Sistemas Operativos

2004/05

Programação concorrente em ambiente LINUX

Guia das aulas práticas

Versão 1.01 – 19/05/2003

Índice

Introdução

A linguagem C

A linguagem de programação C teve a sua origem no início dos anos 70, nos laboratórios Bell da AT&T (uma grande companhia norte-americana), sendo inicialmente projectada para sistemas Unix. O nome escolhido para a linguagem C teve provavelmente a ver com o facto de, antes do C, os laboratórios Bell terem desenvolvido uma linguagem designada B...

Desde a altura em que surgiu até aos dias que correm, o C tem sido muito utilizado, embora hoje em dia tenha a "concorrência" do C++ e do Java, linguagens de mais alto nível e que permitem a programação orientada para objectos. De qualquer modo, quando as aplicações a desenvolver estão mais próximas do hardware, o C continua a ser a escolha habitual – como prova disso, existem vários sistemas operativos escritos em C, como é o caso do Linux e de grande parte dos Windows 2000 e XP.

Na disciplina de Sistemas Operativos, será utilizada a linguagem C, pois os alunos irão efectuar programação concorrente em ambiente Linux, o que implica muitas vezes a invocação de chamadas ao sistema que se encontram escritas em C.

Principais diferenças em relação ao C++

As linguagens C e C++ apresentam uma sintaxe muito semelhante, mas existe uma grande e fundamental diferença: na linguagem C não existem classes nem programação genérica. Deste modo a programação em C é sempre programação funcional (invocam-se funções e não métodos), sendo difícil ou mesmo impossível concretizar a programação orientada para objectos.

Na linguagem C, a sintaxe para controlo de fluxo de execução (ciclos, condições, etc.), operadores, apontadores, etc., é muito semelhante à da linguagem C++. Deste modo, um programador de C++ consegue adaptar-se com relativa facilidade à sintaxe da linguagem C.

Como consequência da inexistência de classes no C, muitos dos operadores que conhece do C++, tais como *new* e *delete*, *cin* e *cout*, não existem em C e, caso os utilize em programas C, irá obter um erro de compilação.

Existem também diferenças mais ao nível do compilador que, como consequência, trazem diferenças no modo de programar:

- A declaração de variáveis é habitualmente feita no início do corpo cada função (ou contrário do C++, onde se costumam declarar variáveis em qualquer ponto do código, desde que seja antes de as utilizar). Contudo, utilizando na compilação a norma c99, a declaração das variáveis pode ser efectuada de forma semelhante ao que é feito em C++.
- Os comentários começam por /* e terminam por */, podendo ocupar qualquer número de linhas. Não utilize as // para efectuar comentários num programa em C.
- Quando se utiliza a directiva `#include` é necessário colocar a extensão .h nos ficheiros a incluir (e.g., `#include <stdio.h>`)

Existem muitas outras diferenças entre as duas linguagens, e poderíamos passar horas e horas a discuti-las, mas, no que toca à disciplina de Sistemas Operativos, estes são os principais pontos onde irá notar diferenças em relação ao que já sabe do C++.

Compilação de programas e makefiles

Considere o seguinte programa em linguagem C:

```
#include <stdio.h>

int main()
{
   printf("Olá mundo !");
   return 0;
}
```

Como já deve ter adivinhado, o objectivo deste programa é apenas efectuar a escrita de *Olá mundo !* para o écran. No entanto, o objectivo pelo qual aparece aqui este programa tão básico é apenas para se habituar à utilização do compilador. Pode editar este programa e gravá-lo com um nome à sua escolha (aqui vai-se admitir que foi gravado com o nome *xpto.c*).

Para compilar poderá dar o seguinte comando:

- `gcc –ansi –pedantic –Wall –g –c xpto.c`

Deste modo é criado um ficheiro chamado *xpto.o* que contém o código *objecto* relativo à compilação de *xpto.c*. Este ficheiro é binário, mas ainda não é executável. Para criar o executável poderá dar o seguinte comando:

- `gcc –g xpto.o –o xpto`

Pode também criar o executável directamente a partir do código fonte. Para tal poderá dar o seguinte comando:

- `gcc –ansi –pedantic –Wall –g xpto.c –o xpto`

(Este comando deve-lhe ser familiar, pois é utilizado um comando muito semelhante nas disciplinas de IP e POO quando compila a partir do xemacs...)

Um outro modo de compilar é a utilização do utilitário *make*. Quando se compila com recurso ao *make* só se actualizam os ficheiros quando necessário – o *make* compara as datas dos ficheiros a fazer – *targets* – com as datas dos ficheiros dos quais depende cada *target*. Só é feita compilação quando existe um ficheiro na lista de dependências com uma data mais recente do que a do *target* a fazer.

Esta característica é especialmente vantajosa quando se encontra a trabalhar num projecto distribuído por vários módulos. Assim, quando faz uma alteração num único módulo, em vez de compilar todo o projecto, utilizando o *make* só são efectuadas actualizações (compilações) aos ficheiros que dependem do módulo alterado.

Para efectuar a compilação, o *make* utiliza a informação contida num ficheiro chamado *makefile* (editado pelo programador). Um exemplo de um *makefile* para o programa *xpto* seria a seguinte:

```
# Exemplo de makefile 1

xpto: xpto.o
xpto.o: xpto.c
```

Experimente editar o ficheiro *makefile*, grave-o na mesma directoria onde está o ficheiro *xpto.c* e compile o programa *xpto* fazendo simplesmente

- `make`

ou

- `make xpto`

Provavelmente o compilador utilizado e as opções de compilação não foram as mesmas definidas anteriormente...

Para garantir que o compilador utilizado é o *gcc* e as opções de compilação são as mesmas que as utilizadas anteriormente, basta editar duas variáveis especiais chamadas CC e CFLAGS, que representam o compilador a utilizar e as opções de compilação, respectivamente. O novo *makefile* seria o seguinte:

```
# Exemplo de makefile 2

CC=gcc
CFLAGS=-ansi –pedantic –Wall -g

xpto: xpto.o
xpto.o: xpto.c
```

A estrutura de um *makefile* simples pode-se reduzir-se a duas partes principais: uma parte onde são definidas variáveis e a uma parte onde são definidas regras.

Cada regra é composta por uma linha onde se especifica o designado *target* e os ficheiros do qual depende, seguindo-se um conjunto de linhas iniciadas por uma tabulação (<tab>), onde se escrevem os comandos necessários para fazer o *target*.

Quando os comandos são omitidos, assume-se uma regra implícita, que corresponde ao comando para compilar. Quando *target* não é um ficheiro, só interessam os comandos que vêm de seguida.

Voltando ao exemplo anterior, para a primeira regra *xpto* é o *target* e *xpto.o* o ficheiro do qual depende *xpto*. Na segunda regra *xpto.o* é o *target* e *xpto.c* a dependência.

Quando se faz *make* seguido do nome do *target* só serão seguidas as regras referentes às dependências desse *target*.

Podem ser efectuadas *makefiles* mais complexos quando o programa a efectuar é composto por vários módulos. O seguinte exemplo ilustra essa situação, na qual o programa *prog* é composto pelos módulos *mod1.c*, *mod2.c* e *mod3.c*.

```
# Exemplo de makefile 3

CC=gcc
CFLAGS=-ansi –pedantic –Wall –g     # flags para o compilador
LDFLAGS=-g                    # flags para o linker

objects=mod1.o mod2.o mod3.o        # definição da variável objects

# Esta linha é especial, pois é invocado o linker com vários objectos
# o símbolo $@ é substituído por prog (o “target”)
prog: $(objects)
   $(CC) $(LDFLAGS) –o $@ $(objects)

mod1.o: mod1.c mod2.h mod3.h
mod2.o: mod2.c
mod3.o: mod3.c

clean:
      rm *.o                        # para remover os .o
```

Os standard streams

Na linguagem C, tanto o écran do terminal como o teclado são encarados como ficheiros. Neste contexto, faz sentido que funções de escrita e leitura em ficheiros possam também ser aplicadas a escrita para o écran e a leitura do teclado. Na linguagem C e à semelhança do que acontece na *shell*, podem ser utilizados três canais especiais, designados por *standard streams*:

- ***stdout*** – canal de saída pré definido, por defeito associado ao écran;
- ***stdin*** – canal de entrada pré definido, por defeito associado ao teclado;
- ***stderr* –** canal de saída dos erros, por defeito associado ao écran.

Cada um destes canais pode ser redireccionado, por exemplo redireccionar o *stdout* de um programa para um ficheiro.

Funções de escrita e leitura formatada

As funções de escrita e leitura formatada são utilizadas fazendo a inclusão de *stdio.h*.

- **fprintf** e **printf**. – escrita formatada

```
int fprintf ( FILE *stream, const char *format, ... )
int printf ( const char *format, ... )
```

fprintf escreve no ficheiro *stream*, os argumentos indicados na lista ... , de acordo com o indicado pela cadeia de caracteres *format*. A função pode ser também utilizada para enviar caracteres para o écran, caso o ficheiro *stream* especificado seja *stdout* ou *stderr*.

A função *printf* escreve sempre em *stdout*, sendo preferível a sua utilização quando se pretende escrita de dados no écran. Ambas devolvem o número de caracteres escritos, ou um número negativo em caso de erro.

A cadeia de caracteres *format* pode ser formada por caracteres convencionais e por especificadores de conversão. Os especificadores de conversão começam pelo símbolo de percentagem (%) e são terminados por um caractere de conversão. Cada um dos caracteres de conversão determinam o modo como os argumentos da lista ... são escritos. Como exemplos de caracteres de conversão tem-se:

- d – número inteiro;
- c – caracter;
- s – cadeia de caracteres (string)
- f – número em virgula flutuante – formato 0.367 ;
- e – número em notação científica – formato 3.67×10^{-1} ;
- x – número em hexadecimal (utilizando abcdef);
- etc.

NOTA: Se pretende escrever o símbolo '%' será necessário fazer '%%'. Para se escrever '\' será necessário fazer '\\', pois '\' é o prefixo de alguns caracteres especiais, como o caractere fim de linha ('\n') ou o caractere fim de string ('\0').

