it should be "at most".
E.g., if there are no threads waiting on cond., then
nothing is woken up, according to description of signal.
If there are many, then I think you want to wake
up only one.  OTHERWISE you should use "broadcast"
instead of "signal".]
-- because of step c. above, the threads will exit
the block, one at a time.
-- You MUST lock "mut" before this call, and unlock "mut" after.
4.  Broadcast Signalling:
int
pthread_cond_broadcast (pthread_cond_t *cond);
-- wakes up all threads blocked on cond.
5.  Waiting with timeout:
int
pthread_cond_timedwait (pthread_cond_t *cond, pthread_mutex_t *mut,
                        const struct timespec *abstime);
-- similar to wait, but with timeout (absolute time of day)
6.  Deallocation:
int
pthread_cond_destroy (pthread_cond_t *cond);
```

- Example of Use of Conditional Variables

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

decrement_count()
   { pthread_mutex_lock(&count_lock);

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

increment_count()
   { 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)

- Other useful functions:

  `pthread_t pthread_self()` returns the its own thread id.