UNIX System Programming
Lecture 13: Pipes

• Outline
  - UNIX IPC Methods
  - An Introduction to (unnamed) Pipes
  - IPC via Pipes
  - Named Pipes (FIFOs)

• Reference
  - BLP: Chapter 13
  - man pages
    • popen, pipe, read, write, fcntl, dup, dup2, mkfifo

Lecture 13: Pipes
UNIX IPC

• UNIX allows several methods of interprocess communication (IPC). Here are ten:
  1) shared file: One process positions a file pointer (using lseek()) a second process reads the pointer position (also lseek()). The position is the (integer) data to be communicated. (This is rarely used. The method is restricted to related processes. Synchronization is also required.)
  2) signals: Signals can be used for notifications or synchronization, but not transfer data.

Lecture 13: Pipes
UNIX IPC

3) process tracing: This is commonly used by debuggers, but is too complicated for normal IPC.
4) files: Passing data via a file is the most common form of IPC, but difficult to use with concurrent processes. Files may get huge when processes run for long periods.
5) pipes (unnamed): They solve the synchronization problem that exists with files. Processes must be _________. Reads and writes may not be atomic on all systems. May be slow.

Lecture 13: Pipes
UNIX IPC

6) FIFOs (named pipes): Similar to unnamed pipes but they allow communication between unrelated processes. They may still be too slow.
7) ________: They are used for process synch. or exclusive access to resources.
8) message queues: Allow small messages to be synchronously transferred between processes.
9) shared memory: The fastest method of IPC. Semaphores or messages may be used for synch.
10) sockets: Used for network comm between processes on different (or the same) hosts.

Lecture 13: Pipes
Introduction to Pipes

• We will discuss IPC via ______ pipes first. You are familiar with pipes as a shell facility:
  $ who | sort | pr
Here three processes are connected via two pipes. The standard output of the process on the left of the $ is connected to the standard input of the process on the right.
• Pipes use the standard UNIX read/write calls for passing data between processes.

Lecture 13: Pipes
The popen() Routine

• The popen() function is built on the lower level pipe() routine. It returns a stdio FILE * stream (so stdio routines must be used):
  FILE *rf = popen("ls -l", "r");
  while(fgets(buf,sizeof(buf),rf)!=NULL) {
    /* process lines */
  }
• The popen() FILE stream is ___________. popen() is expensive in terms of resources.
  (It launches a shell to run the command.)
Lecture 13: Pipes
The pipe() Routine

- pipe() does not require that a shell be started and also gives us more control over the reading and writing of data.
- pipe() returns two file descriptors in an integer. We read from one and write to the other.

Lecture 13: Pipes
Pipe Characteristics

- Several of the system routines act slightly differently when operating on a file descriptor associated with a pipe instead of a file descriptor associated with a file.
- write() will normally close on a full pipe. Pipes are at least 4096 bytes on all UNIX systems. There are no partial writes. To get an end-of-file on a read, you must close the write end of the pipe. (pipe_xmpl2.cpp demos a blocked write.)

Lecture 13: Pipes
IPC via a Pipe

- pipe() is used for IPC between processes. (It is not possible for an unrelated process to get access to a system file descriptor owned by another process, but children inherit all file descriptors from their parents.) (See pipe_xmpl3.cpp)
- A process normally uses only one end of the pipe. Always close the end you are not using!
- Pipes are unidirectional, for bidirectional process comm. you need two pipes.

Lecture 13: Pipes
Pipes to exec()'d Programs

- We rely on exec()'d programs to open file descriptors to allow IPC via pipes.

```c
int pfd[2];
if(pipe(pfd) == -1)
    error(strerror(errno));
// read from pfd[0]/write to pfd[1]
string msg = "Hello world!";
// we can write a short message to
// the pipe and not worry about the
// write blocking and process deadlock
write(pfd[1], msg.c_str(), msg.length());
uint count = read(pfd[0], buf, sizeof(buf));
buffer[count] = '\0';
cout << buffer << endl;
```
Lecture 13: Pipes
Pipes to exec()’d Programs

- Normally we exec a program in the child.
- File descriptors stay open across an _______ (unless fcntl() is used to set the close-on-exec flag), but the exec’d process will not have access to the array containing the file descriptor numbers.
- One method used to get around this problem is to pass the descriptors as arguments to the program. (See wtr_xmpl1.cpp and rdr_xmpl1.cpp)

Lecture 13: Pipes
dup() and dup2() Routines

- Most programs read from standard input and write to standard output. How do we communicate with such a program via a pipe? We use dup() or dup2().
- The dup(olefd) routine ______________ an existing file descriptor. The new file descriptor returned by dup() is guaranteed to be the lowest numbered available.
  dup2(olefd, newfd) duplicates oldfd over newfd closing newfd first if necessary.

