LINKÖPINGS UNIVERSITET

TDDE47/TDDE68: Concurrent programming and Operating Systems

Lab A: C, GDB, and Pintos

2026-01-15

Dag Jönsson
IDA/SaS

1 Introduction

1.1 Goal

This lab has the following goals:

- Setup the project repository
- Practice using GDB to debug
- Practice programming in C
- Learn some tools to debug Pintos

As part of item 3 above you will be implementing a simple linked list in C, both to help you get started writing C code, but also practice on using pointers. Debugging with GDB is to help you debug problems further in the lab series.

Time estimate is roughly 4 hours if you already have an idea of how pointers work, 6 hours if not. The assignment should be seen as an introduction to C, GDB, and Pintos, other concepts shouldn't be new to you.

And we want to address the length of this instruction, the majority of the text is just to read and following along with. We do not expect you to become experts at using GDB, or writing C code, but there is quite a lot to discuss and explain. About 3-5 pages are just pictures.

1.2 Prerequisites

To be able to work on these assignments you will need working knowledge on how to use a Linux shell (bash), and working with git. If you need to refresh your memory, check back on the material in your previous courses. A useful tutorial for git can be found here: https://learngitbranching.js.org

It is highly recommended to configure your *gitlab.liu.se* account to use SSH keys, see the documentation: https://docs.gitlab.com/user/ssh/

2 Project repository setup

In this section the goal is to create your own copy of the Pintos repository that you will work on during this lab series.

2.1 Forking Pintos

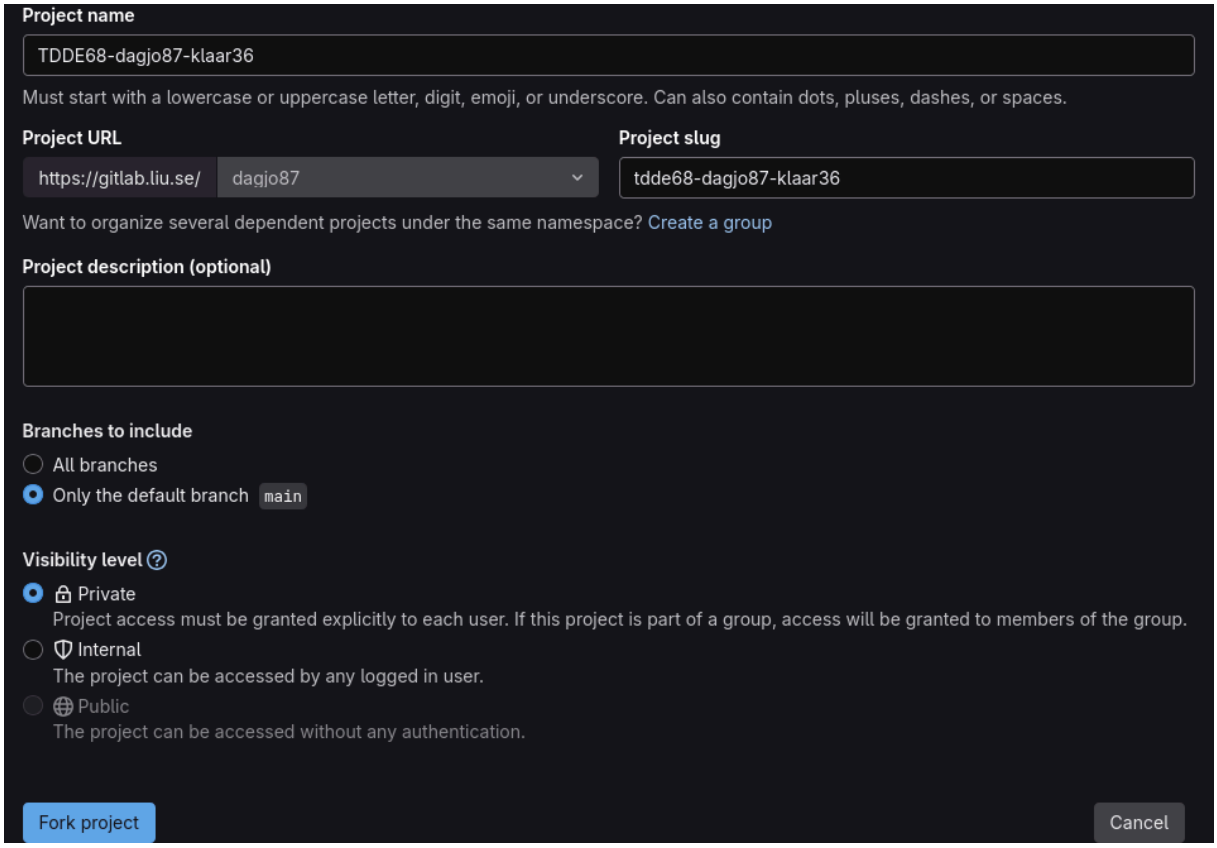
You do not have to demonstrate anything for this section.

Start by navigating to the original repository: https://gitlab.liu.se/cpos/pintos Once there, press the "Fork" button in the top right. Here you can fill out the information as you please. Suggestion is to name the project according to the course code. Before you create the fork, you should check that the following is set:

- Branches: You only need to fork 'main'
- Visibility should be set to **private**

See Figure 1 for an example.

Figure 1 — Example fork page



Once you have entered the requested information, press “Fork project”.

Once the repository is forked, invite your lab partner as **owner**, and your lab assistant as **reporter**. See the list below for a list of LiU-ids.

List of assistants for 2026:

- Amin Bajand: mohba27
- Animesh Thakur: anith60
- Arvid Rämsberg: arvra591
- Dag Jönsson: dagjo87
- Masoud Sadrnezhaad: jjssa24
- Máté Földiák: matfo50

Once you have your repository set up in Gitlab, clone it to your computer. We will be referring to the clone of the repo as `pintos/` in future instructions.

2.2 Setting up the PATH

To be able to run Pintos, you need to modify your PATH variable. The easiest way to do this is the following steps: **Note:** Only needs to be done once per student.

1. Navigate to the `pintos/utils` directory in a terminal
2. Run the following command in the terminal:

```
echo "export PATH=$(pwd):\${PATH}" >> ~/.bashrc; source ~/.bashrc
```

This will add a line to your `~/.bashrc`, updating your PATH variable to include the `pintos/utils` directory. The `source` is just to reload the file you just modified so the changes are immediate.

