CHAPTER VII: PIPES

B. Boufama

University of Windsor

Introduction

Some basic Interprocess communication (IPC) can be achieved using :

- Signals : using the *kill()* system call.
- Files : by passing open files across the *fork()/exec()*

Example 1 :

```
void action(){
 sleep(1);
}
int main(int argc, char *argv[]){
  FILE *fp;
  pid_t pid;
  int childRes;

  fp = fopen("/tmp/ipoc.txt", "w+");
  setbuf(fp, NULL);
```

```
  if((pid=fork()) == 0)
    child(fp);
  else
    parent(fp, pid);
}
void parent(FILE *fp, pid_t pid){
  int childRes, n=0;
  signal(SIGUSR1, action);
  while(1){
    pause();
    rewind(fp);
    fread(&childRes, sizeof(int), 1, fp);
    printf("\nParent: child result: %d\n", childRes);
    if(++n>5){
      printf("Parent: work done, bye bye\n");
      kill(0, SIGTERM);
    }
    printf("Parent: waiting for child\n\n");
    kill(pid, SIGUSR1);
  }
}
```

```
void child(FILE *fp){
  int value;

  signal(SIGUSR1, action);
  while(1){
    sleep(1);
    value = random()%100;
    rewind(fp);
    fwrite(&value, sizeof(int), 1, fp);
    printf("Child: waiting for parent\n\n");
    kill(getppid(), SIGUSR1);
    pause();
  }
}
```

Unnamed Pipes

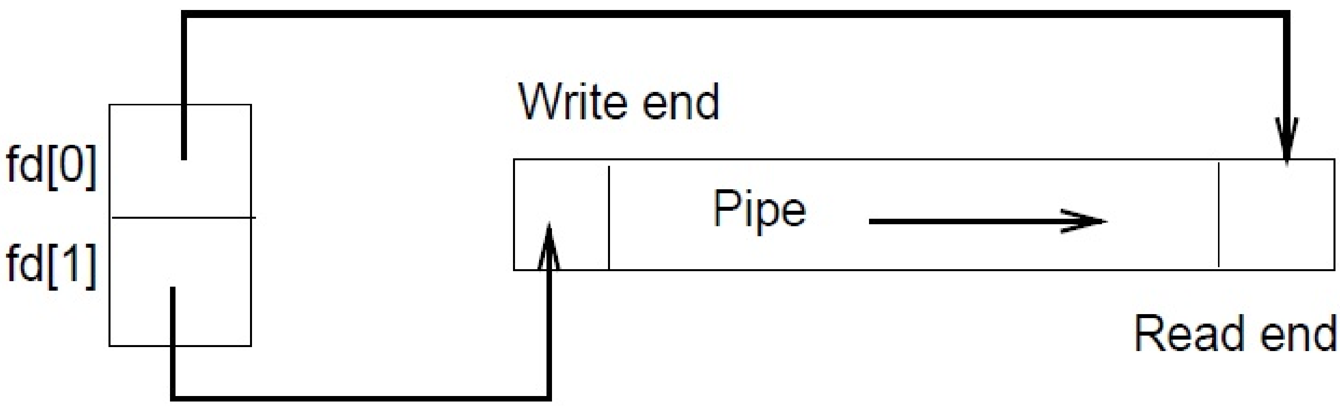
Unnamed Pipes, or *pipes*, are an IPC mechanism.
In particular, they are used by shells to connect one utility's standard output with the standard input of another utility.
Example : ps -ef | grep Chrome | wc -l

Pipes are the oldest form of Unix IPC. They have two limitations :

- They are half-duplex : data flows in one direction only
- They can be used only between processes that have a common ancestor. Typically, a process creates a pipe, forks, then uses the pipe to exchange information with its child.

The *pipe()* System call

Synopsis : int pipe(int fd[2])
returns 0 when successful and -1 otherwise
pipe() creates a pipe and returns two file descriptors, fd[0] and fd[1], where fd[0] is open for reading and fd[1] is open for writing.



$\longrightarrow$ a pipe is a one-way communication channel between two related processes.

