Interprocess Communication

Concurrent Programming, Semaphores & Shared Memory and Deadlock

PEDRO MARTINS

January 13, 2018

Contents

1 Conceitos Introdutórios

Num ambiente multiprogramado, os processos podem ser:

- Independentes:
  - Nunca interagem desde a sua criação à sua destruição
  - Só possuem uma interação implícita: **competir por recursos do sistema**
    - e.g.: jobs num sistema bacth, processos de diferentes utilizadores
  - É da responsabilidade do sistema operativo garantir que a atribuição de recursos é feita de forma controlada
    - É preciso garantir que não ocorre perda de informação
    - Só **um processo pode usar um recurso num intervalo de tempo** - ***Mutual Exclusive Access***
- Cooperativos:
  - **Partilham Informação** e/ou **Comunicam** entre si
  - Para **partilharem** informação precisam de ter acesso a um **espaço de endereçamento comum**
  - A comunicação entre processos pode ser feita através de:
    - Endereço de memória comum
    - Canal de comunicação que liga os processos
  - É da **responsabilidade do processo** garantir que o acesso à zona de memória partilhada ou ao canal de comunicação é feito de forma controlada para não ocorrerem perdas de informação
    - Só **um processo pode usar um recurso num intervalo de tempo** - ***Mutual Exclusive Access***
    - Tipicamente, o canal de comunicação é um recurso do sistema, pelo quais os **processos competem**

O acesso a um recurso/área partilhada é efetuada através de código. Para evitar a perda de informação, o código de acesso (também denominado zona crítica) deve evitar incorrer em **race conditions**.

1.1 Exclusão Mútua

Ao forçar a ocorrência de exclusão mútua no acesso a um recurso/área partilhada, podemos originar:

- **deadlock:**
  - Vários processos estão em espera **eternamente** pelas condições/eventos que lhe permitem aceder à sua respetiva **zona crítica**
    - Pode ser provado que estas condições/eventos **nunca se irão verificar**
  - Causa o bloqueio da execução das operações
- **starvation:**
  - Na competição por acesso a uma zona crítica por vários processos, verificam-se um conjunto de circunstâncias na qual novos processos, com maior prioridade no acesso às suas zonas críticas, continuam a aparecer e **tomar posse dos recursos partilhados**
  - O acesso dos processos mais antigos à sua zona crítica é sucessivamente adiado

2 Acesso a um Recurso

No acesso a um recurso é preciso garantir que não ocorrem **race conditions**. Para isso, **antes** do acesso ao recurso propriamente dito é preciso **desativar o acesso** a esse recurso pelos **outros processos** (reclamar *ownership*) e após o acesso é preciso restaurar as condições iniciais, ou seja, **libertar o acesso** ao recurso.

```
/* processes competing for a resource - p = 0, 1, ..., N-1 */
void main (unsigned int p)
{
    forever
    {
        do_something();
        access_resource(p);
        do_something_else();
    }
}

void acess_resource(unsigned int p)
{
    enter_critical_section(p);
    use_resource();     //  critical section
    leave_critical_section(p);
}
```

3 Acesso a Memória Partilhada

O acesso à memória partilhada é muito semelhante ao aceso a um recurso (podemos ver a memória partilhada como um recurso partilhado entre vários processos).

Assim, à semelhança do acesso a um recurso, é preciso **bloquear o acesso de outros processos à memória partilhada** antes de aceder ao recurso e após aceder, **reativar o acesso a memória partilhada** pelos outros processos.

```
/* shared data structure */
shared DATA d;

/* processes sharing data - p = 0, 1, ..., N-1 */
void main (unsigned int p)
{
    forever
    {
        do_something();
        access_shared_area(p);
        do_something_else();
    }
```

```
}

void access_shared_area(unsigned int p)
{
    enter_critical_section(p);
    manipulate_shared_area(); // critical section
    leave_critical_section(p);
}
```

3.1 Relação Produtor-Consumidor

O acesso a um recurso/memória partilhada pode ser visto como um problema Produtor-Consumidor:

- Um processo acede para **armazenar dados**, **escrevendo** na memória partilhada *(Produtor)*
- Outro processo acede para **obter dados**, **lendo** da memória partilhada *(Consumidor)*

3.1.1 Produtor

O produtor "produz informação" que quer guardar na FIFO e enquanto não puder efetuar a sua escrita, aguarda até puder **bloquear** e **tomar posse** do zona de memória partilhada

```
/* communicating data structure: FIFO of fixed size */
shared FIFO fifo;

/* producer processes - p = 0, 1, ..., N-1 */
void main (unsigned int p)
{
    DATA val;
    bool done;


    forever
    {
        produce_data(&val);
        done = false;
        do
        {
            // Beginning of Critical Section
            enter_critical_section(p);
            if (fifo.notFull())
            {
                fifo.insert(val);
                done = true;
            }
            leave_critical_section(p);
```

```
25              // End of Critical Section
26          } while (!done);
27          do_something_else();
28      }
29  }
```

3.1.2 Consumidor

O consumidor quer ler informação que precisa de obter da FIFO e enquanto não puder efetuar a sua leitura, aguarda até puder **bloquear** e **tomar posse** do zona de memória partilhada

```
1   /* communicating data structure: FIFO of fixed size */
2   shared FIFO fifo;
3
4   /* consumer processes - p = 0, 1, ..., M-1 */
5   void main (unsigned int p)
6   {
7   DATA val;
8   bool done;
9       forever
10      {
11          done = false;
12          do
13          {
14              // Beginning of Critical Section
15              enter_critical_section(p);
16              if (fifo.notEmpty())
17              {
18                  fifo.retrieve(&val);
19                  done = true;
20              }
21              leave_critical_section(p);
22              // End of Critical Section
23          } while (!done);
24          consume_data(val);
25          do_something_else();
26      }
27  }
```

4 Acesso a uma Zona Crítica

Ao aceder a uma zona crítica devem ser verificados as seguintes condições:

- **Efective Mutual Exclusion:** O **acesso** a uma **zona crítica** associada com o mesmo recurso/memória partilhada só pode ser **permitida a um processo de cada vez** entre **todos os processos** a competir pelo acesso a esse mesmo recurso/memória partilhada
- **Independência** do número de processos intervenientes e na sua velocidade relativa de execução
- Um processo fora da sua zona crítica não pode impedir outro processo de entrar na sua zona crítica
- Um processo **não deve ter de esperar indefinidamente** após pedir acesso ao recurso/memória partilhada para que possa aceder à sua zona crítica
- O período de tempo que um processo está na sua **zona crítica** deve ser **finito**

4.1 Tipos de Soluções

Para controlar o acesso às zonas críticas normalmente é usado um endereço de memória. A gestão pode ser efetuada por:

- **Software:**
  - A solução é baseada nas instruções típicas de acesso à memória
  - Leitura e Escrita são independentes e correspondem a instruções diferentes
- **Hardware:**
  - A solução é baseada num conjunto de instruções especiais de acesso à memória
  - Estas instruções permitem ler e de seguida escrever na memória, de forma **atómica**

4.2 Alternância Estrita *(Strict Alternation)*

Não é uma solução válida

- Depende da velocidade relativa de execução dos processos intervenientes
- O processo com menos acessos impõe o ritmo de acessos aos restantes processos
- Um processo fora da zona crítica não pode prevenir outro processo de entrar na sua zona crítica
- Se não for o seu turno, um processo é obrigado a esperar, mesmo que não exista mais nenhum processo a pedir acesso ao recurso/memória partilhada

```
1  /* control data structure */
2  #define R       /* process id = 0, 1, ..., R-1 */
3
4  shared unsigned int access_turn = 0;
5  void enter_critical_section(unsigned int own_pid)
6  {
7      while (own_pid != access_turn);
8  }
9
10 void leave_critical_section(unsigned int own_pid)
11 {
12     if (own_pid == access_turn)
```

```
13          access_turn = (access_turn + 1) % R;
14  }
```

4.3 Eliminar a Alternância Estrita

```
1   /* control data structure */
2   #define R 2     /* process id = 0, 1 */
3
4   shared bool is_in[R] = {false, false};
5
6   void enter_critical_section(unsigned int own_pid)
7   {
8       unsigned int other_pid_ = 1 - own_pid;
9
10      while (is_in[other_pid]);
11      is_in[own_pid] = true;
12  }
13
14  void leave_critical_section(unsigned int own_pid)
15  {
16      is_in[own_pid] = false;
17  }
```

Esta solução não é válida porque não garante **exclusão mútua**.

Assume que:

- P_0 entra na função `enters_critical_section` e testa `is_in[1]`, que retorna Falso
- P_1 entra na função `enter_critical_section` e testa `is_in[0]`, que retorna Falso
- P_1 altera `is_in[0]` para *true* e entra na zona crítica
- P_0 altera `is_in[1]` para *true* e entra na zona crítica

Assim, ambos os processos entra na sua zona crítica **no mesmo intervalo de tempo**.

O principal problema desta implementação advém de **testar primeiro** a variável de controlo do **outro processo** e só **depois** alterar a **sua variável** de controlo.

