6.828 Fall 2007 Lab 6: the shell

Handed out Wednesday, November 21, 2007
Tarball Due Thursday, December 6, 2007
Private Demos on December 10 and 11, 2007
In Class Demos on December 12, 2007

Introduction

In this lab you will flesh out your kernel and library operating system enough to run a shell on the console. You will do this in steps:

1. You will generalize the file descriptor support to handle a variety of file types rather than just on-disk files. This will pave the way for pipes and a console device.
2. You will change `fork` and `spawn` to share their file descriptors with the child environment. Sharing them across `fork` and `spawn` allows the parent to set up i/o redirection before starting the child. This enables i/o redirection of child programs without needing to do anything special in the child.
3. You will implement pipes.
4. You will implement the system call linkage for the keyboard driver. We have provided the hardware driver itself as well as the console device file support.
5. You will implement pipes and redirection in the shell. We have provided a basic shell.

Lab Requirements

Unlike in previous labs, in this lab you may work in pairs. If you are working in a pair, you *must* email by Wednesday, November 28 to tell us.

To complete the assignment, you will have to demonstrate your shell for us in person the last week of the term. Dates and a sign-up form will be posted soon.

Every group should be prepared to do a 5 minute in-class demo on Wednesday December 12, 2007.

Late policy. If you are working in a pair, you may combine your late days, so if one of you has one day and the other also has one day, you can turn in the assignment up to two days late without penalty. However, the assignment *must* be turned in by the evening of Friday, December 14, even if you have enough

late days (or want to use penalty days) to be able to hand it in later. Labs received after Friday, December 14 will receive an F.

Getting Started

Download the lab 6 code from http://pdos.lcs.mit.edu/6.828/2007/labs/lab6/lab6-handout.tar.gz, and unpack it into your 6.828 directory as before. You will need to merge our new code for this lab and your source tree.

The pingpong and primes test should still work. Check this by runing `gmake run-pingpong` and `gmake run-primes`.

The file system tests will not work yet. To facilitate file descriptor sharing and multiple types of files, the implementation of routines like `open`, `close`, `read`, and `write` have changed. As the first exercise in this lab, you will adapt your solutions from lab 5 to fit in this framework.

Exercise 1: the file system switch

In this lab, we will add pipes and console input to the fledgling library operating system. Like Unix does, our system will present these resources as file descriptors manipulated via the `read`/`write`/`close` file interface.

The file support we wrote in the last lab assumed that disk files were the only possible type of file. In this exercise, we will address that shortcoming, making files a generic concept, implemented by the disk file, pipe, and console devices.

The generic file descriptor code is in `lib/fd.c`. You already encountered this code in lab 5, but now you will now need to understand it in more detail in order to implement other kinds of file descriptors.

As we outlined in lab 5, each environment has a file descriptor table located at virtual address `FDTABLE` (which happens to be 0xCFC00000). Each page in the table represents a single file descriptor. For example, file descriptor 2 is represented by the page at `FDTABLE+2*PGSIZE`. If the file descriptor is closed, there is no page mapped there. The file descriptor page contains a `struct Fd`, declared in `inc/fd.h`:

```
struct Fd
{
        u_int fd_dev_id_id;     // device id
        u_int fd_offset;        // offset for read/write
        u_int fd_omode;         // open mode
```

```
};
```

The `fd_offset` and `fd_omode` are the current file descriptor offset and open mode. `Fd_dev_id` lets us know which device implements the `read`, `write`, `close` and `stat`.

Each device exports a `struct Dev` with function pointers:

```
struct Dev
{
        u_int dev_id;
        char *dev_name;
        int (*dev_read)(struct Fd*, void *buf, u_int n);
        int (*dev_write)(struct Fd*, const void *buf, u_int n);
        int (*dev_close)(struct Fd*);
        int (*dev_stat)(struct Fd*, struct Stat*);
};
```

The `devtab` in `lib/fd.c` lists the known devices. To find the device responsible for a given `struct Fd`, `dev_lookup` looks through `devtab` for a device with `dev_id == fd_dev_id_id`. (The `struct Fd` cannot simply contain a pointer to the appropriate `struct Dev`, because we want to share them among different programs. Pointers in one program are not likely to be valid in others.) To keep things simple, the device ids are just characters: `'c'` for console, `'p'` for pipe, and `'f'` for file system.

