Operating Systems Lab
Dept of CSE, LNMIIT, Jaipur
08 March 2021
Lab Session 06

Introduction

This lab session is about the tools for working with large software. We need software development tools to read, understand, and modify PintOS code under directory `src/`. Pintos documentation describes several tools to support software development. Read PintOS booklet to learn and understand these tools. At the minimum, read pintos document section https://web.stanford.edu/class/cs140/projects/pintos/pintos_11.html#SEC158 and also the coding standards https://web.stanford.edu/class/cs140/projects/pintos/pintos_8.html#SEC138

One useful tool discussed in the booklet is cscope. If you want an editor option other than vim with cscope, this link may help (not tried by your LNMIIT tutor): https://atom.io/packages/atom-cscope

We suggest that you learn vim https://youtu.be/wlR5gYd6um0 This will help you over your professional careers. However, if you are reluctant, here is escape sequence is you get struck in vim: <escape>:q!<return> This will ALWAYS get you out of vim trap.

Some tools described in the booklet are now dated. Git has become de facto version control tool. We recommend the students to use Git for version control. A suggested Git tutorial video is here: https://youtu.be/ev_byvSWvr0

This lab session begins with a Git related task. Students will initiate a Git repository for PintOS. This git repository will let the students reconstruct any previously committed version of the software during PintOS development if an attempt does not yield good outcomes.

Through Tasks 02 onwards, students will begin to read PintOS source code under directory `src/threads`. There is more code there than what the students have seen so far; however, pintos code is extremely well written and lucidly documented. It is very educative too.

While working on *threads* project, students can use `printf()`. This will be a lot of relief. If we were to continue to `useprog` project, `printf()` would be unavailable to the user programs till system calls have been implemented. Without `printf()` exploring unknown code is very difficult. Luckily, we will not face this restriction in the lab session today. Insert `printf()` statements to track PintOS execution. However, remember printf() is slow and cannot be in the interrupt handling code.

We have found two small but interesting PintOS tasks from two US universities for your lab today. Both task descriptions include easy-to-follow advices. Completing these tasks will not be too difficult. And the students will learn many PintOS code details including `struct thread` (process control blocks), scheduling and interrupt handling.

Much of the work in this lab session will be completed through command line terminal (windows). Some commands will require you to be in a specified working directory. Use command `cd` to change directory.

Task 01: Learn Git and Setup Git Repository for Pintos

On the virtual computer created for PintOS developments, create a folder in which you will place PintOS working code.

Download and install Git software on your computer. Typically, this will be done as follows:

```
sudo apt-get update
sudo apt-get install git
sudo apt-get install git-gui
```

Create a new work directory where you will develop code today. Let us call this directory `lab06`. Change directory (cd) to `lab06` on you command-line terminal window. And, run the following commands to initiate git local repository (Use your name and email address):

```
git init
git config user.name "tutor"
git config user.email abc@xyz.pqr
git config --list
```

Copy PintOS `src` folder to folder `lab05`. It will be a good idea to run `make clean` commands on your source directories to remove unneeded files:

```
cp -r ~/os/PintOS/src ./
git add .
git commit -m "Testing"
```

Now delete `src/`

```
rm -r src
git checkout ./
```

Your files are again available to you! However, read the footnote[1] carefully. Make some change in a file. For example, in file `src/threads/Make.vars` change the name of the simulator. Use the git's graphic user interface (GUI) tool to view new version of the file. The editor window highlights the changes made to the file:

```
git citool
```

The GUI interface is useful for merging and resolving conflicts that may occur due to simultaneous changes to the same file at different locations. This is often the case when a software is modified by a team of developers. Typically, a common remote repository is used to store the central edition of the shared files. Individual developers pull the files to their local computer-based repositories. The files are then checked out to the work directories.

As a single user you may not need a remote repository. We leave the advanced topics of branching, merging and remote repositories for you to learn later. Here is a good resource to learn more about git: https://www.git-tower.com/learn/git/ebook/en/command-line/remote-repositories/introduction/

[1] Lab06 is your local repository directory. It contains .git folder that carries all information needed to reconstruct previous versions. If we were to delete folder Lab05, the repository will be removed, and we will not be able to restore any file! Only files that are under version control inside lab05 can be restored through git checkout command.

