# Appendix E Debugging Tools

Many tools lie at your disposal for debugging Pintos. This appendix introduces you to a few of them.

## E.1 `printf()`

Don't underestimate the value of `printf()`. The way `printf()` is implemented in Pintos, you can call it from practically anywhere in the kernel, whether it's in a kernel thread or an interrupt handler, almost regardless of what locks are held.

`printf()` is useful for more than just examining data. It can also help figure out when and where something goes wrong, even when the kernel crashes or panics without a useful error message. The strategy is to sprinkle calls to `printf()` with different strings (e.g. `"<1>"`, `"<2>"`, ...) throughout the pieces of code you suspect are failing. If you don't even see `<1>` printed, then something bad happened before that point, if you see `<1>` but not `<2>`, then something bad happened between those two points, and so on. Based on what you learn, you can then insert more `printf()` calls in the new, smaller region of code you suspect. Eventually you can narrow the problem down to a single statement. See Section E.6 [Triple Faults], page 111, for a related technique.

## E.2 `ASSERT`

Assertions are useful because they can catch problems early, before they'd otherwise be noticed. Ideally, each function should begin with a set of assertions that check its arguments for validity. (Initializers for functions' local variables are evaluated before assertions are checked, so be careful not to assume that an argument is valid in an initializer.) You can also sprinkle assertions throughout the body of functions in places where you suspect things are likely to go wrong. They are especially useful for checking loop invariants.

Pintos provides the `ASSERT` macro, defined in '`<debug.h>`', for checking assertions.

`ASSERT` (*expression*) [Macro]
: Tests the value of *expression*. If it evaluates to zero (false), the kernel panics. The panic message includes the expression that failed, its file and line number, and a backtrace, which should help you to find the problem. See Section E.4 [Backtraces], page 103, for more information.

## E.3 Function and Parameter Attributes

These macros defined in '`<debug.h>`' tell the compiler special attributes of a function or function parameter. Their expansions are GCC-specific.

`UNUSED` [Macro]
: Appended to a function parameter to tell the compiler that the parameter might not be used within the function. It suppresses the warning that would otherwise appear.

`NO_RETURN` [Macro]
: Appended to a function prototype to tell the compiler that the function never returns. It allows the compiler to fine-tune its warnings and its code generation.

`NO_INLINE` [Macro]

Appended to a function prototype to tell the compiler to never emit the function in-line. Occasionally useful to improve the quality of backtraces (see below).

`PRINTF_FORMAT (`*`format`*`, `*`first`*`)` [Macro]

Appended to a function prototype to tell the compiler that the function takes a `printf()`-like format string as the argument numbered *format* (starting from 1) and that the corresponding value arguments start at the argument numbered *first*. This lets the compiler tell you if you pass the wrong argument types.

## E.4 Backtraces

When the kernel panics, it prints a "backtrace," that is, a summary of how your program got where it is, as a list of addresses inside the functions that were running at the time of the panic. You can also insert a call to `debug_backtrace()`, prototyped in '`<debug.h>`', to print a backtrace at any point in your code. `debug_backtrace_all()`, also declared in '`<debug.h>`', prints backtraces of all threads.

The addresses in a backtrace are listed as raw hexadecimal numbers, which are difficult to interpret. We provide a tool called `backtrace` to translate these into function names and source file line numbers. Give it the name of your '`kernel.o`' as the first argument and the hexadecimal numbers composing the backtrace (including the '`0x`' prefixes) as the remaining arguments. It outputs the function name and source file line numbers that correspond to each address.

If the translated form of a backtrace is garbled, or doesn't make sense (e.g. function A is listed above function B, but B doesn't call A), then it's a good sign that you're corrupting a kernel thread's stack, because the backtrace is extracted from the stack. Alternatively, it could be that the '`kernel.o`' you passed to `backtrace` is not the same kernel that produced the backtrace.

Sometimes backtraces can be confusing without any corruption. Compiler optimizations can cause surprising behavior. When a function has called another function as its final action (a *tail call*), the calling function may not appear in a backtrace at all. Similarly, when function A calls another function B that never returns, the compiler may optimize such that an unrelated function C appears in the backtrace instead of A. Function C is simply the function that happens to be in memory just after A. In the threads project, this is commonly seen in backtraces for test failures; see [`pass()` Fails], page 18, for more information.

