# Course Project

**Description**

As part of the requirements of this course, each student must work on a final project individually. The final project counts as 25% of the final grade. An additional 5% extra credit will be given to those projects including outstanding features.

The project submission deadline is available on the course calendar and on Canvas. Please consider this a strict deadline with no exceptions. Any project submitted after this date will not be accepted. Given the impact that an accidental late submission can have on the final grade, it is strongly recommended to complete the submission a few days in advance, and to double check whether submitted files were created and uploaded correctly. You can contact our Teaching Assistant ahead of time to verify that the submission has the correct format and that it can be downloaded, unpacked, and compiled without errors.

The following sections of this document describe two different options for your final project: a shell implementation or an extension to the xv6 operating system. Choose only one of these two options for your final project. You may ignore the section of this document that describes the option that you discarded.

# Option 1: Shell

The goal of this project is implementing a shell program, that is, a program that repeatedly reads commands from the user, interprets them, and translates them into a sequence of actions that likely involve an interaction with the operating system through system calls. Your code can rely on the string manipulation routines implemented throughout various homework assignments.

The shell project must include a set of required features, and can include one or more optional additional features for extra credit. This document lists the set of required features, as well as a recommended set of optional features. You are also encouraged to come up with your own ideas on how to extend the shell. Creative additions to the project will be considered as valuable contributions.

### Submission

The shell project should be submitted on Canvas as two separate files:

- One PDF file with a project report, named `report.pdf`. This report should be no longer than 2 pages, using 11pt Times font. It should include a brief description of each feature implemented in the shell, plus an execution example that can be correctly reproduced on the `cslab` machine.

- A ZIP file named `project.zip`, containing a directory named `project` with all your source files. The package should include a `Makefile` which produces an executable file called `shell` when running command `make`. It is essential that your project package can be unpacked, compiled, and executed without errors on `cslab` after running the following sequence of commands:

  ```
  $ unzip project.zip
  $ cd project
  $ make
  $ ./shell
  ```

  If you wrote your code on a machine other than `cslab`, you must make sure that your code still compiles and runs without errors on the `cslab` environment. Notice that different operating systems, LibC versions, or *gcc* versions may be incompatible with each other.

**Required features**

Your shell must include the following baseline features:

- The shell should print a `$` prompt symbol followed by a space when it is ready to receive a new command from the user.
- When a command is launched in the foreground, the shell should wait until the last of its sub-commands finishes before it prints the prompt again. A command is considered to be running in the foreground when operator `&` is not included at the end of the command line.
- When a command is launched in the background using suffix `&`, the shell should print the *pid* of the process corresponding to the last sub-command. The *pid* should be printed in square brackets. It should then immediately display the prompt and accept new commands, even if any of the child processes are still running. For example:

```
$ ls -l | wc &
[3256]
$
```

- When a sub-command is not found, a proper error message should be displayed, immediately followed by the next prompt. Example:

```
$ hello
hello: Command not found
$
```

- The input redirection operator `<` should redirect the standard input of the first sub-command of a command. If the input file is not found, a proper message should be displayed. Example:

```
$ wc < valid_file.txt
223  551  5288

$ wc < invalid_file
invalid_file: File not found
```

- The output redirection operator `>` should redirect the standard output of the last sub-command of a command. If the output file cannot be created, a proper message should be displayed. Example:

```
$ ls -l > invalid/path
invalid/path: Cannot create file

$ ls -l > hello
$ cat hello
< Contents of “hello” displayed here >
```

- Sub-commands separated by pipes should provide the desired redirections. Assuming $N$ sub-commands, you can create $N - 1$ pipes in the parent process, and let them be inherited by the child processes every time `fork()` is executed. Each process $i$ other than the first should read from pipe $i - 1$, and each process $i$ other than the last should write into pipe $i$. Remember that all processes, including the shell process itself, should close all unused file descriptors to avoid undesired blocking reads on pipes.

