Project 0: Wish Shell & Pintos Intro

**Due: Fri Feb 9, 2024 2:59pmDue: Fri Feb 9, 2024 2:59pm**

Ungraded, 100 Possible Points100 Points Possible

Attempt

Submitted on Feb 4, 2024 10:55pmSubmitted on Feb 4, 2024 10:55pm

**NEXT UP: Review Feedback**

Attempt 3 Score:N/A

Add Comment

**Unlimited Attempts Allowed**

Available: Jan 16, 2024 3:00pm until Feb 14, 2024 2:59pmAvailable: Jan 16, 2024 3:00pm until Feb 14, 2024 2:59pm

**Details**

**\*\*Please be aware there are *two* parts to this project. The second part can be found below.\*\***

Project 0, **Part 1**: Wish Shell

1. Unix Shell

In this project, you’ll build a simple Unix shell. The shell is the heart of the command-line interface, and thus is central to the Unix/C programming environment. Mastering use of the shell is necessary to become proficient in this world; knowing how the shell itself is built is the focus of this project.

To get the source files for this project, download aos\_wish\_shell.zip from the "Files" section on Canvas.

**Objectives:**

* Further familiarize yourself with the Linux programming environment
* Learn how processes are created, destroyed, and managed
* Gain exposure to the necessary functionality in shells

In this assignment, you will implement a Command Line Interpreter (CLI) or, as it is more commonly known, a Shell. The shell should operate in this basic way: when you type in a command (in response to its prompt), the shell creates a child process that executes the command you entered and then prompts for more user input when it has finished.

The shells you implement will be similar to, but simpler than, the one you run every day in Unix. If you don’t know what shell you are running, it’s probably Bash. One thing you should do on your own time is learn more about your shell, by reading the man pages or other online materials.

2. Program Specifications - Basic Shell

Your basic shell, called wish, is basically an interactive loop. It prints a prompt and waits for some user input:

wish>

When the user inputs something and hits enter, the shell parses the input, executes the command specified on that line of input, and waits for the command to finish. This is repeated until the user types *exit*. The name of your final executable should be *wish*.

The shell can be invoked with either no arguments or a single argument; anything else is an error. Here is the no-argument way:

prompt> ./wish

wish>

At this point, *wish* is running, and ready to accept commands. Type away!

The mode above is called *‘interactive mode’*, and allows the user to type commands directly. The shell also supports a *‘batch mode’*, which instead reads input from a batch file and executes commands from therein. Here is how you run the shell with a batch file named *batch.txt*:

prompt> ./wish batch.txt

One difference between batch and interactive modes is that, in interactive mode, a prompt is printed (wish> ). In batch mode, no prompt should be printed.

You should structure your shell such that it creates a process for each new command (the exception are *built-in commands*, discussed below). Your basic shell should be able to parse a command and run the program corresponding to the command. For example, if the user types

ls -la /tmp

your shell should run the program */bin/ls*with the given arguments *-la* and */tmp* (how does the shell know to run */bin/ls*? It’s something called the shell path; more on this in the next section).

3. Structure

The shell is very simple (conceptually): it runs in a while loop, repeatedly asking for input to tell it what command to execute. It then executes that command. The loop continues indefinitely, until the user types the built-in command ***exit***, at which point it exits. That’s it!

For reading lines of input, you should use ***getline()***. This allows you to obtain arbitrarily long input lines with ease. Generally, the shell will be run in *interactive mode*, where the user types a command (one at a time) and the shell acts on it. However, your shell will also support *batch mode*, in which the shell is given an input file of commands; in this case, the shell should not read user input (from *stdin*) but rather from this file to get the commands to execute.

In either mode, if you hit the end-of-file marker (EOF), you should call ***exit(0)*** and exit gracefully.

**To parse the input line** into constituent pieces, you might want to use ***strsep()***. Read the man page (carefully) for more details.

**To execute commands**, look into ***fork()***, ***exec()***, and ***wait()***/***waitpid()***. See the man pages for these functions, and also read the relevant book chapter for a brief overview.

You will note that there are a variety of commands in the exec family; for this project, you must use ***execv()***. You should not use the ***system()*** library function call to run a command. Remember that if ***execv()*** is successful, it will not return; if it does return, there was an error (e.g., the command does not exist). The most challenging part is getting the arguments correctly specified.

4. Paths

In our example above, the user typed *ls* but the shell knew to execute the program */bin/ls*. How does your shell know this?

It turns out that the user must specify a *path variable* to describe the set of directories to search for executables; the set of directories that comprise the path are sometimes called the *search path* of the shell. The path variable contains the list of all directories to search, in order, when the user types a command.

