Assignment 2: Concurrency and IPC in Threads Packages and xv6 Threads

TOTAL MARKS: 100

Introduction

For this assignment, you are going to use UBC pthreads and RT Threads to implement synchronization applications and primitives. Also, you are going to continue to examine the source code for xv6 and make some additions to its functionality. Please take note that there are three parts to this assignment, which are **unequally weighted**. It would be a mistake to divide up the work between the two members such that, for example, one did part A and C and one did part B. Not only will this not divide the work evenly, you will be at a disadvantage when it comes to the exams. The first phase has 10 marks associated with the makefile and skeleton.
10 marks for Phase 1, 90 for phase 2.

Part A: Implement Synchronization Primitives/Applications in xv6 (30 marks)

In this part of the assignment you will implement synchronization primitives (semaphores) for xv6 and use them to solve a synchronization problem.

Note: Write readable code!

In your programming assignments, you are expected to write well-documented, readable code. There are a variety of reasons to strive for clear and readable code. Since you will be working in pairs, it will be important for you to be able to read your partner's code. Also, since you will be working on xv6 for the remainder of the semester, you may need to read and understand code that you wrote several months earlier. Finally, clear, well-commented code makes your markers happy!

There is no single right way to organize and document your code. It is not our intent to dictate a particular coding style for this class. The best way to learn about writing readable code is to read other people's code. Read the xv6 code, read your partner's code, read the source code of some freely available operating system. When you read someone else's code, note what you like and what you don't like. Pay close attention to the lines of comments which most clearly and efficiently explain what is going on. When you write code yourself, keep these observations in mind.

Here are some general tips for writing better code:

- Group related items together, whether they are variable declarations, lines of code, or functions.
- Use descriptive names for variables and procedures. Be consistent with this throughout the program.
- Comments should describe the programmer's intent, not the actual mechanics of the code. A comment which says "Find a free disk block" is much more informative than one that says "Find first non-zero element of array."

For the purposes of the grader, mark any changes you make to the code with the following comments:

```
/* CMPT 332 GROUP XX Change, Fall 2024 */
```

Concurrent Programming with xv6

For this problem you will be adding more functionality to xv6 that

1. Supports *user-level threading*, so that concurrency within a single user-level process is possible, and
2. Provide a *basic synchronization mechanism* -- the mutex lock -- for said threads to use.

Threading API

In this part of the assignment, you will design the context switch mechanism for a user-level threading system, and then implement it. To get you started, your xv6 has two files `user/uthread.c` and `user/uthread_switch.S`, and a rule in the Makefile to build a uthread program. `uthread.c` contains most of a user-level threading package, and code for three simple test threads. The threading package is missing some of the code to create a thread and to switch between threads.

You will properly complete the following function to support thread creation and subsequently context switching:

```
void thread_create(void (*tmain)());
```

- This creates a new thread that starts execution at the function `tmain`. The `thread_create` function is responsible for allocating (using `malloc`, given in the file "umalloc.c") a new user stack.

In `user/uthread.c`, you will need to add to `thread_schedule` to switch between threads. That switching happens in `thread_switch` in `user/uthread_switch.S`. You will have to decide where to save/restore registers; modifying `struct thread` to hold registers is a good plan. You'll need to add a call to `thread_switch` in `thread_schedule`; you can pass whatever arguments you need to `thread_switch`, but the intent is to switch from thread `t` to `next_thread`.

To allow compilation and execution you will have to add new lines to the makefile:

- Add `$U/_uthread\` to the list of user programs
- Add the following between the list of user programs and the `fs.img` rule:

```
$U/uthread_switch.o : $U/uthread_switch.S
    $(CC) $(CFLAGS) -c -o $U/uthread_switch.o $U/uthread_switch.S

$U/_uthread: $U/uthread.o $U/uthread_switch.o $(ULIB)
    $(LD) $(LDFLAGS) -N -e main -Ttext 0 -o $U/_uthread $U/uthread.o $U/uthread_switch.o $(ULIB)
    $(OBJDUMP) -S $U/_uthread > $U/uthread.asm
```

Mutex

To allow your newly minted threads to coordinate critical sections and safely share data, you will implement the following functions that expose a mutex lock. The lock will be implemented internally using a spinlock.

```
int mtx_create(int locked);
```

- Creates a mutex lock and returns an opaque ID (you might wish to `typedef` a separate type for the lock ID, though it isn't strictly necessary). The lock starts out in the `locked` state (binary: true or false).

```
int mtx_lock(int lock_id);
```

- Blocks until the lock has been acquired.

```
int mtx_unlock(int lock_id);
```

- Releases the lock, potentially unblocking a waiting thread.

Note that you can then build arbitrarily more sophisticated synchronization mechanisms (e.g., semaphores) in userspace by leveraging this mutex lock. This is not required for this assignment.