As before, each file descriptor has a 4MB region of virtual memory reserved for its own use. The file system device still uses this range to map the file. The pipe device will use it to map the pipe buffer.

To make things concrete, let's consider the implementation of two of the generic functions: `write` and `dup`. `Write` looks like this:

```
int
write(int fdnum, const void *buf, u_int n)
{
        int r;
        struct Dev *dev;
        struct Fd *fd;

        if ((r = fd_lookup(fdnum, &fd)) < 0
        ||  (r = dev_lookup(fd->fd_dev_id_id, &dev)) < 0)
                return r;
        if ((fd->fd_omode & O_ACCMODE) == O_RDONLY)
                return -E_INVAL;
        r = (*dev->dev_write)(fd, buf, n, fd->fd_offset);
        if (r > 0)
                fd->fd_offset += r;
        return r;
}
```

`Fd_lookup`, which you implemented in lab 5, checks whether the page corresponding to `fdnum` is mapped. If not, it returns an error. Otherwise, it

stores sets `fd` to point at the page. Then `dev_lookup` searches for the appropriate device. Next, we check that the file is not open read-only. Now we have the `fd` and the `dev` and can call the `dev`-specific `write` function. If this is successful, we update the `fd_offset`.

`Dup` looks like this:

```
int
dup(int oldfdnum, int newfdnum)
{
        int i, r;
        u_int ova, nva, pte;
        struct Fd *oldfd, *newfd;

        if ((r = fd_lookup(oldfdnum, &oldfd)) < 0)
                return r;
        close(newfdnum);

        newfd = (struct Fd*)INDEX2FD(newfdnum);
        sys_mem_map(0, (u_int)oldfd, 0, (u_int)newfd, vpt[VPN(oldfd)]&PTE_USER);

        ova = fd2data(oldfd);
        nva = fd2data(newfd);
        if (vpd[PDX(ova)])
                for (i=0; i<PDMAP; i+=BY2PG) {
                        pte = vpt[VPN(ova+i)];
                        if(pte&PTE_P)
                                sys_mem_map(0, ova+i, 0, nva+i, pte&PTE_USER);
                }
        return newfdnum;
}
```

`Dup` tweaks the file descriptor table so that after the call, referring to `newfdnum` will be just like referring to `oldfdnum`. We use `fd_lookup` to find the `struct Fd` for the old file descriptor. If the file descriptor isn't valid or isn't open, we return an error. Otherwise, we can go ahead. First, close `newfdnum` in case it is already open. Then just copy the mappings for `oldfdnum` into the mapping area for `newfdnum`. First we copy the `struct Fd` page in the file descriptor table. Then we scan the virtual address space reserved for the old descriptor, copying any mappings into the virtual address space reserved for the new descriptor.

If we needed to, we could add a `dev_dup` function pointer to `struct Dev` in order to allow device implementations to run their own code in response to `dup`, but for our purposes it isn't necessary.

Make sure you understand these code snippets. You may find it useful to consult the rest of `lib/fd.c` as well as `lib/console.c`, an example device implementation.

Run `gmake run-icode` to check that file operations and spawn still work. If you see:

```
init: running sh
init: starting sh
<probably some error here>
```

then all is well.

Exercise 2: 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 that running "`(date; ls) >file`" will not work properly, because even though date updates its own file offset, ls will not see the change.) 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 both `fork` and `spawn` 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. (Note that you should use `PTE_USER`, not `PTE_FLAGS`, to mask out the relevant bits from the page table entry. `PTE_FLAGS` picks up the accessed and dirty bits as well.)

> **Exercise 2.** Change `spawn` in `lib/spawn.c` to propagate the `PTE_SHARE` pages. After it finishes setting up the child virtual address space but before it marks the child runnable, it should loop through all the page table entries in the current process (just like `fork` did), copying any mappings that have the `PTE_SHARE` bit

set.

Use `gmake 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 3. Change the file server so that all the file descriptor table pages and the file data pages get mapped with `PTE_SHARE`.

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

Exercise 3: pipes

Now we are ready to implement a new kind of file: pipes. A pipe is a shared data buffer accessed via two file descriptors: one is for writing data into the pipe, and one is for reading data out of it.

