



Sistemas de Operação

Comunicação entre Processos

António Rui Borges / Artur Pereira

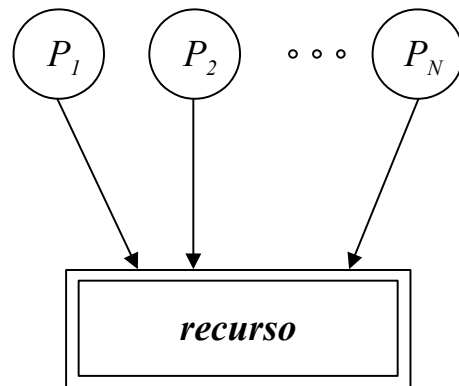
Sumário

- *Princípios gerais*
- *Propriedades do acesso a uma região crítica com exclusão mútua*
- *Soluções software*
 - *Algoritmo de Dekker (2 processos)*
 - *Algoritmo de Dijkstra (extensão a N processos)*
 - *Algoritmo de Peterson*
 - *Sua extensão a N processos*
- *Soluções hardware*
 - *Inibição e ativação de interrupções*
 - *Instruções read-modify-write*
 - *Semáforos*
 - *Monitores*
 - *Mensagens*
- *Problema do jantar dos filósofos*
 - *Resolução com semáforos e memória partilhada*
 - *Resolução com um monitor*
 - *Resolução com mensagens*
- *Ambiente fornecido pelo Unix para programação concorrente*
 - *Semáforos, memória partilhada, monitores, mensagens, sinais e pipes*
- *Deadlock*
 - *Caracterização do problema*
 - *Prevenção nos sentidos estrito e lato*
 - *Deteção e recuperação*
- *Leituras sugeridas*

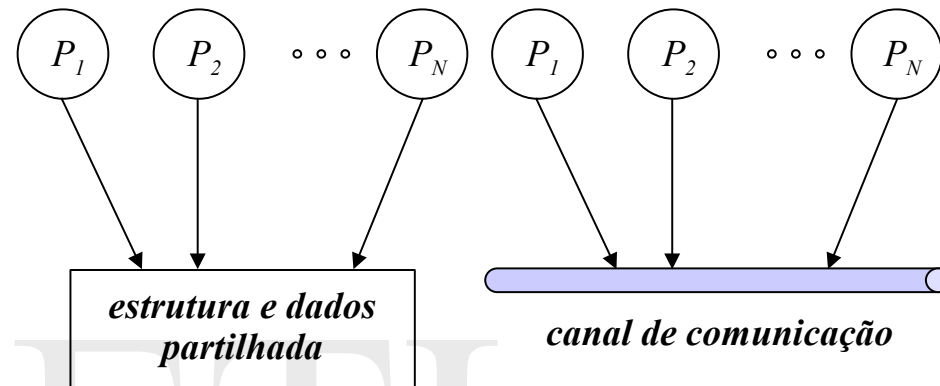
Princípios gerais - 1

- Num ambiente multiprogramado, os processos que coexistem podem, em termos de interação, ser:
- *Processos independentes* – se têm o seu 'tempo de vida' e terminam sem nunca interagirem de um modo explícito
 - a interação que ocorre é *implícita* e tem origem na sua *competição* pelos recursos do sistema computacional;
 - trata-se tipicamente de processos lançados pelos diferentes utilizadores num ambiente interativo e/ou dos processos que resultam do processamento de *jobs* num ambiente de tipo *batch*;
- *Processos cooperantes* – quando partilham informação ou comunicam entre si de um modo explícito;
 - a *partilha* exige um *espaço de endereçamento comum*
 - a *comunicação* pode ser feita tanto através da partilha de um espaço de endereçamento, ou da existência de um *canal de comunicação* que interliga os processos intervenientes.

Princípios gerais - 2



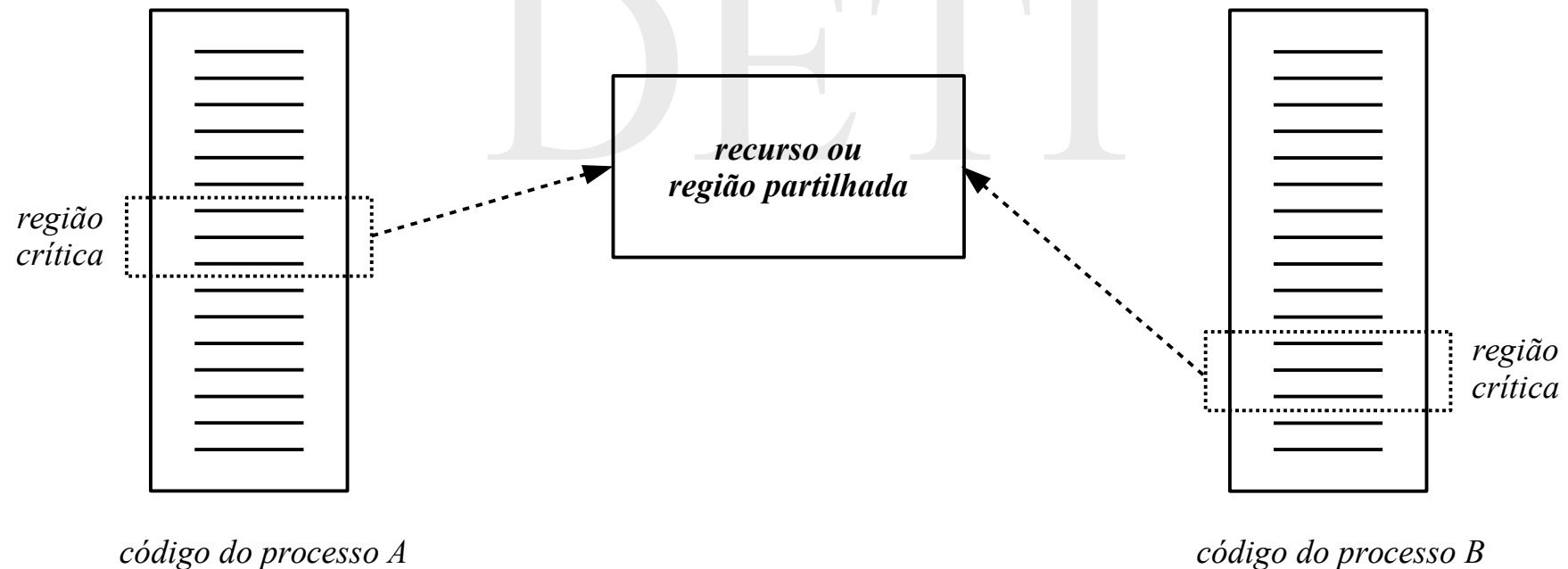
- *processos independentes* que competem por acesso a um recurso comum do sistema computacional;
- é da responsabilidade do *SO* garantir que a atribuição do recurso seja feita de uma forma controlada para que não haja perda de informação;
- isto impõe, em geral, que só um processo de cada vez possa ter acesso ao recurso (*exclusão mútua*).



- *processos cooperantes* que partilham informação ou comunicam entre si;
- é da responsabilidade dos processos envolvidos garantir que o acesso à região partilhada seja feito de uma forma controlada para que não haja perda de informação;
- isto impõe, em geral, que só um processo de cada vez possa ter acesso à região partilhada (*exclusão mútua*);
- o canal de comunicação é tipicamente um recurso do sistema computacional, logo o acesso a ele está enquadrado na *competição* por acesso a um recurso comum.

Princípios gerais - 3

- Tornando a linguagem precisa, *quando se fala em acesso por parte de um processo a um recurso, ou a uma região partilhada*, está na realidade a referir-se a execução por parte do processador do código de acesso correspondente.
- Este código, porque tem que ser executado de modo a evitar *condições de corrida* que conduzem à perda de informação, designa-se habitualmente de *região crítica*.



Princípios gerais - 4

- ♦ A imposição de exclusão mútua no acesso a um recurso, ou a uma região partilhada, pode ter, pelo seu caráter restritivo, duas consequências indesejáveis
- ♦ *deadlock* – quando dois ou mais processos ficam a aguardar eternamente o acesso às regiões críticas respetivas, esperando acontecimentos que, se pode demonstrar, nunca irão acontecer; o resultado é, por isso, o bloqueio das operações;
- ♦ *adiamento indefinido* – quando um ou mais processos competem pelo acesso a uma região crítica e, devido a uma conjunção de circunstâncias em que surgem continuamente processos novos que o(s) ultrapassam nesse desígnio, o acesso é sucessivamente adiado; está-se, por isso, perante um impedimento real à continuação dele(s).