Testing

To test your code, you should write a program that implements the producer/consumer problem by sharing a bounded buffer between a parent and child (or sibling) threads, using the mutex as a primitive for implementing synchronization. You can implement this without counting semaphores. Just have the buffer size and buffer space as integers that are atomically updated.

Add this program to your codebase and the Makefile, and be sure to commit and push it as part of your submission.

Part B: Implement Monitors with UBC Pthreads Semaphores (20 marks) plus 5 in phase 1

This library utility is to provide the functions of a monitor facility. In general, this would require you to implement condition variables (`k` condition variables must be implemented, `k` specified as a compile-time constant), and the wait and signal operations. In keeping with the standard terminology, these operations would be called **MonWait()**, **MonSignal()**. Monitors can be implemented using either semaphores, or traditional IPC mechanisms in language environments that do not provide this facility, and they provide an elegant programming model that does not require as much awareness of the low level support facilities. In this assignment, we want you to implement Monitors using only the P(), and V() operations.

Implementing the Monitor Utilities Library.

In an environment that supports Monitors internally or as an object, simply placing the procedure inside a monitor object is sufficient to automatically get mutual exclusion. Since we are expecting that applications for this Monitor facility will be written in C, we have to fake this part. This is done by requiring each monitor procedure to execute **MonEnter()** as its first executable statement, and **MonLeave()** as its final statement before returning.

You will need to implement 4 main procedures: **MonEnter(), MonLeave(), MonWait(),** and **MonSignal().** You will need to use lists from Assignment 1 in order to make this work properly. You *should* be using your list implementation from Assignment 1. If you are not confident in your list implementation, please *do not* make a new one. Borrow from a classmate (with attribution and their permission). I know there are many good implementations around. Then, this code should be bundled up as a library. Create a file called `libMonitor.a` as the last step of your make process for making the library.

To test the monitor facility, use the readers-writers problem. We will supply an initial version of the readers-writers problem solution using these facilities. You can also use this solution as a guide to how to structure more complex threads applications.

Part C : Chat program using RT Threads IPC (40 marks) + 5 in phase 1

For this part of the assignment you are going to create a simple "chat"-like facility that enables someone at one terminal to communicate with another user at another terminal using the UDP transport protocol and the RT Threads package.

To initiate an s-chat session, the users will first agree on two things:

- the machine that each will be running on, and
- the port number each will use.

Say that Fred and Barney want to talk. Fred is on machine "minion12" and will use port number 30001. Barney is on machine "peon34" and will use port number 39001.

To initiate s-chat, Fred must type:
  s-chat 30001 peon34 39001

  And Barney must type:
  s-chat 39001 minion12 30001.

The result will be that every **line** typed at each terminal will appear on all terminals as they are typed along with the time at which it was sent from the originator of the message in seconds and microseconds. TIME MAY NOT BE SENT AS A STRING. If you want to learn about "curses" and "cbreak" on your own, you can alter this slightly so that every character typed appears on all screens as it is typed, for each user rather than having to wait for each [return]. If you are interested, look in the man pages under "curses" (this is NOT a requirement of the assignment, but will garner BONUS credit).

A user leaves a session by typing some special character on a line by itself (or some special sequence that you define by itself on a line).

In our configuration in the lab, **only ports 30001 to 40000 are available** for your use. Tuxworld is probably a good partner machine, but again be sure that you connect to the correct tux machine.

Expected Process Structure

This part of the assignment will be done using S/R/R IPC for coordination. I want you to use 5 threads. One of the threads does nothing other than await input from the keyboard. Another thread does nothing other than await a UDP datagram. There will also be threads which print characters to the screen, and which send data to the remote UNIX processes over the network using UDP. There will be a 5th thread (the Server) that coordinates the sending of messages. It processes the messages by putting messages from the keyboard thread on a LIST that are to be transmitted by the UDP transmit thread. The transmit thread requests messages from the Server and sends them across the network.

Again, you will need to use lists from Assignment 1 in order to make this work properly. Access to the list is a critical section. Imagine, if you will, a slow screen, or slow network. A user could type many lines before they get sent across the network. These lines must be buffered in some way. Thus, there is a list of all the messages in transit. The network sending thread would take an item off the list and send it, but if there were no items on the list, it will have to wait. Similarly with the screen printing thread. The server handles access to both lists.

Note that the UDP input and keyboard input threads cannot make any blocking UNIX calls as these would block the entire UNIX process (i.e. all the threads). You will therefore have to perform non-blocking input on the keyboard and network reads. This can be done by setting the respective file descriptors to non-blocking status (see below).

Important Note: If you do not set the file descriptor back to blocking before ending the application, the window from which you invoked the application may disappear. That would be bad.