4.4 Garantir a exclusão mútua

```
1   /* control data structure */
2   #define R 2     /* process id = 0, 1 */
3
4   shared bool want_enter[R] = {false, false};
5
6   void enter_critical_section(unsigned int own_pid)
```

```
{
    unsigned int other_pid_ = 1 - own_pid;

    want_enter[own_pid] = true;
    while (want_enter[other_pid]);
}

void leave_critical_section(unsigned int own_pid)
{
    want_enter[own_pid] = false;
}
```

Esta solução, apesar de **resolver a exclusão mútua**, **não é válida** porque podem ocorrer situações de **deadlock**.

Assume que:

- P_0 entra na função `enter_critical_section` e efetua o *set* de `want_enter[0]`
- P_1 entra na função `enter_critical_section` e efetua o *set* de `want_enter[1]`
- P_1 testa `want_enter[0]` e, como é *true*, **fica em espera** para entrar na zona crítica
- P_0 testa `want_enter[1]` e, como é *true*, **fica em espera** para entrar na zona crítica

Com **ambos os processos em espera** para entrar na zona crítica e **nenhum processo na zona crítica** entramos numa situação de **deadlock**.

Para resolver a situação de deadlock, **pelo menos um dos processos** tem recuar na intenção de aceder à zona crítica.

4.5 Garantir que não ocorre deadlock

```
/* control data structure */
#define R 2     /* process id = 0, 1 */

shared bool want_enter[R] = {false, false};

void enter_critical_section(unsigned int own_pid)
{
    unsigned int other_pid_ = 1 - own_pid;

    want_enter[own_pid] = true;
    while (want_enter[other_pid])
    {
        want_enter[own_pid] = false;    // go back
        random_dealy();
        want_enter[own_pid] = true;     // attempt a to go to the critical
            section
    }
```

```
17 }
18
19 void leave_critical_section(unsigned int own_pid)
20 {
21     want_enter[own_pid] = false;
22 }
```

A solução é quase válida. Mesmo um dos processos a recuar ainda é possível ocorrerem situações de **deadlock** e **starvation**:

- Se ambos os processos **recuarem ao "mesmo tempo"** (devido ao `random_delay()` ser igual), entramos numa situação de **starvation**
- Se ambos os processos **avançarem ao "mesmo tempo"** (devido ao `random_delay()` ser igual), entramos numa situação de **deadlock**

A solução para **mediar os acessos** tem de ser **determinística** e não aleatória.

4.6 Mediar os acessos de forma determinística: *Dekker agorithm*

```
1  /* control data structure */
2  #define R 2     /* process id = 0, 1 */
3
4  shared bool want_enter[R] = {false, false};
5  shared uint p_w_priority = 0;
6
7  void enter_critical_section(unsigned int own_pid)
8  {
9      unsigned int other_pid_ = 1 - own_pid;
10
11     want_enter[own_pid] = true;
12     while (want_enter[other_pid])
13     {
14         if (own_pid != p_w_priority)            // If the process is not the
               priority process
15         {
16             want_enter[own_pid] = false;        // go back
17             while (own_pid != p_w_priority);    // waits to access to his
                   critical section while
18                                                 //  its is not the priority
                                                       process
19             want_enter[own_pid] = true;         // attempt to go to his
                   critical section
20         }
21     }
22 }
23
```

```
24 void leave_critical_section(unsigned int own_pid)
25 {
26     unsigned int other_pid_ = 1 - own_pid;
27     p_w_priority = other_pid;                    // when leaving the its
           critical section, assign the
28                                                  // priority to the other
                                                        process
29     want_enter[own_pid] = false;
30 }
```

É uma **solução válida**:

- Garante exclusão mútua no acesso à zona crítica através de um mecanismo de alternância para resolver o conflito de acessos
- **deadlock** e **starvation não estão presentes**
- Não são feitas suposições relativas ao tempo de execução dos processos, i.e., o algoritmo é **independente** do tempo de execução dos processos

No entanto, **não pode ser generalizado** para mais do que 2 processos e garantir que continuam a ser satisfeitas as condições de **exclusão mútua** e a ausência de **deadlock** e **starvation**

4.7 Dijkstra algorithm (1966)

```
1  /* control data structure */
2  #define R 2     /* process id = 0, 1 */
3
4  shared uint want_enter[R] = {NO, NO, ..., NO};
5  shared uint p_w_priority = 0;
6
7  void enter_critical_section(uint own_pid)
8  {
9      uint n;
10     do
11     {
12         want_enter[own_pid] = WANT;              // attempt to access to the
               critical section
13         while (own_pid != p_w_priority)          // While the process is not
               the priority process
14         {
15             if (want_enter[p_w_priority] == NO)  // Wait for the priority
                   process to leave its critical section
16                 p_w_priority = own_pid;
17         }
18
19         want_enter[own_pid] = DECIDED;           // Mark as the next process
               to access to its critical section
```

```
20
21          for (n = 0; n < R; n++)             // Search if another process is
                already entering its critical section
22          {
23              if (n != own_pid && want_enter[n] == DECIDED)   // If so, abort
                    attempt to ensure mutual exclusion
24                  break;
25          }
26      } while(n < R);
27  }
28
29  void leave_critical_section(unsigned int own_pid)
30  {
31      p_w_priority = (own_pid + 1) % R;           // when leaving the its
            critical section, assign the
32                                                  // priority to the next process
33      want_enter[own_pid] = false;
34  }
```

Pode sofrer de **starvation** se quando um processo iniciar a saída da zona crítica e alterar `p_w_priority`, atribuindo a prioridade a outro processo, outro processo tentar aceder à zona crítica, sendo a sua execução interrompida no for. Em situações "especiais", este fenómeno pode ocorrer sempre para o mesmo processo, o que faz com que ele nunca entre na sua zona crítica

4.8 Peterson Algorithm (1981)

```
1   /* control data structure */
2   #define R 2     /* process id = 0, 1 */
3
4   shared bool want_enter[R] = {false, false};
5   shared uint last;
6
7   void enter_critical_section(uint own_pid)
8   {
9       unsigned int other_pid_ = 1 - own_pid;
10
11      want_enter[own_pid] = true;
12      last = own_pid;
13      while ( (want_enter[other_pid]) && (last == own_pid) );     // Only enters
            the critical section when no other
14                                                                  // process
                                                                        wants to
                                                                        enter and
                                                                        the last
                                                                        request
```

```
15                                                                  // to enter is
                                                                        made by the
                                                                        current
                                                                        process
16  }
17
18  void leave_critical_section(unsigned int own_pid)
19  {
20      want_enter[own_pid] = false;
21  }
```

O algoritmo de *Peterson* usa a **ordem de chegada** de pedidos para resolver conflitos:

- Cada processo tem de **escrever o seu ID numa variável partilhada** (*last*), que indica qual foi o último processo a pedir para entrar na zona crítica
- A **leitura seguinte** é que vai determinar qual é o processo que foi o último a escrever e portanto qual o processo que deve entrar na zona crítica

	P_0 quer entrar		P_1 quer entrar	
	P_1 não quer entrar	P_1 quer entrar	P_0 não quer entrar	P_0 quer entrar
last = P_0	P_0 entra	P_1 entra	-	P_1 entra
last = P_1	-	P_0 entra	P_1 entra	P_0 entra

É uma solução válida que:

- Garante exclusão mútua
- Previne deadlock e starvation
- É independente da velocidade relativa dos processos
- Pode ser generalizada para mais do que dois processos (variável partilhada -> fila de espera)

4.9 Generalized Peterson Algorithm (1981)

```
1   /* control data structure */
2   #define R ...   /* process id = 0, 1, ..., R-1 */
3
4   shared bool want_enter[R] = {-1, -1, ..., -1};
5   shared uint last[R-1];
6
7   void enter_critical_section(uint own_pid)
8   {
9       for (uint i = 0; i < R -1; i++)
10      {
11          want_enter[own_pid] = i;
12
```

```
13          last[i] = own_pid;
14
15          do
16          {
17              test = false;
18              for (uint j = 0; j < R; j++)
19              {
20                  if (j != own_pid)
21                      test = test || (want_enter[j] >= i)
22              }
23          } while ( test && (last[i] == own_pid) );       // Only enters the
                critical section when no other
24                                                                  // process
                                                                        wants to
                                                                        enter and
                                                                        the last
                                                                        request
25                                                                  // to enter is
                                                                        made by the
                                                                        current
                                                                        process
26      }
27  }
28
29  void leave_critical_section(unsigned int own_pid)
30  {
31      want_enter[own_pid] = -1;
32  }
```

> *needs clarification*

5 Soluções de Hardware

5.1 Desativar as interrupções

Num ambiente computacional com **um único processador:**

- A alternância entre processos, num ambiente **multiprogramado**, é sempre causada por um evento/dispositivo externo
  - **real time clock (RTC):** origina a transição de time-out em sistemas *preemptive*
  - **device controller:** pode causar transições *preemptive* no caso de um fenómeno de *wake up* de um **processo mais prioritário**
  - Em qualquer dos casos, o **processador é interrompido** e a execução do processo atual parada
- A garantia de acesso em **exclusão mútua** pode ser feita desativando as interrupções

- No entanto, só pode ser efetuada em **modo kernel**
  - Senão código malicioso ou com *bugs* poderia bloquear completamente o sistema

Num ambiente computacional **multiprocessador**, desativar as interrupções num único processador não tem qualquer efeito.

