Unix Kernel Calls: Pipes

CSCI 315
Spring 2005

Inter-Process Communication

- Mechanisms which allow processes to share information enable inter-process communication, *IPC*.
- Have already seen two IPC mechanisms for Unix heavyweight processes:
  - Exit/wait rendezvous allows parent to recover information from a child at the child's termination.
  - Signals allow one process to interrupt the normal flow of control of another.
- We want to enable general data sharing between processes.
- For (pedagogical) example:
  - Parent creates child.
  - Sends child integer.
  - Child returns that value, doubled.
  - Perhaps in a loop.

Inter-Process Communication

- Complication: heavyweight processes share no memory. No common variables through which communication can take place.
- In our favor: child inherits the open file descriptors of the parent. I/O of some sort can allow communication.
- Proposed solution:
  - Parent opens file(s), forks child.
  - Parent and child communicate with `read()`, `write()`, and `lseek()` on the shared file.

Attempt 1

```
// filecommo1: parent/child talk
// via file

main()
{
  int fd, val, dblval;
  fd = open("commofile",
     O_RDWR|O_CREAT|O_TRUNC, 0644);
  if (fork() == 0) { /* CHILD */
    read(fd,&val,sizeof(int));
    lseek(fd, 0, SEEK_SET);
    dblval = 2*val;
    write(fd,&dblval,sizeof(int));
    lseek(fd, 0, SEEK_SET);
    exit(0);
  }
```

```
else { /* PARENT */
  val = 2;
  fprintf(stderr,"Asking child to
              double %d\n", val);
  write(fd, &val, sizeof(int));
  lseek(fd, 0, SEEK_SET);
  read(fd, &dblval, sizeof(int));
  fprintf(stderr,"Child replied
            with %d\n", dblval);
  wait(NULL);
  exit(0);
  }
}
```

Attempt 1

```
% ./filecommo1
Asking child to double 2
Child replied with 2
%
```

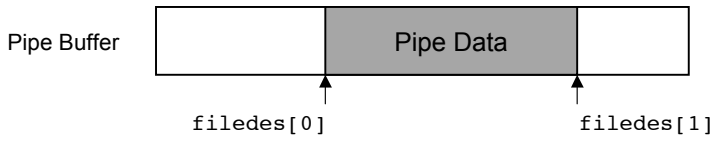
- Child apparently runs to completion before parent does any file i/o. Parent reads its own written value.
- Fundamental problem: need more control over access to the shared file.
- Specifically, read from an empty file (or read when currency indicator is at EOF) should delay caller until data available to be read.
- Can hack up a solution with a loop around the `read()`.
- Unix solution is a construct called a *pipe*.

The `pipe(2)` System Call

```
NAME
  pipe - create pipe
SYNOPSIS
  #include <unistd.h>
  int pipe(int filedes[2]);
```

- Allocates *two* file descriptors. Fills in slots of `filedes[]` with those descriptors.
- The `filedes[0]` descriptor is bound to the *read side* of the pipe.
- The `filedes[1]` descriptor is bound to the write side of the pipe.
- Pipe is implemented as a buffer of some fixed size in the kernel.

The `pipe(2)` System Call



- If process does a `read()` on a pipe with insufficient data to satisfy the read, the process waits until data available.
- If process does a `write()` on a pipe that doesn't have enough free space, the process waits.
- Pipe is a *synchronization* device in addition to a conduit for IPC.
- Departmental Linux systems provide a 4kB pipe buffer.
- If `read()` on a pipe returns 0, *and there are no live write descriptors on the pipe*, then can conclude EOF.
- If `write()` to a pipe *with no live read descriptors*, `SIGPIPE` generated, -1 retval, and `errno` is set to `EPIPE`.

Attempt 2: Using a Pipe

```
// pipecommo1: double parent's ints in a loop; one pipe for commo.

int fd[2], val=0, dblval=0;
main()
{ pipe(fd);
  if (fork() == 0) { /* CHILD */
    while(read(fd[0], &val, sizeof(int)) != 0) {
      dblval = 2*val; write(fd[1], &dblval, sizeof(int));
    }
  exit(0);
  }
  else { /* PARENT */
    for (val=1; val<=3; val++){
      fprintf(stderr,"Asking child to double %d\n", val);
      write(fd[1], &val, sizeof(int)); read(fd[0], &dblval, sizeof(int));
      fprintf(stderr,"Child replied with %d\n", dblval);
    }
    wait(NULL);
  }
}
```

Attempt 2: Using a Pipe

```
% ./pipecommo1
Asking child to double 1
Child replied with 1
Asking child to double 2
Child replied with 2
Asking child to double 3
Child replied with 3
<HANGS>
```

- Base problem: pipe is essentially a construct for *unidirectional* transfer of information. Parent is reading its own data written into the pipe.
- Need one pipe for synchronized parent-to-child commo.
- Need second pipe for synchronized child-to-parent commo.
- EOF?

Attempt 3: Using a Pipe the Right Way

```
// pipecommo3: parent and child talk via two pipes

int p2c[2], c2p[2], val=0, dblval=0;

main()
{ pipe(p2c); pipe(c2p);  // Parent creates pipes; child inherits.
  if (fork() == 0) {              // CHILD
    close(p2c[1]); close(c2p[0]); // Child doesn't need these!
    while(read(p2c[0], &val, sizeof(int)) != 0) {
      dblval = 2*val;
      write(c2p[1], &dblval, sizeof(int));
    }
    exit(0);                      // Will close all file
                                  // descriptors in child.
  }
```

Parent-to-child pipe descriptors.

Child-to-parent pipe descriptors.

EOF test embedded in while test.

Attempt 3: Using a Pipe the Right Way

```
  else { // PARENT
    close(c2p[1]); close(p2c[0]);  // Parent doesn't need these.
    for (val=1; val<=3; val++){
      fprintf(stderr,"Asking child to double %d\n", val);
      write(p2c[1], &val, sizeof(int));
      read(c2p[0], &dblval, sizeof(int));
      fprintf(stderr,"Child replied with %d\n", dblval);
    }
    close(p2c[1]); close(c2p[0]);  // First one is critical!
    wait(NULL);
  }
}
```


Attempt 3: Using a Pipe the Right Way

```
% ./pipecommo3
Asking child to double 1
Child replied with 2
Asking child to double 2
Child replied with 4
Asking child to double 3
Child replied with 6
%
```

- Functionally correct.
- Terminates properly (doesn't hang).
- Big lessons:
  - Parent makes pipes, and child inherits them.
  - A pipe is unidirectional.
  - Close all unneeded descriptors ASAP, else EOF trouble.