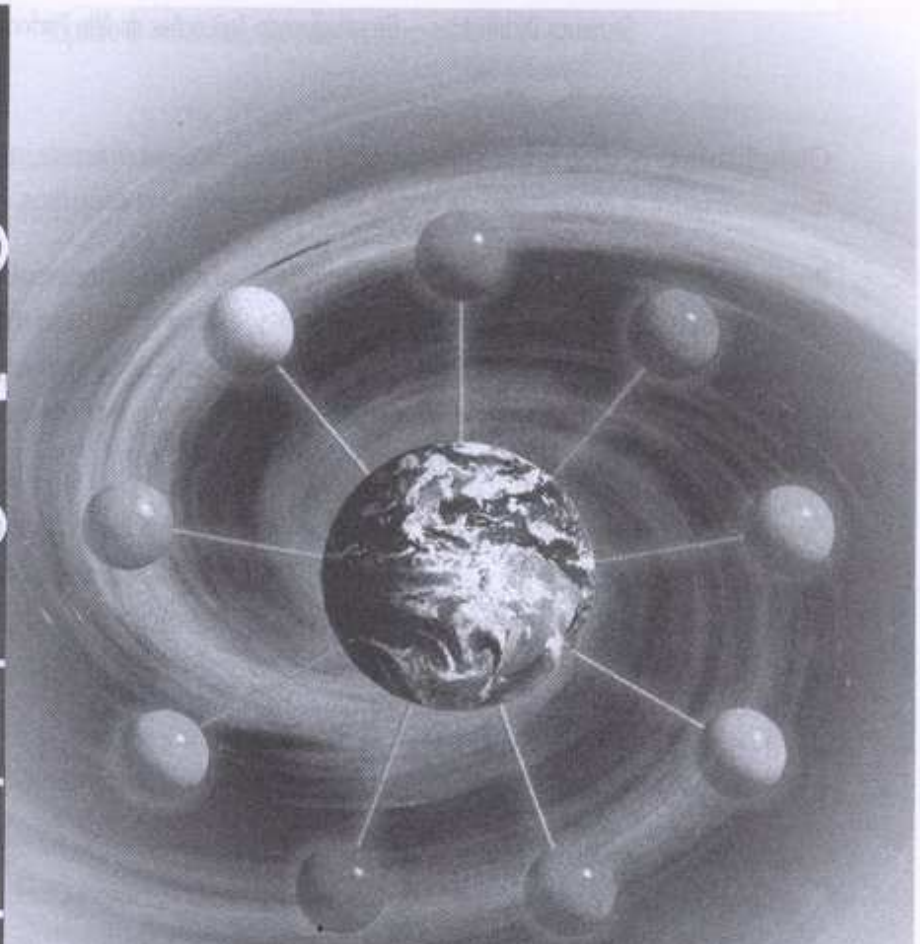
CAPÍTULO



5

Sincronização de Processos

Quando um processo lança mão de processos filhos ou *threads*, um problema computacional novo é criado: a concorrência entre os processos e os recursos compartilhados.

Para resolver este problema, foram criados objetos especiais chamados *semáforos*. Estes objetos fazem a sincronização entre processos para que os recursos compartilhados sejam utilizados de forma ordenada e o resultado de um programa seja sempre o esperado. Aos conflitos de acesso a um mesmo recurso por processos concorrentes dá-se o nome de exclusão mútua.

Observe o seguinte exemplo:

```
#include <pthread.h>
#include <stdio.h>
pthread_t tid1,tid2;
long conta=0;
void * t1()
{
    long i ;
    for (i=0; i<1000000; i++)
    {
    conta = conta + 5 ;
    }
    printf("Encerrei t1\n");
}
void * t2()
{
    long i ;
    for (i=0; i<1000000; i++)
    {
    conta = conta + 2;
    }
    printf("Encerrei a t2\n");
}
int main(){
    pthread_create(&tid1, NULL, t1, NULL) ;
    pthread_create(&tid2, NULL, t2, NULL) ;
    pthread_join(tid1,NULL);
    pthread_join(tid2,NULL);
    printf("O valor da variável conta é: %d\n", conta);
}
```

Para compilar este programa salvo como "*exclusaomutua.c*":

```
# gcc -lpthread exclusaomutua.c -o exclusaomutua
```

No programa acima, as *threads* t1 e t2 compartilham a variável *conta*. Cada processo leve criado faz várias adições a esta variável. O esperado é que este programa sempre produza o valor 7000000, no entanto, executando-se várias vezes, pode-se ver que o resultado é sempre diferente.