Todos os outro processadores (ou *cores*) continuam a responder às interrupções.

5.2 Instruções Especiais em Hardware

5.2.1 Test and Set (TAS primitive)

A função de hardware, `test_and_set` se for implementada atomicamente (i.e., sem interrupções) pode ser utilizada para construir a primitiva **lock**, que permite a entrada na zona crítica

Usando esta primitiva, é possível criar a função *lock*, que permite entrar na zona crítica

```
1  shared bool flag = false;
2
3  bool test_and_set(bool * flag)
4  {
5      bool prev = *flag;
6      *flag = true;
7      return prev;
8  }
9
10 void lock(bool * flag)
11 {
12     while (test_and_set(flag);  // Stays locked until and unlock operation is
           used
13 }
14
15 void unlock(bool * flag)
16 {
17     *flag = false;
18 }
```

5.2.2 Compare and Swap

Se implementada de forma atómica, a função `compare_and_set` pode ser usada para implementar a primitiva lock, que permite a entrada na zona crítica

O comportamento esperado é que coloque a variável a 1 sabendo que estava a 0 quando a função foi chamada e vice-versa.

```
shared int value = 0;

int compare_and_swap(int * value, int expected, int new_value)
{
    int v = *value;
    if (*value == expected)
        *value = new_value;
    return v;
}

void lock(int * flag)
{
    while (compare_and_swap(&flag, 0, 1) != 0);
}

void unlock(bool * flag)
{
    *flag = 0;
}
```

5.3 Busy Waiting

Ambas as funções anteriores são suportadas nos *Instruction Sets* de alguns processadores, implementadas de forma atómica

No entanto, ambas as soluções anteriores sofrem de **busy waiting**. A primitiva lock está no seu **estado ON** (usando o CPU) **enquanto espera** que se verifique a condição de acesso à zona crítica. Este tipo de soluções são conhecidas como **spinlocks**, porque o processo oscila em torno da variável enquanto espera pelo acesso

Em sistemas **uniprocessor**, o **busy_waiting** é **indesejado** porque causa:

- **Perda de eficiência:** O **time quantum** de um processo está a ser desperdiçado porque não está a ser usado para nada
- ** Risco de deadlock**: Se um** processo mais prioritário** tenciona efetuar um **lock** enquanto um processo menos prioritário está na sua zona crítica, **nenhum deles pode prosseguir**.
  - O processo menos prioritário tenta executar um unlock, mas não consegue ganhar acesso a um *time qantum* do CPU devido ao processo mais prioritário
  - O processo mais prioritário não consegue entrar na sua zona crítica porque o processo menos prioritário ainda não saiu da sua zona crítica

Em sistemas **multiprocessador** com **memória partilhada**, situações de busy waiting podem ser menos críticas, uma vez que a troca de processos *(preempt)* tem custos temporais associados. É preciso:

- guardar o estado do processo atual
  - variáveis

  - stack
  - $PC
- copiar para memória o código do novo processo

5.4 Block and wake-up

Em **sistemas uniprocessor** (e em geral nos restantes sistemas), existe a o requerimento de **bloquear um processo** enquanto este está à espera para entrar na sua zona crítica

A implementação das funções `enter_critical_section` e `leave_critical_section` continua a precisar de operações atómicas.

```
#define  R ... /* process id = 0, 1, ..., R-1 */

shared unsigned int access = 1;     // Note that access is an integer, not a
    boolean

void enter_critical_section(unsigned int own_pid)
{
    // Beginning of atomic operation
    if (access == 0)
        block(own_pid);

    else access -= 1;
    // Ending of atomic operation
}

void leave_critical_section(unsigned int own_pid)
{
    // Beginning of atomic operation
    if (there_are_blocked_processes)
        wake_up_one();
     else access += 1;
    // Ending of atomic operation
}
```

```
/* producers - p = 0, 1, ..., N-1 */
void producer(unsigned int p)
{
    DATA data;
    forever
    {
        produce_data(&data);
        bool done = false;
        do
        {
```

```
            lock(p);
            if (fifo.notFull())
            {
                fifo.insert(data);
                done = true;
            }
        unlock(p);
    } while (!done);
    do_something_else();
    }
}
```

```
/* consumers - c = 0, 1, ..., M-1 */
void consumer(unsigned int c)
{
    DATA data;
    forever
    {
        bool done = false;
        do
        {
            lock(c);
            if (fifo.notEmpty())
            {
                fifo.retrieve(&data);
                done = true;
            }
            unlock(c);
        } while (!done);
        consume_data(data);
        do_something_else();
    }
}
```

6 Semáforos

No ficheiro `IPC.md` são indicadas as condições e informação base para:

- Sincronizar a entrada na zona crítica
- Para serem usadas em programação concorrente
- Criar zonas que garantam a exclusão mútua

Semáforos são **mecanismos** que permitem por implementar estas condições e **sincronizar a atividade** de **entidades concorrentes em ambiente multiprogramado**

Não são nada mais do que **mecanismos de sincronização**.

6.1 Implementação

Um semáforo é implementado através de:

- Um tipo/estrutura de dados
- Duas operações **atómicas**:
  - down (ou wait)
  - up (ou signal/post)

```
typedef struct
{
    unsigned int val;   /* can not be negative */
    PROCESS *queue;     /* queue of waiting blocked processes */
} SEMAPHORE;
```

6.1.1 Operações

As únicas operações permitidas são o **incremento**, up, ou **decremento**, down, da variável de controlo. A variável de controlo, `val`, **só pode ser manipulada através destas operações!**

Não existe uma função de leitura nem de escrita para `val`.

- `down`
  - **bloqueia** o processo se `val == 0`
  - **decrementa** `val` se `val != 0`
- `up`
  - Se a `queue` não estiver vazia, **acorda** um dos processos
  - O processo a ser acordado depende da **política implementada**
  - **Incrementa** `val` se a `queue` estiver vazia

6.1.2 Solução típica de sistemas *uniprocessor*

```
/* array of semaphores defined in kernel */
#define R /* semid = 0, 1, ..., R-1 */

static SEMAPHORE sem[R];

void sem_down(unsigned int semid)
{
    disable_interruptions;
    if (sem[semid].val == 0)
        block_on_sem(getpid(), semid);
    else
        sem[semid].val -= 1;
```

```
13      enable_interruptions;
14  }
15
16  void sem_up(unsigned int semid)
17  {
18      disable_interruptions;
19      if (sem[sem_id].queue != NULL)
20          wake_up_one_on_sem(semid);
21      else
22          sem[semid].val += 1;
23      enable_interruptions;
24  }
```

A solução apresentada é típica de um sistema *uniprocessor* porque recorre à diretivas **disable_interruptions** e **enable_interruptions** para garantir a exclusão mútua no acesso à zona crítica.

Só é possível garantir a exclusão mútua nestas condições se o sistema só possuir um único processador, porque as diretivas irão impedir a interrupção do processo que está na posse do processador devido a eventos externos. Esta solução não funciona para um sistema multiprocessador porque ao executar a diretiva **disable_interruptions**, só estamos a **desativar as interrupções para um único processador**. Nada impede que noutro processador esteja a correr um processo que vá aceder à mesma zona de memória partilhada, não sendo garantida a exclusão mútua para sistemas multiprocessador.

Uma solução alternativa seria a extensão do **disable_interruptions** a todos os processadores. No entanto, iriamos estar a impedir a troca de processos noutros processadores do sistema que poderiam nem sequer tentar aceder às variáveis de memória partilhada.

6.2 Bounded Buffer Problem

```
1   shared FIFO fifo; /* fixed-size FIFO memory */
2
3   /* producers - p = 0, 1, ..., N-1 */
4   void producer(unsigned int p)
5   {
6       DATA data;
7       forever
8       {
9           produce_data(&data);
10          bool done = false;
11          do
12          {
13              lock(p);
14              if (fifo.notFull())
15              {
16                  fifo.insert(data);
17                  done = true;
```

```
18              }
19              unlock(p);
20          } while (!done);
21      do_something_else();
22      }
23  }
24
25  /* consumers - c = 0, 1, ..., M-1 */
26  void consumer(unsigned int c)
27  {
28      DATA data;
29      forever
30      {
31          bool done = false;
32          do
33          {
34              lock(c);
35              if (fifo.notEmpty())
36              {
37                  fifo.retrieve(&data);
38                  done = true;
39              }
40              unlock(c);
41          } while (!done);
42          consume_data(data);
43          do_something_else();
44      }
45  }
```

6.2.1 Como Implementar usando semáforos?

A solução para o *Bounded-buffer Problem* usando semáforos tem de:

- Garantir **exclusão mútua**
- Ausência de busy waiting

```
1  shared FIFO fifo;   /*fixed-size FIFO memory */
2  shared sem access;  /*semaphore to control mutual exclusion */
3  shared sem nslots;  /*semaphore to control number of available slots*/
4  shared sem nitems;  /*semaphore to control number of available items */
5
6
7  /* producers - p = 0, 1, ..., N-1 */
8  void producer(unsigned int p)
9  {
10     DATA val;
```

```
11
12      forever
13      {
14          produce_data(&val);
15          sem_down(nslots);
16          sem_down(access);
17          fifo.insert(val);
18          sem_up(access);
19          sem_up(nitems);
20          do_something_else();
21      }
22  }
23
24  /* consumers - c = 0, 1, ..., M-1 */
25  void consumer(unsigned int c)
26  {
27      DATA val;
28
29      forever
30      {
31          sem_down(nitems);
32          sem_down(access);
33          fifo.retrieve(&val);
34          sem_up(access);
35          sem_up(nslots);
36          consume_data(val);
37          do_something_else();
38      }
39  }
```

