COMP3231/9201/3891/9283 Operating Systems 2020/T2

UNSW

Administration
- Notices
- Course Outline
- UNSW Timetable
- Consultations
- Group Nomination
- Survey Results!!

Work
- Lectures
- Tutorials

Support
- Piazza Forums
- Wiki

Assignments
- Submission Guide
- Assignment 0 Warm-up
- Assignment 1
- Assignment 2
- Assignment 3

Exam
- Sample Questions and Answers

Resources
OS/161
- General
- Man Pages
- Sys161 Pages

C coding
- Info Sheet

Debugging
- Learn Debugging

General
- "Hardware" Guide
- R3000 Reference Manual
- Intro. to Prog. Threads

Previous years
- 2020 T1
- 2019 T1
- 2018 S1
- 2017 S1
- 2016 S1
- 2015 S1
- 2014 S1
- 2013 S1
- 2012 S1
- 2011 S1
- 2010 S1
- 2009 S1
- 2008 S1

ASST0: Getting warmed up

Table of Contents

Due Time & Date and Mark Distribution

Due Time & Date: 23:59:59, Sun June 14th (Week 2)

Marks: Worth 10 marks (contributing to the 100 potential class marks available in the course)

This warmup exercise is done as an individual.

None of the early or advanced bonuses apply for this warmup exercise.

Introduction

The aim of the warmup exercise is to have you familiarise yourself with the environment that you will be using for the more substantial assignments. The exercise consists of two parts: an assessed exercise for marks, and a non-assessable component. The non-assessable component consists of a set of directed questions to give you practice navigating and reading the code base. The answers to this code reading Q&A component of the exercise are available on the wiki (improvable by you the students). The assessable exercise consists of you: learning how to build and run OS/161, making a very minor change to the existing OS to fix a bug, and learning the submission process. The change is conceptually trivial, so you can view this exercise as us giving away marks as an incentive for you to get the assignment environment up and running early in the trimester in preparation for the assignments.

Note that this exercise is not indicative of the level of difficulty of the actual assignments. *The assignments are much more challenging*.

The Assignment Environment

The assignment environment consists of broadly two parts. The OS/161 OS teaching environment, and some relatively standard software development tools. This section provides a brief overview of both.

OS/161

The major OS/161 components you will use during the semester are as follows:

- **OS/161**, the educational operating system whose source code you will modify to implement the assignments.
- **System/161** (`sys161`), a component that simulates a MIPS-based computer that OS/161 runs on.

The introduction to the OS/161 system provides more details on how the components of the OS/161 system fit together. You should take the time to read it now.

The Development Tools

The OS/161 development environment uses various standard and modified software development tools.

- **bmake**, a program that takes a specification of what needs to be compiled to build a piece of software (i.e. OS/161) and runs the compilation tools. It determines what tools need to be run in order to finish building software and uses file timestamps to avoid doing work already completed.

- **OS/161 cross compilers**, specifically `os161-gcc`, `os161-ld`, and other tools with the `os161-` prefix. These are specific versions of the GCC C compiler tools that generate MIPS binaries to run on `sys161`, instead of normal Intel x86 binaries that run on a Linux or Windows PC. The OS/161 build system uses `bmake` to hide most of the complexity of using these tools directly.

- **GDB** (`os161-gdb`), a version of the GDB debugger that will make your life much easier.

  You should already be familiar the GDB, the GNU debugger. GDB allows you to set breakpoints to stop your program under certain conditions, inspect the state of your program when it stops, modify its state, and continue where it left off. It is a powerful aid to the debugging process that is worth investing the time needed to learn it. GDB allows you to quickly find bugs that are very difficult to find with the typical `printf` style debugging.

- **git** is a source code management tool used to track changes to a piece of software. We make use of git in this course to allow you to manage the large OS/161 code base. `git` keeps a copy of each *committed* change to your OS/161 files. It allows you to track the changes you have made, and more importantly, rollback to a known state if things get out of hand.