Isso acontece na troca de controle de processador entre as *threads*. Imagine que o processo leve t1 esteja incrementado e o valor de *conta* seja 100. O controle do processador é passado para a *thread* t2 que recebe o valor de *conta* igual a 100. Esta incrementa a variável até 122. Ao retornar o controle para a *thread* t1, o valor de *conta* é restaurado para 100, gerando incoerência no resultado.

A causa do problema está no acesso simultâneo à variável por t1 e t2. Para isto ser resolvido, o acesso à *conta* deverá ser feito de forma mutuamente exclusiva.

A parte do código aonde uma ou mais variáveis compartilhadas com outros processos são atualizadas é chamada de "*sessão crítica*". Elas vão ocorrer todas as vezes que:

- Um sistema executa N processos, sendo N > 1
- Cada processo pode ter um código próprio
- Os processos compartilham dados variáveis, de qualquer tipo
- Cada processo possui sessões críticas onde atualizam os dados compartilhados

A solução para o problema é a utilização de um protocolo de controle que regulamente o acesso de entrada e saída à seção crítica.

Desta forma, para acessar uma variável compartilhada, o processo deve executar uma instrução indicando a sua necessidade de entrar na seção crítica. Se nenhum outro processo estiver fazendo acesso à seção, a entrada será liberada. Ao final, ele deve executar uma instrução indicando que a seção foi liberada. Enquanto isso, se outros processos desejarem fazer o acesso, ficam em espera.

"Escolhe o trabalho que gostas e não terás de trabalhar um único dia em tua vida."

Confúcio

Para que o protocolo de entrada e saída de uma instrução crítica funcione, ele deve prover exclusão mútua, onde somente um processo por vez pode estar na seção crítica. Os outros processos que não estão acessando esta área não devem bloquear outros processos e somente aqueles que queiram entrar na sessão crítica devem participar da seleção do próximo a entrar.

O algoritmo a seguir, desenvolvido por Peterson em 1981, implementa uma solução correta para o problema da exclusão mútua entre dois processos. Ele utiliza uma variável *turn* que indica qual processo deverá entrar na seção crítica e um vetor booleano *flag* que diz se o processo está na seção crítica. Assim as condições de progresso e espera limitada da exclusão mútua são satisfeitas.

```
#include <pthread.h>
#include <stdio.h>
pthread_t tid0,tid1;
int turn ;
int shared ;
int flag [2];
void * p0(){
        int i ;
        for(i=0; i<10000; i++)
        {
                flag[0] = 1 ;
                turn = 0 ;
                while(flag [1]==1 && turn==0 ){}
                shared =shared + 5 ;
                flag [0] = 0 ;
        }
}
void * p1(){
        int i ;
        for (i=0;i<10000; i++)
        {
                flag[1] = 1 ;
                turn = 1 ;
                while(flag [0]==1 && turn==1 ){}
                shared =shared + 2 ;
        flag [1] = 0 ;
```

```
        }
}
int main()
{
        flag [0] = 0 ;
        flag [1] = 0 ;
        pthread_create(&tid0, NULL, p0, NULL) ;
        pthread_create(&tid1, NULL, p1, NULL) ;
        pthread_join(tid0,NULL);
        pthread_join(tid1,NULL);
        printf("O valor de shared é: %d\n", shared);
}
```

Para compilar este programa salvo como "*exclusaomutua2.c*":

```
# gcc -lpthread exclusaomutua2.c -o exclusaomutua2
```

O resultado desta vez será sempre o esperado, sem incoerências. No entanto esta implementação possui um grave problema conhecido como "espera ocupada". Os próprios processos precisam testar se a sua entrada à seção crítica será permitida. Se eles não tiverem permissão, toda a fatia de tempo do processador destinada ao processo será desperdiçada. Seria como atender ao telefone e a pessoa que chamou deixá-lo esperando na linha. Em seções críticas com muitas instruções, esta demora pode comprometer seriamente o desempenho do sistema.

