11

Comunicação entre processos
(IPC – InterProcess Communication)

Notas:

1. É condição prévia à aprendizagem deste assunto (IPC), o domínio sobre as funções **read** e **write** de I/O de baixo-nível.
2. O livro [2] de W. Stevens, *Unix networking programming* é a principal referência sobre este assunto. Nota: Edições posteriores foram ampliadas para dois volumes: Volume 1: A API de Redes com Sockets e Volume 2: Comunicações entre Processos.
3. O livro [1] de W. Stevens, *Advanced programming in the Unix environment*, serve só de apoio à aprendizagem.
4. Acesso aos livros de Stevens pode ser feito através da página http://www.kohala.com/start/
5. Codigo:
   Pode-se criar a biblioteca **libnet.a** (referente ao código do livro [2] da bibliografia) antes de testar quaisquer exemplos e exercícios.
   Em alternativa poderá remover as referências nos programas das funções de error (err_sys etc) e substituir por mensagens mais simples, imprimindo para o stderr ou substituir por macros da sua escolha.

P

No capítulo sobre Processos vimos as principais primitivas de controlo de processos, em particular a primitiva `fork()` que permite gerar processos. Mas, a única maneira destes processos comunicarem entre si era através da passagem de ficheiros abertos aquando dum `fork()` ou dum `exec()`, ou ainda através do próprio sistema de ficheiros.

Neste capítulo, outras técnicas de comunicação entre processos (IPC ou "interprocess communication") serão descritas, nomeadamente:

- Pipetas (pipes)
- FIFOs
- Filas de mensagens
- Memória partilhada
- Semáforos
- Sockets (máteria adicional)

Pipetas

Uma pipeta estabelece um canal IPC unidireccional entre dois programas de tal modo que a saída dum deles será a entrada do outro. Nenhum ficheiro intermédio é utilizado.

Na linha de comando do Unix, uma pipeta é criada com o comando '|'.

Exemplos 11.1, 11.2 e 11.3:

```
1. who | sort
2. ls | wc -l
3. who | sort | grep 2005 | wc -l
```

NOTA:
Embora o resultado das seguintes sequências de comandos:

```
1. com1 | com2
2. com1 > temp ; com2 < temp ; rm temp
```

seja idêntico, existe uma diferença **fundamental** entre elas.

A diferença reside não tanto na utilização do ficheiro intermédio `temp`, mas mais na execução e sincronização dos comandos `com1` e `com2`. No segundo caso, `com2` só começa a ser executado depois de `com1` ter terminado, ao passo que no primeiro caso os dois comandos são executados duma forma concorrente.

De seguida, vamos ver como abrir uma pipeta (|) dentro dum programa C, abrindo assim um canal de comunicação entre dois processos

Há dois tipos de pipetas: (1) pipetas não-formatadas (2) pipetas formatadas

Também existem dois tipos de pipetas dependendo como os processos que vão utilizar as pipetas foram criados. ***Sem-Nome*** no caso erem criados via fork() e ***Com-Nome*** quando são criados independentemente.

Pipetas não-formatadas (ou de baixo-nível)

Uma pipeta não-formatada não tem nome. Ao contrário dos ficheiros normais, uma pipeta sem nome tem dois (2) descritores de ficheiros associados, e não 1 descritor de ficheiro. Uma pipeta não-formatada é criada com a seguinte função ou chamada ao sistema:

```
#include <unistd.h>

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

Retorna: 0 se OK, -1 em caso de erro. Argumento de retorno: 2 descritores de ficheiros no array *filedes*, com *filedes[0]* aberto para leitura e *filedes[1]* aberto para escrita.

Exemplo 11.4:
(see p.102 de [2] da bibliografia; compilação c/ **libnet.a**)
Se não vai usar a biblioteca do libnet.a : #define err_sys(s) fprintf(stderr,s)

O seguinte programa ilustra a criação duma pipeta através da chamada ao sistema **pipe**. Esta função devolve dois ficheiros (abertos) sem nome, identificados pelos descritores (inteiros) armazenados no array *pipefd*. Note-se que este é um caso particular de comunicação dum processo com ele próprio (Figura 11.1).

```
#include <unistd.h>
#include <stdio.h>
#define err_sys(s) fprintf(stderr,s)
main()
{
        int          pipefd[2], n;
        char          buff[100];

        if (pipe(pipefd) < 0)   err_sys("pipe error");

        printf("read fd = %d, write fd = %d ", pipefd[0], pipefd[1]);
        if (write(pipefd[1], "hello world\n", 12) != 12)
                err_sys("write error");

        if ( (n = read(pipefd[0], buff, sizeof(buff))) <= 0 )
                err_sys("read error");

        write(1, buff, n); /* fd 1 = stdout*/
}
```

A saída deste programa é o seguinte:

