12

Comunicação entre processos – 2ª parte
(IPC – InterProcess Communication)

Notas:

1. Deve ser criada a biblioteca **libnet.a** (referente ao código do livro [2] da bibliografia) antes de testar quaisquer exemplos e exercícios.
2. É condição prévia à aprendizagem deste assunto (IPC), o domínio sobre as funções **read** e **write** de I/O de baixo-nível.
3. O livro [2] de W. Stevens, *Unix networking programming*, é a principal referência sobre este assunto.
4. O livro [1] de W. Stevens, *Advanced programming in the Unix environment*, serve só de apoio à aprendizagem.

Este capítulo aborda as seguintes técnicas de sincronização (comunicação) entre processos:

- Segmentos de Memória Partilhada
- Semáforos

Linux man pages. É importante consultar as paginas sobre os comandos (If you don't run Linux, be sure to check your local man pages!)

- `ftok()`
- `ipcrm`
- `ipcs`
- `shmat()`
- `shmctl()`
- `shmdt()`
- `shmget()`

Segmentos de Memória Partilhada

Aquilo que se pode dizer sobre segmentos de memória partilhada é que são o que parecem: um segmento de memória pode ser partilhado por vários processos. Isto é, a comunicação entre processos é multi-direccional. Por exemplo, num jogo com vários jogadores, a comunicação entre os vários jogadores pode ser feita através de memória partilhada. Um segmento de memória partilhada pode ser lido ou escrito por qualquer jogador de modo que qualquer actualização do segmento é imediatamente transparente para qualquer jogado a ele ligado.

Criação dum segmento / ligação a um segmento

À semelhança de outras formas de IPC, um segmento de memória partilhada é criado ou ligado através da chamada ao sistema `shmget()`:

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


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

Retorna: identificador do segmento se OK, -1 em caso de erro.

Em caso de sucesso, esta função devolve o identificador do segmento de memória partilhada. O argumento *`key`* deve ser criado através da função `ftok()`, mas nada obriga a isso desde que saibamos o que estamos a fazer. O segundo argumento, *`size`*, é o tamanho em bytes do segmento de memória partilhada. Por último, o argumento *`shmflg`* especifica as permissões do segmento através dum OR bit-a-bit com `IPC_CREAT` se se quiser criar o segmento; se a este argumento for atribuído o valor 0, então pretende-se fazer uma ligação ao segmento, supondo que ele já foi criado.

Eis um exemplo que cria um segmento com 1KB e com as permissões 644 (`rw-r--r--`):

```
    key_t key;
    int shmid;

    key = ftok("/home/beej/somefile3", 'R');
    shmid = shmget(key, 1024, 0644 | IPC_CREAT);
```

Mas como é que um processo faz a ligação ao segmento através do identificador `shmid`? Isso é feito através da função `shmat()`.

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


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

Retorna: identificador do segmento se OK, -1 em caso de erro.

P

O primeiro argumento *shmid* é o ID do segmento de memória partilhada obtido a partir da chamada da função `shmget()`. O segundo argumento *shmaddr* é usado para dizer a `shmat()` qual o endereço do segmento a usar, mas deve ser activado a 0 por forma a que seja o sistema operativo a escolher o endereço por nós. Por último, o argumento pode ser activado com o valor `SHM_RDONLY` se se quiser só ler, ou 0 no caso contrário.

Aqui está um exemplo de como obter um ponteiro para um segmento de memória partilhada:

```
    key_t key;
    int shmid;
    char *data;

    key = ftok("/home/beej/somefile3", 'R');
    shmid = shmget(key, 1024, 0644 | IPC_CREAT);
    data = shmat(shmid, (void *)0, 0);
```

Obtém-se assim um ponteiro para um segmento de memória partilhada! Note-se que `shmat()` retorna um ponteiro `void`, e, neste caso, nós estamos a tratá-lo como um ponteiro `char`. Mas, nós podemos tratá-lo como quisermos, dependendo dos dados que nós temos no segmento.