IPC – Interprocess Communications

Para resolver o problema da exclusão mútua de uma vez por todas, pode-se utilizar o recurso de semáforos. Esta facilidade é implementada pelo Linux em uma diretiva chamada IPC – Comunicação Inter-Processo – que provê métodos de acesso para as comunicações entre os processos.

Os métodos disponíveis são:

- Condutores de mão única - Half-duplex UNIX pipes
- Condutores de mão dupla - Full-duplex pipes/STREAMS pipes
- Condutores FIFOs - named pipes
- Fila de mensagens

- Semáforos
- Memória compartilhada
- Sockets de rede

Estes métodos são usados para a troca de mensagens entre os processos. As suas características variam de acordo com o objetivo da comunicação, da localização dos processos em um mesmo sistema ou em sistemas independentes, dos meios de transporte utilizados e dos tipos de dados que as mensagens carregam.

O Linux possui dois comandos que interagem com os métodos do IPC: *ipcs* e *ipcrm*. Durante o desenvolvimento das aplicações eles serão úteis para coletar informações e remover recursos desnecessários do sistema.

O Comando *ipcs*

Este comando fornece informações atualizadas de cada um dos recursos IPC implementados no sistema. Seu uso geral é:

```
# ipcs [recurso]
```

Ele aceita as opções a seguir:

- **ipcs -a**: Mostra todos os recursos utilizados
- **ipcs -s**: Mostra informações sobre os semáforos
- **ipcs -m**: Mostra informações relativas aos segmentos de memória compartilhada
- **ipcs -q**: Mostra informações sobre as filas de mensagens

Observe este exemplo:

```
# /ipcs
—— Shared Memory Segments ——
key        shmid      owner      perms      bytes      nattch     status
—— Semaphore Arrays ——
key        semid      owner      perms      nsems
0x00000000 0          nobody     600        1
0x00000000 32769      nobody     600        1
0x00000000 65538      nobody     600        1
0x00000000 98307      nobody     600        1
```

```
0x00000000 131076     nobody     600        1
0x00000000 163845     nobody     600        1
0x00000000 196614     nobody     600        1
0x00000000 229383     nobody     600        1
0x00000000 262152     nobody     600        1
0x00000000 294921     nobody     600        1
0x00000000 327690     nobody     600        1
—— Message Queues ——
key        msqid      owner      perms      used-bytes   messages
```

Pode-se ver que existem onze conjuntos de semáforos criados no sistema, nenhum segmento de memória compartilhada e filas de mensagens. Cada semáforo, segmento de memória compartilhada ou fila de mensagem criado pelas aplicações será registrado e administrado pelo sistema IPC.

O Kernel possui estruturas internas para cada um dos métodos do IPC, possibilitando o controle de acesso aos recursos e outras informações.

O Comando *ipcrm*

Este comando permite que os recursos do IPC possam ser destruídos pelo superusuário ou pelo dono do processo que o criou. É especialmente útil durante o teste de aplicações que tenham acidentalmente criado métodos de forma errada.

Ele exige que seja especificado o tipo de recurso a ser destruído e o identificador associado a esse recurso.

Sua sintaxe é:

```
# ipcrm [sem|shm|msg] <id>
```

No exemplo a seguir o semáforo "*98307*" será destruído:

```
# ipcrm sem 98307
resource deleted
```

Todos os recursos do IPC são associados a uma chave fornecida pela aplicação que retornará um identificador único criado pelo sistema. Isto permite que os recursos possam ser reunidos em um grupo e que as aplicações correlatas possam ter acesso aos recursos compartilhados.

Os Semáforos

Os semáforos podem ser descritos como contadores utilizados para controlar o acesso aos recursos compartilhados. Eles funcionam como uma porta que previne um processo de acessar um recurso particular enquanto outro processo o utiliza. O nome "semáforo" vem das antigas cancelas de cruzamento das rodovias com os trilhos de uma ferrovia evitando a passagem de carros durante a passagem de uma locomotiva.

