.

COMP3231/9201/3891/9283 Operating Systems 2021/T1

UNSW

ASST1: Synchronisation

Table of Contents

Due Dates and Mark Distribution

Due Date & Time: 4pm (16:00), March 12 (Week 4)

Marks: Worth 30 marks (of the class mark component of the course)

The 2% per day bonus for each day early applies, capped at 10%, as per course outline.

Introduction

In this assignment you will solve a number of synchronisation problems within the software environment of the OS/161 kernel. By the end of this assignment you will gain the skills required to write concurrent code within the OS/161 kernel. While the synchronisation problems themselves are only indirectly related to the services that OS/161 provides, they solve similar concurrency problems that you would encounter when writing OS code.

The Week 3 tutorial contains various synchronisation familiarisation exercises. Please prepare for it. Additionally, feel free to ask any assignment related questions in the tutorial.

Setting Up Your Assignment

We assume after ASST0 that you now have some familiarity with setting up for OS/161 development. The following is a brief setup guide. If you need more detail, refer back to ASST0.

Obtain the ASST1 distribution with git

Clone the ASST1 source repository from `gitlab.cse.unsw.edu.au`.

```
% cd ~/cs3231
% git clone https://zXXXXXXX@gitlab.cse.unsw.edu.au/COMP3231/21T1/zXXXXXXX-asst1.git asst1-src
```

Configure OS/161 for Assignment 1

Configure your new sources as follows.

```
% cd ~/cs3231/asst1-src
% ./configure && bmake && bmake install
```

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

You have to configure your kernel itself 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 ~/cs3231/asst1-src/kern/conf
% ./config ASST1
```

You should now see an ASST1 directory in the `kern/compile` directory.

Building ASST1

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

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

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

Tip: Once you start modifying the OS/161 kernel, you can quickly rebuild and re-install with the following command sequence. It will install the kernel if the build succeeds.

```
% bmake && bmake install
```

Check sys161.conf

The `sys161.conf` should be already be installed in the `~/cs3231/root` directory from assignment 0. If not, follow the instructions below to obtain another copy. A pre-configured sys161 configuration is available here: sys161.conf.

```
% cd ~/cs3231/root
% wget http://cgi.cse.unsw.edu.au/~cs3231/21T1/assignments/asst1/sys161.conf
```

Run the kernel

Run the previously built kernel:

```
% cd ~/cs3231/root
% sys161 kernel
sys161: System/161 release 2.0.8, compiled Feb 25 2019 09:34:40

OS/161 base system version 2.0.3
(with locks&CVs solution)
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)

16220k 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
lser0 at lamebus0
con0 at lser0

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

Kernel menu commands and arguments to OS/161

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

Caution!	**Do not** change these menu option strings!

Here are some examples of using command line arguments 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 (";").

Concurrent Programming with OS/161

If your code is properly synchronised, the timing of context switches, the location of `kprintf()` calls, and the order in which threads run should not influence the correctness of your solution. Of course, your threads may print messages in different orders, but you should be able to verify that they implement the functionality required and that they do not deadlock.

Debugging concurrent programs

`thread_yield()` is automatically called for you at intervals that vary randomly. `thread_yield()` context switches between threads via the scheduler to provide multi-threading in the OS/161 kernel. While the 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 the 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 start the kernel via command line arguments and 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 autoseed. This should allow you to test your solutions under varying timing that may expose scenarios that you had not anticipated.

To reproduce your test cases, you need to run your tests via the command line arguments to sys161 as described above, otherwise system behaviour will depend on your precise typing speed (and not be reproducible for debugging).

Tutorial Exercises

The aim of the week 3 tutorial is to have you implement synchronised data structures using the supplied OS synchronisation primitives. See the [Week 03 Tutorial] for details.

It is useful to be prepared to discuss both the questions and the following assignment in your tutorial.

Code reading

The following questions aim to guide you through OS/161's implementation of threads and synchronisation primitives in the kernel itself for those interested in a deeper understanding of OS/161. A deeper understanding can be useful when debugging, but is not strictly required, though recommended especially for Extended OS students.

For those interested in gaining a deeper understanding of how synchronisation primitives are implemented, it is helpful to understand the operation of the threading system in OS/161. After which, walking through the implementation of the synchronisation primitives themselves should be relatively straightforward.

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. What is a wait channel? Describe how `wchan_sleep()` and `wchan_wakeone()` are used to implement semaphores.
10. Why does the lock API in OS/161 provide `lock_do_i_hold()`, but not `lock_get_holder()`?

Coding the 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.

The following problems will give you the opportunity to write some fairly straightforward concurrent systems and get a practical understanding of how to use concurrency mechanisms to solve problems. We have provided you with basic tester code that starts a predefined number of threads that execute a predefined activity (in the form of calling functions that you must implement or modify).

Note: In this assignment, you are restricted to the *lock*, *semaphore*, and *condition variable* primitives provided in OS/161. The use of other primitives such as `thread_yield()`, *spinlocks*, interrupt disabling (*spl*), atomic instructions, and the like are **prohibited**. Moreover, they usually result in a poor solution involving busy waiting.