If you are using a different shell than what is default on the school machines, we expect you to figure out the differences.

3 Assignment A: Reading C and GDB debugging

The goal of this section is to help you familiarize yourself with C, with a focus on debugging using GDB. To keep it simple, this will be done outside of Pintos, which means you will be running GDB normally. You will use GDB in Pintos later in this lab.

In the directory `pintos/standalone/gdb` you will find three C-programs: `debug1.c`, `debug2.c`, and `debug3.c`. These programs contains errors that will cause the programs to crash when run. Your assignment is to find the problems using GDB, and then fix the errors.

Note: In these programs we have supplied our own `malloc` and `free` functions to make the programs crash faster when there are memory errors to make the debugging a bit simpler. This does mean that similar problems made later in Pintos may not cause a crash immediately, but rather later when you are working on new unrelated code.

3.1 Reading material

- GDB Documentation - Use as reference
- Sample session page - Sample session that may be helpful to get started

Some useful GDB commands to know:

- `print var` - prints information about the variable `var`.
- `print *var` - prints information about the dereferenced pointer `var`.
- `run` - run the program until either a problem occurs, a breakpoint occurs, or the program finishes.
- `continue` - Continue running after execution stopped, for example after hitting a breakpoint.
- `break main` - set a breakpoint at the start of the `main` function
- `backtrace` - print information about the call stack. Useful when an error has occurred and you want to know how the program ended up where it is.
- `step` - Step to the next line of code. If the next line is a function call, will step *into* that function.
- `next` - Step to the next line in the current function. Will skip *over* any function calls.
- `finish` - Step to the end of the current function.
- `list` - print the source code around where the marker is.

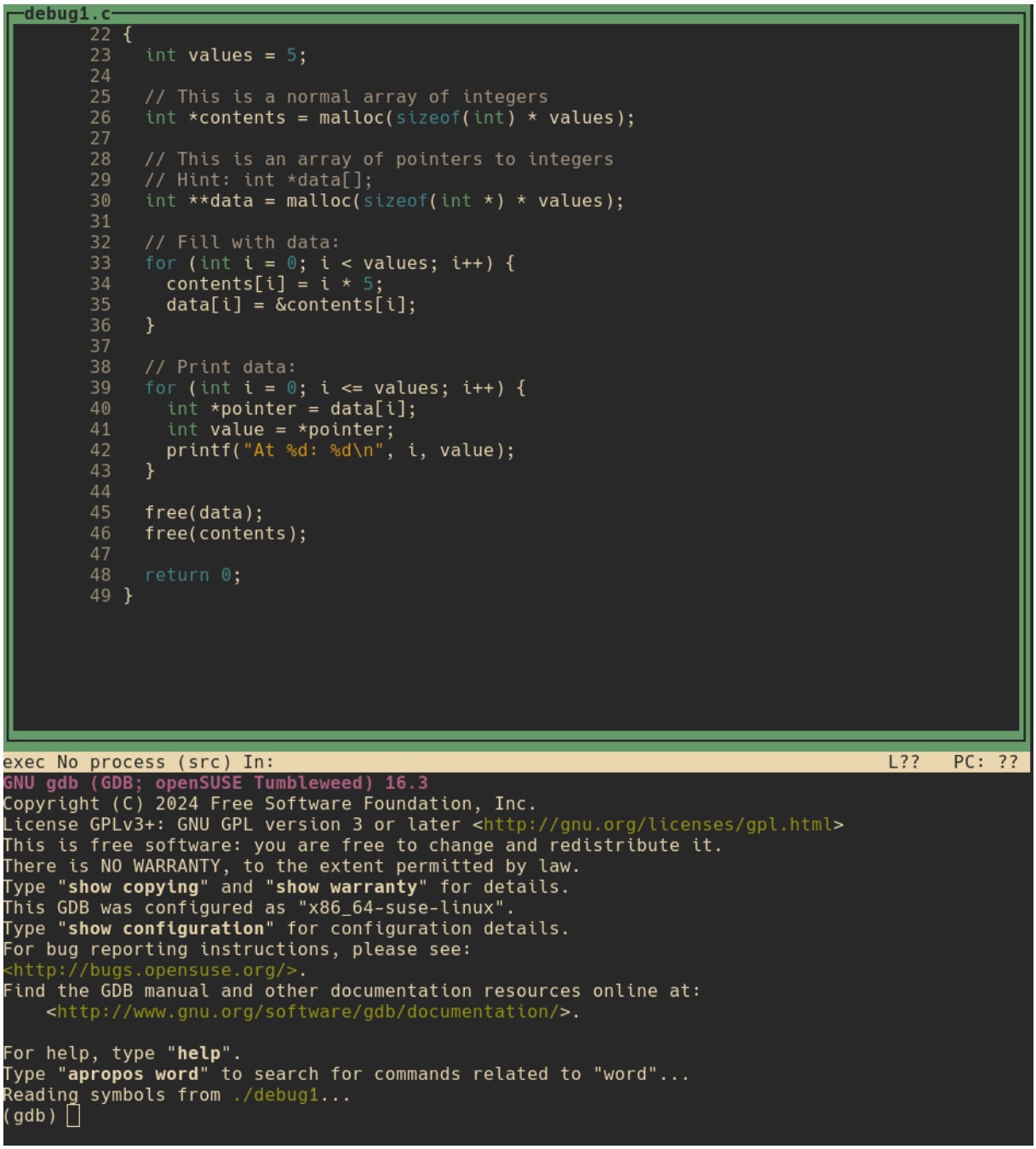
For the instructions below, you should follow along in your GDB sessions.

3.2 Program 1: `debug1.c`

Compile the program with the command `make debug1`. You can then run the program like `./debug1`. You will notice that it crashes with a `Segmentation fault (core dumped)` message. This means the program tried to access memory that for some reason was not valid.

To get started debugging with GDB, you can run the following: `gdb -tui ./debug1`. The `-tui` flags means it will display the code in what GDB calls the *Text User Interface*. If you don't supply the flag and want to enable the TUI after the fact, you can write `tui enable` within GDB, or by pressing `CTRL+X`, followed by `CTRL+A`. When you have started GDB you will see something similar to Figure 2.

Figure 2 — Example GDB startup



If your terminal is small you may see a message that says the following: `Type <RET> for more, q to quit, c to continue without paging`. This happens when GDB tries to print more text than fits in the bottom screen. Simple press `c` and press enter to get to the prompt `(gdb)` as in the picture. If you want to avoid this in the future, make your terminal windows larger.

