ISCTE IUL
Instituto Universitário de Lisboa
Escola de Tecnologias e Arquitectura
Dep. de Ciências e Tecnologias da Informação

Sistemas Operativos

Ano letivo 2013/2014
1º Semestre

# Conteúdo

# System V - IPCs

Este capítulo apresenta os mecanismos de comunicação entre processos normalmente designado de "IPCs do Sistema V", designadamente filas de mensagens, semáforos e memória partilhada. As filas de mensagens permitem aos processos trocarem mensagens entre si. Os semáforos permitem a sincronização (coordenação de acções) entre processos. E a memória partilhada permite a partilha de dados (variáveis) entre processos, normalmente recorrendo à ajuda de semáforos para disciplinar o acesso simultâneo aos dados.

## 1 Filas de mensagens

Uma fila de mensagens pode ser descrita como lista, composta por várias mensagens, guardada num espaço de endereçamento pertencente ao núcleo do Sistema Operativo. Vários processos podem enviar mensagens para a fila e vários outros processos podem ler essas mensagens. Sempre que uma mensagem é lida, é removida da fila.

Cada mensagem pode ter associada a si um número inteiro longo não negativo, que se identifica como sendo o tipo da mensagem, que permite que um dado processo receptor possa esperar apenas mensagem de um determinado tipo. Permite ainda estabelecer esquemas baseados em prioridades nas filas de mensagens.

Quando um processo tenta escrever uma mensagem numa fila cheia, bloqueia até que exista espaço suficiente na fila. De modo semelhante, um processo bloqueia quando tenta ler mensagens de um dado tipo e não existe nenhuma mensagem desse tipo na fila de mensagens.

### 1.1 Criação de uma fila de mensagens

Considere o seguinte programa:

```
#include <stdio.h>
#include <stdlib.h>
#include <sys/ipc.h>
#include <sys/msg.h>
main() {
    int id = msgget ( 1000, IPC_CREAT | 0666 );

    if ( id < 0 ) {
        printf ("Erro na criação da fila de mensagens\n" );
        exit(1);
    }
    printf ("Criada fila de mensagens com id %d\n", id);
}
```

O programa usa a função `msgget` para criar uma fila de mensagens (ou "mailbox").

A função `msgget` recebe dois argumentos. O primeiro é uma chave que tem de ser única para cada fila de mensagens criada no sistema. Neste exemplos usamos normalmente esta chave 1000; adiante voltaremos a este assunto.

No segundo argumento indicam-se duas coisas: `IPC_CREAT`, que pede para criar a fila de mensagens; e 0666 que indica as permissões da fila de mensagens.

As permissões têm uma leitura semelhante ao das permissões no sistema de ficheiros. Neste caso o (número octal) 666 indica o equivalente a rw- ou seja a permissão de "read" e "write" quer para o dono da fila de mensagens quer para todos os outros utilizadores.

Caso a fila de mensagens seja criada com sucesso a função `msgget` devolve um identificador que depois poderá ser usado para enviar ou receber mensagens através da mailbox. Em caso de erro devolve um número negativo; nesse caso o programa publica uma mensagem de erro e termina.

## 1.2 Comandos `ipcs`

A fila de mensagem é um recurso permanente mantido pelo kernel do sistema operativo. Uma vez criado permanece até se pedir ao sistema para o eliminar (ou até o sistema ser desligado).

Ou seja, caso o programa anterior tenha sucesso é criada uma fila de mensagens que fica no sistema. O comando `ipcs` permite listar as filas de mensagens existentes. Fazendo o comando

```
ipcs -q
```

poderá aparecer um output do semelhante a este:

```
------ Message Queue ------
key        msqid      owner      perms      bytes      msg
0x000003e8 0          jrg        666        0          0
```

A coluna key mostra obviamente a chave usada para criar a fila de mensagens (1000 no exemplo, nesta lista representado em hexadecimal). A coluna msqid mostra o id da fila, o mesmo que recebemos da função `msgget`.

O comando ipcrm permite eliminar uma fila de mensagens. Por exemplo, o comando

```
ipcrm -q 0
```

elimina a fila de mensagens com o id dado (neste caso 0). Se fizermos agora

```
ipcs -q
```

a lista aparecerá vazia. Se executarmos o programa outra vez a fila voltará a ser criada, com outro id. Fazendo o `ipcs` a nova fila poderá aparecer, por exemplo:

```
------ Message Queue ------
key        msqid      owner      perms      bytes      msg
0x000003e8 32768      jrg        666        0          0
```

## 1.3 Macro `exit_on_error`

Após uma chamada ao sistema devemos sempre verificar se correu bem ou se deu erro.

A generalidade das chamadas ao sistema devolvem um número negativo quando não têm sucesso. Sendo assim, a seguir a uma chamada ao sistema faremos uma construção do género

```
n = chamada_ao_sistema ( ...);
if ( n < 0 ) {
  // erro: publicar uma mensagem de erro e sair
  printf ("... erro....");
  exit( ... );
}
// ! sucesso : continuar o programa
```

Para a mensagem de erro, em vez de um `printf` formado por nós, podemos usar a função `perror` que mostra uma mensagem de erro produzida pelo sistema operativo.

Traduzimos este arranjo na seguinte macro

```
#define exit_on_error(s,m) if ( s < 0 ) { perror(m); exit(1); }
```

que passaremos a usar normalmente após cada chamada ao sistema.