How to start this part:

First, try out some of the keyboard/screen I/O primitives supplied by UNIX and try reading from the keyboard without blocking (check the section 2 man pages under "read", "sendto" and (optionally) "curses" "cbreak"). To do so, you can either use "select" (look at the man page) or better yet (i.e. easier to get to work), open your file descriptor using the O_NDELAY or O_NONBLOCK flag for non-blocking I/O (again, look at the man pages - this time under "open", and "read").

Check the man pages (or equivalently, many guides to network programming available. I like Beej's Guide to Network Programming listed on the Useful OS Resources page):

- fcntl
- socket
- bind
- sendto
- recvfrom
- getaddrinfo
- htons, htonl, ntohs, ntohl

I'm sure I haven't answered all the questions. Some questions are left to be your judgment. The TAs and I will gladly give help on conceptual questions and hints on implementation issues, you just have to ask.

Finally, bring them all together in a RT Threads application. You will see that the example could have a user on LINUX talking to a user on a POWER architecture (or possibly NetBSD, or another CPU architecture, like an ARM or Motorola processor, which may have different byte orders). In particular, you should test your code on the Linux boxes in the lab as well as on machines of another architecture. We have two machines available for testing different architectures: `cmpt332-arm` and `cmpt332-ppc`.

There should be no version-specific code in your solution. The only changes between platforms will be in the makefile.

Extra credit

Part A (15 marks): Implement counting semaphores using the mutex primitive.
Part B (10 marks): Implement the Dining Philosopher's problem in your monitor, in addition to the Readers-Writers problem.
Part C (15 marks): Implement multi-person chat, including the fancy screen manipulation to divide screen area among the various users using curses.

Testing and Test Plans

Due to the nature of the programs, providing 'canned' test data to these programs will not work, neither will the answers be able to be verified by running a 'diff' of expected output vs. actual output. The testing that you should do would be white-box testing from inside the various features. Turn off the testing features when handing in the assignment. A very valuable way of testing your implementations is to test your chat programs by chatting with another implementation from other groups.

Deliverables - What to hand in:

PHASE 1 (Parts B and C only):

Skeletons/Stubs are code pieces that have the interface to every procedure you have designed, and every thread created, checking the parameters, when appropriate.

Each thread should call the procedures it needs to call, but does not need to be fully functional. Error codes should be checked from system calls, but the logic of the program design can be incomplete.

Part B: Monitors using UBC Pthreads Semaphores

- Documentation describing your solution strategy, including pseudocode in PartB.design.txt. Explain why your solution is correct.
- Monitor.h
- Monitor.c
- reader_writer_monitor.h (can be deduced from the following 2 files)
- reader_writer_monitor.c (provided for you).
- readers_writers.c (provided for you, but you can modify it as you see fit, as long as you document this clearly)
- all the source code for the list library (`list_adders.c, list_movers.c, list_removers.c, list.h`, properly attributed if necessary)
- lines in the makefile to compile the application as `reader_writer_test`, using `libMonitor.a` and `liblist.a` as appropriate (that means `-lnameoflibrary`).

Part C: S-chat using RT Threads IPC

- a single C file (called `s-chat.c`)
- documentation describing your solution strategy, down to the level of general pseudocode.
- lines in the makefile for s-chat

PHASE 2:

Part A: Threads/Synchronization in xv6

- `xv6-synch.diff`: a diff file listing all the changes to the kernel you made for this assignment (As in A1).
- tarball of your kernel source, without the git details (named xv6-A2.tar)
- Design document in PartA.design.txt. Brief description of the implementation of thread implementation and mutex locks and enough explanation for your synchronization tests.
- Test execution of your modified kernel in `xv6-kernel-output.txt.` This will include runs of the producer-consumer problem.

Part B: Monitors using UBC Pthreads Semaphores

Fully functional components of the code and updated documentation, including test results.

Part C: S-chat using RT Tthreads IPC

- a single C file (called `s-chat.c`)
- whitebox test-plan and test results. Show that your program works and why. Paths through the code should be verified.
- documentation describing your solution strategy, down to the level of general pseudocode.
- lines in the makefile for s-chat for 3 architectures: x86_64, ppc and arm

Source code and lines in the makefile for the list library are also required.

No ._ files from OS X.

Finally, proper git logs are required.

All of this should be bundled in a .tar file and submitted via Canvas with no subdirectories in your source code for parts B and C. For Part A, the tar file of the main source tree for xv6 source code, will be the xv6-riscv directory (named xv6-A2.tar), following the same methodology as in A1. So, yes, that's a .tar file that contains a .tar file.

Notes and Warnings:

Documentation: All documentation files are to be in UNIX text file format with lines shorter than 80 characters. Nothing else is acceptable; nothing else is necessary. There will not be diagrams, etc. in your work. This allows the marker to look at both your code and your documentation in the same tool: vi or emacs. It speeds the marking and a happy marker is a generous marker.