You may wish to read the pipe manual page [1] for a description, the V6 pipe implementation (`/usr/sys/ken/pipe.c`) for details, and pipe section of Dennis Ritchie's UNIX history paper [2] for interesting history.

```
[1] http://www.freebsd.org/cgi/man.cgi?query=pipe&manpath=Unix+Seventh+Edition
[2] http://cm.bell-labs.com/cm/cs/who/dmr/hist.html#pipes
```

In your library operating system, a pipe is represented by a single `struct Pipe`. For sharing purposes, each `struct Pipe` is on its own page.

```
#define PIPEBUFSIZ 32
struct Pipe {
        u_int p_rpos;           // read position
        u_int p_wpos;           // write position
        u_char p_buf[PIPEBUFSIZ];       // data buffer
};
```

The bytes written to the pipe can be thought of as numbered starting at 0. The read position gives the number of the next byte to be read. The write position gives the number of the next byte that will be written. The reader and writer share the pipe structure, but coordinate via these two variables: only the reader updates `p_rpos` and only the writer updates `p_wpos`.

Since the pipe buffer is not infinite, byte `i` is stored in pipe buffer index `i%PIPEBUFSIZ`.

To read a byte from a pipe, the reader copies `p_buf[p_rpos%PIPEBUFSIZ]` and increments `p_rpos`. But the pipe might be empty! First the reader has to yield until `p_rpos < p_wpos`.

To write to a pipe, the writer stores into `p_buf[p_wpos%PIPEBUFSIZ]` and increments `p_wpos`. But the pipe might be full! First the writer must wait until `p_wpos - p_rpos < PIPEBUFSIZ`.

There is a final catch -- maybe we are trying to read from an empty pipe but all the writers have exited. Then there is no chance that there will ever be more data in the pipe, so waiting is futile. In such a case, Unix signals end-of-file by returning 0. So will we. To detect that there are no writers left, we could put reader and writer counts into the pipe structure and update them every time we fork or spawn and every time an environment exits. This is fragile -- what if the environment doesn't exit cleanly? Instead we can use the kernel's page reference counts, which are guaranteed to be accurate.

Recall that the kernel page structures are mapped in the user environments as `pages`. The library function `pageref(void *ptr)` returns the number of page table references to the page containing the virtual address `ptr`. So, for example, if `fd` is a pointer to a particular `struct Fd`, `pageref(fd)` will tell us how many different references there are to that structure.

Three pages are allocated for each pipe: the `struct Fd` for the reader descriptor `rfd`, the `struct Fd` for the writer descriptor `wfd`, and the `struct Pipe p` shared by both. The following equation holds: `pageref(rfd) + pageref(wfd) = pageref(p)`. Therefore, a reader can check whether there are any writers left by computing `pageref(wfd) = pageref(p) - pageref(rfd)`: there are no writers if `pageref(p) == pageref(rfd)`. A writer can check for readers in the same manner.

> **Exercise 4.** Implement pipes in `lib/pipe.c`.

Test your code by running `gmake run-testpipe` and `gmake run-primespipe`. You should make it through the first few primes, but don't be surprised if, after a while, `primespipe` panics with a read or write error. We'll fix that in the next exercise.

Exercise 3b: races, races everywhere

We've gotten through the semester without worrying too much about concurrency in JOS. This was mostly intentional, since the best way to tame concurrency is to avoid it completely. We made the kernel *cooperatively scheduled*, meaning it only gives up the processor when it chooses to. Unlike UNIX, our kernel doesn't have to worry about device interrupts causing bits of program to run when we weren't expecting them.

User-space programs are *preemptively scheduled*: if an environment is the middle of something important and the clock interrupt comes along, too bad -- it has to stop and pick up again later. (This is transparent to the environment, since the kernel saves and restores its registers as though nothing had happened.)

Preemptive scheduling isn't a problem as long as all the environments are completely isolated from each other -- if they can't interfere with one another, the task switches aren't likely to be a problem.

We've seen that sometimes it's useful for different environments to "interfere constructively" with each other, in the form of IPC and shared memory pages. Unfortunately, preemptive scheduling and shared mutable state is a recipe for trouble. We'd been lucky so far, but our luck just ran out.

