CSci 4061
Introduction to Operating Systems

IPC: Basics, Pipes
Today

- Directory wrap-up
- Communication/IPC
- Test in one week
Communication

• Abstraction: conduit for data exchange between two or more processes (or threads)

• Examples (same machine) - need better examples
  • Web browser + tabs
  • X-window server + X-window clients
  • Parallel program on multi-core CPU
IPC in Unix

• Pipes: most basic form of IPC in Unix
  • process-process
  • `ps -u jon | grep tcsh` // what happens?

• Pipe has a “read-end” (receive) and a “write-end” (send) : think of this actually as a
  • FIFO communication
    • (write A, write B, read->A, read->B)
  • single channel
IPC in Unix (cont’d)

• Pipe allows communication between a parent and child or related processes
CSci 4061
Introduction to Operating Systems

IPC: Pipes
Today

• Pipes
• Exam #1 discussion
Mistake Alert

Cannot maintain separate permissions on different hard-links

Reason: permissions are a property of the file (i.e. the i-node) itself, NOT the name

Suppose “bar” and “foo” link to the same file

chmod 0777 foo (both foo and bar are 0777)
chmod 0222 bar (both foo and bar are 0222)
#include <unistd.h>

int pipe (int ends[2]); // returns -1 on failure

ends is a 2-integer fd array that represents the ends of the pipe
ends[0] is the “read-end” (receive) and ends[1] is the “write-end” (send)

Integrated into filesystem
Link is “named” by the pipe but we do not name the reader/writer processes
How can pipe fail?
Pipes and FD

[show]
Simple pipe example: single process

#include <unistd.h>
#include <stdio.h>
char *msg1 = "hello, world #1";  
char *msg2 = "hello, world #2";  
void main () {
    char inbuf [MSGSIZE];  
    int ends[2];

    if (pipe(ends) == -1) {
        perror ("pipe error");  
        exit (1);
    }

...
Simple pipe example (cont’d)

// write (send) down pipe
write (ends[1], msg1, MSGSIZE);
write (ends[1], msg2, MSGSIZE);
// read (receive) from pipe
read (ends[0], inbuf, MSGSIZE);
fprintf (stderr, "\%s\n", inbuf);
read (ends[0], inbuf, MSGSIZE);
fprintf (stderr, "\%s\n", inbuf);
Output is:
    hello, world #1
    hello, world #2
Read and write

write (ends[1], msg, MSGSIZE);
read (ends[0], inbuf, MSGSIZE);

Read may not get everything but it “usually does” up to max (MSGSIZE, and pipe contents) blocks if pipe if empty
Pipe have finite size (e.g. 4K/8K)
write blocks if not enough space
why is there a limit?
void main () {
    char inbuf [MSGSIZE];
    int ends[2], j;
    pid_t pid;

    if (pipe(ends) == -1)  {
        perror ("pipe error");
        exit (1);
    }
}
pid = fork ();
if (pid == 0) { // child sends into pipe
    write (ends[1], msg1, MSGSIZE);
    write (ends[1], msg2, MSGSIZE);
}
else if (pid > 0) { // parent receives from pipe
    read (ends[0], inbuf, MSGSIZE);
    fprintf (stderr, "%s\n", inbuf);
    read (ends[0], inbuf, MSGSIZE);
    fprintf (stderr, "%s\n", inbuf);
    wait (NULL);
}

why does this work across processes?
Issues

• Potential problem
  • If both processes write into the pipe, what would happen?
  • Usually, one writes and other reads
Resolving

if (pid == 0) {               // child sends into pipe
    close (ends[0]);
    write (ends[1], msg1, MSGSIZE);
    write (ends[1], msg2, MSGSIZE);
}
else if (pid > 0) {           // parent receives from pipe
    close (ends[1]);
    read (ends[0], inbuf, MSGSIZE);
    fprintf (stderr, “%s\n”, inbuf);
    read (ends[0], inbuf, MSGSIZE);
    fprintf (stderr, “%s\n”, inbuf);
    wait (NULL);
}
Typical Pipe Use Case

• Near infinite stream of data from producer to consumer
  • consumer (reader) had better keep up with producer (writer)
  • why?
  • `cat * | wc`

• You cannot `fseek/seek` a pipe fd
More on pipes

• close write-end (no processes have pipe open for write) and pipe is empty:
  • read returns a 0 (0 is EOF)

• close read-end and write-end is open:
  • write kills the process!
  • “broken pipe”

• Pipes are limited to parent-child siblings, related process relationships must share fds
  • this has now changed in Linux
Example: Knock-Knock

- https://www.youtube.com/watch?v=ZL5fdJ1-iWg

- Write a knock-knock program using a parent (P) that tells jokes to their child (C)
Example: Knock-Knock

- Protocol: sequence of messages

P: w, r, w, r, ...
C: r, w, r, w, ...

