

# Computer Science 161: Operating Systems

## Section 0: Introduction to OS/161

CS161 Course Staff  
cs161@fas.harvard.edu  
<http://www.courses.fas.harvard.edu/~cs161/>  
<http://motelab.eecs.harvard.edu/bb/>

February 9, 2005

### 1 Write Good Code

Throughout this course, you will be adding code to a pre-existing code base. If you follow the conventions already in place, you won't have to think of whether you're interfacing with your code or original code. Also, in general you should strive to write clear, well-structured code. What clear, well-structure code consists of is subjective, but I'll provide some recommendations. For example:

- You'll notice that very few `struct` definitions are `typedef`'d. As a result, functions such as `array_create()` return `struct array *`, rather than the `array *` signature that would result from a `typedef struct array array;`. The bad news is you'll be typing "struct" a lot, the good news is that this confuses `ctags` a lot less than the alternative (see below).
- The compiler is smart. That is, there's no need to be clever by using `x >> 1` to divide by 2, because once the compiler sees the immediate value 2, it knows that it can bit-shift to divide. Simply writing `x/2` is more clear, and equally efficient.
- The preprocessor is smart. Use `#define` and function-like macros when appropriate (for example, see `KVADDR_TO_PADDR` in `kern/arch/mips/include/vm.h`, which does address translation for kernel memory (which is pretty simple because kernel memory is mapped linearly into physical memory in OS/161)), because for simple, repeated operations the overhead of a system call is overkill. Also, if you want to comment out a chunk of code that already has comments, because of the way C comments work, your best bet is often `#if 0 ... #endif`. Realize, though, that macros can't do everything:
  - Be careful to mention an argument to a function-like macro only once, as if you pass an expression into the macro, you don't want to be surprised when that expression is evaluated multiple times (remember, macros are entirely textual substitutions).
  - Basically, put parens around everything, including the full macro.
  - Don't overdo it: you're probably better off using a function if you're doing stuff like branching or assigning temporary variables.
  - `inline` functions should be supported by the `cs161-gcc` and offer a more flexible alternative to macros. Keep in mind that the compiler decides whether or not to actually inline `inline`'d functions.
  - Read <http://gcc.gnu.org/onlinedocs/gcc-3.2.3/cpp> for more.
- Remember, this is C:

- Minimize scope of local variables. A lot of people forget that you can define stack variables at the beginning of any block, so you can do

```
if (x->type == SQUARE) {
    struct square *s = (square *)x->shape
    /*stuff*/
}
```

For debugging purposes it can be useful to open a bare next context to define local variables, since debugging variables defined at the top of a long function frequently get forgotten about when the debugging code is removed. Here's an example of using a bare context:

```
{
    int foo;
    /*debugging stuff*/
}
```

- In case you didn't realize by the above, remember that in C, you're going to be doing a lot of casting and using a lot of `void *` to write pseudo-object-oriented code (for example, the dynamic array implementation we give you has the following signature: `void *array_getguy(struct array *a, int index)`)
- Get your code to compile without warnings. If you're not using an argument to a function, a statement `(void) x;` will get rid of the warning.
- Memory-leaks are very, very bad. For most of our simulation, we'll be running with very little "physical" memory, so any memory you forget to free will quickly bring your system down. When the kernel can't find any data for its accounting structures, you're in trouble. This problem is compounded by the fact that a lot of the code you will be writing will be gradually building structures (viz. allocating memory), aborting if there's a problem. If you read through the OS/161 source (for example, `vfs_doaddd()` in `kern/fs/vfs/vfslist.c`) you'll see pretty much the best option (or at least, not the most awful option) for error recovery in a language without exceptions and with manual memory-management (like C): `goto`'s. Yes, you heard me right. Put cleanup code in an otherwise unreachable section of your code, and jump there if you detect an error; this minimizes duplication of code. At the same time, this is probably the only jumping you should be doing, and you don't need it for every function; just the hairy ones like `rename` and `fork` (which won't be coming up for a while) that would involve lots of duplicated code to clean-up state at each possible error. `goto` style error handling can also be extremely clean when functions involve multiple initializations that might fail:

```
function foobar {
    if (!(foo = kmalloc())) {
        goto out1;
    }
    if (!(bar = kmalloc())) {
        goto out2;
    }
    /*stuff*/
    kfree(bar);
out2:
    kfree(foo);
out1:
    return;
}
```