There are a couple of race conditions in the implementation of pipes. They center around the test for whether a pipe is closed.

Recall that in `_pipeisclosed`, we simply check whether "`pageref(fd structure) == pageref(pipe structure)`", where `pageref` uses the VPT to get the physical page number holding the given pointer and then looks up the kernel-maintained reference count for that page in the `pages` array.

`_pipeisclosed` compares two reference counts which typically change together. If, under some conditions, this comparison gives the wrong answer, we might think the other end of pipe is closed even when it isn't.

If we think of the result of `_pipeisclosed` as being stored across these two words in memory (the two reference counts), then `_pipeisclosed` might fail when the writing of the two words is not done atomically and also might fail when the reading of the two words is not done atomically. It turns out that both cases can happen. We fix the first by being careful about ordering the writes. We fix the second by implementing support for a limited form of restartable atomic sequences (RAS).

Remember that anything we do inside the kernel runs without interruption (unless we explicitly yield the processor). From user space, any such changes are atomic -- the user environment sees the whole change or none of it. Our

problems happen because the reads and writes of the data are done in user space, and thus not atomic.

The update race

The first race happens because the reference count updates do not happen atomically. Recall our supposed invariant: `pageref(p[0]) + pageref(p[1]) == pageref(pipe structure)`. (We're being a little sloppy with notation here -- in the code `pfd[0]` and `pfd[1]` are integers, so we really mean `pageref` applied to the corresponding `struct Fd` pointers.) Because the reference counts change one by one, the invariant can be temporarily violated. For concreteness, suppose we run:

```
pipe(p);
if(fork() == 0){
        close(p[1]);
        read(p[0], buf, sizeof buf);
}else{
        close(p[0]);
        write(p[1], msg, strlen(msg));
}
```

The following might happen:

- The child runs first after the fork. It closes `p[1]` but a clock interrupt comes along before the read, switching execution to the parent.
- The parent starts to `close(p[0])`, which will unmap the shared pipe structure and then unmap the fd page for `p[0]`. A clock interrupt comes along between the two unmappings, switching execution back to the child.
- Now the reference count on `p[0]` is 2 (both parent and child have it mapped) and the reference count on `p[1]` is 1 (only the parent has it mapped). But the reference count for the pipe structure is 2: the child has it mapped for `p[0]`, and the parent has it mapped for `p[1]` but not `p[0]`, which is still in the middle of being closed. The invariant no longer holds.
- The child runs. `Read` sees the buffer is empty and then decides the pipe is closed because `pageref(p[0])` and `pageref(pipe structure)` are both 2. Oops.

The same problem can happen with `dup`, if an environment is interrupted between mapping the file descriptor page and mapping the pipe data page.

Run `gmake run-testpiperace` to see the race in action. When the race actually happens, the test code will print `RACE: pipe appears closed`. (You should read `user/testpiperace.c` and make sure you understand what's going on.)

Since each system call can only map or unmap a single page, the two mappings cannot be updated atomically, so neither can the reference counts. Since two reference counts cannot be updated atomically, the invariant cannot be

maintained -- it is temporarily broken when one reference count is updated but not the other. One way to fix the problem would be to add new system calls to map or unmap two pages at a time. Then the two mappings (and thus the two reference counts) could be updated in one system call, making them atomic from the user-space point of view. We won't do this.

Instead, we can relax the invariant, since `_pipeisclosed` doesn't actually need all of it. What `_pipeisclosed` really needs is the following conditions:

- `pageref(p[0]) == pageref(pipe structure)` if and only if all the writers are closed.
- `pageref(p[1]) == pageref(pipe structure)` if and only if all the readers are closed.

The race happens because while a pipe is half-mapped or half-unmapped there is an fd reference but not a pipe structure reference, so `pageref(p[0])+pageref(p[1]) > pageref(pipe structure)`. As a consequence, `pageref(p[0])` might equal `pageref(pipe structure)` even though `pageref(p[1])` is non-zero.

Suppose instead we arrange that while pipes are half-mapped or half-unmapped, `pageref(p[0])+pageref(p[1]) < pageref(pipe structure)`. Then the conditions would be satisfied, and `_pipeisclosed` would not give incorrect answers about intermediate states.