Exemplo (*printf1.c*) :

```
#include <stdio.h>
int main()
{
   char nome[] = "Guilhermina Silva";
   int idade = 62;
   double peso = 73.2;

   /* %s string; %d inteiro; %.2f fraccionário com 2 casas decimais */
   printf( "Nome: %s\nIdade: %d\nPeso: %.2f\n", nome, idade, peso );

   return 0;
}
```

o output do programa será semelhante a:

```
Nome: Guilhermina Silva
Idade: 62
Peso: 73.20
>
```

- **fscanf** e **scanf** – leitura formatada

```
int fscanf ( FILE *stream, const char *format, ... )
int scanf ( const char *format, ... )
```

A função *fscanf* lê do ficheiro *stream* um conjunto de dados formatados segundo o indicado por *format*. Os dados lidos vão sendo sucessivamente atribuídos aos argumentos da lista ... , os quais deverão ser apontadores ou endereços de variáveis.

A função *scanf* lê sempre de *stdin*. Ambas as funções devolvem o número de campos lidos ou EOF se ocorrer erro ou fim de ficheiro.

A cadeia de formato na função é formada por uma sequência de caracteres normais, espaços, tabulações e especificadores de conversão. Os espaços e tabulações são ignorados. Os caracteres normais devem coincidir com os caracteres encontrados na leitura, nas posições correspondentes. Os especificadores de conversão começam pelo sinal de percentagem e são

seguidos dos caracteres de conversão, à semelhança da função *fprintf*. Como exemplos de caracteres de conversão tem-se:

- d – número inteiro decimal;
- x – número inteiro hexadecimal;
- f – número com ponto decimal;
- c – caracter;
- s – cadeia de caracteres sem espaços em branco (palavra);
- etc.

NOTAS:

- Quando é utilizado o especificador %s, a função *fscanf* lê os caracteres que encontra até surgir um caractere em branco, ou até ser atingida a largura do campo. Após a leitura é acrescentado o caractere terminador '\0'. Quando se utiliza o especificador %c, podem ser lidos *n* caracteres fazendo %*n*c (exemplo %256c – lidos 256 caracteres), mas neste caso o caracter '\0' não é acrescentado.
- Para ignorar campos pode ser utilizado o especificador %* seguido do caracter de conversão. Esta funcionalidade pode ser útil, por exemplo, para descartar o caractere terminador no caso de se estão a receber dados numéricos do teclado (ver exemplo).

Exemplo (*scanf1.c*) :

```
#include <stdio.h>
#define PI 3.14

int main()
{
	float raio, perimetro, area;

	printf("Introduza o raio da circunferencia: ");
	scanf("%f", &raio);

	perimetro = 2 * PI * raio;
	area = PI * raio * raio;

	printf("Dados do circulo:\n");
	printf("Perimetro = %.3f\n", perimetro);
	printf("Area = %.3f\n", area);

	return 0;
}
```

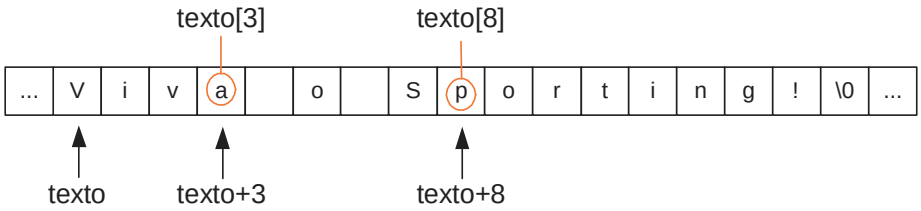
Representação de strings

Na linguagem C, uma *string* não é mais do que uma sequência de caracteres terminada pelo caracter especial '\0', cujo valor é 0. Como está a programar em C, não pode utilizar a classe *string* a que está habituado quando programa em C++. Para guardar o conteúdo de uma string, são utilizadas cadeias (*arrays*) de caracteres.

Exemplo:

```
char texto[] = "Viva o Sporting!";
```

Se olhássemos para as posições de memória onde fica guardada a *string*, visualizaríamos o seguinte:



`texto` é interpretado como uma referência (ou endereço) para a posição de memória onde começa a string; `texto[i]` será o conteúdo da i-ésima posição de memória contada a partir da posição `texto`, ou seja, o conteúdo da posição de memória referenciada por `texto+i`.

Para manipulação de strings encontra-se disponível um conjunto de funções de biblioteca que podem ser utilizadas fazendo a inclusão das funções de biblioteca definidas em *string.h*. Grande parte destas funções encontram-se descritas no anexo A a este documento.

Passagem de argumentos aos programas

Por vezes existem situações em que é útil efectuar a passagem de argumentos a um programa, quer através da linha de comandos, quer numa situação em que um programa lance um outro programa durante a sua execução.

A passagem de argumentos é feita através da função `main`, bastando para tal defini-la de uma forma especial:

```
int main ( int argc, char *argv[] )
```

`argc` contém o número de argumentos passados ao programa e cada elemento do array `*argv[]` é um apontador para a cadeia de caracteres correspondente a cada argumento. Para perceber melhor, considere que um hipotético programa chamado *clube* é lançado a partir da linha de comandos com:

```
> clube Sporting Benfica
```

O código do programa *clube* é o seguinte:

```
#include <stdio.h>

int main ( int argc, char *argv[] )
{
   if (argc != 3)
   {
      printf("Nao introduziu os argumentos como deve ser...\n");
      return -1;
   }
   printf("Eu sou do %s !!!\n", argv[1]);
   return 0;
}
```

Executando o programa com os argumentos "Sporting e "Benfica", o valor de `argc` seria 3 – o nome do programa também conta como argumento; `argv[1]` seria um apontador para o início da string "Sporting" e `argv[2]` um apontador para a string "Benfica". Se fosse passado um argumento numérico, este seria interpretado como uma cadeia de caracteres (seria necessário depois proceder-se à sua conversão, utilizando por exemplo a função `atoi` – conversão de string para inteiro – definida em *stdlib.h*).

Funções de abertura / fecho de ficheiros

Se pretende ler ou escrever em ficheiros, é necessário proceder-se à abertura do ficheiro e, depois de terminadas as operações de leitura/escrita, é necessário proceder-se ao fecho do ficheiro. As funções C habitualmente utilizadas para abertura e fecho de ficheiros são as funções *fopen* e *fclose*. Estas podem ser utilizadas fazendo a inclusão de *stdio.h*.

- **fopen** – abertura de um ficheiro.

```
FILE *fopen( const char *path, const char *mode )
```

Abertura do ficheiro dado em *path* para o modo indicado em *mode*. Os modos mais habituais são "r" para abrir um ficheiro para leitura, "w" para criar um ficheiro para escrita e "a" para abrir um ficheiro existente para escrita. A função devolve um ponteiro para o ficheiro aberto/criado ou NULL em caso de erro.

- **fclose** – fecho de um ficheiro.

```
int fclose( FILE *stream )
```

A função *fclose* fecha o ficheiro *stream*, devolvendo 0 em caso de sucesso, ou EOF em caso de erro.

Exemplo (*file1.c*):

```
#include <stdio.h>

int main() {

  FILE *fp;
  int total, nota, i;
  char nome[32];

  printf("N. de alunos a introduzir no ficheiro: ");
  scanf("%d%*c", &total);

  /* Abertura do ficheiro */
  fp = fopen("Alunos.txt","w");

  /* Ciclo de pedido de dados */
  for (i=0; i<total; i++) {

    printf("Nome (max. 32 caracteres): ");
    fgets(nome, 32, stdin);
    printf("Nota: ");
    scanf("%d%*c", &nota);

    /* Escrever dados no ficheiro */
    fprintf(fp, "%s %d\n", nome, nota);
  }

  /* Fechar o ficheiro */
  fclose(fp);

  return 0;
}
```

Processos

Criação de processos – fork

A chamada ao sistema `fork` permite a criação de um novo processo. O novo processo criado será filho do processo que invoca `fork`. A função `fork` tem o seguinte protótipo:

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

pid_t fork(void);
```

Quando invocada e em caso de sucesso, a função cria um novo processo, devolvendo ao processo criador (pai) o PID do novo processo criado. Ao novo processo (filho) a função devolve 0. Deste modo é possível proceder-se à separação do código relativo ao novo processo.

Em caso de erro, a função devolve o valor –1 ao processo pai e o processo filho não é criado.

A sua utilização será qualquer coisa semelhante a:

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

int main()
{
   pid_t procID;
   ...
   procID = fork();

      if ( procID < 0 )
      /* Erro na criação do novo processo */

      else if ( procID == 0 )
      /* Código do processo filho */

      else
      /* Código do processo pai */
      ...
}
```

Para mais informações sobre `fork()` poderá consultar o manual fazendo

- `man fork`

De seguida apresenta-se um exemplo de programa onde é a criação de um novo processo (exemplo fork1.c). Neste exemplo é também utilizada a função `getpid()`, que devolve o PID do processo que a chama.

```
#include <sys/types.h>
#include <unistd.h>
#include <stdio.h>

int main() {

  pid_t procID;

  procID = fork();
```

```
  if ( procID < 0 )
  {
    printf("Erro na criacao do processo\n");
    return -1;
  }
  else if ( procID == 0 )
  {
    printf("Processo filho - Para mim, o fork devolveu %d\n", procID);
    printf("Processo filho - PID=%d\n", getpid());
  }
  else
  {
    printf("Processo pai - Para mim, o fork devolveu %d\n", procID);
    printf("Processo pai – PID=%d\n", getpid());
  }
  return 0;
}
```

Outra propriedade muito importante que se verifica quando acontece o `fork`, é a herança, por parte do filho, de todo o contexto do processo pai, incluindo o conteúdo das variáveis, descritores de ficheiros abertos, etc.

De facto, após a chamada a `fork`, os processos pai e filho ficam a partilhar as mesmas páginas de memória, até que um deles produza alguma alteração.

No seguinte exemplo (*fork2.c*) pretende-se ilustrar esta situação. É pedido ao utilizador que introduza um número inteiro. O processo filho vai calcular o dobro desse número e o processo pai calcula a metade (inteira) desse número.

```
#include <sys/types.h>
#include <unistd.h>
#include <stdio.h>

int main() {

  pid_t procID;
  int numero;

  printf("Introduza um numero: ");
  scanf("%d", &numero);

  procID = fork();
  if ( procID < 0 ) {

    printf("Erro na criacao do processo\n");
    return -1;
  }
  else if ( procID == 0 ) {

    numero = numero * 2;
    printf("Dobro=%d\n", numero);
  }
  else {

    numero = numero / 2;
    printf("Metade=%d\n", numero);
  }
  return 0;
}
```

Repare que quando se dá o `fork` o valor de `numero` é igual para os dois processos, mas após modificação, esse valor passa a ser diferente para os dois processos.

Terminação de processos – exit

Todo o processo termina (quanto mais não seja quando se desliga o computador). Quando se elaboram programas, muitas vezes é necessário forçar a terminação do programa devido, por exemplo, a um utilizador ter feito uma asneira – neste caso o processo termina voluntariamente devido a um erro.

