TDDB68/TDDE47
Lab 0: Lists and setup

Felipe Boeira

January 2023

These lab instructions are partly based on the original Pintos documentation as well as the instructions developed within the course over several years.

1 Goal

This document describes the course's first lab, which includes linked lists manipulation and pintos setup and debugging. The projects are hosted in the university's Gitlab environment. If you are not familiar with git, please take some time to go through it and learn its concepts. A nice tutorial can be taken at **Learn Git Branching**.

2 Overview

This assignment covers:

- Setting up the environment
- Understanding the internals of linked lists
- Becoming familiar with Pintos doubly linked lists
- Using GDB to debug Pintos

3 Setup Your Git Project

Create a 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.**

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

```
git clone https://gitlab.liu.se/tddb68/pintos
```

Go to the created directory:

```
cd pintos
```

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

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

To push changes to your own project, execute once (later you can push using git push):

```
git push -u lab master
```

4 Linked List Implementation

This exercise aims to implement a single linked list and understand how its manipulation works internally. This will be helpful later when you have to use other implementations of linked lists. You can create a new directory to store your implementation under the cloned project, for example: "pintos/singly-linked-list". There is another directory already there for the next exercise, don't mix these up: "pintos/pintos-linked-list". Please use different files for implementing the functions and for testing the code (you can have a C file only with the main() function to test the implementation and which includes the header where you define the linked list functions).

Using **C**, create a linked list from scratch with the following functions:

```c
void append(struct list_item *first, int x); /* puts x at the end of the list */
void prepend(struct list_item *first, int x); /* puts x at the beginning of the list */
void print(struct list_item *first);  /* prints all elements in the list */
/* input_sorted: find the first element in the list larger than x and input x right before that element */
void input_sorted(struct list_item *first, int x);
void clear(struct list_item *first); /* free everything dynamically allocated */

```

Hint1: You need to use `malloc` (and `free` of course) and a `struct list_item` that contains a pointer to the same struct type (the next element):

```c
struct list_item {
  int value;
  struct list_item * next;
};

```

Hint2: Implementing the list will be easier if you make `first` a static head element that does not contain any real data. (Because there are fewer exceptions if you know `first` is never `NULL`.) Your `main` can start like this:

```c
int main( int argc, char ** argv)
{
  struct list_item root;
  root.value = -1; /* This value is always ignored */
  root.next = NULL;
```

Write a piece of code that tests your linked list implementation. You should *properly* test all the relevant cases including, creation, insertion (all different types), printing, and clearing. Try several different orders of these functions (including printing after clearing the list). Use Valgrind to make sure you have got no memory leaks. You can use the following command:

```
valgrind --tool=memcheck --leak-check=yes ./your-program
```

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

5 Pintos Linked List

Throughout the labs, the Pintos doubly linked list data structure is used. In order to become familiar with its implementation, a standalone version has been included in the pintos repository at:

```
pintos/pintos-linked-list
```

Now, **read carefully** the files `list.h` and `list.c`. It is important that you understand how this list operates in order to use it properly. Use the file `main.c` to develop the exercise as this file contains a skeleton of the solution. The skeleton contains a menu for the program and it consists of a list of students that can be added, removed and listed. Your task is to implement the following functions:

```c
//This function should fetch a student name from the terminal input
     and add it to the list (remember to allocate memory using
    malloc as appropriate).
void insert (struct list *student_list) {
}

//This function should get a student name from the terminal input,
    remove it from the list, and deallocate the appropriate memory
    (if it exists in the list).
```

```c
void delete (struct list *student_list) {
}

//This function should print the entire list.
void list (struct list *student_list) {
}

//This function should clear the list and deallocate the memory for
    all items in the list and quit the program.
void quit (struct list *student_list) {
}
```

Hint: Note that a single `struct list_elem` cannot be used across multiple lists, this is a common source of mistakes during the development of pintos labs.

6 Pintos Setup

Compile the Pintos utilities by going into `pintos/src/utils` and executing `make`, in addition, create a symlink to qemu in that directory and make these files executable:

```
ln -s $(which qemu-system-i386) qemu
chmod +x pintos
chmod +x pintos-mkdisk
chmod +x backtrace
chmod +x pintos-gdb
```

Add the utils directory to your PATH environment variable (this command assumes you have cloned to your home folder, adjust it accordingly). For convenience, you can add this line to your `$HOME/.bashrc`.

```
export PATH="${HOME}/pintos/src/utils/:${PATH}/"
```

Important: Adjust the directory to match your own environment.

6.1 Compilation

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

```
make
```

Go down to the build directory and start Pintos to see that it runs:

```
cd build
pintos --qemu -- run alarm-single
```

You should get a new window with an output that ends with something similar to the following:

```
pintos/src/threads/build$ pintos --qemu -- run alarm-single
Writing command line to /tmp/t7sIkaPA_d.dsk...
Kernel command line: run alarm-single
Pintos booting with 3,968 kB RAM...
356 pages available in kernel pool.
356 pages available in user pool.
Calibrating timer...  246,988,800 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.
```

7 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 GDB manual** 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/src/examples` and compile the programs by issuing `make`.

Now, move into the directory `pintos/src/userprog`, find the function `setup_stack()` in the file `userprog/process.c` and change (you will fix this back when argument passing for programs is implemented):

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

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

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

```
pintos-mkdisk fs.dsk 2
```

Then, format the disk:

```
pintos --qemu -- -f -q
```

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

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

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

```
pintos --qemu -- -q ls
```

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

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

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

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

The GDB shell will be open, execute:

```
target remote localhost:1234
```

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/src/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 the next lab, we are going to insert a breakpoint into the kernel handler of system calls. Find it in: `pintos/src/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 is the macro corresponding to the syscall number in this case?
- 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 **)**.