Com esta macro o programa anterior ficaria:

```
#include <stdio.h>
#include <stdlib.h>
#include <sys/ipc.h>
#include <sys/msg.h>
#define exit_on_error(s,m) if ( s < 0 ) { perror(m); exit(1); }

main() {
    int id = msgget ( 1000, IPC_CREAT | 0666 );
    exit_on_error ( id, "Criação da fila de mensagens");
    printf ("Criada fila de mensagens com id %d\n", id);
}
```

## 1.4 Header file

É possivel definir um ficheiro que inclui todos os #includes e eventuais macros, de forma a não ter de escrever tudo isso num novo programa. Por exemplo, se escrever o ficheiro definicoes.h com o seguinte conteúdo

```
#include <stdio.h>
#include <stdlib.h>
#include <sys/ipc.h>
#include <sys/msg.h>
#define exit_on_error(s,m) if ( s < 0 ) { perror(m); exit(1); }
```

o programa anterior pode ser re-escrito como

```
#include "definicoes.h"
main() {
    int id = msgget ( 1000, IPC_CREAT | 0666 );
    exit_on_error ( id, "Criação da fila de mensagens");
    printf ("Criada fila de mensagens com id %d\n", id);
}
```

## 1.5 Envio de uma mensagem

Considere o seguinte exemplo:

```c
#include "definicoes.h"
#include <string.h>

typedef struct {
    long tipo;
    char texto[250];
} MsgStruct;

main() {
    int msg_id;
    int status;
    MsgStruct msg;

    // ligar à fila de mensagens
    msg_id = msgget ( 1000, 0600 | IPC_CREAT );
    exit_on_error (msg_id, "Criação/Ligação");

    // enviar ums mensagem
    msg.tipo = 1;
    strcpy( msg.texto, "Hello");
    status = msgsnd( msg_id, &msg, sizeof(msg.texto), 0);
    exit_on_error (status, "Envio");

    printf ("Mensagem enviada!\n");
}
```

## Ligação

O primeiro passo para poder usar a fila de mensagens é ligar-se a essa fila; para isso usa-se, também, a função `msgget`. Digamos que a função `msgget` serve para duas coisas: para criar uma fila de menagens ou para fazer a ligação a uma fila de mensagens previamente criada.

Na prática a chamada do exemplo serve para as duas coisas porque a opção `IPC_CREAT` faz com que a função crie a fila de mensagem caso não exista. Se quisermos apenas ligar à fila (e não criar a fila caso não exista) omitimos o `IPC_CREAT`:

```c
msg_id = msgget( 1000, 0666 );
```

Neste caso a função daria erro se não existisse uma fila criada com chave 1000.

## Estrutura de uma mensagem

Para se poder enviar uma mensagem é preciso criar uma estrutura como a que se ilustra no exemplo:

```c
typedef struct {
    long tipo;
    char texto[250];
} MsgStruct;
```

A estrutura contém dois campos: o primeiro, do tipo long, é obrigatório. Podemos escolher o nome, mas é obrigatório que seja o primeiro e que seja do tipo `long`. Mais adiante veremos a utilidade. O outro contém o espaço para o envio da mensagem propriamente dita, neste caso uma string contendo até 250 caracteres.

A mensagem a enviar é então constituída por uma variável do tipo `MsgStruct` que no exemplo é preenchida da seguinte forma:

```
msg.tipo = 1;
strcpy( msg.texto, "Hello");
```

Notar o tipo = 1 que é relevante para, posteriormente, se pode receber ("ir buscar") a mensagem.

### Envio

O envio propriamente dito é feito chamando a função `msgsnd` ("message send"). O msgsnd recebe como argumentos o id da fila de mensagens, a variável com a mensagem (na realidade o endereço dessa variável, formado pelo operador &) e o tamanho da mensagem. Note que o tamanho é o sizeof, neste caso 250 (e não o strlen, que neste caso seria 5). No último argumento, de momento, pomos sempre 0.

A mensagem enviada fica na fila até alguém a receber ("ir buscar"). Podemos aliás um reflexo disso vendo o resultado do `ipcs -q` que agora dará alguma coisa do género:

```
----- Message Queue ------
key          msqid     owner        perms     bytes        msg
0x000003e8   32768      jrg          666      250           1
```

indicando que há uma mensagem em espera na fila.

Se executar o programa de novo mandará outra mensagem; ficarão, então, duas mensagens na fila à espera de serem levantadas.

## 1.6 Recepção de uma mensagem

Considere o seguinte exemplo:

```
#include "definicoes.h"
#include <string.h>

typedef struct {
    long tipo;
    char texto[250];
} MsgStruct;

main() {
    int msg_id;
    int status;
    MsgStruct msg;

    // ligar à fila de mensagens
    msg_id = msgget ( 1000, 0 );
    exit_on_error (msg_id, "Criação/Ligação");

    // receber uma mensagem (bloqueia se não houver)
    status = msgrcv( msg_id, &msg, sizeof(msg.texto), 1, 0);
    exit_on_error (status, "Recepção");

    printf ("MENSAGEM <%s>\n", msg.texto);
}
```

O programa usa a função `msgrcv` para receber (ir buscar) umas das mensagens que estejam na fila. Ao executar este programa deve aparecer o ecrã

```
MENSAGEM <Hello>
```

Os argumentos da função `msgrcv` são até certo ponto parecidos com os do `msgsnd`: primeiro o id da fila, depois uma variável com a estrutura da mensagem e o respectivo tamanho. A diferença é que, agora, a variável msg vai ser preenchida com a mensagem extraída da fila.

Há ainda um argumento importante, o último que deve aderir ao tipo da mensagem a recolher. Neste caso a mensagem foi enviada com o campo tipo = 1 e, em correspondência, indicamos neste argumento que queremos receber uma mensagem enviada com o tipo 1.

## 1.7 Sincronização

A fila de mensagens é mais do que apenas um mecanismo que permite envio e recepção de mensagens. É também um mecanismo de sincronização, ou seja, um mecanismo que permite ordenar uma sequência de acções.