*Note that the shell itself does not implement* ls *or other commands (except built-ins). All it does is find those executables in one of the directories specified by ‘path’ and create a new process to run them.*

To check if a particular file exists in a directory and is executable, consider the ***access()*** system call. For example, when the user types *ls*, and path is set to include both */bin* and */usr/bin*, try:

access("/bin/ls", X\_OK);

If that fails, try:

access("/usr/bin/ls", X\_OK);

If that fails too, it is an error.

Your initial shell path should contain one directory: */bin*

*Note that most shells allow you to specify a binary specifically without using a search path, using either* **absolute paths** *or* **relative paths***. For example, a user could type the absolute path* /bin/ls *and execute the* ls *binary without a search path being needed. A user could also specify a relative path which starts with the current working directory and specifies the executable directly (*e.g. ./main)*. In this project, you do not have to worry about these features.*

5. Built-In Commands

Whenever your shell accepts a command, it should check whether the command is a built-in command or not. If it is, it should not be executed like other programs. Instead, your shell will invoke your implementation of the built-in command. For example, to implement the exit built-in command, you simply call ***exit(0)*** in your wish source code, which then will exit the shell.

In this project, you should implement ***exit***, ***cd***, and ***path*** as built-in commands.

* ***exit***:When the user types ***exit***, your shell should simply call the ***exit()*** system call with 0 as a parameter. It is an error to pass any arguments to exit
* ***cd***: The ***cd*** command always takes one argument (0 or >1 args should be signaled as an error). To change directories, use the ***chdir()*** system call with the argument supplied by the user; if ***chdir()***fails, that is also an error
* ***path***: The ***path***command takes 0 or more arguments, with each argument separated by whitespace from the others. A typical usage would be like this:

wish> path /bin /usr/bin

which would add */bin*and */usr/bin*to the search path of the shell. If the user sets path to be empty, then the shell should not be able to run any programs (except built-in commands). The path command always overwrites the old path with the newly specified path

6. Redirection

Many times, a shell user prefers to send the output of a program to a file rather than to the screen. Usually, a shell provides this nice feature with the ‘>’ character. Formally this is named as redirection of standard output. To make your shell users happy, your shell should also include this feature, but with a slight twist (explained below).

For example, if a user types

ls -la /tmp > output

nothing should be printed on the screen. Instead, the standard output of the *ls* program should be rerouted to the file named *output*. In addition, the standard error output of the program should be rerouted to the *output*file (the twist is that this is a little different than standard redirection).

If the *output*file exists before you run your program, you should simple overwrite it (after truncating it).

The exact format of redirection is a command (and possibly some arguments) followed by the redirection symbol followed by a filename. **Multiple redirection operators or multiple files to the right of the redirection sign are errors.**

*Note: don’t worry about redirection for built-in commands (e.g., we will not test what happens when you type path /bin > file).*

7. Parallel Commands

Your shell will also allow the user to launch parallel commands. This is accomplished with the ampersand operator as follows:

wish> cmd1 & cmd2 args1 args2 & cmd3 args1

In this case, instead of running *cmd1* and then waiting for it to finish, your shell should run *cmd1*, *cmd2*, and *cmd3* (each with whatever arguments the user has passed to it) in parallel, before waiting for any of them to complete.

Then, after starting all such processes, you must make sure to use ***wait()*** (or ***waitpid()***) to wait for them to complete. After all processes are done, return control to the user as usual (or, if in batch mode, move on to the next line).

Handle parallel commands which are not built-in (not implemented by you). For example, you will have to handle the following:

ls & ls

ls & echo Hello

You need not handle when one (or all) of the parallel commands are built-in (implemented by you: *exit, cd,*or *path*)

8. Program Errors

**The one and only error message you need to print whenever you encounter an error of any type:**

char error\_message[30] = "An error has occurred\n";

write(STDERR\_FILENO, error\_message, strlen(error\_message));

The error message should be printed to *stderr*(standard error), as shown above.

After most errors, your shell simply continues processing after printing the one and only error message. However, if the shell is invoked with more than one file, or if the shell is passed a bad batch file, it should exit by calling ***exit(1)***.

There is a difference between errors that your shell catches and those that the program catches. Your shell should catch all the syntax errors specified in this project page. If the syntax of the command looks perfect, you simply run the specified program. If there are any program-related errors (e.g., invalid arguments to *ls* when you run it, for example), the shell does not have to worry about that (rather, the program will print its own error messages and exit).

9. Hints

Remember to get the **basic functionality** of your shell working before worrying about all of the error conditions and end cases. For example, first get a single command running (probably first a command with no arguments, such as *ls*).

Next, add built-in commands. Then, try working on redirection. Finally, think about parallel commands. Each of these requires a little more effort on parsing, but each should not be too hard to implement.

