

Project 2: Operating System Simulator

Eric Cronin, Gaurav Shah & Micah Sherr*

Originally prepared by Insup Lee.

cse381@seas.upenn.edu

CIS 381 - Fall 2007

1 Introduction

Now that you have written your own shell you are ready for a more ambitious group project. After splitting into groups of 2-3 you will create your own UNIX-like operating system: TOYX. TOYX is designed around subsystems which model those in UNIX. The project is designed to give you an understanding of how UNIX works, as well as to teach you about medium-scale programming in C.

In this document we have provided a base for you to build upon. You are encouraged to go beyond what we have provided and implement extra features to make a more robust system. Below, we have provided 3 overall requirements: functionality, demonstrability, and documentation. If you keep them in mind throughout the planning and development process, it will help you in the end.

Very important announcements will go out to the class mailing list, all other announcements and clarifications will be made in the class newsgroup. Useful documentation will be posted on the 381 website. Use the newsgroup instead of e-mail to ask questions.

2 Requirements

2.1 Overall

1. Functionality

you must implement the provided interface in C, and it needs to work as described and be robust in handling errors and unusual conditions. (translation: write good code that works)

2. Demonstrability

you must be able to show that your code works, through simple, clear and complete test programs (translation: you need to show us that it works).

3. Documentation

you must document your code both with in-line comments, and an external description of interfaces and expected inputs and outputs. (translation: your code needs to be understandable.)

*Based on previous projects authored by: Hee Hwan Kwak, Stuart Eichert, Scott Raven, Jon Kaplan, Robert Spier, Dianna Xu, Stefan Miltchev

2.2 Specific

- You must work in groups of 2 or 3. One of the goals of CSE381 is to learn group work and group dynamics through the project. Getting to know your classmates and forming your own group is part of it. TAs will not do any active pairing for you. If you are having trouble finding partners, then you will be assigned to a group with others in similar situations. Forming a good group is critical to the success of your project, so try your best to avoid being assigned to a group arbitrarily. Larger groups will not be considered.
- You must develop TOYX on the eniac-1 Linux cluster using user contexts.
- Give your implementation of TOYX its own name.
- You are limited to one host process for the entire simulator and one host file per simulated file system.
- Interfaces are being provided for many function calls. It is recommended, although not required, that you follow the given semantics.
- Internal implementation details should be hidden behind an abstract, well defined and well documented interface. User level “processes” (i.e. shell) should access the kernel only through these interfaces.
- Your functions should be denoted in a comment as user, kernel, program or helper.
- Observe good practice in coding and commenting. Remember that if something doesn’t work, we can’t give you any credit if we can’t figure out how it was supposed to work. If you clearly document that something doesn’t work, it will be better than not documenting that fact. Also, keep in mind that you are working with other people who will have to understand and modify your code.
- Test your system thoroughly. Stress test each function under as many different circumstances as possible, and make sure that it works even with unusual inputs, or at least does not crash. Sometimes, for the sake of time, you must cut your losses and move on to the next thing. In this case, document that your function does not work 100%, and where. However, for extra credit items, we’d rather see one thing that works 100%, than two things that are only half working. Grading will show accordingly.

3 Required Components

The functionalities of required components are characterized as follows:

(K) denotes a kernel function that may *only* be called by other kernel functions.

(U) denotes a kernel function that is called by user programs (i.e. a system call). It should provide a clear return value that can be checked for errors.

(P) denotes a user program which will be called from the shell.

3.1 Alarm Based Scheduler

TOYX must be able to schedule multiple processes through the use of an alarm-based scheduler. Processes will be based on user contexts, documented in `makecontext(3)` and `getcontext(2)`. For more information, notes from CSE381 tutorial 6 provide a tutorial. Demonstration code (a round-robin scheduler) that works on Linux is provided on the CSE381 homepage. You must modify this code to support a priority-based algorithm and on-the-fly creation and deletion of threads. When the alarm goes off, TOYX should suspend the current running process and select a new process based on the priority of waiting processes. Non-kernel functions should not block the alarm signal.

- Your entire operating system must run as one process, i.e. `fork()` is not allowed. **TOYX's "processes" will be user contexts.**
- You must show that you've protected your operating system from concurrency issues that may arise due to the timer. Pthread mutexes are not allowed for this.
- A PCB, or a process control block, is typically implemented as a structure which contains all the necessary information that is required for bookkeeping a process (for example, among other things, the process context). You must decide yourself what information your OS needs, and design your PCB structure accordingly. Refer to your 380 textbook for more information on PCBs.
- For a data structure housing all your PCBs, **do not use arrays; use linked lists.** No, don't ask, you must use linked lists. Side Note / Hint: Later on, you may find it convenient to nest various structures within your PCBs.
- We use the term 'priority' loosely. You just need to show us (read: document your algorithm) that you implemented some sort of 'priority' algorithm involving different quanta.
- If you've done something extra cool and have documented it well, then we'll consider giving you extra credit.
- Make writing clean code and documenting your code a habit.