Para terminar um processo em qualquer altura da sua execução é utilizada a chamada ao sistema `exit`, que causa a terminação voluntária de um processo, fechando todos os ficheiros que se encontram abertos nessa altura.

```
#include <stdlib.h>

void exit(int status);
```

Ao ser chamada `exit`, o argumento especificado por `status` – o *exit status* – é passado ao processo pai e pode ser lido através da chamada ao sistema `wait` (ou `waitpid`, descritas mais adiante). A norma C especifica dois valores pré-definidos para o *exit status*: EXIT_SUCCESS e EXIT_FAILURE, para saídas voluntárias com sucesso e sem sucesso, respectivamente.

Quando um processo termina devido a uma excepção, ou devido a um sinal vindo de outro processo, diz-se que a saída é involuntária. Nesse caso não é passado o valor do *exit status* ao processo pai, mas sim um conjunto de informações a indicar qual a causa da terminação do processo.

Quando se utiliza `return` na função `main()` de um programa, o valor devolvido é interpretado como o *exit status*. Note que a utilização de `return` só é equivalente a `exit` no caso da função `main()`.

Esperar terminação de processos – wait

Por vezes pode ser útil que um processo pai fique bloqueado à espera que o processo filho termine. Para tal, pode-se utilizar a chamada ao sistema `wait`.

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

pid_t wait(int *status);
```

Esta função causa bloqueio ao processo pai até que um processo filho termine. A função devolve o PID do processo filho que terminou. Se `status` for diferente de NULL, é ainda passado por referência um conjunto de informações sobre a terminação do processo filho.

Caso se pretenda que um processo pai com vários filhos espere a terminação de um dos filhos em particular, pode-se utilizar a função `waitpid` com um funcionamento semelhante a `wait`,

mas na qual é possível especificar o PID do processo filho cuja terminação se pretende esperar, e também opções relacionadas com o tipo de espera a efectuar.

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

waitpid(pid_t pid, int*status, int options);
```

Pode obter mais informações sobre estas duas funções consultando o manual...

- `man 2 wait`

De seguida encontra-se um exemplo (wait1.c) que ilustra a utilização da chamada ao sistema `wait`. Neste exemplo é também utilizada a função `getppid()`, que devolve o PID do processo que é pai do processo que chama a função.

```
#include <sys/types.h>
#include <sys/wait.h>
#include <unistd.h>
#include <stdio.h>

int main() {

  pid_t procID;

  procID = fork();

  if ( procID < 0 ) {

    printf("Erro na criacao do processo\n");
    return -1;
  }

  else if ( procID == 0 ) {

    printf("Processo filho, PID=%d\n", getpid());
    printf("Processo filho, PID do pai=%d\n", getppid());
  }

  else {

    wait(NULL);
    printf("Processo pai - PID %d\n", getpid());
    printf("Processo pai - PID do filho %d\n", procID);
  }

  return 0;

}
```

A utilização de wait serve também para o processo pai ficar a saber de que modo terminou o filho, e qual o seu *exit status*, i.e., o valor devolvido pelo processo quando terminou. Para tal pode-se utilizar um conjunto de macros com diferentes funções (veja o manual do wait). No exemplo que se segue (wait2.c) são utilizadas as seguintes macros :

- WIFEXITED(status) – devolve 1 (true) se o processo filho terminou com saída voluntária.
- WEXITSTATUS(status) – devolve o *exit status* do processo filho que terminou. Só funciona se o processo filho tiver saído voluntariamente.

```
#include <sys/types.h>
#include <sys/wait.h>
#include <unistd.h>
#include <stdio.h>

int main() {

  pid_t procID;
  int estado;

  procID = fork();

  if ( procID < 0 ) {

    printf("Erro na criacao do processo\n");
    return -1;
  }

  else if ( procID == 0 ) {

    printf("Processo filho, PID=%d\n", getpid());
    printf("Processo filho, PID do pai=%d\n", getppid());
    return 1;
  }

  else {

    wait(&estado);

    if ( WIFEXITED(estado) ) {

      printf("O filho terminou normalmente, ");
      printf("com \"exit status\"=%d\n", WEXITSTATUS(estado));
    }

    printf("Processo pai - PID %d\n", getpid());
    printf("Processo pai - PID do filho %d\n", procID);
  }

  return 0;

}
```

Chamada de programas – execl

Muitas vezes também é útil ter o código do processo filho num outro programa distinto. Assim, na altura do fork, poder-se-á chamar um programa distinto onde se encontra o código respeitante ao processo filho. Para tal pode-se utilizar a função `execl` ou outra semelhante (faça `man execl`).

A função `execl` encontra-se definida do seguinte modo:

```
#include <unistd>

int execl( const char *path, const char *arg, ... )
```

`path` é o nome do programa executar (nome relativo ou absoluto), seguindo-se uma lista de argumentos a passar ao programa, terminada por NULL.

Esta função só retorna se tiver ocorrido algum erro (não encontrar o programa passado no argumento `path`, por exemplo), devolvendo –1 nesse caso. Se tudo correr bem, então esta função nunca retorna.

Os seguintes programas pretendem ilustrar o funcionamento de `execl`, combinada com `fork` e `waitpid`. O programa *fork3p.c* corresponde ao código do processo pai e o programa *prog3f.c* ao código do processo filho. Note que o processo pai continua a poder saber o *exit status* do processo filho.

```
/* Programa fork3p.c – codigo do processo pai */
#include <sys/types.h>
#include <sys/wait.h>
#include <unistd.h>
#include <stdio.h>

int main() {

  pid_t procID;
  int estado;
  const char prog[] = "fork3f";

  procID = fork();

  if ( procID < 0 ) {

    printf("Erro na criacao do processo\n");
    return -1;
  }
  else if ( procID == 0 ) {

    printf("Filho: Vai-se chamar outro programa - %s\n", prog);
    execl(prog, "Sporting", "Benfica", NULL);
  }
  else {

    waitpid(procID, &estado, 0);
    if (WIFEXITED(estado)) {
      printf("Pai: Processo filho %d terminado com exit status %d\n",
           procID, WEXITSTATUS(estado));
    }
  }
  return 0;
}


/* Programa fork3f.c – codigo do processo filho */
#include <stdio.h>

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

  printf("Filho: Eu sou do %s\n", argv[0]);
  printf("Filho: O pai é do %s\n", argv[1]);
  printf("Filho: O pai passou %d argumentos\n", argc);
  return 1;
}
```

Envio e tratamento de sinais – kill e signal

Os processos podem enviar sinais uns aos outros, e utilizar rotinas de tratamento dos sinais recebidos. Para enviar um sinal a um processo utiliza-se a chamada ao sistema kill. Esta chamada ao sistema tem um efeito semelhante ao comando do sistema kill, mas não confunda os dois.

A definição de kill é a seguinte:

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

int kill(pid_t pid, int sig);
```

A função aceita como argumentos o PID do processo para o qual se pretende enviar o sinal e o sinal a enviar. A função devolve 0 em caso de sucesso ou –1 em caso de insucesso.

Como exemplos de sinais a enviar, podemos ter os seguintes:

- SIGKILL – terminação drástica do processo
- SIGTERM – terminação do processo
- SIGSTOP – bloqueio do processo
- SIGCONT – continuar após bloqueio
- SIGUSR1 – sinal *user-defined* 1
- SIGUSR2 – sinal *user-defined* 2

(Estes são apenas alguns dos sinais – para uma lista completa faça *man 7 signal* na linha de comandos da shell)

Também é útil preparar os programas para receber sinais e efectuar algum processamento para tratamento dos mesmos. A função onde é feito o tratamento de um sinal, designa-se por *handler* do sinal. Para preparar um processo para receber um sinal e chamar nessa altura o *handler*, utiliza-se a função `signal`.

```
#include <signal.h>

typedef void (*sighandler_t) (int);
sighandler_t signal (int signum, sighandler_t handler);
```

A função aceita como argumentos o sinal a preparar e o nome do *handler* do sinal.

Normalmente a função é utilizada do seguinte modo:

```
#include <signal.h>

void handler (int sinal)
{
   /* Colocar aqui o código de tratamento do sinal
      não é aconselhável ser muito extenso... */

   ...
```

```
   /* No final desta função pode-se preparar
      novamente o processo para receber o sinal */
}

int main()
{
   ...

   signal(SIGUSR1, handler);

   /* A partir deste ponto, o processo está preparado para "apanhar" o
      sinal SIGUSR1. Quando o processo "apanhar" o sinal, salta-se
      para a função handler, executa-se o seu código e depois volta-se
      ao ponto do programa onde se apanhou o sinal */

   ...
}
```

NOTA: Se um processo estiver bloqueado numa chamada ao sistema `wait`, quando recebe um sinal desbloqueia sem o processo filho ter terminado.

Pode-se também ignorar um sinal – nesse caso, o segundo argumento de `signal` deverá ser SIG_IGN, em vez do nome da rotina de tratamento. Os sinais SIGKILL e SIGSTOP nunca são ignorados.

Quando um processo recebe um sinal sem estar preparado para o receber ou ignorar, é feito um tratamento por defeito definido pelo sistema, e que geralmente termina com a terminação do processo.

O seguinte exemplo (*signal1.c*) ilustra a utilização de `signal` e `kill`

```
#include <sys/types.h>
#include <sys/wait.h>
#include <signal.h>
#include <unistd.h>
#include <stdio.h>

/* Funcao para tratamento do sinal */
void handler( int sinal ) {

  printf("Apanhei o sinal que o filho mandou !!!\n");
 }

/* Funcao principal */
int main() {

  pid_t procID, ppid;

  signal(SIGUSR1, handler);

  procID = fork();
  if ( procID < 0 ) {

    printf("Erro na criacao do processo\n");
    return -1;
  }

  else if ( procID == 0 ) {

    printf("Vou enviar o sinal SIGUSR1 ao pai\n");
```

```
    ppid = getppid();
    kill(ppid, SIGUSR1);
  }

  else {

    waitpid(procID, NULL, 0);
  }
  return 0;
}
```

O *handler* do sinal não devolve nenhum valor, embora possam ocorrer situações em seria útil essa funcionalidade. Para contornar esta questão, nos casos em que se pretende que o *handler* devolva valores, terão que ser utilizadas variáveis globais...

Temporizações – alarm, pause e sleep

A chamada a `alarm` faz com que um sinal – SIGALRM – seja enviado ao próprio processo após o fim de um intervalo de tempo (especificado em segundos). Esta chamada ao sistema serve, por exemplo, para que um processo tenha alguma maneira de medir o tempo quando, por exemplo, está à espera de algum acontecimento em particular.

O protótipo da função `alarm` é o seguinte:

```
#include <unistd.h>

unsigned int alarm(unsigned int seconds);
```

Para cancelar uma chamada anterior a `alarm` poderá ser feito `alarm(0)`.

O seguinte exemplo (*alarm1.c*) ilustra a utilização de `alarm`. Neste caso, o processo está preparado para receber o sinal SIGALRM (repare na chamada a `signal`) e, quando o recebe, actualiza o valor de uma variável global que faz com que o processo saia do ciclo `while`.

