Sistemas Operativos
Grado en Ingeniería Informática en Sistemas de Información

Enseñanzas de Prácticas y Desarrollo
PRÁCTICA 10: Inter Process Communication

Objectives

- Use and understand IPC mechanisms

Concepts

Inter Process Communication (IPC) mainly involves the techniques and mechanisms that allow the functions for the communication between processes. We need special techniques and mechanisms since processes do not share the same address space. On the contrary, each process has its own portion of memory, and cannot access the memory assigned to other processes. The OS kernel will act as a communication agent since the kernel has access to all the memory.

There are various IPC mechanisms available, and that are based on different requirements. These mechanims can be divided in the following categories:

- pipes
- fifos
- shared memory
- mapped memory
- message queues
- sockets

In this class we will focus our attention of pipes and fifos.

1. Pipes

Pipes provide a unidirectional communication flow between processes of the same system. In other words, pipes are “half-duplex”, that is, data flow only in one direction.

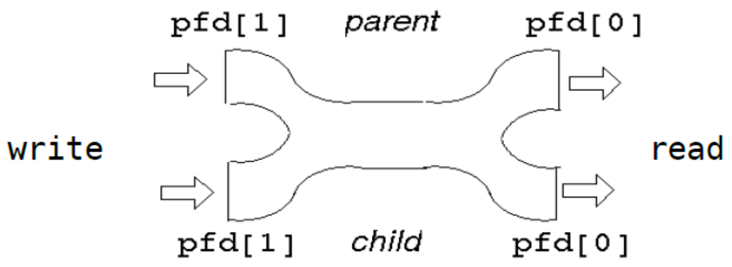
A pipe can be created with the system call pipe, that creates two file descriptors. These descriptors point to the i-node of the pipe, and are returned in a two elements array that need to be passes as a parameter to the call. If the array name is, for instance, filedes, then the descriptor contained in filedes[0] is used for reading and the file descriptor contained in filedes[1] is used for writing.

Read and write operations are carried out using the same instructions that we used for reading/writing a file. An example of the system call pipe is the following:
```
pipe(filedes);
```

where filedes is a bi-dimensional array of integer:
```
int filedes[2];
```

The system call returns -1 in case of errors, 0 otherwise.

To summarize, a pipe can be seen as in the following picture:



An example of use of pipes is the following:

```
#include <stdio.h>
#include <stdlib.h>
#include <errno.h>
#include <unistd.h>
int main()
{
  int pfds[2];
  char buf[30];
  if (pipe(pfds) == -1) {
    perror("pipe");
    exit(1);
  }
  printf("writing to file descriptor #%d\n", pfds[1]);
  write(pfds[1], "test", 5);
  printf("reading from file descriptor #%d\n", pfds[0]);
  read(pfds[0], buf, 5);
  printf("read \"%s\"\n", buf);
}
```

This example shows how to use the system call pipe and how to write and read from the pipe, but it actually does it from the same process, which is of no use. Normally, pipes are used in combination with the fork call (see EPD 3). In this case, we will have a mechanisms that would allow father and child to communicate.

In order to establish a communication between father and child, the father first has to create the pipe. After that, it will create the child with the fork. After that, for instance, the child could send data to the father using the writing end of the pipe. On the other hand, the father could read the data sent using the reading end of the pipe.

It is important to create the pipe **before** the instruction fork, since, in this way, both father and child will have a copy of the pipe descriptor initialized in the same way. In fact, as we have seen in a previous class, when the child is created with the fork, it will have a copy of all the variables declared by the father before the fork, and with the same initialization.

The following code shows an example:

```
#include <stdio.h>
#include <string.h>
#include <stdlib.h>
#include <unistd.h>
#include <sys/types.h>
int main(void)
{
  int fd[2], nbytes;
  pid_t childpid;
  char string[] = "Hello, world!\n";
  char readbuffer[80];
  pipe(fd);
  if((childpid = fork()) == -1)
  {
    perror("fork");
    exit(1);
  }
  if(childpid == 0)
  {
    /* Child process closes up input side of pipe */
    close(fd[0]);
    /* Send "string" through the output side of pipe */
    write(fd[1], string, (strlen(string)+1));
    exit(0);
  }
  else
  {
    /* Parent process closes up output side of pipe */
    close(fd[1]);
    /* Read in a string from the pipe */
    nbytes = read(fd[0], readbuffer, sizeof(readbuffer));
    printf("Received string: %s", readbuffer);
  }
  return(0);
}
```

If we check the code, we will notice that the child closes the reading end of the pipe, while the father closes the writing end. This is a normal operation since the IPC with pipes is much simpler if we use the pipe unidirectionally. In this way, the father will only be able to read from the pipe, while the child will only be able to write in the pipe. If we need a bi-directional communication, then we should use two pipes, one per direction.