Students not familiar with vim and emacs may consult this site to pick a suitable editor. https://git-scm.com/book/tr/v2/Appendix-C%3A-Git-Commands-Setup-and-Config#_core_editor

Task 02: Learn GDB and Explore Code

A self-explanatory hands-on tutorial about use of gdb on C programs has been sourced from a US university. It has been copied as an appendix with this document. Read the lessons there.

PintOS use of GDB is trickier and requires elaborate setups. Fortunately, suitable macros and scripts have been included with PintOS distribution. We will look at GDB in Task 04.

Task 03: Adding Code to Pintos – A Micro Shell

(This exercise is sourced from https://cs.jhu.edu/~huang/cs318/fall18/project/project0.html) Remember to use Git facilities so that you can revert to the original unmodified code if things go astray.

Add a Kernel Monitor

(This part is quoted from the source with a few minor alterations)

> At last, you will get to make a small enhancement to Pintos and write some code! When Pintos finishes booting, it will check for the supplied command line arguments stored in the kernel image. Typically, you will pass some tests for the kernel to run, e.g., `pintos -- run alarm-zero`. If there is no command line argument passed, the kernel will simply finish up. This is a little boring. You task is to add a tiny kernel shell (call it micro-shell) to Pintos so that when no command line argument is passed, it will run this shell interactively. Note that this is a kernel-level shell. In later projects, you will be enhancing the user program and file system parts of Pintos, at which point you will get to run a regular shell.
>
> To inspire script pintos to set up and turn-on bochs/qemu you need to provide some command line arguments. For example, `pintos micro-shell`. Add code for this simple interactive shell. It starts with a prompt `LNMIIT>` and waits for user input. When a newline is entered, it parses the input and checks if it is `whoami`. If it is `whoami`, prints your name. Afterwards, the monitor will print the command prompt `LNMIIT>` again in the next line and repeat. If the user input is `exit`, the monitor will quit to allow the kernel to finish. For the other input, print message `invalid command`. Remember bochs/qemu are bare machines. Neither simulator have OS or elaborate libraries to read strings.

Micro-shell Exercise

Enhance "threads/init.c" to implement a tiny kernel monitor in Pintos. Feel free to add new source files into the Pintos code base for this task, e.g., write a readline library function. Refer to Adding Source Files for how to do so.

Hint: You may need to use some functions provided in "lib/kernel/console.c", "lib/stdio.c" and "devices/input.c".

Note for LNMIIT:

For the prebuild Pintos2021 virtual computer, the new code will be added at line 133 in file `threads/init.c`. You can write code there or decide to write a new function that implements the micro-shell. The latter option is preferred. It will keep your code separate from the original PintOS code. But you need to read the description to find how a new file is included within "make" arrangement available with PintOS.

To plan this function, determine what values `argv` carries when we run test examples using command: `pintos run alarm-single`. Also try command: `pintos shell`

The video introduction and suggestions for writing micro shell are not foolproof. Students must make sure that their final version is free from all warning messages.

Task 04: Tracing PintOS Code

Once you have completed Task 03, you should feel ready to explore PintOS code. Let us use John Hopkin University (JHU) guide to explore and understand PintOS code.

First work on the example in PintOS document at section E.5.2 https://web.stanford.edu/class/cs140/projects/pintos/pintos_10.html#SEC151.

One LNMIIT tutor followed the instructions using a different test program. The tutor changed working directories (cd) for both debugger terminal windows to directory `~/os/PintOS/src/threads/build`. However, the tutor used a different example to start `gdb` aided tracing.

```
pintos -v --gdb -- -q run alarm-single
```

Tutor's first command after `debugpintos` on the second terminal was: `b test_alarm_single`. Look into file `test.h` to understand the reasons for this choice of break-point. Gdb command `bt` helped the tutor to know the (function) call sequence leading to call to function `test_alarm_single`.