As you can see in the picture the interface consists of two parts. The upper part shows the source code for the program. GDB will point out what parts are currently running, what breakpoints are set and a couple of other things. Since the program hasn't started executing yet it doesn't show too much of interest right now.

The bottom part consists of a "terminal" where you can enter commands to GDB, and where GDB will answer with information about the program. The prompt `(gdb)` means that GDB is ready and is waiting for commands. If you turn off the TUI this terminal will be the only thing shown.

It is worth pointing out a couple of things about the TUI. In the picture above the upper part is marked with a colored frame (yours should be blue, the picture shows a green). This means it is in focus. If you press the arrow buttons you will scroll the code view. If you press `CTRL+X` followed by a press of `O`, the frame will disappear. This means the "terminal" in the bottom is in focus, and the arrows buttons will traverse the command history (like in the normal terminal). If you are familiar with Emacs you might notice that `CTRL+X, O` is the same command to switch *window* in Emacs. Several simple commands in Emacs works the same in GDB.

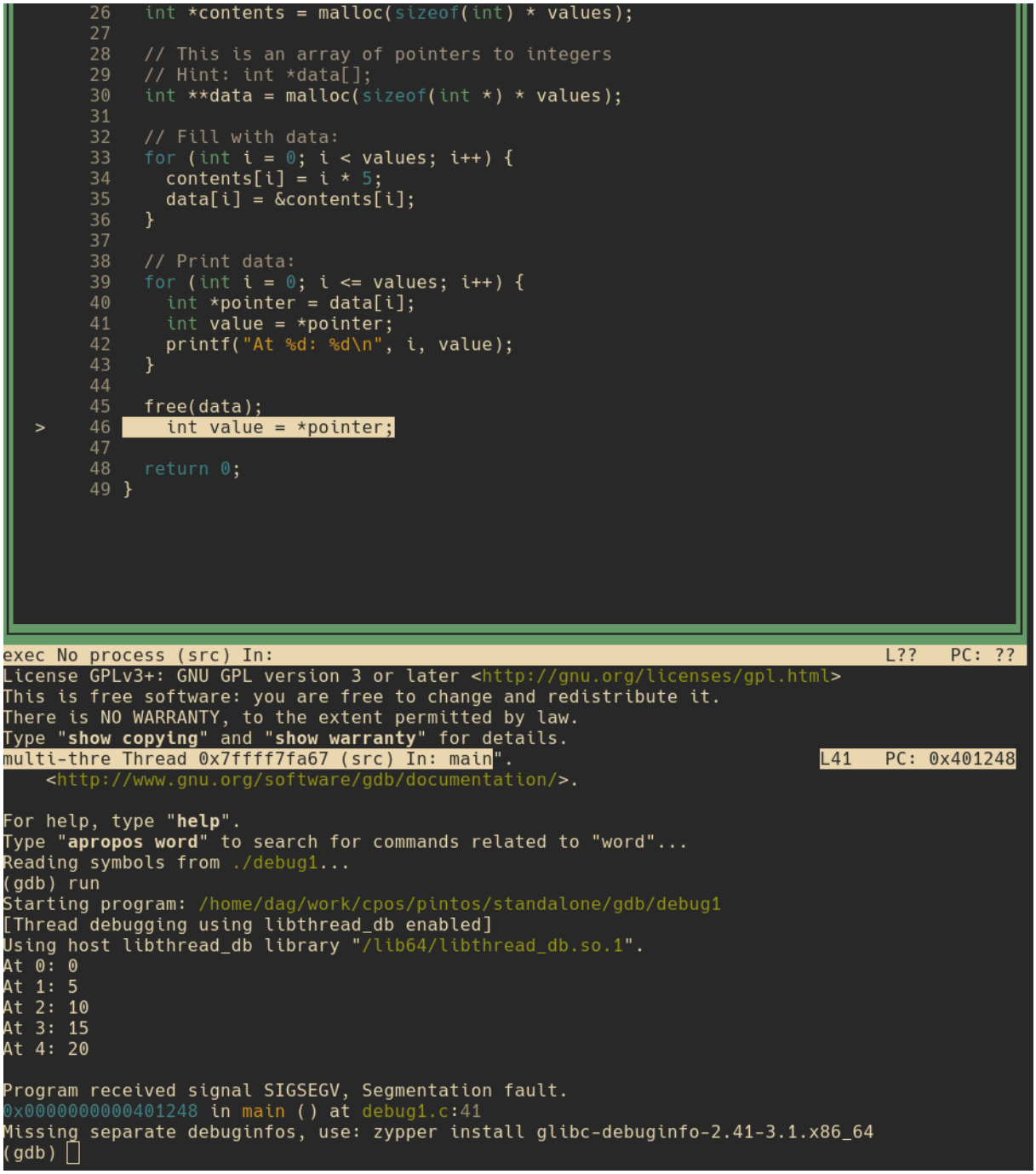
Sometimes the TUI "breaks". This can happen if the program you are debugging prints to standard out, and "prints over" things that GDB handles. This can easily be fixed by pressing `CTRL+L`, this will force GDB to redraw everything properly. In short, if GDB starts to look weird, press `CTRL+L`, it can't hurt!

Now you should be ready to start debugging. Execute the program by typing the command `run` and press enter. It's worth noting that almost all commands in GDB can be shortened, as long as GDB can figure out what you mean it will usually do the right thing. As an example, you can shorten `run` to just `r` instead. In these instructions we will give the long commands, followed by the shorthand in parenthesis when relevant.

When you have typed `run` (`r`) and pressed enter GDB will let the program execute. In this case the program will print a few lines and then crash, like before. On my system it looks like Figure 3.