Relação de competição por um recurso

/ processos que competem pelo recurso - $p = 0, 1, \dots, N-1$ */*

void main (**unsigned int** p)

{

forever

 {

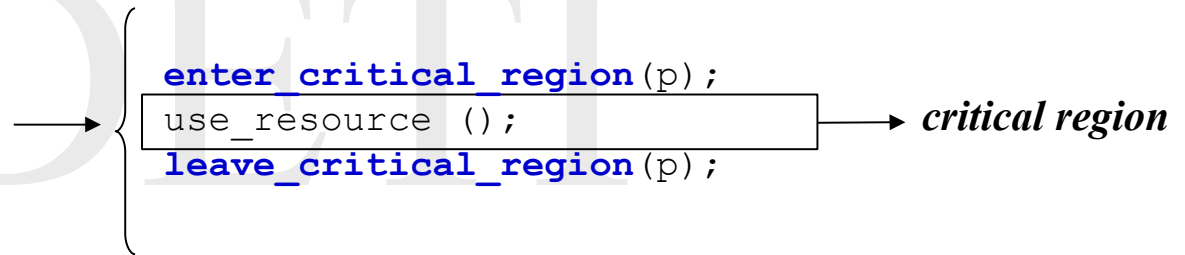
 do_something();

 access_resource(p);

 do_something_else();

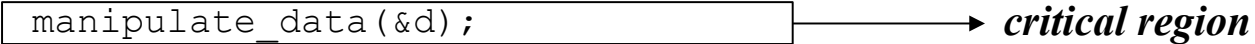
 }

}



Relação de partilha de dados

```
/* estrutura de dados partilhada */
shared DATA d;
/* processos que partilham dados - p = 0, 1, ..., N-1 */
void main (unsigned int p)
{
    forever
    {
        do_something();
        enter_critical_region(p);
        manipulate_data(&d);
        leave_critical_region(p);
        do_something_else();
    }
}
```



Relação produtor / consumidor - 1

/ estrutura de dados de comunicação: memória de tipo FIFO de tamanho fixo */*

shared FIFO fifo;

/ processos produtores - p = 0, 1, ..., N-1 */*

void main (unsigned int p)

{

DATA val;

bool done;

forever

{

produce_data(&val);

done = **false**;

do

{

enter_critical_region(p);

if (fifo.notFull())

{

fifo.insert(val);

done = **true**;

}

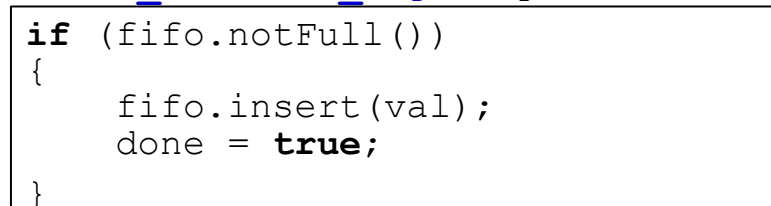
leave_critical_region(p);

} **while** (!done);

do_something_else();

}

}



critical region

Relação produtor / consumidor - 2

/ estrutura de dados de comunicação: memória de tipo FIFO de tamanho fixo */*

shared FIFO fifo;

/ processos consumidores - c = N, N+1, ..., N+M-1 */*

void main (**unsigned int** c)

{

DATA val;

bool done;

forever

{

done = **false**;

do

{

enter_critical_region(c);

if (fifo.notEmpty())

{

fifo.retrieve(&val);

done = **true**;

}

leave_critical_region(c);

} **while** (!done);

consume_data(val);

do_something_else();

}

}

DETI

→ *critical region*

Acesso a uma região crítica com exclusão mútua

- ♦ Propriedades desejáveis que a solução geral do problema deve assumir:
 - ♦ *garantia efetiva de imposição de exclusão mútua* – o acesso à região crítica associada a um mesmo recurso, ou região partilhada, só pode ser permitido a um processo de cada vez, de entre todos os processos que competem pelo acesso
 - ♦ *independência da velocidade de execução relativa dos processos intervenientes, ou do seu número* – nada deve ser presumido acerca destes fatores
 - ♦ *um processo fora da região crítica não pode impedir outro de lá entrar*
 - ♦ *não pode ser adiada indefinidamente a possibilidade de acesso à região crítica a qualquer processo que o requeira*
 - ♦ *o tempo de permanência de um processo na região crítica deve ser necessariamente finito.*

Tipo de soluções

- ♦ ***soluções software*** – são soluções que, quer sejam implementadas num monoprocessador, quer num multiprocessador com memória partilhada, supõem o recurso em última instância ao *conjunto de instruções básico* do processador ou seja, as instruções de transferência de dados de e para a memória são de tipo *standard*: leitura e escrita de um valor
- ♦ a única suposição adicional diz respeito ao caso do multiprocessador, em que a tentativa de acesso simultâneo a uma mesma posição de memória por parte de diferentes processadores é necessariamente serializada por intervenção de um árbitro;
- ♦ ***soluções hardware*** – são soluções que supõem o recurso a instruções especiais do processador para garantir, a algum nível, a atomicidade na leitura e subsequente escrita de uma mesma posição de memória
- ♦ são muitas vezes suportadas pelo próprio sistema de operação e podem mesmo estar integradas na linguagem de programação utilizada.

Alternância estrita - 1

```
/* estrutura de dados de controlo */

#define R    ...    /* número de processos que pretendem acesso à região crítica */
                  /* process id = 0, 1, ..., R-1 */

shared unsigned int access_turn = 0;

void enter_critical_region(unsigned int own_pid)
{
    while (own_pid != access_turn);
}

void leave_critical_region(unsigned int own_pid)
{
    if (own_pid == access_turn)
        access_turn = (access_turn + 1) % R;
}
```

Alternância estrita - 2

Análise Crítica

- ♦ A proposta anterior, ao supor a entrada na região crítica dos processos intervenientes num regime de alternância estritamente sucessiva, não constitui uma solução geral para o problema de acesso a uma região crítica com exclusão mútua.
- ♦ Duas propriedades, anteriormente apresentadas como desejáveis, são violadas
 - ♦ *independência da velocidade de execução relativa dos processos intervenientes* – o ritmo de execução desenvolve-se ao ritmo do processo que faz menos acessos por unidade de tempo à região crítica;
 - ♦ *um processo fora da região crítica não pode impedir outro de lá entrar* - se não for a sua vez de entrada, um processo terá que aguardar, mesmo que nenhum outro processo esteja correntemente a aceder à região crítica.

Construção de uma solução - 1

```
/* estrutura de dados de controlo */

#define R    2          /* número de processos que pretendem acesso à região crítica,
                          process id = 0, 1 */

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

void enter_critical_region(unsigned int own_pid)
{
    unsigned int other_pid_ = 1 - own_pid;
    while (is_in[other_pid]);
    is_in[own_pid] = true;
}

void leave_critical_region(unsigned int own_pid)
{
    is_in[own_pid] = false;
}
```

Construção de uma solução - 2

Análise Crítica

- Uma análise cuidada permite verificar que a exclusão mútua não é garantida em todas as circunstâncias.
- Considere-se a sucessão
 - o processo P_0 invoca a função `enter_critical_region` e testa a variável `is_in[1]` que tem nesse momento o valor *false*;
 - o processo P_1 invoca a função `enter_critical_region` e testa a variável `is_in[0]` que ainda mantém o valor *false*;
 - o processo P_1 altera o valor da variável `is_in[1]` para *true* e acede à região crítica;
 - o processo P_0 altera o valor da variável `is_in[0]` para *true* e acede à região crítica;Ambos os processos entram simultaneamente na região crítica!
- Aparentemente, o problema resulta do facto de *se proceder ao teste da variável do outro e só depois se alterar o valor da variável própria.*

Construção de uma solução - 3

```
/* estrutura de dados de controlo */  
#define R    2          /* número de processos que pretendem acesso à região crítica,  
                        pid = 0, 1 */  
  
shared bool want_enter[R] = {false, false};  
  
void enter_critical_region (unsigned int own_pid)  
{  
    unsigned int other_pid = 1 - own_pid;  
    want_enter[own_pid] = true;  
    while (want_enter[other_pid]);  
}  
  
void leave_critical_region (unsigned int own_pid)  
{  
    want_enter[own_pid] = false;  
}
```

Construção de uma solução - 4

Análise Crítica

- A exclusão mútua passou a ser garantida em todas as circunstâncias, mas surgiu uma consequência perversa - possibilidade de ocorrência de *deadlock*.
- Considere a sucessão
 - o processo P_0 invoca a função `enter_critical_region` e altera o valor da variável `want_enter[0]` para *true*;
 - o processo P_1 invoca a função `enter_critical_region` e altera o valor da variável `want_enter[1]` para *true*;
 - o processo P_1 testa a variável `want_enter[0]` e, como ela tem o valor *true*, fica a aguardar acesso à região crítica;
 - o processo P_0 testa a variável `want_enter[1]` e, como ela tem o valor *true*, fica a aguardar acesso à região crítica.