```
hello world
read fd = 3,  write fd = 4
```

Note-se que a cadeia de caracteres "hello world" pode ser impressa na saída estandardizada (ecrã) antes da saída da instrução `printf`. Isto acontece porque a saída de `printf` é *buffered* pela biblioteca estandardizada de I/O, e portanto a sua saída não chega para preencher o buffer I/O (que é tipicamente 512 ou 1024 bytes). Isto quer dizer que o buffer não é descarregado até o processo terminar. Contudo, a função `write` não é buffered, e portanto a saída é imediata.

O diagrama duma pipeta associada a um único processo é representado na Figura 11.1.

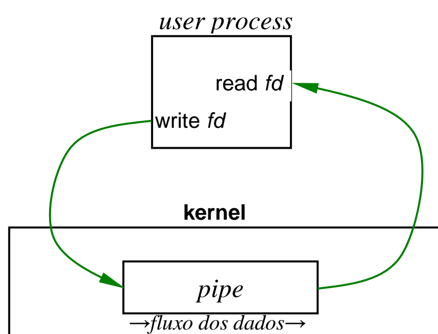

Figura 11.1: Pipeta associada a um só processo.

Mas, as pipetas são de pouca utilidade quando associadas a um único processo. De facto, as pipetas são normalmente usadas para estabelecer a comunicação entre **dois processos** como se segue:

1. Um processo (progenitor) cria uma pipeta;
2. O progenitor faz a chamada ao sistema `fork` para criar uma cópia dele próprio, *i.e.* o processo progénito (Figura 11.2);
3. O progenitor fecha a extremidade de leitura (read) da pipeta e o progénito fecha a extremidade de escrita (write) da pipeta.

Deste modo, estabelece-se um canal IPC unidireccional de dados entre os dois processos (Figura 11.3). Note-se que o que acontece após o `fork` depende do sentido do fluxo de dados que se pretende. Se o sentido fosse do progénito para o progenitor, o progénito fechava a extremidade de leitura e o progenitor fechava a extremidade de escrita da pipeta.

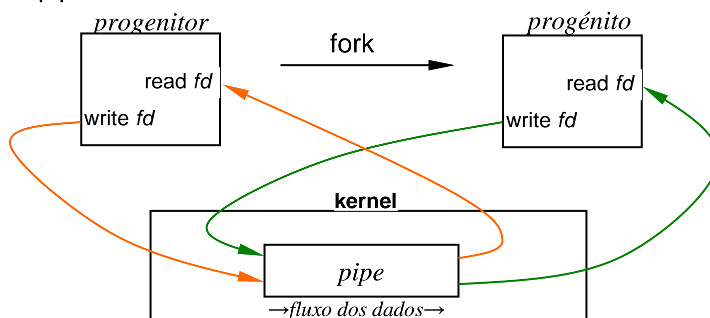

Figura 11.2: Pipeta entre processos, imediatamente após `fork`.

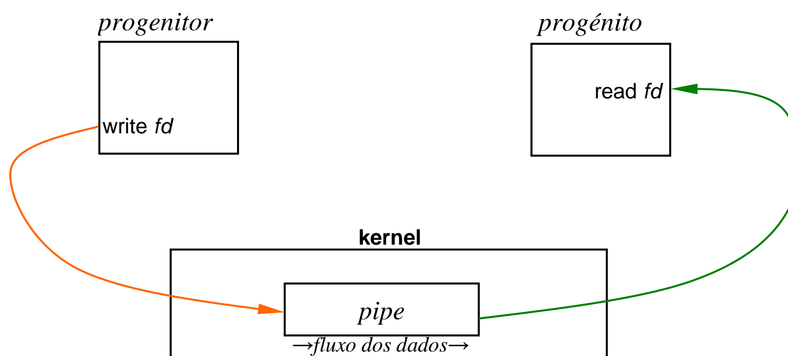

Figura 11.3: Comunicação uni-direccional via pipeta do progenitor para o progénito.

Quando uma extremidade duma pipeta é fechada, as seguintes regras são válidas:

1. Se se ler (`read`) duma pipeta cuja extremidade de escrita foi fechada, após todos os dados terem sido lidos, `read` devolve 0 para indicar o fim do ficheiro.
2. Se se escrever (write) para uma pipeta cuja extremidade de leitura foi fechada, o sinal SIGPIPE é gerado.

<u>Exemplo 11.5</u>:
Envio de dados do progenitor para o progénito através duma pipeta (Figura 11.3).
(see p.431 de [1] da bibliografia)
Se não vai usar a biblioteca do libnet.a : #define err_sys(s) {fprintf(stderr,s); return 0;}

```
#include <unistd.h>
#include <stdio.h>
#define err_sys(s) {fprintf(stderr,s); return 0;}

#define MAXLINE 4096

int main(void)
{
        int     n, fd[2];
        pid_t   pid;
        char    line[MAXLINE];

        if ( pipe(fd) < 0 )  err_sys("pipe error");

        if ( (pid = fork() ) < 0)  err_sys("fork error");

        if (pid > 0) {                          /* progenitor - parent */
                close(fd[0]);                       /* desnecessario */
                write(fd[1], "Hello world\n", 12);      /* escrever para o progénito */
         }
         else  {                                /* progénito – child*/
                close(fd[1]);                       /* desnecessario */
                n = read(fd[0], line, MAXLINE);         /* ler do progenitor */
                write(STDOUT_FILENO, line, n);
        }
}
```

Quando um utilizador digita o seguinte comando

