Inter-process Communication

Using Pipes

Mark Sitkowski C.Eng, M.I.E.E
http://www.designsim.com.au

The Unix pipe is one of the earliest types of inter-process communication device available to systems programmers. The pipe is limited to sending data between processes running on the same machine, and it was the basis for the design of the socket, which extended inter-process communications beyond the machine.
It was, originally, a one-way data transfer device but, on many current versions of Unix, either end can be read from or written to, which permits a few design freedoms, in sending data between processes.

Whereas a socket has one socket descriptor, which serves for both reading and writing, a pipe has a reading end, and a writing end, each of which has an associated pipe descriptor. These pipe descriptors are of type int, and are returned into a two element array, by the pipe() system call, which creates the pipe.

```
int pfd[2];

        if(pipe(pfd) == -1){
           perror("Pipe system call failed");
        }
```

Having acquired our pipe descriptors, all we need to do, is to arrange for one process to write to one end, while another process reads the other end, and we have a data transfer link.
However, if we try to write to a pipe, when there is nothing trying to read from it, an error will result, so the synchronisation is important.

Basic Client Server Example
Here is a simple example of the use of a pipe for communication between a parent process, and its child. To illustrate the bi-directionality of the pipe, the parent first writes to, then reads from pipe descriptor[1], while the child does the same with pipe descriptor[0]. To ensure that we always have a reader before we try to write, we make each process sleep for one second before writing our message..

```
server()            /* server */

{

int nread;
char mesg[512] = "Thanks for connecting\n";
char resp[2048];

   printf("Server running\n");

   if(pipe(pfd) < 0){
      perror("pipe");
   }

   client(pfd);
```

```
    if(write(pfd[1], mesg, strlen(mesg)) != strlen(mesg)){
        perror("Server Write");
    } else {
        printf("Server wrote: >%s<\n", mesg);
        sleep(1);
        if((nread = read(pfd[1], resp, sizeof(resp))) >= 0){
            printf("Server read client's reply >%s<\n", resp);
        }
    }
    close(pfd[0]);
    close(pfd[1]);

}                       /* server */


client(pp)              /* client */

int pp[2];

{

pid_t pid;
int nread;
char buf[2048];
char resp[2048] = "You're welcome\n";

    switch((pid = fork())){
        case -1:
            perror("fork failed");
        break;
        case 0:
            printf("Client running\n");

            if((nread = read(pp[0], buf, sizeof(buf))) > 0){
                printf("Client Read:>%s<\n", buf);
                if(buf[0] == 'T'){
                    sleep(1);
                    if(write(pp[0], resp, sizeof(resp)) == -1){
                        perror("Client Write");
                        exit(-1);
                    } else {
                        printf("Client replied:>%s<\n", resp);
                    }
                }
            } else {
                perror("Client read nothing");
            }
            printf("Client finished conversation\n");
            exit(0);
        break;
        default:
            printf("Client PID %d running\n", pid);
        break;
    }
```

```
}                    /* client */
```

The parent, or server, first creates a pipe, then passes the pipe descriptors to the function which launches the client, as a child process.
The client function forks a child, which inherits the two open pipe descriptors.
Next, the parent (server) writes a message to pipe descriptor 1, and sleeps for one second, to allow the child time to read the message. This step is necessary since, otherwise, the parent would read back its own message, and the child would read EOF, and quit. While the parent sleeps, the child is reading pipe descriptor 0, and receives the message sent by the server. Then, knowing that the server needs a finite time, to switch from writing to reading, the child sleeps for one second, before sending its reply.

The above example is fairly contrived, and shows that using what is basically a uni-directional device in a bi-directional mode needs to have event-driven synchronisation. However, having said that, if we have a situation where we fork a number of children, each of which only sends back one chunk of data, or one message, the pipe provides a fairly cheap solution.

Pipes, as well as sockets, have a finite buffer length. Whereas, with a socket, we can call getsockopt() and increase it to any size we like, with a pipe, this is a fixed number, which is between 4096 and 32k, depending on your version of Unix. The practical implications of this, are that, if there is any discrepancy, between the reading speed, and the writing speed, the pipe will fill up, and block one end, until the other end catches up. This will naturally, have a negative effect on throughput.