Os dois processos entram em *deadlock*

- Numa situação de contenção como esta, *pelo menos um dos processos terá que recuar temporariamente para que o impasse possa ser resolvido!*

Construção de uma solução - 5

```
/* estrutura de dados de controlo */
#define R 2 /* número de processos que pretendem acesso à região crítica,
           process id = 0, 1 */

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

void enter_critical_region(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;
        random_delay();
        want_enter[own_pid] = true;
    }
}

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

Construção de uma solução - 6

Análise Crítica

- ♦ Trata-se de uma solução quase válida e, como tal, é por vezes usada na prática.
 - ♦ Um exemplo disso é o *protocolo Ethernet* que utiliza uma variante deste algoritmo para estabelecer a exclusão mútua no acesso ao canal de comunicação.
- ♦ Sob o ponto de vista teórico, no entanto, é incorrecta.
 - ♦ Pode sempre considerar-se a ocorrência de uma combinação de circunstâncias, por pouco provável que seja, que conduza inevitavelmente a situações de *deadlock* ou de *adiamento indefinido*.
 - ♦ *Procure encontrar exemplos que ilustrem cada uma destas situações!*
- ♦ A situação de contenção tem que ser resolvida de uma maneira *determinística* e não de uma maneira *aleatória*.

Algoritmo de Dekker (1965) - 1

```
#define R      2          /* number of processes accessing the critical region,
                           process id = 0, 1 */

shared bool want_enter[R] = {false, false};
shared uint p_w_priority = 0;

void enter_critical_region(uint own_pid)
{
    uint other_pid = 1 - own_pid;
    want_enter[own_pid] = true;
    while (want_enter[other_pid])
    {
        if (own_pid != p_w_priority)
        {
            want_enter[own_pid] = false;
            while (own_pid != p_w_priority);
            want_enter[own_pid] = true;
        }
    }
}

void leave_critical_region (uint own_pid)
{
    uint other_pid = 1 - own_pid;
    p_w_priority = other_pid;
    want_enter[own_pid] = false;
}
```

Algoritmo de Dekker (1965) - 2

Análise Crítica

- ♦ O *algoritmo de Dekker* usa um mecanismo de alternância para resolver o conflito resultante da situação de contenção de dois processos.
 - ♦ Garante efetivamente a exclusão mútua no acesso à região crítica
 - ♦ Evita *deadlock* e *adiamento indefinido*
 - ♦ Não presume o que quer que seja quanto à velocidade de execução relativa dos processos intervenientes.
- ♦ A sua generalização a N processos não é, contudo, fácil. De facto, não se conhece qualquer algoritmo que o faça, respeitando todas as propriedades desejáveis para a solução.
- ♦ O algoritmo apresentado a seguir, devido a Dijkstra (1966), não elimina o risco de *adiamento indefinido*. Porquê?

Algoritmo de Dijkstra (1966) - 1

```
#define R      ...      /* number of processes accessing the critical region,
                           process id = 0, 1, ..., R-1 */

shared uint want_enter[R] = {NO, NO, ... , NO};
shared uint p_w_priority = 0;

void enter_critical_region(uint own_pid)
{
    uint n;
    do
    {
        want_enter[own_pid] = WANT;
        while (own_pid != p_w_priority)
            if (want_enter[p_w_priority] == NO)
                p_w_priority = own_pid;
        want_enter[own_pid] = DECIDED;
        for (n = 0; n < R; n++)
            if (n != own_pid && want_enter[n] == DECIDED)
                break;
    } while (n < R);
}

void leave_critical_region(uint own_pid)
{
    p_w_priority = (own_pid + 1) % R;
    want_enter[own_pid] = NO;
}
```

Algoritmo de Peterson (1981) - 1

```
#define R    2           /* number of processes accessing the critical region,  
                           process id = 0, 1 */  
  
shared bool want_enter[R] = {false, false};  
shared uint last;  
  
void enter_critical_region(uint own_pid)  
{  
    uint other_pid = 1 - own_pid;  
  
    want_enter[own_pid] = true;  
    last = own_pid;  
    while ((want_enter[other_pid]) && (last == own_pid));  
}  
  
void leave_critical_region(uint own_pid)  
{  
    want_enter[own_pid] = false;  
}
```


Algoritmo de Peterson (1981) - 2

Análise Crítica

- O *algoritmo de Peterson* usa a serialização por ordem de chegada para resolver o conflito resultante da situação de contenção de dois processos. Isto é conseguido forçando cada processo a escrever a sua identificação numa mesma variável (*last*). Assim, uma leitura subsequente permite por comparação determinar qual foi o último que aí escreveu.
- Este algoritmo garante efetivamente a exclusão mútua no acesso à região crítica, evita *deadlock* e *adiamento indefinido* e não presume o que quer que seja quanto à velocidade de execução relativa dos processos intervenientes.
- A sua generalização a N processos é quase direta, se se tiver em conta a analogia de formação de uma fila de espera. Existe, contudo, uma diferença subtil entre a disciplina implementada no algoritmo e a da formação de uma fila de espera convencional. *Consegue descobri-la?*

Extensão do algoritmo de Peterson - 1

```
#define R      ...      /* number of processes accessing the critical region,
                           process id = 0, 1, ..., R-1 */

shared int want_enter[R] = {-1, -1, ... , -1};
shared int last[R-1];

void enter_critical_region(uint own_pid)
{
    for (uint i = 0; i < R-1; i++)
    {
        want_enter[own_pid] = i;
        last[i] = own_pid;
        do
        {
            test = false;
            for (uint j = 0; j < R; j++)
                if (j != own_pid)
                    test = test || (want_enter[j] >= i);
        } while (test && (last[i] == own_pid));
    }
}

void leave_critical_region(int own_pid)
{
    want_enter[own_pid] = -1;
}
```

Sumário

- *Princípios gerais*
- *Propriedades do acesso a uma região crítica com exclusão mútua*
- *Soluções software*
 - *Algoritmo de Dekker (2 processos)*
 - *Algoritmo de Dijkstra (extensão a N processos)*
 - *Algoritmo de Peterson*
 - *Sua extensão a N processos*
- *Soluções hardware*
 - *Inibição e ativação de interrupções*
 - *Instruções read-modify-write*
 - *Semáforos*
 - *Monitores*
 - *Mensagens*
- *Problema do jantar dos filósofos*
 - *Resolução com semáforos e memória partilhada*
 - *Resolução com um monitor*
 - *Resolução com mensagens*
- *Ambiente fornecido pelo Unix para programação concorrente*
 - *Semáforos, memória partilhada, monitores, mensagens, sinais e pipes*
- *Deadlock*
 - *Caracterização do problema*
 - *Prevenção nos sentidos estrito e lato*
 - *Deteção e recuperação*
- *Leituras sugeridas*

Intruções de ativação e de inibição das interrupções - 1

Sistemas Computacionais Monoprocessador

- ♦ A *comutação de processos* num ambiente de multiprogramação é, sempre que existe *preemption*, despoletada por ação de um dispositivo externo
 - ♦ *relógio de tempo real (RTC)* – quando se esgota a janela de execução que foi atribuída, originando a transição *timer-run-out*;
 - ♦ *controlador associado a um dispositivo específico* – que acorda um processo de prioridade mais elevada, originando a transição *priority superseded*;
- ♦ Ela é condicionada por interrupções ao processador.
- ♦ Assim, o acesso com exclusão mútua a uma região crítica seria garantido se, antes do início de execução do código respetivo, as interrupções ao processador forem inibidas, sendo reposta a situação no final.

Intruções de ativação e de inibição das interrupções - 2

- ♦ Esta abordagem, contudo, só é válida em termos do *kernel*.
- ♦ A sua generalização aos processos utilizador conduziria facilmente ao bloqueio completo do sistema.
 - ♦ Um *bug*, que provocasse um ciclo infinito dentro da região crítica, ou que pusesse o processador a aguardar uma transação de entrada / saída que ainda não tivesse ocorrido, impediria qualquer tipo de recuperação
 - ♦ Além disso, um utilizador de má fé poderia sempre apropriar-se sem grande esforço de todos os recursos do sistema.

Sistemas Computacionais Multiprocessador com Memória Partilhada

- ♦ Sendo aqui a contenção estabelecida entre processadores distintos, o recurso à inibição e ativação das interrupções não tem qualquer efeito.

Instruções de tipo read-modify-write - 1

- Se na função `enter_critical_region` fosse possível garantir a *atomicidade* nas leitura e escrita sucessivas da variável `busy`, o acesso com exclusão mútua a uma região crítica poderia ser implementado com uma construção do tipo abaixo.
- Este tipo de construção designa-se habitualmente de *flag de locking*.

