CS 3411 Systems Programming

Department of Computer Science
Michigan Technological University

Pipe Inter-Process Communication in Unix

Today's Topics

- How to communicate between processes without using signals
- Creating and Using Pipes

How to Communicate Between Processes

- Have two primitive mechanisms in hand:
  - exit/wait
  - signals
- The sample program we'll be working on:
  - Parent creates child
  - Send child an int, x
  - Child computes 2x and returns result
  - Maybe a loop that keeps doing this!
- Parent and child share no memory; i.e. no common variables through which to communicate

How to Communicate Between Processes

- Parent and child share open file descriptors, even after fork() or fork()/exec()
- Proposed solution:
  - Parent opens files, forks child
  - Child arranges descriptors with close and dup, execs new binary
  - Parent and child communicate via read()/write() on shared file

Example Code

```
#include <fcntl.h>
#include <stdio.h>
#include <sys/types.h>
#include <unistd.h>
#include <stdlib.h>
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);
   }
}
```

Results

- Why did it turn out like that?
- 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 is available to read
- The Unix solution is a construct called a *pipe*

The pipe() System Call

- The pipe() function takes one argument in the form of an array with 2 elements
- The first element is the *read* end of the pipe
- The second element is the *write* end of the pipe
- Data written to the pipe is buffered by the kernel until it is read

Pipes

- Can think of a pipe as an unnamed, fixed length file maintained by the kernel
- We also looked at the named versions! They're identical once opened - no data is to the device in either case!
- Pipes have separate file descriptors as well as currency indicators for reading and writing
- Some special properties:
  - A read from a pipe that doesn't have sufficient data to satisfy the read will block the reader until the data is available
  - A write to a pipe that is full will delay the writer until space becomes available
  - From a reader's perspective, EOF can only happen if there is no data in the pipe and all write descriptors on the pipe are closed
  - From a writer's perspective, an attempt to write to a pipe without live read descriptors will result in a SIGPIPE signal being sent to the writer

Example Code

```
#include <fcntl.h>
#include <stdio.h>
#include <sys/types.h>
#include <unistd.h>
#include <stdlib.h>
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);
  }
}
```

Results

- What now?
- Another 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 communication
- And a second pipe for synchronized child-to-parent communication
- When do we get EOF?

Example Code

```
#include <fcntl.h>
#include <stdio.h>
#include <sys/types.h>
#include <unistd.h>
#include <stdlib.h>
int p2c[2], c2p[2], val = 0, dblval = 0;
main() {
  pipe(p2c); pipe(c2p);
  if (fork() == 0) { /* CHILD */
    close(p2c[1]); close(c2p[0]);
    while(read(p2c[0], &val, sizeof(int)) != 0) {
      dblval = 2*val;
      write(c2p[1], &dblval, sizeof(int));
    }
    exit(0);
  } else { /* PARENT */
    close(c2p[1]); close(p2c[0]);
    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]);
    wait(NULL);
  }
}
```

Results

- It works as expected this time!
- Let's take a look at some other examples

Real Examples I

```c
#include <fcntl.h>
#include <stdio.h>
#include <sys/types.h>
#include <unistd.h>
#include <stdlib.h>
/* We want to execute "ps -aux | grep root" */
main() {
  int isParent;
  int apipe[2];
  char *cmd[2][3];

  cmd[0][0] = "ps";   cmd[0][1] = "-aux"; cmd[0][2] = NULL;
  cmd[1][0] = "grep"; cmd[1][1] = "root"; cmd[1][2] = NULL;

  pipe(apipe);
  isParent = fork();

  if (!isParent) { /* Child is going to be "grep root" */
    /* We want stdin connected to our pipe! */
    close(apipe[1]);
    close(0);
    dup(apipe[0]);
    close(apipe[0]);
```

Real Examples II

```
        execvp(cmd[1][0],cmd[1]);
        perror("Child exec::"); exit(1);
    }
    else { /* Parent is "ps -aux" */
    /* We want the stdout connected to pipe */
        close(apipe[0]);
        close(1);
        dup(apipe[1]);
        close(apipe[1]);

        execvp(cmd[0][0], cmd[0]);
        perror("Parent exec::"); exit(1);
    }
}
```

Real Examples I

```c
#include <fcntl.h>
#include <stdio.h>
#include <sys/types.h>
#include <unistd.h>
#include <stdlib.h>
/* We want to execute "sort < filecomm1.c | grep fprintf | wc" */
main() {
  int apipe[2];
  int isParent;
  char *cmd[3][3];
  int i, lastChild, fd, saveStdout;
  cmd[0][0] = "sort"; cmd[0][1] = NULL;
  cmd[1][0] = "grep"; cmd[1][1] = "fprintf"; cmd[1][2] = NULL;
  cmd[2][0] = "wc";   cmd[2][1] = NULL;
  saveStdout = dup(1);
  for(i = 2; i >= 0; i--) {
    pipe(apipe);
    isParent = fork();

    if (!isParent) {
      close(apipe[1]);
      close(0);
      if(i != 0) { dup(apipe[0]); }
```

Real Examples II

```
            if(i == 0) {
              fd = open("filecomm1.c", O_RDONLY);
              dup(fd);
            }
            close(apipe[0]);

            execvp(cmd[i][0],cmd[i]);
            exit(1);
          }
          else {
            if(i==2) lastChild = isParent;
            close(apipe[0]);
            close(1);
            if(i!=0) { dup(apipe[1]); }
            close(apipe[1]);
            if(i==0){
              dup2(saveStdout, 1);
              waitpid(lastChild,NULL,0);
            }
          }

      }
  }
```