Workshop on POSIX Threads and the Problem of Deadlock

1 Background

- POSIX is a standard for Unix
- Linux implements POSIX threads
- On Red Hat 8.x, documentation is at

  ```
  $ info '(libc) POSIX Threads'
  ```

  - or in Emacs, `C-H m libc` then middle-click on `POSIX threads`
- Provides:
  - *semaphores*,
  - *mutex*es and
  - *condition variables*

  for locking (synchronisation)

2 Generic Procedure for Compiling POSIX Threads Applications

1. You need to use the `libpthread` library
   - Specify this with the option `-lpthread`
2. Need to tell the other libraries that they should be *reentrant* (or "*thread safe*")
   - This means that the library uses no static variables that may be overwritten by another thread
   - Specify this with the option `-D_REENTRANT`
3. So, to compile the program ⟨*program*⟩.c, do:

   ```
   $ gcc -D_REENTRANT -lpthread -o ⟨program⟩ ⟨program⟩.c
   ```

3 Procedure

1. Download the source code for these programs from the subject web site: `hello.c`, `deadlock.c` and `error.h`.

Program 1 A simple program `hello.c` that is the program `hello.c` given in the lecture.

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

void *print_hello( void *threadid )
{
        printf( "\n%d: Hello World!\n", ( int ) threadid );
        pthread_exit( NULL );
}

int main()
{
        static 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, print_hello, ( void * ) t );
                if ( rc ) {
                        printf( "ERROR; pthread_create() returned %d\n", rc );
                        printf( "Error string: \"%s\"\n", strerror( rc ) );
                        exit( -1 );
                }
        }
        pthread_exit( NULL );
}
```

3.1 An Introduction to Posix Threads

1. Compile and run program 1, using the generic instructions given in section 2 on the previous page.
2. Refer to the lecture notes on the topic *Processes and Threads* and read the few slides starting at slide **??**. Read about the four parameters that are passed to `pthread_create()`. Note that there is a detailed manual page for every single library function used here. For example, you can see `man pthread_create`, and `man strerror`, `man 3 exit`, `man 3 printf`, and so on.
3. Modify `hello.c`, increasing the number of threads until you find the maximum number of threads that your system can support.
4. Download the program file `hello-with-logs.c`, which makes each thread do CPU intensive calculations. Observe the output of `vmstat 1` and also, run `top` in another window. Watch the load average.*

**Load Average* is the average number of processes that are in the *ready-to-run* state. In other words, the number of additional processes that would be running if each had a CPU.

Program 2 Pseudocode for the two threads.

deadlock	*no mutual exclusion*	*no hold & wait*	*no circular wait*

Thread a()
{
    lock mutex1;
    give up CPU;
    lock mutex2;
    unlock mutex1;
    unlock mutex2;
}
Thread b()
{
    lock mutex2;
    give up CPU;
    lock mutex1;
    unlock mutex2;
    unlock mutex1;
}			

3.2 Deadlock

1. Compile and execute the program `deadlock.c` shown in program 3 on page 5 and program 6. Observe what happens when you run it.

2. Read about *mutexes* in slide **??** in the lecture notes on the topic *Processes and Threads*. Also read slides **??**–**??** showing example code using a mutex.

3. Notice that most of the code of program 3 on page 5 is `printf()` statements, error checking and such. The code for the two threads can be expressed quite simply, as shown in program 2. Examine the pseudocode in program 2 and compare it with the C program 3, and see how the pseudocode is a summary of the C program.

4. The standard POSIX library function `sched_yield()` allows a thread or process to voluntarily give up the CPU, so that the scheduler will move it to the end of the queue for its own priority, and allow another thread or process to run. See `man sched_yield`.

5. Note that there are four conditions required for deadlock; if you remove *any* one of these, then deadlock will not happen. They are:

    - Mutual exclusion
        - where only one process can use a resource at one time
    - Hold and Wait
        - Processes holding resources given earlier can request new resources
    - No Preemption
        - Resources given to a process or thread cannot be taken away forcibly by OS or anything other than that process or thread
    - Circular Wait
        - Each process is waiting for a resource held by another

6. Describe here how this program satisfies the deadlock requirement of *mutual exclusion*

✐

7. Describe here how this program satisfies the deadlock requirement of *hold and wait*

✐

8. Describe here how this program satisfies the deadlock requirement of *no preemption*

✐

9. Describe here how this program satisfies the deadlock requirement of *circular wait*

✐

10. Copy the original program (`deadlock.c`) to a new file name, and make simple changes to it to remove at least one of the conditions required for deadlock.

> Note that there are *many* ways of solving this problem, not just one or two. Look for as many as you can. Use simple logic rather than spending a lot of time reading the manuals for new POSIX threads library functions. First, simply try rearranging the pseudocode in program 2 on the previous page.

- Do this a number of times, using a different method for each program.
- Put a comment at the top of the program, indicating which conditions for deadlock you have removed.
- Put your name and class in a comment at the top of each file.
- Demonstrate to your lab supervisor.
- Write a text file containing your answers to questions 6, 7, 8 and 9.
- Use the `tar` or `zip` program to combine the sources and the text file mentioned above into one file, and
- Submit to the online submission system at `http://nicku.org/perl2/submit.cgi`.

Program 3 The first part of the program `deadlock.c` that is unable to run to completion.