## 2 Navigate Code Effectively

Read Phil's guide to `ctags(1)` for Code Navigation document at <http://www.courses.fas.harvard.edu/~cs161/handouts/ctags.html>. Tags are your best friend when you're navigating a large source-tree split over many files and directories: you don't have to remember where a function is defined, you just have to remember to hit `Ctrl+]`. Obviously the above was specific to Vim. There are similar tools for Emacs. Regardless of your choice of editor, perhaps one of the most personal choices we can make in our brief time here, tweak it and learn it until yanking/commenting/removing blocks of code is natural.

## 3 Navigate CVS Effectively

More than anything, this involves coordinating with your partner about committing so that you avoid conflicts and the need to merge. While it's tempting to consider CVS as this magical tool that will allow you and your partner to modify code with impunity, it's ability to figure out the Right Thing after several commits to the same file is not astounding. So, if there are two modified source trees, one in each of your accounts, figure out which partner should just `commit`, and which should `update` and then `commit`. Also, something that should be spelled out explicitly: don't commit code that doesn't compile, or causes a kernel panic, or in any other way is a step backward rather than a step forward. One of the skills you will learn in this class is the ability to break a large project into discrete chunks that can be written and tested progressively, rather than getting your code in a state that *should* compile 6 hours before the assignment is due.

CVS is powerful, and should be created with the same respect that `rm` receives. Every time you type a CVS command take a deep breath, think about what you are trying to do, and only after a moment's reflection hit `Enter`.

## 4 GDB

Sooner or later this semester you're going to need to use `gdb`. You'll probably be able to get by doing Assignments 0 and 1 without it, but it's surprisingly powerful and might save you a lot of time later.

### 4.1 Why not `kprintf()`?

- Your kernel is multi-threaded; it's hard to figure out what's going on from just print output.
- `kprintf()` is unwieldy—depending on what you're doing it might give you so much output that it'll be painful to slog through it
- Too slow! You're concerned with development time, not simulator time. For `printf()` debugging, you need to re-compile once every few minutes. Compiling OS/161 is slow<sup>1</sup>.
- It's really hard to inspect complicated data structures with `kprintf()`. It's easier to type `print *some_struct_name`

### 4.2 So how?

#### 4.2.1 Initialization

You need to use two terminals. In the first terminal:

---

<sup>1</sup>21sec on `ice`, 17sec on `nice`. It turns out that it's slow because of NFS; try moving your files to `/tmp` (a local partition) on a workstation machine—it'll compile much faster, but you don't really want to do this.

```
[0]ice3:~(1010)>cd ~/cs161/root
[0]ice3:~/cs161/root(1011)>sys161 -w kernel
sys161: System/161 release 1.11, compiled Feb  3 2004 17:33:22
sys161: Waiting for debugger connection...
```

Then, in the second:

```
[0]ice3:~/cs161/src/kern/compile/ASST5(1074)>cs161-gdb kernel
GNU gdb 6.0
Copyright 2003 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) target remote unix:../../../../../root/.sockets/gdb
Remote debugging using unix:../../../../../root/.sockets/gdb
__start () at ../../arch/mips/mips/start.S:24
24      addiu sp, sp, -20
Current language:  auto; currently asm
```