É interessante notar que `shmat()` retorna –1 em caso de insucesso. Mas, como é que obtém –1 quando se usa um ponteiro `void`? Faz-se uma coerção (cast) durante o teste de erros:

```
    data = shmat(shmid, (void *)0, 0);
    if (data == (char *)(-1))
        perror("shmat");
```

Desligação a um segmento

Quando um processo não precisa mais de usar um segmento, o programa deve desligar-se do segmento através da seguinte função:

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


int shmdt(void* shmaddr);
```

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

O único argumento *shmaddr* é o endereço obtido a partir de `shmdt()`.

A desligação dum processo (ou de todos os processos) a um segmento não destrói este segmento de memória partilhada. A eliminação dum segmento só pode ser feita através da seguinte chamada ao sistema: função `shmctl()`:

```
shmctl(shmid, IPC_RMID, NULL);
```

P

A função `shmctl()` elimina um segmento de memória partilhada, assumindo que nenhum processo está a ele ligado. A função `shmctl()` tem a seguinte sintaxe:

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


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

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

Esta função elimina um segmento de memória partilhada, assumindo que nenhum processo está a ele ligado.

A eliminação dum segmento de memória partilhada também pode ser feita na linha de comando através da utilização do comando **`ipcrm`**. Aliás, todos os mecanismos IPC activos numa máquina Unix podem ser vistos através do comando **`ipcs`**.

Exemplo 12.10

Este exemplo foi extraído do livro intitulado "Distributed Programing" de Chris Brown (etiqueta I-3.4-78 da bibioteca), pág. 45.

Este exemplo compreende 3 ficheiros: line.h, cline.c e pline.c. O ficheiro line.h define o que será segmento de memória partilhada. O ficheiro pline.c cria o segmento de memória partilhada.

line.h

```
* The header file line.h */

struct info {
        char c;
        int length;
};

#define KEY             ((key_t)(1243))
#define SEGMENTSIZE     sizeof(struct info)
```

O processo pline deve iniciar a sua exeução em primeiro lugar e em bastidor (ou em background), já que é o processo que cria o segmento de memória partilhada, da seguinte forma:

```
$ pline &
```

O processo cliente deve ser executado de seguida em primeiro plano (ou em foreground) após alguns segundos, da seguinte forma:

```
$ cline
```

cline.c

```
* ------------- cline.c
 This is an example of using shared memory. The program operates in
 conjunction with the pline program.
 It aloows the line length and character  parameters of that program
 to be manually adjusted, by writing them into a shared memory segment.
 --------------*/

#include <stdio.h>
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/shm.h>
#include "line.h"

main(int argc, char *argv[])
{
       int     id;
       struct info *ctrl;
       struct shmid_ds shmbuf;

      if (argc != 3)
       {
              fprintf(stderr, "usage: cline <char> <length>\n");
              exit(3);
       }

       id = shmget(KEY, SEGMENTSIZE, 0);

       if (id < 0)
      {
              printf("cline: shmget failed: ");
              exit(1);
       }

       ctrl = (struct info *)shmat(id,0,0);
       if (ctrl <= (struct info *)(0))
       {
              printf("cline: shmat failed: ");
              exit(2);
       }

       /* Copy command line data to shared memory region */
       ctrl -> c       = argv[1][0];
       ctrl -> length  = atoi(argv[2]);

       exit(0);
}
```

pline.c

```
* ------------- pline.c
 This is an example of using shared memory. The program prints a line of
 text every four seconds. The length of the line, and the character
 printed, are controlled by two numbers held in a small data structure
 in a shared memory segment. Some other program can attach the segment and
 change the numbers.
 --------------
*/

#include <stdio.h>
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/shm.h>
#include "line.h"

main()
{
        int     i, id;
        struct info *ctrl;
        struct shmid_ds shmbuf;

       id = shmget(KEY, SEGMENTSIZE, IPC_CREAT | 0666);

        if (id < 0)
        {
                printf("pline: shmget fdailed: ");
                exit(1);
        }

        ctrl = (struct info *)shmat(id,0,0);
        if (ctrl <= (struct info *)(0))
        {
                printf("pline: shmat failed: ");
                exit(2);
        }

        /* Set default parameters */
        ctrl -> c       = 'a';
        ctrl -> length  = 10;