Use gdb as one of the support tools to trace the execution of the program. Other program trace options, including `printf()` are also necessary and useful in understanding the program behaviour.

If you do not know the I-386 assembly language, do not be too concerned. Trust the available descriptions in the document. The JHU guide discloses that loaded PintOS C-code takes control at function `pintos_init()`.

Tracing execution from this point on will help you understand how threads are created and assigned different test functions to run during the `make check` sessions.

Another place of interest in the code is function `run_actions()` in file `init.c`. This is where the tests included in collection `threads/tests` (used by `make check`) land. However, file `src/Makefile` does not include `src/tests/threads` directory for creating `cscope` indices. You may include it the test cases in the index by updating the first line in src/Makefile as follows:

```
BUILD_SUBDIRS = threads userprog vm filesys tests/threads
```

Task 05 may be used to provide purpose and directions your explorations in Task 04.

Terminal 1 Activities	Terminal 2 Activities
`cd ~os/PintOS/src/threads/build`	`cd ~os/PintOS/src/threads/build`
`pintos -v -gdb -- -q run alarm-single`	
	`pintos-gdb kernel.o`
	`debugpintos`
	Use commands to trace things. Some of them are: `b` To set break point `d` To remove breakpoint `bt` Back trace `n, s` Next, Single statement `c` Continue `p var` Print var `b funcName` breakpoint `b Line number`

Task 05: Using Semaphores

(Sourced from University of Chicago: https://uchicago-cs.github.io/mpcs52030/p0.html) I copy the relevant part of the description from their website:

> The goal of Project 0 is to ensure you have gotten to the point where you are comfortable compiling the Pintos code, running it, as well as running the tests (including individual tests). It will also require that you start familiarizing yourself with the data structures in the Pintos kernel.
>
> To do this, you are only required to implement the Alarm Clock part of Project 1. However, the Alarm Clock requires using a semaphore, but we will not have covered semaphores in depth before Project 0 is due. So, we will tell you the exact semaphore code you need to include:
>
> - You will need to add the following field to struct thread in thread.h:
>   `struct semaphore timer_sema;`
> - You will need to include the following call in init_thread in thread.c:
>   `sema_init(&t->timer_sema, 0);`
> - Suppose variable `cur` contains a pointer to the currently running thread (a pointer to `struct thread`, as returned by `thread_current()`). To put that thread to sleep, do the following:
>   `sema_down (&cur->timer_sema);`

- Suppose variable `th` contains a pointer to a `struct thread` for a thread that was previously put to sleep as described above. To wake up that thread, do the following:
  ```
  sema_up (&th->timer_sema);
  ```

With the above, you should be able to implement Project 0 just using the Pintos documentation and using what you will learn in the Processes and Threads part of the first lecture.

You will also have to avoid certain race conditions, but this will not require using semaphores, locks, etc. As stated in the Pintos documentation "the only class of problem best solved by disabling interrupts is coordinating data shared between a kernel thread and an interrupt handler" which is precisely what happens when implementing the Alarm Clock. This, in turn, will require that you understand interrupts and interrupt handlers.

To the LNMIIT Students on Task 05:

We will not include MLFQS exercises in any lab session this year. You may ignore all test cases with mlfqs tags.

Returning to the Chicago task: The available guidance blocks the threads using `timer-Sema` semaphore. You need to create and initialise one `timer_sema` for each thread. To guide you a bit more, a possible place for invoking `sema_down (&cur->timer_sema);` is in function `timer_speep()` in file `timer.c`. A thread can be blocked before or after call to `thread_yield()`. If the thread is unblocked before the desired delay, it will block again. (Do not remove `thread_yield()` from the statements in the loop yet.)

Who releases a blocked thread? How is the wait time for each thread determined? These are important answers that you need but we will leave them for a later lab session.

This lab session is primarily to learn Git and GDB basics; we will play dirty to get out of jail. Invoke `sema_up()` in the code run by thread `idle`. It is an indiscriminate act but ok for now. You may wish to use function `thread_foreach()` to help you in this task.