Lecture 13: Pipes
dup() and dup2() Routines

```c
// See wtr_xmpl2.cpp and rdr_xmpl2.cpp
// See wtr_xmpl3.cpp for dup2() example.
pipe(pfd);
if(fork() == 0) {
    close(0);       // so 0 is available
dup(pfd[0]);     // will dup over 0
    close(pfd[0]);
    close(pfd[1]);
    execl("/bin/echo", "echo", "Hello", "world", (char *)0);
    exit(0);
}
// close read end of pipe
n = write(pfd[1], msg, bufsize);
close(pfd[1]);
```

Lecture 13: Pipes
Named Pipes

- Named pipes (_______) are special types of “files” that can be used for communication between unrelated processes. (They are not real files and take up no disk space.)
- A FIFO combines features of files and pipes. It has a name and may be opened for reading or writing by any process with appropriate permissions. Once opened, a FIFO acts more like a pipe. Data, once read, can’t be read again, nor does lseek() work.

Lecture 13: Pipes
Named Pipes

- FIFOs can be created from the command line using the ________ command (or mknod):
  $ mknodo myfifo
  $ ls -l myfifo
  prw-rw-r-- 1 ar63 ar63 0 Oct 22 17:00 myfifo
- The FIFO can be used to pass data between programs:
  $ cat /etc/passwd > myfifo &
  $ cat myfifo
  $ gnuplot -persist < myfifo &
  $ echo "set nokey; plot sin(x)" > myfifo

Lecture 13: Pipes
mkfifo() Routine

- From within a program a FIFO can be created using mkfifo() (or mknod()):
  mkfifo("/tmp/myfifo", 0777);
  The second argument is the desired FIFO _________. mkfifo() returns -1 on error.
- The FIFO will exist until deleted (with rm at the command line or by calling the unlink() routine from within a program).
Lecture 13: Pipes
FIFO Characteristics

- An open() for read-only on a FIFO will block until another process opens the FIFO for writing. An open() for write-only will block until a reader opens the FIFO. This allows _________ before data transmission. (The open() routine will not block if the O_NONBLOCK flag is used in the open().)
- read() will block if no data is present.
- write() will block on a full FIFO. (But not if O_NONBLOCK is used.)

Lecture 13: Pipes
Example Writer Code

```c
// Error checking omitted for brevity
// See fifo wtr.cpp and server.cpp
const char *FIFO_NAME = "/tmp/db_server";

// Check for existence
if(access(FIFO_NAME, F_OK) == -1) {
    // doesn't exist, create it
    mkfifo(FIFO_NAME, 0777);
}
// block on open until reader opens too
wfd = open(FIFO_NAME, O_WRONLY);
count = write(wfd, buffer, length);

close(wfd);
unlink(FIFO_NAME);
```

Lecture 13: Pipes
Example Reader Code

```c
// Error checking omitted for brevity
// See fifo rdr.cpp and client.cpp
// We wait on server to create fifo
while(true) {
    rfd = open(FIFO_NAME, O_RDWR);
    if(rfd == -1) {
        if(errno == ENOENT) {
            // doesn't exist, wait
            sleep(1); continue;
        } else { exit(1); }
    }
} // end while

count = read(rfd, buffer, length);
```

Lecture 13: Pipes
In Class Exercise

- Copy pipe_xmpl3.cpp and comment out the line in the child that closes the write end of the pipe. Compile and run the program. Why doesn't the child exit? (Be sure to kill the child manually using the kill command.)
- Modify pipe_xmpl3.cpp so that the parent reads /etc/passwd and passes all the data to the child via the pipe. The child should read from the pipe and write the data to standard output.