7. Memória partilhada e sincronização em UNIX

7.1. Memória partilhada

Os sistemas Unix derivados do Unix System V, e outros compatíveis (quase todos), definem serviços que permitem partilhar entre vários processos um bloco de memória. Assim aquilo que um processo escreve numa determinada posição desse bloco de memória é imediatamente visto por outros processos que partilhem o mesmo bloco. O pedaço de memória física que constitui o bloco partilhado entra no espaço de endereçamento de cada processo que o partilha como um pedaço de memória lógica (cada processo que partilha o bloco de memória pode vê-lo com endereços lógicos diferentes).

Para partilhar um bloco de memória por vários processos é necessário que um deles crie o bloco partilhado, podendo depois os outros associá-lo ao respectivo espaço de endereçamento. Quando o bloco de memória partilhado não for mais necessário é muito importante libertá-lo explicitamente, uma vez que a simples terminação dos processos que o partilham não o liberta. Todos os sistemas suportam um número máximo de blocos de memória partilháveis. Se não forem convenientemente libertados, quando se atingir esse número máximo, nenhum processo poderá criar mais.

Criação e associação de blocos de memória partilhados:

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

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

Retorna um valor positivo (identificador do bloco) no caso de sucesso ou -1 no caso de erro.

Para criar um bloco de memória partilhada usa-se o serviço acima, que deverá retornar um valor inteiro positivo conhecido como o idenficador do bloco de memória partilhada (*shmid* ou *shared memory identifier*); este identificador terá depois de ser usado nos outros serviços que dizem respeito à utilização da memória partilhada.

O valor **`key`** (geralmente um inteiro longo) pode ser a constante `IPC_PRIVATE` ou um valor arbitrário diferente dos já utilizados na criação de outros blocos de memória partilhada.

Quando se usa a constante `IPC_PRIVATE`, o bloco criado só pode ser utilizado em processos que sejam descendentes do processo que cria o bloco, e mesmo para isso é necessário passar-lhes de alguma maneira (p. ex. na linha de comando) o identificador retornado por `shmget()`. Os processos descendentes podem (e devem) usar esse identificador para aceder ao bloco de memória partilhada.

Quando se usa um valor específico para **`key`**, qualquer outro processo (incluindo os descendentes) pode partilhar o bloco (desde que o mesmo tenha permissões compatíveis). Para isso quem cria o bloco tem de incluir em **`flag`** a constante `IPC_CREAT`. Uma vez criado o bloco, outro processo que o queira usar tem também de chamar o serviço `shmget()`, especificando a mesma **`key`** (mas sem `IPC_CREAT` em **`flag`**) para obter o identificador *shmid*. Quando se usa uma **`key`** específica, conhecida de todos os processos que querem usar o bloco de memória partilhada, corre-se o risco, embora remoto (há

alguns biliões de *keys* diferentes), de que outros processos não relacionados já tenham utilizado essa **key** (quando se usa a constante IPC_PRIVATE é sempre criado um novo bloco com um identificador diferente). Para garantir que o sistema assinala um erro quando se está a usar uma **key** idêntica a um bloco já existente é necessário acrescentar (com *or* (|)) a constante IPC_EXCL a **flag**.

O parâmetro **size** especifica em bytes o tamanho do bloco de memória partilhada a criar.

O parâmetro **flag**, além de poder conter os valores IPC_CREAT e/ou IPC_EXCL também serve para especificar as permissões do bloco a criar no que diz respeito à leitura ou escrita por parte do *owner* (utilizador do processo que cria o bloco), *group* ou *others*. Para isso devem-se acrescentar (novamente com *or* (|)) respectivamente as constantes: SHM_R, SHM_W, SHM_R>>3, SHM_W>>3, SHM_R>>6 e SHM_W>>6.