```
#include <sys/types.h>
#include <signal.h>
#include <unistd.h>
#include <stdio.h>

int fim=1;

void handler( int sinal ) {
  printf("Recebi o SIGALRM - acordei !\n");
  fim=0;
}

int main() {
  signal(SIGALRM, handler);
  alarm(3);

  printf("Vou dormir...\n");
  while(fim);                 /* Ciclo em espera activa */

  return 0;
}
```

A chamada `pause` que faz com que um processo permaneça bloqueado até que receba um sinal que cause a sua terminação ou que tenha um *handler* definido.

```
#include <unistd.h>

int pause(void);
```

A função pause só retorna se for apanhado um sinal e o *handler* respectivo retornar. Nesse caso a função devolve sempre –1.

Para terminar, resta acrescentar que existe uma função de biblioteca chamada `sleep`, que faz com que o processo que a invoca permaneça no estado bloqueado durante um certo intervalo de tempo, ou até receber um sinal.

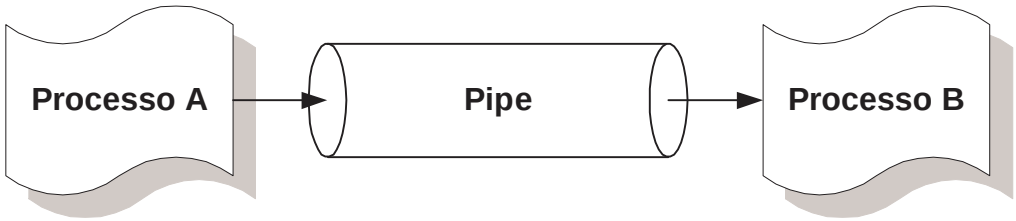
```
#include <unistd.h>

unsigned int sleep(unsigned int seconds);
```

A função `sleep` encontra-se implementada combinando as chamadas ao sistema `alarm` e `pause`. Misturar `sleep` com estas chamadas não é uma boa ideia...

Pipes

Os *pipes* são o mais antigo mecanismo para comunicação entre processos – desde as primeiras versões do Unix que existem e são utilizados. Actualmente são utilizados quando se pretende estabelecer um canal de comunicação unidireccional entre dois processos, como acontece muitas vezes na *shell*. A vantagem em utilizar os *pipes* é a simplicidade na sua programação.



Na disciplina de Sistemas Operativos iremos estudar formas de implementar em C dois tipos de pipes: *half-duplex pipes* e *pipes* com nome.

Half-duplex pipes

Os *pipes half-duplex* são o tipo de *pipes* utilizados na shell, quando introduz, por exemplo, o comando:

- `ps –aux | grep $USER`

É estabelecido um canal de comunicação pelo núcleo do sistema operativo, que permite o envio de dados do processo "ps" para o processo "grep". Os *pipes half-duplex* são caracterizados por permitirem o fluxo de informação num único sentido e pelo facto de apenas permitirem a comunicação entre processos com relação pai-filho.

Podem-se também escrever programas em C onde se criam novos processos e se estabelecem *pipes* para trocas de informação entre eles.

Criação de um pipe half-duplex

A criação de um *pipe half-duplex* pode ser feita recorrendo à chamada ao sistema `pipe`, cujo protótipo é:

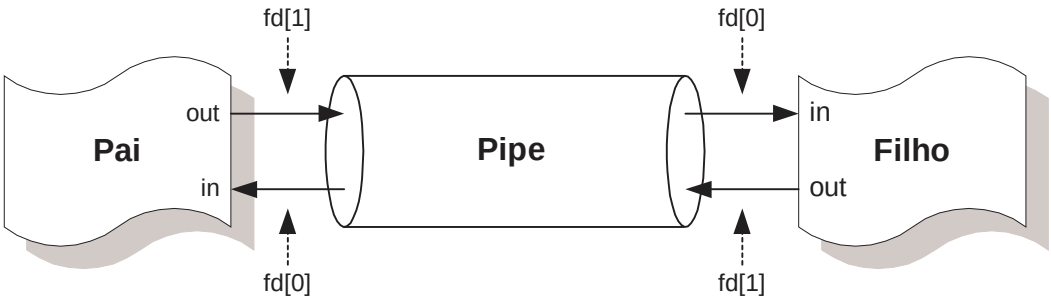
```
#include <unistd.h>

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

A função devolve 0 em caso de sucesso, ou –1 caso contrário. Para além disso, abre dois canais, um para escrita e outro para leitura, cujos descritores são devolvidos por referência:

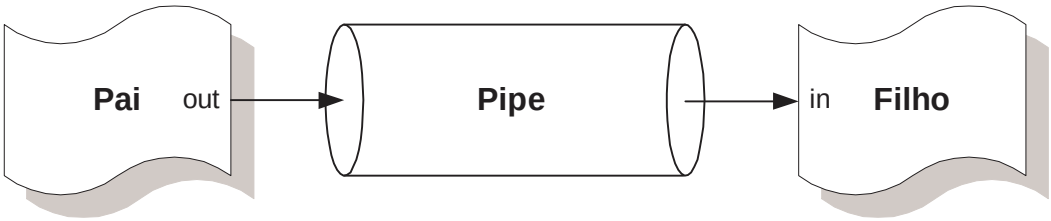
- Canal de leitura – descrito por `fd[0]`
- Canal de escrita – descrito por `fd[1]`

Após a criação do pipe e o lançamento de um novo processo, a comunicação entre os dois processos ficará então com a seguinte estrutura:



Como se pretende uma comunicação unidireccional, fecha-se o canal que não interessa em cada uma das extremidades *pipe*. Para tal, utiliza-se a chamada ao sistema `close`, que recebe como argumento o descritor do canal a fechar.

Por exemplo, após o fecho de `fd[0]` no processo pai e de `fd[1]` no processo filho, o *pipe* fica com a seguinte estrutura:



A partir deste ponto, o processo pai pode utilizar o *pipe* para enviar mensagens e o processo filho para recebê-las. A direcção do fluxo de informação poderia ser a inversa – para tal seria o filho a fechar o descritor `fd[0]` e o pai a fechar `fd[1]`.

O esqueleto de um programa que cria um *pipe* para comunicação entre um processo pai e um processo filho será o seguinte:

```
#include <unistd.h>
…
int main()
{
   int desc[2];
   …
   /* Criação do pipe – ambos os processo vão herdar o pipe */
   pipe(desc);

   if ( fork() == 0 )
   {
      /* Processo filho - vai ler, por isso fecha-se
                          o descritor de escrita */

      close(desc[1]);
      ...
   }
   else
   {
      /* Processo pai - vai escrever,  por isso fecha-se
                        o descritor de leitura */
      close(desc[0]);
      ...
   }
...
}
```

Escrita e leitura

Após a criação do *pipe* e fechados os canais não utilizados em cada uma das suas extremidades, podem ser utilizadas as chamadas ao sistema `read` e `write` para envio de mensagens de um processo para o outro.

Como facilmente se deduz, `read` serve para ler dados e `write` para escrevê-los[1].

A chamada `read` encontra-se definida do seguinte modo:

```
#include <unistd.h>

ssize_t read(int fd, void *buf, size_t count);
```

Os seus parâmetros de entrada são os seguintes:

- `fd` – descritor do canal de leitura;
- `buf` – apontador para o buffer onde serão guardados os dados recebidos;
- `count` – número máximo de bytes a ler do canal.

O valor devolvido pela função é o número de bytes de facto lidos do canal – este valor poderá ser inferior ao especificado em `count` (no caso em que se tenta ler mais do que de facto se encontra no canal). Uma importante característica da chamada `read` quando aplicada a *pipes*, é o facto de causar bloqueio ao processo que a chama no caso do *pipe* se encontrar vazio.

Quanto à chamada `write`:

```
#include <unistd.h>

ssize_t write(int fd, const void *buf, size_t count);
```

Os seus argumentos são análogos a `read`:

- `fd` – descritor do canal escrita;
- `buf` – apontador para o buffer onde estão guardados os dados a enviar;
- `count` – número de bytes a enviar para o canal.

A função devolve o número de bytes enviados para o canal que, tal como em `read`, poderá ser inferior ao especificado em `count`. Quando a chamada `write` é aplicada aos *pipes*, causa bloqueio ao processo que a chama quando o *pipe* se encontra cheio. A capacidade do *pipe* encontra-se definida pela constante PIPE_BUF – pela norma POSIX, a capacidade de um pipe é de 512 bytes, mas é habitual encontrar em distribuições Linux capacidades de 4Kbytes, ou mesmo superiores.

As chamadas `read` e `write` podem ser consideradas atómicas, pois o SO garante que não é feita nenhuma leitura ao *pipe* a meio de uma escrita, e vice-versa.

De seguida encontra-se um exemplo muito simples de comunicação entre um processo pai e um processo filho (*pipe1.c*), no qual o processo pai envia um mensagem (definida pelo utilizador) para o processo filho.

[1] `read` e `write` são chamadas ao sistema genéricas, que servem também para ler/escrever dados em ficheiros e também em periféricos.

```
#include <sys/types.h>
#include <unistd.h>
#include <stdlib.h>
#include <stdio.h>

int main() {

   char buffer[256];
   pid_t procID;
   int desc[2];          /* Descritores */

   /* Criação do pipe */
   if ( pipe(desc) != 0 ) {

      printf("Erro na criação do pipe\n");
      return EXIT_FAILURE;
   }

   /* Criação de um processo */
   procID = fork();

   if ( procID < 0 ) {

      printf("Erro na criação do processo\n");
      return EXIT_FAILURE;
   }
   else if ( procID == 0 ) {

      /* Processo filho */
      close(desc[1]);

      /* Receber do pipe */
      read(desc[0], buffer, 255);
      printf("O filho recebeu a seguinte mensagem:\n%s", buffer);
   }
   else {

      /* Processo pai */
      close(desc[0]);

      /* Pedir uma mensagem ao utilizador */
      printf("Introduza uma mensagem para enviar ao processo filho
              (Max. 255 caracteres):\n");
      fgets(buffer, 255, stdin);

      /*Enviar para o pipe */
      write(desc[1], buffer, 255);
   }
   return EXIT_SUCCESS;
}
```

Pipes com nome (FIFOS)

A grande diferença entre os *pipes* com nome e os *pipes half-duplex* referidos na secção anterior, é o facto dos *pipes* com nome residirem no sistema de ficheiros, ficando assim "visíveis" para todos os processos. Deste modo passa a ser possível utilizar um *pipe* para comunicação entre dois processos sem relação pai-filho.