Note: In some instances, the comments within the code also form part of the specification and give guidance as to what is required. Make sure you read the provided code carefully.

Check that you have specified a seed to use in the random number generator by examining your `sys161.conf` file, and run your tests using System/161 command line arguments. It is much easier to debug initial problems when the sequence of execution and context switches are reproducible.

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

Part 1: The Concurrent Counter Problem

Marks: 5

For the first problem, we ask you to solve a mutual exclusion problem. The code in `kern/asst1/counter.c` provides an interface to initialise (`counter_initialise()`), increment (`counter_increment()`), decrement (`counter_decrement()`), and read and cleanup a counter when finished (`counter_read_and_destroy()`). The increment and decrement code can be called concurrently by multiple threads and is unsynchronised.

The testing code provided in `kern/asst1/counter_tester.c` exercises a subset of the counter code and produces an incorrect result similar to the following. Note that the final count *is* dependent on scheduling and hence will vary.

```
OS/161 kernel [? for menu]: 1a
Starting 10 incrementer threads
The final count value was 5083 (expected 10000)
```

Your Task

Your task is to modify `kern/asst1/counter.c` by synchronising the code appropriately such that incrementing and decrementing the counter works correctly.

You can assume that `counter_initialise()` and `counter_read_and_destroy()` are **not** called concurrently, and `counter_read_and_destroy()` is always called sometime after the a call to `counter_initialise()`, before any later call to `counter_initialise()`. `counter_increment()` and `counter_decrement()` are only ever called (multiple time) after a call to `counter_initialise()` and before the final call to `counter_read_and_destroy()`.

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

```
OS/161 kernel [? for menu]: 1a
Starting 10 incrementer threads
The final count value was 10000 (expected 10000)
```

Part 2: Simple Deadlock

Marks: 5

This task involves modifying an example such that the example no longer deadlocks and is able to finish. The example is in `twolocks.c`.

In the example, `bill()`, `bruce()`, `bob()` and `ben()` are threads that need to hold one or two locks at various times to make progress: `lock_a` and `lock_b`. While holding one or two locks, the threads call *holds_lockX* that just consumes some CPU. The way the current code is written, the code deadlocks and triggers OS/161's deadlock detection code, as shown below.

```
OS/161 kernel: 1b
Locking frenzy starting up
Hi, I'm Bill
Hi, I'm Ben
Hi, I'm Bruce
Hi, I'm Bob
hangman: Detected lock cycle!
hangman: in ben thread (0x80031ed8);
hangman: waiting for lock_a (0x80032d04), but:
lockable lock_a (0x80032d04)
    held by actor bill thread (0x80031f58)
    waiting for lockable lock_b (0x80032cc4)
    held by actor ben thread (0x80031ed8)
panic: Deadlock.
sys161: trace: software-requested debugger stop
sys161: Waiting for debugger connection...
```

You task is to modify the existing code such that:

- you apply resource-ordering deadlock prevention such that the code no longer deadlocks, and runs to completion as shown below (the ordering may vary);
- the modified solution still calls the *holds_lockX* functions in the same places, and only the locks indicated are held by the thread at that point in the code;
- your deadlock free solution only uses the existing locks and calls them the same number of times; and
- you document the overall resource order chosen in the comment indicated in the code.

```
OS/161 kernel: 1b
Locking frenzy starting up
Hi, I'm Bill
Hi, I'm Bruce
Hi, I'm Ben
Hi, I'm Bob
Bruce says 'bye'
Bob says 'bye'
Ben says 'bye'
Bill says 'bye'
Locking frenzy finished
```

Part 3: Bounded-buffer producer/consumer problem

Marks: 10

Your next task in this part is to synchronise a solution to a producer/consumer problem. In this producer/consumer problem, one or more *producer* threads allocate data structures, and call `producer_send()`, which copies pointers to the data structures into a fixed-sized buffer, while one or more *consumer* threads retrieve those pointers using `consumer_receive()`, and inspect and de-allocate the data structures.

The code in `kern/asst1/producerconsumer_tester.c` starts up a number of producer and consumer threads. The producer threads attempt to send pointers to the consumer threads by calling the `producer_send()` function with a pointer to the data structure as an argument. In turn, the consumer threads attempt to receive pointers to the data structure from the producer threads by calling `consumer_receive()`. **These functions are currently partially implemented. Your job is to modify and synchronise them**.

Here's what you might see before you have implemented any code:

```
OS/161 kernel [? for menu]: 1c
run_producerconsumer: starting up
Waiting for producer threads to exit...
Consumer started
Producer started
Consumer started
Producer finished
Consumer started
Producer started
*** Error! Unexpected data -2147287680 and -2147287680
Consumer started
*** Error! Unexpected data -2147287712 and -2147287712
Consumer started
*** Error! Unexpected data -2147287648 and -2147287648
*** Error! Unexpected data -2147287712 and -2147287712
*** Error! Unexpected data -2147287648 and -2147287648
*** Error! Unexpected data -2147287648 and -2147287648
*** Error! Unexpected data -2147287648 and -2147287648
*** Error! Unexpected data -2147287712 and -2147287712
*** Error! Unexpected data -2147287664 and -2147287664
*** Error! Unexpected data -2147287664 and -2147287664
*** Error! Unexpected data -2147287600 and -2147287600
*** Error! Unexpected data -2147287600 and -2147287600
*** Error! Unexpected data -2147287664 and -2147287664
*** Error! Unexpected data -2147287600 and -2147287600
panic: Assertion failed: fl != fl->next, at ../../vm/kmalloc.c:1134 (subpage_kfree)
```

Note that code will panic (crash) in different ways depending on the timing.

