# GDB Assignments - CSCI 402, Summer 2024

merlot.usc.edu/cs402-m24/projects/gdb

Please note that these assignments will **not** be graded. But you are strongly encouraged to do them.

Overview

In previous semesters, I noticed that some students are debugging programs the old fashion way (i.e., using print statements and observing program printouts). It's very painful and inefficient. (And you should never let your employer knows that this is how you debug.) You have to learn how to use a debugger when developing for a large software project such as the weenix kernel!

System Prerequisites

You are required get your code to work under Ubuntu 16.04 running QEMU 2.5 (any subversion of Ubuntu 16.04 is fine). If you don't have such a system, please see the instructions on how to install Ubuntu 16.04 on a Windows or a Mac OS X machine.

I would prefer to see lso accept **Eclipse**. If you want **oval of the instructor** (and plea ble).

General Grading Co

**[ This section no lo raded. ]**

Grading is to be don **lesk hours**. You can go to **any** course

Since these are **not** uss how to accomplish the tasks etail! It's perfectly okay to share inform mmands to enter.

These assignments a re working on have certain level of are not done correctly, you will not ing guidelines below; and thus, you

Late Submission Pol

**[ This section no longer applies since GDB assignments are not graded. ]**

The regular late policy does **not** apply for these assignments since the only time these assignments can be graded is **during course producers' posted helpdesk hours**. Please note that the course producers **only** hold helpdesk hours during the 15 weeks of classes (and **not** during final exam weeks or study days).

After the 15 weeks of classes is over, if you would still like a GDB assignment graded, you can make an appointment with the instructor and get a 50% deduction in the corresponding GDB assignment grade. My availability is unpredictable after classes are over and you need to plan ahead. The absolute deadline for getting the GDB assignments graded is **(do not turn in)**.

GDB Assignment 1

You are required to debug `listtest` in Warmup Assignment #1 on Ubuntu 16.04. If you don't know what to do, please feel free to start a discussion in the class Google Group.

## Grading Guidelines

- **(1 pt)** You've got Ubuntu 16.04 running on your desktop/laptop.
- **(2 pt)** Set a breakpoint at the beginning of `main()`. Run the equivalent of "`listtest 1 2 3`" in the debugger. Print out `argc`, `argv[0]`, `argv[1]`, `argv[2]`, and `argv[3]` when you are at the breakpoint and explain to the course producer what you see. Quit the debugger.
- **(3 pt)** Set a **conditional breakpoint** in the middle of `CreateTestList()` to break when `i` is 10. Also set a breakpoint at the beginning of `RandomShuffle()`. Run the equivalent of "`listtest`" in the debugger. At the breakpoint, make sure `i` is indeed 10. Then change both `pList->anchor->next` and `pList->anchor->prev` to NULL and continue. Your program will either crash or reach the beginning of `RandomShuffle()`. In either case, do a stack trace and explain to the course producer what you see.
- **(4 pt)** Change the first line of `DoTest()` to:

  ```
  int num_items=3;
  ```

  and recompile. Then do the following:

  ```
  gdb listtest
  (gdb) break DoTest
  (gdb) run
  ```

  When you get t mes:

  ```
  (gdb) n
  ```

  Now you shoul `()`. Do:

  ```
  (gdb) p &l
  (gdb) p li
  ```

  Read the abov t you are looking at an empty list

  ```
  (gdb) n
  ```

  Now you shoul need to verify that the list is s

  ```
  (gdb) p li
  (gdb) p li
  (gdb) p *(
  (gdb) p *(
  (gdb) p *(list.anchor.next->next->next)
  ```

  Read the above printout very carefully. Get a piece of paper and **draw a picture** that looks like the picture in the warmup1 spec, and **label** every part of the list and list elements and convince yourself that you are looking at a doubly-linked circular list of 3 objects (where the objects are integers with values 0, 1, and 2). As you proceed, any time you call a function, repeat the above print statements to understand how the list is changing and make sure that you still have a valid doubly-linked circular list.

If your `my402list.c` has bugs and cannot go far enough to demonstrate the above to the course producer, you will not receive points for the corresponding items.