```
who | sort | lpr
```

na linha de comando do Unix, acontece que são criados três processos e duas pipetas entre eles:

- `who` é um programa que faz a saída dos *login names*, *terminal names* e *login times* de todos os utilizadores activos no sistema;
- `sort` é um programa que ordena a lista anterior pelos login names;
- `lpr` é um programa que envia a lista ordenada anterior para a impressora.

Este pipeline está representado na Figura 11.4.

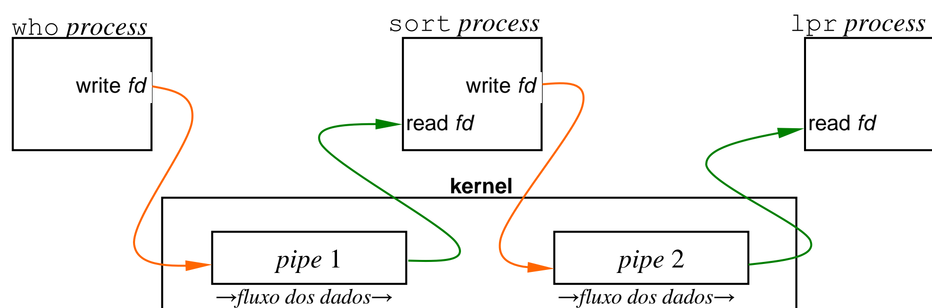

Figura 11.4: Duas pipetas entre três processos.

<u>Exercício :</u>

Usando o exemplo 11.5 como base elabora um programa (mypipe.c) que utilize uma pipeta e o processo fork() para fazer o simulacro do comando do bash "ls | sort "

O processo progenitor executar o programa "sort" usando a chamada execvp() e o processo progénito deverá executar o comando "ls".

Nota que terá de fechar o STDIN/STDOUT_FILENO do processo após o fork() e fazer a duplicação (dup) do descritor do ficheiro do pipeta de leitura/escrita e depois fechar os descritores do pipeta antes de chamar a função execvp()

- Create Pipe
- Fork
- if (child) close stdout file descriptor, dup , close pipe file descriptors, execvp ls
- else close stdin file descriptor, dup , close pipe file descriptors, execvp sort

Pipetas Bidireccionais

Note-se que todas as pipetas criadas até agora são todas unidireccionais. Quando se pretende um canal bidireccional entre dois processos temos que criar duas pipetas, uma para cada sentido. Isto requer os seguintes passos (Figura 11.5):

- Criar pipeta 1 e pipeta 2;
- Bifurcação de processos (fork);
- Progenitor fecha a extremidade de leitura da pipeta 1;
- Progenitor fecha a extremidade de escrita da pipeta 2;
- Progénito fecha a extremidade de escrita da pipeta 1;
- Progénito fecha a extremidade de leitura da pipeta 2.

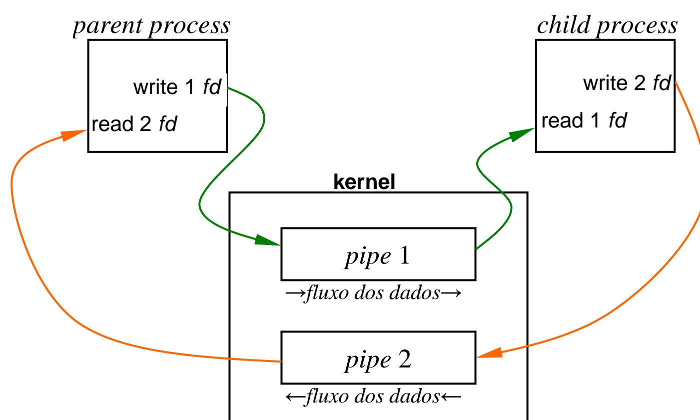

Figura 11.5: Duas pipetas constituintes dum canal bidireccional.

Exemplo 11.6: (Cliente-Servidor)
(see p.431 de [1] da bibliografia; compilação c/ **$ make s_pipe** na directoria ~/stevens.net/ipc)

Em alternativa pode obter o código do gitlab do Professor disponibilizado nas aulas
cd sisops ; cd ipc

Pretende-se estabelecer a comunicação bidireccional entre um cliente e um servidor através de pipetas (Figura 11.6). O cliente lê o nome dum ficheiro a partir da entrada estandardizada, e de seguida envia-o para o canal IPC. Por sua vez, o servidor lê o tal nome do ficheiro a partir do canal IPC, e tenta depois abrir o ficheiro no modo de leitura. Se o servidor consegue abrir o ficheiro, o ficheiro é lido e depois escrito para o canal IPC; caso contrário, o servidor responde com uma mensagem de erro. Então, o cliente lê a partir do canal IPC, escrevendo para a saída estandardizada o que recebe do canal IPC. Se o ficheiro não pode ser aberto pelo servidor, o cliente lê uma mensagem de erro do canal IPC; caso contrário, o cliente lê o conteúdo do ficheiro.

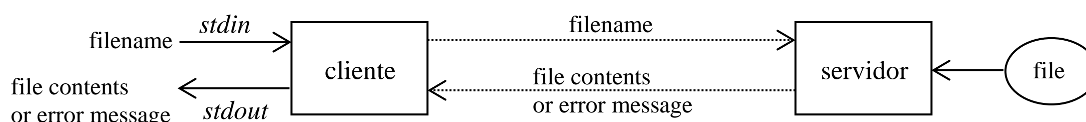

Figura 11.6: Exemplo cliente-servidor.

A função `main` cria a pipeta e o processo progénito através de `fork`. O cliente corre no progenitor e o servidor é executado no progénito.

A compilação do programa está descrita no ficheiro **Makefile** da directoria $CURSO/stevens/unp/ipc como se segue:

```
s_pipe: mainpipefork.c sub_clifd.c sub_servfd.c $(MYLIB)
	cc $(CFLAGS) -o mainpipe mainpipefork.c sub_clifd.c sub_servfd.c $(MYLIB)
```

onde $(MYLIB) é a biblioteca libnet.a.
Como se pode ver, este programa envolve três ficheiros:

- mainpipefork.c - que contém o programa principal;
- sub_clifd.c - que contém a função client;
- sub_servfd.c - que contém a função server.

Alternativamente para quem não quer usar a biblioteca do Stevens

- mainpipefork.c - que contém o programa principal;
- sub_clifd.c - que contém a função client;
- sub_servfd.c - que contém a função server.
- error.c -que contem funções auxiliares e de error
- cs.h -que contenha os ficheiros de inclusão, protótipos e constantes
- Makefile -para gerir o Projecto

O programa principal é o seguinte:

mainpipefork.c

```
#include "cs.h"

main()
{
        int childpid, pipe1[2], pipe2[2];

        if (pipe(pipe1) < 0 || pipe(pipe2) < 0)
                err_sys("Can't create pipes! ");

        if ( (childpid = fork()) < 0 )
                err_sys("can't fork! ");

        if ( childpid > 0 )                    /* parent */
        {
                close(pipe1[0]);
                close(pipe2[1]);

                client(pipe2[0],pipe1[1]);

                while (wait((int *) 0) != childpid)         /* wait for child to finish*/
                        ;

                close(pipe1[1]);
                close(pipe2[0]);
                exit(0);
        }
        else                                   /* child */
        {
                close(pipe1[1]);
                close(pipe2[0]);

                server(pipe1[0],pipe2[1]);

                close(pipe1[0]);
                close(pipe2[1]);
                exit(0);

        }
}
```

A função `server` encontra-se no seguinte ficheiro:

sub_servfd.c

```
#include "cs.h"

int server(int readfd, int writefd)
{
        char    buff[MAXBUFF];
        char    errmesg[256];
        int     n, fd;
        extern int errno;

        /* Read the filename from the IPC descriptor */
        if ( (n = read(readfd, buff, MAXBUFF)) != MAXBUFF )
                err_sys("Server: filename read error!");

        if ( (fd = open(buff, 0)) <  0 )
        {
                /* Error. Format an error message and send it back to the client. */
                sprintf(errmesg, ": can't open, %s\n", sys_err_str());
                strcat(buff, errmesg);
                n = strlen(buff);
                if (write(writefd, buff, n) != n)
                        err_sys("Server: errmesg write error! ");
        }
        else
        {
                /* Read the data from the file and write to the IPC descriptor. */
                while ( (n = read(fd, buff, MAXBUFF)) > 0 )
                        if (write(writefd, buff, n) != n)
                                err_sys("Server: data write error! ");
                if (n < 0)
                        err_sys("Server: read error! ");
        }
}
```

A função `client` encontra-se no seguinte ficheiro:

sub_clifd.c

