6.828 Fall 2008 Lab 7: Final Project

Handed out Friday, November 18ish, 2011
Tarball Due Friday, December 9, 2011
In Class Demos on December 12, 2011
Private Demos December 14, 2011

Introduction

In this lab you will flesh out your kernel and library operating system enough to run a shell on the console. You will then do the core of the lab: the final project.

Lab Requirements

Unlike in previous labs, in this lab you may work in pairs. Here is a list of examples and projects from past years:

- Write a user-level debugger; add strace-like functionality; hardware register profiling (e.g. Oprofile); call-traces
- Make JOS a distributed system; implement live migration of processes/fault tolerance/distributed shared memory/etc.
- Make the file system fault tolerant: using journaling or soft updates, etc.
- Build a network file server, perhaps supporting the NFS protocol
- Add users and access control to JOS + add telnet to show users login and access control + some other security feature
- Make JOS run on real hardware and demonstrate a compelling reason for running JOS on real hardware (e.g. performance)
- Implement a virtual machine monitor with hardware support (qemu can emulate VMRUN).
- VESA graphics driver + compelling app to show it off (don't just use the "Bochs Graphics Adaptor" ports, since no real hardware supports those)
- Loadable kernel modules in JOS
- KeyKOS-style single-level store
- Binary emulation for (static) Linux executables

The project you choose must have a significant operating systems component. For example, you shouldn't simply port a user-level application that requires little or no kernel modification. You should mail a proposal to us by November 30th.

To complete the assignment, you will turn in a tarball of your code as usual. In addition, you will have to demonstrate your final project for the TAs in person on December 8th or 9th. A sign-up form will be posted soon. Every group should also be prepared to do a 5 minute in-class demo on December 12th or 14th (this may change, depending on how many groups there are).

Late policy. If you are working in a pair, you may combine your late hours, so if one of you has 20 late hours and the other has 30, you can turn in the finished code up to 50 hours late without penalty. However, the assignment *must* be turned in by the evening of Friday, December 16, even if you have enough late hours (or want to use penalty hours) to be able to hand it in later. Labs received after Friday, December 16 will receive an F. Regardless of when you turn in your code, you must still be prepared to demo your project in class on December 12th or 14th.

Getting Started

Use Git to commit your Lab 6 source (if you haven't already), fetch the latest version of the course repository, and then create a local branch called lab7 based on our lab7 branch, origin/lab7:

```
athena% cd ~/6.828/lab
athena% add git
athena% git commit -am 'my solution to lab6'
nothing to commit (working directory clean)
athena% git pull
Already up-to-date.
athena% git checkout -b lab7 origin/lab7
Branch lab7 set up to track remote branch refs/remotes/origin/lab7.
Switched to a new branch "lab7"
athena% git merge lab6
Merge made by recursive.
 kern/env.c |   42 +++++++++++++++++++
 1 files changed, 42 insertions(+), 0 deletions(-)
athena%
```

Shared Repository

Since most of you will be working in groups for this lab assignment, you may want to use git to share your project code between group members. To set up a shared git repository in AFS, one of your group members should run the following commands to create the repository in their home directory:

```
athena% cd ~/6.828
athena% mkdir project-repo.git
athena% cd project-repo.git
athena% fs sa . groupmember1 rlidwk
athena% fs sa . groupmember2 rlidwk
athena% git --bare init
Initialized empty Git repository in /afs/athena.mit.edu/user/g/r/groupmember0/6.828/project-repo.git/
athena%
```

Now you will need to decide on whose source code you will use as a starting point for your group project. That group member should run the following commands to place his or her lab7 source code into the group repository:

```
athena% cd ~/6.828/lab
athena% git remote add group /afs/athena.mit.edu/user/g/r/groupmember0/6.828/project-repo.git
athena% git checkout -b project lab7
Switched to a new branch "project"
athena% git push -u group project
Counting objects: 3, done.
Writing objects: 100% (3/3), 221 bytes, done.
Total 3 (delta 0), reused 0 (delta 0)
Unpacking objects: 100% (3/3), done.
To /afs/athena.mit.edu/user/g/r/groupmember0/6.828/project-repo.git
 * [new branch]      project -> project
athena%
```

The other members of your group should configure their git checkouts to track the project branch from this shared group repository, as follows:

```
athena% cd ~/6.828/lab
athena% git remote add group /afs/athena.mit.edu/user/g/r/groupmember0/6.828/project-repo.git
athena% git fetch group
```

```
From /afs/athena.mit.edu/user/g/r/groupmember0/6.828/project-repo.git
 * [new branch]      project    -> group/project
athena% git checkout -b project group/project
Branch project set up to track remote branch refs/remotes/group/project.
Switched to a new branch "project"
athena%
```

At this point, all of your group members should be able to pull and push commits from the shared repository by running `git pull` and `git push` on the `project` branch, respectively. Keep in mind that git only tracks files that you have explicitly committed, so you may want to run `git status` periodically to see if there are any lingering files in your checkout that you haven't committed. You can use `git add` *filename* to tell git about files you would like to commit with the next `git commit` command. If there are files that you will never want to commit, and you want to prevent them from showing up in the output of `git status`, you should create a file named `.gitignore` and add the filenames that you'd like `git status` to ignore to that file, one per line.