```
#include <pthread.h>
#include <stdio.h>
#include "errors.h"

/* Initialize 2 mutexes. */
pthread_mutex_t mutex1 = PTHREAD_MUTEX_INITIALIZER;
pthread_mutex_t mutex2 = PTHREAD_MUTEX_INITIALIZER;

void *locking_thread_a( void *arg )
{
        int status;
        printf( "locking_thread_a starting\n" );
        status = pthread_mutex_lock( &mutex1 );
        if ( status != 0 )
                err_abort( status, "lock 1" );
        printf( "a has lock 1\n" );
        sched_yield();
        printf( "a now trying to get lock 2\n" );
        status = pthread_mutex_lock( &mutex2 );
        printf( "a has lock 2\n" );
        if ( status != 0 )
                err_abort( status, "lock 2" );
        pthread_mutex_unlock( &mutex1 );
        pthread_mutex_unlock( &mutex2 );
        printf( "locking_thread_a finishing\n" );
        pthread_exit( NULL );
}

void *locking_thread_b( void *arg )
{
        int status;
        printf( "locking_thread_b starting\n" );
        status = pthread_mutex_lock( &mutex2 );
        if ( status != 0 )
                err_abort( status, "lock 2" );
        printf( "b has lock 2\n" );
        sched_yield();
        printf( "b now trying to get lock 1\n" );
        status = pthread_mutex_lock( &mutex1 );
        if ( status != 0 )
                err_abort( status, "lock 1" );
        printf( "b has lock 1\n" );
        pthread_mutex_unlock( &mutex2 );
        pthread_mutex_unlock( &mutex1 );
        printf( "locking_thread_b finishing\n" );
        pthread_exit( NULL );
}
```

Program 4 This method is based on `backoff.c`; see section 3.3 on page 8.

```
#include <pthread.h>
#include <stdio.h>
#include "errors.h"

/* Initialize 2 mutexes. */
pthread_mutex_t mutex1 = PTHREAD_MUTEX_INITIALIZER;
pthread_mutex_t mutex2 = PTHREAD_MUTEX_INITIALIZER;

void *locking_thread_a( void *arg )
{
        printf( "locking_thread_a starting\n" );
        for (;;) {
                int status;
                status = pthread_mutex_lock( &mutex1 );
                if ( status != 0 )
                        err_abort( status, "lock 1" );
                printf( "a has lock 1\n" );
                sched_yield();
                printf( "a now trying to get lock 2\n" );
                status = pthread_mutex_trylock( &mutex2 );
                if ( status == 0 )
                        break;
                printf( "a failed to lock; backing off\n" );
                pthread_mutex_unlock( &mutex1 );
                sched_yield();
        }
        printf( "a has lock 2\n" );
        pthread_mutex_unlock( &mutex1 );
        pthread_mutex_unlock( &mutex2 );
        printf( "locking_thread_a finishing\n" );
        pthread_exit( NULL );
}

void *locking_thread_b( void *arg )
{
        int status;
        printf( "locking_thread_b starting\n" );
        status = pthread_mutex_lock( &mutex2 );
        if ( status != 0 )
                err_abort( status, "lock 2" );
        printf( "b has lock 2\n" );
        sched_yield();
        printf( "b now trying to get lock 1\n" );
        status = pthread_mutex_lock( &mutex1 );
        if ( status != 0 )
                err_abort( status, "lock 1" );
        printf( "b has lock 1\n" );
        pthread_mutex_unlock( &mutex2 );
        pthread_mutex_unlock( &mutex1 );
        printf( "locking_thread_b finishing\n" );
        pthread_exit( NULL );
}
```

Program 5 By re-ordering the reequest for resources, we can eliminate the condition of circular wait. We simply request the resources in the same order.

