Using GDB

You can run Pintos under the supervision of the GDB debugger.

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

`c`

> Continues execution until `ctrl+c` or the next breakpoint.

`break function`

`break file:line`

`break *address`

> 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`

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

`l *address`

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

`bt`

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

`p/a address`

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

`diassemble function`

> 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:

`dumplist &list type element`

> 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`

> 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`

> 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`

> Short-hand for `btthreadlist all_list allelem`.

`btpagefault`

> 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:

```
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 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 userprog{}, 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 breakpoint in `page_fault` in `exception.c`, which you will need to modify accordingly.

> 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:

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

> Followed by the output of the `btpagefault` command.

`loadusersymbols`

> 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:

```
loadusersymbol 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
```

> 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.