       /* Main loop - print a line every 4 seconds */
        while (ctrl -> length > 0)
        {
                for (i=0; i < ctrl->length; i++)
                        putchar(ctrl->c);
                putchar('\n');
                sleep(4);
        }
       exit(0);
}
```

Semáforos

Um semáforo não é, na realidade, uma forma de IPC semelhante às anteriores (pipetas, FIFOs e filas de mensagens). Um semáforo não é um mecanismo de comunicação de dados entre processos, mas tão somente uma primitiva de sincronização. Embora seja considerado um mecanismo IPC, um semáforo não é usado para trocar grandes quantidades de dados entre processos como acontece com as pipetas, FIFOs e filas de mensagens, mas são usados com o objectivo de vários processos e threads poderem sincronizar as suas operações.

O **principal objectivo** da utilização de semáforos é, pois, sincronizar o acesso a segmentos de memória partilhada. Para que esta sincronização entre processos seja eficaz, um semáforo deve ser independente dos processos, e é por isso que são armazenados no kernel do sistema operativo (Figura 12.1).

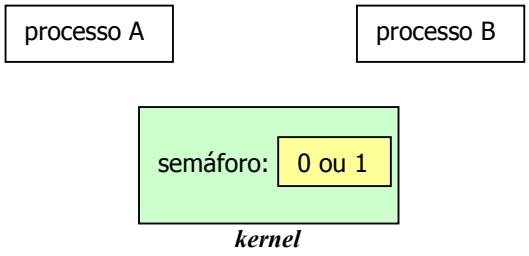

Figura 12.1: Valor de semáforo armazenado no kernel.

Um semáforo não é mais do que uma variável inteira que funciona como contador de recursos. O valor corrente desta variável representa o número de unidades disponíveis de um dado recurso num dado instante. Se há um só recurso (p.ex., um ficheiro) partilhado, então os valores possíveis do semáforo são 0 e 1.

Para obter um recurso que é controlado por um semáforo, há que:

1. Testar o valor corrente do semáforo.
2. Se o valor do semáforo for positivo, o processo pode usar o recurso. O processo decrementa unitariamente o semáforo para indicar que vai usar uma unidade do recurso.
3. Se o valor do semáforo é 0, o processo vai dormir até que o valor do semáforo fique positivo. Quando o processo acorda, ele volta ao passo 1 anterior.

Note-se que a utilização dum recurso por um processo requer que este decremente o valor do semáforo (passo 2), ao passo que a libertação dum recurso requer que o processo incremente o valor do semáforo (passo 3).

A implementação correcta de semáforos requer que o teste (passo 1) e a decrementação (passo 2) do semáforo constituam uma operação atómica. É esta a principal razão porque os semáforos são normalmente implementados dentro do kernel do sistema operativo, onde é possível garantir que um conjunto de operações sobre um semáforo seja feita atomicamente.

Infelizmente, os semáforos têm três características que podem comprometer o seu correcto funcionamento:

1. Um semáforo não é só um valor não-negativo. De facto, um semáforo é um conjunto de um ou mais valores inteiros. Quando um semáforo é criado, temos de especificar o seu número de valores possíveis. Além do mais, pode ter valores negativos.

2. A criação dum semáforo (**semget**) é independente da sua inicialização (**semctl**). Isto pode causar problemas visto que a criação e a inicialização não constituem uma operação atómica.
3. Dado que todas as formas de IPC permanecem no sistema mesmo quando nenhum processo está a usá-las, o utilizador/programador tem de se preocupar com a libertação dos semáforos antes da terminação do qualquer processo que faça IPC via semáforos. A libertação de semáforos faz-se à custa do mecanismo UNDO.

O kernel contém a seguinte estrutura **semid_ds** para cada conjunto de semáforos (Figura 12.2):

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

struct semid_ds {
      struct      ipc_perm    sem_perm;    /* operation permission struct */
      struct      sem         *sem_base;   /* ptr to first semaphore in set*/
      ushort                  sem_nsems;   /* no. of semaphores in set */
      time_t                  sem_otime;   /* last-semop() time*/
      time_t                  sem_ctime;   /* last change time*/
};
```