We can do this by making sure that the fd structure is never mapped without the pipe structure also being mapped. This leads to a pair of rules:

- always map the pipe structure, then the fd structure.
- always unmap the fd structure, then the pipe structure.

If we follow these rules, then the reader reference count and the pipe reference count can only be equal when all the writers are gone, and vice versa.

> **Exercise 5.** Change the code that implements closing and dupping of pipes to follow these rules, eliminating the race. Hint: you only need to change `pipeclose` and `dup`.

Make sure you understand why it's okay that `pipe` (the function that creates a new pipe) doesn't follow these rules. (Maybe we will ask you during the meeting with the TAs.)

Test your code by running `gmake run-testpiperace`. If the race is gone, you won't see "`RACE: pipe appears closed`" and should see `race didn't happen`.

The read race

The second race occurs because there is no guarantee that the reads in `_pipeisclosed` will happen atomically. If another process dups or closes `fd` between the call to `pageref(fd)` and the call to `pageref(p)`, the comparison will be meaningless. To make it concrete, suppose that we run:

```
pipe(p);
if(fork() == 0){
        close(p[1]);
        read(p[0], buf, sizeof buf);
}else{
        close(p[0]);
        write(p[1], msg, strlen(msg));
}
```

The following might happen:

1. Suppose the child runs first after the fork. It closes `p[1]` and then tries to read from `p[0]`. The pipe is empty, so `read` checks to see whether the pipe is closed before yielding. Inside `_pipeisclosed`, `pageref(fd)` returns 2 (both the parent and the child have `p[0]` open), but then a clock interrupt happens.
2. Now the kernel chooses to run the parent for a little while. The parent closes `p[0]` and writes `msg` into the pipe. `Msg` is very long, so the `write` yields halfway to let a reader (the child) empty the pipe.
3. Back in the child, `_pipeisclosed` continues. It calls `pageref(p)`, which returns 2 (the child has a reference associated with `p[0]`, and the parent has a reference associated with `p[1]`). The counts match, so `_pipeisclosed` reports that the pipe is closed. Oops.

   (If the child checked again, `pageref(fd)` would now return 1, but `_pipeisclosed` is holding onto the 2 from before, unaware that something might have happened in the interim to change the count.)

Run "gmake run-testpiperace2" to see this race in action. Like before, you should see "`RACE: pipe appears closed`" when the race occurs.

This race is a little simpler to fix. Comparing the counts can only be incorrect if another environment ran between when we looked up the first count and when we looked up the second count. In other words, we need to make sure that `_pipeisclosed` executes atomically.

Since `_pipeisclosed` does not change any variables (it only reads them), it is safe to run multiple times. If we had some way to check whether `_pipeisclosed` got interrupted, we could repeatedly run it until it a run completed without being interrupted. Since `_pipeisclosed` is so short, it will usually not be interrupted.

To tell whether it is being interrupted, we will use the `env_runs` variable in the environment structure. Each time the kernel context switches back to an environment, it will increment `env_runs`. Thus, user code can record `env->env_runs`, do its computation, and then look at `env->env_runs` again. If `env_runs` didn't change, then the environment was not interrupted. Conversely, if `env_runs` did change, then the environment was interrupted.

Effectively, the `env_runs` counter enables support for restarting read-only atomic sequences from user space.

> **Exercise 6.** If your kernel does not already maintain the `env_runs` counter properly, change it to do so. (Hint: you only need to add one line of code.)

> **Exercise 7.** Change `_pipeisclosed` to repeat the check until it completes without interruption. Print "`pipe race avoided\n`" when you notice an interrupt *and* the check would have returned 1 (erroneously indicating that the pipe was closed).

Run "gmake run-testpiperace2" to check whether the race still happens. If it's gone, you should not see "`RACE: pipe appears closed`", and you should see "`race didn't happen`". You should also see plenty of your "avoided" messages, indicating places where the race would have happened if you weren't being so careful. The test prints the current iteration count every ten iterations. You should see a couple of "avoided" messages per ten iterations.

> *Challenge!* If multiple environments read from a pipe simultaneously, what happens? Fix this.