When reading from or writing to a pipe, the following rules apply :

- If a process reads from a pipe whose write end has been closed, after all data has been read, *read()* returns 0 (end-of-file).
- If a process reads from an empty pipe whose write end is still open, it sleeps until some input becomes available.
- If a process tries to read from a pipe more bytes than are present, *read()* reads all available bytes and returns the number of bytes read.
- If a process writes to a pipe whose read end has been closed, the write operation fails and the writer process recieves a SIGPIPE.

Example 2: child is a writer, parent is a reader

```
#include <stdio.h>
#include <unistd.h

void child(int *);
void parent(int *);

int main(int argc, char *argv[]){
  int fd[2];

  if(pipe(fd) == -1)
    exit(1);

  if(fork() == 0)
    child(fd);
  else
    parent(fd);
  exit(0);
}
```

```
void parent(int *fd){
  char ch;

  close(fd[1]);
  printf("Child has sent the message:\n");
  do{
    read(fd[0], &ch, 1);
    printf("%c", ch);
    if(ch == '\n')
      break;
  }while(1);
}
void child(int *fd){
  char message[255]="Hello, here is my data...\n";

  close(fd[0]);

  write(fd[1], message, 26);
}
```

Note that two pipes are necessary in order to get a bidirectional communication.

The system call : dup2()

Synopsis : int dup2(int fd, int fd2);

The *dup2()* function causes the file descriptor *fd2* to refer to the same file as *fd*. The *fd* argument is a file descriptor referring to an open file, and *fd2* is a non-negative integer less than the current value for the maximum number of open file descriptors allowed the calling process.
If *fd2* already refers to an open file, not *fd*, it is closed first. If *fd2* refers to *fd*, or if *fd* is not a valid open file descriptor, *fd2* will not be closed first.
If successful, *dup2()* returns a non-negative integer representing the file descriptor *fd2*. Otherwise, -1 is returned

Example: *dup2(fd, 0);*
→ reading from *stdin* will mean reading from the file whose descriptor is *fd*.

Example 3: implementing the shell pipe mechanism

```
int main(int argc, char *argv[]){
  int fd[2];

  if(pipe(fd)==-1) exit(1);
  if(fork() > 0)
    parent(fd, argv);
  else child(fd, argv);
}
void parent(int *fd, char *argv[]){ // A writer
  close(fd[0]);
  dup2(fd[1], 1); // 1 is the standard output
  close(fd[1]); // close original file descriptor
  execlp(argv[1], argv[1], NULL);
}
void child(int *fd, char *argv[]){ // A reader
  close(fd[1]);
  dup2(fd[0], 0); // 0 is the standard input
  close(fd[0]);  // close original file descriptor
  execlp(argv[2], argv[2], NULL);
}
```

Example 4: implementing the shell pipe mechanism

```
int main(int argc, char *argv[]){
  int fd1[2], fd2[2], fd3[2], fd4[2];
  char turn='T';

  printf("This is a 2-player game with a referee\n");
  pipe(fd1);
  pipe(fd2);
  if(!fork())
    player("TOTO", fd1, fd2);

  close(fd1[0]);   // parent only write to pipe 1
  close(fd2[1]);   // parent only reads from pipe 2

  pipe(fd3);
  pipe(fd4);
  if(!fork())
    player("TITI", fd3, fd4);

  close(fd3[0]);   // parent only write to pipe 3
  close(fd4[1]);   // parent only reads from pipe 4
```

```
  while(1){
    printf("\nReferee: TOTO plays\n\n");
    write(fd1[1], &turn, 1);
    read(fd2[0],  &turn, 1);

    printf("\nReferee: TITI plays\n\n");
    write(fd3[1], &turn, 1);
    read(fd4[0],  &turn, 1);
  }
}
```

```
void player(char *s, int *fd1, int *fd2){
  int points=0;
  int dice;
  long int ss=0;
  char turn;

  while(1){
    read(fd1[0], &turn, 1);
    printf("%s: playing my dice\n", s);
    dice =(int) time(&ss)%10 + 1;
    printf("%s: got %d points\n", s, dice);
    points+=dice;
    printf("%s: Total so far %d\n\n", s, points);
    if(points >= 50){
      printf("%s: game over I won\n", s);
      kill(0, SIGTERM);
    }
    sleep(5);    // to slow down the execution
    write(fd2[1], &turn, 1);
  }
}
```

FIFOs or Named Pipes

FIFOs(First In First Out), sometimes called named pipes, offer the following adavantages over pipes :

- They have a name that exists in the file system.
- They can be used by unrelated processes.
- They exist until explicitly deleted.

