Assignment 1: Multi-resource Synchronisation

CSE3211: Operating Systems

Version 1.0
Due Time & Date: Monday, September 24, Time: 23:59:59

Contents

- Introduction
- Setting Up
- Begin Your Assignment
- Concurrent Programming with OS/161
- Reading Exercises
- Coding Assignment
- Generating Your Assignment Submission
- Plagiarism

Introduction

In this assignment you will solve a number of synchronization and locking problems. You will also get experience with data structure and resource management issues.

Please complete the reading exercises for your lecture 8 and 9.

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. Code that is understandable to others is a requirement for any real-world programmer, not to mention the fact that after enough time, you will be in the shoes of one of the *others* when attempting to understand what you wrote in the past. Finally, clear, concise, well-commented code makes it easier for the assignment marker to award you marks! (This is especially important if you can't get the assignment running. If you can't figure out what is going on, how do you expect us to).

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, for example OS/161. 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:

- Split large functions. If a function spans multiple pages, it is probably too long.
- 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."

Setting Up Assignment 1

download from the course website
do the following after download "asst1.zip" in your home directory(only once).

```
% tar zxvf asst1.zip
% cd asst1/src
% ./configure
```

Begin Your Assignment

Configure OS/161 for Assignment 1

Before proceeding further, configure your new sources.

```
% cd ~/asst1/src
% ./configure --ostree=~/os161/root
```

We have provided you with a framework to run your solutions for ASST1. This framework consists of driver code (found in kern/asst1) and menu items you can use to execute your solutions from the OS/161 kernel boot menu.

You have to reconfigure your kernel before you can use this framework. The procedure for configuring a kernel is the same as in ASST0, except you will use the ASST1 configuration file:

```
% cd ~/asst1/src/kern/conf
% ./config ASST1
```

You should now see an ASST1 directory in the compile directory.

Building for ASST1

When you built OS/161 for ASST0, you ran make from compile/ASST0 . In ASST1, you run make from (you guessed it) compile/ASST1 .

```
% cd ../compile/ASST1
% make depend
% make
% make install
```

If you are told that the compile/ASST1 directory does not exist, make sure you ran config for ASST1. Run the resulting kernel:

```
% cd ~/cs161/root
% ./sys161 kernel
sys161: System/161 release 2.0.8, compiled Jul  9 2018 11:30:58

OS/161 base system version 2.0.3
Copyright (c) 2000, 2001-2005, 2008-2011, 2013, 2014
   President and Fellows of Harvard College.  All rights reserved.

Put-your-group-name-here's system version 0 (ASST1 #1)

352k physical memory available
Device probe...
lamebus0 (system main bus)
emu0 at lamebus0
ltrace0 at lamebus0
```

```
ltimer0 at lamebus0
beep0 at ltimer0
rtclock0 at ltimer0
lrandom0 at lamebus0
random0 at lrandom0
lhd0 at lamebus0
lhd1 at lamebus0
lser0 at lamebus0
con0 at lser0

cpu0: MIPS/161 (System/161 2.x) features 0x0
OS/161 kernel [? for menu]:
```

Command Line Arguments to OS/161

Your solutions to ASST1 will be tested by running OS/161 with command line arguments that correspond to the menu options in the OS/161 boot menu.

IMPORTANT: Please DO NOT change these menu option strings!

Here are some examples of using command line args to select OS/161 menu items:

```
sys161 kernel "at;bt;q"
```

This is the same as starting up with `sys161 kernel`, then running "at" at the menu prompt (invoking the array test), then when that finishes running "bt" (bitmap test), then quitting by typing "q".

```
sys161 kernel "q"
```

This is the simplest example. This will start the kernel up, then quit as soon as it's finished booting. Try it yourself with other menu commands. Remember that the commands must be separated by semicolons (";").

"Physical" Memory

HEADS UP!!!! Make sure you do the following Failing to do so will potentially lead to subtle problems that will be very difficult to diagnose.

In order to execute the tests in this assignment, you will need more than the 512 KB of memory configured into System/161 by default. We suggest that you allocate at least 2MB of RAM to System/161. This configuration option is passed to the busctl device with the ramsize parameter in your `~/cs161/root/sys161.conf` file. Make sure the busctl device line looks like the following:

```
31 busctl ramsize=2097152
```

Note: 2097152 bytes is 2MB.