### E.4.1 Example

Here's an example. Suppose that Pintos printed out this following call stack, which is taken from an actual Pintos submission for the file system project:

```
Call stack: 0xc0106eff 0xc01102fb 0xc010dc22 0xc010cf67 0xc0102319
0xc010325a 0x804812c 0x8048a96 0x8048ac8.
```

You would then invoke the `backtrace` utility like shown below, cutting and pasting the backtrace information into the command line. This assumes that '`kernel.o`' is in the current directory. You would of course enter all of the following on a single shell command line, even though that would overflow our margins here:

```
backtrace kernel.o 0xc0106eff 0xc01102fb 0xc010dc22 0xc010cf67
0xc0102319 0xc010325a 0x804812c 0x8048a96 0x8048ac8
```

The backtrace output would then look something like this:

```
0xc0106eff: debug_panic (lib/debug.c:86)
0xc01102fb: file_seek (filesys/file.c:405)
0xc010dc22: seek (userprog/syscall.c:744)
0xc010cf67: syscall_handler (userprog/syscall.c:444)
0xc0102319: intr_handler (threads/interrupt.c:334)
0xc010325a: intr_entry (threads/intr-stubs.S:38)
0x0804812c: (unknown)
0x08048a96: (unknown)
0x08048ac8: (unknown)
```

(You will probably not see exactly the same addresses if you run the command above on your own kernel binary, because the source code you compiled and the compiler you used are probably different.)

The first line in the backtrace refers to `debug_panic()`, the function that implements kernel panics. Because backtraces commonly result from kernel panics, `debug_panic()` will often be the first function shown in a backtrace.

The second line shows `file_seek()` as the function that panicked, in this case as the result of an assertion failure. In the source code tree used for this example, line 405 of '`filesys/file.c`' is the assertion

```
ASSERT (file_ofs >= 0);
```

(This line was also cited in the assertion failure message.) Thus, `file_seek()` panicked because it passed a negative file offset argument.

The third line indicates that `seek()` called `file_seek()`, presumably without validating the offset argument. In this submission, `seek()` implements the `seek` system call.

The fourth line shows that `syscall_handler()`, the system call handler, invoked `seek()`.

The fifth and sixth lines are the interrupt handler entry path.

The remaining lines are for addresses below `PHYS_BASE`. This means that they refer to addresses in the user program, not in the kernel. If you know what user program was running when the kernel panicked, you can re-run `backtrace` on the user program, like so: (typing the command on a single line, of course):

```
backtrace tests/filesys/extended/grow-too-big 0xc0106eff 0xc01102fb
0xc010dc22 0xc010cf67 0xc0102319 0xc010325a 0x804812c 0x8048a96
0x8048ac8
```

The results look like this:

```
0xc0106eff: (unknown)
0xc01102fb: (unknown)
0xc010dc22: (unknown)
0xc010cf67: (unknown)
0xc0102319: (unknown)
0xc010325a: (unknown)
0x0804812c: test_main (...xtended/grow-too-big.c:20)
0x08048a96: main (tests/main.c:10)
```

```
0x08048ac8: _start (lib/user/entry.c:9)
```

You can even specify both the kernel and the user program names on the command line, like so:

```
backtrace kernel.o tests/filesys/extended/grow-too-big 0xc0106eff
0xc01102fb 0xc010dc22 0xc010cf67 0xc0102319 0xc010325a 0x804812c
0x8048a96 0x8048ac8
```

The result is a combined backtrace:

```
In kernel.o:
0xc0106eff: debug_panic (lib/debug.c:86)
0xc01102fb: file_seek (filesys/file.c:405)
0xc010dc22: seek (userprog/syscall.c:744)
0xc010cf67: syscall_handler (userprog/syscall.c:444)
0xc0102319: intr_handler (threads/interrupt.c:334)
0xc010325a: intr_entry (threads/intr-stubs.S:38)
In tests/filesys/extended/grow-too-big:
0x0804812c: test_main (...xtended/grow-too-big.c:20)
0x08048a96: main (tests/main.c:10)
0x08048ac8: _start (lib/user/entry.c:9)
```

Here's an extra tip for anyone who read this far: `backtrace` is smart enough to strip the `Call stack:` header and '`.`' trailer from the command line if you include them. This can save you a little bit of trouble in cutting and pasting. Thus, the following command prints the same output as the first one we used:

```
backtrace kernel.o Call stack: 0xc0106eff 0xc01102fb 0xc010dc22
0xc010cf67 0xc0102319 0xc010325a 0x804812c 0x8048a96 0x8048ac8.
```