To sum things up, the prototype of the system call pipe is the following:

```
int pipe(int fd[2]);
```

return 0 on success, -1 otherwise.
fd[0] is set up for reading, fd[1] is set up for writing

Pipes have a **synchronization** mechanism: a read operation blocks the process that runs it until there are data available in the pipe.

A write operation blocks the process that run it until there is space in the pipe.

2. FIFO

A FIFO (First In, First Out) is also known as a named pipe. The FIFO is a special file that two or more processes can open, read and write.

FIFO were created to solve a problem that characterized normal pipes, i.e., a process cannot use a pipe created by a nonrelated process. Normal pipes can be used, for example, in order to have the father communicating with a child. However, two unrelated processes cannot communicate via a pipe, since they cannot share the descriptor of the pipe.

Instead, a FIFO is a special file and is similar to a pipe, but instead of being an anonymous, temporary connection, a FIFO has a name or names like any other file. Processes open the FIFO by name in order to communicate through it. Since a FIFO is a special file, it will be placed somewhere in the file system, thus it will be visible to all the process that run in the same system.

In order to create a FIFO from a C program, we can use the mknod() command.

`mknod();`
prototype: `int mknod(char *pathname, mode_t mode, dev_t dev);`
return 0 on success, -1 otherwise

char* pathname specifies the name of the FIFO, with its path
mode_t mode specifies the permissions assigned to the FIFO, in the same way as when a file is created
dev_t dev specifies the device number. This parameter can be ignored when a FIFO is created, and we'll set it to 0.

An example of creating a FIFO is the following:

```
mknod("/tmp/MYFIFO", S_IFIFO|0666, 0);
```

The above call creates a file "/tmp/MYFIFO" as FIFO. Permission will be "0666". The third parameter is ignored, since it is used only when a device file is created.

2.1. OPERATIONS WITH FIFOs

I/O operations with FIFOs are essentially the same as with the pipe, with one different, we must open a FIFO with the open() instruction. With normal pipes, this is not necessary, since the pipe is managed by the OS kernel, and but by the file system.

It is important to notice that if a process open a FIFO for read/write, the process blocks until another process will open the same FIFO for read/write. So this provides a simple synchronization mechanism.

Bibliography

S. Goldt, S. van der Meer, S. Burkett, M. Welsh, The Linux Programmer's Guide, Version 0.4, March 1995, capitulo 6, pag. 16-76.

Experimentos

E1. Escribir, compilar y ejecutar este programa:

```
/***** KEYBOARD HIT PROGRAM *****/
#include <stdio.h>
#include <stdlib.h>
#include <sys/types.h>
#include <unistd.h>
#include <pthread.h>
#include <ctype.h>
#include <string.h>
int filedes[2];
void read_char()
{
  char c;
  char s[300];
  int num;
  printf("Entering routine to read character......\n Type a character: \n");
  c = getchar();
  while (c != '0'){
    if ((num = write(filedes[1], &c, 1)) == -1){
      perror("write");
      exit(1);
    }
    else {
      if (c != '\n'){
        printf("Child: wrote character: %c\n",c);
        printf("Child: Type a character: \n");
      }
    }
    c = getchar();
  }
  write(filedes[1], &c, 1);
}

void check_hit()
{
  char c;
  char s[300];
  int num;
  printf("Entering routine to check hit.........\n");
  do {
    //printf("Father: waiting\n");
    if ((num = read(filedes[0], &c, 1)) == -1)
      perror("read");
    else {
      if (c != '\n')
```

```
        printf("Father: bytes %d read \"%c\"\n",num, c);
    }
  } while (c != '0');
}

int main()
{
  int i;
  pthread_t tid1, tid2;
  pipe(filedes);
  int pid = fork();
  if (pid == -1)
    exit(1);
  if (pid == 0){
    //hijo
    read_char();
  }
  else{
    check_hit();
  }
  exit(0);
}
```

Analizar la ejecución del programa y el código.
Este código implementa la misma función, pero utilizando threads.

```
#include <stdio.h>
#include <stdlib.h>
#include <sys/types.h>
#include <unistd.h>
#include <pthread.h>
#include <ctype.h>
int filedes[2];
void *read_char()
{
  char c;
  int num =0;
  printf("Entering routine to read character.........\nType a character: \
n");
  c = getchar();
  while (c != '0'){
    if ((num = write(filedes[1], &c, 1)) == -1){
      perror("write");
      exit(1);
    }
    else {
      if (c != '\n'){
        printf("Writer: wrote character: %c\n",c);
        printf("Writer: Type a character: \n");
      }
    }
    c = getchar();
  }
  write(filedes[1], &c, 1);
  pthread_exit(0);
}

void *check_hit()
{
  char c;
  int num =0;
  printf("Entering routine to check hit.........\n");
  do {
    if ((num = read(filedes[0], &c, 1)) == -1)
      perror("read");
    else {
      if (c != '\n')
        printf("Reader: bytes %d read \"%c\"\n",num, c);
    }
  } while (c != '0');
  printf("Reader exiting\n");
  exit(0);
}

int main()
{
  int i;
  pthread_t tid1, tid2;
  pipe(filedes);
  /* Create thread for reading characters. */
  i = pthread_create(&tid1, NULL, read_char, NULL);
  /* Create thread for checking hitting of any keyboard key. */
  i = pthread_create(&tid2, NULL, check_hit, NULL);
  //if(i == 0) while(1);
  pthread_join(tid1,NULL);
  pthread_join(tid2,NULL);
  return 0;
}
```