Concurrent Programming with OS/161

If your code is properly synchronised, the timing of context switches and the order in which threads run should not change the behaviour of your solution. Of course, your threads may print messages in different orders, but you should be able to easily verify that they follow all of the constraints applied to them and that they do not deadlock.

Built-in thread tests

When you booted OS/161 in ASST0, you may have seen the options to run the thread tests. The thread test

code uses the semaphore synchronization primitive. You should trace the execution of one of these thread tests in GDB to see how the scheduler acts, how threads are created, and what exactly happens in a context switch. You should be able to step through a call to mi_switch() and see exactly where the current thread changes.

Thread test 1 (" tt1 " at the prompt or tt1 on the kernel command line) prints the numbers 0 through 7 each time each thread loops. Thread test 2 (" tt2 ") prints only when each thread starts and exits. The latter is intended to show that the scheduler doesn't cause starvation--the threads should all start together, spin for a while, and then end together.

Debugging concurrent programs

`thread_yield()` is automatically called for you at intervals that vary randomly. While this randomness is fairly close to reality, it complicates the process of debugging your concurrent programs.

The random number generator used to vary the time between these `thread_yield()` calls uses the same seed as the random device in System/161. This means that you can reproduce a specific execution sequence by using a fixed seed for the random number generator. You can pass an explicit seed into random device by editing the "random" line in your sys161.conf file. For example, to set the seed to 1, you would edit the line to look like:

```
28 random seed=1
```

We recommend that while you are writing and debugging your solutions you pick a seed and use it consistently. Once you are confident that your threads do what they are supposed to do, set the random device to auto-seed. This should allow you to test your solutions under varying conditions and may expose scenarios that you had not anticipated. To reproduce your test cases, you additionally need to run your tests via command line args to sys161 as described above.

Reading Exercises

You must go through all the questions -- given below. Please discuss possible answer with your group member. This time you do not need to submit the answers. This will help you to understand codes required for your assignment1.

Code reading

To implement synchronization primitives, you will have to understand the operation of the threading system in OS/161. It may also help you to look at the provided implementation of semaphores. When you are writing solution code for the synchronization problems it will help if you also understand exactly what the OS/161 scheduler does when it dispatches among threads.

Thread Questions

1. What happens to a thread when it exits (i.e., calls thread_exit())? What about when it sleeps?
2. What function(s) handle(s) a context switch?
3. How many thread states are there? What are they?
4. What does it mean to turn interrupts off? How is this accomplished? Why is it important to turn off interrupts in the thread subsystem code?
5. What happens when a thread wakes up another thread? How does a sleeping thread get to run again?

Scheduler Questions

6. What function is responsible for choosing the next thread to run?
7. How does that function pick the next thread?
8. What role does the hardware timer play in scheduling? What hardware independent function is called on a timer interrupt?

Synchronisation Questions

9. Describe how thread_sleep() and thread_wakeup() are used to implement semaphores. What is the purpose of the argument passed to thread_sleep() ?
10. Why does the lock API in OS/161 provide lock_do_i_hold() , but not lock_get_holder() ?

Synchronisation Problems

The following problems are designed to familiarize you with some of the problems that arise in concurrent programming and help you learn to identify and solve them.

Identify Deadlocks

11. Here are code samples for two threads that use binary semaphores. Give a sequence of execution and context switches in which these two threads can deadlock.
12. Propose a change to one or both of them that makes deadlock impossible. What general principle do the original threads violate that causes them to deadlock?

```
semaphore *mutex, *data;

void me() {
        P(mutex);
        /* do something */

        P(data);
        /* do something else */

        V(mutex);

        /* clean up */
        V(data);
}

void you() {
        P(data)
        P(mutex);

        /* do something */

        V(data);
        V(mutex);
}
```