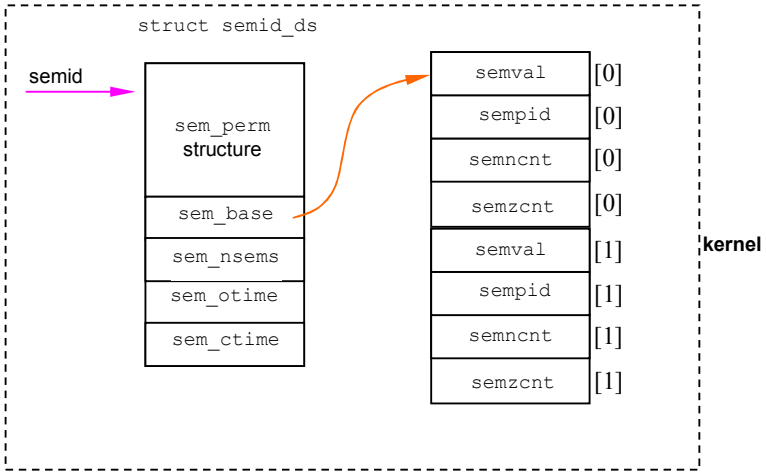

Figura 12.2: Estruturas de dados dum conjunto de dois semáforos (`sem_nsems`=2).

A estrutura `ipc_perm` (já descrita anteriormente) contém as permissões de acesso ao semáforo em questão. O endereçador `sem_base` não tem qualquer utilidade para processos do utilizador, visto que ele aponta para memória reservada do kernel. Este endereçador aponta para um array de estruturas `sem` contendo `sem_nsems` elementos, cada elemento para cada valor do semáforo no conjunto. A estrutura `sem` contém o valor (`semval`) do semáforo, o identificador (`sempid`) do processo que fez a última operação sobre este valor, o contador (`semncnt`) do número de processos à espera que o valor do semáforo aumente, e o contador (`semzcnt`) do número de processos à espera que o valor do semáforo se torne 0.

```
struct sem {
      ushort      semval;     /* semaphore value, always >=0 */
      pid_t       sempid;     /* pid for last operation*/
      ushort      semncnt;    /* no. of processes awaiting semval > currval */
      ushort      semzcnt;    /* no. of processes awaiting semval = 0*/
};
```

P

Criação/Acesso a semáforos

Um semáforo é criado (ou acedido) através da seguinte chamada ao sistema:

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

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

Retorna: identificador do semáforo (`semid`) se OK, -1 em caso de erro.

O segundo argumento `nsems` é o número de semáforos no conjunto. Se um novo conjunto de semáforos está a ser criado (tipicamente no servidor), nós temos de especificar `nsems`. Se o conjunto já existe (um cliente), `nsems` deve ter o valor 0.
O terceiro argumento `flag` é uma combinação de constantes da seguinte tabela (Figura 12.3):

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

Figura 12.3: Valores de *flag* na chamada de `semget`.

Quando um conjunto de unidades dum semáforo é criado, os seguintes elementos da estrutura **semid_ds** são inicializados:

- Todos os elementos (com a excepção de `seq`) da estrutura **ipc_perm** são inicializados na altura da criação da estrutura IPC. Mais tarde, os campos **uid**, **gid** e **mode** podem ser alterados através da chamada de `semctl`. A mudança destes campos é semelhante à chamada de **chown** ou **chmod** para um ficheiro. O campo **mode** é actividado com os bits de permissão de *flag* (Figura 12.3).
- `sem_nsems` é activado com o valor de `nsems`.
- `sem_otime` é activado com o valor 0.
- `sem_ctime` é activado com o valor do tempo corrente.