## 4.2.2 Breakpoints

We continue with breakpoints. Breaking on `panic()` is *really* handy. `panic()` gets called on kernel faults and when assertions fail. (Another debugging hint: use a lot of assertions. You will be really glad you put in a lot of stupid assertions when some assertion from Assignment 1 fails when you're working on Assignment 3.) Then, you can type `bt` to get a backtrace, and use `up` and `down` to navigate up and down the call stack.

```
(gdb) b panic
Breakpoint 1 at 0x8001c6c8: file ../../lib/kprintf.c, line 94.
(gdb) c
Continuing.
```

[Type `panic` into first terminal]

```
Breakpoint 1, panic (fmt=0x80038f98 "User requested panic\n")
  at ../../lib/kprintf.c:94
94      DEBUG(DB_VM, "evil currently is: %d", evil);
Current language:  auto; currently c
(gdb) bt
#0  panic (fmt=0x80038f98 "User requested panic\n") at ../../lib/kprintf.c:94
#1  0xffffffff80027344 in cmd_panic (nargs=1, args=0x8003eef4)
  at ../../main/menu.c:251
#2  0xffffffff800280e8 in cmd_dispatch (cmd=0x8003ef8c "panic")
  at ../../main/menu.c:761
#3  0xffffffff80028214 in menu_execute (line=0x8003ef8c "panic", isargs=0)
  at ../../main/menu.c:803
#4  0xffffffff800282d8 in menu (args=0x8003d5d0 "") at ../../main/menu.c:840
#5  0xffffffff80026cd8 in kmain (arguments=0x8003d5d0 "")
  at ../../main/main.c:190
#6  0xffffffff80017fe8 in __start () at ../../arch/mips/mips/start.S:163
(gdb) up
#1  0xffffffff80027344 in cmd_panic (nargs=1, args=0x8003eef4)
```

```

    at ../../main/menu.c:251
251         panic("User requested panic\n");
(gdb) up
#2 0xffffffff800280e8 in cmd_dispatch (cmd=0x8003ef8c "panic")
    at ../../main/menu.c:761
761         result = cmdtable[i].func(nargs, args);
(gdb) up
#3 0xffffffff80028214 in menu_execute (line=0x8003ef8c "panic", isargs=0)
    at ../../main/menu.c:803
803         result = cmd_dispatch(command);
(gdb) print *command
$1 = 112 'p'
(gdb) print command
$2 = 0x8003ef8c "panic"

```

### 4.2.3 Inspecting Memory

When you're working on, say, the `exec()` system call, you'll have to do a lot of careful pointer arithmetic. It turns out that it's really helpful to examine memory. Let's try it out with my implementation:

```

(gdb) l sys_exec
/* shows code for sys_exec() function */
(gdb) b 715
Breakpoint 3 at 0x8002b47c: file ../../userprog/proc_syscall.c, line 715.
(gdb) c
/* Type "s" to get a shell; then "/testbin/argtest cs161 rulez" */
Breakpoint 3, sys_exec (tf=0x80052f6c) at ../../userprog/proc_syscall.c:715
715         ret = copyargs_to_userland(exec_argbuf, &stackptr, totlen, nargs);
Current language: auto; currently c
(gdb) print (void *)stackptr
$4 = (void *) 0x80000000
(gdb) x /31c exec_argbuf
0x8003cdc0 <exec_argbuf_real>: 47 '/' 116 't' 101 'e' 115 's' 116 't' 98 'b' .. 110 'n'
0x8003cdc8 <exec_argbuf_real+8>: 47 '/' 97 'a' 114 'r' 103 'g' 116 't'.. 116 't'
0x8003cdd0 <exec_argbuf_real+16>: 0 '\0' 99 'c' 115 's' 49 '1' 54 '6' .. 114 'r'
0x8003cdd8 <exec_argbuf_real+24>: 117 'u' 108 'l' 101 'e' 122 'z' 0 '\0' .. 0 '\0'
(gdb) n
(gdb) print (void *) stackptr
$7 = (void *) 0x7fffffd0
(gdb) x /50c stackptr
0x7fffffd0: 127 '\177' -1 '?' -1 '?' -30 '?' 127 '\177' -1 '?' .. -13 '?'
0x7fffffd8: 127 '\177' -1 '?' -1 '?' -7 '?' 0 '\0' 0 '\0' 0 '\0' 0 '\0'
0x7fffffe0: 0 '\0' 0 '\0' 47 '/' 116 't' 101 'e' 115 's' 116 't' 98 'b'
0x7fffffe8: 105 'i' 110 'n' 47 '/' 97 'a' 114 'r' 103 'g' 116 't' 101 'e'
0x7fffff0: 115 's' 116 't' 0 '\0' 99 'c' 115 's' 49 '1' 54 '6' 49 '1'
0x7fffff8: 0 '\0' 114 'r' 117 'u' 108 'l' 101 'e' 122 'z' 0 '\0' 0 '\0'
0x80000000: 3 '\003' -96 '?'