Real-world example: reading 64-bit interrupt counters on a 32-bit machine

This trick of restarting interrupted reads appears in other contexts as well. For example, in a preemptible x86 kernel, a system call might need to read a 64-bit counter that an interrupt routine updates. Since the x86 is a 32-bit machine, reading a 64-bit value is not atomic. Suppose the counter is at `0x 00000001 FFFFFFFF` and gets incremented to `0x 00000002 00000000` during a read. The read might get one half before the update and the other half after, yielding either `0x 00000001`

00000000 or 0x 00000002 FFFFFFFF, both of which are nowhere near the real value. We'd like the read be atomic; that is, we'd like it to return either the old or new value, but not a mix of the two.

If the counter runs slowly enough that it takes a significant amount of time to go up by 2^32, then instead of protecting the value with a lock, the reader can use the high 32 bits as a way to see whether an interrupt happened between the two word accesses:

```
// Copy 64-bit counter src into dst, being careful about read races.
struct split64 {
        // assumes little-endian memory
        uint lo;
        uint hi;
};
void
readcounter(uint64 *src64, uint64 *dst64)
{
        struct split64 *src;
        struct split64 *dst;

        src = (struct split64*)src64;
        dst = (struct split64*)dst64;

        do {
                dst->hi = src->hi;      // read high bits
                dst->lo = src->lo;      // read low bits
        } while (dst->hi != src->hi);   // do over if high bits changed
}
```

Suppose src->hi changes between the two times it is read (in the body and in the condition). Then an interrupt happened at some point and we may or may not have a bogus dst like in the examples above. We assume the worst and try again.

On the other hand, suppose src->hi is the same before and after the read of src->lo. Then perhaps no interrupt happened. In this case, the copy is good, so we can stop. But perhaps an the interrupt occurred that changed only the bottom 32 bits. If the read of src->lo happened before the interrupt, then dst has the pre-interrupt value. If the read of src->lo happened after the interrupt, then dst has the post-interrupt value. Either way, the read executed atomically, so we can stop.

Exercise 4: the keyboard interface

We're going to write a shell, so we need a way to type at it. bochs has been displaying output we write to the printer port, but there is no good way to give it input. Instead, we'll use the X11-based interface and use CGA output and a

keyboard driver. We've written the keyboard driver for you in `kern/console.c`, but you need to attach it to the rest of the system.

> **Exercise 8.** In your `kern/trap.c`, call `kbd_intr` to handle trap `IRQ_OFFSET+IRQ_KBD`.

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

Test your code by running `gmake xrun-testkbd` and type a few lines. The system should echo your lines back to you as you finish them. Make sure you type into the X window `bochs` brings up, not the console.

Exercise 5: the shell

Run `gmake xrun-icode`. This will run your kernel inside the X11 Bochs starting `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. Run `ls`.

> **Exercise 9.** The shell can only run simple commands. It has no redirection or pipes. It is your job to add these. Flesh out `user/sh.c`.

Once your shell is working, 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 `printf` 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, "...", ...)`. See `user/ls.c` for examples.

Run `gmake 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`.

Challenge! Add more features to the shell. Some possibilities include:

- backgrounding commands (`ls &`)
- multiple commands per line (`ls; echo hi`)
- command grouping (`(ls; echo hi) | cat > out`)
- environment variable expansion (`echo $hello`)
- quoting (`echo "a | b"`)
- command-line history and/or editing
- tab completion
- directories, cd, and a PATH for command-lookup.
- file creation
- ctl-c to kill the running environment

but feel free to do something not on this list. Be creative.

Challenge! There is a bug in our disk file implementation related to multiple programs writing to the same file descriptor. Suppose they are properly sequenced to avoid simultaneous writes (for example, running "`(ls; ls; ls; ls) >file`" would be properly sequenced since there's only one writer at a time). Even then, this is likely to cause a page fault in one of the `ls` instances during a write. Identify the reason and fix this.

This ends the lab. As usual, you can grade your submission with `gmake grade` and hand it in with `gmake handin`. You only need to handin *non-challenge* portions of the assignment by December 6th. By December 10th and 11th, you should be prepared to show off your challenge to the TAs and then to the class.

Version: $Revision: 1.5 $. Last modified: $Date: 2007/12/04 06:00:06 $