```
#include "cs.h"

int client(int readfd, int writefd)
{
	char	buff[MAXBUFF];
	int	n;

	/* Read the filename from the standard input, write it to the IPC descriptor */
	if (fgets(buff, MAXBUFF, stdin) == NULL)
		err_sys("Client: filename read error!");

	n = strlen(buff);
	if (buff[n-1] == '\n')
		buff[n-1]='\0';				/* ignore newline from fgets() */
	if (write(writefd, buff, MAXBUFF) != MAXBUFF)
		err_sys("Client: filename write error!");

	/* Read the data from the IPC descriptor and write to standard output */
	while ( (n = read(readfd, buff, MAXBUFF)) > 0 )
		if (write(1, buff, n) != n)		/* fd 1 = stdout */
			err_sys("Client: data write error!");
	if (n < 0)
		err_sys("Client: data read error!");
}
```

Makefile

```
CC = cc
OBJS = error.o mainpipefork.o sub_clifd.o sub_servfd.o
fs_pipe : $(OBJS) cs.h
	$(CC) -o s_pipe $(OBJS)
clean :
	rm fs_pipe $(OBJS)
```

As funções de erro, constantes e protótipos encontram-se nos seguintes ficheiros:

error.c

```
#include "cs.h"
extern int errno;
char * sys_err_str( void ) {  //return string representation of last system call error
   return strerror(errno);     //can add extra information to this function
}
```

Nota do Programador: A principal diferença entre as funções **perror(char *s)** e **strerror(int errnum)** em C está na forma como elas exibem mensagens de erro. A função perror imprime diretamente uma mensagem de erro no **stderr**, combinando um texto fornecido pelo programador com a descrição do erro baseada no valor de errno, a função strerror retorna uma **string** com a descrição do erro correspondente ao código passado (geralmente errno), permitindo ao programador formatar ou manipular a mensagem como quiser antes de exibi-la, por exemplo para ficheiros de Log ou GUI's etc.

cs.h

```
#include <unistd.h>
#include <stdio.h>
#include <stdlib.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>
#include <errno.h>
#include <unistd.h>
#include <sys/wait.h>
#include <string.h>

#define err_sys(STR)    fprintf(stderr,STR)
#define err_sys2(STR,V) fprintf(stderr,STR,V)

#define MAXBUFF 1024

char *sys_err_str(void);

int client  (int readPipe , int  writePipe);
int server (int readPipe , int  writePipe);
```

Exercícios:

1) Insere um "prompt" no cliente para pedir o nome do ficheiro

2) No servidor não permite que sejam pedidos ficheiros cujo nomes contêm caminhos relativos (baste utilizar a função strstr e pesquisa se o nome do ficheiro contenha o string "..") nem ficheiros que começam com os caracteres ´/' ou '\'

3) (Difícil) Escreva um cliente e servidor de ficheiros onde ficheiros podem ser pedidos e servidos até que no cliente seja introduzido a palavra "sair"

No protocolo original quando o server envie "zero bytes" e quando o cliente recebe "zero bytes" temos o fim do processo. Portanto - Sugestão mudar o protocolo de transmissão dos dados de ficheiro

Novo Protocolo:

C-S -> Nome
S-C -> Pacote
Pacote = Um Block de 4bytes(tamanho do Payload) +payload (BUFFSIZE)
Assim - Tamanho do payload =0 → fim do processo de transmissão.

Exemplo para o ficheiro "test.txt" que tem 8 bytes "123345678"

C-S → test.txt
S-C → Pacote = 12 bytes um inteiro de 4 bytes = 8 + "12345678"
S-C → Pacote = 4 bytes um inteiro de 4 bytes = 0

Nota que o cliente em vez de fazer um read terá que fazer dois !!

Pipetas formatadas (alto-nível)

A biblioteca estandardizada de I/O (`stdio.h`) fornece uma função, designada por **popen**, que permite criar uma pipeta e um novo processo que ou lê ou escreve da/para a pipeta.

```
#include <unistd.h>

FILE *popen(char *command, const char *type);
```

Retorna: file pointer se OK, NULL em caso de erro.

```
int pclose(FILE *fp);
```

Retorna: estado de terminação de `command`, ou -1 em caso de erro.

A função **popen** abre uma pipeta para I/O, em que *command* é o processo que será ligado ao processo invocador. Esta função faz um **fork** e um **exec** para executar *command*, e devolve um ponteiro para um ficheiro I/O estandardizado. Se *type* é "r", o ponteiro é ligado à saída estandardizada de command (Figura 11.7). Se type é "w", o ponteiro é ligado à entrada estandardizada de command (Figura 6.8).
A função **pclose** fecha o stream I/O estandardizado e espera pela terminação do comando.

Figura 11.7: Resultado de fp = popen(*command*, "r").

Figura 11.8: Resultado de fp = popen(*command*, "w").

Exemplo 11.7: (Cliente-Servidor)
As funções `popen` e `pclose`, conjuntamente com o comando `cat`, podem ser usadas para implementar o nosso exemplo anterior de comunicação bidireccional entre um processo cliente e um processo servidor.

mainpopen.c

```
#include <stdio.h>

#define MAXLINE1024

main()
{
	int	n;
	char	line[MAXLINE], command[MAXLINE + 10];
	FILE	*fp;

	/* Read the filename from standard input*/
	if (fgets(line, MAXLINE, stdin) == NULL)
		err_sys("filename read error! ");

	/* Use popen to create a pipe and execute the command. */
	sprintf(command, "cat %s", line);
	if ( (fp = popen(command, "r")) == NULL )
		err_sys("popen error");
	/* Read the data from the FILE pointer and write to standard output. */
	while ( (fgets(line, MAXLINE, fp)) != NULL )
	{
		n = strlen(line);
		if (write(1,line,n) != n)
			err_sys("data write error! ");
	}

	if ( ferror(fp) )
		err_sys("fgets error! ");

	pclose(fp);
	exit(0);
}
```

A compilação do programa está descrita no **Makefile** da directoria $CURSO/stevens/unp/ipc como se segue:

```
mainpopen:	mainpopen.c $(MYLIB)
	cc $(CFLAGS) -o mainpopen mainpopen.c $(MYLIB)
```

onde $(MYLIB) é a biblioteca libnet.a.
Como se pode ver, este programa envolve um só ficheiro:

- mainpopen.c - que contém o programa principal.

Exemplo 11.8:
Escreva um programa que obtenha a directoria corrente (através do comando `pwd`) e que, depois, a escreva no ecrã.

getpwd.c

```
#include <stdio.h>

#define         MAXLINE         255

main()
{
        FILE    *fp;
        char    line[MAXLINE];

        if ( (fp = popen("/bin/pwd", "r")) == NULL )
                err_sys("popen error! ");

        if (fgets(line, MAXLINE, fp) == NULL)
                err_sys("fgets error! ");

        printf("%s", line);         /* pwd inserts the newline*/

        pclose(fp);
        exit(0);
}
```

A compilação do programa está descrita no ficheiro **Makefile** da directoria $CURSO/stevens/unp/ipc como se segue:

```
getpwd:     getpwd.o $(MYLIB)
        cc $(CFLAGS) -o getpwd getpwd.o $(MYLIB)
```

onde $(MYLIB) é a biblioteca libnet.a.
Como se pode ver, este programa envolve um só ficheiro:

- getpwd.c - que contém o programa principal;

A grande desvantagem das pipetas é que só podem ser usadas entre processos que têm um processo progenitor em comum. Isto acontece porque uma pipeta é passada dum processo para outro através da chamada ao sistema `fork`, e também porque todos os ficheiros abertos são partilhados entre o processo progenitor e o processo progénito após o `fork`. Não há qualquer possibilidade de criar uma pipeta entre dois processos independentes ou não-relacionados.

FIFOs (pipetas c/ nome)

As FIFOs são também designadas por pipetas com nome. O fluxo dos dados também é unidireccional, mas as FIFOs têm um identificador alfanumérico ou nome.

Para criar/utilzar named pipes no bash sehll ver: http://www.linuxjournal.com/article/2156

Para utilizar programaticamente vamos ver o named pipe API.

```
#include <sys/types.h>
#include <sys/stat.h>

int mknod(char *pathname, int mode, int dev);
int mkfifo(char *pathname, mode_t mode);
```

Retorna: 0 se OK, -1 em caso de erro.

As duas funções anteriores podem ser usadas indistintamente para criar uma FIFO, pois a função **mkfifo** chama sempre a função **mknod**. A função mknod é mais de baixo-nível, e serve para criar não só FIFOs, mas também para criar e montar dispositivos no sistema. Isto significa que o valor da variável *dev* é ignorado no caso da criação duma FIFO.
A variável *pathname* indica um caminho da árvore de ficheiros que não é mais do que o nome da FIFO. O argumento *mode* especifica o acesso à FIFO. Note-se que uma FIFO não é mais do que um ficheiro. Portanto, o acesso a uma FIFO é também definido pelas permissões de *read* and *write* para o proprietário, o grupo, e restantes utilizadores.

As pipetas e as FIFOs seguem as seguintes regras (e mais algumas) de leitura e escrita:

- Se num acesso de leitura forem solicitados MENOS dados do que aqueles que estão efectivamente na pipeta/FIFO, os dados restantes são lá deixados para subsequentes leituras.
- Se num acesso de leitura forem solicitados MAIS dados do que aqueles que estão efectivamente na pipeta/FIFO, só os dados que estão disponíveis são devolvidos.
- Se um processo debita dados e não esgota a capacidade da pipeta/FIFO (a qual tem pelo menos 4096 bytes), a escrita é garantidamente atómica. Isto significa que se dois processos iniciam a escrita de dados para a pipeta/FIFO mais ou menos na mesma altura, então há a garantia de que os dados de um dos processos são escritos após todos os dados do outro processo terem sido escritos. Não há mistura de dados dos dois processos na pipeta/FIFO. No entanto, se o débito dum dos processos ultrapassa a capacidade da pipeta/FIFO, então não há garantia de a operação de escrita ser atómica.

No Exemplo 11.6 usámos pipetas para fazer a comunicação entre um processo servidor e um processo cliente, em que um dos processos era filho do outro. De facto, as pipetas não podem ser partilhadas por processos independentes. Mas, com as FIFOs não há esta restrição.

Vejamos então o mesmo exemplo cliente-servidor implementado com dois processos independentes que partilham duas FIFOs como canal IPC de comunicação bidireccional:

Exemplo 11.9: (Cliente-servidor através de FIFOs) A compilação do programa está descrita no ficheiro **Makefile** da directoria $CURSO/stevens/unp/ipc

Alternativamente ao livro do Stevens pode-se usar o Makefile e procedimento a seguir

```
s_fifo:        s_fifoserv s_fifocli

s_fifoserv:    mainfifoserv.c sub_servfd.c error.c
        cc $(CFLAGS) -o server mainfifoserv.c sub_servfd.c error.c

s_fifocli:     mainfifocli.c  sub_clifd.c error.c
        cc $(CFLAGS) -o client mainfifocli.c sub_clifd.c error.c
```

Como se pode ver, este programa envolve os seguintes ficheiros:

- mainfifoserv.c - que contém o programa principal do processo servidor;
- sub_servfd.c - que contém a função **server**;
- mainfifocli.c - que contém o programa principal do processo cliente;
- sub_clifd.c - que contém a função **client**.
- error.c -que contém funções de erro
- cs.h - que contêm definições comuns
- fifo.h - que contêm definições comuns relativos aos ficheiros dos pipes.

O programa principal do **cliente** é o seguinte:

mainfifocli.c

```
#include  "cs.h"
#include  "fifo.h"

int main()
{
      int   readfd, writefd;
      umask(0);

