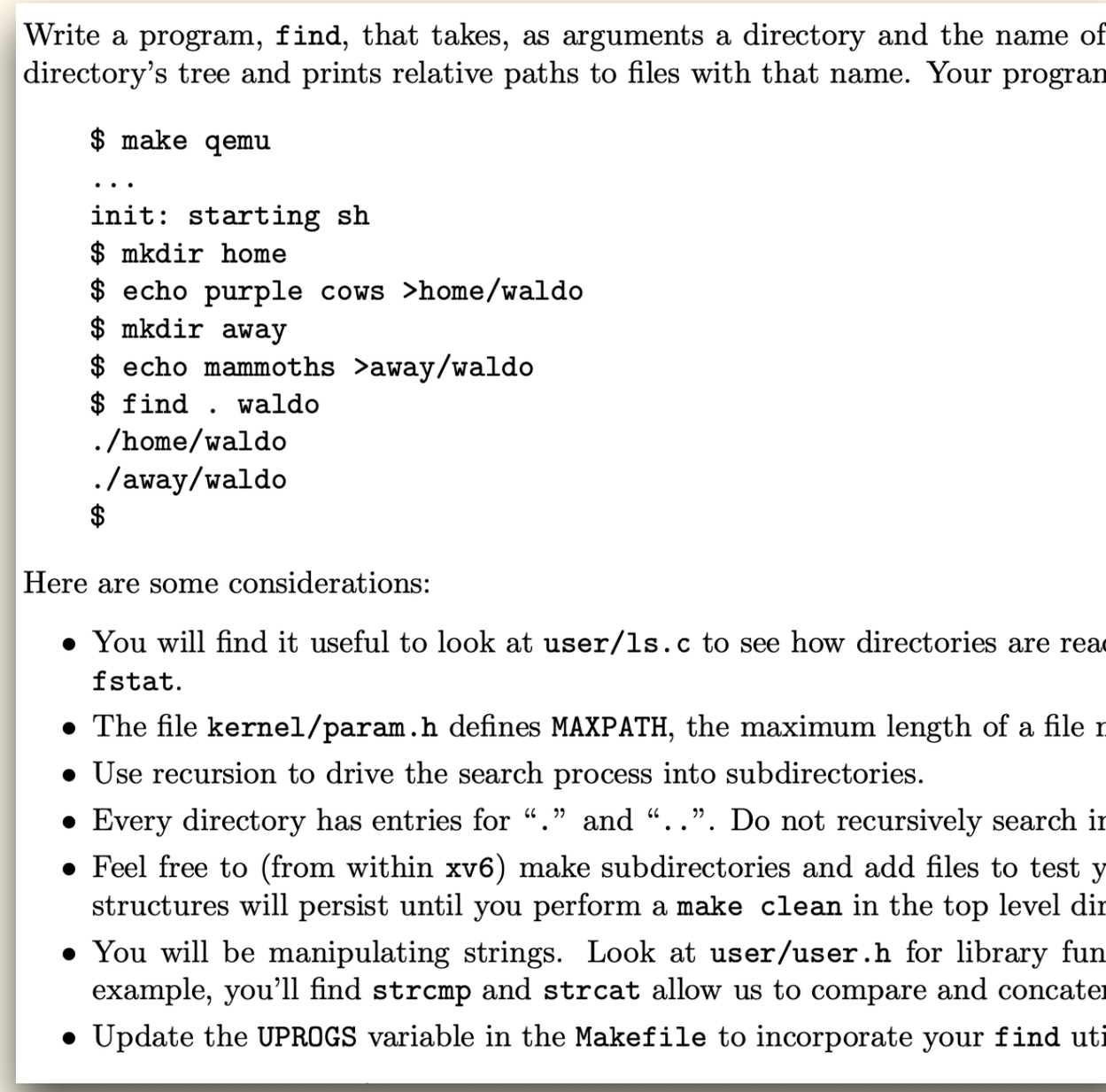
Working with xv6

Computer Science 432 - Lecture 4 - Duane Bailey

February 14, 2022

Announcements

- Everything in-person this week!
  - Lectures in Wach. 114
  - Labs in Ward Lab (TBL301)
  - Reviews in Knuth (TCL312b)
- Lab 1 due today, before lab
  - Questions?
- Lab 2 out today, due in a week
  - Implement: `sleep`, `find`, `xargs`