For more details, if needed, read information included in files `src/lib/kernel/list.h/c`.

It will be useful to include a few `printf()` calls with `sema_down()` and `sema_up()` to list the name of the thread as their wait status changes.

Students may wish to watch video prepared for Lab Session 08 for some guidance.

NOTE: You must study library `src/lib/kernel/list` comprehensively before Lab Session 08.

Appendix 1: GDB (Sourced from the link below on 22 Jan 2021

https://usfca.instructure.com/courses/1298668/pages/lab-gdb?module_item_id=14034149)

Lab: GDB
Debugging with GDB
In this lab you will learn how to use GDB for debugging C programs. GDB is useful for debugging normal applications as well as kernel code. A little investment in learning the basics of GDB will save you a lot of time later.

1 An Introduction to GDB

Debugging is programming. You will spend more of your time debugging than writing code. Even very experienced programmers spend a significant time debugging. Furthermore, even if you write rock solid code, chances are you are going to be using libraries and system calls written by someone else. So, you can also end up debugging problems surrounding the interfaces between your code and other code.

The GNU debugger (gdb) is a very popular command-line tool for stepping through and analyzing C code. It can also be used with other languages such as C++ and assembly. For our purposes, we will also use gdb to debug some user-level C code the Pintos operating system kernel. Pintos is setup to allow for remote debugging of the Pintos kernel. Using gdb with Pintos will be invaluable for the remaining projects.

The basic idea is that you first start gdb by giving it the name of the program (executable) that you want to debug (debugging a kernel will be a little bit different). Once inside gdb, you can set breakpoints at arbitrary locations in the source code. Then you can run your program until the code hits a breakpoint or until the program causes an exception (like a segmentation fault, also known as dereferencing a bad pointer).

The gdb program is most useful in finding out how a program is executing by watching the program flow and by inspecting the values of variables at run time. Although gdb is very sophisticated, you only need to learn a small subset of the commands to make it useful. In linux, there are some GUI debuggers based on gdb, such as ddd. To get more information on gdb type info gdb.

2 GDB Summary

- gdb is an interactive, source level, command-line debugger.
- Prepare a binary at compile time using the -g option: `gcc -g -o foo foo.c`
- gdb can be used to set breakpoints. While running your program from within gdb, you program will stop at each break point so you can inspect the values of important variables.
- gdb allows you to step through your program one statement at a time.
- gdb can be used to find out where a segmentation fault occurs.
- gdb allows you to show a *backtrace* of the call stack. This allows you to see the chain of functions calls that leads to the current statement.

3 GDB Command Summary

- r (run) - start the program to be debugged

- b line-number (breakpoint) - set a breakpoint at the specified line number. Use b function-name to break at a function.
- info b - show all your breakpoints
- d - delete all breakpoints
- d number - delete a specific breakpoint
- c (continue) - continue program execution after stopping at a breakpoint or for CTRL-C.
- n (next) - execute the next statement (do not go into functions)
- s (step) - execute the next statement (go into functions)
- p var (print) - print the value of a variable
- bt (backtrace) - show the function call stack
- q (quit) - quit gdb

4 Preparing a program for debugging.

To debug a program with gdb you need to compile your source code (all source modules that you want to debug) with the -g option:

```
gcc -g -o hello hello.c
```

5 Starting GDB

Simply start gdb with the name of your executable:

```
$ gcc -g -o hello hello.c

$ gdb hello

GNU gdb 5.1.1
Copyright 2002 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-suse-linux"...
(gdb)
```

To start your program within gdb, simply use the “r” command (for run):

```
(gdb) r
```

Optionally, you can put command arguments that you want to send to your program after the “r” command:

```
(gdb) r arg1 arg2 ... argN
```

6 Example 1 - Setting Breakpoints and Examining Variables

Here is a program that is supposed to add 2 to the variable j in every iteration of the for loop:

```
#include <stdio.h>

int main(int argc, char *argv[])
{
        int i, j = 0;
        for (i = 0; i < 100; i++); /* <-- unwanted semicolon */
```

```
                j += 2;
        printf("The value of j is: %d\n", j);
        return 0;
}
```