More Deadlock Identification

13. Here are two more threads. Can they deadlock? If so, give a concurrent execution in which they do and propose a change to one or both that makes them deadlock free.

```
lock *file1, *file2, *mutex;

void laurel() {
        lock_acquire(mutex);
```

```
        /* do something */

        lock_acquire(file1);
        /* write to file 1 */

        lock_acquire(file2);
        /* write to file 2 */

        lock_release(file1);
        lock_release(mutex);

        /* do something */

        lock_acquire(file1);

        /* read from file 1 */
        /* write to file 2 */

        lock_release(file2);
        lock_release(file1);
}

void hardy() {
        /* do stuff */

        lock_acquire(file1);
        /* read from file 1 */

        lock_acquire(file2);
        /* write to file 2 */

        lock_release(file1);
        lock_release(file2);

        lock_acquire(mutex);
        /* do something */
        lock_acquire(file1);
        /* write to file 1 */
        lock_release(file1);
        lock_release(mutex);
}
```

Synchronised Queues

14. The thread subsystem in OS/161 uses a queue structure to manage some of its state. This queue structure is not synchronised. Why not? Under what circumstances should you use a synchronised queue structure?

Describe (and give pseudocode for) a synchronised queue data structure based on the queue structure included in the OS/161 codebase. You may use semaphores, locks, and condition variables as you see fit. You must describe (a proof is not necessary) why your algorithm will not deadlock.

Make sure you clearly state your assumptions about the constraints on access to such a structure and how you ensure that these constraints are respected.

Coding Assignment

We know: you've been itching to get to the coding. Well, you've finally arrived!

This is the assessable component of this assignment. It is worth 25% (design 5 and coding 20) marks of the 100% available for the lab. mark component of the course.

Solving Synchronisation Problems

The following problems will give you the opportunity to write one fairly straightforward concurrent programs and get a more detailed understanding of how to use concurrency mechanisms to solve problems.

We have provided you with basic driver code that starts a predefined number of threads that execute a predefined activity (in the form of calling functions that you must implement). You are responsible for implementing the functions called.

Remember to specify a seed to use in the random number generator by editing your sys161.conf file, and run your tests using sys161 command line args. It is much easier to debug initial problems when the sequence of execution and context switches is reproducible.

When you configure your kernel for ASST1, the driver code and extra menu options for executing your solutions are automatically compiled in.

Solving Synchronisation Problems

The following problems will give you the opportunity to write two fairly straightforward concurrent programs and get a more detailed understanding of how to use concurrency mechanisms to solve problems.

We have provided you with basic driver code that starts a predefined number of threads that execute a predefined activity (in the form of calling functions that you must implement). You are responsible for implementing the functions called.

Remember to specify a seed to use in the random number generator by editing your sys161.conf file, and run your tests using sys161 command line args. It is much easier to debug initial problems when the sequence of execution and context switches is reproducible.

When you configure your kernel for ASST1, the driver code and extra menu options for executing your solutions are automatically compiled in.

Concurrent Mathematics Problem

For the first problem, we ask you to solve a very simple mutual exclusion problem. The code in `kern/asst1/math.c` counts from 0 to 10000 by starting several threads that increment a common counter.

You will notice that as supplied, the code operates incorrectly and produces results like `345 + 1 = 352` .

Once the count of 10000 is reached, each thread signals the main thread that it is finished and then exits. Once all `adder()` threads exit, the main (`math()`) thread cleans up and exits.

Your Job

Your job is to modify `math.c` by placing synchronisation primitives appropriately such that incrementing the counter works correctly. The statistics printed should also be consistent with the overall count.

Note that the number of increments each thread performs is dependent on scheduling and hence will vary. However, the total should equal the final count.

To test your solution, use the "1a" menu choice. Sample output from a correct solution in included below.

```
% sys161 kernel "1a;q"
```

```
sys161: System/161 release 2.0.8, compiled Aug 26 2018 15:26:35

OS/161 base system version 2.0.3
Copyright (c) 2000, 2001-2005, 2008-2011, 2013, 2014
   President and Fellows of Harvard College.  All rights reserved.

Put-your-group-name-here's system version 0 (ASST1 #1)

352k physical memory available
Device probe...
lamebus0 (system main bus)
emu0 at lamebus0
ltrace0 at lamebus0
ltimer0 at lamebus0
beep0 at ltimer0
rtclock0 at ltimer0
lrandom0 at lamebus0
random0 at lrandom0
lhd0 at lamebus0
lhd1 at lamebus0
lser0 at lamebus0
con0 at lser0

cpu0: MIPS/161 (System/161 2.x) features 0x0
OS/161 kernel [? for menu]: 1a
Starting 10 adder threads
In thread 0, 6033 + 1 == 10001?
Adder threads performed 10001 adds
Adder 0 performed 10001 increments.
Adder 1 performed 0 increments.
Adder 2 performed 0 increments.
Adder 3 performed 0 increments.
Adder 4 performed 0 increments.
Adder 5 performed 0 increments.
Adder 6 performed 0 increments.
Adder 7 performed 0 increments.
Adder 8 performed 0 increments.
Adder 9 performed 0 increments.
The adders performed 10001 increments overall
Operation took 0.883992080 seconds
OS/161 kernel [? for menu]:q
Shutting down.
The system is halted.
```