O ponto fundamental é este: o `msgrcv` bloqueia caso não haja nenhuma mensagem na fila. Quer dizer: se não há nenhuma mensagem na fila, o programa que faz `msgrcv` fica parado até que apareça uma nova mensagem - ou seja, até que outro programa faça um `msgsnd`.

Experiência: execute o programa do ponto 1.4 (aqui designado "enviar") numa janela e o programa do ponto 1.5 (aqui designado "Receber") noutra janela. Partindo da fila de mensagens vazia verifique a seguinte sequência

| Janela 1 | Janela 2 | mgs na fila |
|---|---|---|
| | | 0 |
| ./Enviar | | 1 |
| | ./Receber | 0 |
| | ./Receber -> bloqueia | 0 |
| ./Enviar | -> desbloqueia | 0 |
| ./Enviar | | 1 |
| ./Enviar | | 2 |
| ./Enviar | | 3 |
| | ./Receber | 2 |
| | ./Receber | 1 |
| | ./Receber | 0 |
| | ./Receber -> bloqueia | |

Assim, as filas de mensagens podem servir quer para diferentes programas (ou mais rigorosamente "processos") comunicarem entre si trocando mensagens, quer para sincronizarem as suas acções.

Imaginemos a arquitectura típica correspondente ao conceito "cliente"/"servidor". A ideia deste conceito é a cooperação entre dois tipos de processos: o "cliente" que submete pedidos; e o "servidor" que recebe e executa esses pedidos.

A ideia do "servidor" corresponde seguinte ciclo lógico:

```
repetir
```

```
esperar pedido
receber pedido
executar pedido
```

ou seja, um "ciclo infinito" em que o servidor está à espera de um pedido, quando recebe um pedido executa-o, e repete ficando à espera do próximo.

O "pedido" corresponde a uma mensagem colocada pelo cliente no fila de mensagens. O servidor está continuamente à espera que apareça uma mensagem; quando isso suceder executa-a e repete ficando à espera da seguinte. A ideia é traduzida no seguinte exemplo:

```
#include "definicoes.h"

typedef struct {
    long tipo;
    char texto[250];
} MsgStruct;

main() {
    //ligar à fila de mensagens
    int msg_id;
    msg_id = msgget ( 1000, 0666 | IPC_CREAT );
    exit_on_error (msg_id, "Criação/Ligação");

    int status;
    MsgStruct msg;

    while ( 1 ) {
        printf ("Esperar pedido...\n");

        status = msgrcv( msg_id, &msg, sizeof(msg.texto), 1, 0);
        exit_on_error (status, "Recepção");

        printf ("Recebido pedido <%s>. Executar!\n", msg.texto);
    }
}
```

Colocando este programa em execução ele ficará continuamente à espera do aparecimento de uma mensagem no fila de mensagens (o envio pode ser simulado com o programa do ponto 1.4) a que responderá com a mensagem ".... Executar!".

## 1.8 Estruturação das mensagens

Considere o seguinte exemplo:

```c
#include "definicoes.h"

typedef struct {
    long tipo;
    struct {
        int  num;
        char nome[100];
        int  nota;
    } msg;
} MsgStruct;

main() {
    //ligar à fila de mensagens
    int msg_id;
    msg_id = msgget ( 1000, 0666 | IPC_CREAT );
    exit_on_error (msg_id, "Criação/Ligação");

    int status;
    MsgStruct m;

    while ( 1 ) {
        printf ("Esperar pedido...\n");

        status = msgrcv( msg_id, &m, sizeof(m.msg), 1, 0);
        exit_on_error (status, "Recepção");

        FILE *fp = fopen ("pauta.txt", "a" );
        fprintf(fp,"%d\n%s\n%d\n", m.msg.num, m.msg.nome, m.msg.nota);
        fclose(fp);
    }
}
```

Neste exemplo a mensagem tem um conteúdo estruturado, composto por um conjunto de campos. A estrutura `StructMsg` continua a incluir o primeiro campo obrigatório que define o tipo da mensagem; o conteúdo da mensagem é definido numa estrutura interna.

O exemplo represente um servidor que recebe mensagens, representando entradas de uma pauta, que vai descarregando para um ficheiro. O exemplo seguinte apresenta um programa que manda mensagens para o servidor:

```c
#include "definicoes.h"
#include <string.h>

typedef struct {
    long tipo;
    struct {
        int  num;
        char nome[100];
        int  nota;
    } msg;
} MsgStruct;

int limpar_enter()  {
    while ( getchar() != '\n' );
}

main() {
    //ligar à fila de mensagens
    int msg_id;
    msg_id = msgget ( 1000, 0666 | IPC_CREAT );
    exit_on_error (msg_id, "Criação/Ligação");

  int status;
    MsgStruct m;
    char s[100];
    m.tipo = 1;

    printf ("Numero: ");
    scanf ("%d", &(m.msg.num) );
    limpar_enter();

    printf ("Nome: ");
    fgets ( m.msg.nome, 100, stdin );
    m.msg.nome[ strlen(m.msg.nome) -1 ] = 0;

    printf ("Nota: ");
    fgets (s, 100, stdin );
    sscanf (s, "%d", &(m.msg.nota) );

    status = msgsnd( msg_id, &m, sizeof(m.msg), 0);
    exit_on_error (status, "Recepção");
}
```

## 1.9 Encaminhamento

O tipo das mensagens (que até agora deixámos sempre como 1) permite fazer o encaminhamento das mensagens (diferenciar emissores e destinatários).

Exemplo: pretendemos construir um modelo em que um servidor P1 recebe mensagens de diferentes clientes P2, P3, etc. e responde, individualmente, a cada um deles.

Para construir este modelo fazemos o seguinte plano baseado nos tipos:

- P2, P3, ... enviam a P1 mensagens do tipo 1;

no conteúdo da mensagem indicam o emissor;

esperam como resposta mensagens dos tipos 2, 3, ...;

- P1 espera mensagens do tipo 1;

responde com mensagens do tipo 2, 3, ... conforme o emissor indicado no conteúdo.