Não são necessárias as funções `fifo.empty()` e `fifo.full()` porque são implementadas indiretamente pelas variáveis:

- **nitens:** Número de "produtos" prontos a serem "consumidos"
  - Acaba por implementar, indiretamente, a funcionalidade de verificar se a FIFO está empty
- **nslots:** Número de slots livres no semáforo. Indica quantos mais "produtos" podem ser produzidos pelo "consumidor"
  - Acaba por implementar, indiretamente, a funcionalidade de verificar se a FIFO está full

Uma alternativa **ERRADA** a uma implementação com semáforos é apresentada abaixo:

```
1  shared FIFO fifo;   /*fixed-size FIFO memory */
2  shared sem access;  /*semaphore to control mutual exclusion */
3  shared sem nslots;  /*semaphore to control number of available slots*/
4  shared sem nitems;  /*semaphore to control number of available items */
5
6
```

```
7  /* producers - p = 0, 1, ..., N-1 */
8  void producer(unsigned int p)
9  {
10     DATA val;
11
12     forever
13     {
14         produce_data(&val);
15         sem_down(access);       // WRONG SOLUTION! The order of this
16         sem_down(nslots);       // two lines are changed
17         fifo.insert(val);
18         sem_up(access);
19         sem_up(nitems);
20         do_something_else();
21     }
22 }
23
24 /* consumers - c = 0, 1, ..., M-1 */
25 void consumer(unsigned int c)
26 {
27     DATA val;
28
29     forever
30     {
31         sem_down(nitems);
32         sem_down(access);
33         fifo.retrieve(&val);
34         sem_up(access);
35         sem_up(nslots);
36         consume_data(val);
37         do_something_else();
38     }
39 }
```

A diferença entre esta solução e a anterior está na troca de ordem de instruções `sem_down(access)` e `sem_down(nslots)`. A função `sem_down`, ao contrário das funções anteriores, **decrementa** a variável, não tenta decrementar.

Assim, o produtor tenta aceder à sua zona crítica sem primeiro decrementar o número de slots livres para ele guardar os resultados da sua produção *(needs_clarification)*

6.3 Análise de Semáforos

6.3.1 Vantagens

- **Operam ao nível do sistema operativo:**

  - As operações dos semáforos são implementadas no *kernel*
  - São disponibilizadas aos utilizadores através de *system_calls*

- São **genéricos** e **modulares**
  - por serem implementações de baixo nível, ganham **versatilidade**
  - Podem ser usados em qualquer tipo de situação de programação concorrente

6.3.2 Desvantagens

- Usam **primitivas de baixo nível**, o que implica que o programador necessita de conhecer os **princípios da programação concorrente**, uma vez que são aplicadas numa filosofia *bottom-up* - Facilmente ocorrem **race conditions** - Facilmente se geram situações de **deadlock**, uma vez que **a ordem das operações atómicas são relevantes**
- São tanto usados para implementar **exclusão mútua** como para **sincronizar processos**

6.3.3 Problemas do uso de semáforos

Como tanto usados para implementar **exclusão mútua** como para **sincronizar processos**, se as condições de acesso não forem satisfeitas, os processos são bloqueados **antes** de entrarem nas suas regiões críticas.

- Solução sujeita a erros, especialmente em situações complexas
  - pode existir **mais do que um ponto de sincronismos** ao longo do programa

6.4 Semáforos em Unix/Linux

POSIX:

- Suportam as operações de `down` e `up`
  - `sem_wait`
  - `sem_trywait`
  - `sem_timedwait`
  - `sem_post`
- Dois tipos de semáforos:
  - **named semaphores:**
    * São criados num sistema de ficheiros virtual (e.g. /dev/sem)
    * Suportam as operações:
      · `sem_open`
      · `sem_close`
      · `sem_unlink`
  - **unnamed semaphores:**
    * São *memory based*

    * Suportam as operações
        · `sem_init`
        · `sem_destroy`

System V:

- Suporta as operações:
  - `semget` : criação
  - `semop` : as diretivas `up` e `down`
  - `semctl` : outras operações

7 Monitores

Mecanismo de sincronização de alto nível para resolver os problemas de sincronização entre processos, numa perspetiva **top-down**. Propostos independentemente por Hoare e Brinch Hansen

Seguindo esta filosofia, a **exclusão mútua** e **sincronização** são tratadas **separadamente**, devendo os processos:

1. Entrar na sua zona crítica
2. Bloquear caso não possuam condições para continuar

Os monitores são uma solução que suporta nativamente a exclusão mútua, onde uma aplicação é vista como um conjunto de *threads* que competem para terem acesso a uma estrutura de dados partilhada, sendo que esta estrutura só pode ser acedida pelos métodos do monitor.

Um monitor assume que todos os seus métodos **têm de ser executados em exclusão mútua**:

- Se uma *thread* chama um **método de acesso** enquanto outra *thread* está a executar outro método de acesso, a sua **execução é bloqueada** até a outra terminar a execução do método

A sincronização entre threads é obtida usando **variáveis condicionais**:

- `wait`: A *thread* é bloqueada e colocada fora do monitor
- `signal`: Se existirem outras *threads* bloqueadas, uma é escolhida para ser "acordada"

7.1 Implementação

```
1     monitor example
2     {
3     /* internal shared data structure */
4     DATA data;
5
6     condition c; /* condition variable */
7
8     /* access methods */
```

```
9       method_1 (...)
10      {
11          ...
12      }
13      method_2 (...)
14      {
15          ...
16      }
17
18      ...
19
20      /* initialization code */
21      ...
```

7.2 Tipos de Monitores

7.2.1 Hoare Monitor

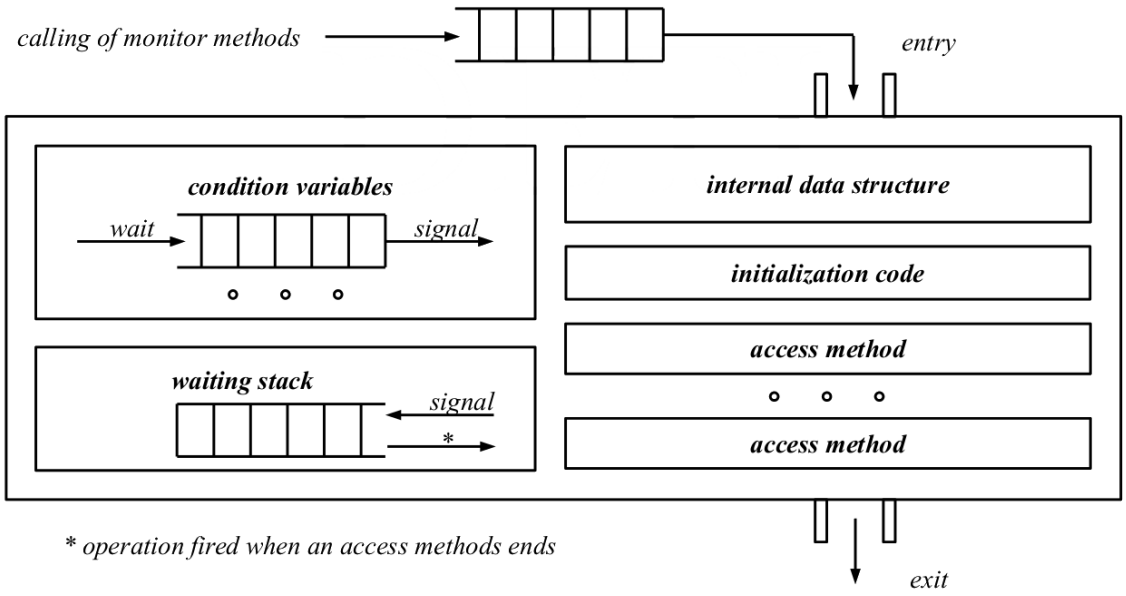

Figure 1: Diagrama da estrutura interna de um Monitor de Hoare

- Monitor de aplicação geral
- Precisa de uma stack para os processos que efetuaram um `wait` e são colocados em espera
- Dentro do monitor só se encontra a *thread* a ser executada por ele
- Quando existe um `signal`, uma *thread* é **acordada** e posta em execução