À semelhança do que foi referido na secção anterior, antes de se poder utilizar o *pipe*, há que criá-lo. Tal pode ser efectuado (no Linux) utilizando a função `mkfifo`[2]:

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

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

Esta função aceita como argumentos o nome com o qual *pipe* passa a residir no sistema de ficheiros e as permissões (em octal) de acesso ao *pipe* (semelhante às permissões dos ficheiros). Em caso de sucesso a função devolve 0, caso contrário devolve –1.

Depois de criado o *pipe*, este só ficará a funcionar após a abertura de ambas as extremidades (uma para leitura e outra para escrita). Para abrir o *pipe*, pode ser utilizada a função `fopen`, pois um *pipe* com nome é tratado como um ficheiro...

Para escrever / ler dados no *pipe* utilizam-se as mesmas funções para escrita / leitura em ficheiros (`fprintf`, `fscanf`, `fgets`, `fputs`, etc.). Quando um processo já não vai utilizar mais o *pipe*, é conveniente proceder ao seu fecho, utilizando a função `fclose`.

Exemplo (fifo1.c) – neste exemplo é criado um *pipe* com nome para comunicar entre dois processos pai e filho. Um *pipe* deste tipo também serviria para processos sem essa relação. Se correr este programa, poderá também observar na directoria de trabalho o aparecimento do *pipe* (faça *ls*).

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

int main() {

  const char nomepipe[] = "fifofile";
  pid_t procID;
  char buffer[256];
  FILE *fp;

  /* Criação do pipe FIFO */
  mkfifo(nomepipe, 0600);

  procID = fork();
  if (procID == 0) {

    /* Processo filho */
    fp = fopen(nomepipe, "w");
    printf("Introduza a mensagem a enviar ao pai:\n");
    fgets(buffer, 255, stdin);
    fputs(buffer, fp);
    fclose(fp);
  }

  else {
```

[2] Em muitos sistemas Unix, não existe a função `mkfifo`. Nesses casos utiliza-se a função `mknod` que, para além da criação de fifos, permite ainda a criação de outros tipos de ficheiros "especiais". Se quiser obter mais informações sobre `mknod` faça *man 3 mknod* na linha de comandos.

```
    /* Processo pai */
    fp = fopen(nomepipe,"r");
    fgets(buffer, 255, fp);
    printf("Recebida a mensagem do filho:\n");
    printf("%s",buffer);
    fclose(fp);
  }
  return 0;
}
```

NOTA: Para o exemplo ser mais correcto, deveria ser feita protecção contra erros que eventualmente possam ocorrer nas funções `mkfifo`, `fork`, etc... No entanto, o objectivo do exemplo é ilustrar o modo como pode ser efectuado o envio de mensagens através de um FIFO, razão pela qual o código apresentado não é mais complexo.

Mecanismos IPC do Sistema V

Introdução

Para concluir o estudo sobre comunicação entre processos no Linux, falta referir a existência dos chamados mecanismos IPC (ou objectos IPC). Já deve ter ouvido falar da existência destes mecanismos nas aulas teóricas:

- Filas de mensagens
- Memória partilhada
- Semáforos

Existem também comandos de sistema relacionados com os IPCs, que pode executar a partir da linha de comandos da shell:

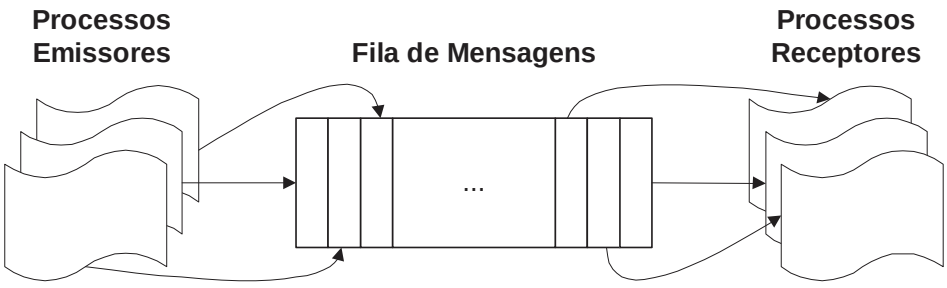
- `ipcs` – serve para visualizar todos o objectos IPC que se encontram criados no sistema;
- `ipcrm` – serve para remover do sistema um dado objecto IPC criado pelo utilizador.

(Para mais informações sobre estes comandos faça *man*)

De seguida descrevem-se mais em pormenor cada um destes mecanismos e as funções relacionadas com cada um deles, apresentando-se também alguns exemplos ilustrativos.

Filas de Mensagens (Message Queues ou Mailboxes)

Uma fila de mensagens pode ser descrita como lista, composta por várias mensagens, guardada num espaço de endereçamento pertencente ao núcleo do Sistema Operativo. A comunicação entre os processos é efectuada seguindo um modelo de caixa de correio.



Vários processos (emissores) podem enviar mensagens através da fila, e vários processos (receptores) podem recebê-las[3]. Sempre que um processo receptor lê uma mensagem, esta é removida da fila. Cada mensagem pode ter associada a si um determinado tipo, definido por um número inteiro não negativo (não confundir com tipos em C/C++). Esta propriedade permite que um dado processo receptor possa esperar uma mensagem de determinado tipo. Permite ainda estabelecer esquemas baseados em prioridades nas filas de mensagens.

[3] Um processo pode ser simultaneamente emissor e receptor – utilizar a fila de mensagens tanto para ler como para escrever.

Num esquema baseado no modelo da caixa de correio, um processo emissor que tenta escrever uma mensagem numa fila cheia, bloqueia até que exista espaço suficiente na fila para que possa escrever a mensagem. De modo semelhante, um processo receptor bloqueia quando não existem na fila mensagens do tipo pretendido.

Estrutura das mensagens

Esta é a estrutura mínima possível para uma mensagem, definida no sistema como sendo:

```
struct msgbuf {
   long mtype;          /* tipo de mensagem */
   char mtext[1];       /* texto da mensagem */
};
```

O programador pode definir uma estrutura personalizada para as suas mensagens. A seguinte estrutura é mais adequada para uma mensagem:

```
struct msgbuf {
   long mtype;          /* tipo */
   char mtext[256];     /* texto – até 256 caracteres */
};
```

É necessário ter em conta que uma mensagem não pode exceder um tamanho pré-definido pela constante `MSGMAX` (só o *root* a pode alterar e o valor por defeito é de 4Kbytes).

As mensagens residem no espaço do núcleo, sendo representadas pelo Sistema Operativo como uma lista de vários elementos com a seguinte estrutura:

```
struct msg {
   struct msg *msg_next;      /* próxima mensagem na fila */
   long msg_type;             /* tipo de mensagem */
   char *msg_spot;            /* inicio da mensagem */
   short msg_ts;              /* dimensão da mensagem */
};
```

Criação de uma fila de mensagens

Para criação de uma fila de mensagens ou associação a uma já existente é utilizada a função `msgget`. Esta função, tal como todas as funções C relacionadas com filas de mensagens, pode ser utilizada fazendo a inclusão dos ficheiros *sys/ipc.h* e *sys/msg.h*.

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

`key` Chave de criação ou associação à fila – criada anteriormente recorrendo à função `ftok()`, por exemplo (ver anexo B).

`msgflg` Permissões da fila de mensagens (em octal) em combinação com as seguintes opções, combinadas utilizando o operador '`|`' (*or*):

- `IPC_CREAT` Cria a fila de mensagens, ou então associa-se se esta já existir;
- `IPC_EXCL` Se for utilizada em conjunto com `IPC_CREAT`, faz com que a função `msgget` devolva erro se uma fila de mensagens associada à chave `key` já existir no sistema.

Em caso de sucesso, a função devolve um identificador para a fila de mensagens (criada ou associada ao processo). Em caso de erro devolve o valor –1, com a variável `errno` a indicar a causa do erro.

Envio de uma mensagem

Para envio de uma mensagem é utilizada a função *msgsnd*.

```
int msgsnd ( int msqid, struct msgbuf *msgp, int msgsz, int msgflg );
```

`msqid`	Identificador da fila, devolvido anteriormente por `msgget()`.
`msgp`	Referência (ou apontador) da estrutura da mensagem, devidamente preenchida com os dados a enviar.
`msgsz`	Tamanho da mensagem (em bytes).
`msgflg`	Opção – se for ignorada, isto é, `msgflg` = 0, o processo que chama `msgsnd` bloqueia no caso da fila se encontrar cheia. Utilizando a opção `IPC_NOWAIT`, no caso da fila se encontrar cheia, `msgsnd` não causa bloqueio, mas devolve erro.

A função devolve o valor 0 se for bem sucedida e, em caso de erro, devolve o valor –1.

Recepção de uma mensagem

```
int msgrcv ( int msqid, struct msgbuf *msgp, int msgsz, long mtype,
             int msgflg );
```

`mtype` indica o tipo de mensagem que se pretende receber

Os restantes argumentos são semelhantes aos de `msgsnd()`.

No caso do argumento `msgflg` for especificado utilizando a opção `IPC_NOWAIT`, o processo que chama a função `msgrcv` não bloqueia no caso de não existirem na fila mensagens do tipo pretendido. Nesta situação a função `msgrcv` devolve erro, mas o processo continua em execução.

Controlo/Remoção de uma fila de mensagens

Para operações de controlo sobre uma fila de mensagens é utilizada a função `msgctl`.