       /* Open the FIFOs.  We assume the server has already created them. */

      if ( (writefd = open(FIFO1, 1)) < 0)
            err_sys2("client: can't open write fifo: %s", FIFO1);
      if ( (readfd = open(FIFO2, 0)) < 0)
            err_sys2("client: can't open read fifo: %s", FIFO2);

      client(readfd, writefd);

      close(readfd);close(writefd);

      /* Delete the FIFOs, now that we're finished */

      if (unlink(FIFO1) < 0)
            err_sys2("client: can't unlink %s", FIFO1);
      if (unlink(FIFO2) < 0)
            err_sys2("client: can't unlink %s", FIFO2);

      return(0);
}
```

O ficheiro `fifo.h` tem o conteúdo seguinte:

fifo.h

```
#include    <sys/types.h>
#include    <sys/stat.h>
#include    <sys/errno.h>
extern int  errno;

#define     FIFO1 "/tmp/fifo.1"
#define     FIFO2 "/tmp/fifo.2"
#define     PERMS 0666
```

O programa principal do **servidor** é o seguinte:

mainfifoserv.c

```
#include    "cs.h"
#include    "fifo.h"

int main()
{
      int   readfd, writefd;

      /* Create the FIFOs, then open them - one for reading and one for writing.
       */
      umask(0);

      if ( (mknod(FIFO1, S_IFIFO | PERMS, 0) < 0) && (errno != EEXIST))
            err_sys2("can't create fifo: %s", FIFO1);
      if ( (mknod(FIFO2, S_IFIFO | PERMS, 0) < 0) && (errno != EEXIST)) {
            unlink(FIFO1);
            err_sys2("can't create fifo: %s", FIFO2);
      }

      if ( (readfd = open(FIFO1, 0)) < 0)
            err_sys2("server: can't open read fifo: %s", FIFO1);
      if ( (writefd = open(FIFO2, 1)) < 0)
            err_sys2("server: can't open write fifo: %s", FIFO2);

      server(readfd, writefd);

      close(readfd);
      close(writefd);

      return(0);
}
```

O processo servidor (chamado `server`) deve iniciar a sua execução em primeiro lugar e em segundo plano (ou *background*) da seguinte forma. Se executar em primeiro plano (foreground) o cliente terá de ser executado num outro terminal/consola.

```
$ server &
```

O processo cliente (chamado `client`) deve ser executado de seguida em primeiro plano (visto que o cliente lê da entrada estandardizada e escreve para a saída estandardizada):

```
$ client
```

Filas de mensagens

A comunicação através de mensagens é agora usual nos sistemas operativos modernos.

Todas as mensagens são armazenadas no núcleo (kernel) do sistema operativo, cada uma das quais inclui o identificador da fila a que está associada.

Os processos lêem e escrevem mensagens de e para filas arbitrárias. Não existe qualquer requisito que impeça que um processo que se encontra à espera duma mensagem específica não possa escrever uma mensagem para a mesma fila. Isto contrasta com as pipetas e as FIFOs, para as quais não faz sentido ter um processo escritor a não ser que exista um processo leitor. É bem possível que um processo escreva algumas mensagens para a fila, termine a escrita e continue a executar outras tarefas, e só mais tarde as ditas mensagens serem lidas por outro processo.

Qualquer mensagem tem os seguintes atributos:

- *type* (long int)
- *length* (comprimento dos dados da mensagem)
- *data* (se length não é nulo)

```
typedef struct { long int mesg_type;
                 char mesg_data[MAXMESGDATA]; int mesg_len;  }  Mesg;
```

Qualquer fila de mensagens tem a seguinte estrutura:

```
#include <sys/types.h>
#include <sys/ipc.h>          /* defines the ipc_perm structure*/

struct msqid_ds {
      struct      ipc_perm    msg_perm;    /* op. permission struct */
      struct      msg         *msg_first;  /* ptr to first message on q*/
      struct      msg         *msg_last;   /* ptr to last message on q*/
      ushort                  msg_cbytes;  /* current no. bytes on q */
      ushort                  msg_qnum;    /* current no. messages on q */
      ushort                  msg_qbytes;  /* max no. bytes allowed on q */
      ushort                  msg_lspid;   /* pid of last msgsnd*/
      ushort                  msg_lrpid;   /* pid of last msgrcv*/
      time_t                  msg_stime;   /* time of last msgsnd*/
      time_t                  msg_rtime;   /* time of last msgrcv*/
      time_t                  msg_ctime;   /* pid of last msgctl*/
};
```

A estrutura `ipc_perm` contém as permissões de acesso a uma fila de mensagens. Cada estrutura `msg` define uma mensagem numa lista ligada em que `msg_first` e `msg_last` são apontadores para a primeira e última mensagens, respectivamente.

Por exemplo, na Figura 11.9 representa-se uma fila com três mensagens cujos comprimentos são de 1 byte, 2 bytes e 3 bytes, respectivamente. Os tipos das mensagens são 100, 200 e 300, respectivamente.

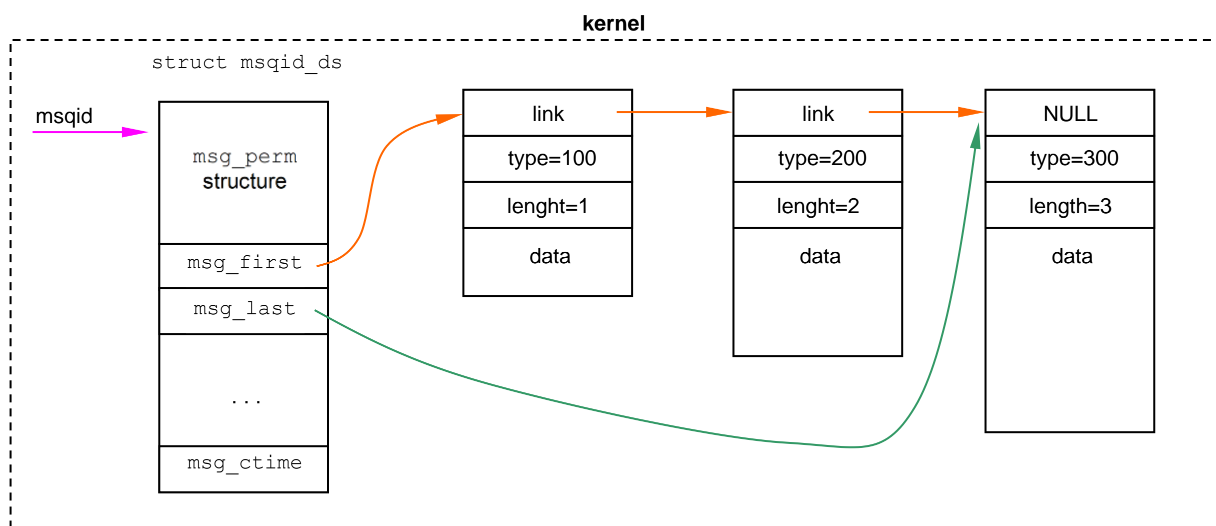

Figura 11.9: Fila de mensagens no núcleo do sistema operativo.

Criação/Acesso a fila de mensagens

Uma fila de mensagens é criada (ou acedida) através da seguinte chamada ao sistema:

```
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/msg.h>

int msgget(key_t key, int msgflag);
```

Retorna: identificador da fila de mensagens, -1 em caso de erro.

O valor de *msgdflag* é uma combinação das constantes da Figura 11.10, as quais definem o modo de acesso à fila.

Numérico	Simbólico	Descrição
0400	MSG_R	Read by owner
0200	MSG_W	Write by owner
0040	MSG_R >> 3	Read by group
0020	MSG_W >> 3	Write by group
0004	MSG_R >> 6	Read by world
0002	MSG_W >> 6	Write by world
	IPC_CREAT	
	IPC_EXCL	

Figura 11.10: Valores de *msgflag* na chamada de `msgget`.

Emissão de mensagem
O envio duma mensagem para uma fila de mensagens é feita através da seguinte chamada ao sistema:

```
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/msg.h>

int msgsnd(int msqid, struct msgbuf *ptr, int length, int flag);
```

Retorna: 0 se OK, -1 em caso de erro.

Expliquemos os argumentos da função anterior:

- *msqid*. É o identificador da fila de mensagens.
- *ptr*. É um endereçador duma estrutura `msgbuf` (veja-se `<sys/msg.h>`) que contém dois campos: o primeiro `mtype` indica o tipo da mensagem, e o segundo `mtext` é um array que contém os dados da mensagem. Os dados podem ser textuais ou binários.
- *length*. Especifica o comprimento da mensagem em bytes. O comprimento pode ser zero.
- *flag*. Toma os valores `IPC_WAIT` ou zero. `IPC_WAIT` permite o retorno imediato da função no caso de não existir espaço na fila para a nova mensagem. Se não existe espaço na fila, e se `IPC_WAIT` é especificado, `msgsnd` devolve o valor –1 e `errno` toma o valor `EAGAIN`. Se esta chamada ao sistema é feita com sucesso, ela devolverá o valor zero.

Recepção de mensagem
Uma mensagem é lida a partir duma fila de mensagens através da seguinte chamada ao sistema:

```
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/msg.h>

int msgrcv(int msqid, struct msgbuf *ptr, int length, long msgtype, int flag);
```

Retorna: tamanho da porção de dados da mensagem se OK, -1 em caso de erro.

- *msqid*. Identifica a fila de mensagens.
- *ptr*. É um endereçador duma estrutura `msgbuf` (veja-se `<sys/msg.h>`) que contém a mensagem.
- *length*. Especifica o comprimento da mensagem em bytes.
- *msgtype*. Especifica qual é a mensagem que se pretende ler da fila:
  - Se $msgtype=0$, a primeira mensagem da fila é devolvida. Dado que a fila segue uma política FIFO de escalonamento, a mensagem mais antiga é devolvida.
  - Se $msgtype>0$, a primeira mensagem com o tipo igual a *msgtype* é devolvida.
  - Se $msgtype<0$, a primeira mensagem com o tipo mais baixo que é menor ou igual ao valor absoluto de *msgtype* é devolvida.
- *flag*. Especifica o que fazer no caso duma mensagem dum dado tipo não se encontrar na fila. Se o bit `IPC_WAIT` estiver activo, `msgrcv` faz retorno imediato no caso de a mensagem não estar disponível. Neste caso, `msgrcv` devolve o valor –1 e `errno` toma o valor `ENOMSG`. Caso contrário, o processo invocador é suspendido até um dos seguintes eventos ocorra:
  - Uma mensagem do tipo pedido está disponível.
  - A fila de mensagens é removida do sistema.
  - O processo recebe um sinal que é capturado.

Se tudo correr bem, `msgrcv` devolve o número de bytes dos dados existentes na mensagem recebida.

Controlo da fila de mensagens

As operações de controlo sobre a fila de mensagens são feitas através da seguinte chamada ao sistema:

```
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/msg.h>

int msgctl(int msqid, int cmd, struct msqid_ds *buff);
```

Retorna: 0 se OK, -1 em caso de erro.

No exemplo que se segue só usaremos o valor `IPC_RMID` em *cmd* para remover uma fila de mensagens do sistema operativo.

Exemplo 11.10: (Cliente-Servidor através de 2 filas de mensagens)
A compilação do programa está descrita no **Makefile** da directoria $CURSO/stevens/unp/ipc como se segue:

```
tudo:        filas_server  fila_cliente

file_server:     mainmsgqserv.c sub_servmesg.c sub_mesgmsgq.c $(MYLIB)
        cc $(CFLAGS) -o server mainmsgqserv.o sub_servmesg.o sub_mesgmsgq.o $(MYLIB)

fila_cliente:     mainmsgqcli.c sub_climesg.c sub_mesgmsgq.c $(MYLIB)
        cc $(CFLAGS) -o client mainmsgqcli.o sub_climesg.o sub_mesgmsgq.o $(MYLIB)
```

onde $(MYLIB) é a biblioteca libnet.a.
Como se pode ver, este programa envolve os seguintes ficheiros:

- mainmsgqserv.c- que contém o programa principal do processo servidor;
- sub_servmesg.c- que contém a função **server**;
- mainmsgqcli.c - que contém o programa principal do processo cliente;
- sub_climesg.c - que contém a função **client**;
- sub_mesgmsgq.c – que contêm rotinas comuns de envio e receber

O programa principal do **servidor** é o seguinte:

mainmsgqserv.c