  `git` also enables you and your partner to work on the same code repository in a coordinated way using `gitlab.cse.unsw.edu.au` to share a `git` repository between you. You can even work on versions of your own code at home and at CSE, with `git` keeping them consistent with each other.

`git` is a large system of which you need to know only a small subset. We will give you directions for the parts you need to know, but feel free to expand your knowledge as git is a common tool worth knowing, and is used widely in industry.

Getting Started

In this section, we will walk through obtaining the OS/161 source code using git, looking around the source tree, and finally, building and running your copy of OS/161.

Setting up your development environment

These instructions assume you are working at CSE in some way (e.g. remotely via VLAB). Using CSE machines requires the least effort to set up and is supported, but is also less efficient than working on your own machine.

See the Wiki for details of how to set up your own development environment. Working on your own equipment is the most efficient, but requires a little time to set up – time that you have at the start of the trimester.

At CSE, run the `3231` command in each new terminal window to make the tools visible on the commandline. See the transcript below for an example of how running `3231` makes os161-gcc visible in on your commandline.

```
% which os161-gcc
os161-gcc not found
% 3231
newclass starting new subshell for class COMP3231...
% which os161-gcc
/home/cs3231/bin/os161-gcc
%
```

If you are here, we assume you have the cross compilations tools accessible on the commandline.

Obtaining the OS/161 source code with git

The following steps obtains a copy of the assignment sources

- Create a sub-directory in your home directory to contain your cs3231 work. **Note: The source code expects the name "cs3231"**

```
% cd ~
% mkdir cs3231
```

- Clone the source repository from `gitlab.cse.unsw.edu.au`. Note: You should change the student number to match your own, and use the COMP3231 course code even if you are enrolled in another course code.

```
% cd ~/cs3231
% git clone https://z8888888@gitlab.cse.unsw.EDU.AU/COMP3231/20T
2/z8888888-asst0.git asst0-src
```

You should now have an `asst0-src` directory to work on.

Looking around the source code

Now that you have an OS/161 source tree, you can go through the walkthrough exercise.

This optional but strongly encouraged component of this warm-up that aims to guide you through the code base to help you develop an overview of its contents (its too big to get a detailed understanding quickly).

Note that while the code reading component is not assessable, we view it as the ideal opportunity to get to know your way around the code, including practising with whatever code browsing tool or editor you plan to use (e.g. Visual Studio Code (`code` at CSE)). You will benefit in the assignments by practising navigating around and developing a rudimentary understanding of the code base. The code reading component is here to give you some of that practice.

Building the Application Executables

Just like Linux (or some other OS), OS/161 has executable applications installed in its file system. This step builds the executables and installs them in `~/cs3231/root`, which forms the root of the file system that OS/161 sees. This only needs to be done once for the assignment.

- You first have to configure your source tree. This step sets up some information for later compilation, such as the location of various directories, the compiler to use, and compilation flags to supply.

```
% cd ~/cs3231/asst0-src
% ./configure
```

- Now we build the applications using `bmake`.

```
% bmake
% bmake install
```

Hopefully you've successfully installed the OS/161 applications. You can list the root of the file system as shown below, and you'll see a UNIX–like set of directories containing binaries, libraries, etc.

```
% ls ~/cs3231/root
bin  hostbin  hostinclude  include  lib  man  sbin  testbin  tes
tscripts
%
```

Building and Rebuilding the OS/161 OS Kernel

Now lets build the OS/161 kernel for the first time.

- First, configure the kernel. This step prepares for a compilation of the specific set of files that make up the warmup kernel.

```
% cd ~/cs3231/asst0-src/kern/conf
% ./config ASST0
```

- The next task is to build the kernel.

```
% cd ../compile/ASST0
% bmake depend
% bmake
```

- Now install the kernel in `~/cs3231/root`

```
% bmake install
```

If you list the root file system, you'll now see the `kernel` has been installed.

```
% ls ~/cs3231/root
bin  hostbin  hostinclude  include  kernel  kernel-ASST0  lib  m
an  sbin  testbin  testscripts
%
```

Rebuilding after Code Changes

Building an updated version of the OS/161 kernel after making code changes is simpler.