```
int msgctl ( int msgqid, int cmd, struct msqid_ds *buf );
```

As operações de controlo possíveis são dadas pelo argumento `cmd`. Este pode ser um dos seguintes:

`IPC_STAT` – Consulta a informações sobre a fila de mensagens.

IPC_SET – Alteração dos dados respeitantes à configuração da fila de mensagens.

IPC_RMID - Remoção da fila de mensagens do sistema.

Quando é criada uma fila de mensagens, é também criada no núcleo uma estrutura designada por msqid_ds, constituída por diversos campos que, no seu conjunto, descrevem a fila de mensagens e o seu estado. Essa estrutura encontra-se definida como[4]:

```
struct msqid_ds {
   struct ipc_perm msg_perm;  /* permissões, criador, etc */
   struct msg *msg_first;     /* primeira mensagem */
   struct msg *msg_last;      /* ultima mensagem */
   time_t msg_stime;          /* data/hora do ultimo msgsnd */
   time_t msg_rtime;          /* data/hora do ultimo msgrcv */
   time_t msg_ctime;          /* data/hora da ultima alteracao */
   struct wait_queue *wwait;
   struct wait_queue *rwait;  /* utilizados p/ gestão dos bloqueios */
   ushort msg_cbytes;         /* n. de bytes na fila */
   ushort msg_qnum;           /* n. de mensagens na fila
   ushort msg_qbytes;   /* máximo n.de bytes permitido na fila */
   ushort msg_lspid;    /* pid do ultimo proc. que chamou msgsnd */
   ushort msg_lrpid;    /* pid do ultimo proc. que chamou msgrcv*/
};
```

A grande maioria destes campos não podem ser alterados, existindo apenas para consulta, utilizando o comando IPC_STAT em msgctl. Neste caso, os valores dos diversos campos na estrutura mantida no núcleo são copiados para o argumento buf.

Alguns dos campos podem ser alterados pelo *root* ou pelo criador da fila de mensagens (embora neste caso a alteração de campos seja ainda mais restritiva). Tal alteração é possível utilizando o comando IPC_SET em msgctl, sendo nesse caso copiados os valores dos campos em buf para a estrutura mantida pelo núcleo.

Os campos que podem ser alterados são as permissões da fila e o máximo número de bytes permitidos na fila (mas apenas o root pode alterar este valor para um valor superior à constante MSGMNB definida no sistema – habitualmente este valor é de 16Kbytes).

Exemplos

Os exemplos que se seguem ilustram o modo como podem ser utilizadas as funções msgget, msgsnd, msgrcv e msgctl. Foram elaborados 4 programas:

msg1 – Cria (ou associa) uma fila de mensagens e envia 2 mensagens com tipos diferentes;

msg2 – Cria (ou associa) uma fila de mensagens e recebe 2 mensagens com tipos diferentes;

msg3 – Remove a fila criada por um dos programas anteriores;

msg4 – Caso a fila exista, exibe algumas informações sobre a fila.

[4] NOTA: a estrutura msqid_ds tem ligeiras diferenças de sistema para sistema.

```c
/* msg1.c - Envio de mensagens */

#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <sys/ipc.h>
#include <sys/msg.h>

#define NMSGS 2

/* Definicao da estrutura das mensagens */

struct mensagem {

  long tipo;
  char texto[256];
};

int main() {

  key_t chave;
  int filaID, i;
  struct mensagem msg;
  char buffer[256];

  chave = ftok(".",'a');

  /* Criar / associar fila de mensagens */
  filaID = msgget( chave, IPC_CREAT | 0666 );
  if ( filaID == -1) {
    printf("Erro na criação da fila de mensagens\n");
    return EXIT_FAILURE;
  }

  for (i=1; i<=NMSGS; i++) {

    /* Pedir uma mensagem */
    printf("Introduza a mensagem %d:\n", i);
    fgets(buffer, 255, stdin);

    /* Compor a mensagem */
    strcpy( msg.texto, buffer );
    msg.tipo = i;

    /* Enviar mensagem */
    if ( msgsnd( filaID, &msg, sizeof(msg), 0) == 0 ) {
       printf("Mensagem enviada com sucesso\n");
    }
    else {
      printf("Erro no envio da mensagem\n");
      return EXIT_FAILURE;
    }
  }
  return EXIT_SUCCESS;
}
```

```
/* msg2.c - Recepção de mensagens */

#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <sys/ipc.h>
#include <sys/msg.h>

#define NMSGS 2

/* Definicao da estrutura das mensagens */

struct mensagem {

  long tipo;
  char texto[256];
};

int main() {

  key_t chave;
  int filaID, i;
  struct mensagem msg;

  chave = ftok(".",'a');

  /* Criar / associar fila de mensagens */
  filaID = msgget( chave, IPC_CREAT | 0666 );
  if ( filaID == -1) {
    printf("Erro na criação da fila de mensagens\n");
    return EXIT_FAILURE;
  }

  for (i=1; i<=NMSGS; i++) {

    /* Receber mensagem */
    if ( msgrcv( filaID, &msg, sizeof(msg), i, 0) < 0 ) {
      printf("Erro a receber a mensagem\n");
      return EXIT_FAILURE;
    }
   printf("Mensagem do tipo %d:\n",i);
    printf("%s",msg.texto);
  }
  return EXIT_SUCCESS;
}




/* msg3.c - remoção da fila de mensagens */

#include <stdio.h>
#include <stdlib.h>
#include <sys/ipc.h>
#include <sys/msg.h>

int main() {

  key_t chave;
  int filaID;
```

```
  chave = ftok(".",'a');

  /* Associar fila de mensagens */
  filaID = msgget( chave, 0666 );
  if ( filaID == -1) {
    printf("Erro: A fila de mensagens não existe !\n");
    return EXIT_FAILURE;
  }

  /* Remover a fila */
  msgctl(filaID, IPC_RMID, NULL);

  return EXIT_SUCCESS;
}


/* msg4.c - Informações sobre a fila */

#include <stdio.h>
#include <stdlib.h>
#include <sys/ipc.h>
#include <sys/msg.h>

int main() {

  key_t chave;
  int filaID;
  struct msqid_ds buf;

  chave = ftok(".",'a');

  /* Associar fila de mensagens */
  filaID = msgget( chave, 0666 );
  if ( filaID == -1) {
    printf("Erro: A fila de mensagens não existe !\n");
    return EXIT_FAILURE;
  }

  /* Obter informações sobre a fila */
  msgctl( filaID, IPC_STAT, &buf);

  printf("\nInformações da fila %d:\n", filaID);
  printf("UID do criador: %d\n", buf.msg_perm.uid);
  printf("Permissões: %o\n",buf.msg_perm.mode);
  printf("N. de mensagens: %ld\n", buf.msg_qnum);

  return EXIT_SUCCESS;
}
```

Memória partilhada (Shared Memory)

Dois ou mais processos podem também efectuar trocas de informação se partilharem entre si um conjunto de posições de memória. Utilizando o mecanismo IPC da memória partilhada, pode-se definir um bloco de posições consecutivas de memória partilhado por vários processos.

Um processo pode escrever dados nesse bloco e outro processo pode lê-los. A vantagem da utilização de um esquema deste tipo é a rapidez na comunicação. A desvantagem é a inexistência de sincronização no acesso à memória.

Deste modo, a parte do código onde são efectuados acessos à memória partilhada, constitui geralmente uma região crítica, sendo portanto necessário assegurar a exclusão mútua de processos, utilizando por exemplo, os semáforos (vistos mais adiante).

Criação/associação de um bloco de memória partilhada

Para um processo criar um bloco de memória partilhada, ou para que possa utilizar um bloco já existente, utiliza-se a função `shmget`.

```
#include <sys/shm.h>
#include <sys/ipc.h>

int shmget ( key_t key, int size, int shmflg );
```

`key` Chave de criação ou associação ao bloco de memória partilhada – criada anteriormente recorrendo à função `ftok()`, por exemplo.

`size` Dimensão do bloco de memória a criar (em bytes)

`shmflg` Permissões do bloco de memória partilhada (em octal) em combinação com as seguintes opções, combinadas utilizando o operador ‘ | ’ (*or*):

- `IPC_CREAT` Cria o bloco de memória partilhada ou associa-se ao mesmo caso outro processo já o tenha criado;
- `IPC_EXCL` Se for utilizada em conjunto com `IPC_CREAT`, faz com que a função `msgget` devolva erro se um bloco de memória partilhada com chave `key` já existir.

A função devolve o identificador da zona de memória criada, ou –1 em caso de erro.

Após a criação ou associação a um bloco de memória partilhada, este ainda não está pronto a ser utilizado pelo processo, pois falta um dado essencial – o endereço (virtual), no espaço do processo, a partir do qual começa o bloco partilhado...

Attach e *Detach*

Para um processo mapear o bloco de memória partilhada para o seu espaço de endereçamento virtual, é utilizada a função `shmat`. Esta operação designa-se na gíria da programação por *attach*.

```
void *shmat ( int shmid, char *shmaddr, int shmflg);
```

`shmid`	Identificador da memória, obtido anteriormente por `shmget()`.
`shmaddr`	Endereço virtual a partir do qual se pretende que comece o bloco de memória partilhada. Se este argumento for NULL, o sistema encarrega-se de encontrar um endereço adequado – recomenda-se aos alunos que utilizem a função deste modo.
`shmflg`	Opções:

`0`	Sem opções;
`SHM_RDONLY`	Impede o processo de escrever no bloco de memória.
`SHM_RND`	No caso de `shmaddr` ser diferente de NULL, arredonda o endereço especificado para um múltiplo da dimensão de uma página de memória.

Em caso de sucesso, a função devolve um apontador para o início do bloco de memória partilhada. Em caso de erro devolve –1.

Exemplo:

```
...
int *shmptr;
key_t chave;
int shmID;
...

/* Criação/associação de um bloco com capacidade para 1024 inteiros */
shmID = shmget( chave, 1024 * sizeof(int), 0600 | IPC_CREAT );
...

/* Attach */
shmptr = shmat( shmID, NULL, 0 );
...
```

Após a utilização do bloco de memória partilhada, um processo pode libertar o mapeamento do seu espaço de endereçamento, utilizando a função `shmdt` – operação de *detach*.

```
int shmdt ( char *shmaddr );
```

onde `shmaddr` é o apontador devolvido anteriormente por `shmat()`.

Atenção que o bloco de memória partilhada continua a existir – efectuar o *detach* é diferente de remover o bloco.

Leitura e escrita na memória partilhada

As operações de leitura e escrita são efectuadas são operações de acesso à memória, com base no apontador devolvido por `shmat`.

Exemplo de algumas operações:

```
...
int shmID;
char *ptrshm;
char variavel, char buffer[256];
...

/* Criação/associação, supondo já obtida uma chave */
shmID = shmget( chave, 256, 0600 | IPC_CREAT);
...

/* Obter o endereço, indexando o bloco ao caractere */
ptrshm = shmat( shmID, NULL, 0 );
...

/* Escrever um caractere na primeira posição */
ptrshm[0] = 'A';              /* ou *ptrshm = 'A'; */
...

/* Ler o caractere da 5ª posição e guardá-lo numa variável */
variavel = ptrshm[4];         /* ou variavel = *(ptrshm+4); */
...