Paint Shop Synchronisation Problem

You're in the middle of renovating your apartment and need to purchase some paint. You arrive at your local paint shop to find the shop in complete chaos. Customers are receiving empty cans of paint or paint with weird colors, the staff are fighting over the various tints used to color the paint, orders are getting lost, cans of paint mixed up, some customers are waiting forever for their can of paint, while others seems to get all the service.

Being an operating system expert, you quickly realize that the shop's problems are related to concurrency issues between the customers and shop staff. You volunteer your services to provide a solution to the shop's problems, reduce the chaos, and restore order to the store.

The Basic Paint Shop

To provide a solution, you must come to terms with the basic elements of the paint shop that you have to work with. The shop consists of a set of tints (such as RED, GREEN, BLUE) used to color paint. They are mixed with paint by shop staff in various ways according to customer requests. Customers bring their own

empty paint can, record the tints they require on the can itself, and give the can to shop staff to mix the paint. The basic elements are defined in `kern/asst1/paintshop_driver.h`. The actions of customers and shop staff are defined in `kern/asst1/paintshop_driver.c`. See the file for detailed comments.

- **Customers** arrive with their paint can, write their requested colour on the can in terms of tints, submit their can as an order to the paint shop staff and wait.

  Eventually their can returns with the requested contents (exactly as requested), they paint until the can is empty, take a short break, and do it all again until they have emptied the desired number of cans, then they go home.
- **Shop Staff** are only slightly more complicated than the customers. They take_orders, and if valid, they fill them and serve them. When all the customers have left, the staff go home.

  An invalid order signals that the staff member should go home.

The function `runpaintshop()` is called via the menu in OS/161 (item `1b`). `runpaintshop()` does the following:

- It initialises all the tint containers to have served zero doses.
- It calls `paintshop_open`, a routine you will provide to set up the shop.
- It then creates some threads to run as shop staff, and some more threads to run as customers. Note these threads obviously run concurrently.
- The driver thread then waits on a semaphore for all the staff and customers to finish, after which we print out the tint statistics for the day.
- Finally, it calls `paintshop_close`, a procedure you provide to clean up when the shop has closed.

The function `mix()` takes a can and associated tint request and "mixes" the tints into the can such that the content is exactly as requested. The tints are represented by numbers, each number corresponds to the tint container number (and colour). The meaning of the tint numbers are defined in `paintshop_driver.h`.

You can assume that all the tint containers in the shop are infinite in size and hence will never be empty.

Have a quick look through both `paintshop_driver.c` and `paintshop_driver.h` to reinforce your understanding of what is going on (well, at least what is expected to go on).

Your Job

Your job is to write the functions outlined in `paintshop.c` (and potentially modify `paintshop.h` that perform most of the work. Each function is described in `paintshop.c`.

Generally, your solution must result in the following when `runpaintshop()` is called during testing.

- The shop being prepared for opening.
- All customers having their orders served with the correctly tinted paint in a can. "Correct" means that each corresponding entry in the `contents` array contains what was originally requested in the `requested_colours` array.
- The shop staff all going home after all the customers are finished.
- The shop being suitably cleaned up afterwards (allocated memory or locks, semaphores, etc being freed).
- Statistics kept on tint usage are consistent with the orders made.

You can modify `paintshop_driver.c` and `paintshop_driver.h` to test different scenarios (e.g

vary the number and colour of paint cans ordered), but **your solution must also work with an unmodified version of the `paintshop_driver.c` file.**

You will have to modify `paintshop.c` to implement your solution. However, your modifications have the constraint that they must still work with an original `paintshop_driver.c`.