Sharing library code across fork and spawn

We would like to share file descriptor state across `fork` and `spawn`, but file descriptor state is kept in user-space memory. Right now, on `fork`, the memory will be marked copy-on-write, so the state will be duplicated rather than shared. (This means environments won't be able to seek in files they didn't open themselves and that pipes won't work across a fork.) On `spawn`, the memory will be left behind, not copied at all. (Effectively, the spawned environment starts with no open file descriptors.)

We will change `fork` to know that certain regions of memory are used by the "library operating system" and should always be shared. Rather than hard-code a list of regions somewhere, we will set an otherwise-unused bit in the page table entries (just like we did with the `PTE_COW` bit in `fork`).

We have defined a new `PTE_SHARE` bit in `inc/lib.h`. This bit is one of the three PTE bits that are marked "available for software use" in the Intel and AMD manuals. We will establish the convention that if a page table entry has this bit set, the PTE should be copied directly from parent to child in both `fork` and `spawn`. Note that this is different from marking it copy-on-write: as described in the first paragraph, we want to make sure to *share* updates to the page.

Exercise 1. Change `duppage` in `lib/fork.c` to follow the new convention. If the page table entry has the `PTE_SHARE` bit set, just copy the mapping directly. (You should use `PTE_SYSCALL`, not `0xfff`, to mask out the relevant bits from the page table entry. `0xfff` picks up the accessed and dirty bits as well.)

Likewise, implement `copy_shared_pages` in `lib/spawn.c`. It should loop through all page table entries in the current process (just like `fork` did), copying any page mappings that have the `PTE_SHARE` bit set into the child process.

Use `make run-testpteshare` to check that your code is behaving properly. You should see lines that say "`fork handles PTE_SHARE right`" and "`spawn handles PTE_SHARE right`".

Exercise 2. Change the file server so that all the file descriptor pages get mapped with PTE_SHARE.

Use make run-testfdsharing to check that file descriptors are shared properly. You should see lines that say "read in child succeeded", "read in parent succeeded", and "write to file succeeded".

The keyboard interface

For the shell to work, we need a way to type at it. QEMU has been displaying output we write to the CGA display and the serial port, but so far we've only taken input while in the kernel monitor. In QEMU, input typed in the graphical window appear as input from the keyboard to JOS, while input typed to the console appear as characters on the serial port. kern/console.c already contains the keyboard and serial drivers that have been used by the kernel monitor since lab 1, but now you need to attach these to the rest of the system.

Exercise 3. In your kern/trap.c, call kbd_intr to handle trap IRQ_OFFSET+IRQ_KBD and serial_intr to handle trap IRQ_OFFSET+IRQ_SERIAL.

We implemented the console input/output file type for you, in lib/console.c.

Test your code by running make run-testkbd and type a few lines. The system should echo your lines back to you as you finish them. Try typing in both the console and the graphical window, if you have both available.

The Shell

Run make run-icode or make run-icode-nox. This will run your kernel and start user/icode. icode execs init, which will set up the console as file descriptors 0 and 1 (standard input and standard output). It will then spawn sh, the shell. You should be able to run the following commands:

```
echo hello world | cat
cat lorem >out
cat out
cat lorem |num
cat lorem |num |num |num |num |num
lsfd
cat script
sh <script
```

Note that the user library routine cprintf prints straight to the console, without using the file descriptor code. This is great for debugging but not great for piping into other programs. To print output to a particular file descriptor (for example, 1, standard output), use fprintf(1, "...", ...). printf("...", ...) is a short-cut for printing to FD 1. See user/lsfd.c for examples.

Run make run-testshell to test your shell. testshell simply feeds the above commands (also found in fs/testshell.sh) into the shell and then checks that the output matches fs/testshell.key. If you're on Athena, you might need to add QEMUEXTRA=-snapshot or the test may run very slowly.

Your code should pass all tests at this point. As usual, you can grade your submission with `make grade` and hand it in with `make handin`.

The project

Challenge! Do an awesome final project!

Before you present your project during the last lecture, please submit another handin with your project code, including a brief write-up about what you did and how.

Congratulations on completing all of the 6.828 labs!