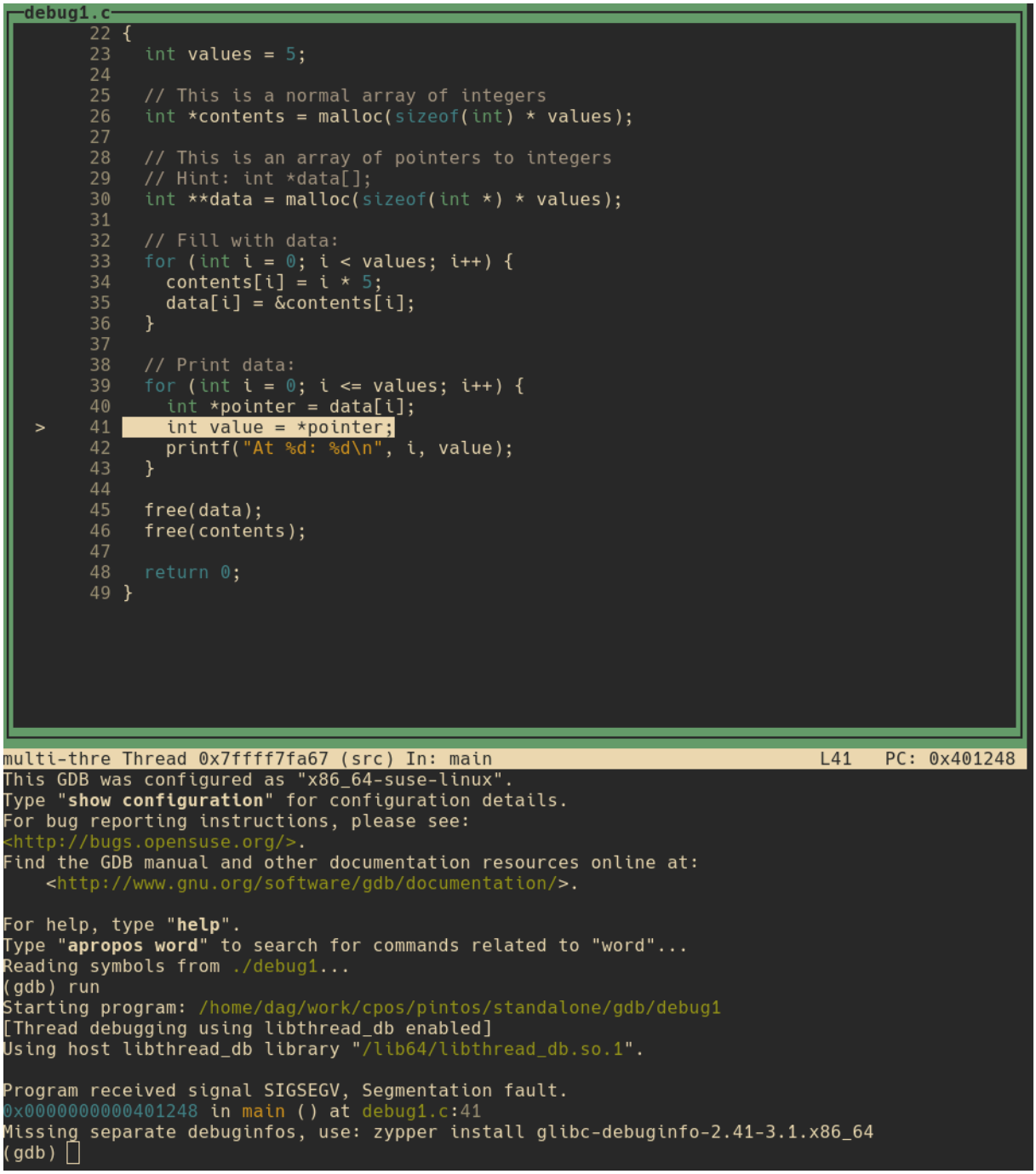
(Exactly what is printed and how differs between systems and configurations, if I'm not explaining or pointing out specific lines it means they are not important)

Figure 3 — GDB TUI “breaking”



It’s not that bad in this picture, but the TUI has bugged out a bit, (because of the printout from the program) so I press `CTRL+L` to redraw it and get result captured in Figure 4.

Figure 4 — Example of segfault



In the upper part we can see that GDB has marked the line that was executing when the program crashed. In the bottom part we can see the message `received signal SIGSEGV, Segmentation fault`. This is the same information we got in the terminal previously. The next line tells us that the program crashed on address `0x0000000000401248`. This is not that interesting (we don't know what is there, it will also differ between different runs). GDB does however tell you it matches the line 41 in the file `debug1.c`, and that it is in the function `main`. Same information is also shown in the upper window.

To start figuring out what went wrong in our program, we can inspect it with the help of the command `print` (`p`). The command will evaluate a C-expression and print the result. Almost everything that works in C also works in GDB (but not everything). Here we can inspect the variable `i` by typing `print i` (`p i`). We then get the following:

```
$1 = 5
```

This means that `i` was 5. The `$1 =` part means we can use `$1` in a later statement if we want to reuse the result (i.e. `gdb print $i + 2` gives 7).

Knowing that `i` was 5 we can now focus on the index 5 in the array. We can again use `print` to inspect the program. Either we can do `print data[i]` , or we realize that the value we are interested in already exists in the variable `pointer` from the line before and print the `pointer` instead. We should then get the following back:

```
$2 = (int *) 0xcccccccccccccccc
```

This means that index 5 in the array (and the pointer) has the value `0xcccccccccccccccc` , and the type is `int *` (pointer to `int`). Since the program crashed on the line trying to dereference the pointer we should then suspect that this pointer is not valid. To confirm this we can try to dereference the pointer with the following `print *pointer` (in other words, the statement that we think crashed). We then get the following:

```
Cannot access memory at address 0xcccccccccccccccc
```

This confirms our suspicion that the pointer is invalid. From here we can draw the conclusion that index 5 in `data` is incorrect. Next question is why? We can continue inspecting the program using GDB, by using a *breakpoint*. We do this by typing in `break` (`b`), and supplying where the put the breakpoint. Either we can supply a function name (e.g. `break main`) or a filename and a line number (e.g. `break debug1.c:41`). In our case we want to set a breakpoint on line 35, so we type `break debug1.c:35` (or just `break 35` since this is the file we can see in the upper window). We should see a mark (`b+`) to the left of the line, and get the following output:

```
Breakpoint 1 at 0x4011e9: file debug1.c, line 35.
```

Our breakpoint is in other words number 1. If we later want to disable this breakpoint we can type `disable 1` (`dis 1`). We can now rerun our program with the command `run` (`r`) as before. GDB will ask you to confirm it's okay to terminate the currently running instance, answer `yes` (`y`). GDB will restart our program and print the following:

```
Breakpoint 1, main () at debug1.c:35
```

This means the program has stopped at the breakpoint number 1 that we just created. We can also note that the mark in the upper window now is shown as `B+` to indicate that we have hit that breakpoint at least once. We can again inspect our variables `print i` (`p i`) to check the value of our counter. First time it's obviously 0. From here

we can ask GDB to continue running the program with `continue` (`c`). GDB will then answer with:

```
Continuing.
Breakpoint 1, main () at debug1.c:35
```

This means we have again hit our breakpoint 1. We can verify this by printing the `i` variable again, this time showing 1. Since we will want to this several times we can ask GDB to print it after each command, by typing `display i` (`disp i`). GDB will answer with:

```
1: i = 1
```

This means our statement got assigned the ID 1, and that `i` had the value 1. If we `continue` again GDB will print something similar each time:

```
Continuing.

Breakpoint 1, main () at debug1.c:41
1: i = 2
```

If we in the future don't want to see the statement anymore we can type `undisplay 1` (`und 1`) (where 1 is the ID from before).