7.2.2 Brinch Hansen Monitor

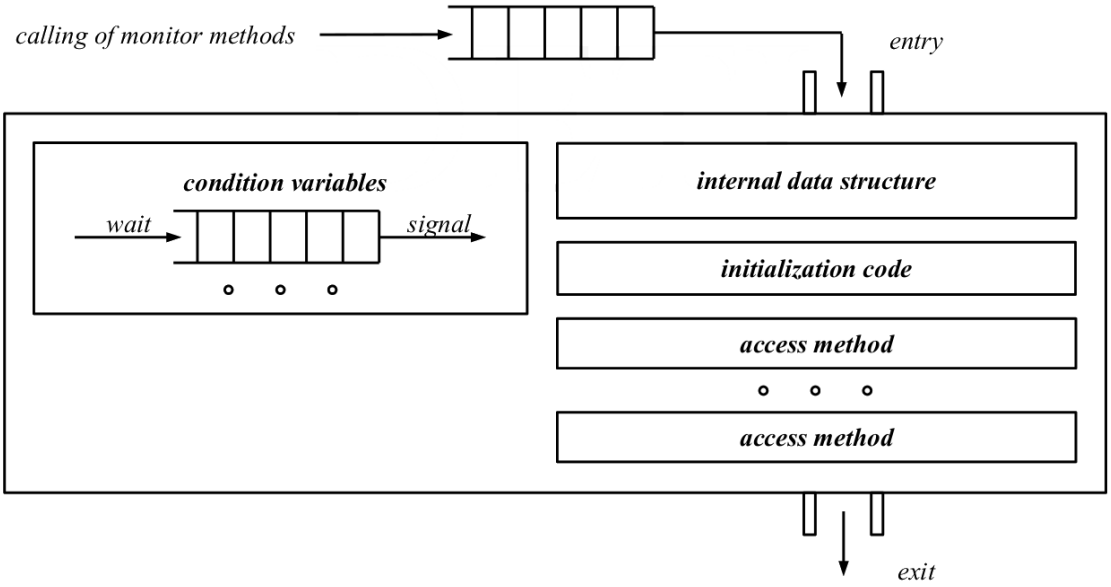

Figure 2: Diagrama da estrutura interna de um Monitor de Brinch Hansen

- A última instrução dos métodos do monitor é `signal`
  - Após o `signal` a *thread* sai do monitor
- **Fácil de implementar:** não requer nenhuma estrutura externa ao monitor
- **Restritiva: Obriga** a que cada método só possa possuir uma instrução de `signal`

7.2.3 Lampson/Redell Monitors

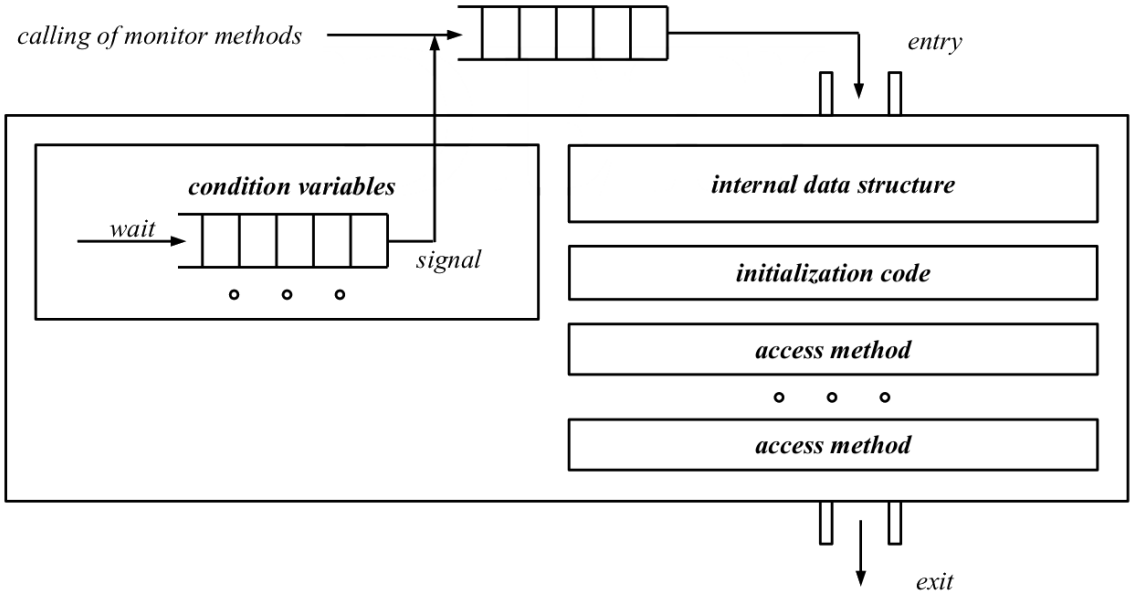

Figure 3: Diagrama da estrutura interna de um Monitor de Lampson/Redell

- A *thread* que faz o `signal` é a que continua a sua execução (entrando no monitor)
- A *thread* que é acordada devido ao `signal` fica fora do monitor, **competindo pelo acesso** ao monitor
- Pode causar **starvation**.
  - Não existem garantias que a **thread** que foi acordada e fica em competição por acesso vá ter acesso
  - Pode ser **acordada** e voltar a **bloquear**
  - Enquanto está em `ready` nada garante que outra *thread* não dê um `signal` e passe para o estado `ready`
  - A *thread* que tinha sido acordada volta a ser **bloqueada**

7.3 Bounded-Buffer Problem usando Monitores

```
1  shared FIFO fifo;            /* fixed-size FIFO memory */
2  shared mutex access;         /* mutex to control mutual exclusion */
3  shared cond nslots;      /* condition variable to control availability of slots
       */
4  shared cond nitems;      /* condition variable to control availability of items
        */
5
6  /* producers - p = 0, 1, ..., N-1 */
7  void producer(unsigned int p)
8  {
```

```
9       DATA data;
10      forever
11      {
12          produce_data(&data);
13          lock(access);
14          if/while (fifo.isFull())
15          {
16              wait(nslots, access);
17          }
18          fifo.insert(data);
19          unlock(access);
20          signal(nitems);
21          do_something_else();
22      }
23  }
24
25  /* consumers - c = 0, 1, ..., M-1 */
26  void consumer(unsigned int c)
27  {
28      DATA data;
29      forever
30      {
31          lock(access);
32          if/while (fifo.isEmpty())
33          {
34              wait(nitems, access);
35          }
36          fifo.retrieve(&data);
37          unlock(access);
38          signal(nslots);
39          consume_data(data);
40          do_something_else();
41      }
42  }
```

O uso de `if`/`while` deve-se às diferentes implementações de monitores:

- `if`: **Brinch Hansen**
  - quando a *thread* efetua o `signal` sai imediatamente do monitor, podendo entrar logo outra *thread*
- `while`: **Lamson Redell**
  - A *thread* acordada fica à espera que a *thread* que deu o `signal` termine para que possa **disputar** o acesso
- O `wait` internamente vai **largar a exclusão mútua**
  - Se não larga a exclusão mútua, mais nenhum processo consegue entrar
  - Um wait na verdade é um `lock(..)` seguido de `unlock(...)`

- Depois de efetuar uma **inserção**, é preciso efetuar um `signal` do nitems
- Depois de efetuar um **retrieval** é preciso fazer um `signal` do nslots
  - Em comparação, num semáforo quando faço o up é sempre incrementado o seu valor
- Quando uma *thread* emite um `signal` relativo a uma variável de transmissão, ela só **emite** quando alguém está à escuta
  - O `wait` só pode ser feito se a FIFO estiver cheia
  - O `signal` pode ser sempre feito

É necessário existir a `fifo.empty()` e a `fifo.full()` porque as variáveis de controlo não são semáforos binários.

O valor inicial do **mutex** é 0.

7.4 POSIX support for monitors

A criação e sincronização de *threads* usa o *Standard POSIX, IEEE 1003.1c*.

O *standard* define uma API para a **criação** e **sincronização** de *threads*, implementada em Unix pela biblioteca *pthread*

O conceito de monitor **não existe**, mas a biblioteca permite ser usada para criar monitores *Lampsom/Redell* em C/C++, usando:

- `mutexes`
- `variáveis de condição`

As funções disponíveis são:

- `ptread_create`: **cria** uma nova *thread* (similar ao *fork*)
- `ptread_exit`: equivalente à `exit`
- `ptread_join`: equivalente à `waitpid`
- `ptread_self`: equivalente à `getpid`
- `pthread_mutex_*`: manipulação de **mutexes**
- `ptread_cond_*`: manipulação de **variáveis condicionais**
- `ptread_once`: inicialização

8 Message-passing

Os processos podem comunicar entre si usando **mensagens**.

- Não existe a necessidade de possuírem memória partilhada
- Mecanismos válidos quer para sistemas **uniprocessor** quer para sistemas **multiprocessador**

A **comunicação** é efetuada através de **duas operações**:

- `send`

- `receive`

Requer a existência de um **canal de comunicação**. Existem 3 implementações possíveis:

1. **Endereçamento direto/indireto**
2. Comunicação **síncrona/assíncrona**
    - Só o `sender` é que indica o **destinatário**
    - O destinatário **não indica** o `sender`
    - Quando existem **caixas partilhadas**, normalmente usam-se mecanismos com políticas de **round-robin**
        1. Lê o processo N
        2. Lê o processo $N + 1$
        3. etc…
    - No entanto, outros métodos podem ser usados
3. **Automatic or explicit buffering**

8.1 Direct vs Indirect

8.1.1 Symmetric direct communication

O processo que pretende comunicar deve **explicitar o nome do destinatário/remetente:**

- Quando o `sender` envia uma mensagem tem de indicar o **destinatário**
    - `send(P, message`
- O destinatário tem de indicar de quem **quer receber** (`sender`)
    - `receive(P, message)`

A comunicação entre os **dois processos** envolvidos é **peer-to-peer**, e é estabelecida automaticamente entre entre um conjunto de processos comunicantes, só existindo **um canal de comunicação**

8.2 Asymmetric direct communications

Só o `sender` tem de explicitar o destinatário:

- `send(P, message`:
- `receive(id, message)`: receive mensagens de qualquer processo

8.3 Comunicação Indireta

As mensagens são enviadas para uma **mailbox** (caixa de mensagens) ou **ports**, e o `receiver` vai buscar as mensagens a uma `poll`

- `send(M, message`

- `receive(M, message)`

O canal de comunicação possui as seguintes propriedades:

- Só é estabelecido se o **par de processos** comunicantes possui uma **`mailbox` partilhada**
- Pode estar associado a **mais do que dois processos**
- Entre um par de processos pode existir **mais do que um link** (uma mailbox por cada processo)

Questões que se levantam. Se **mais do que um processo** tentar **receber uma mensagem da mesma `mailbox`** …

- … é permitido?
  - Se sim. qual dos processos deve ser bem sucedido em ler a mensagem?