And here's what you will see with a (possibly) correct solution:

```
OS/161 kernel: 1c
run_producerconsumer: starting up
Consumer started
Waiting for producer threads to exit...
Producer started
Consumer started
Consumer started
Producer started
Consumer started
Consumer started
Producer finished
Producer finished
All producer threads have exited.
Consumer finished normally
Consumer finished normally
Consumer finished normally
```

```
Consumer finished normally
Consumer finished normally
```

The files:

- `producerconsumer_tester.c`: Starts the producer/consumer simulation by creating producer and consumer threads that will call `producer_send()` and `consumer_receive()`. You are welcome to modify this simulation when testing your implementation — in fact, you are encouraged to — but remember that it will be overwritten when your solution is tested.
- `producerconsumer.h`: Contains prototypes for the functions in `producerconsumer.c`, as well as the description of the data structure that is passed from producer to consumer (the uninterestingly-named `data_item_t`). This file will also be overwritten when your solution is tested (or automarked).
- `producerconsumer.c`: Contains your implementation of `producer_send()` and `consumer_receive()`. It also contains the functions `producerconsumer_startup()` and `producerconsumer_shutdown()`, which you can implement to initialise any variables and any synchronisation primitives you may need.

Suggestions on how to implement your solution

You must implement a data structure representing a buffer capable of holding `BUFFER_SIZE` `data_item_t` pointers. This means that calling `producer_send()` `BUFFER_SIZE` times should not block (or overwrite existing items, of course), but calling `producer_send()` one more time **should** block, until an item has been removed from the buffer using `consumer_receive()`. We have provided an unsynchronised skeleton of circular buffer code, though you will of course have to use appropriate synchronisation primitives to ensure that concurrent access is handled safely.

The data structure should function as a circular buffer with first-in, first-out semantics.

Part 4: The Client/Server System

Marks: 10

This part simulates a client/server system where multiple client threads put requests into a single queue where server threads can dequeue the requests, process them, and mark the requests as done so the client can continue.

Clients are in a loop that issue a number of requests one after another. The code that drives the system is in `client_server_tester.c`. You should review the code to develop an understanding of the system. You'll see it starts a number of client and server threads and then waits for the clients to issue all their requests. It then signals the servers to exit, waits, and completes.

The client and server threads interact via the functions implemented in `client_server.c`. At a high level, these functions manage a work queue of requests for the servers. A more detailed specification of the each function is provided in the comments in the code itself.

Your task is to implement the functions such that the client/server system will execute correctly with all the requests processed.

- Your solution should not busy-wait when a thread can't make progress.
- You should not rely on any changes to code in the `client_server_tester.c` or `client_server.h` files. It will be changed for testing. You can vary the code for your own testing purposes, but we'll replace them for our own testing.

The code as supplied fails as follows.

```
OS/161 kernel [? for menu]: 1d
run_client_server_system: starting up
panic: work queue setup returned an error
```

A potentially correct solution generates output similar to that below. Note: The order of starting and finishing, and the number of tickets validated by each validator will vary on each execution run. The total number of tickets validated will be 300 in the supplied code.

```
OS/161 kernel [? for menu]: 1d
run_client_server_system: starting up
Waiting for client threads to exit...
Client 3 started
Server 0 started
Client 6 started
Client 4 started
Client 8 started
Client 5 started
Client 0 started
Server 2 started
Client 9 started
Client 7 started
Client 2 started
Server 1 started
Client 1 started
Client 3 finished
Client 6 finished
Client 4 finished
Client 0 finished
Client 5 finished
Client 8 finished
Client 7 finished
Client 9 finished
Client 2 finished
Client 1 finished
All 10 client threads have exited.
All 3 server threads have exited.
Server 0 processed 197 requests
Server 1 processed 142 requests
Server 2 processed 161 requests
Giving a total of 500 (expected 500)
```

Submitting

The submission instructions are available on the Wiki. Like ASST0, you will be submitting the git repository bundle via CSE's `give` system. For ASST1, the submission system will do a test build and run a simple test to confirm your bundle at least compiles. It does not exhaustively test you submission

> **Warning** **Don't ignore the submission system! If your submission fails the simple tests in the submission process, you may not receive any marks.**

To submit your bundle:

```
% cd ~
% give cs3231 asst1 asst1.bundle
```

You're now done.

Even though the generated bundle should represent all the changes you have made to the supplied code, occasionally students do something "ingenious". So always keep your git repository so that you may recover your assignment should something go wrong. We recommend to `git push` it back to `gitlab.cse.unsw.edu.au` for safe keeping.

Page last modified: 10:19pm on Thursday, 25th of February, 2021

Screen Version