Quando se pretende obter garantidamente uma **key** diferente associada a um determinado processo (proveniente de um ficheiro executável) pode usar-se o seguinte serviço:

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

key_t ftok(char *pathname, int nr);
```

onde **pathname** será o nome do ficheiro executável que pretende a **key** (e respectivo *path*) e **nr** pode ser um valor entre 0 e 255, para prever diversas instâncias ou diversos blocos criados pelo processo.

Uma vez criado o bloco, e obtido o seu identicador por parte do processo que o quer utilizar, (ou tendo o mesmo sido passado a um filho quando se usa IPC_PRIVATE), é necessário agora mapear o bloco para o espaço de endereçamento do processo e obter um apontador para esse bloco (operação designada por *attach*). Para isso utiliza-se o serviço:

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

void *shmat(int shmid, void *addr, int flag);

Retorna um apontador válido no caso de sucesso ou -1 no caso de erro.
```

O serviço retorna um apontador (genérico) para o início do bloco partilhado. O parâmetro **shmid** é o identificar do bloco, obtido em shmget(); o parâmetro **addr** poderá servir para sugerir um endereço lógico de mapeamento, no entanto geralmente usa-se para **addr** o valor 0, para deixar o sistema escolher esse endereço; por sua vez, o parâmetro **flag** pode ser 0 ou conter a constante SHM_RDONLY se se pretender apenas ler o bloco partilhado.

Quando um processo não necessitar de aceder mais ao bloco partilhado deve desassociá--lo do seu espaço de endereçamento com o serviço shmdt(), que significa *shared memory detach*.

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

int shmdt(void *addr);
```

addr é o apontador retornado por `shmat()`.

Retorna 0 no caso de sucesso ou -1 no caso de erro.

Notar que este serviço não destrói o bloco partilhado, nem sequer a terminação de todos os processos que o utilizaram, incluindo o que o criou. Para libertar totalmente um bloco de memória partilhado é necessário usar o serviço seguinte (com **cmd** igual a `IPC_RMID`):

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

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

Retorna 0 no caso de sucesso ou -1 no caso de erro.

Além de se indicar o identificador do bloco (em **shmid**) é necessário especificar um "comando" (no parâmetro **cmd**) com a acção a executar, que pode ser:

- `IPC_RMID` - liberta o bloco, quando o último processo que tenha feito um `shmat()` execute o correspondente `shmdt()` ou termine. Após a execução deste comando já não é possível efectuar outras operações de `shmat()`, mesmo que o bloco ainda se mantenha em memória. Só um processo que pertença ao mesmo utilizador que criou o bloco pode executar este comando. O parâmetro **buf** deve ser `NULL` para este comando.
- `IPC_STAT` - Preenche a estrutura apontada por **buf** com informações acerca do bloco (permissões, *pid* do owner, instante de criação, número de associações, etc) (ver `man`).
- `IPC_SET` - Através dos campos `shm_perm.uid`, `shm_perm.gid` e `shm_perm.mode` da estrutura apontada por **buf** permite modificar as permissões, o dono e o grupo do bloco de memória. (ver `man`).

Exemplo 1: Criação e partilha de um bloco de memória entre pai e filho

```
...
int shmid;
int *pt1, *pt2;
pid_t pid;

shmid = shmget(IPC_PRIVATE, 1024, SHM_R | SHM_W);
pid = fork();
if (pid > 0) {
    pt1 = (int *) shmat(shmid, 0, 0);
    ...
    pt1[0] = 20;
    pt1[1] = 30;
    ...
    shmdt(pt1);
    waitpid(pid, ... );
    shmctl(shmid, IPC_RMID, NULL);
    exit(0);
}
else {
    pt2 = (int *) shmat(shmid, 0, 0);
```

```
        ...
        pt2[2] = pt2[0] * pt2[1];
        ...
        shmdt(pt2);
        exit(0);
    }
    ...
```

Exemplo 2: Criação e partilha de um bloco de memória entre quaisquer 2 processos

Processo 1:

```
    ...
    key_t key;
    int shmid;
    int *pt;

    key = ftok("proc1", 0);
    shmid = shmget(key, 1024, IPC_CREAT | IPC_EXCL | SHM_R | SHM_W);
    pt = (int *) shmat(shmid, 0, 0);
    ...
    pt[0] = 20;
    pt[1] = 30
    pt[100] = 0;
    ...
    while (pt[100] != 1) { }
    shmdt(pt);
    shmctl(shmid, IPC_RMID, NULL);
    exit(0);
    ...
```

Processo 2:

```
    ...
    key_t key;
    int shmid;
    int *pt;

    key = ftok("proc1", 0);              /* usa a mesma key */
    shmid = shmget(key, 0, 0);           /* não cria, apenas utiliza */
    pt = (int *) shmat(shmid, 0, 0);
    ...
    pt[2] = pt[0] * pt[1];
    ...
    pt[100] = 1;
    ...
    shmdt(pt);
    exit(0);
```

É claro que o acesso a uma mesma área de memória partilhada por parte de vários processos deverá ser sincronizada, utilizando semáforos ou mutexes.

Recentemente a norma POSIX introduziu um conjunto de novos serviços para a criação, utilização e destruição de blocos de memória partilhada. No entanto como essa norma é muito recente ainda são poucos os sistemas que a suportam.

Outra alternativa para a implementação de blocos de memória partilhada entre processos é o mapeamento de um ficheiro no espaço de endereçamento do processo. Para isso um ficheiro existente em disco é aberto com o serviço `open()`, obtendo-se assim um seu descritor. De seguida, utilizando o serviço `mmap()` (usar o `man` para uma descrição), é

possível mapear esse ficheiro em memória e acedê-lo usando apontadores. Vários processos independentes podem mapear o mesmo ficheiro nos seus espaços de endereçamento (com as permissões adequadas). A operação inversa executa-se com o serviço `munmap()`.

7.2. Mutexes

A norma POSIX que definiu a API de utilização dos *threads* em UNIX também definiu os objectos de sincronização denominados por mutexes e variáveis de condição. Os mutexes podem ser vistos como semáforos que só existem em 2 estados diferentes e servem fundamentalmente para garantir, de forma eficiente, a exclusão mútua de secções críticas de vários *threads* ou processos que executam concorrentemente. Quando um *thread* adquire (ou tranca (*locks*), na nomenclatura usada em Unix) um mutex, a tentativa de aquisição do mesmo mutex por parte de outro *thread* leva a que este fique bloqueado até que o primeiro *thread* liberte o mutex.

Assim, a protecção de uma secção crítica por parte de um *thread* ou processo deverá fazer-se usando as operações de aquisição e libertação (*lock* e *unlock*) de um mesmo mutex:

```
pthread_mutex_lock(&mutex);
    …  /* secção crítica */
pthread_mutex_unlock(&mutex);
```

Um mutex é simplesmente uma variável do tipo `pthread_mutex_t` definido no ficheiro de inclusão `pthread.h`. Antes de poder ser utilizado um mutex tem de ser inicializado. Pode fazer-se essa inicialização quando da sua declaração, usando uma constante pré-definida em `pthread.h` denominada `PTHREAD_MUTEX_INITIALIZER`:

```
pthread_mutex_t mut = PTHREAD_MUTEX_INITIALIZER;
```

Também é possível inicializar um mutex depois de declarado com o seguinte serviço:

```
#include <pthread.h>

int pthread_mutex_init(pthread_mutex_t *mptr,
                       const pthread_mutexattr_t *attr);
```

Retorna 0 se OK ou um valor positivo (com o código do erro) no caso de erro.

O parâmetro **`mptr`** é o endereço da variável que representa o mutex e que se pretende inicializar. O parâmetro **`attr`** permite especificar os atributos que o mutex irá ter. Para inicializar o mutex com os seus atributos por defeito (de forma equivalente à constante `PTHREAD_MUTEX_INITIALIZER`) podemos passar aqui o valor `NULL`.

Na maior parte dos sistemas, por defeito, os mutexes só podem ser utilizados em *threads* diferentes de um mesmo processo. Para utilizar mutexes em processos diferentes estes terão de residir em memória partilhada por esses processos e deverão ser inicializados de modo a que possam ser usados dessa forma. Nos sistemas que suportam este modo de funcionamento (sistemas que definem a constante `_POSIX_THREAD_PROCESS_SHARED` em `unistd.h`) a inicialização terá de ser feita como se mostra no seguinte exemplo:

```
#include <pthread.h>

pthread_mutex_t *mptr;
pthread_mutexattr_t mattr;

…
mptr = …        /* endereço em memória partilhada pelos vários processos */
pthread_mutexattr_init(&mattr);      /* inicializa variável de atributos */
pthread_mutexattr_setpshared(&mattr, PTHREAD_PROCESS_SHARED);
pthread_mutex_init(mptr, &mattr);
```

Uma vez inicializado podemos então operar sobre um mutex utilizando um dos seguintes serviços:

```
#include <pthread.h>

int pthread_mutex_lock(pthread_mutex_t *mptr);
int pthread_mutex_unlock(pthread_mutex_t *mptr);
int pthread_mutex_trylock(pthread_mutex_t *mptr);
```

mptr é o endereço da variável que representa o mutex.

Retornam 0 se OK ou um código de erro positivo no caso contrário.

O serviço `pthread_mutex_lock()` adquire (tranca) o mutex se este estiver livre, ou bloqueia o *thread* (ou processo) que o executa se o mutex já pertencer a outro *thread* até que este o liberte (destranque).
O serviço `pthread_mutex_unlock()` liberta um mutex previamente adquirido. Em princípio esta operação deverá ser efectuada pelo *thread* que detém o mutex.
Por fim, o serviço `pthread_mutex_trylock()` tenta adquirir o mutex; se este estiver livre é adquirido; no caso contrário não há bloqueio e o serviço retorna o código de erro `EBUSY`.

Quando um determinado mutex não for mais necessário, os seus recursos podem ser libertados com o serviço:

```
#include <pthread.h>

int pthread_mutex_destroy(pthread_mutex_t *mptr);
```

Retorna 0 se OK, ou um código de erro positivo no caso contrário.

Exemplo:
Pretende-se um programa *multithreaded* que preencha um array comum com o máximo de 10000000 de entradas em que cada entrada deve ser preenchida com um valor igual ao seu índice. Devem ser criados vários *threads* concorrentes para executar esse preenchimento. Após o preenchimento, um último *thread* deverá verificar a correcção desse preenchimento. O número efectivo de posições do *array* a preencher e o número efectivo de *threads* devem ser passados como parâmetros ao programa.

Segue-se uma solução utilizando as funções `fill()` e `verify()` para os *threads* de preenchimento e verificação. Além disso toma-se nota do número de posições preenchidas por cada *thread* `fill()`. Os *threads* partilham globalmente o *array* a

preencher, a posição actual, e o valor de preenchimento actual (que por acaso é igual ao índice actual):

```
#include <pthread.h>
#include <stdio.h>
#include <stdlib.h>

#define MAXPOS 10000000                /* nr. max de posições */
#define MAXTHRS 100                    /* nr. max de threads */
#define min(a, b) (a)<(b)?(a):(b)

int npos;
pthread_mutex_t mut=PTHREAD_MUTEX_INITIALIZER;      /* mutex para a s.c. */
int buf[MAXPOS], pos=0, val=0;                   /* variáveis partilhadas */

void *fill(void *);
void *verify(void *);

int main(int argc, char *argv[])
{
  int k, nthr, count[MAXTHRS];                    /* array para contagens */
  pthread_t tidf[MAXTHRS], tidv;                  /* tid's dos threads */