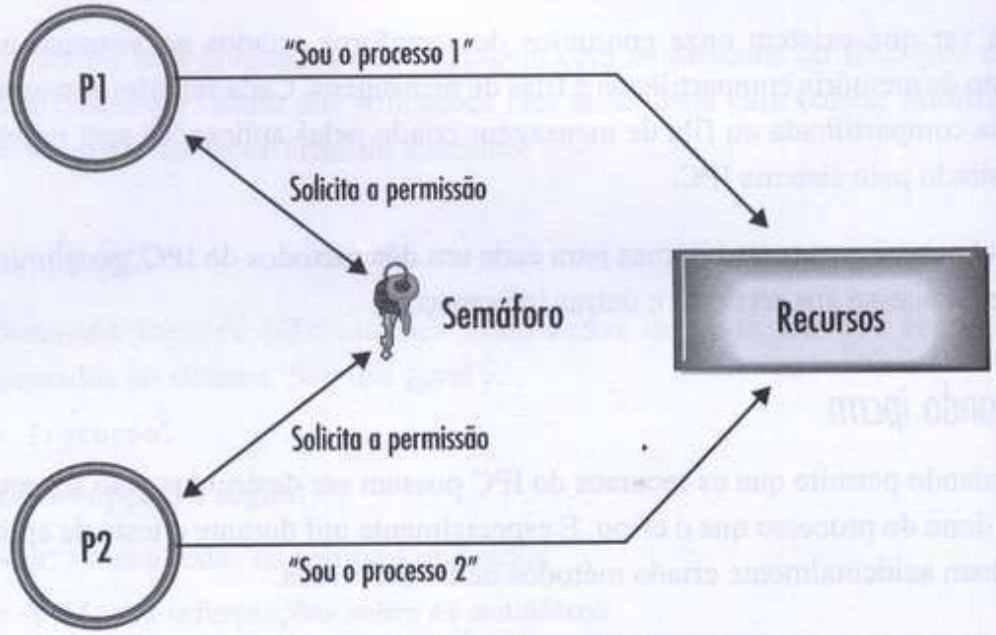


Figura 5.1: Esquema de funcionamento dos semáforos.

A Primitiva *semget( )*

A função *semget( )* é utilizada para criar um novo conjunto de semáforos ou para obter a identificação de um conjunto de semáforos já existente. Ela pode ser usada para criar um semáforo ou vários semáforos dentro de um mesmo conjunto. Isso permite que várias partes do código possam ser protegidas com semáforos diferentes.

```
# include <sys/types.h>
# include <sys/ipc.h>
# include <sys/sem.h>
int semget (key_t key, int nsems, int semflg)
```

O argumento *key* deverá indicar a chave identificadora de um conjunto de semáforos e *nsems* deve indicar o número de semáforos a ser criado no conjunto. O argumento *semflg* trabalha com as permissões de criação.

O *semflg* pode assumir os valores *IPC_CREAT* e *IPC_EXCL*. O primeiro valor determina a criação de um semáforo se este não existir. O segundo determina que a função *semget* poderá falhar se o conjunto de semáforos já existir. Além disso, este argumento pode especificar os direitos de acesso.

As constantes que podem ser utilizadas no argumento *semflg* são predefinidas nos arquivos *sys/sem.h* e *sys/ipc.h*, a seguir:

- **IPC_CREAT**: Cria uma chave se uma já não existir
- **IPC_EXCL**: Falha se a chave já existir
- **IPC_RMID**: Remove o identificador

A função *semget( )* irá retornar o identificador do conjunto de semáforos criado pelo sistema quando for bem-sucedida ou -1 em caso de erro.

