Chapter 7

IPC - I

Chapter Objectives

To understand concepts of IPC used over the duration of the course.

Objectives

For this chapter, the following are the objectives:
- pipes,
- signals,

Slide #7-1

Notes

In this chapter, we examine asynchronous events and Tools.

The objective of this chapter is to provide an understanding of concepts on Tools that are used over the duration of the course.

Chapter Organization

1. **Objective**: Introduction to
   - pipes,
   - signals

2. **Description**: This an introduction to IPC mechanisms in Linux.
   This chapter provides an introduction to the concepts used in course.

3. **Concepts Covered in Chapter**:
   - Introduction to various tools.
   - LINUX Notes

4. **Prior Knowledge**:
   same as Chapter #1

5. **Teaching & Learning Strategy**:
   Discussion questions are,
   - What are these tools for?

6. **Teaching Format**:
   Theory + Homework Assignments

7. **Study Time**: 120 Minutes (Lecture & Theory)
   + ~45 minutes (Homework Assignments)

8. **Assessment**: Group Homework Assignments

9. **Homework Eval**: Group

10. **Chapter References**:

Introduction to IPC

IPC

Various Options
- Pipes and Named Pipes,
- SYS V IPC (`shm, sem and msg`)
- Sockets (TCP and UDP)

Slide #7-2

Notes

IPC

1. Allows processes to communicate.
2. Using a file IO based approach - write and reads from a file.

   Alternatively,

   2.1. use file descriptors for syscalls such as read()
   - pipes, sockets.

   2.2. use special functions
   - shm, sem , msg, signals

IPC Considerations

- Related Processes
- Unrelated Processes on same system
- Unrelated Processes on different machines

Slide #7-3

Notes

IPC

1. Related Processes share a common ancestor.
   e.g. pipes
   ```
   ls | more
   ```
2. On the same system, IPC between unrelated processes can use memory or files, named pipes, signals, semaphores, shared memory, and message queues.
3. Across multiple systems, socket communications can be used to facilitate IPC

Syscall pipe()

pipes

- Uses file descriptors
- fifo

pipe ( )

- `int pipe (int fd [2])`.
- Provides a one-way communication path.
- between related processes.

Notes

Basic Concepts – pipe

1. pipe() uses a pair of file descriptors:
   1.1. read, and
   1.2. write

   ends for the pipe.
2. A process calls pipe().
   It creates read and write ends for the pipe.
   2.1. calls fork().
   2.2. parent and child agree on who the “talker” is and who is the “listener”, and have a communication channel established.

Notes

1. A pipe consists of 2 file descriptions which are connected to a kernel buffer.
   1.1. One file descriptor is used for the read end of the pipe. the other is used for the write end of the pipe.

2. pipe()
   2.1 The system called `pipe()` sets up the buffer and returns 2 file descriptors.
   2.2 `int (int fd[2]);`
   `fd[0]` is the read end… similar to `stdin...fd 0`
   `fd[1]` is the write end… similar to `stdout...fd 1`

Code Example // `pipe()`

```
1    // Chap 5_1.c
2    void main()
3    {
4       int n =0;
5       int fd[2];
6       char buf[1024];
7
8       pipe(fd);
9
10      write(fd[1], “Hello World\n”, 12);
11      n = read(fd[0], buf, sizeof(buf));
12
13      printf(“%d == read(%d, %#010x, %d) => ‘%’ \n”,
14           buf, sizeof(buf), buf);
15   }
16
```

pipe ( ) Conventions … 1

- Call `pipe()`
- Call `fork()`
- close() IN parent & child .. who's listener & who's talker;
- And, unused endpoints.

Pipe ( ) Conventions … 2

	`read()` from	`write()` to
empty pipe	blocks	√
full pipe	√	blocks

1. A process calls pipe().
   It creates read and write ends for the pipe.
   1.1. The process now calls fork().
   1.2. The parent and child agree on who the "talker" is and who is the "listener". Then the parent and child, each close the appropriate read or write end, and thus establish a pipe between two processes.

Notes:

1. read from an empty pipe … block
2. writes to full pipe … block
3. Normal operation when
   a. `read()` on `pipe()` with data
   b. `write()` to non-full `pipe()`

Pipe ( ) Conventions … 3

	All opposite ends closed	At least one end of opposite type is open
`read(0`	Returns EOF	√
`write()`	• Return -1 • Sends `SIGPIPE` to process	√

Slide #7-7

Notes

1. reads()

   1.1. `read()` with at least one `write()` end open, will return the data, until no more data is available to be consumed. Subsequent calls to `read()` will return 0.

   1.2. `read()` with all `write()` ends closed will return EOF.

2. writes()

   2.1 writes with at least one `read()` end open will be successful, until such time that the pipe does not become full, then `write()` blocks.

   2.2 It is an error for `write()`s to a pipe, whose read ends are `close()'`d. This will cause `errno` to be set. Also, the signal `SIGPIPE` will be sent to the process.