```

### 4.2.4 More

There are lots of other useful gdb commands. Take a look at <http://www.courses.fas.harvard.edu/~cs161/handouts/gdb.html>.

### 4.2.5 Can I make it pretty?

It's worthwhile to fuss a bit to now and work on your development environment. Definitely use `.gdbinit` to alleviate common typing (like that `remote`) command. Write some shell scripts to set some stuff.

I setup `screen(1)` to give me a split-screen debugging view that simulated two windows in one. There are instructions at <http://www.courses.fas.harvard.edu/~cs161/handouts/screen.html>. It is conceivably possible to use `xxx` or `ddd` (X11 graphical debuggers) to debug with a graphical interface, but I don't know anyone who's done it. If you use the `emacs(1)` Operating System, there's probably a `gdb` mode in there somewhere.

If you do this stuff, share it with your classmates by posting to the bulletin board.

### 4.3 When to debug?

If you've set it up nicely, you can be in `gdb` all the time, with a breakpoint on `panic()`. Obviously you don't want to do performance analysis while debugging, but on the whole you might as well use it.<sup>2</sup>

Some people say that you should step through every line of code that you write in the debugger. It's not a bad idea. You'll definitely want to step through tricky code.

### 4.4 One More Note

Don't be afraid of adding functionality to OS/161 if it seems worthwhile for debugging, even if it's not in the assignment. It's pretty easy to add a menu item to print some internal data out (or to make runtime changes). You could add a system call that lets user-level programs write integers via `kprintf()` (useful because one of you will be working on `exec()` while your partner works on `write()`).

## 5 The CS161 Toolchain

### 5.1 What is the CS161 toolchain?

The CS161 toolchain is the set of utilities (other than `sys161`) that you need to develop for the MIPS r2000/r3000. Again, we've got it compiled and built by default for you. If you want to work at home, you'll need to download and build this yourself.

### 5.2 Compilation/Debugging tools:

- `cs161-gcc` - the compiler for MIPS r2000/r3000. You should probably never have to invoke this directly.
- `cs161-gdb` - the debugger, which you will have to invoke more often than you care.
- `cs161-{ld, as, ar, ranlib, ...}` - all the normal build utilities you know and love.

### 5.3 Special Debugging Tools:

#### 5.3.1 `stat161`

**Description:**

`stat161` displays useful statistics about how the various components of `sys161/os161`. This can be useful in profiling and detecting deadlocks.

---

<sup>2</sup>Be grateful you have a debugger. The Mica2 notes that Matt does research with have three LEDs to debug with which you're supposed to infer their state.

**Usage:**

In order to use `stat161`, run the kernel and then run `stat161` from a different terminal session (make sure you're logged into the same box and are in the system root directory).

**Interpreting stat161 output:**

- `kern` - cycles spent executing kernel code.
- `user` - cycles spent executing user level code.
- `idle` - idle cycles.
- `irqs` - interrupts raised.
- `exns` - exceptions.
- `disk` - disk usage.
- `con` - console usage.
- `emu` - emufs usage.
- `net` - network interface usage.