Compiling and running the program reveals that it is not adding 2 to j 100 times:

```
$ gcc -g -o ex1 ex1.c
$ ./ex1
The value of j is: 2
```

Because we compiled ex1.c with the -g option we can run gdb on the ex1 executable.

```
$ gdb ex1
GNU gdb 5.1.1
Copyright 2002 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-suse-linux"...
(gdb) l
1   #include <stdio.h>
2
3   int main(int argc, char *argv[])
4   {
5        int i, j = 0;
6        for (i = 0; i < 100; i++); /* <-- unwanted semicolon */
7               j += 2;
8        printf("The value of j is: %d\n", j);
9        return 0;
10      }
(gdb)
```

Using the b command, we can set a breakpoint at the beginning of the main function.

```
(gdb) b main
Breakpoint 1 at 0x8048456: file ex1.c, line 5.
```

Now we can begin executing the program using the the r command:

```
(gdb) r
Starting program: /home/benson/tch/usf/cs326/2002F/handouts/03-gdb/src/ex1

Breakpoint 1, main (argc=1, argv=0x3ffff184) at ex1.c:5
5               int i, j = 0;
```

The n command executes the next statement in the program:

```
(gdb) n
6       for (i = 0; i < 100; i++); /* <-- unwanted semicolon */
(gdb) n
7       j += 2;
(gdb)
```

Note that the line displayed after typing n is the next line of the program to be executed. Up to this point we have executed the for-loop, but have not executed j += 2. Now we can use the print command to view the values of i and j:

```
(gdb) p i
$1 = 100
(gdb) p j
$2 = 0
```

After executing the for-loop, j should be 200. It seems that j is not getting incremented. Close inspection of the for-loop reveals that we have a semi-colon where we do not want one. If we remove the semi-colon, j will get incremented properly. To end the gdb session we can use the quit command:

```
(gdb) q
The program is running.  Exit anyway? (y or n) y
```

7 Example 2 - Finding a Segmentation Fault

You can also use gdb to find the location of a segfault.

Consider the following program:

```
#include<stdio.h>

int main(int argc, char *argv[])
{
        int a[10];
        int *a1;

        a1 = a;
        a1 = 0;
        a1[0] = 1;
        return 0;
}
```

Let us run it under gdb:

```
[2785]orb: gdb segfault
GNU gdb 5.1.1
Copyright 2002 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-suse-linux"...
(gdb) r
Starting program: /home/benson/tch/usf/cs326/2002F/handouts/03-gdb/src/segfault

Program received signal SIGSEGV, Segmentation fault.
0x08048426 in main (argc=1, argv=0x3ffff174) at segfault.c:13
13              a1[0] = 1;
(gdb)
```

Notice how gdb shows exactly where the segfault occurs.

8 Example 3 - Examining Memory

Consider the following program:

```
#include<stdio.h>
#include<stdint.h>
#include<string.h>

struct foo {
        char name[32];
        int age;
        int weight;
```

```
};

uint8_t memory[32768];

void print_foo(struct foo *fp)
{
        printf("Name: %s, Age: %d, Weight: %d\n",
                fp->name, fp->age, fp->weight);
}

int main(int argc, char *argv[])
{
        struct foo *foop;

        foop = (struct foo *) &memory[0];

        strcpy(foop->name, "Greg");
        foop->age = 29;
        foop->weight = 150;

        foop = (struct foo *) &memory[(sizeof(struct foo))];

        strcpy(foop->name, "Tony");
        foop->age = 20;
        foop->weight = 200;

        print_foo((struct foo *) &memory[0]);
        print_foo((struct foo *) &memory[(sizeof(struct foo))]);

        return 0;
}
```