  if (argc != 3) {
    printf("Usage: fillver <nr_pos> <nr_thrs>\n");
    return 1;
  }
  npos = min(atoi(argv[1]), MAXPOS);       /* nr. efectivo de posições */
  nthr = min(atoi(argv[2]), MAXTHRS);      /* nr. efectivo de threads */
  for (k=0; k<nthr; k++) {
    count[k] = 0;                          /* criação dos threads fill() */
    pthread_create(&tidf[k], NULL, fill, &count[k]);
  }
  for (k=0; k<nthr; k++) {
    pthread_join(tidf[k], NULL);          /* espera pelos threads fill() */
    printf("count[%d] = %d\n", k, count[k]);
  }
  pthread_create(&tidv, NULL, verify, NULL);
  pthread_join(tidv, NULL);               /* thread verificador */
  return 0;
}

void *fill(void *nr)
{
  while (1) {
    pthread_mutex_lock(&mut);
    if (pos >= npos) {
      pthread_mutex_unlock(&mut);
      return NULL;
    }
    buf[pos] = val;
    pos++; val++;
    pthread_mutex_unlock(&mut);
    *(int *)nr += 1;
  }
}

void *verify(void *arg)
{
  int k;
```

```
  for (k=0; k<npos; k++)
    if (buf[k] != k)          /* escreve se encontrar valores errados */
      printf("buf[%d] = %d\n", k, buf[k]);
  return NULL;
}
```

Se tudo correr bem o programa deverá apenas escrever o número de posições preenchidas por cada *thread* `fill()`.

Experimentar usando por exemplo:

```
fillver 1000000 5
fillver 5000000 5     e
fillver 10000000 5
```

7.3. Variáveis de condição

A utilização de variáveis de condição no Unix é adequada na seguinte situação:
Um determinado *thread* (ou processo) pretende aceder à sua secção crítica apenas quando uma determinada condição booleana (também chamada predicado) se verifica. Enquanto essa condição não se verificar o *thread* pretende ficar bloqueado sem consumir tempo de CPU.
A utilização exclusiva de mutexes não é adequada para esta situação.

Os serviços de variáveis de condição permitem programar esta situação de forma relativamente simples. Vamos supor que 2 *threads* partilham alguma informação e entre aquilo que é partilhado estão duas variáveis `x` e `y`. Um dos *threads* só pode manipular a informação partilhada se o valor de `x` for igual a `y`.

Usando um mutex, o acesso à informação partilhada (que inclui `x` e `y`) por parte do *thread* que necessita que `x` e `y` sejam iguais, poderia ser feito da seguinte forma:

```
while(1) {
    pthread_mutex_lock(&mut);
    if (x == y)
        break;
    pthread_mutex_unlock(&mut);
}
…… /* secção crítica */
pthread_mutex_unlock(&mut);
```

Ora esta solução pode consumir toda a fatia de tempo deste *thread* à espera da condição `(x==y)`, desperdiçando o CPU.

Usando variáveis de condição, este pedaço de código seria substituído por:

```
pthread_mutex_lock(&mut);
while (x != y)
    pthread_cond_wait(&var, &mut);
…… /* secção crítica */
pthread_mutex_unlock(&mut);
```

O que o serviço `pthread_cond_wait()` faz é bloquear este *thread* e ao mesmo tempo (de forma indivisível) libertar o mutex `mut`. Quando um outro *thread* sinalizar a variável de condição `var`, este *thread* é colocado pronto a executar; no entanto antes do serviço `pthread_cond_wait()` retornar terá de novamente adquirir o mutex `mut`. Assim, quando

`pthread_cond_wait()` retorna, o *thread* está garantidamente de posse do mutex `mut`.

Um outro *thread* que modifique de alguma maneira as variáveis `x` ou `y`, possibilitando assim que o estado do predicado que envolve `x` e `y` possa mudar, terá obrigatoriamente de sinalizar a variável de condição `var`, permitindo assim que um *thread* bloqueado em `var` possa novamente testar a condição. O código para fazer isso poderá ser:

```
pthread_mutex_lock(&mut);  /*quando o outro thread bloqueou, libertou mut*/
... ...  /* modifica o valor de x ou y ou ambos */
pthread_cond_signal(&var);                    /* sinaliza a variável var */
pthread_mutex_unlock(&mut);    /* permite que o outro thread adquira mut */
```

Notar que quando um *thread* sinaliza a variável de condição isso não significa que o predicado se satisfaça; daí a necessidade do ciclo `while()` no código que testa o predicado (ver atrás).

7.3.1. Como usar as variáveis de condição e seus serviços

Uma variável de condição é simplesmente uma variável declarada como pertencendo ao tipo `pthread_cond_t` que está definido em `pthread.h`. Da mesma forma que as variáveis que representam os mutexes, as variáveis de condição também têm de ser inicializadas antes de poderem ser utilizadas. A inicialização faz-se de forma em tudo semelhante à que já foi descrita para os mutexes; ou seja, uma variável de condição pode ser inicializada quando da sua declaração, usando a constante pré-definida `PTHREAD_COND_INITIALIZER`, ou então após a declaração, com o serviço:

```
#include <pthread.h>

int pthread_cond_init(pthread_cond_t *cvar,
                      const pthread_condattr_t *attr);
```

`cvar` - endereço da variável de condição a inicializar;
`attr` - endereço de uma variável de atributos; para inicializar com os atributos por defeito deverá usar-se aqui o valor `NULL`.

Retorna 0 se OK, ou um código de erro positivo no caso contrário.

Após a inicialização as variáveis de condição podem ser usadas através dos serviços descritos a seguir. Uma variável de condição tem sempre um mutex associado, como se viu nos exemplos acima. Esse mutex é utilizado no serviço `pthread_cond_wait()`, e por isso terá de lhe ser passado.

```
#include <pthread.h>

int pthread_cond_wait(pthread_cond_t *cvar, pthread_mutex_t *mptr);
int pthread_cond_signal(pthread_cond_t *cvar);
int pthread_cond_broadcast(pthread_cond_t *cvar);
int pthread_cond_destroy(pthread_cond_t *cvar);
```

`cvar` - endereço da variável de condição a inicializar;
`mptr` - endereço do mutex associado.

Retorna 0 se OK, ou um código de erro positivo no caso contrário.

O serviço `pthread_cond_broadcast()` desbloqueia todos os *threads* que nesse momento estão bloqueados na variável **`cvar`**, em vez de desbloquear apenas um *thread* como faz o serviço `pthread_cond_signal()`. No entanto como os *threads* desbloqueados terão de adquirir o mutex antes de prosseguirem, só um deles o poderá fazer. Os outros seguem-se-lhe à medida que o mutex for sendo libertado.

O serviço `pthread_cond_destroy()` deverá ser chamado quando não houver mais necessidade de utilizar a variável de condição por parte de nenhum dos *threads*.

Exemplo:

Thread 1:

```
#include <pthread.h>

pthread_cond_t cvar=PTHREAD_COND_INITIALIZER;
pthread_mutex_t mut=PTHREAD_MUTEX_INITIALIZER;
int x, y, …;      /* variáveis globais */

…
pthread_mutex_lock(&mut);
while (x != y)
    pthread_cond_wait(&cvar, &mut);
…
pthread_mutex_unlock(&mut);
…
pthread_cond_destroy(&cvar);
…
```

Thread 2:

```
…
pthread_mutex_lock(&mut)
x++;
… /* outras ops em vars comuns */
pthread_cond_signal(&var);
pthread_mutex_unlock(&mut);
…
```

7.4. Semáforos

Todos os sistemas Unix derivados do UNIX System V têm uma implementação de semáforos semelhante à que já vimos para a memória partilhada. A norma POSIX, muito recentemente definiu uma adenda também para a implementação de semáforos. No entanto, devido ao pouco tempo da sua existência, esta adenda ainda não foi implementada em alguns dos sistemas Unix actuais.

Passemos a descrever a implementação do UNIX System V. Nesta implementação é necessário criar os semáforos com um serviço próprio e obter um identificador (*semid*). Seguidamente é necessário inicializá-los com um valor positivo ou zero (geralmente pelo processo que os cria), e só depois é possível utilizá-los (usando as operações *wait*, *signal* e outras) através do respectivo identificador. Por fim é necessário libertar explicimente os semáforos criados. (Cada sistema suporta um número máximo de semáforos; quando se atinge esse número não é possível criar mais, sem libertar alguns).

A criação de um conjunto de semáforos faz-se usando o serviço:

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

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

Retorna um valor positivo (identificador do conjunto) no caso de sucesso ou -1 no caso de erro.
```

Os parâmetros **`key`** e **`flag`** têm exactamente o mesmo significado que foi já descrito para

o serviço `shmget()` (é apenas necessário substituir as constantes `SHM_R` e `SHM_W` por `SEM_R` e `SEM_A` (*alter*)). Este serviço cria um conjunto de semáforos contituído por um número de semáforos indicado em **`nsems`**, que deverá ser maior ou igual a 1. Retorna um identificador do conjunto que deverá ser utilizado nos outros serviços de manipulação dos semáforos.

Após a criação é necessários inicializar os semáforos que fazem parte do conjunto. Isso é feito com o serviço `semctl()`, que também executa outras acções:

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

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

Retorna -1 no caso de erro; no caso de sucesso depende de **cmd**.

Para alguns dos "comandos" (que são especificados em **`cmd`**) deste serviço usa-se o parâmetro **`arg`** que é uma união entre 3 entidades, cada uma delas usada em "comandos" específicos. A união `semun` define-se então como:

```
union semun {
    int val;                    /* para o comando SETVAL */
    struct semid_ds *buf;       /* para os comandos IPC_STAT e IPC_SET */
    unsigned short *array;      /* para os comandos GETALL e SETALL */
};
```

O parâmetro **`semid`** especifica o identificador de um conjunto de semáforos obtido com `semget()`, enquanto que **`semnum`** espeficica qual o semáforo do conjunto, a que se refere o "comando" **`cmd`**. (**`semnum`** é um valor entre 0 e **`nsems`**`-1`). **`cmd`** pode ser um dos seguintes valores:

- `IPC_STAT` - Preenche a estrutura apontada por **`arg.buf`** com informações acerca do conjunto de semáforos (permissões, *pid* do owner, instante de criação, número de processos bloqueados, etc) (ver `man`).
- `IPC_SET` - Através dos campos `sem_perm.uid`, `sem_perm.gid` e `sem_perm.mode` da estrutura apontada por **`arg.buf`** permite modificar as permissões, o dono e o grupo do conjunto de semáforos. (ver `man`).
- `IPC_RMID` - Remove o conjunto de semáforos do sistema. Esta remoção é imediata. Os processos bloqueados em semáforos do conjunto removido ficam prontos a executar (mas a respectiva operação de *wait* retorna um erro). Este comando só pode ser executado por um processo cujo dono o seja também do conjunto de semáforos.
- `GETVAL` - Faz com que o serviço retorne o valor actual do semáforo **`semnum`**.
- `SETVAL` - Inicializa o semáforo indicado em **`semnum`** com o valor indicado em **`arg.val`**.
- `GETNCNT` - Faz com que o serviço retorne o número de processos bloqueados em operações de *wait* no semáforo **`semnum`**.
- `GETZCNT` - Faz com que o serviço retorne o número de processos bloqueados à espera que o semáforo **`semnum`** se torne 0.
- `GETALL` - Preenche o vector apontado por **`arg.array`** com os valores actuais de todos os semáforos do conjunto. O espaço de memória apontado por **`arg.array`** deverá ser suficiente para armazenar **`nsems`** valores *unsigned short*.
- `SETALL` - Inicializa todos os semáforos do conjunto com os valores indicados no vector **`arg.array`**. Este vector deverá conter pelo menos **`nsems`** valores *unsigned short*.