Veja o seguinte exemplo:

```
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/sem.h>
#include <stdio.h>
#define KEY 123
int main()
{
        int semid;
        if (( semid = semget(KEY, 4, IPC_CREAT|IPC_EXCL|0600)) == -1)
        {
                printf("Erro ao criar o semáforo\n");
                exit(1);
        }
        printf("Os semáforos foram criados com o identificador:
%d\n",semid);
        printf("Este conjunto tem a chave única: %d\n",KEY);
        exit(0);
}
```

Para compilar o programa salvo como "*semget.c*":

```
# gcc semget.c -o semget
```

O Resultado da execução será:

```
# ./semget
```

Os semáforos foram criados com o identificador: 360459

Este conjunto tem a chave única: 123

Os semáforos criados podem ser visualizados no sistema com o comando *ipcs*:

```
# ipcs -s
—— Semaphore Arrays ——
key        semid      owner      perms      nsems
0x0000007b 360459     root       600        4
```

Neste exemplo, serão criados 4 semáforos utilizando a chave única 123 representada em hexadecimal como 0x0000007b. O identificador único criado pelo sistema para este conjunto será 360459 cujo dono será o superusuário.

Se este programa for executado novamente, ele irá falhar na criação do semáforo, pois o argumento *semflg* com o valor "IPC_EXCL" não permitirá que um conjunto com a chave 123 já exista.

```
# ./semget
Erro ao criar o semáforo
```

Todos os processos que compartilham recursos com este programa deverão conhecer a chave 123 e ter permissões de leitura e/ou escrita para ter acesso a este conjunto de semáforos.

A Função *semctl( )*

A função *semctl( )* é utilizada para controlar as operações em um conjunto de semáforos. Estas operações permitem solicitar ou alterar as informações dos semáforos e registrar os acessos às seções críticas.

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

O primeiro argumento recebido deverá ser o identificador único criado pela função *semget()*. O argumento *semun* deverá indicar o número do semáforo que irá sofrer controle. Este número deve ser um índice que começa sempre com zero.

O argumento *cmd* representa o comando que será executado no semáforo selecionado. Os comandos podem ser:

- **IPC_STAT**: Faz a cópia da estrutura de dados do semáforo para uma estrutura definida em *arg*.
- **IPC_SET**: Escreve os valores da estrutura definida em *arg* para o semáforo.
- **IPC_RMID**: Remove o semáforo do sistema.
- **GETALL**: Obtém todos os valores de todos os semáforos.
- **GETNCNT**: Retorna o número de processos à espera de recursos.
- **GETPID**: Retorna o PID do processo que fez a última operação.
- **GETVAL**: Retorna o valor de somente um semáforo.
- **SETALL**: Gravam os valores da estrutura em todos os semáforos.
- **SETVAL**: Grava somente um valor a um semáforo.

O argumento *arg* representa uma estrutura chamada *semnum*. Os membros desta estrutura determinam os tipos de dados que um semáforo poderá guardar. O conteúdo desta estrutura poderá ser carregado pelo sistema ou o seu conteúdo poderá ser gravado no sistema alterando as informações do semáforo. Sua estrutura é definida a seguir:

```
union semun
{
    int val;                /* valor configurado por SETVAL */
    struct semid_ds *buf;   /* buffer para IPC_STAT e IPC_SET */
    ushort *array;          /* array para GETALL e SETALL */
    struct seminfo *__buf;  /* buffer para IPC_INFO */
    void *__pad;
};
```

"A experiência é uma professora muito severa porque primeiro ela aplica a prova e, só depois, ensina a lição."

Vernon Law

A Função *semop( )*

A função *semop( )* é utilizada para realizar operações simples em um conjunto de semáforos.

```
#include <sys/sem.h>
int semop(int semid, struct sembuf *sops, size_t nsops);
```

O conjunto de semáforos que irá sofrer as operações deve ser identificado pelo parâmetro *semid*, que foi retornado pela função *semget( )*.

O parâmetro *sops* é um ponteiro para a estrutura dos semáforos que conterá os valores a serem alterados. O último argumento *nsops* recebe o número de semáforos na estrutura que sofrerão a operação.

A estrutura de um semáforo inclui os seguintes membros:

Tabela 5.1: Estrutura de um semáforo.

Objeto	Tipo	Descrição
sem_num	short	Especifica o número do semáforo.
sem_op	short	Especifica a operação que será realizada no semáforo.
sem_flg	short	Especifica o flag da operação.

As operações que poderão ser realizadas em um semáforo dependem do novo valor da variável *sem_op* e do valor que já está gravado no semáforo.