## E.5 GDB

You can run Pintos under the supervision of the GDB debugger. First, start Pintos with the '`--gdb`' option, e.g. `pintos --gdb -- run mytest`. Second, open a second terminal on the same machine and use `pintos-gdb` to invoke GDB on '`kernel.o`':[1]

```
pintos-gdb kernel.o
```

and issue the following GDB command:

```
target remote localhost:1234
```

Now GDB is connected to the simulator over a local network connection. You can now issue any normal GDB commands. If you issue the '`c`' command, the simulated BIOS will take control, load Pintos, and then Pintos will run in the usual way. You can pause the process at any point with ⟨Ctrl+C⟩.

### E.5.1 Using GDB

You can read the GDB manual by typing `info gdb` at a terminal command prompt. Here's a few commonly useful GDB commands:

[1] `pintos-gdb` is a wrapper around `gdb` (80x86) or `i386-elf-gdb` (SPARC) that loads the Pintos macros at startup.

`c` [GDB Command]

Continues execution until ⟨Ctrl+C⟩ or the next breakpoint.

`break` *function* [GDB Command]
`break` *file:line* [GDB Command]
`break` **address* [GDB Command]

Sets a breakpoint at *function*, at *line* within *file*, or *address*. (Use a '`0x`' prefix to specify an address in hex.)

Use `break main` to make GDB stop when Pintos starts running.

`p` *expression* [GDB Command]

Evaluates the given *expression* and prints its value. If the expression contains a function call, that function will actually be executed.

`l` **address* [GDB Command]

Lists a few lines of code around *address*. (Use a '`0x`' prefix to specify an address in hex.)

`bt` [GDB Command]

Prints a stack backtrace similar to that output by the `backtrace` program described above.

`p/a` *address* [GDB Command]

Prints the name of the function or variable that occupies *address*. (Use a '`0x`' prefix to specify an address in hex.)

`diassemble` *function* [GDB Command]

Disassembles *function*.

We also provide a set of macros specialized for debugging Pintos, written by Godmar Back `gback@cs.vt.edu`. You can type `help user-defined` for basic help with the macros. Here is an overview of their functionality, based on Godmar's documentation:

`debugpintos` [GDB Macro]

Attach debugger to a waiting pintos process on the same machine. Shorthand for `target remote localhost:1234`.

`dumplist` *list type element* [GDB Macro]

Prints the elements of *list*, which should be a `struct` list that contains elements of the given *type* (without the word `struct`) in which *element* is the `struct list_elem` member that links the elements.

Example: `dumplist all_list thread allelem` prints all elements of `struct thread` that are linked in `struct list all_list` using the `struct list_elem allelem` which is part of `struct thread`.

`btthread` *thread* [GDB Macro]

Shows the backtrace of *thread*, which is a pointer to the `struct thread` of the thread whose backtrace it should show. For the current thread, this is identical to the `bt` (backtrace) command. It also works for any thread suspended in `schedule()`, provided you know where its kernel stack page is located.

`btthreadlist` *list element* [GDB Macro]

Shows the backtraces of all threads in *list*, the `struct list` in which the threads are kept. Specify *element* as the `struct list_elem` field used inside `struct thread` to link the threads together.

Example: `btthreadlist all_list allelem` shows the backtraces of all threads contained in `struct list all_list`, linked together by `allelem`. This command is useful to determine where your threads are stuck when a deadlock occurs. Please see the example scenario below.

`btthreadall` [GDB Macro]

Short-hand for `btthreadlist all_list allelem`.

`btpagefault` [GDB Macro]

Print a backtrace of the current thread after a page fault exception. Normally, when a page fault exception occurs, GDB will stop with a message that might say:[2]

```
Program received signal 0, Signal 0.
0xc0102320 in intr0e_stub ()
```

In that case, the `bt` command might not give a useful backtrace. Use `btpagefault` instead.

You may also use `btpagefault` for page faults that occur in a user process. In this case, you may wish to also load the user program's symbol table using the `loadusersymbols` macro, as described above.

`hook-stop` [GDB Macro]

GDB invokes this macro every time the simulation stops, which Bochs will do for every processor exception, among other reasons. If the simulation stops due to a page fault, `hook-stop` will print a message that says and explains further whether the page fault occurred in the kernel or in user code.