3.1.1 Scheduler related functions

Your scheduler, in addition to the scheduling portion itself, should at a minimum support the following functions:

k_process_create (K) creates a new TOYX process and adds it to the running process queue. The new process should retain most of the parent's attributes. For example, user-id. Should be similar to what happens on an actual Unix system to a reasonable degree.

k_process_kill (K) removes a TOYX process from the process queue and cleans up all its allocated memory. May only be run by a user with the same user id as the process's user id. Root can kill all processes regardless of the process's user id.

k_handle_timer (K) called once on every clock interrupt. Chooses the next running process. Should handle the case where there are no currently running processes gracefully in order to reduce the ill-effects of busy waiting.

3.2 Process Control

Process control is another layer of abstraction above the kernel scheduling. It is a user-level interface to the functions in Section 3.1. Process Control provides a user-level interface to process management. Processes can be dynamically created, destroyed, or paused.

p_wait (U) wait (do not return) until any child process of the current process terminates, and return the exit value. If there are no children, wait should return immediately.

p_sleep (U) suspend execution of the current process for 'n' clock ticks. Other processes should continue to execute normally.

p_kill (U) kill the selected process. Kill should return an error if the process doesn't exist, or if the current user does not have permission to kill the selected process. See `k_process_kill`.

p_spawn (U) this is our equivalent of UNIX fork. create a new process, as a child of the current process.
See `k_process_create`.

p_exit (U) this is a special version of `p_kill`, which terminates the current process.

3.3 Local File System

TOYX's local file system will allow it to load and save data from permanent storage. Each "disk" in TOYX is a single UNIX file that is "formatted" so that it is understandable by your OS.

Your file system should have a hierarchical tree structure, support file permissions, and multiple disks and mounting. Implementation hints and suggestions are contained in Section 4. Provide an external "format" utility that creates and initializes your disk "files". The format program should run outside of the TOYX environment.

The listed calls are very similar to the equivalent standard ANSI C library functions. Consult K&R or man pages for specific semantics.

An implementation of these functions which passes calls through to the Linux file system (instead of implementing its own) is available on the CSE381 homepage. You may use this to develop your scheduler and user programs in parallel with your full filesystem.

f_open (U) open the specified file with the specified access (read, write, read/write, append). If the file does not exist, handle accordingly. (rule of thumb: create file if writing/appending, return error if reading is involved). Returns a file handle if successful.

f_read (U) read the specified number of bytes from a file handle at the current position. Returns the number of bytes read, or an error. (See `f_lseek`)

f_write (U) write some bytes to a file handle at the current position. Returns the number of bytes written, or an error. (See `f_lseek`)

f_close (U) close a file handle.

f_lseek (U) move to a specified position in a file.

f_stat (U) retrieve information about a file.

f_chmod (U) change permissions of a file.

f_unlink (U) delete a file.

f_opendir (U) under UNIX and our OS, directories are handled as special cases of files. open a "directory file" for reading, and return a directory handle.

f_readdir (U) returns a pointer to a "directory entry" structure representing the next directory entry in the directory file specified.

f_closedir (U) close an open directory file.

f_mkdir (U) make a new directory at the specified location.

f_rmdir (U) delete a specified directory, which must be empty.

f_mount (U) mount a specified file system into your directory tree at a specified location.

f_umount (U) unmount a specified file system.

3.4 Shell

For this assignment, you will write a simplified UNIX shell. It doesn't have to be as sophisticated as the shell you developed for Project 1. Feel free to re-use your code where appropriate.

- Your shell must be able to background processes (e.g. 'longprocess &')
- Your shell should support arbitrary redirection of input and output to files using <, > and >> (e.g. "ls > somefile"). If the specified file does not exist with > and >>, create it.

3.5 Simple Unix Programs (P):

You are required to write some user-level programs to demonstrate the functionality of your operating system. You should already be familiar with most of these programs. For more information, try them on eniac-1, and see the man pages. You should support the basic functionality of all these programs. Additional functionality triggered by command line arguments is optional, unless explicitly required below. The command line syntax of these programs should be similar to their UNIX equivalents where appropriate.