O programa seguinte ilustra o servidor P1:

```c
#include "definicoes.h"
#include <string.h>

typedef struct {
    long tipo;
    struct {
        int emissor;
        char texto[250];
    } msg;
} MsgStruct;

main() {
    int status;
    MsgStruct m;
    int msg_id;

    msg_id = msgget ( 1000, 0666 | IPC_CREAT );
    exit_on_error (msg_id, "Criação/Ligação");

    while ( 1 ) {
        printf ("Esperar...\n");

        status = msgrcv( msg_id, &m, sizeof(m.msg), 1, 0);
        exit_on_error (status, "Recepção");

        m.tipo = m.msg.emissor;
        sprintf ( m.msg.texto, "Hello P%d\n", m.msg.emissor);

        printf ("Resposta: %s\n", m.msg.texto);

        status = msgsnd( msg_id, &m, sizeof(m.msg), 0);
        exit_on_error (status, "Envio");
    }
}
```

O programa seguinte apresenta o servidor P2. Os outros seriam semelhantes usando o tipo de mensagem 3, 4, ...

```c
#include "definicoes.h"
#include <string.h>

main() {
    int status;
    MsgStruct m;
    int msg_id;

    msg_id = msgget ( 1000, 0666 | IPC_CREAT );
    exit_on_error (msg_id, "Criação/Ligação");

    m.tipo = 1;
    m.msg.emissor = 2;

    status = msgsnd( msg_id, &m, sizeof(m.msg), 0);
    exit_on_error (status, "Envio");

    status = msgrcv( msg_id, &m, sizeof(m.msg), 2, 0);
    exit_on_error (status, "Recepção");

    printf ("Resposta: %s\n", m.msg.texto);
```

```
}
```

O cliente pode-se generalizar para usar como tipo de mensagem o próprio pid. Este é uma forma expedita de atribuir a cada processo um número genérico e seguramente não repetido.

```
#include "definicoes.h"
#include <string.h>

main() {
    int status;
    MsgStruct m;

    msg_id = msgget ( 1000, 0666 | IPC_CREAT );
    exit_on_error (msg_id, "Criação/Ligação");

    m.tipo = 1;
    m.msg.emissor = getpid();

    status = msgsnd( msg_id, &m, sizeof(m.msg), 0);
    exit_on_error (status, "Envio");

    status = msgrcv( msg_id, &m, sizeof(m.msg), getpid(), 0);
    exit_on_error (status, "Recepção");

    printf ("Resposta: %s\n", m.msg.texto);
}
```

# 2 Semáforos

Os semáforos são um dos mecanismos de comunicação entre processos, que permitem a sincronização de processos. Estes são habitualmente usados para garantir a exclusão mútua de processos em recursos partilhados.

As operações atómicas básicas com semáforos são:

- UP – Incrementa o valor do semáforo em uma unidade
- DOWN – Tenta decrementar o valor do semáforo uma unidade , cansando um bloqueio do 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. Ou seja, o valor de um semáforo é sempre positivo.
- ZERO - Um processo que efectua esta operação permanece bloqueado até que o valor do semáforo seja igual a zero

## 2.1 Criação e inicialização

Considere que o ficheiro `definicoes.h` contém

```
#include <stdio.h>
#include <stdlib.h>
#include <sys/ipc.h>
#include <sys/sem.h>
#define exit_on_error(s,m) if ( s < 0 ) { perror(m); exit(1); }
```

O programa seguinte cria e inicializa um semáforo

```
#include "definicoes.h"

main() {
    int id = semget ( 1000, 1, IPC_CREAT | 0666 );
    exit_on_error (id, "Criação/Ligação");

    printf ("Criado semáforo com id %d\n", id);

    int status = semctl ( id, 0, SETVAL, 0);
    exit_on_error (status, "Inicialização");
}
```

Há algumas semelhanças de nomenclatura com as filas de mensagens. A criação do semáforo é feita pela função `semget`, que tem como primeiro argumento uma chave. Esta chave deve ser única; tal como nos exemplos anteriores usamos 1000 como chave.

O semáforo é também um recurso permanente gerido pelo sistema operativo. Para ver os semáforos criados podemos fazer o comando:

```
ipcs -s
```

A função `semctl` é usada no exemplo, para inicializar o semáforo. Neste caso o semáforo é inicializado com o valor 0 (o terceiro argumento da função).

## 2.2 Operações (up e down)

Os semáforos implementam o conceito apresentado nas aulas teóricas: um semáforo é uma variável, gerida pelo do sistema operativo, com valor >=, sobre a qual são se podem realizar duas operações:

- UP(x) : aumenta 1 ao valor de x;
- DOWN(x): diminui 1 ao valor de x; bloqueia enquanto não puder fazê-lo.

O sublinhado é o ponto importante: um semáforo tem um valor >=0; se o valor for 0 e um processo tentar fazer um down não pode; nessas condições fica bloqueado até poder fazê-lo (ou seja, até que outro processo faça um up!).

Os exemplos seguintes implementam funções UP e DOWN. O programa P1 faz um *down*. A operação é realizada pela função `semop`, que recebe como argumento a estrutura DOWN. O programa P2 faz um *up*. A operação é realizada pela função `semop`, que recebe como argumento a estrutura UP.