Operações atómicas sobre semáforos

Operações atómicas sobre semáforos são feitas à custa da seguinte chamada ao sistema:

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

int semop(int semid, struct sembuf semoparray[], size_t nops);
```

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

O terceiro argumento *nops* especifica o número de operações (elementos) no array *semoparray*.
O segundo argumento *semoparray* é um ponteiro para um array de operações:

```
struct sembuf {
      ushort      sem_num;          /* member # in set (0,1,..., nsems-1) */
      short       sem_op;           /* operation (negative, 0, or positive) */
      short       sem_flg;          /* IPC_NOWAIT, SEM_UNDO*/
};
```

O campo `sem_op` especifica a operação sobre cada membro do conjunto:

1. **`sem_op > 0`**. Isto significa que o processo quer libertar recursos. O valor de `sem_op` é adicionado ao valor corrente do semáforo. Se o flag SEM_UNDO é especificada, `sem_op` é também subtraída ao valor de ajustamento do semáforo.
2. **`sem_op < 0`**. Isto significa que o processo quer obter recursos controlados pelo semáforo.
    a. Se **`valor do semáforo >= |sem_op|`**, i.e., se os recursos estão disponíveis, o valor absoluto de `sem_op` é subtraído ao valor do semáforo. Fica assim garantido que o resultado é >=0. Se a flag SEM_UNDO é especificada, o valor absoluto de `sem_op` é também adicionado ao valor de ajustamento do semáforo.
    b. Se **`valor do semáforo < |sem_op|`**, i.e., se os recursos não estão disponíveis:
        i. Se IPC_NOWAIT é especificada, é feito o retorno com o erro EAGAIN;
        ii. Se IPC_NOWAIT não é especificada, o valor de `semncnt` do semáforo é incrementado (visto que está perto de ir dormir) e o processo invocador é suspenso até que um dos seguintes eventos ocorra:
            - O valor do semáforo torna-se maior ou igual ao valor absoluto de `sem_op` (i.e. algum outro processo acabou de libertar alguns recursos). O valor de `semncnt` deste semáforo é decrementado (uma vez que estamos à espera) e o valor absoluto de `sem_op` é subtraído ao valor do semáforo. Se a flag SEM_UNDO é especificada, o valor absoluto de `sem_op` é também adicionado ao valor de ajustamento do semáforo deste processo.
            - O semáforo é removido do sistema. Neste caso, a função retorna o erro ERMID.
            - Um sinal é apanhado pelo processo. Neste caso, o valor de `semncnt` do semáforo é decrementado (visto não estarmos mais à espera), e a função retorna o erro EINTR.
3. **`sem_op = 0`**. Isto significa que queremos esperar até que o valor do semáforo seja 0. Portanto, se o valor do semáforo for 0, a função retorna de imediato. Caso contrário:
    a. Se IPC_NOWAIT é especificada, é feito o retorno com o erro EAGAIN;
    b. Se IPC_NOWAIT não é especificada, o valor de `semzcnt` do semáforo é incrementado (visto

P

que está perto de ir dormir) e o processo invocador é suspenso até que um dos seguintes eventos ocorra:

i. O valor do semáforo torna-se 0. O valor de `semzcnt` deste semáforo é decrementado (dado estarmos à espera).
ii. O semáforo é removido do sistema. Neste caso, a função retorna o erro ERMID.
iii. Um sinal é apanhado pelo processo. Neste caso, o valor de `semzcnt` do semáforo é decrementado (visto não estarmos mais à espera), e a função retorna o erro EINTR

Há problema quando um processo que termina a sua execução tem ainda recursos afectados a um semáforo. Sempre que se especifica a flag SEM_UNDO numa operação de semáforos, e recursos são reservados (a `sem_op < 0`), o kernel regista quantos recursos foram afectados àquele semáforo (o valor absoluto de `sem_op`). Quando o processo termina, voluntária ou involutariamente, o kernel verifica se o processo tem alguns ajustamentos a fazer ao semáforo, e em caso afirmativo, fá-lo.
Se o valor do semáforo for activado através de semctl, com SETVAL ou SETALL, o valor de ajustamento em todos os processos é activado a 0.

P

Operações de controlo sobre semáforos

A função de controlo de semáforos é a seguinte:

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

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

Retorna: ver abaixo.

O quarto argumento *arg* desta chamada ao sistema é definida do seguinte modo:

```
union semun {
      int                     val;        /* for SETVAL */
      struct      semid_ds    *buf;       /* for IPC_STAT and IPC_SET */
      ushort                  *array;     /* for GETALL and SETALL */
};
```

O terceiro argumento *cmd* especifica um dos seguintes 10 comandos a ser executado no conjunto de sinalização especificado por *semid*:

- **IPC_STAT**. Procura a estrutura semid_ds deste conjunto, armazenando-a na estrutura apontada por *arg.buf*.
- **IPC_SET**. Activa os três campos sem_perm.uid, sem_perm e sem_perm.mode do semáforo a partir da estrutura *arg.buf*. Este comando só pode ser executado por um processo cujo effective user ID é igual a sem_perm.cuid ou sem_perm.uid, ou por um processo com privilégios de super-utlizador.
- **IPC_RMID**. Remove o semáforo do sistema. Qualquer outro processo que esteja a usar o semáforo obterá um erro de EIDRM na próxima operação sobre o semáforo. Este comando só pode ser executado por um processo cujo effective user ID é igual a sem_perm.cuid ou sem_perm.uid, ou por um processo com privilégios de super-utlizador.
- **GETVAL**. Retorna o valor de `semval` para o membro *semnum*.
- **SETVAL**. Activa o valor de `semval` para o membro *semnum*. O valor é especificado por *arg.val*.
- **GETPID**. Retorna o valor de `sempid` para o membro *semnum*.
- **GETNCNT**. Retorna o valor de `semncnt` para o membro *semnum*.
- **GETZCNT**. Retorna o valor de `semzcnt` para o membro *semnum*.
- **GETALL**. Procura todos os valores do semáforo no conjunto. Estes valores estão armazenados no array apontado por *arg.array*.
- **SETALL**. Activa todos os valores do semáforo no conjunto com os valores armazenados em *arg.array*.

Os cinco comandos que se referem a um valor particular do semáforo usam *semnum* para especificar um membro do conjunto. O valor de *semnum* está entre 0 e *nsems-1*, inclusivé.
Na execução de todos os comandos GET, à excepção de GETALL, a função retorna o valor correspondente. Nos comandos restantes, o valor devolvido é 0.

Exemplo 12.1: (Semáforo binário)
O acesso à secção crítica (segmento de memória partilhada) faz-se através da função **lock** que decrementa o valor do semáforo, ao passo que a libertação do recurso (segmento de memória partilhada) é feita pela invocação da função **unlock**.
Nota: este exercício não vem nos livros de Stevens.

lock.h

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

union semun {
      int               val;        /* for SETVAL */
      structsemid_ds    *buf;       /* for IPC_STAT and IPC_SET */
      ushort            *array;     /* for GETALL and SETALL */
};

int lock(int, int);
int unlock(int, int);
```

lock.c

```
#include "lock.h"

int lock(int semid, int sem_num)
{
  struct sembuf s;
  s.sem_num=sem_num;
  s.sem_op=-1;
  s.sem_flg=SEM_UNDO;
  return semop(semid,&s,1);
}

int unlock(int semid, int sem_num)
{
 struct sembuf s;
 s.sem_num=sem_num;
 s.sem_op=1;
 s.sem_flg=SEM_UNDO;
 return semop(semid,&s,1);
}
```

segment.h

```
/* The header file segment.h
   Defines a shared memory segment for an integer */

#define KEY             ((key_t)(1244))
#define SEGMENTSIZE     sizeof(int)
```

binarysem.h

```
#include "lock.h"
#include "segment.h"

void
main ()
{
  int id;
  int *ctrl;
  struct shmid_ds shmbuf;

  union semun init;
  int semid;

  id = shmget(KEY, SEGMENTSIZE, IPC_CREAT | 0666);
  ctrl = (int *)shmat(id, 0, 0);

  semid = semget (IPC_PRIVATE, 1, IPC_CREAT | 0600);

  init.val = 1;                  // inicializes semaphore with 1
  semctl(semid,0,SETVAL,init);

  (*ctrl) = 5;                   // initializes integer shared segment with 5

  if (fork () == 0)              // child process
    {
      printf ("FILHO: tenta entrar! \n");
      lock (semid, 0);           // critical section follows
      printf("FILHO: entrou!\n");
      printf("FILHO: o valor da variavel partilhada = %d\n", ++(*ctrl));
      sleep (5);
      printf("FILHO vai sair!\n");
      unlock (semid, 0);
    }
  else                           // parent process
    {
      printf ("PAI: tenta entrar\n");
      lock (semid, 0);           // critical section follows
      printf ("PAI: entrou!\n");
      printf("PAI: o valor da variável partilhada = %d\n", --(*ctrl));
      sleep (5);
      /* parent always enters first */
      printf ("PAI: vai sair\n");
      unlock (semid, 0);
      wait (0);
      semctl (semid, 1, IPC_RMID, 0);   // removes semaphore
    }
}
```