**Identifying deadlocks:**

If running a user-level program (e.g. user-level tests), check for user activity. Look for time spent in kernel, if this is about the same the entire time, that's usually a good sign of deadlock ( $\approx 33000$  is the standard, if you're seeing just that, either your system is idle or there's a good chance you're in a deadlock). If thread uses disks, look for disk activity. `stat161` is an important tool for figuring out if your kernel is livelocked, deadlocked, or just really dog slow.

**5.3.2 cs161-objdump****Description:**

`cs161-objdump` can be used to disassemble user level programs. This is useful for comparing with trace output (see `trace161`) to ensure you're mapping the right pages or to locate the source of exceptions. You can also disassemble within `gdb` by using the `disassemble` command or `x/i`.

**Usage:**

To disassemble a program:

```
cs161-objdump --disassemble --source program
```

If you compiled the program with debug information, you can do the following to mix the source code in with the assembly:

```
cs161-objdump --disassemble --source program
```

Make sure you are actually disassembling the same copy of the program that you are running, or you will get very confused.

**5.3.3 cs161-addr2line****Description:**

`cs161-addr2line` converts an address in hex to a line number, so you can figure out where your program crashed and burned. When the kernel crashes, it prints the offending EPC. Using `cs161-addr2line` on this EPC gives the line number of corresponding line of code. This isn't incredibly useful though, so we hope that had a debugger attached breaking on `panic`

**Usage:**

```
cs161-addr2line hexaddr
```

### 5.3.4 trace161

#### Description:

`trace161` is useful for tracing through the execution of both user level programs and the kernel. Unfortunately, we can't debug user level programs, but we can trace them. Use `ltrace_on()/ltrace_off()` to turn tracing on/off under software control (see below).

#### Usage:

Using `trace161` instead of `sys161`, you can have it report various execution details, right down to the level of individual instructions. `trace161` has various useful options (you can list these by just running `trace161`). The most commonly used ones are:

- `trace161 -tu kernel` - reports every instruction executed in user mode. Useful when compared with the disassembly of the program, to ensure that you are mapping the right pages into memory.
- `trace161 -tk kernel` - reports every instruction executed in kernel (NOTE: in general, produces too much output to be useful).

### 5.3.5 ltrace\_debug()

#### Description:

`ltrace_debug` is useful for printing simple indications that a certain piece of code has been reached, like one might use `kprintf`, except that it is less invasive than `kprintf`. Think of it as setting the value of a readout on the system's front panel. (In real life, since computers don't have front panels with blinking lights any more, people often use the speaker or the top left corner of the screen for this purpose.) Although we don't recommend `printf` debugging, sometimes it can be useful (e.g. examining every `vm_fault` breakpoint may be inefficient). In those cases, using `ltrace_debug` is less likely to cause the bug to disappear (e.g. race conditions can be difficult to find using `kprintf` since a `kprintf` call can change the scheduler behavior. `ltrace_debug` is less likely to cause this).

#### Usage:

`ltrace_debug(CODE)` causes `sys161/trace161` to print a message with `CODE`, where `CODE` is a `textttu_int32_t`.

### 5.3.6 ltrace\_on()/ltrace\_off()

#### Description:

Turn tracing on/off under software control using `ltrace_on()/ltrace_off()`.

#### Usage:

`ltrace_on()` turns on the `trace161` tracing flag `CODE`.

`ltrace_off()` turns off the `trace161` tracing flag `CODE`.

### 5.3.7 kmalloc heap integrity mode

**Description:** `kmalloc` has two pound defines, `SLOW` and `SLOWER`, that turn on kernel heap integrity checking (these are off by default, as they make the system quite slow). These will catch various kinds of `malloc` bugs and thus can be useful in debugging. (When you write your VM system, you might want to use a similar `SLOW` mechanism to introduce some consistency checks of your own.)

### 5.3.8 profiling

#### Description:

Profiling is useful for performance analysis and the call graph can be useful for debugging. Check out `gprof`.