P1)

```
#include "definicoes.h"

struct sembuf UP = { 0, 1, 0 } ;
struct sembuf DOWN = { 0, -1, 0 };
main() {
  int sem_id;
  int status;
  sem_id = semget ( 1000, 1, 0600 );
  exit_on_error (sem_id, "Criação/Ligação");

  printf ("Esperar...\n");
```

```
    // DOWN() - para sair daqui alguém tem que fazer UP()
    status = semop ( sem_id, &DOWN, 1 );
    exit_on_error (status, "DOWN");

    printf ("Ok\n");
  }
```

P2)

```
  #include "definicoes.h"

  struct sembuf UP = { 0, 1, 0 } ;
  struct sembuf DOWN = { 0, -1, 0 } ;
  main() {
    int sem_id;
    int status;

    //ligar ao semaforo
    sem_id = semget ( 1000, 1, 0600 );
    exit_on_error (sem_id, "Criação/Ligação");

    // UP()
    status = semop ( sem_id, &UP, 1 );
    exit_on_error (status, "UP");

    printf ("Ok\n");
  }
```

A estrutura sembuf inclui os parâmetros necessários à execução da operação. Se fizer "man semop", poderá verificar que a estrutura sembuf inclui três elementos: o primeiro corresponde ao número do semáforo; o segundo corresponde ao valor que se pretende adicionar ao semáforo; e o terceiro são apenas opções extra, raramente usadas e portanto normalmente matém o valor 0.

```
struct sembuf {
            u_short sem_num;        /* semaphore # */
            short   sem_op;         /* semaphore operation */
            short   sem_flg;        /* operation flags */
};
```

Experiência: execute o programa de inicialização do ponto 1.1, deixando portanto o semáforo com o valor 0. Se, de seguida, executar P1, este ficará bloqueado no down. Executando (noutra janela) o programa P2 este porá o semáforo a 1, permitindo assim a P1 desbloquear e prosseguir. O valor final do semáforo será 0.

Experiência: execute o programa de inicialização do ponto 1.1, deixando portanto o semáforo com o valor 0. De seguida executa 3 vezes o programa P2, deixando portanto o semáforo com valor 3. De seguida poderá executar P1 também 3 vezes; à 4ª vez P1 bloqueia.

## 2.3 Exemplo: cliente / servidor

No exemplo seguinte temos vários processos clientes sobem o valor do semáforo e um processo servidor que baixa o valor do semáforo. Verifique a execução de vários clientes e um servidor.

s1.c

```
#include "definicoes.h"

struct sembuf UP = { 0, 1, 0 } ;
struct sembuf DOWN = { 0, -1, 0 } ;

main() {
    int sem_id;
    int status, i = 1;
    sem_id = semget ( 1010, 1, 0666 | IPC_CREAT );
    exit_on_error (sem_id, "Criação/Ligação");

    while ( 1 ) {
      status = semop ( sem_id, &DOWN, 1 );
      exit_on_error (status, "DOWN");
      printf ("passagem %d\n", i++);
    }
}
```

c1.c

```
#include "definicoes.h"

struct sembuf UP = { 0, 1, 0 } ;
struct sembuf DOWN = { 0, -1, 0 } ;

main() {
    int sem_id;
    int status;
    sem_id = semget ( 1010, 1, 0666 | IPC_CREAT );
    exit_on_error (sem_id, "Problema de Criação/Ligação");

    status = semop ( sem_id, &UP, 1 );
    exit_on_error (status, "Problema com o UP");
}
```

## 2.4 Exemplo: zona de exclusão

Uma zona de exclusão é uma secção do programa com acesos condicionados a um processo de cada vez. Os exemplos seguintes simulam uma zona de exclusão. O programa de inicialização P1 cria um semáforo e inicializa-o com 1.

```
#include "definicoes.h"


main() {


    int id = semget ( 1000, 1, IPC_CREAT | 0666 );
    exit_on_error (id, "Criação/Ligação");


    printf ("Criado semáforo com id %d\n", id);


    int status = semctl ( id, 0, SETVAL, 1);
    exit_on_error (status, "Inicialização");
}
```

O programa P2 simula uma zona de exclusão.

```
#include "definicoes.h"

struct sembuf UP = { 0, 1, 0 } ;
struct sembuf DOWN = { 0, -1, 0 } ;

main() {
    int sem_id;
    int status;
    sem_id = semget ( 1000, 1, 0600 | IPC_CREAT );
    exit_on_error (sem_id, "Criação/Ligação");

    printf ("Entrada..\n");
    status = semop ( sem_id, &DOWN, 1 );
    exit_on_error (status, "DOWN");

    printf ("ENTER para continuar....");
    getchar();

    status = semop ( sem_id, &UP, 1 );
    exit_on_error (status, "DOWN");
    printf ("Saída\n");
}
```

Experiência: execute o programa de inicialização P1. Execute P2 em várias janelas; verifique apenas um dos processos entra, de cada vez, na zona de exclusão (simulada pelo "ENTER para continuar...").

## 2.5 Exemplo: "brancas e pretas"

O exemplo seguinte simula uma situação em dois processos executam à vez (como num jogo entre brancas e pretas).

O programa usa dois semáforos. O programa seguinte cria e inicializa os dois semáforos.

```
#include "definicoes.h"

main() {

    int id_brancas = semget ( 1001, 1, IPC_CREAT | 0666 );
    int id_pretas = semget ( 1002, 1, IPC_CREAT | 0666 );

    semctl ( id_brancas, 0, SETVAL, 1);
    semctl ( id_pretas, 0, SETVAL, 0);
}
```

O programa seguinte implementa o ciclo de jogo das brancas: esperam pela sua vez, jogam (simulado pelo "ENTER para continuar") e cedem a vez.

```
#include "definicoes.h"

void down ( id ) {
    struct sembuf DOWN = { 0, -1, 0 } ;
    int status = semop ( id, &DOWN, 1 );
    exit_on_error (status, "DOWN");
}

void up ( id ) {
    struct sembuf UP = { 0, 1, 0 } ;
    int status = semop ( id, &UP, 1 );
```

```
    exit_on_error (status, "UP");
}

main() {

    int id_brancas = semget ( 1001, 1, IPC_CREAT | 0666 );
    int id_pretas = semget ( 1002, 1, IPC_CREAT | 0666 );

    while ( 1 ) {
        down ( id_brancas );
        printf ("Jogam as brancas (ENTER para continar)...");
        getchar();
        up ( id_pretas);
    }
}
```