En este segundo caso, ¿es necesario utilizar una pipe para poder realizar la comunicación?

E2. Escribir, compilar y ejecutar este programa:

```
#include <stdio.h>
#include <stdlib.h>
#include <sys/types.h>
#include <unistd.h>
int main()
{
  int pfds[2];
  char buf[30];
  pipe(pfds);
  if (!fork()) {
    printf(" CHILD: writing to the pipe\n");
    write(pfds[1], "test", 5);
    printf(" CHILD: exiting\n");
    exit(0);
  } else {
    printf("PARENT: reading from pipe\n");
    read(pfds[0], buf, 5);
    printf("PARENT: read \"%s\"\n", buf);
    wait(NULL);
  }
}
```

Analizar la ejecución del programa y el código.

E3 Guardar este código en un fichero llamado speak.c:

```
#include <stdio.h>
#include <stdlib.h>
#include <errno.h>
#include <fcntl.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <unistd.h>
#define FIFO_NAME "fifoPractica"
main()
{
  char s[300];
  int num, fd;
  /* don't forget to error check this stuff!! */
  mknod(FIFO_NAME, S_IFIFO | 0666, 0);
  printf("waiting for readers...\n");
  fd = open(FIFO_NAME, O_WRONLY);
  printf("got a reader--type some stuff\n");
  while (gets(s), !feof(stdin)) {
    if ((num = write(fd, s, strlen(s))) == -1)
      perror("write");
    else
      printf("speak: wrote %d bytes\n", num);
  }
}
```

Y este código en un fichero llamado tick.c:

```
#include <stdio.h>
#include <stdlib.h>
#include <errno.h>
#include <string.h>
#include <fcntl.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <unistd.h>
#define FIFO_NAME "fifoPractica"
main()
{
  char s[300];
  int num, fd;
  /* don't forget to error check this stuff!! */
  mknod(FIFO_NAME, S_IFIFO | 0666, 0);
  printf("waiting for writers...\n");
  fd = open(FIFO_NAME, O_RDONLY);
  printf("got a writer:\n");
  do {
    if ((num = read(fd, s, 300)) == -1)
      perror("read");
    else {
      s[num] = '\0';
      printf("tick: read %d bytes: \"%s\"\n", num, s);
    }
  } while (num > 0);
}
```

compilar los dos programas y darle nombre speak y tick respectivamente.

Ahora ejecutar speak primero. ¿Qué pasa? Ahora en otra shell ejecutar tick, ¿qué ha pasado? ¿Cuál es el efecto de la ejecución de los dos programas? ¿Qué pasa si terminamos uno de los dos procesos?

Problemas

P1. (20 mins) Implementar un programa que utilice dos pipe para comunicar entre un proceso padre y un proceso hijo. El proceso padre lee cadenas desde el teclado, y las envía al hijo a través de la primera pipe. El hijo lee las cadenas enviadas por el padre y las devuelve al padre a través de la segunda pipe. Los dos procesos imprimen en pantalla las cadenas que se leen.

P2. (20 mins) Implementar un programa que cree un proceso hijo. El proceso padre interactúa con el usuario y lee caracteres y los envía a través de una pipe al proceso hijo. El proceso hijo traduce todos las mayúsculas en minúsculas, y devuelve el resultado al padre a través de otra pipe. El padre imprime a pantalla la traducción. Sugerencia, la función tolower(char c) retorna la minúscula de un carácter c.

P3. (20 mins) Implementar los problemas P1 y P2 utilizando una FIFO y dos programas guardados en ficheros distintos.

Ampliación de Problemas

AP1. Implementar el comando “ls | wc -l” en C. Sugerencia. el comando dup() tiene como argumento un descriptor de fichero y hace un duplicado del descriptor in el primer descriptor de fichero disponible.

Por ejemplo, con las instrucciones

```
close(1);
dup(pdfs[1]);
```

podemos establecer que la salida standard sea pdfs[1], donde pdfs es un array que contiene los descriptores de ficheros creados con una llamada al sistema pipe(pdfs). Esto es porque el primer descriptor disponible era 1 (salida standard), como antes de ejecutar el comando dup hemos cerrado la salida standard, que es 1.