COMP3231/9201/3891/9283 Operating Systems 2005/S2

UNSW

GDB and OS/161

This page contains a short tutorial on using GDB with OS/161.

Setting up GDB

Every time you start GDB you will need to tell it the location of your source and how to communicate with system 161. This can become tedious, so we create a shortcut.

Add the following (adjusted for your setup, of course) into your root directory, usually ~/cs3231/root in a file called .gdbinit.

```
define lab2
target remote unix:.sockets/gdb
dir ~/cs3231/lab5/kern/compile/LAB2
set print array on
set print pretty on
b panic
```

Whenever you start GDB in this directory, you can type lab2 and the above commands will be run. Note that we also set a breakpoint at the panic function --- whenever the kernel panics the debugger will be entered. The two "set print" line will make your GDB output a little bit easier to read.

An example problem

Consider the following simple implementation of a communication channel where the sender never blocks and the receiver specifies the ID of the thread it wants to receive messages from. (This is sometimes called a *closed receive*.) Note that this is quite different from the communication channel to be implemented in assignment 1.

```
/* an extremely simple channel which can queue only one message */
struct simple_channel {
    const char *payload;        /* a string passed by reference */
};

/* send a message on a channel without blocking */
void
simple_channel_send(struct simple_channel *handle,
                    const char *payload)
{
    int spl;

    /* assume for this example that we have exclusive access to the channel */
    handle->payload = payload;

    spl = splhigh();
    /* wake up anyone who is expecting a message from me */
    thread_wakeup(curthread);
    splx(spl);
}

/* receive a message from the the thread specified by "from" */
const char *
simple_channel_receive_from(struct thread *from,
                            struct simple_channel *handle)
{
```

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

Staff
- Gabrielle Keller (LiC)

Grievances
- Student Reps


```
    int spl;

    spl = splhigh();
    thread_sleep(from);         /* wait for "from" to send me a message */
    splx(spl);

    return handle->payload;
}
```

For the sake of example, we have also limited each channel to only one message at a time, and have neglected to protect the channel from access by multiple senders.

Setting up the Lab

In order to set up the sample code download the tar ball into ~/cs3231/ and untar by

```
%tar -xvzf lab5.tar.gz
```

This will create a directory call lab5 with a copy of OS161. The file `kern/lab2/simple_sync.c` contains the simple channel implementation. Have a look!

Follow the same instructions as asst0 to configure and compile the kernel.

- You first have to configure your source tree.

  ```
  % cd ~/cs3231/lab5
  % ./configure
  ```

- Now you must configure the kernel itself.

  ```
  % cd ~/cs3231/lab5/kern/conf
  % ./config LAB2
  ```

- The next task is to build the kernel.

  ```
  % cd ../compile/LAB2
  % make depend
  % make
  ```

Note: this will overwrite your ~cs3231/root/ so when you return to doing your normal assignments be sure to repeat the following steps

- Now install the kernel

  ```
  % make install
  ```

- In addition to the kernel, you have to build the user-level utilities.

  ```
  % cd ~/cs3231/src
  % make
  ```

Running your Kernel

Now let's test our simple channel implementation with a couple of test threads which send and receive one message to each other. These are called `simple_thread_A` and `simple_thread_A` in `simple_sync.c`.

- Change to the root directory of your OS.

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

- Now run system/161 (the machine simulator) on your kernel.

  ```
  % sys161 kernel
  ```

- At the prompt type `ss` for the *simple sync test*.

```
$ sys161 kernel ss
sys161: System/161 release 1.1, compiled Feb 24 2003 21:57:51

OS/161 base system version 1.08
Copyright (c) 2000, 2001, 2002, 2003
   President and Fellows of Harvard College.  All rights reserved.

Put-your-group-name-here's system version 0 (LAB2 #11)

Cpu is MIPS r2000/r3000
1876k physical memory available
Device probe...
lamebus0 (system main bus)
emu0 at lamebus0
ltrace0 at lamebus0
ltimer0 at lamebus0
hardclock on ltimer0 (100 hz)
beep0 at ltimer0
rtclock0 at ltimer0
lrandom0 at lamebus0
random0 at lrandom0
lser0 at lamebus0
con0 at lser0
pseudorand0 (virtual)

OS/161 kernel: ss
simple sync program
started thread A
started thread B
hello I'm thread A
hello I'm thread B
```

You can see that simple sync test hasn't terminated. What is the problem? Without a debugger, it can be difficult to know.