- Ensure you are in the `kern/compile/ASST0` directory.

```
% cd ~/cs3231/asst0-src/kern/compile/ASST0
```

- Use `bmake` to re-compile only the files that have changed, and then install the new kernel.

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

Note the `&&` will only run the install if the first `bmake` succeeds.

Running your OS/161 Kernel

If you have made it this far, you have built and installed the entire operating system and applications from source code. Now it is time to run the OS on the MIPS system simulator `sys161`.

- First, we need a configuration file for `sys161`. Download the sample sys161.conf and copy it to `~/cs3231/root/sys161.conf`, or use the command below. This only needs to be set up once.

```
% cd ~/cs3231/root
% wget http://cgi.cse.unsw.edu.au/~cs3231/20T2/assignments/asst
0/sys161.conf
```

- Ensure you are in the root directory of your OS.

```
% cd ~/cs3231/root
```

- Now run system/161 (`sys161`) on your kernel.

```
% sys161 kernel
```

- You will observe OS/161 begin its boot sequence and then encounter an error, panic, and wait for a debugger. It does not get the point where you could start an application.

```
sys161: System/161 release 2.0.8, compiled Feb 19 2017 14:31:53

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 (ASST0 #9)

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
lser0 at lamebus0
con0 at lser0

cpu0: MIPS/161 (System/161 2.x) features 0x0
panic: Fatal exception 3 (TLB miss on store) in kernel mode
panic: EPC 0x8000b504, exception vaddr 0x0
panic: I can't handle this... I think I'll just die now...
sys161: trace: software-requested debugger stop
sys161: Waiting for debugger connection...
```

Summarising, you have configured and installed the applications that will eventually run on OS/161 and the OS/161 kernel itself. Going forward, you would normally only be modifying the kernel (the files in `~/cs3231/asst0-src/kern`) and rebuilding updated OS/161 kernels.

Your Assignment Task

The task for this assignment is to find the bug that is causing OS/161 to crash and fix it. It is an exercise to familiarise yourself with further developing OS/161, and get some practice with GDB and git.

Firstly, lets examine the output above where OS/161 crashes. Skip the initialisation status messages until you reach the point where the *panic* begins. Let's examine each line at a high level (as the semester progresses you'll get a deeper understanding of what is going on).

- A fatal exception has occured, i.e. something OS/161 can't recover from, so it has panic-ed. The exception is related to storing to memory while in kernel mode.
- The EPC (Exception Program Counter) address is the address in memory of the intruction that caused the problem. The `vaddr` is the address it was attempting to store to. You might guess at this point that the code might be attempting to store to a `NULL` pointer -- `NULL` has the value zero when viewed as a integer.
- Yes, we are panic-ing, kernels should not reference `NULL` pointers.
- sys161 is nice enough to stop and wait for you to connect `os161-gdb` to inspect the problem.

Using GDB

GDB is a tool that can help you understand what went wrong with your code. It has the advantage over `kprintf()` style debugging in that it does not change the behaviour of the code, and it can be used when something unexpected happens and not require you to change the code and reproduce the problem in order to debug it. This is very useful for intermittant bugs.

The Wiki has a setup guide for `os161-gdb`. **Now is a good time to set it up.** You'll need it to follow the example below. If you wish to learn more about debugging, see the general debugging material at learn/debugging

Finding the bug

Now we will look at two methods to find the offending code. The first using mostly just GDB, then second method will take advantage of the fact with know the address of the faulting instruction already.