Os valores com que se inicializam os semáforos devem ser maiores ou iguais a 0.

As operações sobre os semáforos de um conjunto executam-se com o serviço `semop()`. É possível especificar várias operações numa só chamada. Todas as operações especificadas numa única chamada são executadas atomicamente. O serviço `semop()` define-se como:

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

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

Retorna 0 no caso de sucesso e -1 no caso de erro.

O parâmetro **`semid`** especifica o identificador do conjunto de semáforos. As operações a realizar sobre semáforos do conjunto, especificam-se no vector **`semoparray`**, cujo número de elementos se indica em **`nops`**. Cada operação é especificada num elemento diferente de **`semoparray`**. Cada elemento de **`semoparray`**, por sua vez, é uma estrutura do tipo `struct sembuf`, que se define como:

```
struct sembuf {
    unsigned short sem_num;     /* número do semáforo no conjunto */
    short sem_op;         /* tipo de operação a realizar sobre o semáforo */
    short sem_flg;              /* uma das flags IPC_NOWAIT e/ou SEM_UNDO */
};
```

Cada operação afecta apenas um semáforo do conjunto, que é especificado no campo `sem_num`; o campo `sem_flg` pode conter o valor `SEM_UNDO`, que indica que a operação especificada deve ser "desfeita" no caso do processo que a executa terminar sem ele próprio a desfazer, e/ou o valor `IPC_NOWAIT`, que indica que se a operação fosse bloquear o processo (p. ex. no caso de um *wait*), não o faz, retornando o serviço `semop()` um erro. O campo *sem_op* indica a operação a realizar através de um valor negativo, positivo ou zero:

- negativo - corresponde a uma operação de *wait* sobre o semáforo; o módulo do valor indicado é subtraído ao valor do semáforo; se o resultado for positivo ou 0 continua-se; se for negativo, o processo é bloqueado e não se mexe no valor do semáforo;
- positivo - corresponde a uma operação de *signal* sobre o semáforo; o valor indicado é somado ao valor do semáforo; o sistema verifica se algum dos processos bloqueados no semáforo pode prosseguir;
- zero - significa que este processo deseja esperar que o semáforo se torne 0; se o valor do semáforo for 0 prossegue-se imediatamente; se for maior do que 0 o processo bloqueia até que se torne 0.

Exemplo: Utilização de um semáforo

```
...
key_t key;
int semid;
union semun arg;
struct sembuf semopr;
...
key = ftok("myprog", 1);
```

```
semid = semget(key, 1, SEM_R | SEM_A);      /* cria 1 semáforo */
arg.val = 1;
semctl(semid, 0, SETVAL, arg);              /* inicializa semáforo com 1 */
semopr.sem_num = 0;
semopr.sem_op = -1;                         /* especifica wait */
semopr.sem_flg = SEM_UNDO;
semop(semid, &semopr, 1);                   /* executa wait */
.....                                       /* secção crítica */
semopr.sem_op = 1;                          /* especifica a op. signal */
semop(semid, &semopr, 1);                   /* executa op. signal */
...
semctl(semid, 0, IPC_RMID, arg);            /* liberta semáforo */
...
```

Muito importante:
Existem geralmente 2 comandos, que correm na *shell* do sistema operativo, que permitem listar e libertar semáforos e blocos de memória partilhada quando os processos que os criaram o não fizeram. São eles *ipcs* e *ipcrm* (ver `man`).