```
shared bool busy = false;
```

```
void enter_critical_region(bool busy)
```

```
{
```

```
    bool prev;
```

```
    do
```

```
    {
```

```
        prev = busy;
```

```
        busy = true;
```

```
    } while (prev);
```

```
}
```

```
void leave_critical_region(bool busy)
```

```
{
```

```
    busy = false;
```

```
}
```

→ { *operação atômica*
(a sua execução não pode ser interrompida)

Instruções de tipo read-modify-write - 2

- As instruções convencionais do *conjunto de instruções* de um processador efetuam apenas a leitura ou a escrita de uma posição de memória na mesma instrução, não permitindo a implementação de uma *flag de locking*.
- Os processadores atuais possuem uma instrução especial, designada normalmente por *test-and-set (tas)*, que permite em sucessão e de uma maneira atômica, a leitura de uma posição de memória, a afetação do registo de *status* em função do seu conteúdo e a escrita na mesma posição de um valor diferente de zero.
- Assumindo que, quando a região crítica não está a ser acedida, a posição de memória reservada para a variável `busy` tem o valor zero (`false`), tem-se então que

```
void enter_critical_region (bool * busy)
{
    lea    ad_reg, busy
loop: tas  (ad_reg)
    bnz   loop
}
```

$\} \longleftrightarrow \{$

```
while (!lock (busy));
```

Busy waiting - 1

- Um problema comum às *soluções software* e ao uso de *flags de locking*, implementadas a partir de instruções de tipo *read-modify-write*, é que os processos intervenientes aguardam a entrada na região crítica no estado ativo – *busy waiting*.
- Este facto é indesejável em sistemas comput. monoprocessador já que conduz a
 - *perda de eficiência* – a atribuição do processador a um processo que pretende acesso a uma região crítica, associada a um recurso ou uma região partilhada em que um segundo processo se encontra na altura no seu interior, faz com que o intervalo de tempo de atribuição do processador se esgote sem que qualquer trabalho útil tenha sido realizado;
 - *constrangimentos no estabelecimento do algoritmo de escalonamento* – numa política *preemptive* de escalonamento onde os processos que competem por um mesmo recurso, ou partilham uma mesma região de dados, têm prioridades diferentes, existe o risco de *deadlock* se for possível ao processo de mais alta prioridade interromper a execução do outro.
- Assim, torna-se conveniente procurar soluções em que um processo bloqueie quando é impedido de entrar na região crítica respetiva.

Busy waiting - 2

- Em sistemas computacionais multiprocessador com memória partilhada, e mais concretamente no caso de multiprocessamento simétrico, o problema de *busy waiting* não é tão crítico.
- Quando a execução do código das regiões críticas tem uma duração relativamente curta, a alternativa de bloquear o processo enquanto aguarda uma oportunidade de acesso à região crítica, exige uma *mudança de contexto* para calendarizar outro processo para execução nesse processador e pode tornar-se, por isso, menos eficiente.
- É neste contexto que as *flags de locking* se tornam importantes, sendo também referidas na literatura pelo nome de *spinlocks* (o processo *gira* em torno da variável enquanto aguarda o *locking*).

sleep e wake up

- O recurso direto a primitivas de tipo *sleep e wake up* não resolve por si só o problema. Continua a ser necessário garantir a *atomicidade* das operações!

```
#define R    ...    /* number of processes accessing the critical region,  
                    process id = 0, 1, ..., R-1 */
```

```
shared unsigned int access = 1;
```

```
void enter_critical_region(unsigned int own_pid)
```

```
{  
    if (access == 0) sleep(own_pid);  
    else access -= 1;  
}
```

→ { *operação atômica*
(a sua execução não pode ser interrompida)

```
void leave_critical_region(unsigned int own_pid)
```

```
{  
    if (there_are_blocked_processes) wake_up_one();  
    else access += 1;  
}
```

→ { *operação atômica*
(a sua execução não pode ser interrompida)

Semáforos - 1

- Um *semáforo* é um dispositivo de sincronização, originalmente inventado por Dijkstra, que pode ser concebido como uma variável do tipo

```
typedef struct
{
    unsigned int val; /* valor de contagem */
    NODE *queue;      /* fila de espera dos processos bloqueados */
} SEMAPHORE;
```

sobre a qual é possível executar as duas operações atômicas seguintes

- *sem_down* – se o campo `val` for não nulo, o seu valor é decrementado; caso contrário, o processo que executou a operação é bloqueado e a sua identificação é colocada na fila de espera `queue`;
- *sem_up* – se houver processos bloqueados na fila de espera `queue`, um deles é acordado (de acordo com uma qualquer disciplina previamente definida); caso contrário, o valor do campo `val` é incrementado.
- Um semáforo só pode ser manipulado desta maneira e *é precisamente para garantir isso* que toda e qualquer referência a um semáforo particular é sempre feita de uma forma indireta.

Semáforos - 2

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

static SEMAPHORE sem[R];

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

void sem_up(unsigned int semid)
{
    disable_interruptions;
    if (sem[sem_id].queue != NULL) // queue not empty
        wake_up_one_on_sem(semid);
    else
        sem[semid].val += 1;
    enable_interruptions;
}
```

- Os semáforos podem ser binários e não binários
- Como implementar *exclusão mútua* usando semáforos?

- Implementação característica de um monoprocesador
Porquê?
- Como fazer num multiprocessador?

Problema dos produtores / consumidores

```
shared FIFO fifo;    /* fixed-size FIFO memory */

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

    forever
    {
        produce_data(&val);
        done = false;
        do
        {
            enter_critical_region(p);
            if (fifo.notFull())
            {
                fifo.insert(val);
                done = true;
            }
            leave_critical_region(p);
        } while (!done);
        do_something_else();
    }
}

/* consumers - c = N, N+1, ..., N+M-1 */
void consumer(unsigned int c)
{
    DATA val;
    bool done;

    forever
    {
        done = false;
        do
        {
            enter_critical_region(c);
            if (fifo.notEmpty())
            {
                fifo.retrieve(&val);
                done = true;
            }
            leave_critical_region(c);
        } while (!done);
        consume_data(val);
        do_something_else();
    }
}
```

- Como implementar usando semáforos?
 - Garantindo exclusão mútua e ausência de *busy waiting*

Problema dos produtores / consumidores (semáforos)

```
shared FIFO fifo;      /* fixed-size FIFO memory */
shared int access;     /* mutual exclusion access semaphore */
shared int emptyness; /* control of empty cells semaphore */
shared int fullness;   /* control of occupied cells semaphore */
```

```
/* producers - p = 0, 1, ..., N-1 */
```

```
void producer(unsigned int p)
{
    DATA val;

    forever
    {
        produce_data(&val);
        sem_down(emptyness);
        sem_down(access);
        fifo.insert(val);
        sem_up(access);
        sem_up(fullness);
        do_something_else();
    }
}
```

```
/* consumers - c = N, N+1, ..., N+M-1 */
```

```
void consumer(unsigned int c)
{
    DATA val;

    forever
    {
        sem_down(fullness);
        sem_down(access);
        fifo.retrieve(&val);
        sem_up(access);
        sem_up(emptyness);
        consume_data(val);
        do_something_else();
    }
}
```

- `fifo.empty()` e `fifo.full()` não são necessárias. *Porquê?*
- Valores iniciais dos semáforos?