O valor do semáforo poderá ser lido com a função:

```
semval = semctl(semid,sem_num,GETVAL,arg)
```

O resultado da operação é determinado pela interação entre o valor corrente do semáforo *semval* e o valor de *sem_op*. O detalhamento de todas as combinações possíveis é complexo, e, na maioria dos casos, os seguintes valores de *sem_op* permitem estas operações:

- Se o valor de *sem_op* for menor que zero, significa um pedido do uso de um determinado recurso.

- Se o valor de *sem_op* for igual a zero, o processo deverá ler as permissões de acesso do conjunto de semáforos.
- Se o valor de *sem_op* for maior que zero, significa que o recurso deverá ser liberado.

Observe o seguinte exemplo:

```
#include <stdio.h>
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/sem.h>
#include <unistd.h>
#define KEY 1234
union semun {
            int val ;
            struct semid_ds buf[2] ;
            unsigned short int array[4] ;
            struct seminfo *__buf;
};

int main()
{
       struct sembuf sempar;
       int semid, semval , sempid;
       union semun arg;
       if (( semid = semget(KEY, 4, IPC_CREAT|IPC_EXCL|0600)) == -1 )
       {
              printf("Erro ao criar semáforo.\n");
              exit(1);
       }
       printf("O conjunto de semáforos foi criado com a
identificação: %d.\n",semid);
       if ((semval = semctl(semid,2,GETVAL,arg)) == -1)
       {
              printf("Erro ao ler o terceiro semáforo.\n");
              exit(1);
       }
       else
       {
       printf("O valor do terceiro semáforo é: %d\n",semval);
```

```
        }
        sleep(3);
        sempar.sem_num = 2 ;
        sempar.sem_op = 1 ;
        sempar.sem_flg = SEM_UNDO ;
        if (semop(semid, &sempar, 1) == -1)
        {
                printf("Não foi possível atualizar o semáforo.\n");
                exit(-1);
        }
        if ((semval = semctl(semid,2,GETVAL,arg)) == -1)
        {
               printf("Não foi possível pegar o valor do semáforo.\n");
                exit(1);
        }
        else
        {
              printf("O valor do terceiro semáforo é agora: %d\n",semval);
        }
        sleep(3);
        if (semctl(semid,0,IPC_RMID,0)==-1)
        {
                printf("Impossível destruir o semáforo.\n");
                exit(1);
        }
        else
        {
                printf("O semáforo com ID %d foi destruído.\n",semid);
        }
        exit(0);
}
```

Para compilar o programa salvo como "*semctl.c*":

```
# gcc semctl.c -o semctl
```

O resultado da execução será:

```
# ./semctl
O conjunto de semáforos foi criado com a identificação: 360459.
O valor do terceiro semáforo é: 0
O valor do terceiro semáforo é agora: 1
O semáforo com ID 360459 foi destruído.
```

Detalhando o código deste programa, a estrutura padrão do semáforo será criada como *semun* e unida à variável *arg*.

```
union semun {
            int val ;
            struct semid_ds buf[2] ;
            unsigned short int array[4] ;
            struct seminfo *__buf;
};

        struct sembuf sempar;
        int semid, semval , sempid;
        union semun arg;
```

Um conjunto de quatro semáforos será criado com a chave 123 no sistemas que ganharão a identificação 360459 através da função *semget(KEY, 4, IPC_CREAT|IPC_EXCL|0600))*.

Os argumentos "*IPC_CREAT|IPC_EXCL|0600*" definirão que os semáforos serão criados somente se já não existirem e com as permissões de leitura e gravação para o usuário dono do processo.

O conteúdo do terceiro semáforo será recuperado com a função *semctl(semid,2,GETVAL,arg)* e gravado na variável *semval*. Esta função receberá a identificação do conjunto de semáforos *semid*. O comando *GETVAL* retornará o valor somente de um semáforo. O valor do semáforo então será impresso na tela.

Logo em seguida o valor do semáforo será alterado pela função *semop( )*. A seguinte estrutura receberá o conteúdo da operação:

```
sempar.sem_num = 2 ;
sempar.sem_op = 1 ;
sempar.sem_flg = SEM_UNDO ;
```

A estrutura *sempar* receberá três valores: o número do semáforo no conjunto, a operação 1 e o flag *SEM_UNDO*. Este flag informa que todas as operações de um semáforo deverão ser desfeitas caso o processo morra.