Exploring OS161 internals with GDB

Now let us explore OS161's internal structure using GDB. First we need to break out of OS161 and into the debugger. You can do this by pressing `Ctl + g` while OS161 is running. This will look like this.

```
$ sys161 kernel ss
sys161: System/161 release 1.1, compiled Feb 24 2003 21:57:51

OS/161 base system version 1.08
Copyright (c) 2000, 2001, 2002, 2003
   President and Fellows of Harvard College.  All rights reserved.
.
.
.

OS/161 kernel: ss
simple sync program
started thread A
started thread B
hello I'm thread A
hello I'm thread B
(press ctl + g here)
sys161: Waiting for debugger connection...
```

Now we hook GDB up to os161,in another terminal, we change directory to the root directory and run GDB. We run the lab2 command to setup GDB.

```
~/cs3231/root$ cs161-gdb kernel
GNU gdb 4.18
Copyright 1998 Free Software Foundation, Inc.
GDB is free software, covered by the GNU General Public License, and you are
```

```
welcome to change it and/or distribute copies of it under certain conditions.
Type "show copying" to see the conditions.
There is absolutely no warranty for GDB.  Type "show warranty" for details.
This GDB was configured as "--host=i686-pc-linux-gnu --target=mips-elf"...
(gdb) lab2
cpu_idle () at ../../arch/mips/mips/spl.c:155
155             interrupts_onoff();
Breakpoint 1 at 0x800111f4: file ../../lib/kprintf.c, line 94.
(gdb)
```

Let's see which thread is currently running. This can be checked by looking at the global variable in the kernel called `curthread`.

```
(gdb) p curthread
$1 = (struct thread *) 0x0
```

This shows that there is currently no thread running. Let's have a look what the OS is doing by running it for a few more steps.

```
(gdb) next
93              while (q_empty(runqueue)) {
(gdb) next
94                      cpu_idle();
(gdb) next
93              while (q_empty(runqueue)) {
```

Which indicates that the runqueue is empty and there is no thread to schedule.

Next, let's have a look at the number of threads that currently exist the system and what state they're in.

```
(gdb) p numthreads
$2 = 3
(gdb) p *zombies
$3 = {
  num = 0,
  max = 6,
  v = 0x8002dd80
}
(gdb) p *sleepers
$4 = {
  num = 3,
  max = 6,
  v = 0x8002dda0
}
```

We can now conclude that there's currently 3 threads, that none of them is a zombie and they are all sleeping for one reason or another.

Next lets have a look at the threads that are asleep.

```
(gdb) p *((struct thread **) (sleepers->v))[0]
$5 = {
  t_pcb = {
    pcb_switchstack = 2147659208,
    pcb_kstack = 2147659776,
    pcb_ininterrupt = 0,
    pcb_badfaultfunc = 0,
    pcb_copyjmp =       {0 <repeats 11 times>}
  },
  t_name = 0x8002bfc0 "<boot/menu>",
  t_sleepaddr = 0x8002bd00,
  t_stack = 0x0,
  t_vmspace = 0x0,
  t_cwd = 0x8002df40
}
(gdb) p *((struct thread **) (sleepers->v))[1]
```

```
$6 = {
  t_pcb = {
    pcb_switchstack = 2147683844,
    pcb_kstack = 2147684352,
    pcb_ininterrupt = 0,
    pcb_badfaultfunc = 0,
    pcb_copyjmp =       {0 <repeats 11 times>}
  },
  t_name = 0x8002bce0 "thread A",
  t_sleepaddr = 0x8002ce00,
  t_stack = 0x80030000 "®\021Ú3",
  t_vmspace = 0x0,
  t_cwd = 0x8002df40
}
(gdb) p *((struct thread **) (sleepers->v))[2]
$7 = {
  t_pcb = {
    pcb_switchstack = 2147687900,
    pcb_kstack = 2147688448,
    pcb_ininterrupt = 0,
    pcb_badfaultfunc = 0,
    pcb_copyjmp =       {0 <repeats 11 times>}
  },
  t_name = 0x8002bcc0 "thread B",
  t_sleepaddr = 0x8002ce80,
  t_stack = 0x80031000 "®\021Ú3",
  t_vmspace = 0x0,
  t_cwd = 0x8002df40
}
(gdb)
```

The interesting field in the thread control block is the `t_name` and the `t_sleepaddr`. The `t_name` is the name of the thread, which makes identifying the threads very easy. The thread with the name `"<boot/menu>"` is the thread that runs the menu and then the initial test command. In the case of this lab, it is block on a semaphore called "Finished" in `cmd_simple_sync()` which is in `kern/lab2/simple_sync.c`. We can confirm this by printing out the `t_sleepaddr` in the TCB.