We can use gdb to look at structs in memory.

```
$ gdb memory
GNU gdb 6.1-20040303 (Apple version gdb-384) (Mon Mar 21 00:05:26 GMT 2005)
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 "powerpc-apple-darwin"
...Reading symbols for shared libraries ... done

(gdb) l
warning: Source file is more recent than executable.

15      {
16                      printf("Name: %s, Age: %d, Weight: %d\n",
17                                      fp->name, fp->age, fp->weight);
18      }
19
20      int main(int argc, char *argv[])
21      {
22                      struct foo *foop;
23
24                      foop = (struct foo *) &memory[0];
(gdb) l
25
26                      strcpy(foop->name, "Greg");
27                      foop->age = 29;
```

```
28                      foop->weight = 150;
29
30                      foop = (struct foo *) &memory[(sizeof(struct foo))];
31
32                      strcpy(foop->name, "Tony");
33                      foop->age = 20;
34                      foop->weight = 200;
(gdb) l
35
36                      print_foo((struct foo *) &memory[0]);
37                      print_foo((struct foo *) &memory[(sizeof(struct foo))]);
38
39              return 0;
40      }
(gdb) b 36
Breakpoint 1 at 0x2ac8: file memory.c, line 36.
(gdb) r
Starting program: /Users/benson/Share/Teaching/cs326/handouts/01-gdb/src/memory
Reading symbols for shared libraries . done

Breakpoint 1, main (argc=1, argv=0xbffff740) at memory.c:36
36                      print_foo((struct foo *) &memory[0]);
(gdb) print *((struct foo *) &memory[0])
$1 = {
  name = "Greg", '\0' <repeats 27 times>,
  age = 29,
  weight = 150
}
(gdb) print sizeof(struct foo)
$2 = 40
(gdb) print *((struct foo *) &memory[40])
$3 = {
  name = "Tony", '\0' <repeats 27 times>,
  age = 20,
  weight = 200
}
(gdb) x/100c memory
0x30d0 <memory>:        71 'G'  114 'r' 101 'e' 103 'g' 0 '\0'  0 '\0'  0 '\0'  0 '\0'
0x30d8 <memory+8>:      0 '\0'  0 '\0'  0 '\0'  0 '\0'  0 '\0'  0 '\0'  0 '\0'  0 '\0'
0x30e0 <memory+16>:     0 '\0'  0 '\0'  0 '\0'  0 '\0'  0 '\0'  0 '\0'  0 '\0'  0 '\0'
0x30e8 <memory+24>:     0 '\0'  0 '\0'  0 '\0'  0 '\0'  0 '\0'  0 '\0'  0 '\0'  0 '\0'
0x30f0 <memory+32>:     0 '\0'  0 '\0'  0 '\0'  29 '\035'       0 '\0'  0 '\0'  0 '\0'  -
106 '\226'
0x30f8 <memory+40>:     84 'T'  111 'o' 110 'n' 121 'y' 0 '\0'  0 '\0'  0 '\0'  0 '\0'
0x3100 <memory+48>:     0 '\0'  0 '\0'  0 '\0'  0 '\0'  0 '\0'  0 '\0'  0 '\0'  0 '\0'
0x3108 <memory+56>:     0 '\0'  0 '\0'  0 '\0'  0 '\0'  0 '\0'  0 '\0'  0 '\0'  0 '\0'
0x3110 <memory+64>:     0 '\0'  0 '\0'  0 '\0'  0 '\0'  0 '\0'  0 '\0'  0 '\0'  0 '\0'
0x3118 <memory+72>:     0 '\0'  0 '\0'  0 '\0'  20 '\024'       0 '\0'  0 '\0'  0 '\0'  -
56 '?'
0x3120 <memory+80>:     0 '\0'  0 '\0'  0 '\0'  0 '\0'  0 '\0'  0 '\0'  0 '\0'  0 '\0'
0x3128 <memory+88>:     0 '\0'  0 '\0'  0 '\0'  0 '\0'  0 '\0'  0 '\0'  0 '\0'  0 '\0'
0x3130 <memory+96>:     0 '\0'  0 '\0'  0 '\0'  0 '\0'
(gdb)
```