If we continue to step through the loop with `continue` we will see that index 5 is never assigned a value, and this is why we crash. We can now quit GDB with the command `quit` (`q`) and solve the program.

Assignment: Implement a solution the problem found above.

There are other ways to step through the code that doesn't involve setting breakpoints and letting the program execute to them. Try playing around with the `step` (`s`), `next` (`n`), and `finish` (`fin`) commands.

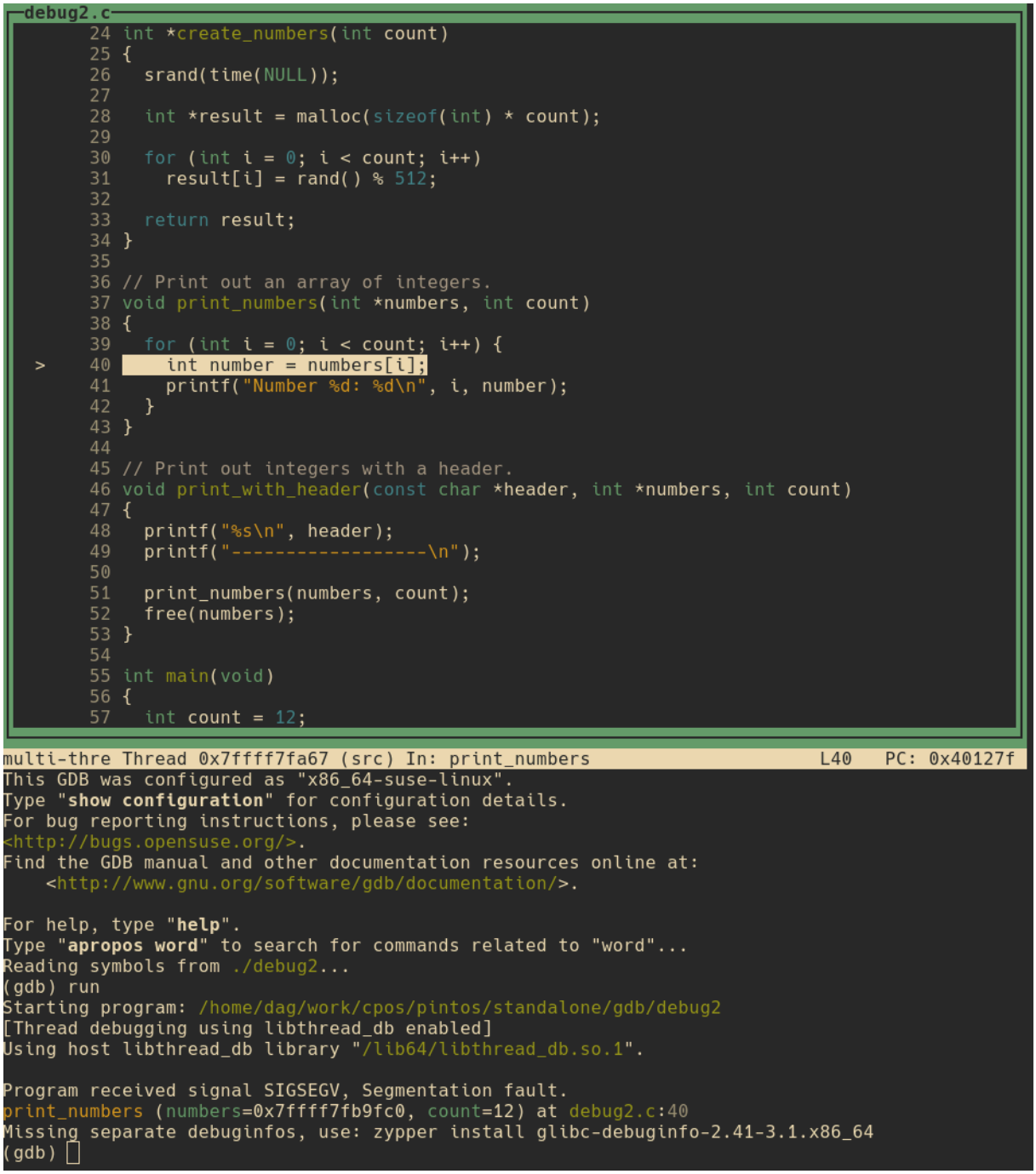
Lastly it's worth pointing out that you usually don't have to restart GDB if you have made changes to the code. Simply compile the program in a separate terminal, and then tell GDB to rerun (`run` / `r`) the program and it will notice that the program changed, and will try to keep any breakpoints. Sometimes it doesn't work though, and will sadly crash because of this, so keep expectations reasonable.

3.3 Program 2: `debug2.c`

Like before, you can compile the program with `make debug2` , and run it with `./debug2` . You should see that the program prints a list of numbers, and then crash with the message `Segmentation fault` again.

Like before we can debug the program with GDB. Run GDB on the file as before, and run the program until you get it to crash. You should see something similar as Figure 5.