Named pipes

named pipes

- fifo
- uses `mkfifo` OS command
- allows IPC for unrelated processes
- same semantics as `pipe()`

Slide #7-8

Notes

1. uses mkfifo OS command
   1.1. creates a fixed length file
   1.2. In the outputs of 'ls –l', the pipe is displayed with a 'p' as the file type – as part of the file permissions' bits.
2. uses a fixed-length size
   2.1. readers can block
   2.2. writers can block

   depending on the producer-consumer sematics.
3. unrelated processes can communicate using IPC.

Code Example // named `pipe()`

```
$ cat ./producer.sh
#!/bin/bash -a

i=1
I=9

while [ "${i}" -le "${I}" ]
do
         DATE="`date +%m%d%y.%H%M%S`";
         echo "Hello from $$ ( @$DATE )";
        (( i= i+1 ))
        sleep 5
done
```

Code Example // named `pipe()`

```
$ cat ./consumer.sh
#!/bin/bash -a

i=1
I=9

while [ "${i}" -le "${I}" ]
do
         sleep 5
         read input_line
         echo "just got $input_line"
        (( i=i+1 ))
done
```

Code Example // named `pipe()`

```
$ cat named_pipe.sh
#!/bin/bash -a

mkfifo /tmp/mkfifo

ls -l /tmp/mkfifo

./producer.sh > /tmp/mkfifo &
./consumer.sh < /tmp/mkfifo &
```

mkfifo()

- mkfifo()
- programmatic interface to create named pipe.

Slide #7-9

Notes

1. used by mkfifo OS command
2. usage:

```
#include <sys/types.h>
#include <sys/stat.h>
```

3. internally uses mknod() call

signals

signals

- Can be used for IPC.
- Two signals SIGUSR1 and SIGUSR2
- Conventions established for appropriate “handling”

- Typically used for synchronizing processes

Slide #7-10

Notes

1. Processes can notify other processes, using a signal.
2. Signals SIGUSR1 and SigUSR2 are reserved for this purpose.

Signals – passing data

- Signals generally, do not pass data.
- However, can use SA_SIGINFO flag in sigaction()
- Uses siginfo_t data structure to pass data to handler.
- Use sigqueue() to send signal.

Slide #7-11

Notes

1. use sigaction() to set the SA_SIGINFO flag.
2. use sigqueue() instead of kill() to send signal

sigqueue()

- Alternative to kill
- Allows multiple occurrences of signal to be queued.
- Allows specific values to be passed to handler

Slide #7-12

Notes

1. alternative to kill().
2. usage:

```
#include <signal.h>

int sigqueue (pid_t pid, int sig, const union sigval sv);

union sigval {
   int sival_int;
   void * sival_ptr
}
```

where sv can be either an int value or a pointer - used when a process sends a signal to itself.

Signal handler

Without SA_SIGINFO,
- Handler receives one parameter

With SA_SIGINFO
- Handler receives three parameters

Slide #7-13

Notes

1. alternative to kill().
2. usage:

```
myhandler(int sig, siginfo_t *siginfo, void *vp);
```

where, the second argument siginfo, contains the value to be passed via sigqueue().

and, the third argument points to a structure containing information about the process context.

http://developer.apple.com/documentation/Darwin/Reference/ManPages/man2/sigaction.2.html

Links

http://developer.apple.com/documentation/Darwin/Reference/ManPages/man2/sigaction.2.html

Chapter 7
IPC - I

Assignment Questions

Questions:

7.1 Modify “mysh” (from chapter 6) to include to create a parent shell that will “pass” the commands to the “child” shell.

Assignment Hints

http://www.redhat.com/docs/books/max-rpm/max-rpm-html/index.html

Useful Links

http://www.justlinux.com/nhf/Miscellaneous/Creating_Your_Man_Page.html

http://www.unixreview.com/documents/s=8925/ur0312i/

http://tldp.org/HOWTO/Man-Page/index.html

http://en.wikipedia.org/wiki/RPM_Package_Manager

https://pmc.ucsc.edu/~dmk/notes/RPMs/Creating_RPMs.html

http://www.ibm.com/developerworks/library/l-rpm1/

http://genetikayos.com/code/repos/rpm-tutorial/trunk/rpm-tutorial.html

http://docs.python.org/dist/creating-rpms.html

http://www.amath.washington.edu/~lf/tutorials/autoconf/toolsmanual.html#SEC21

http://www.mtsu.edu/~csdept/FacilitiesAndResources/make.htm

http://www.cs.colorado.edu/~kena/classes/3308/f04/lectures/

automake

autoconf

http://en.wikipedia.org/wiki/GDB

http://sourceware.org/gdb/

http://www.gnu.org/software/ddd/

gprof

http://sourceware.org/binutils/docs-2.17/gprof/index.html

http://sam.zoy.org/writings/programming/gprof.html