If the exception occurred from user code, `hook-stop` will say:

```
pintos-debug: a page fault exception occurred in user mode
pintos-debug: hit 'c' to continue, or 's' to step to intr_handler
```

In Project 2, a page fault in a user process leads to the termination of the process. You should expect those page faults to occur in the robustness tests where we test that your kernel properly terminates processes that try to access invalid addresses. To debug those, set a break point in `page_fault()` in '`exception.c`', which you will need to modify accordingly.

In Project 3, a page fault in a user process no longer automatically leads to the termination of a process. Instead, it may require reading in data for the page the process was trying to access, either because it was swapped out or because this is the first time it's accessed. In either case, you will reach `page_fault()` and need to take the appropriate action there.

If the page fault did not occur in user mode while executing a user process, then it occurred in kernel mode while executing kernel code. In this case, `hook-stop` will print this message:

[2] To be precise, GDB will stop only when running under Bochs. When running under QEMU, you must set a breakpoint in the `page_fault` function to stop execution when a page fault occurs. In that case, the `btpagefault` macro is unnecessary.

```
pintos-debug: a page fault occurred in kernel mode
```

followed by the output of the `btpagefault` command.

Before Project 3, a page fault exception in kernel code is always a bug in your kernel, because your kernel should never crash. Starting with Project 3, the situation will change if you use the `get_user()` and `put_user()` strategy to verify user memory accesses (see Section 3.1.5 [Accessing User Memory], page 27).

### E.5.2 Example GDB Session

This section narrates a sample GDB session, provided by Godmar Back. This example illustrates how one might debug a Project 1 solution in which occasionally a thread that calls `timer_sleep()` is not woken up. With this bug, tests such as `mlfqs_load_1` get stuck.

This session was captured with a slightly older version of Bochs and the GDB macros for Pintos, so it looks slightly different than it would now. Program output is shown in normal type, user input in **strong** type.

First, I start Pintos:

```
$ pintos -v –gdb – -q -mlfqs run mlfqs-load-1
Writing command line to /tmp/gDAlqTB5Uf.dsk...
bochs -q
========================================================================
                       Bochs x86 Emulator 2.2.5
             Build from CVS snapshot on December 30, 2005
========================================================================
00000000000i[     ] reading configuration from bochsrc.txt
00000000000i[     ] Enabled gdbstub
00000000000i[     ] installing nogui module as the Bochs GUI
00000000000i[     ] using log file bochsout.txt
Waiting for gdb connection on localhost:1234
```

Then, I open a second window on the same machine and start GDB:

```
$ pintos-gdb kernel.o
GNU gdb Red Hat Linux (6.3.0.0-1.84rh)
Copyright 2004 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 "i386-redhat-linux-gnu"...
Using host libthread_db library "/lib/libthread_db.so.1".
```

Then, I tell GDB to attach to the waiting Pintos emulator:

```
(gdb) debugpintos
Remote debugging using localhost:1234
0x0000fff0 in ?? ()
Reply contains invalid hex digit 78
```

Now I tell Pintos to run by executing `c` (short for `continue`) twice:

```
(gdb) c
Continuing.
Reply contains invalid hex digit 78
(gdb) c
Continuing.
```

Now Pintos will continue and output:

```
Pintos booting with 4,096 kB RAM...
Kernel command line: -q -mlfqs run mlfqs-load-1
374 pages available in kernel pool.
373 pages available in user pool.
Calibrating timer...  102,400 loops/s.
Boot complete.
Executing 'mlfqs-load-1':
(mlfqs-load-1) begin
(mlfqs-load-1) spinning for up to 45 seconds, please wait...
(mlfqs-load-1) load average rose to 0.5 after 42 seconds
(mlfqs-load-1) sleeping for another 10 seconds, please wait...
```

. . . until it gets stuck because of the bug I had introduced. I hit ⟨Ctrl+C⟩ in the debugger window:

```
Program received signal 0, Signal 0.
0xc010168c in next_thread_to_run () at ../../threads/thread.c:649
649   while (i <= PRI_MAX && list_empty (&ready_list[i]))
(gdb)
```