Write a program, **find**, that takes, as arguments a directory and the name of directory's tree and prints relative paths to files with that name. Your program

```
$ make qemu
...
init: starting sh
$ mkdir home
$ echo purple cows >home/waldo
$ mkdir away
$ echo mammoths >away/waldo
$ find . waldo
./home/waldo
./away/waldo
$
```

Here are some considerations:

- You will find it useful to look at **user/ls.c** to see how directories are read **fstat**.
- The file **kernel/param.h** defines **MAXPATH**, the maximum length of a file n
- Use recursion to drive the search process into subdirectories.
- Every directory has entries for "." and "..". Do not recursively search in
- Feel free to (from within **xv6**) make subdirectories and add files to test y structures will persist until you perform a **make clean** in the top level dir
- You will be manipulating strings. Look at **user/user.h** for library fun example, you'll find **strcmp** and **strcat** allow us to compare and concater
- Update the **UPROGS** variable in the **Makefile** to incorporate your **find** uti

Kernel Entrypoints

Processes:

I/O:

Filesystem:

System call	Description
int fork()	Create a process, return child's PID.
int exit(int status)	Terminate the current process; status reported to wait(). No return.
int wait(int *status)	Wait for a child to exit; exit status in *status; returns child PID.
int kill(int pid)	Terminate process PID. Returns 0, or -1 for error.
int getpid()	Return the current process's PID.
int sleep(int n)	Pause for n clock ticks.
int exec(char *file, char *argv[])	Load a file and execute it with arguments; only returns if error.
char *sbrk(int n)	Grow process's memory by n bytes. Returns start of new memory.
int open(char *file, int flags)	Open a file; flags indicate read/write; returns an fd (file descriptor).
int write(int fd, char *buf, int n)	Write n bytes from buf to file descriptor fd; returns n.
int read(int fd, char *buf, int n)	Read n bytes into buf; returns number read; or 0 if end of file.
int close(int fd)	Release open file fd.
int dup(int fd)	Return a new file descriptor referring to the same file as fd.
int pipe(int p[])	Create a pipe, put read/write file descriptors in p[0] and p[1].
int chdir(char *dir)	Change the current directory.
int mkdir(char *dir)	Create a new directory.
int mknod(char *file, int, int)	Create a device file.
int fstat(int fd, struct stat *st)	Place info about an open file into *st.
int stat(char *file, struct stat *st)	Place info about a named file into *st.
int link(char *file1, char *file2)	Create another name (file2) for the file file1.
int unlink(char *file)	Remove a file.

Figure 1.2: Xv6 system calls. If not otherwise stated, these calls return 0 for no error, and -1 if there's an error.

System Calls for Processes — fork

- Fork system call - creates new process
  - When you call fork, a second "child" process is created
  - Child process is a *copy* of parent; child has a different process id.
  - Both processes return from fork…to the same caller…yikes!
    - In the child process, fork returns 0.
    - In the parent process, fork returns pid of child
  - As with all syscalls, fork will return a negative value on error.
- See user/sh.c, for example.

```
pid = fork();
if (pid == 0) {
   // child code…
} else if (pid > 0) {
   // parent code…
} else {
   fprintf(2, "Fork error.\n");
}
```

System Calls for Processes — exit

- exit system call - destroy process and return resources
  - The exit call is used to tell the kernel the process is to be ended
  - Exit takes a status code.
    - 0 indicates a normal, successful exit.
    - Non-zero values indicate an error.
- The exit syscall does not return.
- In xv6, we use exit at the end of main (other systems: different).

```
pid = fork();
if (pid == 0) {
   // child code…
} else if (pid > 0) {
   // parent code…
} else {
   fprintf(2, “Fork error.\n”);
   exit(1); // exit with error
}
exit(0); // normal exit
```

System Calls for Processes — wait

- wait system call - wait for a child process to exit
  - We use the wait system call to wait for a child process to exit.
  - Returns the process id of the child that returns.
  - You can pass a pointer to an integer to receive the child's exit status
  - If you pass a null pointer ((int*)0), no status will be returned.
- In other versions of unix there's a variety of wait routines.

```
pid = fork();
if (pid == 0) {
   // child code…
   exit(status); // status is local to child
} else if (pid > 0) {
   // parent code…
   wait(&status); // parent process captures status
}
```

System Calls for Processes

- The kill(pid) system call externally terminates a process by pid
  - We use the kill system call to terminate a child process.
  - The pid identifies the child process.
  - Returns 0, or -1 on error.
  - Typically you can only kill child processes.