3.5.1 General Commands:

echo echos a string back to the terminal

ps lists all current processes, including priority

kill kills the specified process. The pid to kill is passed at the first command line argument.

nice runs a process with a specified scheduling priority.

sleep does nothing for the specified number of seconds. The process should not busy wait.

man displays a list of all available commands. When invoked with the name of a command as a parameter, it displays help on that command. Specifically it should list all command line options.

logout/exit terminates your OS, or current shell, if you are implementing multiple shells

3.5.2 Filesystem Related Commands:

All filesystem related commands, if applicable, must also work with stdin/stdout, that is they must also work with redirections (and pipes for extra credit).

ls lists all the files in the current or specified directory. Support flags -F, -R, -l and -t (see ls(3) for meaning)

chmod changes the permissions mode of a file. Support absolute mode (e.g. chmod 4 file)

mkdir creates a directory

rmdir removes a directory, which must be empty.

cd changes the current working directory according to the specified path. Support . and ..

pwd print current working directory

cat displays the content of one or more files to the output. Redirection should work with cat

wc displays a count of lines, words and characters in a file

rm deletes a file.

df print usage information on used/free disk space for each mounted filesystem.

mount mounts a file system at a specified location

umount unmounts a file system

3.6 Extra Credit:

Extra credit should only be attempted once the rest of your operating system is functioning perfectly. You may find it useful to have extra credit in mind during the design phase, but please don't submit an almost working OS with lots of extra bells and whistles... Below are ideas. If you have something else in mind, e-mail cse381@seas with your idea.

Multiple user support, single login: Augment your filesystem to support user IDs and encrypted/hashed passwords. See *crypt(3)*.

Groups and group permissions: Add group support to your shell. Groups consist of one or more users. As with UNIX, your filesystem should support "user", "group", and "other" permissions.

hard and soft links: Add support in your filesystem for hard and soft links (see *ln(3)*).

batch scripts: Add support for an execute bit to your filesystem. Executable files should be treated as a list of shell commands to be run in series.

sockets: Sockets allow processes to communicate with each other through system-wide named or numbered buffers. Sockets are an implementation of Local Inter-Process Communication. Messages (an arbitrary length of bytes) can be read or written.

There should be a system-wide table mapping socket numbers to their memory buffers. Only one process should be able to read from a given socket at a particular time, although anyone should be able to write to it.

You might want to implement a message protocol which is a standard way of passing data through sockets. A message might contain the source address, destination address, length of data, and the data itself.

You will demonstrate local socket support in your OS by using them to implement pipes for your simplified shell.

sort command: Implement a "sort" utility which sorts arbitrarily large input files in $O(n)$ time or better.

complex filesystems: Add support for Logical Volume Management (LVM) to your filesystem. Your filesystem should also support resizing and defragmenting.

livefloppy: Create a 1.44MB disk image, which, when copied onto a floppy disk, will boot an x86 machine into your operating system.

4 Implementations

4.1 File System Implementation Details

The file system is probably the largest piece of TOYX. Implementing it cleanly, efficiently and extensibility is a challenge.

A file system should have two components: the physical structure and the virtual structure. The basic concept of any file system is to take a large area of permanent storage and break it up into small chunks. We recommend implementing a MS-DOS compatible FAT system. As disks, floppy disk images are the perfect size. Documentation for FAT is available at http://www.seas.upenn.edu/~cse381/Disks_Directories_and_Files.pdf. Implementing this will allow you to read and write real DOS images.

If you would rather implement a different filesystem, this is allowed, but please check with the TAs first. An inode (ffs, ext2fs, sys5fs, minixfs) based filesystem is another possibility. Whichever type you choose it will need to have some method of clustering, i.e., breaking the UNIX file into sections of a fixed number of bytes. You will also need to provide for the case of a file becoming too big to fit in one cluster. It must also support a hierarchical directory system. Tannenbaum discusses physical implementations of file systems in Chapter 6, which is a good reference.

Clarification about mmap: The *mmap(2)* system call maps a file into memory. You may use *mmap* to map pages of your filesystem into “kernel” memory. For example, you may *mmap* a page into memory, read or write to that page, and then use *munmap* to commit any updates. You may **not** *mmap* your entire filesystem into memory. Using *mmap* is not necessary to complete the assignment.

4.2 Implementation of processes.

This topic is too large to cover in this document. Sample code and helpful documentation for Linux will be provided on the CSE381 web site.