Figure 5 — Result of running debug2



Here we can see that GDB prints out the parameters to `print_numbers`, which can be useful. Like before we can print the value of `i` with `print i`:

```
$1 = 0
```

We can also check the values inside of `numbers`. If we do that we get something similar (the address is probably different):

```
Cannot access memory at address 0x7ffff7fc9fc0
```

This indicates that the pointer is pointing to invalid memory. The question is where did this pointer originate from? With the command `backtrace` (`bt`) we can have GDB print out a *stack trace*:

```
#0  0x0000555555555332 in print_numbers (numbers=0x7ffff7fc9fc0, count=12)
    at debug2.c:40
#1  0x00005555555553a0 in print_with_header (header=0x555555556032
"Second time:", numbers=0x7ffff7fc9fc0, count=12) at debug2.c:51
#2  0x0000555555555406 in main () at debug2.c:62
```

Here we see a numbered list of *stack frames*. Right now we are in frame number 0, since we are in the `print_numbers` function. It was called by the code in frame number 1, within `print_with_header`. This was in turn called by `main`. We can jump to earlier frames with the command `frame` (`f`). If we jump to frame 1 (`frame 1` or `f 1`) GDB will remind us where we are:

```
#1  0x00005555555553a0 in print_with_header (header=0x555555556032
"Second time:", numbers=0x7ffff7fc9fc0, count=12) at debug2.c:51
```

Here we can see that `numbers` is a parameter to `print_with_header`, and the only thing done to `numbers` is that is passed along as an argument to `print_numbers`; we can also verify that it contains the same pointer value that we saw earlier, so the problem is most likely not here. If we go back to frame 2 (`frame 2`):

```
#2  0x0000555555555406 in main () at debug2.c:62
```

Here we can see that it is the same `numbers` that was sent to both calls to `print_with_header`. We know from before that the first call worked, so what happened between those calls? If we place a breakpoint on line 59 (the line with the first call to `print_with_header`), with `break 59`, and run the program again with `run`.

Now we can see what happens with the variable `numbers` when the code is run. Since we will want to check it all the time we can tell GDB to `display numbers` to continually print the address. To also show what it contains we can also run `display numbers[0]`. Right now GDB should print out something similar to:

```
1: numbers = (int *) 0x7ffff7fc9fc0
2: numbers[0] = 299
```

We can step forward in the program with `next` (`n`). When we get to the line `printf("\n");` we get the following from GDB (remember to press `CTRL+L` if the TUI breaks):

```
1: numbers = (int *) 0x7ffff7fc9fc0
2: numbers[0] = <error: Cannot access memory at address 0x7ffff7fc9fc0>
```

This is interesting. The address is the same, but we can no longer read the first element. Something happened to it within `print_with_header`! We restart the program with `run` to get back to our breakpoint and step into the call instead with `step` (`s`). GDB will tell us that we are inside a different function. Since we are inside a different function we will need to retell it to display our variables. We should see the same thing as before (but probably a different random number).

```
1: numbers = (int *) 0x7ffff7fc9fc0
2: numbers[0] = 299
```

We can now step through `print_with_header` with `next` (`n`). To avoid having to type `next` over and over, we can enter it once, and then press just enter to rerun the last command.

When stepping through the function we notice that the printout of `numbers[0]` is correct until we hit the `free` call. Here is our problem!

Assignment: Now it's up to you to solve the problem in a reasonable way. Consider what mistake the programmer made in this case made.

3.4 Program 3: `debug3.c`

Compile the program and run it and you should see the message `Strings in C are fun!` and then a crash with `Segmentation fault` again. Like before, run the program in GDB with `gdb -tui ./debug3` and run it with `run` . This time however we don't get a line in our source code, and instead we just get:

```
Program received signal SIGSEGV, Segmentation fault.
__strlen_avx2 () at ../sysdeps/x86_64/multiarch/strlen-avx2.S:141
../sysdeps/x86_64/multiarch/strlen-avx2.S: No such file or directory.
```

It seems that the function `__strlen_avx2` (or similar, depends on the system) crashed when it tried to execute. This happens to be a implementation of the `strlen` function in the standard library, it's used to calculate the length of a C-string. GDB can't find any source code for the function, which is missing in the upper window. All this obviously means we have found a bug in the standard library and we should go about patching it and rebuild it on our machine. Or more likely it's the way we have used the function that is wrong. So let's check our own code before we start patching the standard library. We can again use `backtrace` (`bt`) to check how we got here:

```
#0  __strlen_avx2 () at ../sysdeps/x86_64/multiarch/strlen-avx2.S:141
#1  0x00007ffff7df1d15 in __vfprintf_internal (s=0x7ffff7f666a0
    <_IO_2_1_stdout_>, format=0x555555556028 "Copy:     %s\n",
    ap=ap@entry=0x7fffffffd5c0, mode_flags=mode_flags@entry=0) at
    vfprintf-internal.c:1688
#2  0x00007ffff7ddad3f in __printf (format=<optimized out>) at printf.c:33
#3  0x00005555555552ca in main () at debug3.c:38
```

We again get our *stack frames*. Remember that 0 is our current frame where we crashed. In this case the stack trace isn't that long, but a fairly reasonable strategy is to start from the bottom (#3 in this case) and go up until you find a function call that you don't recognize as yours. We will immediately see that from `main` in `debug3.c:38` we made a call to `__printf` , so already we have gone outside of code we control.

An observation that can be relevant later in Pintos is that the parameter to frame number 2 (`__printf` just says `<optimized out>` . This happens occasionally when optimizations are involved. The compiler made the decision that a value of a certain variable wasn't needed in the code, and because of this GDB can't show it. This can also happen with the `print` and `display` commands.

If we move back to frame 4 (`frame 4`) we can see that we crashed on the line trying to print the value of `copy` , which would indicate that there is something wrong with that variable. Inspecting it with `print` we get the following:

```
$2 = 0x7ffff7fc9fe0 "Strings in C are fun!
\314\314\314\314\314\314\314\314\314\314", <incomplete sequence \314>
```

GDB has tried to be kind and viewed the variable as a C-string. We can also see that the beginning of the string looks reasonable, but we get a bunch of garbage values at the end. This means that `copy` is not a valid C-string, since we are missing an ending NULL character (just the value 0). This means that `strlen` (called in `printf`) will not be able to calculate the length of the ‘’string’‘, since it doesn’t know when to stop, and crashed as it started to read memory it shouldn’t. Analyzing the code in `main` further shows that `copy` is a result of the call to `my_strdup` , so our investigation should continue there.

Assignment: Solve the problem in a reasonable way. It might be worth thinking about what the result from `strlen` actually is.

4 Assignment B: Writing C and wrangling pointers

This section aims to give you some hands on experience writing C code, working with pointers and managing memory, something that will be important once you start working on Pintos. The assignment is to implement a singly linked list in C. You are given some skeleton code in `pintos/standalone/slist` to get you started.