/* Escrever uma string */
strcpy( ptrshm, "Vamos a trabalhar !");
...
```

Operações de controlo e remoção

Para se proceder a operações de controlo sobre o bloco de memória partilhada, utiliza-se a função `shmctl`:

```
int shmctl ( int shmid, int cmd, struct shmid_ds *buf );
```

`shmid` é o identificador da zona de memória, obtido previamente com `shmget()`.

À semelhança do que acontecia nas filas de mensagens, `cmd` pode ser um de vários comandos:

`IPC_STAT` – Obter informações e guardá-las em `buf`.

`IPC_SET` – Alterar as permissões, dono e grupo da zona de memória partilhada de acordo com o passado em `buf`.

`IPC_RMID` – Remover o bloco de memória partilhada. O bloco de memória partilhada só será efectivamente removido quando todos os processos tiverem efectuado o *detach* do mesmo – quando um processo termina, o *detach* é feito de forma automática.

A estrutura com as informações e configurações sobre a memória partilhada é a estrutura `shmid_ds`, mantida no núcleo:

```
struct shmid_ds {
   struct ipc_perm shm_perm;  /* Permissões */
   int shm_segsz;             /* Dimensão do bloco (em bytes) */
   time_t shm_atime;          /* Data/hora do último attach */
   time_t shm_dtime;          /* Data/hora do último detach */
   time_t shm_ctime;          /* Data/hora da última escrita */
   unsigned short shm_cpid;   /* PID do processo criador */
   unsigned short shm_lpid;   /* PID do ultimo processo a aceder */
   short shm_nattch;          /* N. de attaches */

   /* Os seguintes campos são privados */
   unsigned short shm_npages;
   unsigned long *shm_pages;
   vm_area_struct *attaches;
};
```

Apenas a estrutura `shm_perm` pode ser alterada pelo *root*, ou pelo utilizador criador do bloco de memória partilhada. Os restantes campos servem apenas para consulta.

Exemplo (*shm1.c*) – neste exemplo, é criado um bloco de memória partilhada, utilizado pelo processo filho para enviar uma cadeia de caracteres ao pai.

```
#include <stdio.h>
#include <string.h>
#include <unistd.h>
#include <sys/wait.h>
#include <sys/types.h>
#include <sys/shm.h>

int main() {

  key_t chave;
  int shmID;
  char *shmptr;
  char buffer[256];

  /* Obter chave para objecto IPC */
  chave = ftok(".",'A');

  /* Criação do bloco de memória partilhada */
  shmID = shmget(shmID, 256, IPC_CREAT | 0600);
  if (shmID == -1) {
    printf("Erro na criação do bloco de mem. partilhada\n");
    return -1;
  }

  if ( fork() == 0 ) { /* Processo filho */

    /* Attach */
    shmptr = shmat(shmID, NULL, 0);

    /* Pedir uma mensagem ao utilizador */
    printf("Introduza a mensagem a enviar ao processo pai:\n");
    fgets(buffer, 255, stdin);

    /* Copiar mensagem para a memória partilhada */
    strcpy(shmptr, buffer);

    /* Detach */
    shmdt(shmptr);
  }
```

```
  else { /* Processo pai */

    /* Attach */
    shmptr = shmat(shmID, NULL, 0);

    /* Esperar que o filho termine */
    wait(NULL);

    /* Copiar mensagem da memória partilhada */
    strcpy(buffer, shmptr);
    printf("O pai leu a seguinte mensagem:\n%s",buffer);

    /* Detach */
    shmdt(shmptr);

    /* Remover IPC */
    shmctl(shmID, IPC_RMID, NULL);
  }
  return 0;
}
```

Semáforos

Os mecanismos IPC para sincronização de processos são os semáforos. Os semáforos são habitualmente utilizados quando se pretende garantir a exclusão mútua de processos em recursos partilhados (como por exemplo, memória partilhada).

As operações atómicas básicas com semáforos já foram descritas nas aulas teóricas da disciplina:

- UP – Incrementa o valor do semáforo em uma unidade
- DOWN – Tenta decrementar o valor do semáforo em uma unidade – causa bloqueio ao processo que o invoca se o resultado fosse dar negativo. O processo permanece bloqueado até que o valor do semáforo lhe permita efectuar o decremento.

Nos sistemas UNIX, podem ser efectuadas estas operações básicas, mas também existem algumas variantes:

- Os semáforos são criados em grupos (um grupo pode ser constituído por um único semáforo).
- Podem ser feitos UPs e DOWNs em mais de uma unidade – tirando este pormenor, o comportamento é semelhante aos UPs e DOWNs clássicos.
- Existe ainda uma terceira operação – designada por operação ZERO. Um processo que efectua esta operação permanece bloqueado até que o valor do semáforo seja igual a zero.

Criação de um grupo de semáforos

Para criação de um grupo de semáforos é utilizada a função `semget`:

```
#include <sys/sem.h>
#include <sys/ipc.h>

int semget ( key_t key, int nsems, int semflg );
```

`key` é a chave IPC obtida previamente com ftok().

`nsems` é o número de semáforos no grupo a criar.

`semflag` contem as permissões de acesso aos semáforos, e as opções:

- `IPC_CREAT` – cria o grupo ou associa-se ao mesmo se outro processo já a tiver criado.
- `IPC_EXCL` – se for usado em combinação com IPC_CREAT, dá erro se os semáforos já existirem.

Em caso de sucesso, a função devolve um identificador para o grupo de semáforos criado. Caso contrário devolve –1.

Operações atómicas sobre semáforos

Para efectuar operações atómicas sobre semáforos, utiliza-se a função `semop`:

```
int semop ( int semid, struct sembuf *sops, unsigned nsops);
```

`semid` é o identificador do grupo de semáforos.

`sops` é um ponteiro para o início de um *array* de estruturas que descrevem as operações a efectuar no grupo de semáforos.

`nsops` é o número de operações a efectuar. Este argumento só tem interesse se pretendermos efectuar operações em muitos semáforos. Quando só se pretende mudar um semáforo do grupo, este argumento deverá ser 1.

A operação a efectuar sobre um semáforo é definida recorrendo à estrutura `sembuf`:

```
struct sembuf {
   ushort sem_num;      /* Índice do semáforo no grupo */
   short sem_op;        /* Operação */
   short sem_flg;       /* Opções */
};
```

onde:

`sem_num` é o índice do semáforo (dentro do grupo de semáforos).

`sem_op` é o número de unidades a adicionar ou subtrair ao valor do semáforo.

`sem_flag` são opções

- `IPC_NOWAIT` – neste caso o processo que chama `semop` nunca bloqueia. No caso de se efectuar um DOWN que conduziria a um bloqueio, é devolvido um erro que pode depois ser analisado pelo processo.
- `SEM_UNDO` – causa a anulação da operação sobre o semáforo no caso de terminação do processo.

Para efectuar as clássicas operações de UP e DOWN, descritas nas aulas teóricas, poder-se-iam definir duas estruturas de operação de acordo com o seguinte (supondo um único semáforo no grupo – o semáforo 0):

```
/* Decrementar o semáforo 0 do grupo em uma unidade, sem opções */
struct sembuf DOWN = {0, -1, 0};

/* Incrementar o semáforo 0 do grupo em uma unidade, sem opções */
struct sembuf UP = {0, 1, 0};
```

Para efectuar uma operação UP, por exemplo, utilizar-se-ia `semop` do seguinte modo:

```
semop(semID, &UP, 1);
```

onde `semID` é o identificador do grupo de semáforos.

Operações de controlo

No caso dos semáforos, existem mais operações de controlo do que nos restantes mecanismos IPC. Entre as operações de controlo, encontram-se a inicialização dos semáforos (muito importante), consulta a informações sobre um semáforo do grupo, consulta global, remoção, etc.

A função que permite efectuar todas estas operações é a função `semctl`:

```
int semctl ( int semid, int semnum, int cmd, union semun arg );
```

em que:

`semid` é o identificador do grupo de semáforos.

`semnum` é o índice (dentro do grupo) do semáforo a controlar.

`cmd` é a operação a efectuar:

- IPC_STAT – Escrever informações sobre o grupo de semáforos para `arg.buf`;
- IPC_SET – Alterar o estado do grupo de semáforos de acordo com `arg.buf`;
- IPC_RMID – Remover o grupo de semáforos;
- GETNCNT – Devolver o número de processos que estão bloqueados no semáforo `semnum`;
- GETPID – Devolver PID do último processo que executou um `semop`;
- GETVAL – Devolver valor de um semáforo do grupo;
- GETALL – Obter os valores de todos os semáforos do grupo;
- SETVAL – Inicializar o valor de um semáforo do grupo;
- SETALL – Inicializar o valor de todos os semáforos do grupo;

A união[5] `semun` geralmente não se encontra definida no sistema (de acordo com X/OPEN). No caso de a termos que definir, podemos utilizar a seguinte:

```
union semun {
  int val;                    /* valor para SETVAL */
  struct semid_ds *buf;       /* buffer para IPC_STAT, IPC_SET */
  unsigned short int *array;  /* array para GETALL, SETALL */
  struct seminfo *__buf;      /* buffer para IPC_INFO */
};
```

A estrutura `semid_ds` é mantida pelo núcleo para representar um grupo de semáforos:

```
struct semid_ds {
   struct ipc_perm sem_perm;  /* Permissões */
   time_t sem_otime;          /* Data/hora da última operação */
   time_t sem_ctime;          /* Data/hora da última modificação */
   struct sem *sem_base;      /* Lista de semáforos */
   struct wait_queue *eventn;
   struct wait_queue *eventz;
   struct sem_undo *undo;
   ushort sem_nsems;          /* Nº de semáforos no grupo */
};
```

E cada semáforo é descrito pela estrutura `sem`:

```
struct sem {
   short sempid;   /* PID do processo que efectuou última operação */
   ushort semval;  /* Valor do semáforo */
   ushort semncnt; /* N. processos bloqueados por falta de unidades */
   ushort semzcnt; /* N. processos bloqueados à espera de semval=0 */
};
```

NOTA: O valor devolvido por `msgctl` depende da operação de controlo efectuada.

[5] Uma união é semelhante a uma estrutura, mas apenas um dos campos declarados é utilizado, dependendo da situação em questão.

Exemplo

Nos seguinte exemplo (*sem1.c*) ilustra-se a utilização dos semáforos. Um processo filho escreve números num bloco de memória partilhada. O processo pai lê esses números, calcula os seus quadrados, e escreve-os de volta no bloco. O processo filho lê os quadrados calculados pelo pai e mostra-os no écran.

É necessário garantir que a ordem das operações seja a correcta e que apenas um processo de cada vez se encontra a aceder ao bloco de memória partilhada. Estas garantias são asseguradas utilizando-se dois semáforos.