5 Documentation

You **must** document your code with an automatic documentation system. You embed special tags inside C style comments `/* */` and then the program extracts them to make pretty html documentation. You will be using *Doxygen*:

<http://www.doxygen.org/>

6 General Tips

- It will be easier to implement some things as independent modules, and then merge them in once they work.
- Plan ahead. Or just plan. In the “real world” more time is often spent on the planning of projects than the actual implementation. Obviously this isn’t the real world, but every little bit of planning helps.
- You **must** use the following directory structure:
 - project2/bin/** - all compiled binaries (toyx, format) should go here.
 - project2/src/** - all source code should go here.
 - project2/doc/** - all documentation should go here.
- Use the man pages, Luke. (And K&R.)
- When in doubt about how a command behaves, always check with the real UNIX system. The closer you get to the real thing, the better your grade will be.
- Use these flags with `gcc -g, -Wall`, note that if you don’t use `-g`, symbol tables will not be generated for the debugger. Try your hardest to get rid of all warnings.

- Get to know gdb and ddd well. Don't be afraid to debug.
- Use emacs to edit your code. Let it help you with proper indentation. If you are working on an X terminal, make use of font-lock-mode, for syntax hi-lighting.
- You **must** use a Makefile to compile your project. 'make all' should compile everything. 'make clean' should delete all .o, core, and the generated executable files.

7 The 10 Commandments of TOYX

1. THOSE WHO HESITATE GET NO FRUIT CAKE - If you put this project off to the last possible moments before due dates, you will need to set up a hammock in the Moore Building. Work on a consistent basis so you are not pulling 72 hour shifts when we get into early December.
2. PLAN TWICE, CODE ONCE - One of the major keys to this project is to think ahead - how will this subsystem interact with other subsystems that I have yet to implement? What will be the best way to get them all to interact with each other? The worst way to start this project is to hit the ground coding - you *will* find yourself consistently returning to modify code.
3. CHOOSE YOUR PARTNERS WISELY - Your best friends, that person you've always had a crush on, and other folks you know may not necessarily be the best people to work with over the span of many weeks. Make sure you pick people who are diligent and will do their share of the work.
4. LEARN TO COM-MU-NI-CATE - Another key to working in a group is making sure everyone is in the loop. That includes not duplicating another persons work, or not overwriting changes another person made without them knowing it, but most importantly, not to trust an integral part of your OS (such as the scheduler) to one group member based entirely on blind faith! Your group mates may have the best of intentions, but they also have three other classes and (possibly) a life ;). Depending on your group dynamics, you may decide that you need to elect a leader to keep everyone on track.
5. DIFFICULT = SIMPLE + SIMPLE - This project may seem massive at first, but if you attack it in parts, it becomes a lot easier. For example, you can take the Local File System and break it down into a first step of getting TOYX to correctly retrieve the boot information from the disk. Once that is complete, move on.
6. IF YOU DON'T KNOW, ASK - First place to check: the man pages. Learn how to use specific catalogs by typing in the number, such as man 2 open. Next, ask your fellow students, because they may have had a similar problem and figured out a way to solve it. Finally, post to the newsgroup, and one of the TAs will answer your questions.
7. THERE IS ALWAYS ONE MORE BUG - There will always be one nagging item in a particular code segment that you think if you had five more minutes you can solve it. At some point, you have to let it go. Your time will be better spent on other tasks that need to be completed instead of making a single part of your code "perfect".
8. KEEP IT SIMPLE, STUPID - Nine times out of ten, KISS theory will give you the most comprehensible and quickest running code. If you can't understand what you wrote, your partners and the TAs won't be able to either.
9. DUE DATES ARE FIRM - This is a long term project and you should start working early enough to meet the deadline. If you cannot complete the whole project on time make sure you have at least a subset working that you can demonstrate.

10. CELEBRATE YOUR SMALL MORAL VICTORIES - Probably the item that gets overlooked, but it's very important that you recognize your accomplishments along the way. Feel good that when your presentation date arrives, you will have a fantastic project that makes other groups envious.

8 Deadlines, Grading and Turnin

Project 2 is due at **6:00pm, December 7**. Since this is a long-term project no late submissions will be accepted.

Project 2, like Project 1 will be worth 100 points. The extra credit will be worth 20 points.

For this project documentation and demonstrability will be very important and will make up a large part of your grade:

Component	Percent
Functionality	60%
Demonstrability	20%
Documentation	20%

To submit your project you should use the *turnin* program on the Eniac cluster machines. For example, if your project is contained in a subdirectory `project2` you would use:

```
turnin -c cse381 project2
```