Crítica ao uso de semáforos

- ♦ A construção de soluções concorrentes baseadas em *semáforos* apresenta vantagens e desvantagens
- ♦ *vantagens*
 - ♦ *suporte ao nível do sistema de operação* – porque a sua implementação é feita pelo *kernel*, as operações sobre semáforos estão diretamente disponíveis ao programador de aplicações, constituindo-se como uma biblioteca de *chamadas ao sistema* que podem ser usadas em qualquer linguagem de programação;
 - ♦ *universalidade* – são construções de muito baixo nível e podem, portanto, devido à sua versatilidade, ser usadas no desenho de qualquer tipo de soluções;
- ♦ *desvantagens*
 - ♦ *conhecimento especializado* – a sua manipulação direta exige ao programador um domínio completo dos princípios da programação concorrente, pois é muito fácil cometer erros que originam *condições de corrida* e, mesmo, *deadlock*.
 - ♦ Veja-se o exemplo seguinte!

Problema dos produtores / consumidores (semáforos)

```
shared FIFO fifo;      /* fixed-size FIFO memory */
shared int access;     /* mutual exclusion access semaphore */
shared int emptyness; /* control of empty cells semaphore */
shared int fullness;   /* control of occupied cells semaphore */
```

```
/* producers - p = 0, 1, ..., N-1 */
```

```
void producer(unsigned int p)
{
    DATA val;

    forever
    {
        produce_data(&val);
        sem_down(access);
        sem_down(emptyness);
        fifo.insert(val);
        sem_up(access);
        sem_up(fullness);
        do_something_else();
    }
}
```

```
/* consumers - c = N, N+1, ..., N+M-1 */
```

```
void consumer(unsigned int c)
{
    DATA val;

    forever
    {
        sem_down(fullness);
        sem_down(access);
        fifo.retrieve(&val);
        sem_up(access);
        sem_up(emptyness);
        consume_data(val);
        do_something_else();
    }
}
```

- Que está mal nesta implementação?

Monitores - 1

- Conceptualmente, o principal problema com o uso dos *semáforos* é que eles servem simultaneamente para garantir o acesso com exclusão mútua a uma região crítica e para sincronizar os processos intervenientes.
- Assim, e porque se trata de primitivas de muito baixo nível, a sua aplicação é feita segundo uma perspectiva *bottom-up* (os processos são bloqueados antes de entrarem na região crítica, se as condições à sua continuação não estiverem reunidas) e não *top-down* (os processos entram na região crítica e bloqueiam, se as condições à sua continuação não estiverem reunidas).
- A primeira abordagem torna-se logicamente confusa e muito sujeita a erros, sobretudo em interações de alguma complexidade, porque as primitivas de sincronização podem estar dispersas por todo o programa.
- Uma solução é introduzir *ao nível da própria linguagem de programação* uma construção [concorrente] que trate separadamente o acesso com exclusão mútua a uma dada região de código e a sincronização dos processos.

Monitores - 2

- Um *monitor* é um dispositivo de sincronização, proposto de uma forma independente por Hoare e por Brinch Hansen, que pode ser concebido como um módulo especial, suportado pela linguagem de programação [concorrente] e constituído por uma estrutura de dados interna, por código de inicialização e por um conjunto de primitivas de acesso.

```
monitor example
  (* internal data structure *)
  var
    val: DATA;                (* shared data *)
    c: condition;            (* synchronization condition variable *)

  (* access methods *)
  procedure pa1 (...);
  end (* pa1 *)
  function pa2 (...): real;
  end (* pa2 *)
  (* initialization *)

  begin
    ...
  end
end monitor;
```

Monitores - 3

- Uma aplicação escrita numa linguagem concorrente que implementa o *paradigma de variáveis partilhadas*, é vista como um conjunto de *threads* que competem pelo acesso a estruturas de dados partilhadas.
- Quando as estruturas de dados são implementadas com *monitores*, a linguagem de programação garante que a execução de uma primitiva do *monitor* é feita em regime de exclusão mútua.
 - O compilador, ao compilar um *monitor*, gera o código necessário à imposição desta situação.
- Um *thread* entra no *monitor* por invocação de uma das primitivas
 - Única forma de acesso à estrutura de dados interna.
- Como a execução das primitivas decorre em regime de exclusão mútua, quando um outro *thread* está no seu interior, o *thread* é bloqueado à entrada, aguardando a sua vez.

Monitores - 4

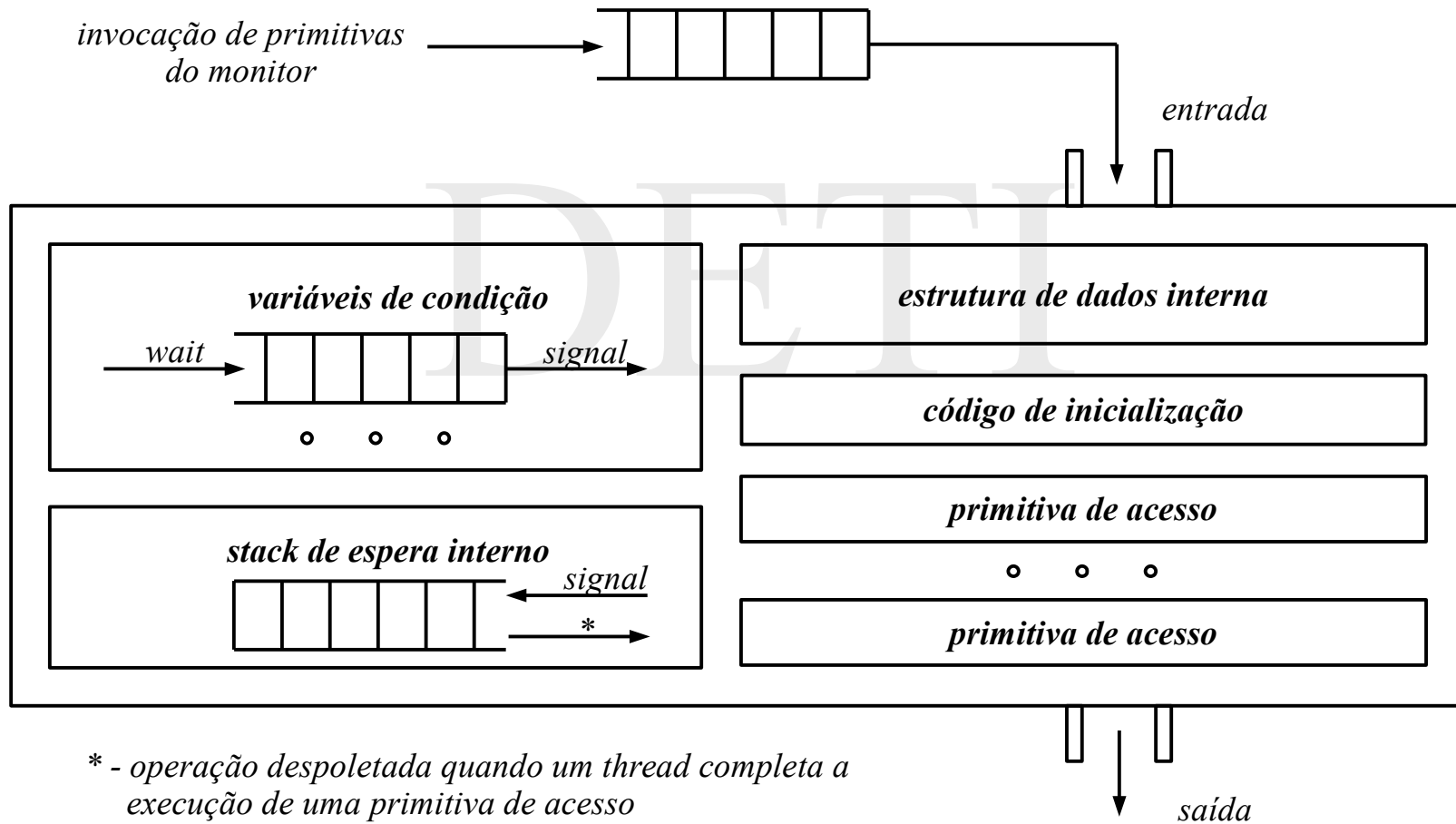
- ♦ A sincronização entre *threads* é gerida pelas *variáveis de condição*.
- ♦ Existem duas operações que podem ser executadas sobre uma *variável de condição*
 - ♦ *wait* – o *thread* que invoca a operação é bloqueado na *variável de condição* argumento e é colocado *fora do monitor* para possibilitar que um outro *thread* que aguarda acesso, possa prosseguir;
 - ♦ *signal* – se houver *threads* bloqueados na *variável de condição* argumento, um deles é acordado; caso contrário, *nada acontece*.