```
#include <sys/sem.h>
#include <sys/shm.h>
#include <sys/ipc.h>
#include <unistd.h>
#include <stdio.h>
#include <sys/types.h>


/* Definir semun caso não esteja definida */
#ifdef _SEM_SEMUN_UNDEFINED
#undef _SEM_SEMUN_UNDEFINED
union semun {
  int val;
  struct semid_ds *buf;
  unsigned short int *array;
  struct seminfo *__buf;
};
#endif


int main() {

  pid_t procID;
  int semID, shmID, i;
  int *shmptr;
  key_t chave;
  union semun semopts;

  /* Definir as operações sobre os semáforos */
  struct sembuf UP1 = {0, 1, 0};
  struct sembuf DOWN1 = {0, -1, 0};
  struct sembuf UP2 = {1, 1, 0};
  struct sembuf DOWN2 = {1, -1, 0};

  /* Obter chave */
  chave = ftok(".",'A');

  /* Criar um grupo com 2 semáforos */
  semID = semget(chave, 2, 0600 | IPC_CREAT);

  /* Criar mem. partilhada com capacidade para 10 inteiros */
  shmID = shmget(chave, 10*sizeof(int), 0600 | IPC_CREAT);

  /* Inicializar os semáforos (não é necessário pois já estão a 0) */
  semopts.val = 0;
  semctl(semID, 0, SETVAL, semopts);
  semopts.val = 0;
  semctl(semID, 1, SETVAL, semopts);
```

```
  procID = fork();
  if (procID == 0) {

    /* Attach */
    shmptr = shmat( shmID, NULL, 0);

    /* Escrever valores na mem. partilhada */
    for (i=0; i<10; i++)
      shmptr[i] = i;

    /* Desbloquear o pai */
    semop(semID, &UP1, 1);

    /* Bloquear à espera que o pai leia e escreva novos valores */
    semop(semID, &DOWN2, 1);

    /* Ler e mostrar valores */
    printf("Processo filho - valores recebidos:\n");
    for (i=0; i<10; i++)
      printf("%d ",shmptr[i]);
    printf("\n");

    /* Detach */
    shmdt(shmptr);

    /* Desbloquear o pai */
    semop(semID, &UP1, 1);
  }

  else {

    /* Attach */
    shmptr = shmat( shmID, NULL, 0);

    /* Bloquear à espera que o filho escreva valores */
    semop(semID, &DOWN1, 1);

    /* Ler os valores e escrever os quadrados */
    printf("Processo pai - valores recebidos:\n");
    for (i=0; i<10; i++)
      {
      printf("%d ",shmptr[i]);
      shmptr[i] = shmptr[i] * shmptr[i];
      }
    printf("\nProcesso pai - escritos os quadrados\n");

    /* Desbloquear o filho */
    semop(semID, &UP2, 1);

    /* Bloquear à espera que o filho leia e mostre os novos valores */
    semop(semID, &DOWN1, 1);

    /* Detach */
    shmdt(shmptr);

    /* Remover IPCs */
    shmctl(shmID, IPC_RMID, NULL);
    semctl(shmID, 0, IPC_RMID);
  }
  return 0;
}
```

Anexo A

Funções de leitura e escrita de cadeias de caracteres

Funções de I/O

As funções que se seguem referem-se a leitura e escrita de caracteres ou cadeias de caracteres (strings) para ficheiros e para os *standard streams*. Deste modo, as funções são apresentadas aos pares: a primeira função do par refere-se ao caso geral de escrita e leitura em qualquer ficheiro (incluindo os *standard streams*), a segunda função do par executa uma tarefa análoga, mas para escrita no *stdout* ou leitura do *stdin*.

Estas funções são utilizadas fazendo a inclusão de *stdio.h*.

- **fputs** e **puts** – escrita de uma sequência de caracteres.

```
int fputs ( const char *s, FILE *stream )
int puts( const char *s )
```

A função *fputs* escreve no ficheiro indicado por *stream* a cadeia de caracteres *s*, até ser encontrado o caracter terminador '\0'. A função *puts* é análoga mas escreve em *stdout*. Ambas devolvem um número não negativo em caso de sucesso ou EOF em caso de erro.

- **fgets** e **gets** – leitura de uma sequência de caracteres.

```
char *fgets ( char *s, int n, FILE *stream )
char *gets( char *s )
```

A função *fgets* lê de um ficheiro *stream* um máximo de *n* caracteres que são carregados para a sequência de caracteres indicada por *s*. A leitura termina após ser detectado o caracter terminador ou o caracter mudança de linha. O caracter mudança de linha é também copiado para *s*. É acrescentado o caracter terminador no final da sequência de caracteres lida.

A função *gets* lê uma cadeia de caracteres de *stdin*, guardando-os em *s*. Existe no entanto uma diferença em relação à função anterior – o caracter mudança de linha não é copiado para *s*. Ambas as funções devolvem um ponteiro para a sequência de caracteres lida ou NULL em caso de erro.

NOTA: evite utilizar a função gets, pois existe uma falha na sua implementação que pode conduzir ao chamado *buffer overflow*.

- **fputc** e **putchar** – escrita de um caracter.

```
int fputc( int c, FILE *stream )
int putchar (int c)
```

fputc escreve um caracter c (após uma conversão de *int* para *unsigned char*) no ficheiro *stream*. *putchar* é semelhante sendo o caracter escrito em *stdout*. Ambas as funções devolvem o caracter escrito, ou EOF em caso de erro.

- **fgetc** e **getchar** – leitura de um caracter.

```
int fgetc( FILE *stream )
int getchar( void )
```

fgetc lê um caracter do ficheiro *stream* e *getchar* lê um caracter de *stdin*. Ambas devolvem o caracter lido sob a forma de inteiro ou EOF em caso de erro.

- **perror** – escrita para o *stderr*, com indicação do erro ocorrido.

```
void perror( const char *s)
```

A função *perror* escreve em *stderr* a cadeia de caracteres *s*, seguida do caracter ':' , e de uma indicação textual último erro ocorrido finalizada com o caracter mudança de linha.

Funções para manipulação de strings

Existe um conjunto de funções em C para manipulação de strings. Na linguagem C, uma string não é mais do que uma sequência de caracteres terminada pelo caracter '\0'. Para poderem ser utilizadas estas funções, é necessário efectuar a inclusão do ficheiro "*string.h*", onde se encontram as declarações das funções.

De seguida descrevem-se muito sucintamente algumas dessas funções:

- **strcpy** – copia a string *src* para a string *dest*. São copiados sequencialmente os caracteres de *src* até ser encontrado o caracter terminador '\0'. É necessário ter em conta que o comprimento de *dest* terá que ser suficiente para guardar *src*. Em caso de sucesso a função devolve um ponteiro para a string *dest*.

  ```
  char *strcpy( char *dest, const char *src )
  ```

- **strcmp** – compara a string *s1* com a string *s2*. A função devolve 0 no caso das strings serem iguais.

  ```
  int strcmp ( const char *s1, const char *s2 )
  ```

- **strncmp –** esta função é semelhante à anterior mas apenas são comparados os primeiros *n* caracteres de cada uma das strings *s1* e *s2*.

  ```
  int strcmp ( const char *s1, const char *s2, size_t n )
  ```

- **strcat** – concatenação de uma string *src* a uma string *dest*. Devolve um ponteiro para a string *dest*. Atenção que *dest* deverá ter uma dimensão suficiente para suportar a concatenação.

  ```
  char *strcat ( char *dest, const char *src )
  ```

- **strlen** – esta função devolve o comprimento de uma string *s* (i.e., o número de caracteres até ser encontrado o caracter terminador '\0', não incluído para calcular o comprimento).

  ```
  size_t strlen ( const char *s )
  ```

- **strchr** – procura na string *s* a 1ª ocorrência de um caracter *c* especificado. É devolvido um ponteiro para o caracter, ou NULL se este não for encontrado.

  ```
  char * strchr ( const char *s, char c )
  ```

- **strrchr** – procura na string *s* a última ocorrência de um caracter *c* especificado. Devolve um ponteiro para o caracter encontrado ou NULL se este não for encontrado.

  ```
  char *strrchr ( const char *s, char c )
  ```

- **strstr** – procura na string *s1* a 1ª ocorrência de uma substring *s2*. Devolve um ponteiro para o início da substring encontrada em *s1* ou NULL se esta não for encontrada.

  ```
  char *strchr ( const char *s1, const char *s2 )
  ```

- **sprintf** – escrita formatada para a string *s*. O funcionamento desta função é semelhante ao funcionamento da função *printf*, mas o *output* é escrito na string *s*, em vez de ser escrito no *stdout*. A função devolve o número de caracteres escritos em *s*.

  ```
  int sprintf( char *s, const char *format, ... )
  ```

- **sscanf** – leitura formatada de uma string s. O funcionamento desta função é semelhante ao funcionamento da função *scanf*, mas o *input* é obtido a partir da string *s*, e não do *stdin*. Devolve o número de argumentos correctamente lidos.

  ```
  int sscanf( const char *s, const char *format, ... )
  ```

NOTA: – as funções *sprintf* e *sscanf* podem ser utilizadas fazendo a inclusão de *stdio.h*.

Anexo B

A função ftok

Para obter uma chave única para criação dos objectos IPC, é normalmente utilizada a função `ftok`.

O protótipo da função `ftok` é o seguinte:

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

key_t ftok( const char *pathname, int proj_id );
```

A função devolve uma chave para objectos IPC, que pode depois ser utilizada na chamada às funções `msgget`, `shmget` e `semget`.

A geração da chave é baseada num caminho – `pathname` – existente no sistema de ficheiros, que é depois combinada com os 8 bits menos significativos de `proj_id` (estes bits têm que ser diferentes de zero).

Como a função só utiliza os 8 bits menos significativos de `proj_id`, é habitual passar neste argumento o código ASCII de um caractere.

Exemplo:

```
…
key_t chave;
…
chave = ftok(“.”, ‘A’);
```

Neste exemplo, a chave é gerada com base na directoria actual e nos 8 bits correspondentes ao código ASCII do caractere A – recomenda-se que os alunos utilizem em `pathname` a directoria actual. Deste modo evita-se que alunos diferentes gerem chaves iguais e, consequentemente, utilizem os mesmos objectos IPCS, interferindo nos trabalhos uns dos outros...