A função *semop(semid, &sempar, 1)* receberá o identificador do conjunto de semáforos, a estrutura com os valores da operação e o número de semáforos declarados na estrutura.

Novamente o valor do terceiro semáforo será lido pela função *semval = semctl(semid,2,GETVAL,arg)*. O novo valor do semáforo será impresso na tela.

Para terminar, o semáforo será destruído pela função *semctl(semid,0,IPC_RMID,0)*. O comando *IPC_RMID* remove todo o conjunto de semáforos identificados pela variável *semid*.

Este tipo de implementação pode ser um pouco complexa para programadores que estão iniciando no paralelismo de processos.

Uma solução mais simples foi desenvolvida pelo professor Edsger Wybe Dijkstra da Universidade do Texas nos Estados Unidos em 1979, chamada semáforo binário.

Edsger foi um dos mais influentes membros da indústria da computação e, durante seus mais de quarenta anos de trabalho, ganhou muitos prêmios, incluindo o *ACM Turing*, a maior honra que um cientista da computação pode receber.

O Semáforo Binário de Dijkstra

O semáforo binário de Dijkstra (lê-se "déquistra") simplifica a operação de semáforos permitindo resolver o problema de exclusão mútua.

Dijkstra desenvolveu duas operações básicas que podem ser feitas no seu semáforo: P (aquisição) e V (liberação). Quando a operação *P* é realizada sobre um semáforo, ele permite que um processo possa acessar um recurso compartilhado se não houver outro processo acessando. A operação *V* libera o recurso para ser utilizado por outro processo que esteja na fila.

Características do semáforo binário:

- Seu semáforo é uma estrutura de dados formada por um contador e um apontador para uma fila de processos bloqueados no semáforo;
- Somente pode ser acessado por duas operações atômicas (P e V);
- A operação P bloqueia o processo, se o valor do semáforo for 0;
- A operação V incrementa o valor do semáforo. Existindo processos bloqueados, o primeiro da fila obtém o controle;

- Se dois processos tentam simultaneamente executar P ou V, a execução será em ordem arbitrária.

A implementação do semáforo de binário de Dijkstra é apresentada a seguir:

```
/* SEMÁFOROS DE DIJKSTRA - BIBLIOTECA */
int sem_create(key_t key, int initval)
{
    int semid;

    union semun {
    int val;
    struct semid_ds *buf ;
    ushort array[1];
} arg_ctl;

semid = semget(ftok("dijkstra.h",key),1,IPC_CREAT|IPC_EXCL|0666);
if (semid == -1) {
    semid = semget(ftok("dijkstra.h",key),1,0666);
    if (semid == -1) {
       printf("Erro semget()\n");
    exit(1);
    }
}

 arg_ctl.val = initval;
if (semctl(semid,0,SETVAL,arg_ctl) == -1) {
    printf("Erro semctl()\n");
    exit(1);
}
return(semid) ;
}

void P(int semid)
{
    struct sembuf sempar[1];
    sempar[0].sem_num = 0;
    sempar[0].sem_op = -1;
    sempar[0].sem_flg = SEM_UNDO;
    if (semop(semid, sempar, 1) == -1) printf("Erro P()\n");
```

```
}

void V(int semid)
{
    struct sembuf sempar[1];
    sempar[0].sem_num = 0;
    sempar[0].sem_op =  1;
    sempar[0].sem_flg = SEM_UNDO ;
    if (semop(semid, sempar, 1) == -1) printf("Erro V()\n");
}

void sem_delete(int semid)
{
    if (semctl(semid,0,IPC_RMID,0) == -1) printf("Erro
IPC_RMID\n");
}
```

Este código é uma biblioteca que oferece basicamente quatro funções aos programas:

- *sem_create(key_t key, int initval)*: Cria um conjunto de semáforos com a chave *key* e com o valor inicial do semáforo *initval*.
- *p(int semid)*: A operação P bloqueia o recurso se o valor do semáforo for 0 e -1 bloqueia o processo.
- *v(int semid)*: A operação V libera o recurso.
- *sem_delete(int semid)*: Remove o semáforo criado.