8.4 Implementação

Existem várias opções para implementar o **send** e **receive**, que podem ser combinadas entre si:

- **blocking send:** o `sender` **envia** a mensagem e fica **bloqueado** até a mensagem ser entregue ao processo ou mailbox destinatária
- **nonblocking send:** o `sender` após **enviar** a mensagem, **continua** a sua execução
- **blocking receive:** o `receiver` bloqueia-se até estar disponível uma mensagem para si
- **nonblocking receiver:** o `receiver` devolve a uma mensagem válida quando tiver ou uma indicação de que não existe uma mensagem válida quando não tiver

8.5 Buffering

O link pode usar várias políticas de implementação:

- **Zero Capacity:**
  - Não existe uma `queue`
  - O `sender` só pode enviar uma mensagem de cada vez. e o envio é **bloqueante**
  - O `receiver` lê uma mensagem de cada vez, podendo ser bloqueante ou não
- **Bounded Capacity:**
  - A `queue` possui uma capacidade finita
  - Quando está cheia, o `sender` bloqueia o envio até possuir espaço disponível
- **Unbounded Capacity:**
  - A `queue` possui uma capacidade (potencialmente) infinita
  - Tanto o `sender` como o `receiver` podem ser **não bloqueantes**

8.6 Bound-Buffer Problem usando mensagens

```
shared FIFO fifo;            /* fixed-size FIFO memory */
shared mutex access;         /* mutex to control mutual exclusion */
shared cond nslots;      /* condition variable to control availability of slots */
shared cond nitems;      /* condition variable to control availability of items */

/* producers - p = 0, 1, ..., N-1 */
void producer(unsigned int p)
{
    DATA data;
    MESSAGE msg;

    forever
    {
        produce_data(&val);
        make_message(msg, data);
        send(msg);
        do_something_else();
    }
}

/* consumers - c = 0, 1, ..., M-1 */
void consumer(unsigned int c)
{
    DATA data;
    MESSAGE msg;

    forever
    {
        receive(msg);
        extract_data(data, msg);
        consume_data(data);
        do_something_else();
    }
}
```

8.7 Message Passing in Unix/Linux

System V:

- Existe uma fila de mensagens de **diferentes tipos**, representados por um inteiro
- `send` **bloqueante** se **não existir espaço disponível**
- A recepção possui um argumento para especificar o **tipo de mensagem a receber**:

  - Um tipo específico
  - Qualquer tipo
  - Um conjunto de tipos
- Qualquer que seja a política de recepção de mensagens:
  - É sempre **obtida** a mensagem **mais antiga** de uma dado tipo(s)
  - A implementação do `receive` pode ser **blocking** ou **nonblocking**
- System calls:
  - `msgget`
  - `msgsnd`
  - `msgrcv`
  - `msgctl`

POSIX

- Existe uma **priority queue**
- `send` **bloqueante** se **não existir espaço disponível**
- `receive` obtêm a mensagem **mais antiga** com a **maior prioridade**
  - Pode ser blocking ou nonblocking
- Funções:
  - `mq_open`
  - `mq_send`
  - `mq_receive`

9 Shared Memory in Unix/Linux

- É um recurso gerido pelo sistema operativo

Os espaços de endereçamento são **independentes** de processo para processo, mas o **espaço de endereçamento** é virtual, podendo a mesma **região de memória física**(memória real) estar mapeada em mais do que uma **memórias virtuais**

9.1 POSIX Shared Memory

- Criação:
  - `shm_open`
  - `ftruncate`
- Mapeamento:
  - `mmap`
  - `munmap`

- Outras operações:
  - `close`
  - `shm_unlink`
  - `fchmod`
  - …

9.2 System V Shared Memory

- Criação:
  - `shmget`
- Mapeamento:
  - `shmat`
  - `shmdt`
- Outras operações:
  - `shmctl`

10 Deadlock

- **recurso:** algo que um processo precisa para prosseguir com a sua execução. Podem ser:
  - **componentes físicos** do sistema computacional, como:
    * processador
    * memória
    * dispositivos de I/O
    * …
  - **estruturas de dados partilhadas**. Podem estar definidas
    * Ao nível do sistema operativo
      · PCT
      · Canais de Comunicação
    * Entre vários processos de uma aplicação

Os recursos podem ser:

- **preemptable:** podem ser retirados aos processos que estão na sua posse por entidades externas
  - processador
  - regiões de memória usadas no espaço de endereçamento de um processo
- **non-preemptable:** os recursos só podem ser libertados pelos processos que estão na sua posse
  - impressoras
  - regiões de memória partilhada que requerem acesso por exclusão mútua

O **deadlock** só é importante nos recursos **non-preemptable**.

O caso mais simples de deadlock ocorre quando:

1. O processo P_0 pede a posse do recurso A
    - É lhe dada a posse do recurso A, e o processo P_0 passa a possuir o recurso A em sua posse
2. O processo P_1 pede a posse do recurso B
    - É lhe dada a posse do recurso B, e o processo P_1 passa a possuir o recurso B em sua posse
3. O processo P_0 pede agora a posse do recurso B
    - Como o recurso B está na posse do processo P_1, é lhe negado
    - O processo P_0 fica em espera que o recurso B seja libertado para puder continuar a sua execução
    - No entanto, o processo P_0 não liberta o recurso A
4. O processo P_1 necessita do recurso A
    - Como o recurso A está na posse do processo P_0, é lhe negado
    - O processo P_1 fica em espera que o recurso A seja libertado para puder continuar a sua execução
    - No entanto, o processo P_1 não liberta o recurso B
5. Estamos numa situação de **deadlock**. Nenhum dos processos vai libertar o recurso que está na sua posse mas cada um deles precisa do recurso que está na posse do outro

10.1 Condições necessárias para a ocorrência de deadlock

Existem 4 condições necessárias para a ocorrência de **deadlock:**

1. **exclusão mútua:**
    - Pelo menos um dos recursos fica em posse de um processo de forma não partilhável
    - Obriga a que outro processo que precise do recurso espere que este seja libertado
2. **hold and wait:**
    - Um processo mantêm em posse pelo menos um recurso enquanto espera por outro recurso que está na posse de outro processo
3. **no preemption:**
    - Os recursos em causa são non preemptive, o que implica que só o processo na posse do recurso o pode libertar
4. **espera circular:**
    - é necessário um conjunto de processos em espera tais que cada um deles precise de um recurso que está na posse de outro processo nesse conjunto

Se **existir deadlock**, todas estas condições se verificam. *(A => B)*

Se **uma delas não se verifica**, não há deadlock. *(~B => ~A)*

10.1.1 O Problema da Exclusão Mútua

Dijkstra em 1965 enunciou um conjunto de regras para garantir o acesso **em exclusão mútua** por processo em competição por recursos de memória partilhados entre eles.[1]

1. **Exclusão Mútua:** Dois processos não podem entrar nas suas zonas críticas ao mesmo tempo
2. **Livre de Deadlock:** Se um process está a tentar entrar na sua zona crítica, eventualmente algum processo (não necessariamente o que está a tentar entrar), mas entra na sua zona crítica
3. **Livre de Starvation:** Se um processo está atentar entrar na sua zona crítica, então eventualmente esse processo entra na sua zona crítica
4. **First In First Out:** Nenhum processo a iniciar pode entrar na sua zona crítica antes de um processo que já está à espera do seu turno para entrar na sua zona crítica

10.2 Jantar dos Filósofos

- 5 filósofos sentados à volta de uma mesa, com comida à sua frente
  - Para comer, cada filósofo precisa de 2 garfos, um à sua esquerda e outro à sua direita
  - Cada filósofo alterna entre períodos de tempo em que medita ou come
- Cada **filósofo** é um **processo/thread** diferente
- Os **garfos** são os **recursos**

Uma possível solução para o problema é:

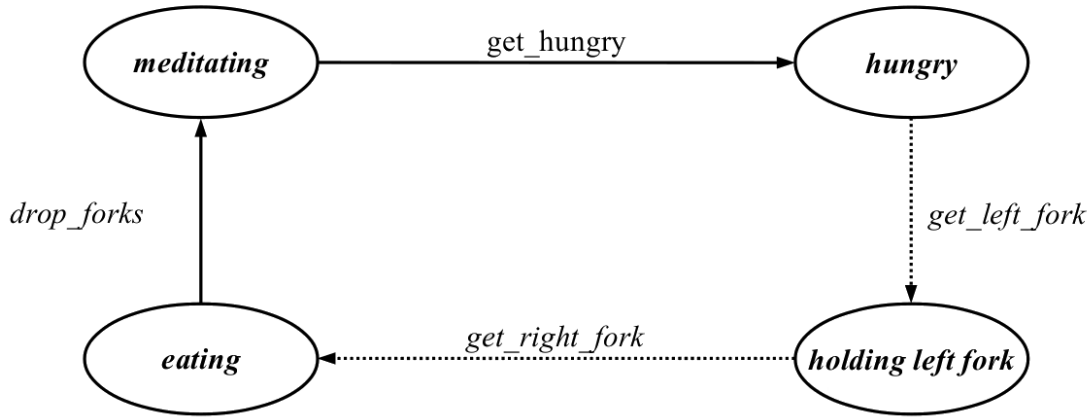

Figure 4: Ciclo de Vida de um filósofo

```
1 enum {MEDITATING, HUNGRY, HOLDING, EATING};
2
```

[1] *"Concurrent Programming, Mutual Exclusion (1965; Dijkstra)".* Gadi Taubenfeld, The Interdisciplinary Center, Herzliya, Israel

```
3  typedef struct TablePlace
4  {
5      int state;
6  } TablePlace;
7
8  typedef struct Table
9  {
10     Int semid;
11     int nplaces;
12     TablePlace place[0];
13 } Table;
14
15 int set_table(unsigned int n, FILE *logp);
16 int get_hungry(unsigned int f);
17 int get_left_fork(unsigned int f);
18 int get_right_fork(unsigned int f);
19 int drop_forks(unsigned int f);
```

Quando um filósofo fica *hungry*:

1. Obtém o garfo à sua esquerda
2. Obtém o garfo à sua direita

A solução **pode sofrer de deadlock:**

1. **exclusão mútua:**
   - Os garfos são partilháveis
2. **hold and wait:**
   - Se conseguir adquirir o `left_fork`, o filósofo fica no estado `holding_left_fork` até conseguir obter o `right_fork` e não liberta o `left_fork`
3. **no preemption:**
   - Os garfos são recursos non preemptive. Só o filósofo é que pode libertar os seus garfos após obter a sua posse e no fim de comer
4. **espera circular:**
   - Os garfos são partilhados por todos os filósofos de forma circular
     - O garfo à esquerda de um filósofo, `left_fork` é o garfo à direita do outro, `right_fork`

Se todos os filósofos estiverem a pensar e decidirem comer, pegando todos no garfo à sua esquerda ao mesmo tempo, entramos numa situação de **deadlock**.

10.3 Prevenção de Deadlock

Se uma das condições necessárias para a ocorrência de deadlock não se verificar, não ocorre deadlock.

As **políticas de prevenção de deadlock** são bastantes **restritas**, **pouco efetivas** e **difíceis de aplicar** em várias situações.

- **Negar a exclusão mútua** só pode ser aplicada a **recursos partilhados**
- **Negar *hold and wait*** requer **conhecimento *a priori* dos recursos necessários** e considera sempre o pior caso, no qual os recursos são todos necessários em simultâneo (o que pode não ser verdade)
- **Negar *no preemption***, impondo a libertação (e posterior reaquisição) de recursos adquiridos por processos que não têm condições (aka, todos os recursos que precisam) para continuar a execução pode originar grandes atrasos na execução da tarefa
- **Negar a *circular wait*** pode resultar numa má gestão de recursos

10.3.1 Negar a exclusão mútua

- Só é possível se os recursos puderem ser partilhados, senão podemos incorrer em **race conditions**
- Não é possível no jantar dos filósofos, porque os garfos não podem ser partilhados entre os filósofos
- Não é a condição mais vulgar a negar para prevenir *deadlock*

10.3.2 Negar *hold and wait*

- É possível fazê-lo se um processo é obrigado a pedir todos os recursos que vai precisar antes de iniciar, em vez de ir obtendo os recursos à medida que precisa deles
- Pode ocorrer **starvation**, porque um processo pode nunca ter condições para obter nenhum recurso
  - É comum usar *aging mechanisms* to para resolver este problema
- No jantar dos filósofos, quando um filósofo quer comer, passa a adquirir os dois garfos ao mesmo tempo
  - Se estes não tiverem disponíveis, o filósofo espera no `hungry state`, podendo ocorrer **starvation**

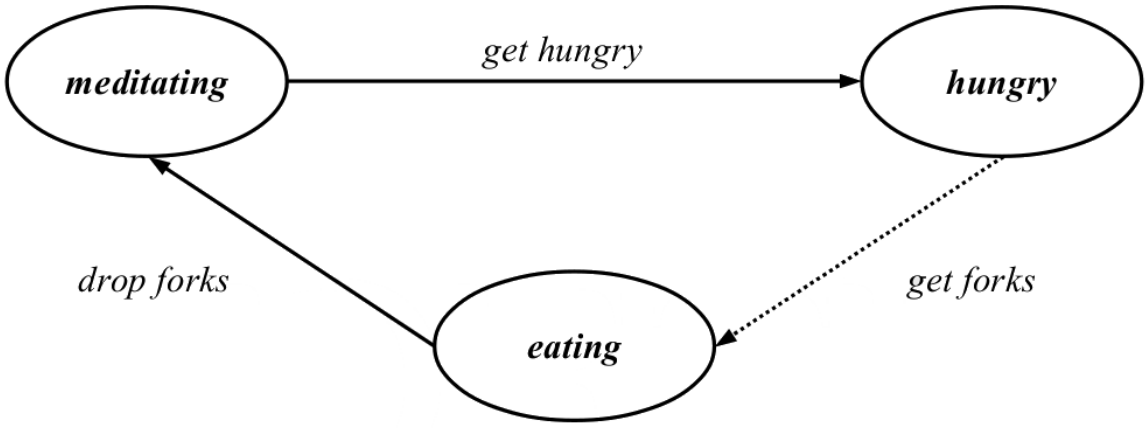

Figure 5: Negar *hold and wait*

Solução equivalente à proposta por Dijkstra.

10.3.3 Negar *no preemption*

- A condição de os recursos serem *non preemptive* pode ser implementada fazendo um processo libertar o(s) recurso(s) que possui se não conseguir adquirir o próximo recurso que precisa para continuar em execução
- Posteriormente o processo tenta novamente adquirir esses recursos
- Pode ocorrer **starvation** and **busy waiting**
  - podem ser usados *aging mechanisms* para resolver a starvation
  - para evitar busy waiting, o processo pode ser bloqueado e acordado quando o recurso for libertado
- No janta dos filósofos, o filósofo tenta adquirir o `left_fork`
  - Se conseguir, tenta adquirir o `right_fork`
    * Se conseguir, come
    * Se não conseguir, liberta o `left_fork` e volta ao estado `hungry`

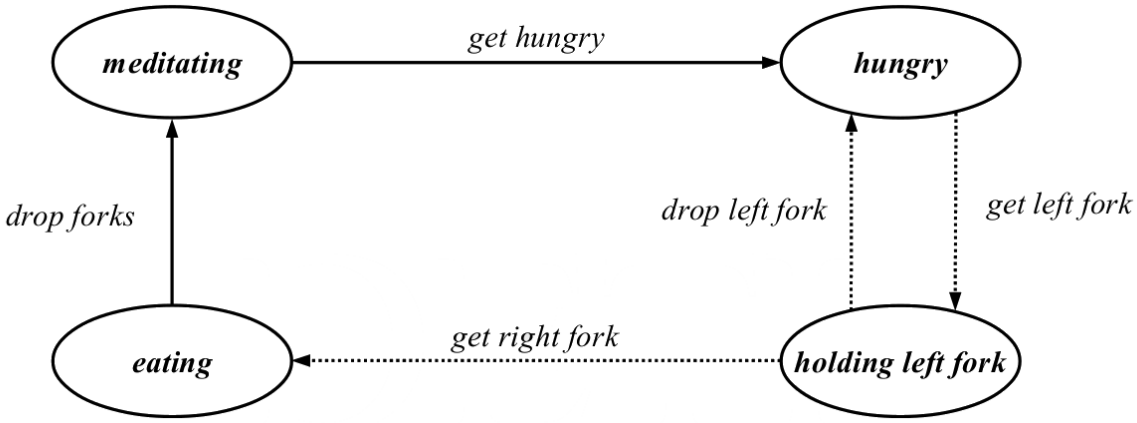

Figure 6: Negar a condição de *no preemption* dos recursos

10.3.4 Negar a espera circular

- Através do uso de IDs atribuídos a cada recurso e impondo uma ordem de acesso (ascendente ou descendente) é possível evitar sempre a espera em círculo
- Pode ocorrer **starvation**
- No jantar dos filósofos, isto implica que nalgumas situações, um dos filósofos vai precisar de adquirir primeiro o `right_fork` e de seguida o `left_fork`
  - A cada filósofo é atribuído um número entre 0 e N
  - A cada garfo é atribuído um ID (e.g., igual ao ID do filósofo à sua direita ou esquerda)
  - Cada filósofo adquire primeiro o garfo com o menor ID
  - obriga a que os filósofos 0 a N-2 adquiram primeiro o `left_fork` enquanto o filósofo N-1 adquirir primeiro o `right_fork`

