Computer Science 322
Operating Systems
Mount Holyoke College
Spring 2010

Topic Notes: Unix Systems Programming

Quote: UNIX system calls, reading about those can be about as interesting as reading the phone book... – George Williams, 3/12/91

We will consider several aspects of Unix systems programming, focusing first on those things you will need for the shell project.

Error checking/reporting

Most Unix system calls may fail for a variety of reasons. You should always check the return value of system calls that may fail. The reason for a failure in the `errno` variable. A list of errors can be found in `intro(2)`.

The system calls `perror(3)` and `strerror(3)` allow you to print out (hopefully) meaningful error messages when you detect a failed system call.

See Example:
`/home/jteresco/shared/cs322/examples/perror`

With Unix system calls, there are a lot of good reasons that something can fail. It's worth your trouble to check these return conditions and print meaningful error messages.

Process Management

You need to use a number of Unix system calls related to process management to implement the shell. We have seen a few of these:

`getpid()` – get current process ID

`getppid()` – get parent's process ID

`fork()` – duplicate process. Child is a copy of the parent - in execution at the same point, the statement after the return from `fork()`.

The return value indicates if you are child or parent. 0 is child, > 0 means parent, -1 means failure (limit reached, permission denied)

Example:

```
pid=fork();
if (pid) {
  parent stuff;
```

```
}
else {
  child stuff;
}
```

`exit()` – terminate a process. If it's a child, it waits for its parent to accept its return code. If this doesn't happen, the child is called a "zombie" process.

To avoid this – call `wait()` (or `waitpid()`) from the parent – parent stops and waits for the child to terminate (call `exit()` or `_exit()`).

Returns PID of child, and in its argument, the status includes the value the child passed to `exit()`.

Recall example from earlier:

See Example:
`/home/jteresco/shared/cs322/examples/forking`

Be careful not to confuse this `wait()` *with the* `wait()` *operation on semaphores that we'll see later!*

Running a new program – `exec` calls

`fork()` lets you have two copies of a process – the same process. Sometimes this is just what you want, but what if you want to start a new process running some other program.

To create processes that do other stuff, the `fork()` is followed by one of these "exec" calls, in the child process:

`execl()` – exec a process with list of arguments

`execv()` – exec a process with args specified in an array

`execlp()` – list, but search the existing path for the program.

`execvp()` – array, but search the existing path for the program.

`execvP()` – array, but specify a search path for the program.

The man pages have details.

The related `vfork()` system call is often more appropriate when the child process will be doing an `exec()` immediately. It doesn't duplicate all of the memory for the parent process. Beware: this may cause you trouble in the shell if you use it, since the parent is usually suspended until the child `exit`s or calls an `exec`.

See Example:
`/home/jteresco/shared/cs322/examples/exec`

1. Start by looking at `exec`:
    - `execlp` parameters: program to run, arguments

- this is a varargs function call – we can send any number of parameters
- see what happens if we exec something not in the `PATH` (try it)
- can specify fully-qualified path

2. Look at `procinfo` program:
   - just print some information about the process
   - pid, arguments (including one beyond the last)
3. Use `execprocinfo` to execute it. Note that `argv[0]` isn't always the command that was executed – just whatever was passed as the second parameter to the `exec`.
4. Try `exec2`:
   - sometimes an array of parameters is more convenient: `execvp()`
5. Try `exec2nonull`:
   - what if we forget the `NULL`?
   - what if we have a `NULL`, but not right away?
6. Try `execwithargs`:
   - note that it works as expected
   - note the use of the `argv` as passed in (except `argv[0]` – that would be a problem – try it).
   - use `execwithargs` to exec itself
   - and itself followed by something else
   - have it exec something that's not in the path

Signals

Unix processes can communicate by sending each other *signals*.

Type `kill -l` at your favorite Unix prompt to see the names of the signals it supports.