- The getpid() system call allows a process to determine its own PID
  - The first process (init) has process id 1.
  - Process ids are assigned incrementally, reflecting creation order

- The sleep(length) system call suspends a process for length “ticks”

- The sbrk(size) system call is used to extend memory high water mark.
  - Returns pointer to new memory.
  - Used by heaps.

Executing Code — exec

- The exec(prog, argv) system call *replaces* process with prog
  - The prog string is a path, identifying an executable.
  - The argv vector is a vector of arguments
  - The first entry is typically the name of the program.
  - The vector is null-terminated.
- There are *many* forms of exec in other implementations.

```
pid = fork();
if (pid == 0) {
   char *args[3];
   args[0] = “echo”; args[1] = “hello”; args[2] = 0;
   exec(args[0], args);  // only returns if error
   exit(1);  // if we get to exit, it’s an error
} else if (pid > 0) {
   // parent code…
}
```

I/O System Calls

- Unix processes maintain a list of file descriptors, small integers that refer, indirectly, to targets for input and output.
- By convention:
  - Descriptor 0 is “standard input” (typically a keyboard)
  - Descriptor 1 is “standard output” (typically a monitor)
  - Descriptor 2 is “standard error” (typically the same monitor as 1)
- When processes are created, these first three descriptors are assigned
- Children inherit the descriptors of parents.
- While exec replaces the process with new code, file descriptors are preserved.

I/O System Calls — open, close

- The open(path, access) and close(fd) system calls allow you to control access to files
  - If open is successful, it returns the smallest unused file descriptor.
  - The mode of access is determined by the access parameter
    - Read-only, write-only, read/write, create, and truncate
      See kernel/fcntl.h.
- The close syscall flushes the file buffers and marks descriptor free
  - We typically close(0) and then immediately open.
    Result: input redirection. Same with 1 (stdout), 2 (stderr).

```
char *argv[2];    // code to run input.txt through child cat
argv[0] = “cat";
argv[1] = 0;
if(fork() == 0) {
      close(0);
      open("input.txt", O_RDONLY);
      exec("cat", argv);
}
```

I/O System Calls — read, write

- The read(fd, ptr, n) reads up to n bytes from fd into memory at ptr
  - The number of bytes actually read is returned.
  - At end-of-file, read returns 0.

- The write(fd, ptr, n) writes up to n bytes to fd from memory at ptr
  - The number of bytes actually written is returned.
  - Returns fewer than n if there was an error.

```
// code typical of ‘cat’; see user/cat.c
char buffer[100];
int number;

while ((number = read(0, buffer, 100))) {
   write(1, buffer, number) < number);
}
```

I/O System Calls — dup

- The dup(fd) system call creates a second reference to a file
  - The new descriptor assigned is the first free descriptor
  - We typically call close(fd) after the dup call.
  - Useful in manipulating pipes and redirection.

```
// code typical for building standard error
close(2); // old standard error closed
dup(1);   // standard output duplicated to fd 2.
```

I/O System Calls — pipe

- The pipe(int p[2]) system call creates a memory buffer, a pipe.
  - Returns negative value on error.
  - Allocates a memory-based buffer for reading & writing.
  - You can read from the buffer using p[0]
  - You write to the buffer using p[1].
  - Pipes are often shared between processes; with each process exclusively using one end and closing the other. See user/sh.c.

```
int pid = fork();
int p[2];
pipe(p);
if (pid == 0) { // child will read
from pipe
   close(0);
   dup(p[0]);   // standard in
   from p[0]
   close(p[0]);
   close(p[1]);
  // process input from 0
}
else if (pid > 0) {
   close(1);
   dup(p[1]);
   close(p[0]);
   close(p[1]);
   // send data to child via 1
}
```

File Status Calls — fstat, stat

- The fstat(fd, struct stat *st) and stat(char *path, struct stat *st) calls get meta-data about file descriptors and files by name
  - struct stat is described in kernel/stat.h
  - Describes type of file, size, and location on device
  - Most common use is to identify type of file — file, directory, or device

- When you open directories, the file consists of a list of directory entries
  - struct dirent is described in kernel/fs.h
  - one file is the file name
  - we typically call stat on a path augmented by this file name.

- See user/ls.c