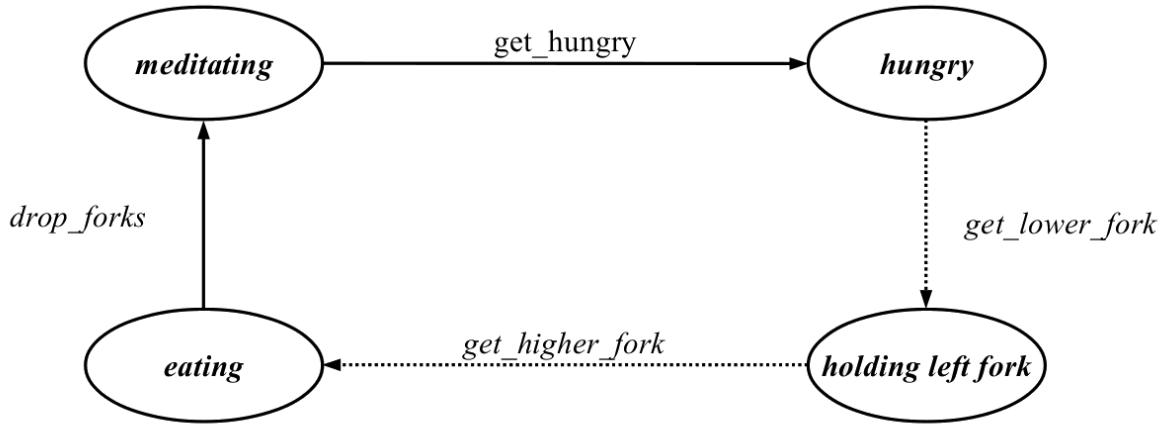

Figure 7: Negar a condição de espera circular no acesso aos recursos

10.4 Deadlock Avoidance

Forma menos restritiva para resolver situações de deadlock, em que **nenhuma das condições necessárias à ocorrência de deadlock é negada**. Em contrapartida, o sistema é **monitorizado continuamente** e um recurso **não é atribuído** se como consequência o sistema entrar num **estado inseguro/instável**

Um estado é considerado seguro se existe uma sequência de atribuição de recursos na qual todos os processos possa terminar a sua execução (não ocorrendo *deadlock*).

Caso contrário, poderá ocorrer deadlock (pode não ocorrer, mas estamos a considerar o pior caso) e o estado é considerado inseguro.

Implica que:

- exista uma lista de todos os recursos do sistema
- os processos intervenientes têm de declarar *a priori* todas as suas necessidades em termos de recursos

10.4.1 Condições para lançar um novo processo

Considerando:

- NTR_i - o número total de recursos do tipo `i` *(i = 0, 1, …, N-1)*
- $R_{i,j}$: o número de recursos do tipo `i` requeridos pelo processo `j`, *(i=0, 1, …, N-1 e j=0, 1, …, M-1)*

O sistema pode impedir um novo processo, `M`, de ser executado se a sua terminação não pode ser garantida. Para que existam certezas que um novo processo pode ser terminado após ser lançado, tem de se verificar:

$$NTR_i \geq R_{i,M} + \sum_{j=0}^{M-1} R_{i,j}$$

10.4.2 Algoritmo dos Banqueiros

Considerando:

- NTR_i: o número total de recursos do tipo `i` *(i = 0, 1, …, N-1)*
- $R_{i,j}$: o número de recursos do tipo `i` requeridos pelo processo `j`, *(i=0, 1, …, N-1 e j=0, 1, …, M-1)*
- $A_{i,j}$: o número de recursos do tipo `i` atribuídos/em posse do processo `j`, *(i=0, 1, …, N-1 e j=0, 1, …, M-1)*

Um novo recurso do tipo `i` só pode ser atribuído a um processo **se e só se** existe uma sequência $j' = f(i, j)$ tal que:

$$R_{i,j'} - A_{i,j'} < \sum_{k \geq ji'}^{M-1} A_{i,k}$$

Table 1: Banker's Algorithm Example

		A	B	C	D
	total	6	5	7	6
	free	3	1	1	2
maximum	p1	3	3	2	2
	p2	1	2	3	4
	p3	1	3	5	0
granted	p1	1	2	2	1
	p2	1	0	3	3
	p3	1	2	1	0
needed	p1	2	1	0	1
	p2	0	2	0	1
	p3	0	1	4	0
new Grant	p1	0	0	0	0
	p2	0	0	0	0
	p3	0	0	0	0

Para verificar se posso atribuir recursos a um processo, aos recursos `free` subtraio os recursos `needed`, ficando com os recursos que sobram. Em seguida simulo o que aconteceria se atribuisse o recurso ao processo, tendo em consideração que o processo pode usar o novo recurso que lhe foi atribuído sem libertar os que já possui em sua posse (estou a avaliar o pior caso, para garantir que não há deadlock)

Se o processo **p3** pedir 2 recursos do tipo C, o **pedido é negado**, porque **só existe 1 disponível**

Se o processo **p3** pedir 1 recurso do tipo B, o **pedido é negado**, porque apesar de existir 1 recurso desse tipo disponível, ao **longo da sua execução processo vai necessitar de 4** e só **existe 1 disponível**, podendo originar uma situação de **deadlock**, logo o **acesso ao recurso é negado**

Algoritmo dos banqueiros aplicado ao Jantar dos filósofos

- Cada filósofo primeiro obtém o `left_fork` e depois o `right_fork`
- No entanto, se um dos filósofos tentar obter um `left_fork` e o filósofo à sua esquerda já tem na sua posse um `left_fork`, o acesso do filósofo sem garfos ao `left_fork` é negado para não ocorrer **deadlock**

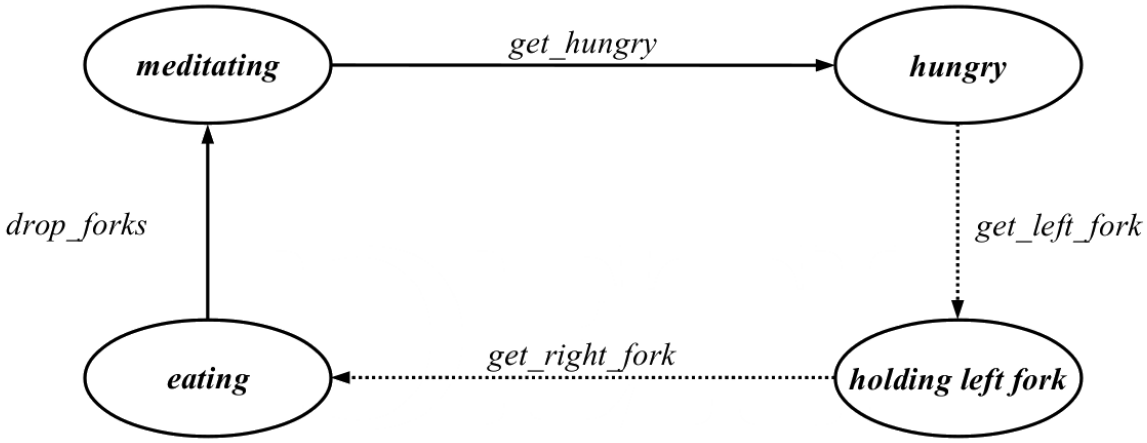

Figure 8: Algoritmo dos banqueiros aplicado ao Jantar dos filósofos

10.5 Deadlock Detection

Não são usados mecanismos nem para prevenir nem par evitar o deadlock, podendo ocorrer situações de deadlock:

- O estado dos sistema deve ser examinado para determinar se ocorreu uma situação de deadlock
  - É preciso verificar se existe uma **dependência circular de recursos** entre os processos
  - Periodicamente é executado um algoritmo que verifica o estado do registo de recursos:
    * recursos `free` vs recursos `granted` vs recursos `needed`
  - Se tiver ocorrido uma situação de deadlock, o SO deve possuir uma **rotina de recuperação** de situações de deadlock e executá-la
- Alternativamente, de um ponto de vista "arrogante", o problema pode ser ignorado

Se **ocorrer uma situação de deadlock**, a rotina de recuperação deve ser posta em prática com o objetivo de interromper a dependência circular de processos e recursos.

Existem três métodos para recuperar de deadlock:

- **Libertar recursos de um processo**, se possível
  - É atividade de um processo é suspensa até se puder devolver o recurso que lhe foi retirado
  - Requer que o estado do processo seja guardado e em seguida recarregado

  - Método eficiente

- **Rollback**
  - O estado de execução dos diferentes processos é guardado periodicamente
  - Um dos processos envolvidos na situação de deadlock é *rolled back* para o instante temporal em que o recurso lhe foi atribuído
  - A recurso é assim libertado do processo

- **Matar o processo**
  - Quando um processo entra em deadlock, é terminado
  - Método radical mas fácil de implementar

Alternativamente, existe sempre a opção de não fazer nada, entrando o processo em deadlock. Nestas situações, o utilizador é que é responsável por corrigir as situações de deadlock, por exemplo, terminando o programa com `CTRL + C`