TDDE47/TDDE68
Lab 0: Lists and setup

Dag Jönsson

January 16, 2025

1 Goal

This document describes the course's first lab, which includes linked lists manipulation, debugging, and Pintos setup. Completing this lab should give you the basic knowledge to get started on working on Pintos.

2 Prerequisites

To be able to work on the labs you need working knowledge on how to use the university's Linux systems, as well as on how to work with git. If you need to refresh your memory, check back on the introductionary courses. A useful tutorial for git can be taken at **Learn Git Branching**.

3 Overview

This assignment covers:

- Setting up your git project
- Understanding the internals of linked lists (pointers)
- Using GDB to debug C programs (and Pintos)

4 Setup Your Git Project

Create a **PRIVATE** project at `gitlab.liu.se`, suppose in this example that the project path is `https://gitlab.liu.se/LIU-ID/PROJECT-NAME`.
Important: Untick the checkbox to initialise the repository with a README. Failure to do this creates a default branch called 'main'.

Remember to invite your lab partner as at least developer, and your assistant as at least reporter.

Clone Pintos from our repository using the following command in the terminal:

```
git clone git@gitlab.liu.se:cpos/pintos.git
```

or if you don't have SSH keys configured (highly recommend to fix the SSH keys):

```
git clone https://gitlab.liu.se/cpos/pintos
```

Go to the created directory:

```
cd pintos
```

Add a remote repository, the one you created for yourself:

```
git remote add student git@gitlab.liu.se:LIU-ID/PROJECT-NAME.git
```

or, again if you don't have SSH keys configured:

```
git remote add student https://gitlab.liu.se/LIU-ID/PROJECT-NAME
```

Push the project to the new remote, only needed to be done once:

```
git push -u student main
```

Now you can work with the git project like normal.

5 Linked List Implementation

This excersise 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.

Your task is to implement a singly linked list. In the `pintos/standalone/slist` directory you are given some skeleton files to get you started.
Using **C**, implement the following functions (already declared in the `list.h` file):

```
1  /* puts x at the end of the list */
2  void append(struct list_item *first, int x);
3
4  /* puts x at the beginning of the list */
5  void prepend(struct list_item *first, int x);
6
```

```
7  /* input_sorted: find the first element in the list larger than x
      and input x right before that element */
8  void input_sorted(struct list_item *first, int x);
9
10 /* prints all elements in the list */
11 void print(struct list_item *first);
12
13 /* free everything dynamically allocated */
14 void clear(struct list_item *first);
15
```

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

Your solution should dynamically allocate the necessary memory with `malloc`, and of course `free` it when it's no longer used. You will also need to write your own tests to validate that your solution works as intended. Make sure you cover all the relevant cases. Another requirement is that your solution doesn't leak any memory, to check this you can use Valgrind:
`valgrind --tool=memcheck --leak-check=yes ./main`

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

Optional challenge
In the skeleton files the list is set up with a sentinel starting node that is a known value that is not a part of the actual list. If you want to challenge yourself, implement the list without such a node. You will want to look into double pointers.

Demonstration: At the end of the entire lab assignment, you will be asked some questions on how you implemented and tested your solution.

6 GDB

This excersise aims to give you a short introduction to GDB. This will be done outside of Pintos as to make it easier for you to focus on GDB. After the completion of this excersise you should have basic knowledge on how to use GDB to find and correct bugs in your code.
The GDB manual can be found here: GDB Documentation. Use it as a reference when debugging. To get started you should look through the Sample session page.

In the `pintos/standalone/gdb` directory there are three C-programs, `debug1.c`, `debug2.c`, and `debug3.c`. These programs contains bugs that will cause the program to crash. You task is to find these with the help of GDB, and then correct them in a reasonable way.

Demonstration: At the end of the entire lab, you will be asked some questions related to the errors in these programs, and how you found them.

Note: The versions of `malloc` and `free` used in these programs are designed to crash faster than the normal versions. This does mean that if you later make the same errors in your Pintos code you might not crash during your initial testing, but rather after adding unrelated code.

Some useful GDB commands to know:

- `print var` - prints information about the variable var.
- `print *var` - prints information about the derefenced pointer var.
- `run` - run the program until either a problem occurs, a breakpoint occurs, or the program finishes.
- `continue` - Continue running after execution stopped.
- `break main` - set a breakpoint at the start of the main function
- `backtrace` - print information about the call stack. Useful when an error has occured 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.

6.1 `debug1.c`

Navigate to the `pintos/standalone/gdb` directory and compile `debug1.c` with the command `make`. Running this program will result in a `Segmentation fault (core dumped)` message. To get started debugging the issue, execute the command `gdb ./debug1`. You should be greeted by the GDB shell.
Once GDB has started up, you can give it the command `run` to run the program, and GDB will run the `debug1` program and stop once it runs into the SEGFAULT.

Use the `print` command in GDB to figure out what the issue is. Start with examining what the variable `i` is, and how it relates to the array we are iterating over.
Once you figured out what the problem is, implement a fix and recompile and check if it worked.

6.2 `debug2.c` and `debug3.c`

Repeat the process for the two other files, compile and run them under the debugger, figure out the problem and implement a fix.

7 Pintos Setup

Add the `pintos/utils` directory to your PATH environment variable. This can be done with the following if you are currently located in the `pintos/utils` directory: (**You only need to do this once!**)

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

After that is done, execute the following to reload your `.bashrc` for your current session, you do not need to do this again in the future.

```
source ~/.bashrc
```

7.1 Compilation

To compile Pintos for the first time, change your current directory to the threads directory (`cd pintos/threads`) and issue the command:

```
make -j
```

The "-j" flag tells make to use several threads to compile, speeding up the process.