The thread that was running when I interrupted Pintos was the idle thread. If I run `backtrace`, it shows this backtrace:

```
(gdb) bt
#0  0xc010168c in next_thread_to_run () at ../../threads/thread.c:649
#1  0xc0101778 in schedule () at ../../threads/thread.c:714
#2  0xc0100f8f in thread_block () at ../../threads/thread.c:324
#3  0xc0101419 in idle (aux=0x0) at ../../threads/thread.c:551
#4  0xc010145a in kernel_thread (function=0xc01013ff , aux=0x0)
    at ../../threads/thread.c:575
#5  0x00000000 in ?? ()
```

Not terribly useful. What I really like to know is what's up with the other thread (or threads). Since I keep all threads in a linked list called `all_list`, linked together by a `struct list_elem` member named `allelem`, I can use the `btthreadlist` macro from the macro library I wrote. `btthreadlist` iterates through the list of threads and prints the backtrace for each thread:

```
(gdb) btthreadlist all_list allelem
pintos-debug: dumping backtrace of thread 'main' @0xc002f000
#0  0xc0101820 in schedule () at ../../threads/thread.c:722
#1  0xc0100f8f in thread_block () at ../../threads/thread.c:324
#2  0xc0104755 in timer_sleep (ticks=1000) at ../../devices/timer.c:141
#3  0xc010bf7c in test_mlfqs_load_1 () at ../../tests/threads/mlfqs-load-1.c:49
#4  0xc010aabb in run_test (name=0xc0007d8c "mlfqs-load-1")
    at ../../tests/threads/tests.c:50
#5  0xc0100647 in run_task (argv=0xc0110d28) at ../../threads/init.c:281
#6  0xc0100721 in run_actions (argv=0xc0110d28) at ../../threads/init.c:331
#7  0xc01000c7 in main () at ../../threads/init.c:140

pintos-debug: dumping backtrace of thread 'idle' @0xc0116000
#0  0xc010168c in next_thread_to_run () at ../../threads/thread.c:649
#1  0xc0101778 in schedule () at ../../threads/thread.c:714
#2  0xc0100f8f in thread_block () at ../../threads/thread.c:324
#3  0xc0101419 in idle (aux=0x0) at ../../threads/thread.c:551
#4  0xc010145a in kernel_thread (function=0xc01013ff , aux=0x0)
    at ../../threads/thread.c:575
#5  0x00000000 in ?? ()
```

In this case, there are only two threads, the idle thread and the main thread. The kernel stack pages (to which the `struct thread` points) are at `0xc0116000` and `0xc002f000`, respectively. The main thread is stuck in `timer_sleep()`, called from `test_mlfqs_load_1`.

Knowing where threads are stuck can be tremendously useful, for instance when diagnosing deadlocks or unexplained hangs.

`loadusersymbols` [GDB Macro]

You can also use GDB to debug a user program running under Pintos. To do that, use the `loadusersymbols` macro to load the program's symbol table:

```
loadusersymbols program
```

where *program* is the name of the program's executable (in the host file system, not in the Pintos file system). For example, you may issue:

```
(gdb) loadusersymbols tests/userprog/exec-multiple
add symbol table from file "tests/userprog/exec-multiple" at
    .text_addr = 0x80480a0
(gdb)
```

After this, you should be able to debug the user program the same way you would the kernel, by placing breakpoints, inspecting data, etc. Your actions apply to every user program running in Pintos, not just to the one you want to debug, so be careful in interpreting the results: GDB does not know which process is currently active (because that is an abstraction the Pintos kernel creates). Also, a name that appears in both the kernel and the user program will actually refer to the kernel name. (The latter problem can be avoided by giving the user executable name on the GDB command line, instead of '`kernel.o`', and then using `loadusersymbols` to load '`kernel.o`'.) `loadusersymbols` is implemented via GDB's `add-symbol-file` command.

### E.5.3 FAQ

GDB can't connect to Bochs.

If the `target remote` command fails, then make sure that both GDB and `pintos` are running on the same machine by running `hostname` in each terminal. If the names printed differ, then you need to open a new terminal for GDB on the machine running `pintos`.

GDB doesn't recognize any of the macros.

If you start GDB with `pintos-gdb`, it should load the Pintos macros automatically. If you start GDB some other way, then you must issue the command `source pintosdir/src/misc/gdb-macros`, where *pintosdir* is the root of your Pintos directory, before you can use them.

Can I debug Pintos with DDD?

Yes, you can. DDD invokes GDB as a subprocess, so you'll need to tell it to invokes `pintos-gdb` instead:

```
ddd --gdb --debugger pintos-gdb
```

Can I use GDB inside Emacs?