For testing, we will replace `paintshop_driver.c` and `.h` with logically equivalent versions that may vary the numbers of participants, and the colours requested. We may also vary the timing of various functions. A correct solution will work for all variations we test. Sample output from a correct solution is included below.

```
% sys161 kernel "1b;q"
sys161: System/161 release 2.0.8, compiled Jul  9 2018 11:30:58

OS/161 base system version 2.0.3
Copyright (c) 2000, 2001-2005, 2008-2011, 2013, 2014
   President and Fellows of Harvard College.  All rights reserved

Put-your-group-name-here's system version 0 (ASST1 #3)

Cpu is MIPS r2000/r3000
848k physical memory available
Device probe...
lamebus0 (system main bus)
emu0 at lamebus0
ltrace0 at lamebus0
ltimer0 at lamebus0
hardclock on ltimer0 (10000 hz)
beep0 at ltimer0
rtclock0 at ltimer0
lrandom0 at lamebus0
random0 at lrandom0
lser0 at lamebus0
con0 at lser0
pseudorand0 (virtual)

OS/161 kernel: 1b
S 0 going home after mixing 34 orders
S 1 going home after mixing 33 orders
S 2 going home after mixing 33 orders
Tint 1 used for 0 doses
Tint 2 used for 0 doses
Tint 3 used for 0 doses
Tint 4 used for 0 doses
Tint 5 used for 0 doses
Tint 6 used for 0 doses
Tint 7 used for 0 doses
Tint 8 used for 100 doses
Tint 9 used for 0 doses
Tint 10 used for 0 doses
The paint shop is closed, bye!!!
Operation took 0.529411720 seconds
OS/161 kernel: q
Shutting down.
The system is halted.
```

Before Coding!!!!

You should have a very good idea of what your attempting to do before you start. Concurrency problems are very difficult to debug, so it's in your best interest that you convince yourself you have a correct solution before you start.

The following questions may help you develop your solution.

- What are the shared resources (e.g. tint containers)?
- Who shares what resources?
- Who produces what and who consumes what (e.g. customers produce orders consumed by staff)?
- What states can the various resources be in?
- What do you need to keep a count of (e.g. number of customers in the shop)?
- How does your solution prevent deadlock or starvation?

Try to frame the problem in terms of resources requiring concurrency control, and producer-consumer problems. A diagram may help you to understand the problem.

Evaluating your solutions

Your solutions will be judged in terms of its correctness, conciseness, clarity, and performance.

Performance will be judged in at least the following areas.

- Do all the staff members participate?
- Can staff mix in parallel if they do not require the same tints?
- Do you define critical sections larger than needed?

Documenting your solutions

This is a compulsory component of this assignment. You must write a small design document in latex (.tex) identifying the basic issues in both of the concurrency problems in this assignment, and then describe your solution to the problems you have identified. For example, detail which data structures are shared, and what code forms a critical section. The document must be plain ASCII source script of latex. We expect such a document to roughly 200 - 1000 words, i.e. clear and to the point.

The document will be used to guide our markers in their evaluation of your solution to the assignment. In the case of a poor results in the functional testing combined with a poor design document, we will base our assessment on these components alone. If you can't describe your own solution clearly, you can't expect us to reverse engineer the code to a poor and complex solution to the assignment.

Create your design document as a latex source script `design.tex`. After creating design.tex, will produce design.dvi when you compile with latex.

```
% latex design.tex
%
```

Generating Your Assignment Submission

As with assignment 0, you again will be submitting a set of files (including a design document) which you have changed to the original os161 asst1/src tree.

Submitting Your Assignment

You must submit the following files.

1. design.tex
2. paintshop.c
3. painshop.h

4. math.c
5. math.h

Submission asst1. You're now done. Submit before the due date and time.

Note: If for some reason you need to change and re-submit your assignment. The last submission will be accepted only for mark.

Plagiarism

We take cheating seriously!!!
Penalties include
-- Copying of code: 0
-- Help with coding: negative half the assignment's max marks
-- Originator of a plagiarised solution: 0 for the particular assignment
-- Team work outside group: 0 for the particular assignments