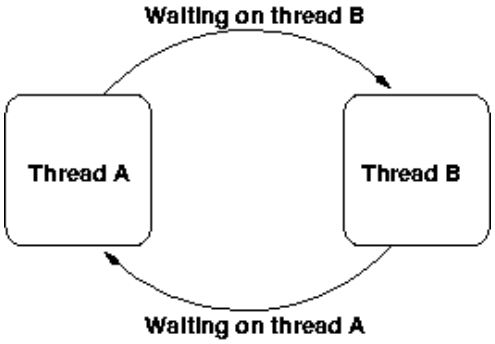
```
(gdb) p *((struct semaphore *) ((*((struct thread **) (sleepers->v))[0])->t_sleepaddr))
$8 = {
  name = 0x8002bcf0 "Finished",
  count = 0
}
```

Now lets take a look at what our threads are blocked on. If we take a look at the code in `kern/lab2/simple_sync.c`, We can see that on `simple_channel_send()` the thread calls `thread_sleep()`on the sender in order to wait for the sender to prepare the message and wake the receiver up when the message is ready. As the address of the sender is given as a sleeping address we can see what thread the sleeping threads are waiting on.

```
(gdb)  p *((struct thread *)  (*((struct thread **) (sleepers->v))[1])->t_sleepaddr)
$9 = {
  t_pcb = {
    pcb_switchstack = 2147687900,
    pcb_kstack = 2147688448,
    pcb_ininterrupt = 0,
    pcb_badfaultfunc = 0,
    pcb_copyjmp =       {0 }
  },
  t_name = 0x8002bcc0 "thread B",
  t_sleepaddr = 0x8002ce80,
  t_stack = 0x80031000 "®\021Ú3",
  t_vmspace = 0x0,
  t_cwd = 0x8002df40
}
(gdb)  p *((struct thread *)  (*((struct thread **) (sleepers->v))[2])->t_sleepaddr)
```

```
$10 = {
  t_pcb = {
    pcb_switchstack = 2147683844,
    pcb_kstack = 2147684352,
    pcb_ininterrupt = 0,
    pcb_badfaultfunc = 0,
    pcb_copyjmp =       {0 }
  },
  t_name = 0x8002bce0 "thread A",
  t_sleepaddr = 0x8002ce00,
  t_stack = 0x80030000 "®\021Ú3",
  t_vmspace = 0x0,
  t_cwd = 0x8002df40
}
```

Aha! The two threads are deadlocked. Thread A is waiting for thread B to send, while thread B is waiting for thread A to send. The following picture illustrates the situation.



Fixing the problem

Now the fun part, using what you have just seen figure out how to solve the deadlock. When your done it should come out like the following.

```
$ sys161 kernel ss
sys161: System/161 release 1.1, compiled Feb 24 2003 21:57:51

OS/161 base system version 1.08
Copyright (c) 2000, 2001, 2002, 2003
   President and Fellows of Harvard College.  All rights reserved.

Put-your-group-name-here's system version 0 (LAB2 #12)

Cpu is MIPS r2000/r3000
1876k physical memory available
Device probe...
lamebus0 (system main bus)
emu0 at lamebus0
ltrace0 at lamebus0
ltimer0 at lamebus0
hardclock on ltimer0 (100 hz)
beep0 at ltimer0
rtclock0 at ltimer0
lrandom0 at lamebus0
random0 at lrandom0
lser0 at lamebus0
con0 at lser0
pseudorand0 (virtual)

OS/161 kernel: ss
simple sync program
started thread A
started thread B
hello I'm thread A
hello I'm thread B
```

```
thread B:I'm sending to thread A
thread A:I got sent ('Hello A, this is B!')
thread A:I'm sending to thread B
thread B:I got sent ('Hello B, this is A!')
Operation took 0.934908680 seconds
OS/161 kernel [? for menu]:
```

Things to be careful about

There are a few things to keep in mind...

- Not every "hang" is a deadlock.
  It could be an infinite loop, it could be just sleeping, or it could be waiting for some input.
- Not all deadlocks are as easy to detect as this one was.
- Threads can sleep on anything, not just semaphores or other threads.

Happy hacking!

Page last modified: 6:09pm on Thursday, 22nd of July, 2004

Print Version