At some point, you should make sure your code is robust to white space of various kinds, including spaces and tabs. In general, the user should be able to put variable amounts of white space before and after commands, arguments, and various operators; however, the operators (redirection and parallel commands) do not require whitespace.

Check the return codes of all system calls from the very beginning of your work. This will often catch errors in how you are invoking these new system calls. It’s also just good programming sense.

Beat up your own code! You are the best (and in this case, the only) tester of this code. Throw lots of different inputs at it and make sure the shell behaves well. Good code comes through testing; you must run many different tests to make sure things work as desired. Don’t be gentle – other users certainly won’t be.

Finally, keep versions of your code. It is highly recommended to use a source control system such as *git*. This is oftentimes an industry-standard skill to be able to utilize source control systems as a programmer. However, at the very least, when you get a piece of functionality working, make a copy of your .c file (perhaps a subdirectory with a version number, such as v1, v2, etc.). By keeping older, working versions around, you can comfortably work on adding new functionality, safe in the knowledge you can always go back to an older, working version if need be.

10. Running Tests

* Add your code to the ***wish.c*** file
* Use the ***Makefile*** to compile
* For running tests, run the corresponding test script. For example:

prompt> ./test-wish.sh

Appendix - Test Options

The *run-tests.sh* script is called by various testers to do the work of testing. Each test is actually fairly simple: it is a comparison of standard output and standard error, as per the program specification.

In any given program specification directory, there exists a specific *tests/* directory which holds the expected return code, standard output, and standard error in files called *n.rc*, *n.out*, and *n.err* (respectively) for each test *n*. The testing framework just starts at 1 and keeps incrementing tests until it can't find any more or encounters a failure. Thus, adding new tests is easy; just add the relevant files to the tests directory at the lowest available number.

The files needed to describe a test number *n* are:

* *n.rc*: the return code the program should return (usually 0 or 1)
* *n.out*: the standard output expected from the test
* *n.err*: the standard error expected from the test
* *n.run*: how to run the test (which arguments it needs, etc.)
* *n.desc*: a short text description of the test
* *n.pre*(optional): code to run before the test, to set something up
* *n.post*(optional): code to run after the test, to clean something up

There is also a single file called *pre* which gets run once at the beginning of testing; this is often used to do a more complex build of a code base, for example. To prevent repeated time-wasting, pre-test activity, suppress this with the -s flag (as described below).

In most cases, a wrapper script is used to call ***run-tests.sh*** to do the necessary work.

The options for ***run-tests.sh*** include:

* -h (the help message)
* -v (verbose: print each test description)
* -t *n* (run only test number *n*)
* -c (continue even after a test fails)
* -d (run tests not from *tests/* directory but from this directory instead)
* -s (suppress running the one-time set of commands in *pre*file)

Project 0, **Part 2**: Pintos Intro

1. Introduction

To develop a better sense of how an operating system works, you will also do a few projects inside a real OS kernel. The kernel we’ll be using is called Pintos and is runnable on the x86 architecture. It was developed at Stanford and is a simplified operating system that supports kernel threads, loading and running user programs, and a file system.

The primary goal of this project is to give you a chance to dip your toes into the Pintos codebase, and more specifically, to understand how the list data structure, which you'll likely be using in future projects, is implemented in Pintos.

What’s really great about Pintos is that it comes along with some pretty comprehensive documentation that provides a lot of resources and information on the OS. Before you start, it is *highly* encouraged to look through the provided information, or at the very least, look through the table of contents and see what information is provided. That way, you’ll have a better idea of where to look if you encounter issues down the road.

*Read through the Pintos Introduction [HERE](https://www.cs.utexas.edu/~vijay/teaching/mscs-aos/pintos_docs/pintos_1.html" \l "SEC1" \t "_blank)*[Links to an external site.](https://www.cs.utexas.edu/~vijay/teaching/mscs-aos/pintos_docs/pintos_1.html" \l "SEC1" \t "_blank)*.*

To get the Pintos source files, download aos\_pintos.zip from the "Files" section in Canvas.

**Objectives:**

* Familiarize yourself with some of the Pintos Operating System codebase
* Familiarize yourself with the associated Pintos documentation
* Understand how lists are implemented in Pintos

**Note on your solution:**

* You can also use make within src/lib directory to compile the files for this project and use make check in the same directory to run the test cases
* Each should pass the tests we supply to you
* If we see you are just hard coding to pass the test cases when we review your Pintos patch file, you will receive no credit

2. Background

Within the source files, “lib and ”lib/kernel” contain useful library routines. For this project, you won't be using or modifying most of these files, but it's important to know what's already provided since they'll come in handy later on. (”lib/user” will be used by user programs, starting in project 2, but it is not part of the kernel.) Here's a few more details:

*ctype.h, inttypes.h, limits.h, stdarg.h, stdbool.h, stddef.h, stdint.h, stdio.c, stdio.h, stdlib.c, stdlib.h, string.c, string.h*

A subset of the standard C library. See section [C.2 C99Links to an external site.](https://www.cs.utexas.edu/~vijay/teaching/mscs-aos/pintos_docs/pintos_8.html#SEC145), for information on a few recently introduced pieces of the C library that you might not have encountered before. See section [C.3 Unsafe String FunctionsLinks to an external site.](https://www.cs.utexas.edu/~vijay/teaching/mscs-aos/pintos_docs/pintos_8.html#SEC146), for information on what's been intentionally left out for safety.