GDB Assignment 2

You will be debugging the **weenix kernel** in this assignment on Ubuntu 16.04. You need to write some working code first before you can finish with this assignment. If you don't know what to do, please feel free to start a discussion in the class Google Group.

## Grading Guidelines

- **(2 pt)** Start debugging the weenix kernel. Show that you would get a breakpoint at the beginning of `bootstrap()`. List the source code and single step in `bootstrap()`. When your code is about to create the idle process, step **into** the function and do a stack trace and explain to the course producer what you see.
- **(1 pt)** Set a breakpoint at the beginning of `idleproc_run()` and show that you get there. Try the following gdb command and make sure it works:

  ```
  kernel info pt_mapping_info curproc->p_pagedir
  ```

  You don't need to understand the printout. You just need to make sure that you don't get a bunch of Python error messages. If you get a bunch of Python error messages, it may be that you didn't initialize and setup the page table for the idle process correctly or your `curproc` is pointing at the wrong place.
- **(1 pt)** Set a breakpoint at the beginning of `initproc_run()` and show that you get there. Explain the output of the "`kernel proc`" gdb command to the course producer. List the source code and single step in `initproc_run()`.
- **(2 pt)** Set a breakpoint at the beginning of `sched_switch()` and show that you get there. Do a stack trace and explain to the course producer which process is giving up the CPU by calling `sched_switch()` and why it is going to sleep and which queue it
- **(3 pt)** Your `proc` a new process is created. Set return from `proc_create` wi ewly created process has a print the `proc_t` data structure a ed the `pageoutd` process. Do a s ere the `pageoutd` process is crea
- **(1 pt)** Set a bre `ber_test.c`" and show that you gdb command to the course proc

GDB Assignment 3

You will be debuggin **rogram** in this assignment on Ubun code first before you can finish with th se feel free to start a discussion in s in the Kernel 3 FAQ to figure out hov

## Grading Guidelines

- **(2 pt)** Set a breakpoint at the **startup** routine (i.e., `__libc_static_entry()`) of the **`hello` user space program** and show that you can get there. Set a breakpoint at `handle_pagefault()` and show that if you do a single-step in gdb at `__libc_static_entry()` of `hello`, you will reach `handle_pagefault()`.
- **(2 pt)** In `handle_pagefault()`, display the virtual memory map by issuing the "`kernel info vmmap_mapping_info curproc->p_vmmap`" gdb command and explain to the course producer what you see. You will not get full credit if your virtual memory map looks "wrong".
- **(2 pt)** Use the "`add-symbol-file user/usr/bin/hello.exec 0x08048094`" gdb command to add debugging information about the "`/usr/bin/hello`" user space program into gdb. Set a breakpoint in `main()` and continue. When you get to `main()` in user space, type "list" to see the code in "`hello.c`". In order to get to `main()` in user space, your need to handle the first two page faults correctly.

- **(2 pt)** The **`hello` user space program** simply makes 4 system calls. It calls `open()` twice, `write()` once, and `exit()` once. Set a breakpoint at `syscall_dispatch()` and show how the kernel reaches `sys_open()`, `sys_write()`, and `do_exit()`. In order to get to all these breakpoints, your need to handle the all the page faults correctly.
- **(2 pt)** The **`hello` user space program** simply makes 4 system calls. It calls `open()` twice, `write()` once, and `exit()` once. Show the course producer how you would do the following:
    1. Step into the `write()` C library function. Single step a few statements to get to the `trap()` function (defined in "`user/include/weenix/trap.h`").
    2. When you get inside the `trap()` function, list the C code for `trap()`. You should see that it invokes the software interrupt machine intructions twice. Switch gdb layout to "`asm`" mode. Location the two trap machine instructions ("`int 0x2e`"). Step breakpoints on the machine instructions immediately **after** these two trap machine instructions.
    3. Use the gdb "`cont`" command to execute till it returns from the first trap machine instruction. Use the gdb "`info registers`" command to display the values of the CPU registers. Explain the values of `eax`, `esp`, `ebp`, and `eip` to the course producer.

---