CSci 4061
Introduction to Operating Systems

IPC: Basics, Pipes
Communication
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
Pipes

```
#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

```c
#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 \textit{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?
Simple pipe example: multi process

void main () {
    char inbuf [MSGSIZE];
    int ends[2], j;
    pid_t pid;

    if (pipe(ends) == -1) {
        perror ("pipe error");
        exit (1);
    }
}
Multi process (cont’d)

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
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);
}
New picture
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

• `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`
Example: Knock-Knock
Example: Knock-Knock

• Protocol: sequence of messages
P: w, r, w, r, ..
C: r, w, r, w, ...

P: “k-k” “orange” “aren’t you glad this isn’t Java?”
C: “w-t?” “orange-who?”
Solution

Sol 1:

pipe (ends);
fork ();

// parent
write (ends[0], “k-k”, …)
read (ends[0], buf, …);
write (ends[1], “orange”, …);
read (ends[0], buf, …);
write (ends[1], “aren’t …”, …);

// child
read (ends[0], buf, …);
write (ends[1], “w-t?”, …);
read (ends[0], buf, …);
write (ends[1], “orange-who?”, …);
read (ends[0], buf, …);

Issues?
Better one?
Takeaway Lesson
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?
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];

// 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));
```
Pipes in the shell

- ps -u jon | grep tcsh
- How does the shell do it?

```c
pipe (ends);
if (childpid = fork ()) == 0) {
    dup2(ends[1], 1);
    // close ends[0] and ends[1]
    execl("/bin/ps", ….);
}
else {
    dup2(ends[0], 0);
    // close ends[0] and ends[1]
    execl("/bin/grep", …);
}
```

[picture]
Next Time

• More IPC
  • Message Passing
  • Shared Memory