*debug.c, debug.h*

Functions and macros to aid debugging. See section [E. Debugging ToolsLinks to an external site.](https://www.cs.utexas.edu/~vijay/teaching/mscs-aos/pintos_docs/pintos_10.html#SEC150), for more information.

*random.c, random.h*

Pseudo-random number generator. The actual sequence of random values will not vary from one Pintos run to another, unless you do one of two things: specify a new random seed value on the -rs kernel command-line option on each run or specify the -r option to pintos.

*round.h*

Macros for rounding.

*syscall-nr.h*

System call numbers. Not used until project 2

*kernel/list.c, kernel/list.h*

Doubly linked list implementation. Used all over the Pintos code, and you'll probably want to use it yourself down the road. Take time to understand this code.

*kernel/bitmap.c, kernel/bitmap.h*

Bitmap implementation. You won't have any need for it in this project, but may find it useful in the future.

*kernel/hash.c, kernel/hash.h*

Hash table implementation. Likely to come in handy for project 3.

*kernel/console.c, kernel/console.h, kernel/stdio.h*

Implements printf() and a few other functions.

3. Requirements

Implement the following list function:

int get\_sorted\_index(struct list \*threads, struct thread \*target);

Given an ***unsorted*** list of thread structs, threads, return the "sorted index" of target. If target does not exist in threads, or if the function receives NULL inputs, return -1.

In the context of this function, the "sorted index" of a thread is the index of the thread in the list when threads are sorted by their tid. Each thread has a unique tid.

Here are some examples:

Input:

threads = [

{ tid: 34, ... },

{ tid: 22, ... },

{ tid: 9, ... },

{ tid: 35, ... },

target -> { tid: 33, ... },

{ tid: 3, ... },

{ tid: 2, ... },

{ tid: 7, ... },

{ tid: 26, ... },

{ tid: 4, ... }

]

Output: 7

Input:

threads = [

{ tid: 34, ... },

{ tid: 22, ... },

{ tid: 9, ... },

{ tid: 35, ... },

{ tid: 33, ... },

{ tid: 3, ... },

{ tid: 2, ... },

{ tid: 7, ... },

{ tid: 26, ... },

{ tid: 4, ... }

]

target = { tid: 49, ... }

Output: -1

Input: get\_sorted\_list(NULL, target);

Output: -1

Don't reinvent the wheel here. Make use of the given list functions that are already implemented for you. It is required to use the given list\_sort() function for sorting the input list.

***Note****: The test cases make use of the*struct list\_elem allelem *field in the*struct thread.*Do not use the*struct list\_elem elem*field, or else you will not pass the test cases.*

Appendix - Testing & Turn-in Instructions

For Pintos projects (Project 0, Part 2 and onwards), the testing is a little different. Refer to [the Pintos documentationLinks to an external site.](https://www.cs.utexas.edu/~vijay/teaching/mscs-aos/pintos_docs/pintos_1.html#SEC8) for instructions on how to run tests within Pintos.

To turn in your assignment, submit a zip file containing only your solution .c or .patch file as indicated on Canvas. Name the file exactly as shown on Canvas. Do not place the files inside a folder and then zip the folder, as this will result in a folder inside another folder when unzipped. Instead, zip with the following command from your terminal:

zip <zip-file-name> <your-solution-file>

For example, for Project 0, you would zip up your solution with the following:

zip YOUR\_EID.zip wish.c pintos.patch

When you unzip your submission, your files should be at the top level of the directory and there should not be any nested folders.

**In order to create your Pintos patch file:**

1. Rename your Pintos solution directory to pintos-solution
2. Download and extract a fresh copy of the Pintos codebase from Canvas
3. Run the command diff -ruN aos\_pintos pintos-solution > pintos.patch
4. Ensure you pack this pintos.patch file into your final .zip submission

**If you would like to check your patch was created correctly:**

1. Download and extract a fresh copy of the Pintos codebase from Canvas
2. Run the command patch -p0 < pintos.patch
3. Re-run your tests to make sure everything was patched correctly