O programa seguinte implementa o ciclo de jogo das pretas.

```
#include "definicoes.h"

void down ( id ) {
    struct sembuf DOWN = { 0, -1, 0 } ;
    int status = semop ( id, &DOWN, 1 );
    exit_on_error (status, "DOWN");
}

void up ( id ) {
    struct sembuf UP = { 0, 1, 0 } ;
    int status = semop ( id, &UP, 1 );
    exit_on_error (status, "UP");
}


main() {
    int id_brancas = semget ( 1001, 1, IPC_CREAT | 0666 );
    int id_pretas = semget ( 1002, 1, IPC_CREAT | 0666 );

    while ( 1 ) {
        down ( id_pretas );
        printf ("Jogam as pretas (ENTER para continar)...");
        getchar();
        up ( id_brancas);
    }

}
```

## 2.6 Arrays de semáforos

O programa seguinte usa a função semget para criar um "array de semáforos" (neste caso um array com dois semáforos).

```
#include "definicoes.h"

main() {
    int id = semget ( 1007, 2, IPC_CREAT | 0666 );
    exit_on_error (id, "Criação/Ligação");

    printf ("Criado semáforo com id %d\n", id);
```

```
    semctl ( id, 0, SETVAL, 1);
    semctl ( id, 1, SETVAL, 0);
}
```

A diferença para os exemplos anteriores é o segundo argumento do semget que agora pede que o recurso criado seja um array com 2 semáforos (em todos os exemplos anteriores tínhamos usado 1).

Os dois semáforos criados são identificados como os elementos de um array; ou seja, o primeiro é identificado como semáforo nº 0 e o segundo á identificado como semáforo nº 1. A utilização do semctl já reflecte esta nomenclatura. O primeiro setctl inicializa o primeiro semáforo (o nº 0) com o valor 1. O segundo semctl inicializa o segundo semáforo (o nº 1) com o valor 0.

Os programas seguintes implementam uma nova versão do modelo "brancas e pretas" usando, desta vez, um array com dois semáforos (em vez de dois semáforos com chaves distintas, como na secção anterior).

O exemplo seguinte implementa o ciclo das brancas.

```
#include "definicoes.h"

void s_operacao ( int id, int idx, int v ) {
    struct sembuf s;
    s.sem_num = idx;
    s.sem_op = v;
    s.sem_flg = 0;

    int status = semop ( id, &s, 1 );
    exit_on_error (status, "DOWN");
}

main() {
    int id = semget ( 1007, 2, IPC_CREAT | 0666 );

    while ( 1 ) {
        s_operacao ( id, 0 , -1 );
        printf ("Jogam as brancas (ENTER para continar)...");
        getchar();
        s_operacao ( id, 1 , 1 );
    }
}
```

Desta vez é usada uma função genérica s_operacao que recebe o id do semáforo, o número do índice o valor a adicionar ao semáforo: 1 para up e -1 para down.

```
// Pretas
#include "definicoes.h"

main() {
    int id = semget ( 1007, 2, IPC_CREAT | 0666 );

    while ( 1 ) {
        s_operacao ( id, 0 , -1 );
        printf ("Jogam as brancas (ENTER para continar)...");
        getchar();
        s_operacao ( id, 1 , 1 );
    }
}
```

# 3 Memória partilhada

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.

Considere que o ficheiro `definicoes.h` contém

```
#include <stdio.h>
#include <string.h>
#include <stdlib.h>
#include <sys/ipc.h>
#include <sys/shm.h>
#define exit_on_error(s,m) if ( s < 0 ) { perror(m); exit(1); }
#define exit_on_error(s,m) if ( s < 0 ) { perror(m); exit(1); }
```

## 3.1 Criação

Considere o seguinte exemplo:

```
#include "definicoes.h"
main() {
    int id = shmget ( 1000, 100, IPC_CREAT | 0666 );
    exit_on_error (id, "Criação/Ligação");

    printf ("Criada memória partilhada com id %d\n", id);

    void *p = shmat ( id, 0, 0 );
    if ( p == NULL ) {
        printf ("Erro no attach\n");
        exit(1);
    }

    char *s = (char *) p;
    strcpy ( s, "Hello\n" );
}
```

A função `shmget` permite criar um bloco de memória partilhada. Quer dizer: estamos a pedir ao sistema operativo para disponibilizar um bloco de memória e manter o seu conteúdo de forma a poder ser usado por vários programas.

A exemplo dos IPCs anteriores podemos ver a lista de "memórias partilhadas" com o comando ipcs, desta vez com a opção -m. Também a exemplo dos IPCs anteriores, podemos destruir a memória partilhada com o comando ipcrm -m.

O objectivo final é poder usar a memória partilhada da mesma maneira que se usa a memória "normal" do programa, ou seja, sob a forma de variáveis. Para isso são precisas duas coisas:

1º) arranjar uma maneira de chegar à memória partilhada;

2º) definir o tipo de dados que se vão colocar nessa memória (int, char ... ???).

O primeiro passo designa-se attach e é realizado pela chamada à função shmat, que devolve um "ponteiro" para o bloco de memória partilhada. No exemplo o ponteiro devolvido por shmat é guardado na variável p.

A função shmat devolve um ponteiro que, tal como p, é do tipo void *,. Este ponteiro indica a localização da memória partilhada mas não indica o tipo de dados que ela contém, e por isso não pode ser directamente usado para lhe aceder.