Para utilizar os semáforos de Dijkstra, salve a biblioteca acima como "*dijkstra.h*".

Sua utilização será demonstrada no seguinte exemplo:

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

#define KEY 123

int main()
```

```
{
  int sem;
  sem = sem_create(KEY,1) ;
  printf("Um semáforo foi criado com o identificador %d.\n",sem);
  if (fork() == 0)
  {
    printf("\tProcesso filho usa o recurso.\n");
    P(sem);
    sleep(8);
    printf("\tProcesso filho libera o recurso.\n");
    V(sem);
    sleep(1);
  }
  else
  {
    sleep(1);
    printf("Processo PAI bloqueia ao tentar acessar recurso.\n");
    P(sem);
    printf("Recurso disponível para o processo PAI.\n");
    sem_delete(sem);
  }
  exit(0);
}
```

Para compilar este programa salvo como "*semaforo_binario.c*":

```
gcc semaforo_binario.c -o semaforo_binario
```

Ao ser executado, este programa produz o seguinte resultado:

```
# ./semaforo_binario
Um semáforo foi criado com o identificador 98304.
        Processo filho usa o recurso.
Processo PAI bloqueia ao tentar acessar recurso.
        Processo filho libera o recurso.
Recurso disponível para o processo PAI.
```

Neste exemplo o semáforo vai controlar um recurso compartilhado qualquer disputado por dois processos concorrentes criados com *fork()*. O processo filho bloqueia os recursos com a instrução *P()*. O processo pai ao tentar utilizar o recurso também com a instrução *P()* é bloqueado até que o processo filho libere os recursos com a função *V()*.

O primeiro exemplo do capítulo com problema de *race-condition* poderá ser resolvido com pequenas modificações incluindo um semáforo binário. Observe novamente este exemplo:

```
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/sem.h>
#include <unistd.h>
#include "dijkstra.h"
#include <pthread.h>
#include <stdio.h>

#define KEY 1234

pthread_t tid1,tid2;
long conta=0;
void * t1(int sem)
{
    P(sem);
    long i ;
    for (i=0; i<1000000; i++)
    {
       conta = conta + 5 ;
    }
    printf("Encerrei t1\n");
    V(sem);
}
void * t2(int sem)
{
    P(sem);
    long i ;
    for (i=0; i<1000000; i++)
    {
       conta = conta + 2;
    }
    printf("Encerrei a t2\n");
    V(sem);
}
int main()
```

```
{
    int sem;
    sem = sem_create(KEY,1) ;
    printf("Um semáforo foi criado com o identificador %d.\n",sem);
    pthread_create(&tid1, NULL, t1, (void *)sem) ;
    pthread_create(&tid2, NULL, t2, (void *)sem) ;
    pthread_join(tid1,NULL);
    pthread_join(tid2,NULL);
    sem_delete(sem);
    printf("O valor da variável conta é: %d\n", conta);
}
```

Para compilar este exemplo salvo como "*exclusaomutua_ok.c*":

```
# gcc -lpthread exclusaomutua_ok.c -o exclusaomutua_ok
```

Na nova versão deste programa, um semáforo binário será criado com a função *sem_create( )* e seu identificador será passado como argumento para as threads criadas com a instrução *pthread_create( )*. Cada processo leve irá solicitar o recurso compartilhado antes de começar suas operações na sessão crítica com a instrução *P( )* e ao término irá liberá-los com *V( )*.

Quando algum *thread* ganhar a concessão do recurso com a função *P( )*, o outro processo que tentar acessar o recurso fica suspenso até que o primeiro execute *V( )*. O semáforo irá garantir que não haverá conflito pelo uso da variável *conta* e o resultado da operação vai ser sempre exato.

Os semáforos binários são simples e fáceis implantar em programas concorrentes e resolvem com elegância a maioria dos problemas de exclusão mútua.

"A alegria está na luta, na tentativa, no sofrimento envolvido, não na vitória propriamente dita."

Gandhi