```
#include    "msgq.h"

main()
{
        int     readid, writeid;

        /* Create the message queues, if required. */

        if ( (readid = msgget(MKEY1, PERMS | IPC_CREAT)) < 0)
                err_sys("server: can't get message queue 1");
        if ( (writeid = msgget(MKEY2, PERMS | IPC_CREAT)) < 0)
                err_sys("server: can't get message queue 2");

        server(readid, writeid);

        exit(0);
}
```

O programa principal do **cliente** é o seguinte:

mainmsgqcli.c

```
#include    "msgq.h"

main()
{
      int   readid, writeid;

      /*
       * Open the message queues.  The server must have
       * already created them.
       */

      if ( (writeid = msgget(MKEY1, 0)) < 0)
            err_sys("client: can't msgget message queue 1");
      if ( (readid = msgget(MKEY2, 0)) < 0)
            err_sys("client: can't msgget message queue 2");

      client(readid, writeid);

      /*
       * Now we can delete the message queues.
       */

      if (msgctl(readid, IPC_RMID, (struct msqid_ds *) 0) < 0)
            err_sys("client: can't RMID message queue 1");
      if (msgctl(writeid, IPC_RMID, (struct msqid_ds *) 0) < 0)
            err_sys("client: can't RMID message queue 2");

      exit(0);
}
```

A função `server` é agora a seguinte:

sub_servmesg.c

```
#include    <stdio.h>
#include    "mesg.h"

extern int  errno;
Mesg  mesg;

int server(int ipcreadfd, int ipcwritefd)
{
      int   n, filefd;
      char  errmesg[256], *sys_err_str();

      /* Read the filename message from the IPC descriptor. */

      mesg.mesg_type = 1L;
      if ( (n = mesg_recv(ipcreadfd, &mesg)) <= 0)
            err_sys("server: filename read error");
      mesg.mesg_data[n] = '\0';     /* null terminate filename */

      if ( (filefd = open(mesg.mesg_data, 0)) < 0) {
            /*
             * Error.  Format an error message and send it back to the client.
             */
        sprintf(errmesg, ": can't open, %s\n", sys_err_str());
            strcat(mesg.mesg_data, errmesg);
            mesg.mesg_len = strlen(mesg.mesg_data);
            mesg_send(ipcwritefd, &mesg);

      } else {
             /* Read the data from the file and send a message to
                the IPC descriptor.
             */
            while ( (n = read(filefd, mesg.mesg_data, MAXMESGDATA)) > 0) {
                  mesg.mesg_len = n;
                  mesg_send(ipcwritefd, &mesg);
            }
            close(filefd);

            if (n < 0)
                  err_sys("server: read error");
      }

      /* Send a message with a length of 0 to signify the end. */

      mesg.mesg_len = 0;
      mesg_send(ipcwritefd, &mesg);
}
```

… ao passo que a função `client` é a seguinte:

sub_climesg.c

```
#include     <stdio.h>
#include     "mesg.h"

Mesg  mesg;

int client(int ipcreadfd, int ipcwritefd)
{
      int   n;

      /*
       * Read the filename from standard input, write it as
       * a message to the IPC descriptor.
       */

      if (fgets(mesg.mesg_data, MAXMESGDATA, stdin) == NULL)
            err_sys("filename read error");

      n = strlen(mesg.mesg_data);
      if (mesg.mesg_data[n-1] == '\n')
            n--;              /* ignore newline from fgets() */
      mesg.mesg_len = n;
      mesg.mesg_type = 1L;
      mesg_send(ipcwritefd, &mesg);

      /*
       * Receive the message from the IPC descriptor and write
       * the data to the standard output.
       */

      while ( (n = mesg_recv(ipcreadfd, &mesg)) > 0)
            if (write(1, mesg.mesg_data, n) != n)
                  err_sys("data write error");

      if (n < 0)
            err_sys("data read error");
}
```

P

Por sua vez, as funções `server` e `client` invocam as funções `mesg_send` e `mesg_recv` abaixo descritas:

sub_mesgmsgq.c

```
#include    "mesg.h"

/*
 * Send a message using the System V message queues.
 * The mesg_len, mesg_type and mesg_data fields must be filled
 * in by the caller.
 */

mesg_send(int id, Mesg  *mesgptr)
{
      /*
       * Send the message - the type followed by the optional data.
       */

      if (msgsnd(id, (char *) &(mesgptr->mesg_type),
                              mesgptr->mesg_len, 0) != 0)
            err_sys("msgsnd error");
}

/*
 * Receive a message from a System V message queue.
 * The caller must fill in the mesg_type field with the desired type.
 * Return the number of bytes in the data portion of the message.
 * A 0-length data message implies end-of-file.
 */

int mesg_recv(int id, Mesg *mesgptr)
{
      int   n;

      /*
       * Read the first message on the queue of the specified type.
       */

      n = msgrcv(id, (char *) &(mesgptr->mesg_type), MAXMESGDATA,
                              mesgptr->mesg_type, 0);
      if ( (mesgptr->mesg_len = n) < 0)
            err_dump("msgrcv error");

      return(n);        /* n will be 0 at end of file */
}
```

O ficheiro .h

```
#include <unistd.h>
#include <string.h>
#include <stdio.h>
#include <stdlib.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>
#define err_sys(STR) fprintf(stderr,STR)
#define err_sys2(STR,XXX) fprintf(stderr,STR,XXX)
#define err_dump(XXX) fprintf(stderr,XXX)
#include <sys/types.h>
#include <sys/stat.h>
#include <sys/errno.h>
extern int errno;
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/msg.h>

#define MKEY1 0x11
#define MKEY2 0x12
#define PERMS 0666

#define MAXMESGDATA 100

typedef struct { long int mesg_type;
                 char mesg_data[MAXMESGDATA]; int mesg_len;  }  Mesg;

int server(int , int);
int client(int,  int);

void mesg_send (int id, Mesg * mesgptr);
int mesg_recv(int id, Mesg *mesgptr);


/* funcões uteis da bash shell para manipular e gerir filas
ipcs
ipcrm
*/
```