Communicating with Stdin and Stdout of External Processes

The classic use of pipes is in establishing communication between an application, and external utilities, whose only interface with the outside world is via stdin and stdout.
A good example of this, is where an application needs to send email, (with or without file attachments), on occurrence of a fatal error. It is easy enough to fork and exec the sendmail utility, but passing it the information is not so easy.

The technique employed to do this, works as follows:

1. Create a pipe, to carry data from our application to the utility,
2. Create a second pipe, to carry data from the utility to our application.
3. Fork the application.
4. In the child, close the reading end of the pipe created in (2) and the writing end of the pipe created in (1).
5. Close file descriptor 0 (stdin) and file descriptor 1 (stdout)
6. Dup the open pipe descriptor of the pipe coming from the parent.
7. Dup the open pipe descriptor of the pipe going to the parent
8. Exec the utility

The interesting part of the above sequence of events occurs in the child process. We close file descriptors 0 and 1, (Step 5), which are the lowest numbered file descriptors. Then, the first time we call dup(), (Step 6) with the reading end of the pipe from the application, we get handed the lowest numbered file descriptor, which is 0. This operation glues the incoming pipe to the child’s stdin, which means that any data appearing on that pipe appears to come from the keyboard. The second time we call dup(), (Step 7) with the writing end of the pipe to our application, we again get handed fhe lowest numbered file descriptor, which is 1. Now, anything the child tries to write to the screen, gets sent to our application.

The sendmail Example

```
send_mail(char *who, char *subj, char *msg)           /* send_mail */

{

char mail[2048];    /* subject and message */
char arg1[255];      /* command line argument */
char arg2[255];      /* command line argument */
char arg3[255];       /* recipient */
char errors[255];    /* messages returned from sendmail */

    sprintf(mail, "Subject: %s\n\n%s\n", subj,, msg);
    strcpy(arg1, "-R");
    strcpy(arg2, "HDRS");
    strcpy(arg7, who);

   /* pipe from parent to child */
   if(pipe(pfd) == -1){
      printf("Failed to create outgoing pipe\n");
      exit(-1);
   }
    /* pipe from child to parent */
   if(pipe(qfd) == -1){
      printf("Failed to create incoming pipe\n");
      exit(-1);
   }

   switch((mpid = fork())){
      case -1:
         perror("Fork failed in send_mail");
         exit(-1);
      break;
      case 0:                /* Child process */
         /* this child needs to read the pipe instead of stdin */
         if(close(0) == -1){
            perror("close stdin");
            exit(-1);
         }
         /* we'll only be reading */
         if(close(pfd[1]) == -1){
            perror("close writing end of pipe");
            exit(-1);
         }
         /* make stdin the same as the pipe */
         if(dup(pfd[0]) != 0){
            perror("dup reading pipe descriptor");
         }
           /* the child needs to write to the pipe instead of stdout */
         if(close(1) == -1){
            perror("close stdout");
            exit(-1);
         }
         /* we'll only be writing */
         if(close(qfd[0]) == -1){
            perror("close reading end of pipe");
            exit(-1);
         }
         /* make stdout the same as the pipe */
         if(dup(qfd[1]) != 0){
            perror("dup writing pipe descriptor");
         }

         /* now, when the child writes to stdout, it writes to the pipe
           * and, when it reads stdin, it reads the other pipe
            */
         execl("/usr/lib/sendmail", "sendmail", "-f", "Sender", arg1, arg2
, arg3, NULL);

      break;
      default:
         /* close reading end of pipe to sendmail */
         if(close(pfd[0]) == -1){
            perror("close reading end of outgoing pipe");
            exit(-1);
         }
          /* close writing end of pipe from sendmail */
         if(close(qfd[1]) == -1){
            perror("close writing end of incoming pipe");
            exit(-1);
         }

         /* send the message *
              if(write(pfd[1], mail, strlen(mail)) != strlen(mail)){
                     perror("write mesg");
                 }
         /* read the reply */
                 if(read(qfd[0], errors, strlen(errors)) == 0){
                     printf("No errors reported\n");
                 }
       break;
      }

}                                              /* send_mail */
```