Once compiled, run the following command:

```
pintos run alarm-single
```

Your terminal should output something similar to the following:

```
SeaBIOS (version 1.13.0-1ubuntu1.1)
Booting from Hard Disk...
PPiiLLoo  hhddaa1
1
LLooaaddiinngg.................
Kernel command line: run alarm-single
Pintos booting with 3,968 kB RAM...
367 pages available in kernel pool.
367 pages available in user pool.
Calibrating timer...  392,704,000 loops/s.
Boot complete.
Executing 'alarm-single':
(alarm-single) begin
(alarm-single) Creating 5 threads to sleep 1 times each.
(alarm-single) Thread 0 sleeps 10 ticks each time,
```

```
(alarm-single) thread 1 sleeps 20 ticks each time, and so on.
(alarm-single) If successful, product of iteration count and
(alarm-single) sleep duration will appear in nondescending order.
(alarm-single) thread 0: duration=10, iteration=1, product=10
(alarm-single) thread 1: duration=20, iteration=1, product=20
(alarm-single) thread 2: duration=30, iteration=1, product=30
(alarm-single) thread 3: duration=40, iteration=1, product=40
(alarm-single) thread 4: duration=50, iteration=1, product=50
(alarm-single) end
Execution of 'alarm-single' complete.
```

To quit and go back, press `CTRL+C` to terminate the pintos process.

8 Pintos GDB debugging

Using a debugger is a useful way to find bugs or learn about programs. In Pintos, you may use GDB as a debugger and issue commands to insert breakpoints or inspect memory, for example. You may use the **Pintos debugging material** as a reference although other GDB references should work in this context too. Complete the following task by using GDB:

Move into the directory `pintos/examples` and compile the programs by issuing `make -j`.

Now, move into the directory `pintos/userprog`, find the function `setup_stack()` in the file `userprog/process.c` and change (you will fix this back in the next lab):

```
*esp = PHYS_BASE; to
*esp = PHYS_BASE - 12;
```

Compile the code by issuing `make -j` in that directory.

This should have created the `build` subdirectory, move into it. Use the following command to create a simulated disk:

```
pintos-mkdisk filesys.dsk --filesys-size=2
```

Then, format the disk:

```
pintos -f -q
```

Still in the `build` directory, copy the previously compiled binary `printf` to the simulated disk with the folllowing command:

```
pintos -p ../../examples/printf -a printf -- -q
```

The `printf` program should now be listed when you run:

```
pintos -q ls
```

You will start pintos with the GDB flag and tell it to execute the `printf` program with the following command:

```
pintos --gdb -- run printf
```

In another shell, move to `pintos/userprog/build` and execute:

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

The GDB shell will be open, execute the following to connect to the VM:

```
debugpintos
```

If you looked at the source of the `printf` program, you noticed it just calls the `printf` function and passes a string as the argument. Internally, a wrapper will push three parameters to the stack to perform a system call: the number of the system call (you can check the macros in the file `pintos/lib/syscall-nr.h`), a file descriptor to print to and a pointer to the string to be printed. Since the system call implementation is a task for a future lab, we are going to insert a breakpoint into the kernel handler of system calls. Find it in: `pintos/userprog/syscall.c`:

```
(gdb) break syscall_handler

Breakpoint 1 at 0xc010a800:  file ../../userprog/syscall.c, line
18.
```

Continue the execution until the program finishes or a breakpoint is hit:

```
(gdb) continue

Continuing.

Breakpoint 1, syscall_handler (f=0xc011ffb0)
at ../../userprog/syscall.c:17
```

You are now in the GDB shell at the breakpoint that was hit. You may use the `list` command to check the source code of the program at the current part that is being executed. Note that the `syscall_handler` function receives a struct as

parameter, check out the members of the struct with the command:

```
ptype f
```

As you can see, the interrupt handler has provided a struct with the registers and other information from the userspace program. Since the wrapper of the `printf` function has pushed the parameters to the stack, we are interested in the stack pointer to inspect its memory contents:

```
(gdb) print f->esp

$1 = (void *) 0xbffffed8
```

Based on the stack pointer, take a look at the memory contents using examine (short command `x`). In the following command we are examining five words as of the address of the stack pointer:

```
(gdb) x/5w f->esp

0xbffffed8:  9 1 -1073742044 55

0xbffffee8:  -1073742072
```

To print the third parameter (recall it is the pointer to the string), you can manipulate the `esp` pointer. Given that each parameter is 4 bytes long, issue the following command to print the address of the third argument:

```
(gdb) p f->esp+8

$2 = (void *) 0xbffffee0
```

Use GDB and the information presented to answer the following:

- What name does the syscall number correspond to? (Remeber you can find the names in `pintos/lib/syscall-nr.h`.)
- What is the second parameter related to and what does it mean in this case?
- Use GDB to print the string that the pointer in the third parameter refers to. **Hint:** Use the `x/s` command variant to examine memory and treat it as a string. You need to dereference the pointer using an asterisk to access the contents of the memory (just like in **C**). Since GDB doesn't know the data type of the memory location that the pointer points to (it's a void pointer), you also need to cast it to a **(char **)**.

9 Acknowledgement

Parts of this document contains material from the TDIU16 course at LiU, previous lab instructions found on the course web page, or from previous lab instructions written by Felipe Boeira.