`kill -SIGNAL pid` will send signal `SIGNAL` to a process `pid`:

```
-> sleep 60&
[1] 96132
-> kill -TERM %1
[1]+  Terminated              sleep 60
-> sleep 60&
[1] 96133
```

```
-> kill -STOP %1
[1]+  Stopped                 sleep 60
-> kill -CONT %1
-> jobs
[1]+  Running                 sleep 60 &
->
[1]+  Done                    sleep 60
->
```

Every process has *signal handlers* that are used to respond to signals sent to the process. Basically, it's a function that gets called *asynchronously* when a signal is received.

A *default signal handler* is installed when a process begins.

Two system calls are used to send and catch signals:

`signal()` – replace default handler. Lets you *trap* many signals and handle them appropriately.

Be careful not to confuse this `signal()` *with the* `signal()` *operation on semaphores when we get to that topic!*

See Example:
`/home/jteresco/shared/cs322/examples/signals`

Example: A compute-bound process that "wakes up" every 5 seconds to report on its progress. See: `sigalrm-example.c`

Note the use of `setitimer(2)`.

We can ignore a signal completely by setting its handler to `SIG_IGN`, and restore the default handler with `SIG_DFL`.

Enhanced example: `sigalrm-example2.c`

A process can also send signals with `kill()`. Don't let the name fool you, you can send any signal with `kill()`, not just `SIGKILL`.

Note that `SIGTERM`'s handler sends the process a `SIGINT`.

Note that we do not trap `SIGSTOP` and `SIGCONT`, we can try these out.

Note that we do not trap other signals, like `SIGUSR1`.

Note: `SIGCHLD` will be useful for your shell projects.

Low-level File Operations

You may (or may not) be familiar with the C standard file I/O routines defined in `stdio.h`, such as `fopen()`, `fscanf()`, `fprintf()`, and `fclose()`. These provide relatively "high-level" access to files in that you deal with data types rather than a low-level stream of bytes.

Underneath the stdio functions, you will find those low-level operations: `open()`, `close()`,

`read()`, `write()`.

The man pages describe these in great detail.

See Example:
`/home/jteresco/shared/cs322/examples/everyother.c`

Note that there are three automatic file descriptors:

```
0    stdin
1   stdout
2   stderr
```

These operate only on raw data and pay no attention to data type or any formatting.

Pipes

Processes may wish to send data streams to each other. Unix *pipes* are one way to achieve this. You've almost certainly used Unix pipes at the command line. You can also use them in programs.

An unnamed pipe can be created using the

```
int pipe(int fd[]);
```

system call. `fd` is an array of two `int` values. These are file descriptors, very similar to the file descriptors used for file I/O using `open()`, `read()`, and `write()`.

`fd[0]` is the "read end" and `fd[1]` is the "write end". 0 return means success. -1 means failure.

`read()` and `write()` again operate only on basic streams of bytes – any structure must be added.

See Example:
`/home/jteresco/shared/cs322/examples/pipes`

An example of communication between two processes, a parent and its child created by `fork()`, communicating via an unnamed pipe is in `pipe1.c`

This required the shared values of `fd`. This is fine when you create your pipe just before a `fork()`, but what if we have two processes already in existence that wish to communicate through a pipe?

We can create a *named pipe* with `mkfifo` (command or system call).

We can make our simple example using the named pipe: `pipe2.c`

We can make an example that's a little more interesting, where two independent processes communicate through a pipe: `pipeprocs.c`

Duplicating file descriptors

We can use the `dup2()` system call to reroute things that were going to one file descriptor into another file descriptor. This is how your I/O redirection and pipes will work in the shell.

See example `execredir.c`

Note that we don't close the file here and in fact are not given an opportunity to do so.

We have seen that you can also obtain file descriptors from `open()`, `pipe()`

Note that the fd's at the ends of a pipe can be passed to `dup2()` – this will be useful – set the output of one process to be the input of another through a pipe.