No passo seguinte, converte-se o ponteiro void * num ponteiro para um tipo de dados concreto, no caso char *. Isto quer dizer que, a partir daqui, podemos usar a shared memory através do ponteiro p_char tratando-a como contendo chars, ou seja como um array de chars.

No seguimento do exemplo, copia-se "Hello\n" para esse array, inicializando desta forma as primeiras 7 posições.

## 3.2 Outros exemplos de manipulação

Os exemplos seguintes manipulam a mesma memória de 100 bytes do exemplo anterior.

Exemplo 1

O programa seguinte escreve o conteúdo da memória presumindo que contém uma string.

```
#include "definicoes.h"
main() {
    int id = shmget ( 1000, 100, IPC_CREAT | 0666 );
    exit_on_error (id, "Criação/Ligação");

    printf ("Criada memória partilhada com id %d\n", id);

    void *p = shmat ( id, 0, 0 );
    exit_on_null (p, "Attach");

    char *s = (char *) p;
    printf ( "M: %s\n", s );
}
```

Executando este programa a seguir ao da secção anterior o output será:

```
M : Hello
```

Exemplo 2

O exemplo seguinte faz "dump" do conteúdo de cada uma das 100 posições de memória, consideradas ainda como caracteres, escrevendo o código ASCII de cada um desses caracteres.

dump.c

```
#include "definicoes.h"
main() {
    int id = shmget ( 1000, 100, IPC_CREAT | 0666 );
    exit_on_error (id, "Criação/Ligação");

    printf ("Criada memória partilhada com id %d\n", id);

    char *s = shmat ( id, 0, 0 );
    exit_on_null (s, "Attach");

    int i;
    for ( i = 0; i < 100; i++ ) {
        int c = s[i];
```

```
        printf ("%5d", c );
        if ( (i+1) % 10 == 0 ) printf ("\n");
    }
    printf ("\n");
}
```

Executando este programa a seguir ao da secção anterior o output será

```
   72  101  108  108  111   10    0    0    0    0
    0    0    0    0    0    0    0    0    0    0
    0    0    0    0    0    0    0    0    0    0
    0    0    0    0    0    0    0    0    0    0
    0    0    0    0    0    0    0    0    0    0
    0    0    0    0    0    0    0    0    0    0
    0    0    0    0    0    0    0    0    0    0
    0    0    0    0    0    0    0    0    0    0
    0    0    0    0    0    0    0    0    0    0
    0    0    0    0    0    0    0    0    0    0
```

onde se podem ver os caracteres resultantes da inicializaçao feita pelo programa na secção anterior: os caracteres Hello (código ASCII 72, ...) o '\n', códido ASCII 10 e o terminador, código ASCII 0.

Experiência: execute o seguinte programa para inicializar a shared memory

```
#include "definicoes.h"
main() {
    int id = shmget ( 1000, 100, IPC_CREAT | 0666 );
    exit_on_error (id, "Criação/Ligação");

    char *s = shmat ( id, 0, 0 );
    exit_on_null (s, "Attach");

    int i;
    for ( i = 0; i < 50; i++ ) {
        s[i] = 'x';
    }
    strcpy ( s, "hello, world\n");;
}
```

Execute o programa anterior, "dump", para mostrar o conteúdo da shared memory e interprete o resultado.

## 3.3 Estruturar o conteúdo da memória (Caso 2)

Neste exemplo é criada uma memória partilhada para guardar 25 números inteiros.

```
#include "definicoes.h"
main() {
    int id = shmget ( 1001, 25*sizeof(int), IPC_CREAT | 0666 );
    exit_on_error (id, "Criação/Ligação");

    int *p = shmat ( id, 0, 0 );
    exit_on_null (p, "Attach");

    int i;
    for ( i = 0; i < 10; i++ ) {
        p[i] = i + 1;
    }
}
```

A dimensão da memória pedida é agora 25*sizeof(int), ou seja, 25 vezes o tamanho (o número de bytes) ocupado por um inteiro. Admitindo que cada inteiro ocupa 4 bytes, estamos a falar, na mesma, de 100 bytes.

O resultado do attach é agora convertido para um tipo int *, permitindo assim usar a memória partilhada como um array de inteiros.

O exemplo seguinte mostra o conteúdo da memória partilhada, assumindo agora que a memória contém números inteiros.

```
#include "definicoes.h"
main() {
    int id = shmget ( 1001, 25*sizeof(int), IPC_CREAT | 0666 );
    exit_on_error (id, "Criação/Ligação");

    int *p = shmat ( id, 0, 0 );
    exit_on_null (p, "Attach");

    int i;
    for ( i = 0; i < 25; i++ ) {
        printf ("%5d", p[i] );
        if ( (i+1)%10 == 0 ) printf ("\n");
    }
    printf ("\n");
}
```

## 3.4 Estruturar o conteúdo da memória (Caso 2)

O exemplo seguinte cria uma memória partilhada para guardar dados do tipo definido pela estrutura Aluno.

```
#include "definicoes.h"

typedef struct {
    int num;
    char nome[100];
    int nota;
} Aluno;

int main() {
    int id = shmget ( 1002, 25*sizeof(Aluno), IPC_CREAT | 0666 );
    exit_on_error (id, "Criação/Ligação");

    Aluno *p = shmat ( id, 0, 0 );
    exit_on_null (p, "Attach");

    int i;
    for ( i = 0; i < 25; i++ ) {
        p[i].num = 0;
    }
}
```

O programa cria uma memória partilhada com o tamanho necessário para guardar 25 estruturas Aluno. O resultado do attach é convertido para o ponteiro p do tipo Aluno *. Através deste ponteiro podemos usar a memória como um array de 25 elementos do tipo Aluno.