GDB is used via a second terminal window. The table below gives some examples of establishing a GDB session with `sys161`. We'll use a variation of the last row in our example after the table.

	terminal 1	terminal 2

Run OS/161 without GDB	$ cd ~/cs3231/root
	
	$ sys161 kernel	

Debug from the beginning	$ cd ~/cs3231/root
	
	$ sys161 –w kernel
	
	sys161: Waiting for debugger connection..
	
	sys161: New debugger connection
		$ cd ~/cs3231/root
		
		$ os161–gdb kernel
		
		(gdb) connect

	terminal 1	terminal 2

Debug already running instance	$ cd ~/cs3231/root
	
	$ sys161 kernel
	
	OS/161 kernel [? for menu]:
	
	<Execute various commands in OS/161>
	
	ctrl+G (if required)
	
	sys161: Waiting for debugger connection..
	
	sys161: New debugger connection
		$ cd ~/cs3231/root
		
		$ os161-gdb kernel
		
		(gdb) connect

Run GDB as follows, then use the `connect` and `where` commands. Note: If you are using `login.cse.unsw.edu.au`, it is actually and alias for several separate machines. Ensure both terminals are open on the *on the same machine*.

In the second terminal

```
% cd ~/cs3231/root
% os161-gdb kernel
GNU gdb (GDB) 7.8
Copyright (C) 2014 Free Software Foundation, Inc.
License GPLv3+: GNU GPL version 3 or later <http://gnu.org/licen
ses/gpl.html>
This is free software: you are free to change and redistribute i
t.
There is NO WARRANTY, to the extent permitted by law.
Type "show copying" and "show warranty" for details.
This GDB was configured as "--host=x86_64-unknown-linux-gnu --ta
rget=mips-harvard-os161".
Type "show configuration" for configuration details.
For bug reporting instructions, please see: <http://www.gnu.org/
software/gdb/bugs/>.
Find the GDB manual and other documentation resources online at:
<http://www.gnu.org/software/gdb/documentation/>.
For help, type "help".
Type "apropos word" to search for commands related to "word"...
Reading symbols from kernel...done.
(gdb) connect
membar_any_any () at includelinks/machine/membar.h:47
47              __asm volatile(
Breakpoint 1 at 0x8000ac8c: file ../../lib/kprintf.c, line 135.
```

```
(gdb) where
#0  membar_any_any () at includelinks/machine/membar.h:47
#1  membar_store_store () at includelinks/machine/membar.h:58
#2  lamebus_write_register (bus=<optimised out>, slot=<optimised
out>, offset=offset@entry=16, val=val@entry=0) at ../../arch/sys
161/dev/lamebus_machdep.c:184
#3  0x800052ec in ltrace_stop (code=code@entry=0) at ../../dev/l
amebus/ltrace.c:87
#4  0x8000ad58 in panic (fmt=fmt@entry=0x80023bd8 "I can't handl
e this... I think I'll just die now...\n") at ../../lib/kprintf.
c:184
#5  0x8001b2a8 in mips_trap (tf=0x80027f28) at ../../arch/mips/l
ocore/trap.c:315
#6  <signal handler called>
#7  0x8000b504 in boot () at ../../main/main.c:140
#8  0x8000b5d4 in kmain (arguments=0x80026020 "") at ../../main/
main.c:218
#9  0x8001d1ac in __start () at ../../arch/sys161/main/start.S:2
16
(gdb)
```

The `where` command shows the current nesting of function calls as represented by stack frames on the current kernel stack. Here are some observations.

- Frames 9-7 show the kernel starting, entering `kmain`, and then `boot`.
- Frame 6 indicates some kind of exception occured.
- Frames 5 - 0 show the kernel catching the exception, calling panic and stopping.