```
#include <pthread.h>
#include <stdio.h>
#include "errors.h"

/* Initialize 2 mutexes. */
pthread_mutex_t mutex1 = PTHREAD_MUTEX_INITIALIZER;
pthread_mutex_t mutex2 = PTHREAD_MUTEX_INITIALIZER;

void *locking_thread_a( void *arg )
{
        int status;
        printf( "locking_thread_a starting\n" );
        status = pthread_mutex_lock( &mutex1 );
        if ( status != 0 )
                err_abort( status, "lock 1" );
        printf( "a has lock 1\n" );
        sched_yield();
        printf( "a now trying to get lock 2\n" );
        status = pthread_mutex_lock( &mutex2 );
        printf( "a has lock 2\n" );
        if ( status != 0 )
                err_abort( status, "lock 2" );
        pthread_mutex_unlock( &mutex1 );
        pthread_mutex_unlock( &mutex2 );
        printf( "locking_thread_a finishing\n" );
        pthread_exit( NULL );
}

void *locking_thread_b( void *arg )
{
        int status;
        printf( "locking_thread_b starting\n" );
        status = pthread_mutex_lock( &mutex1 );
        if ( status != 0 )
                err_abort( status, "lock 1" );
        printf( "a has lock 1\n" );
        sched_yield();
        printf( "a now trying to get lock 2\n" );
        status = pthread_mutex_lock( &mutex2 );
        printf( "a has lock 2\n" );
        if ( status != 0 )
                err_abort( status, "lock 2" );
        pthread_mutex_unlock( &mutex1 );
        pthread_mutex_unlock( &mutex2 );
        printf( "locking_thread_b finishing\n" );
        pthread_exit( NULL );
}
```

Program 6 The second part of the program `deadlock.c` that is unable to run to completion.

```
int main()
{
        pthread_t thread1, thread2;
        int status;
        printf( "Creating thread a\n" );
        status = pthread_create( &thread1, NULL, locking_thread_a, NULL );
        if ( status )
                err_abort( status, "locking_thread_a" );
        printf( "Creating thread b\n" );
        status = pthread_create( &thread2, NULL, locking_thread_b, NULL );
        if ( status )
                err_abort( status, "locking_thread_b" );
        pthread_exit( NULL );
}
```

3.3 Some Other Resources

Some students wanted to know how you check if a mutex is locked without the calling thread going to sleep if the mutex is already locked by another thread. The answer is the POSIX threads library function:

```
int pthread_mutex_trylock( pthread_mutex_t *MUTEX );
```

which immediately returns the value `EBUSY` if the mutex is locked. You can see an interesting example from the file `backoff.c`, shown on pages 66–69 of Butenhof (see [But97] below). You can read `backoff.c` from the subject web site at `http://nicku.org/ossi/lectures/processes/programming-posix-threads/backoff.c`.

References

There are many good sources of information in the library and on the Web about processes and threads. Here are some I recommend:

[tut] `http://www.llnl.gov/computing/tutorials/workshops/workshop/pthreads/MAIN.html` gives a good online tutorial about POSIX threads.

[links] `http://www.humanfactor.com/pthreads/` provides links to a lot of information about POSIX threads

[But97] The best book about POSIX threads is *Programming with POSIX Threads*, David Butenhof, Addison-Wesley, May 1997. Even though it was written so long ago, David wrote much of the POSIX threads standard, so it really is the definitive work. It made me laugh, too!

[Nut02] *Operating Systems: A Modern Perspective: Lab Update*, 2nd Edition, Gary Nutt, Addison-Wesley, 2002. A nice text book that emphasises the practical (like I do!)