O exemplo seguinte lê dados e preenche o próximo Aluno livre na memória partilhada com esses dados.

```c
#include "definicoes.h"

void limpar_enter() { while ( getc(stdin) != '\n' ); }

int main() {
    int id = shmget ( 1002, 25*sizeof(Aluno), IPC_CREAT | 0666 );
    exit_on_error (id, "Criação/Ligação");

    Aluno *p = shmat ( id, 0, 0 );
    exit_on_null (p, "Attach");

    Aluno a;

    printf ("Numero: ");
    scanf ("%d", &(a.num) );
    limpar_enter();

    printf ("Nome: ");
    fgets ( a.nome, 100, stdin );
    a.nome[ strlen(a.nome) -1 ] = 0;

    printf ("Nota: ");
    scanf ("%d", &(a.nota) );
    limpar_enter();

    int i;
    for ( i = 0; i < 25; i++ ) {
        if ( p[i].num == 0 ) {
            p[i] = a;
            break;
        }
    }
}
```

O exemplo seguinte lista no ecrã os alunos registados na memória partilhada.

```c
#include "definicoes.h"
int main() {
    int id = shmget ( 1002, 25*sizeof(Aluno), IPC_CREAT | 0666 );
    exit_on_error (id, "Criação/Ligação");

    Aluno *p = shmat ( id, 0, 0 );
    exit_on_null (p, "Attach");

    int i;
    for ( i = 0; i < 25; i++ ) {
        if ( p[i].num == 0 ) continue;
        printf ("%7d %-40s %3d\n", p[i].num, p[i].nome, p[i].nota );
    }
}
```

# 4 Complementos

## 4.1 IPC_NOWAIT

Já antes vimos que um processo bloqueia quando tenta baixar um semáforo que está com o valor zero. Isso também acontece quando se tenta, por exemplo, ler uma mensagem de um determinado tipo e a fila de mensagens não tem mensagens desse tipo.

Por vezes, pode ser útil não bloquear o processo quando tenta fazer uma destas operações. Por exemplo, suponha que um processo quer simplesmente verificar se existem mensagens para si. Se não existirem, pretende-se que o processo continue a sua execução. Pode-se utilizar para isso o valor IPC_NOWAIT.

No caso das filas de mensagem, esse valor é colocado no último argumento da função `msgrcv`. Neste caso o `msgrcv` não bloqueia. Vai buscar uma mensagem e se houver recolhe, senão continua a sua execução.

```
// bloqueia se não houver mensagens
status = msgrcv( msg_id, &msg, sizeof(msg.texto), 1, 0);
// continua a execução, mesmo que não hajam mensagens
status = msgrcv( msg_id, &msg, sizeof(msg.texto), 1, IPC_NOWAIT);
```

No caso dos semáforos

```
// bloqueia, se não conseguir baixar o semáforo
status = semop ( sem_id, &DOWN, 0 );
// Não bloqueia
status = semop ( sem_id, &DOWN, IPC_NOWAIT );
```

## 4.2 ftok()

Nos exemplos anteriores, foi sempre usada uma chave fixa nas funções `msgget`, `semget` e `shmget`. No entanto, a função `ftok()` permite criar uma chave única muito mais interessante, que depende da localização de um dado ficheiro ou directório, bem como de um ID definido pelo utilizador.

O programa seguinte devolve uma chave diferente, cada vez que é executado em directórios diferentes, mas qualquer programa que seja executado no mesmo directório produzirá a mesma chave. Claro que em vez do directório actual "." se poderia usar por exemplo um caminho absoluto, por exemplo: "/home". Nesse caso, a chave seria sempre fixa.

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

main() {
  int chave;
  chave = ftok(".", 'a');
  printf("A chave em uso é: %d\n", chave);
}
```

Alem disso, o ID definido pelo utilizador, neste caso a letra 'a', permite que possamos ter diferentes chaves para diferentes programas que se encontrem no mesmo directório, por exemplo: programas do projecto A e programas do projecto B.

O exemplo seguinte mostra um exemplo de utilização desta função, ilustrando o exemplo clássico do produtor/consumidor, em que vários processos consomem recursos e outros processos produzem esses recursos. Verifique que ambos os processos usam a mesma chave.

prod.c

```
#include <stdio.h>
#include <sys/ipc.h>
#include <sys/sem.h>
#include <stdlib.h>
#define exit_on_error(s,m) if ( s < 0 ) { perror(m); exit(1); }

struct sembuf UP = { 0, 1, 0 } ;
struct sembuf DOWN = { 0, -1, 0 } ;

main() {
    int chave = ftok(".", 'a');
    int sem_id;
    int status, i = 1;
    sem_id = semget ( chave, 1, 0666 | IPC_CREAT );
    exit_on_error (sem_id, "Criação/Ligação");

    while (1) {
      sleep( getpid() % 5 );  // Simula o tempo de produção
      printf ("Acabei de produzir um recurso\n");
      status = semop ( sem_id, &UP, 1 );
      exit_on_error (status, "Problema com o UP");
    }
}
```

cons.c

```
#include <stdio.h>
#include <sys/ipc.h>
#include <sys/sem.h>
#include <stdlib.h>
#define exit_on_error(s,m) if ( s < 0 ) { perror(m); exit(1); }

struct sembuf UP = { 0, 1, 0 } ;
struct sembuf DOWN = { 0, -1, 0 } ;

main() {
    int chave = ftok(".", 'a');
    int sem_id;
    int status, i = 1;
    sem_id = semget ( chave, 1, 0666 | IPC_CREAT );
    exit_on_error (sem_id, "Criação/Ligação");

    while (1) {
      printf ("Estou à espera de um recurso\n");
      status = semop ( sem_id, &DOWN, 1 );
      printf ("Consumi um recurso\n");
      exit_on_error (status, "Problema com o UP");
      sleep( getpid() % 5 );
    }
}
```