Monitores - 5

- ♦ Para impedir a coexistência de dois *threads* dentro do *monitor*, é necessária uma regra que estipule como a contenção decorrente de *signal* é resolvida
- ♦ *monitor de Hoare* – o *thread* que invoca a operação de *signal* é colocado *fora do monitor* para que o *thread* acordado possa prosseguir;
 - ♦ muito geral, mas a sua implementação exige a existência de uma *stack*, onde são colocados os *threads* postos *fora do monitor* por invocação de *signal*;
- ♦ *monitor de Brinch Hansen* – o *thread* que invoca a operação de *signal* liberta imediatamente o *monitor* (*signal* é a última instrução executada);
 - ♦ é simples de implementar, mas pode tornar-se bastante restritivo porque só há possibilidade de execução de um *signal* em cada invocação de uma primitiva de acesso;
- ♦ *monitor de Lampson / Redell* – o *thread* que invoca a operação de *signal* prossegue a sua execução e o *thread* acordado mantém-se *fora do monitor*, competindo pelo acesso a ele;
 - ♦ é simples de implementar, mas pode originar situações em que alguns *threads* são colocados em *adiamento indefinido*.

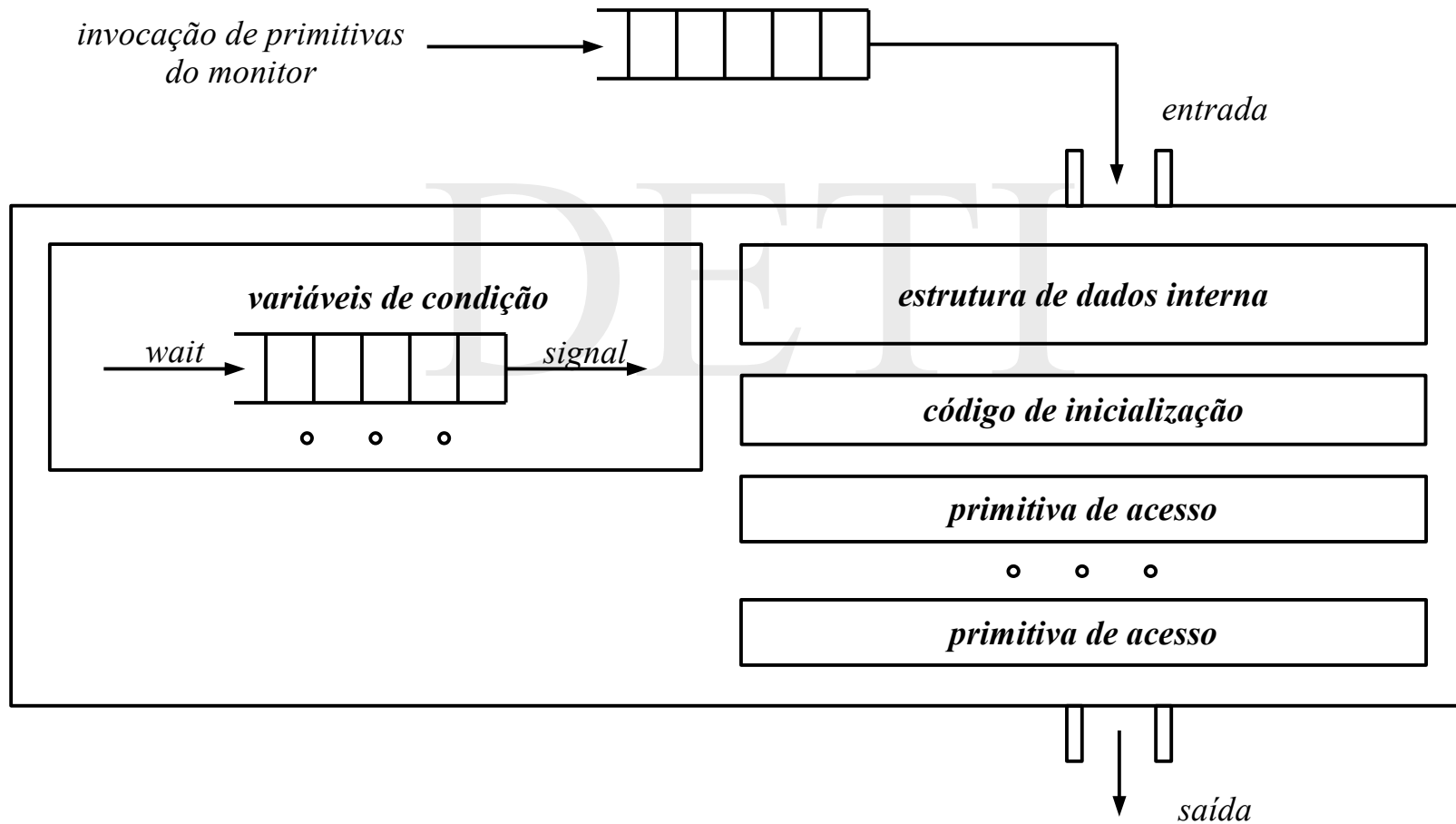
Monitores - 6

Monitor de Hoare



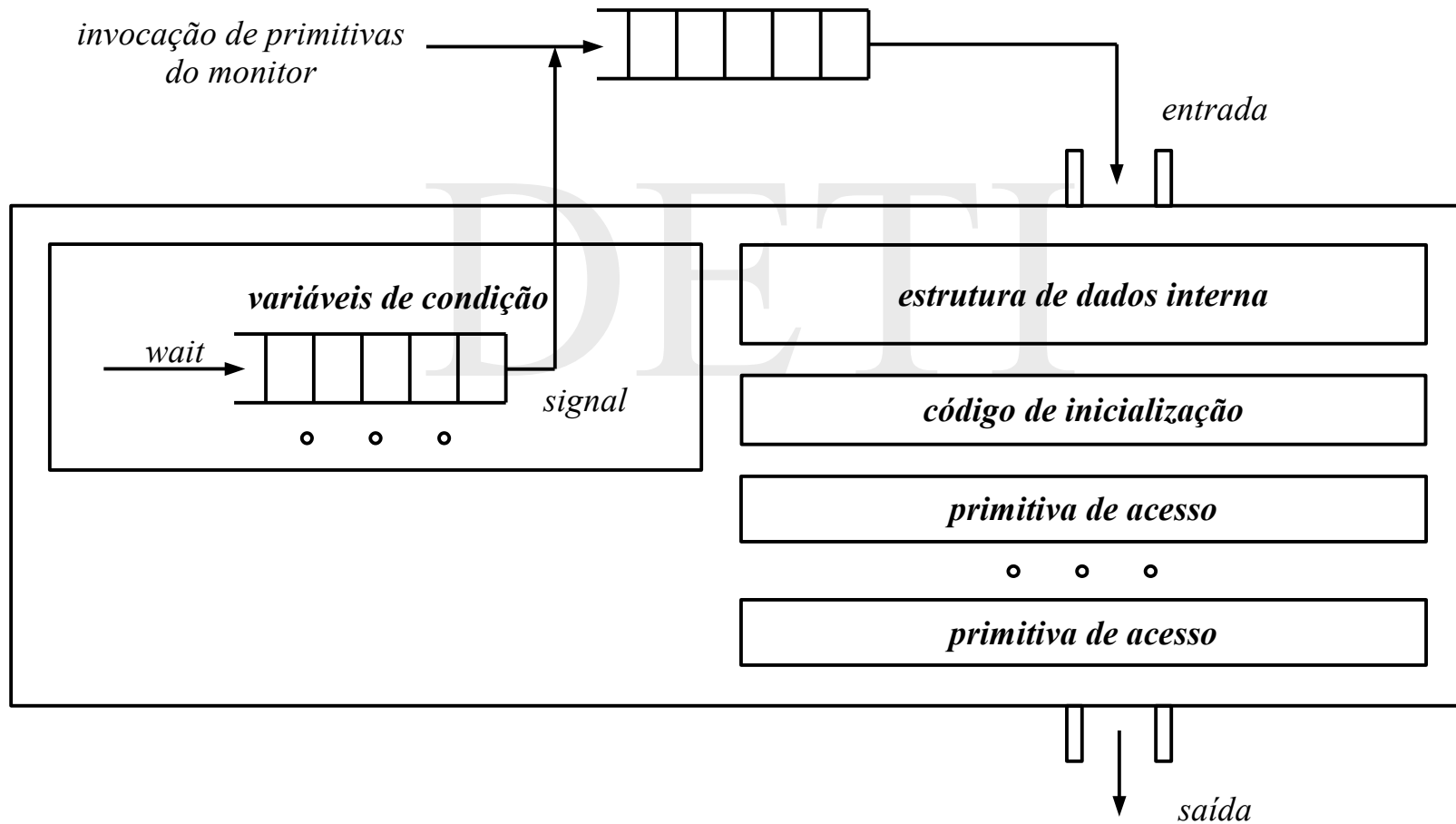
Monitores - 7

Monitor de Brinch Hansen



Monitores - 8

Monitor de Lampson / Redell



Problema dos produtores / consumidores (monitores) - 1

```
(* threads producers - p = 0, 1, ..., N-1 *)
procedure producer (p: integer);
  var
    val: DATA;
begin
  forever
    begin
      produce_data(val);
      transf.put(val); (* transfer data to ... *)
      do_something_else();
    end
end; (* producer *)

(* threads consumers - c = N, N+1, ..., N+M-1 *)
procedure consumer(c: integer);
  var
    val: DATA;
begin
  forever
    begin
      transf.get(val); (* retrieve data from ... *)
      consume_data(val);
      do_something_else();
    end
end; (* consumer *)
```

Problema dos produtores / consumidores (monitores) - 2

```
monitor transf;                                (* Hoare / Brinch Hansen approach *)

var
  fifo: FIFO;                                (* K-size FIFO *)
  ecnt: integer;                            (* n. of empty cells *)
  not_empty, not_full: condition;          (* synchronization condition variables *)

procedure put(val: DATA);
begin
  if ecnt = 0 then wait(not_full);            (* wait if FIFO is full *)
  fifo.insert(val);
  ecnp := ecnt - 1;
  signal(not_empty)                          (* signal that FIFO is not empty *)
end; (* put *)

procedure get(var val: DATA);
begin
  if ecnt = K then wait(not_empty);          (* wait if FIFO is empty *)
  fifo.retrieve(val);
  ecnt := ecnt + 1;
  signal(not_full);                          (* signal that FIFO is not full *)
end; (* get *)
begin
  ecnt := K;                                (* initial state *)
end;
end monitor; (* transf *)
```

Problema dos produtores / consumidores (monitores) - 2

```
monitor transf;                                (* Lampson / Redell approach *)

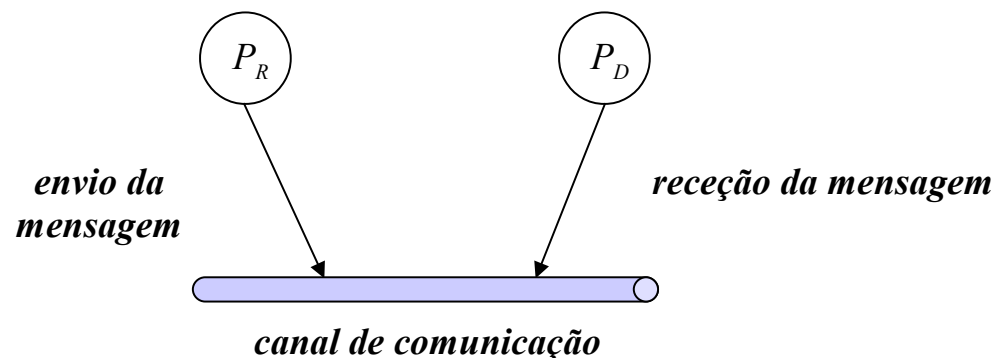
var
  fifo: FIFO;                                  (* K-size FIFO *)
  ecnt: integer;                             (* n. of empty cells *)
  not_empty, not_full: condition;          (* synchronization condition variables *)

procedure put(val: DATA);
begin
  while ecnt = 0 do wait(not_full);          (* wait if FIFO is full *)