P: “k-k”  “orange”
C: “w-t?”  “orange-who?”

“aren’t you glad this isn’t Java?”
Solution

Sol 1:

```c
char buf [maxBUF];
pipe (ends);
fork ();

// parent
write (ends[1], "k-k", …)
read (ends[0], buf, …);
write (ends[1], "orange", …);
read (ends [0], buf, …);
write (ends[1], "aren't …", …);
```

// child

```c
read (ends[0], buf, …);
write (ends[1], "w-t?", …);
read (ends [0], buf, …);
write (ends[1], "orange-who?", …);
read (ends [0], buf, …);
```

Issues?
Issues

• Pipe access not coordinated

• A process that writes can read what it just wrote in the pipe!

```c
// parent
write (ends[1], "k-k", ...)
read (ends[0], buf, ...);
```
Better one?

- Need two pipes
- One parent->child (parent writes, child reads)
- One child->parent (child writes, parent reads)
Solution

Sol 2:

```c
int P_C[2], C_P[2];  // w_r[2]
pipe (P_C);
pipe (C_P);
fork ();

// parent
close (P_C[0]);
close (C_P[1]);
write (P_C[1], ...)
read (C_P[0], ...);
...

// child
close (P_C[1]);
close (C_P[0]);
read (P_C[0], ...);
write (C_P[1], ...);
...
```
Takeaway Lesson

• Need two-way communication
• Most likely will need a pair of pipes
Non-blocking pipes: example

• Children may inform the parents of various events or ask for things to do ... BUT this is unpredictable ...

• Suppose we do blocking I/O (parent reads)?
**Non-blocking pipes**

- Default I/O behavior is blocking
- Non-blocking I/O can be handy
- Since pipe is a file ... can control attributes

```c
#include <fcntl.h>
int fcntl (int fd, int cmd, ...);
int ends[2], flags, nread;

pipe (ends);
flags = fcntl (fd, F_GETFL, 0);
fcntl (ends[0], F_SETFL, flags | O_NONBLOCK);
...
nread = read (ends[0], buf, size);

// if nothing to read, returns -1, errno set to EAGAIN
```
Sending Discrete “Data”

- Sending a message into a pipe

```c
typedef struct {
    int x;
    int y;
    char str[20];
} message_t;

message_t m1, m2;
int ends[2];
pipe (ends);

// send m1 into the pipe
write (ends[1], &m1, sizeof (message_t));

// pull data into m2 from the pipe
read (ends[0], &m2, sizeof (message_t));
```
Finally back to Pipes in the shell

- `ps -u jon | grep tcsh`
- How does the shell do it? Why is this an issue?

```c
pipe (ends);
if (childpid = fork ()) == 0) {
    dup2(ends[1], 1);
    // close ends[0] and ends[1]
    execl("/bin/ps", ...,);
    // when ps does a write (1, ...) where does I/O go?
}
else {
    dup2(ends[0], 0);
    // close ends[0] and ends[1]
    execl("/bin/grep", ...,);
    // when grep does a read (0, ...) where does I/O go?
}
```

[picture]--> draw a picture of the fd table
Exam

• Closed book and notes
• We’ll provide code cheat sheets
• Material through last Thursday’s lecture

How to study
  • Read the book
  • Look at notes!
  • Study lecture content
  • Write small programs
  • Form a study group
Topics

• Programs and Processes
  • fork/exec/exit/wait, C programs (basic pointers, flags, ...)
  • identities, crashes, zombie/orphans

• I/O
  • Low-level, high-level, redirection, semantics, binary, random, buffering, errors, control, formatting

• File systems
  • Files and directories
  • Links, i-nodes, permissions

• IPC
  • Basics, pipes
Exam Structure

• Focus is on system/library calls not C per-se

• Around 35% short answer
  • Why does Unix treat all devices as part of the file system?
  • What is random I/O and why is it useful?
  • Why does this simple code break?
  • How can fork fail?

• Around 65% code oriented
  • What does this code fragment do?
  • Write a function to do X
  • How would you modify lab #1 to add this feature?
Analyze Example

What does this program do?

```c
main () {
    int ends[2];
    char buf[10];
    int p1, p2;
    pipe (ends);
    p1 = fork ();
    if (p1) {
        p2 = fork ();
        if (p2) {
            wait ();
            wait ();
        }
    } else {
        read (ends[0], buf, 5);
        write (1, buf, 5);
    }
} else {
    write (ends[1], "hello", 5);
}
```
Coding Examples

• Implement this function using low-level I/O:

```c
char *fgets(char *s, int n, FILE *stream)
```

• Create a ring of N processes and have them pass a simple integer value around using pipes

```
# #
```

```
Hints

• You will have a programming question on processes and the use of pipes

• You will have a programming question on file meta-data

• I will post some more sample questions
Next Time: after the test

• More IPC
  • Message Passing
  • Shared Memory