4.1 Reading material

To help you get started with the C programming language, here are some useful sources:

- Slides from C lecture
- Introduction to Concurrent Programming in C Chapter 2
- C reference (C99)

4.2 Requirements

The following functions need to be defined (already declared in `list.h`):

```
bool is_empty(struct list_item const *root);
int get(struct list_item const *root, int idx);
void append(struct list_item *root, int x);
void prepend(struct list_item *root, int x);
void input_sorted(struct list_item *root, int x);
void print(struct list_item *root);
void clear(struct list_item *root);
```

To build your solution, run `make` in the `pintos/standalone/slist` directory.

Note: The given code in `main.c` includes a *sentinel node* (the root Node), this node should not be considered to be a part of the list, and is just there hold on to the list. This makes it possible to avoid certain edge cases and should help you keep the solution simple. In other words, a call to `print` with just the given root node should result in nothing being printed.

You need to keep the following in mind as you implement your solution:

- The nodes in the list are dynamically allocated and freed once they are not needed.
- You need to write a test program that tests all the functions (`main.c`)
- If we clear a list we should be able to use it again
- No memory leaks

Use: `valgrind --tool=memcheck --leak-check=yes ./main` to check for memory leaks. Feel free to use GDB to debug problems you might run into.

Important! Treat WARNINGS in Valgrind as ERRORS. You should for example not have invalid writes or reads.

4.3 Optional challenge

As mentioned above, the given code uses a *sentinel node* to avoid certain edge cases.

If you want to practice your pointer knowledge further, implement the list without this solution. You will want to look into using double pointers.

5 Assignment C: Debugging Pintos

This section aim to give you the tools to start working in and debugging Pintos. It will also give you a short introduction to the project structure.

5.1 Reading material

Below you have some reading material from Stanford (upstream Pintos). Please do remember that you are taking a course at LiU, and any *instruction* given in the below documentation should be ignored, our instructions take precedence. These links should be treated only as documentation, i.e. ignore any statements like "You will implement this is project 1".

- Stanford Pintos: Appendix E: Debugging Tools Backtraces and GDB
- Stanford Pintos: Source Tree Overview

5.2 Building and running Pintos

As you probably have noticed, the Pintos project consists of several different directories. There are a couple you can ignore for now, since we will focus on the `userprog` and `threads` directories in this lab.

To compile Pintos, you should navigate to the `pintos/userprog` directory, and run the command `make -j8`. The `-j8` flag is just there to speed up compilation by telling `make` to use several threads.

We will also need some applications to run, so from `pintos/userprog`, run the command `make -j8 -C ../examples` to build the example programs available. The resulting files will be put in `pintos/examples`. You are free and even encouraged to read through the example files as you feel necessary.

Now that we have both Pintos and some example programs compiled and ready, let's run Pintos for the first time. If you inspect the file `pintos/examples/noop.c` you will find some documentation at the top of the file on how to run it. Follow those instructions and you should be able to have Pintos running. It's not going to do much, but we can use this to verify our setup. After about 5 seconds Pintos should terminate if you followed the instructions, otherwise, just press `CTRL+C` to terminate the process. Your result should be similar to Figure 6. Anything before the line `Boot complete.` you can ignore, it's just noise.

Figure 6 — Result of running noop

```
SeaBIOS (version rel-1.16.3-2-gc13ff2cd-prebuilt.qemu.org)
Booting from Hard Disk...
PPiiLLoo  hhddaa1
1
LLooaaddiinngg.....................
Kernel command line: -f -q extract run 'binary -s 17'
Pintos booting with 3,968 kB RAM...
367 pages available in kernel pool.
367 pages available in user pool.
# main#1: thread_create("idle", ...) RETURNS 2
Calibrating timer...  728,064,000 loops/s.
ide0: unexpected interrupt
hda: 5,040 sectors (2 MB), model "QM00001", serial "QEMU HARDDISK"
hda1: 175 sectors (87 kB), Pintos OS kernel (20)
hda2: 4,096 sectors (2 MB), Pintos file system (21)
hda3: 89 sectors (44 kB), Pintos scratch (22)
ide1: unexpected interrupt
filesys: using hda2
scratch: using hda3
Formatting file system...done.
Boot complete.
Extracting ustar archive from scratch device into file system...
Putting 'binary' into the file system...
Erasing ustar archive...
Executing 'binary -s 17':
# main#1: process_execute("binary -s 17") ENTERED
# main#1: thread_create("binary -s 17", ...) RETURNS 3
# main#1: process_execute("binary -s 17") RETURNS 3
# main#1: process_wait(3) ENTERED
# binary#3: start_process("binary -s 17") ENTERED
# binary#3: start_process(...): load returned 1
# binary#3: start_process("binary -s 17") DONE
system call!
# binary#3: process_exit() ENTERED
binary: exit(-1073515896)
# binary#3: process_exit() DONE with status -1073515896

TIMEOUT after 5 seconds of host CPU time
```

5.3 Debugging tools in Pintos

As you can read in Stanford Pintos, Appendix E, there are several options available to you when it comes to debugging in Pintos. Here we will walk you through the ones we think will be the most important for you.

The `debug(...)` macro

As you can see in Figure 6, there are lines like `# main#1: process_execute("binary -s 17") ENTERED`, these are the result of calls to the predefined `debug(...)` macro in Pintos. It works just like `printf(...)`, in fact it

really only is a `printf` call prepending the #-sign to whatever is going to be printed. If you want to add any of your own debug-printouts, use the `debug` macro as defined in `pintos/lib/debug.h` (just `#include <debug.h>`). The reason to use it is that later in the lab series you will be running a test suite that tries to match the output from Pintos against what it expects to find, and if anything extra is found it will fail the test.

The `backtrace` tool

Something that you will probably run into while developing Pintos is the *backtrace*, also knows as the *stack trace*. To experiment with this, run `noop` again, but this time change the last 'binary' to for example 'binar' to make it not find the program. This will cause Pintos to print out a *backtrace*. The result might look something like Figure 7.