Exercício 12.1:
Transcreva e teste o código que usa um conjunto de três semáforos do livro [2] de W. Stevens, *Unix networking programming*, pp.145-152.

Exercício 12.2:
Simulação do funcionamento através de semáforos para sincronizar um conjunto de processos.

O Problema da Barbearia

- A barbearia tem 3 barbeiros (e 3 cadeiras de barbeiro).
- A barbearia tem um sofá com 4 lugares de espera.
- A barbearia tem uma caixa registadora.
- A lotação é de 20 clientes.
- Os clientes são atendidos por ordem de chegada.
- A registadora é operada pelos barbeiros e fornece um recibo.

Escreva a lista dos semáforos necessários e o programa relativo aos vários processos.

Semáforo	**Descrição**	**Valor inicial**
capacidade_disponível	Cliente espera para entrar na sala	20
lugares_sofa_disponíveis	Cliente espera por lugar no sofa	4
cadeiras_barbeiro_disponíveis	Cliente espera por cadeira de barbeiro	3
cliente_senta_cadeira_barbeiro	Barbeiro espera que um cliente se sente na sua cadeira	0
barbeiro_termina_tarefa	Cliente espera que o barbeiro termine a sua tarefa	0
cliente_deixa_cadeira_barbeiro	Barbeiro espera que o cliente se levante da cadeira	0
pagamento	Registadora espera pelo pagamento do cliente	0
recibo	Cliente espera pelo recibo	0
associa_barbeiro_a_cadeira_ou_registadora	Atribui barbeiro a uma cadeira ou à registadora	3

O pseudo-código dos processos relativos aos clientes da barbearia é o seguinte:

cliente.c

```
void main()
{
     wait(capacidade_disponível);                          entra na barbearia;
     wait(lugares_sofa_disponíveis);                          senta-se no sofa;
     wait(cadeiras_barbeiro_disponíveis);                  levanta-se do sofá;
     signal(lugares_sofa_disponíveis);       senta-se na cadeira do barbeiro;
     signal(cliente_senta_cadeira_barbeiro);
     wait(barbeiro_termina_tarefa);                deixa cadeira do barbeiro;
     signal(cliente_deixa_cadeira_barbeiro);                              paga;
     signal(pagamento);
     wait(recibo);                                            sai da barbearia;
     signal(capacidade_disponível);
}
```

O pseudo-código dos processos relativos aos barbeiros é o seguinte:

barbeiro.c

```
void main()
{
   do
   {
        wait(cliente_senta_cadeira_barbeiro);
        wait(associa_barbeiro_a_cadeira_ou_registadora);
               corta cabelo;
        signal(associa_barbeiro_a_cadeira_ou_registadora);
        wait(barbeiro_termina_tarefa);
        wait(cliente_deixa_cadeira_barbeiro);
        signal(cadeiras_barbeiro_disponíveis);
   } while (1);
}
```

O pseudo-código do processo relativo à caixa registadora é o seguinte:

caixa.c

```
void main(){
   do
   {
        wait(pagamento);
        wait(associa_barbeiro_a_cadeira_ou_registadora);
               aceita pagamento;
        signal(associa_barbeiro_a_cadeira_ou_registadora);
        signal(recibo);
   } while (1);
}
```

O pseudo-código do processo relativo gerador de clientes, barbeiros e da caixa registadora é o seguinte:

mainbarbas.c

```
void main(){
      criar semáforos;
        criar caixa registadora;
        criar 3 barbeiros;
        criar 25 clientes;
}
```