The system call *mkfifo():*
int mkfifo(const char *path, mode_t mode)
mkfifo() return 0 if OK, -1 otherwise.

Creating a FIFO is similar to creating a file.
Example : mkfifo("/tmp/channel.fif", "0755");

Once a FIFO has been created, it can treated it as a file. In particular, the system calls *open()*, *close()*, *read()*, *write()* and *unlink()*(to delete a file) can be used on a FIFO.

By default, we have :

- Calling *open()* for read only blocks the caller until some other process opens the FIFO for writing.
- Calling *open()* for write only blocks the caller until some other process opens the FIFO for reading.
- If a process writes to a FIFO that no process has open for reading, the signal *SIGPIPE* will be generated.
- When the last writer for a FIFO closes the FIFO, an end-of-file will be generated for the reader.
- Like pipes, FIFOS are one-way communication channels.

```
Example 5: a client/server application where a server
accepts data from clients using the FIFO /tmp/server.
#include <fcntl.h>
#include <stdio.h>      // This is the server

int main(int argc, char *argv[]){
  int fd;
  char ch;

  unlink("/tmp/server");  // delete it if it exists
  if(mkfifo("/tmp/server", 0777)!=0)
    exit(1);
  chmod("/tmp/server", 0777);
  while(1){
    fprintf(stderr, "Waiting for a client\n");
    fd = open("/tmp/server", O_RDONLY);
    fprintf(stderr, "Got a client: ");
    while(read(fd, &ch, 1) == 1)
      fprintf(stderr, "%c", ch);
  }
}
```

```
#include <fcntl.h>           // This is the client
#include <stdio.h>

int main(int argc, char *argv[]){
  int fd;
  char ch;

  while((fd=open("/tmp/server", O_WRONLY))==-1){
    fprintf(stderr, "trying to connect\n");
    sleep(1);
  }

  printf("Connected: type in data to be sent\n");

  while((ch=getchar()) != -1) // -1 is CTR-D
    write(fd, &ch, 1);

  close(fd);
}
```

Example 6: a client/server application where a server creates a child for each client. Then, the child creates a seperate FIFO to send data to the client.

```
#include <wait.h>
#include <fcntl.h>            // This is the server
#include <stdio.h>

void child(pid_t client);

int main(int argc, char *argv[]){
  int fd, status;
  char ch;
  pid_t pid;

  unlink("/tmp/server");
  if(mkfifo("/tmp/server", 0777)){
    perror("main");
    exit(1);
  }
  chmod("/tmp/server", 0777);
```

```
  while(1){
    fprintf(stderr, "Waiting for a client\n");
    fd = open("/tmp/server", O_RDONLY);
    fprintf(stderr, "Got a client: ");
    read(fd, &pid, sizeof(pid_t));
    fprintf(stderr, "%ld\n", pid);
    if(fork()==0){
      close(fd);
      child(pid);
    }else
      waitpid(0, &status, WNOHANG);
  }
}
```

```
void child(pid_t pid){
  char fifoName[100];
  char newline='\n';
  int fd, i;

  sprintf(fifoName,"/tmp/fifo%d", pid);
  mkfifo(fifoName, 0777);
  chmod(fileName, 0777);
  fd = open(fifoName, O_WRONLY)

  for(i=0; i < 10; i++){
    write(fd, fifoName, strlen(fifoName));
    write(fd, &newline, 1);
  }
  close(fd);
  unlink(fifoName);
  exit(0);
}
```

```
int main(int argc, char *argv[]){  // Client
  char ch, fifoName[50];
  int fd;
  pid_t pid;

  while((fd=open("/tmp/server", O_WRONLY))==-1){
    fprintf(stderr, "trying to connect\n");
    sleep(1);
  }
  pid = getpid();
  write(fd, &pid, sizeof(pid_t));
  close(fd);

  sprintf(fifoName,"/tmp/fifo%ld", pid);
  sleep(1);

  fd = open(fifoName, O_RDONLY);
  while(read(fd, &ch, 1) == 1)
    fprintf(stderr, "%c", ch);
  close(fd);
}
```