**Optional Features**

The following list of features are mere recommendations on how to extend your shell for extra credit. Feel free to introduce new ideas that are not included in the list.

- Modify the command prompt so that it displays the current directory, and implement the functionality of built-in command `cd` to change it. For example:

  ```
  /home/user$ cd test
  /home/user/test$ cd ..
  /home/user$
  ```

- Implement support for process-specific environment variables. These variables can be specified with a pair `VARIABLE=VALUE` appearing as a prefix of the command. For example:

  ```
  $ VAR=Hello ./test
  ```

  Write a program that prints all environment variables available to the process, and launch it from your shell. Verify that the explicit environment variables are indeed being set in the child process.

- Have the shell print a message whenever a background process finishes execution. This message would be printed when the user presses *Enter* (whether a valid or an empty command was entered), and before the prompt is displayed again. The message should print, in brackets, the *pid* of the process that finished. Example:

  ```
  $ sleep 2 &
  [5344]
  $
  [5344] finished
  $
  ```

## Option 2: Extension of xv6

The goal of this project is extending the xv6 operating system with one or more additional features. The exact extension can be selected from the project ideas included below. Alternatively, you can use these suggestions simply as a reference to the expected complexity of the project, and instead come up with your own idea. Creative and novel project ideas are strongly encouraged.

### Submission

Your xv6 project must be submitted on Canvas as two separate files:

- One PDF file with a project report, named `report.pdf`. This report should be no longer than 2 pages, using 11pt Times font. It should include a description of the modification you have applied on xv6, and an execution example that can be reproduced on the `cscigpu07` machine.

- A ZIP file named `project.zip`, containing a directory named `project` with the complete xv6 source code and your extensions. The directory should not contain any compiled binaries, which you can guarantee by running command `make clean` before generating the ZIP file.

  Your extended version of xv6 should be unpacked, compiled, and executed without errors when running the following commands on the `cscigpu07` machine:

```
$ unzip project.zip
$ cd project
$ make qemu-nox
```

**Project idea: Scheduler**

Modify the default round-robin scheduler implemented in xv6 to provide a more sophisticated policy, such as multi-level feedback queue. Optionally, you can:

- Search the literature (class textbook, other textbooks on operating systems, or research papers) or find online resources for more information on schedulers. Then implement one.
- Implement system call `nice()` to allow a process to change its own priority, in a similar way to how this system call works on Linux (see `man 2 nice`). Check how your scheduler is affected by this.

**Project idea: Virtual memory**

Add system calls that allow users to explore or manipulate the page table of the current process.

- Add a `translate()` system call that returns the physical address associated with a given virtual address, or an error code if the page is not allocated. You can add an `rtranslate()` call that does the reverse translation.
- Add an `mmap()` system call that allows the user to request new memory pages at a fixed virtual memory address, rather than just at the end of the heap.
- Modify xv6 so that NULL-pointer dereferences (and dereferences of other invalid pointers) cause segmentation faults that kill the program and notify the user.

**Project idea: Kernel threads**

Implement support for threads in xv6.

- Implement a new system call `clone()` that operates similarly to how `fork()` does, but does not replicate the memory content. Instead, the child thread's page table has the same mappings as its parent.
- Write a small library with a similar interface to *pthread*, that allows for basic thread creation and synchronization.
- Add support for locks, involving the required support from the hardware (assembly code), operating system, and thread library.

**Project idea: File system checker**

Write a tool that explores the xv6 file system, and optionally detects/fixes inconsistencies.

- The tool should traverse the xv6 file system image and dump a hierarchical representation of it

based on its directories and files. Print this information with a graphical layout using only ASCII characters in the terminal.

- Add detection for one or more of the possible inconsistencies that typical file system checkers identify, as presented in class. For example, consistency between the number of hard links and the `links` field in *inodes*.
- Extend your tool to fix recoverable inconsistencies.