Yes, you can. Emacs has special support for running GDB as a subprocess. Type *M-x gdb* and enter your `pintos-gdb` command at the prompt. The Emacs manual has information on how to use its debugging features in a section titled "Debuggers."

GDB is doing something weird.

> If you notice strange behavior while using GDB, there are three possibilities: a bug in your modified Pintos, a bug in Bochs's interface to GDB or in GDB itself, or a bug in the original Pintos code. The first and second are quite likely, and you should seriously consider both. We hope that the third is less likely, but it is also possible.

## E.6 Triple Faults

When a CPU exception handler, such as a page fault handler, cannot be invoked because it is missing or defective, the CPU will try to invoke the "double fault" handler. If the double fault handler is itself missing or defective, that's called a "triple fault." A triple fault causes an immediate CPU reset.

Thus, if you get yourself into a situation where the machine reboots in a loop, that's probably a "triple fault." In a triple fault situation, you might not be able to use `printf()` for debugging, because the reboots might be happening even before everything needed for `printf()` is initialized.

There are at least two ways to debug triple faults. First, you can run Pintos in Bochs under GDB (see Section E.5 [GDB], page 105). If Bochs has been built properly for Pintos, a triple fault under GDB will cause it to print the message "Triple fault: stopping for gdb" on the console and break into the debugger. (If Bochs is not running under GDB, a triple fault will still cause it to reboot.) You can then inspect where Pintos stopped, which is where the triple fault occurred.

Another option is what I call "debugging by infinite loop." Pick a place in the Pintos code, insert the infinite loop `for (;;);` there, and recompile and run. There are two likely possibilities:

- The machine hangs without rebooting. If this happens, you know that the infinite loop is running. That means that whatever caused the reboot must be *after* the place you inserted the infinite loop. Now move the infinite loop later in the code sequence.
- The machine reboots in a loop. If this happens, you know that the machine didn't make it to the infinite loop. Thus, whatever caused the reboot must be *before* the place you inserted the infinite loop. Now move the infinite loop earlier in the code sequence.

If you move around the infinite loop in a "binary search" fashion, you can use this technique to pin down the exact spot that everything goes wrong. It should only take a few minutes at most.

## E.7 Modifying Bochs

An advanced debugging technique is to modify and recompile the simulator. This proves useful when the simulated hardware has more information than it makes available to the OS. For example, page faults have a long list of potential causes, but the hardware does not report to the OS exactly which one is the particular cause. Furthermore, a bug in the kernel's handling of page faults can easily lead to recursive faults, but a "triple fault" will cause the CPU to reset itself, which is hardly conducive to debugging.

In a case like this, you might appreciate being able to make Bochs print out more debug information, such as the exact type of fault that occurred. It's

not very hard. You start by retrieving the source code for Bochs 2.2.6 from `http://bochs.sourceforge.net` and saving the file '`bochs-2.2.6.tar.gz`' into a directory. The script '`pintos/src/misc/bochs-2.2.6-build.sh`' applies a number of patches contained in '`pintos/src/misc`' to the Bochs tree, then builds Bochs and installs it in a directory of your choice. Run this script without arguments to learn usage instructions. To use your '`bochs`' binary with `pintos`, put it in your `PATH`, and make sure that it is earlier than '`/usr/class/cs140/`‘`uname -m`‘`/bin/bochs`'.

Of course, to get any good out of this you'll have to actually modify Bochs. Instructions for doing this are firmly out of the scope of this document. However, if you want to debug page faults as suggested above, a good place to start adding `printf()`s is `BX_CPU_C::dtranslate_linear()` in '`cpu/paging.cc`'.

## E.8 Tips

The page allocator in '`threads/palloc.c`' and the block allocator in '`threads/malloc.c`' clear all the bytes in memory to `0xcc` at time of free. Thus, if you see an attempt to dereference a pointer like `0xcccccccc`, or some other reference to `0xcc`, there's a good chance you're trying to reuse a page that's already been freed. Also, byte `0xcc` is the CPU opcode for "invoke interrupt 3," so if you see an error like `Interrupt 0x03 (#BP Breakpoint Exception)`, then Pintos tried to execute code in a freed page or block.

An assertion failure on the expression `sec_no < d->capacity` indicates that Pintos tried to access a file through an inode that has been closed and freed. Freeing an inode clears its starting sector number to `0xcccccccc`, which is not a valid sector number for disks smaller than about 1.6 TB.