  fifo.insert(val);
  ecnt := ecnt - 1;
  signal(not_empty)                          (* signal that FIFO is not empty *)
end; (* put *)

procedure get(var val: DATA);
begin
  while ecnt = K do wait(not_empty);        (* wait if FIFO is empty *)
  fifo.retrieve(val);
  ecnt := ecnt + 1;
  signal(not_full);                          (* signal that FIFO is not full *)
end; (* get *)
begin
  ecnt := K;                                (* initial state *)
end;
end monitor; (* transf *)
```

Passagem de mensagens - 1

- Uma forma alternativa de comunicação entre processos é através da *troca de mensagens*. Trata-se de um mecanismo absolutamente geral. Não exigindo partilha do espaço de endereçamento, a sua aplicação é igualmente válida, de um modo mais ou menos uniforme, tanto em ambientes monoprocessador, como em ambientes multiprocessador, ou de processamento distribuído.
- O princípio em que se baseia é muito simples:
 - sempre que um processo P_R , dito *remetente*, pretende comunicar com um processo P_D , dito *destinatário*, envia-lhe uma mensagem através de um canal de comunicação estabelecido entre ambos (*operação de envio*);
 - o processo P_D , para receber a mensagem, tem tão só que aceder ao canal de comunicação e aguardar a sua chegada (*operação de receção*).



Passagem de mensagens - 2

- ♦ A *troca de mensagens* só conduzirá, porém, a uma comunicação fiável entre os processos *remetente* e *destinatário*, se for garantida alguma forma de sincronização entre eles.
- ♦ O grau de sincronização existente pode ser classificado em dois níveis
 - ♦ *sincronização não bloqueante* - quando a sincronização é da responsabilidade dos processos intervenientes;
 - ♦ a *operação de envio* envia a mensagem e regressa sem qualquer informação sobre se a mensagem foi efetivamente recebida;
 - ♦ a *operação de receção*, por seu lado, regressa independentemente de ter sido ou não recebida uma mensagem;

```
/* operação de envio */
```

```
void msg_send_nb (unsigned int destid, MESSAGE msg);
```

```
/* operação de receção */
```

```
void msg_receive_nb (unsigned int srcid, MESSAGE *msg, bool *msg_arrival);
```

Passagem de mensagens - 3

- ♦ *sincronização bloqueante* - quando as operações de envio e de recepção contêm em si mesmas elementos de sincronização;
 - ♦ a *operação de envio* envia a mensagem e bloqueia até que esta seja efetivamente recebida;
 - ♦ a *operação de recepção*, por seu lado, só regressa quando uma mensagem tiver sido recebida;

/ operação de envio */*

void msg_send (unsigned int destid, MESSAGE msg);

/ operação de recepção */*

void msg_receive (unsigned int srcid, MESSAGE *msg);

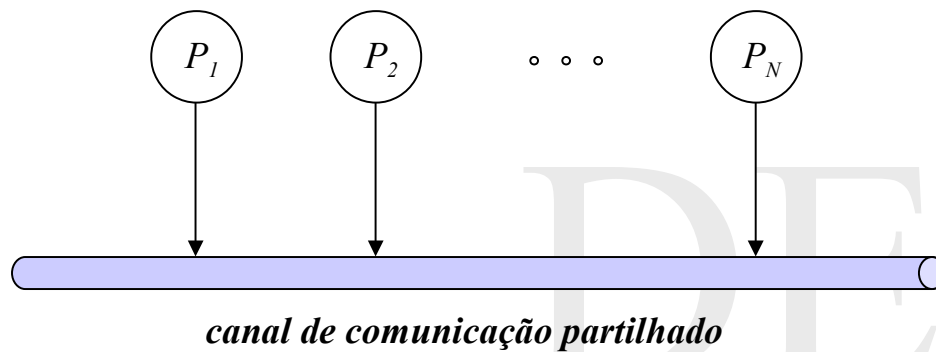
Passagem de mensagens - 4

- ♦ a *sincronização bloqueante* divide-se ainda em dois tipos
 - ♦ *rendez-vous* - quando, só após uma sincronização prévia entre os processos interlocutores, a transferência da mensagem tem efetivamente lugar;
 - ♦ não exige um armazenamento intercalar da mensagem e é típica de canais de comunicação dedicados (*ligações ponto a ponto*);
 - ♦ *remota* - quando a *operação de envio* envia a mensagem e bloqueia, aguardando confirmação de que a mensagem foi efetivamente recebida pelo destinatário;
 - ♦ pode existir ou não armazenamento intercalar da mensagem e é típica de canais de comunicação partilhados.