Figure 7 — Result of trying to run 'binar'

```
SeaBIOS (version rel-1.16.3-2-gc13ff2cd-prebuilt.qemu.org)
Booting from Hard Disk...
PPiiLLoo  hhddaa1
1
LLooaaddiinngg.....................
Kernel command line: -f -q extract run 'binar -s 17'
Pintos booting with 3,968 kB RAM...
367 pages available in kernel pool.
367 pages available in user pool.
# main#1: thread_create("idle", ...) RETURNS 2
Calibrating timer...  723,968,000 loops/s.
ide0: unexpected interrupt
hda: 5,040 sectors (2 MB), model "QM00001", serial "QEMU HARDDISK"
hda1: 175 sectors (87 kB), Pintos OS kernel (20)
hda2: 4,096 sectors (2 MB), Pintos file system (21)
hda3: 89 sectors (44 kB), Pintos scratch (22)
ide1: unexpected interrupt
filesys: using hda2
scratch: using hda3
Formatting file system...done.
Boot complete.
Extracting ustar archive from scratch device into file system...
Putting 'binary' into the file system...
Erasing ustar archive...
Executing 'binar -s 17':
# main#1: process_execute("binar -s 17") ENTERED
# main#1: thread_create("binar -s 17", ...) RETURNS 3
# main#1: process_execute("binar -s 17") RETURNS 3
# main#1: process_wait(3) ENTERED
# binar#3: start_process("binar -s 17") ENTERED
load: binar: open failed
# binar#3: start_process(...): load returned 0
# binar#3: start_process("binar -s 17") DONE
Page fault at 0: not present error reading page in kernel context.
Interrupt 0x0e (#PF Page-Fault Exception) at eip=0xc002a5f3
 cr2=00000000 error=00000000
 eax=00000000 ebx=00000003 ecx=00000007 edx=c0100008
 esi=c0108000 edi=c0109000 esp=ffffff00 ebp=c0109f30
 cs=0008 ds=0010 es=0010 ss=0010
Kernel PANIC at ../../userprog/exception.c:96 in kill(): Kernel bug - unexpected interrupt in kernel
backtrace 0xc00284ea 0xc002b0dc 0xc002b17a 0xc0021a5b 0xc0021c27 0xc002a5f3 0xc00211ac 0xc002a95d 0xc0021207.
The `backtrace' program can make call stacks useful.
Read "Backtraces" in the "Debugging Tools" chapter
of the Pintos documentation for more information.
Simply copy-paste the backtrace command line above.
Timer: 442 ticks
Thread: 5 idle ticks, 426 kernel ticks, 11 user ticks
hda2 (filesys): 43 reads, 182 writes
hda3 (scratch): 88 reads, 2 writes
Console: 1967 characters output
Keyboard: 0 keys pressed
Exception: 1 page faults
Powering off...
```

Below the `Kernel PANIC at ...` line, there is a line like `backtrace 0xc002...` that is of interest to you. If you copy that entire line, from `backtrace` up to and including

the `.` and paste it into a terminal, you will get something that should be familiar to you; it looks like the result of running `backtrace` in GDB, because it pretty much is the same thing! Reading the resulting printout should help you start understand why, or rather from what function Pintos crashed.

There is one more thing that can be worth considering in the PANIC printout, what fault was triggered. In this case we panicked because of a `Page-Fault Exception` with the message: `Page fault at 0: not present error reading page in kernel context.`

Assignment: From reading the *backtrace*, figure out what function caused Pintos to panic.

GDB and Pintos

We can of course run Pintos through GDB. To do this you will need 2 terminals open, both standing in the `pintos/userprog` directory. In one of the terminals, run `pintos --gdb --filesys-size=2 ... run 'binar -s 17'`, so replace the `--T 5` flag with `--gdb`, this will cause Pintos to boot up in debug mode, waiting for a GDB session to connect and take over execution control.

In the other terminal, run the command `pintos-gdb build/kernel.o`, this will start GDB, load a couple of macros, and make it ready to debug a Pintos instance. First you need to connect to a Pintos instance though, and this is easiest done by typing `debugpintos` and pressing enter.

Note: There might be issues with this if you are working through ThinLinc, read the Appendix E: GDB: about setting the GDB_PORT variable. If you are working on an physical computer it shouldn't happen, unless you left some Pintos instance running. See the next note.

Note: Sometimes, by sheer luck (or lack thereof), the Pintos process can completely hang after working with it through GDB, and you will not be able to terminate it with `CTRL+C`. To solve this you can run the command `pkill qemu` in a terminal to kill any Pintos process on the system.

Note: The GDB TUI is still available, you will just have to enable it after starting GDB. It should work well enough, but usability may vary.

At this stage you should be sitting in a fairly normal GDB session, with a printout similar to `0x0000fff0 in ?? ()`. You can set breakpoints as usual, and you can let Pintos run with the command `continue` (`c`). If you do this now, Pintos will crash again like before and detach from the GDB session, without you able to debug anything and you will have to redo the previous steps. Let's start with adding a breakpoint at the function you figured out in the backtrace assignment. Reminder: `break function_name` to set a breakpoint. As long as the function name is unique GDB should find it.

After you've set the breakpoint, you can let Pintos `continue` (`c`), and use `next` and/or `step` to step through the code. By doing this you should be able to figure out what specific line is causing the page fault. Remainder, if you aren't using the TUI you can

use `list` in GDB to get a printout of the code in the vicinity of the GDB marker. Or open the file in question in your favorite editor.

Assignment: Once you have a theory about what specific operation is causing the page fault, by inspecting variables and looking at the exception message, figure out a solution to not have the page fault happen and implement it. You may want to study the `struct thread` in `threads/thread.h`, and the `start_process()` function in `userprog/process.c`. The solution should not be more than 1-2 lines of code.

Note: Once you get started working on Pintos properly, you will find that you are unable to step through the code of the programs you run within Pintos (like `noop`). This is expected behaviour since `pintos-gdb` is debugging the OS code, and can't quite handle debugging separate code in a different program outside of Pintos. This should not present any issues though, since the example programs provided by us should not include bugs (the programs in Assignment A were buggy on purpose, they don't count).

Reading the code

One very important aspect in this lab series is to read code and documentation to build your own understanding of what is going on, reason about why certain solutions were implemented instead of others and so on. Probably the most important tool you will need to use is "Read the code, and start to reason about it". We do not expect you to read and understand the entire Pintos project, but we do expect you to read and study the functions and structures that is in direct contact with your solutions. We also expect you to understand the code that you add, or otherwise work towards understanding it!

6 Demonstration

To demonstrate this lab, you will need to be able to:

- Explain the issues in Assignment A and how you solved them. Think about the mistakes the original programmer made.
- Explain the pointer manipulations done in Assignment B and show that there isn't a potential memory leak in your list.
- Explain the problem in Assignment C and your solution for it.