Let's investigate by switching to frame 7 so we can inspect it's local variables.

```
(gdb) frame 7
#7  0x8000b504 in boot () at ../../main/main.c:140
140                     * foo = 'x';            /* attempt to ac
cess it */
(gdb) print foo
$1 = 0x0
(gdb)
```

The exception occurs while storing the character `x` to the pointer `foo`, which has a value of zero, i.e. `NULL`. Lets use `list` to see more context.

```
(gdb) list
135             kheap_nextgeneration();
136
137             {
138                     /* remove this section of code to fix AS
ST0 */
139                     char *foo = NULL;       /* create a NULL
pointer */
140                     * foo = 'x';            /* attempt to ac
cess it */
141             }
142
```

```
143             /*
144              * Make sure various things aren't screwed up.
(gdb)
```

It should be clear now how to fix the bug

Using EPC and GDB

Let's use our knowledge of EPC to go directly to the problem using the address of the offending instruction.

```
(gdb) list *0x8000b504
0x8000b504 is in boot (../../main/main.c:140).
135             kheap_nextgeneration();
136
137             {
138                     /* remove this section of code to fix AS
ST0 */
139                     char *foo = NULL;       /* create a NULL
pointer */
140                     * foo = 'x';            /* attempt to ac
cess it */
141             }
142
143             /*
144              * Make sure various things aren't screwed up.
(gdb)
```

Here we have been taken directly to the offending statement in the code base. Note that while we can identify the line of code this way, we can't print out local variables unless we are in the correct stack frame, as illustrated below.

```
(gdb) print foo
No symbol "foo" in current context.
(gdb) frame 7
#7  0x8000b504 in boot () at ../../main/main.c:140
140                     *foo = 'x';            /* attempt to acc
ess it */
(gdb) print foo
$2 = 0x0
(gdb)
```

Fixing the bug

Remove the offending code and rebuild the kernel only, and rerun OS/161.

> In `asst0-src/kern/compile/ASST0`.

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

The should get the following when you run sys161 on your new kernel where it runs until you get a boot prompt. You can enter `q` to exit at this point

```
% sys161 kernel
sys161: System/161 release 2.0.8, compiled Feb 19 2017 14:31:53

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 (ASST0 #10)

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
lser0 at lamebus0
con0 at lser0

OS/161 kernel [? for menu]: q
Shutting down.
The system is halted.
sys161: 5411375122 cycles (39129232 run, 5372245890 global-idle)
sys161:   cpu0: 3339292 kern, 0 user, 0 idle; 7328 ll, 7328/0 s
c, 66160 sync
sys161: 783 irqs 0 exns 0r/0w disk 2r/592w console 0r/0w/1m emuf
s 0r/0w net
sys161: Elapsed real time: 213.657611 seconds (25.3273 mhz)
sys161: Elapsed virtual time: 215.027700909 seconds (25 mhz)
%
```

Note: The following sample output will vary slightly depending on the year it was originally generated.

You can also test OS/161 works with the menu choice supplied on the command line as follows.

```
% sys161 kernel q
```

This is generally the way we test your submission, **so make sure that you have tested using the above method, even if you have also tested interactively**.

You should now have a working code base.

Committing your changes with git

First check what your local status is to avoid anything unexpected.

```
% git status
On branch master
Your branch is up-to-date with 'origin/master'.
Changes not staged for commit:
(use "git add <file>..." to update what will be committed)
```

```
(use "git checkout -- <file>..." to discard changes in working d
irectory)

modified:   kern/main/main.c

no changes added to commit (use "git add" and/or "git commit -
a")
%
```

All looks as expected, so lets commit the changes and push them back to `gitlab`. The `-m` is to add a descriptive commit message on the command line. The `-a` tell git to include all files that have been modified in the commit.

```
% git commit -a -m "foo pointer fix"
% git push
```

You now have a completed assigment on your machine and on `gitlab`. Feel free to log into gitlab.cse.unsw.edu.au via the web and explore your assigment.

Submitting

The submission instructions are available on the Wiki. To overview submission, you will be submitting a git repository bundle via CSE's `give` system. For ASST0, the submission system will automark it immediately and tell you if you received the 10 marks. You can submit as many times as you like up until the deadline until you get the marks.

Warning! Don't ignore the submission system! If your submission fails the submission process, you will NOT receive any marks.

Once you have submitted successfully, you're done.

Page last modified: 7:26pm on Monday, 1st of June, 2020

Print Version