Passagem de mensagens - 5

- ♦ Para que a mensagem possa ser encaminhada entre o processo *remetente* e o processo *destinatário*, é fundamental que as operações de *envio* e de *receção* tenham um parâmetro que faça de algum modo referência à identidade do processo interlocutor.
 - ♦ Se essa referência é explícita, o endereçamento diz-se *direto*.
 - ♦ Caso contrário, o que é referenciado explicitamente é o canal de comunicação e o endereçamento diz-se *indireto*.
- ♦ Além disso, o canal de comunicação pode permitir o armazenamento intercalar da mensagem, tipicamente em estruturas de dados partilhadas que implementam filas de espera onde a ordem cronológica de chegada é em princípio respeitada – as chamadas *caixas de correio*.

Passagem de mensagens - 6



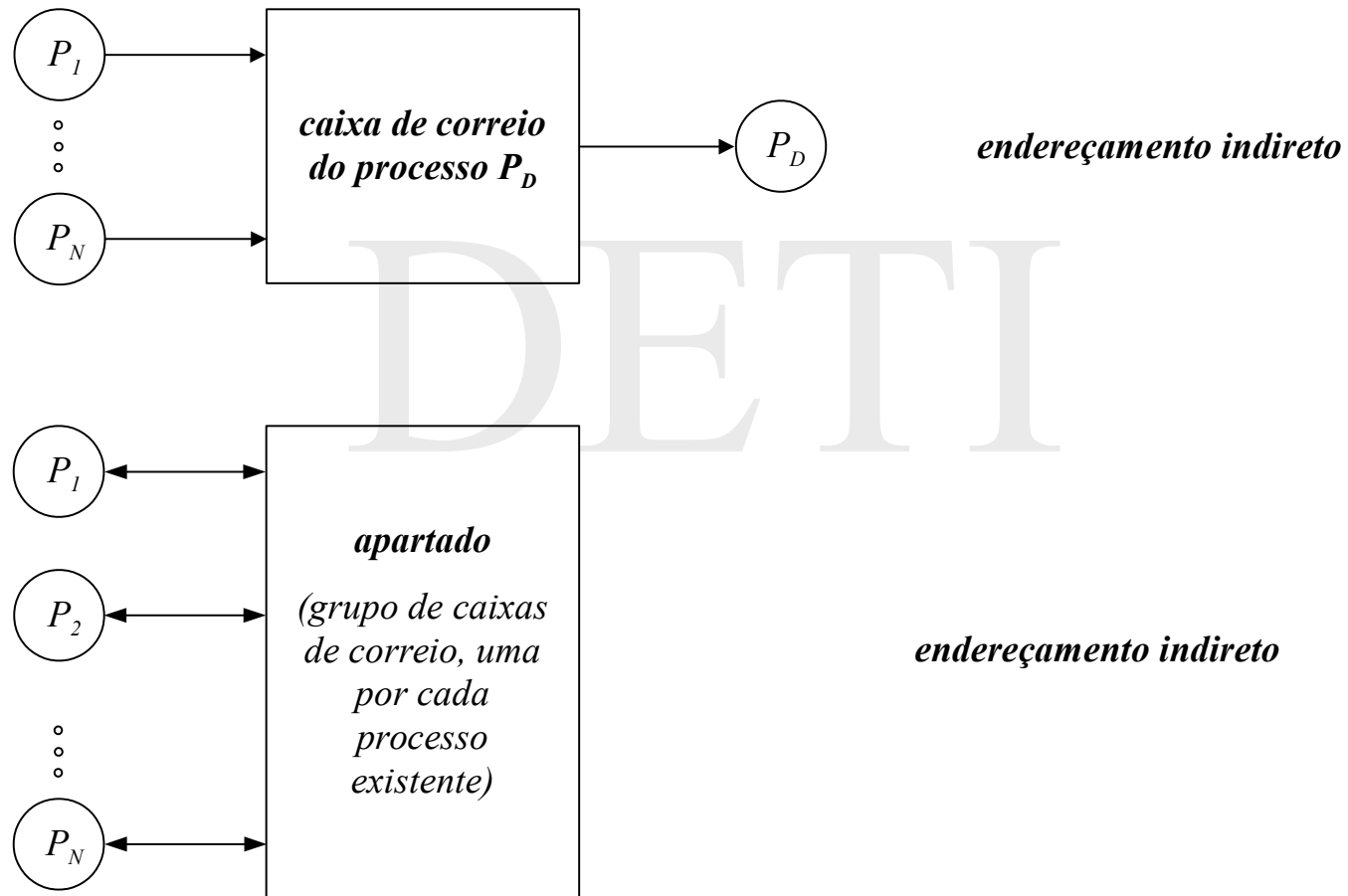
endereçamento direto – se baseado no *id* do processo

endereçamento indireto – se baseado no *porto* de acesso



endereçamento indireto

Passagem de mensagens - 7



Passagem de mensagens - 8

Formato da Mensagem

cabeçalho

- identificação do remetente
- identificação do destinatário
- tipo
- informação de controlo
- n.º da mensagem
- comprimento do conteúdo informativo
- ...

conteúdo informativo

- comprimento fixo ou variável
(geralmente estruturado segundo um tipo de dados definido pelos processos envolvidos na comunicação)

Problema dos produtores / consumidores (pass. de mens.) - 1

```
/* supõe-se existir uma caixa de correio, com capacidade para K mensagens,
   que foi previamente criada e que é acessível a todos os processos
   produtores (como remetentes) e consumidores (como destinatários)
*/

static unsigned int com;    /* identificador da caixa de correio */
/* estrutura de dados da mensagem trocada */
typedef struct
{
    DATA info;    /* só tem conteúdo informativo */
} MESSAGE;

/* processos produtores
   * p = 0, 1, ..., N-1 */
void produtor(unsigned int p)
{
    MESSAGE msg;
    forever
    {
        produce_data(&msg.info);
        /* envia a mensagem e regressa imediatamente; se cheia bloqueia */
        msg_send_nb(com, msg);
        do_something_else();
    }
}
```

Problema dos produtores / consumidores (pass. de mens.) - 2

```
/* supõe-se existir uma caixa de correio, com capacidade para K mensagens,  
   que foi previamente criada e que é acessível a todos os processos  
   produtores (como remetentes) e consumidores (como destinatários)  
*/
```

```
static unsigned int com;    /* identificador da caixa de correio */
```

```
/* estrutura de dados da mensagem trocada */
```

```
typedef struct
```

```
{  
    DATA info;    /* só tem conteúdo informativo */  
} MESSAGE;
```

```
/* processos consumidores - c = N, N+1, ..., N+M-1 */
```

```
void consumer(unsigned int c)
```

```
{
```

```
    MESSAGE msg;
```

```
    forever
```

```
    {
```

```
        /* aguarda a chegada de uma mensagem */
```

```
        msg_receive(com, &msg);
```

```
        consume_data(msg.info);
```

```
        do_something_else();
```

```
    }
```

```
}
```

Threads e monitores em Unix - 1

- O standard *POSIX, IEEE 1003.1c*, define um interface de programação de aplicações (*API*) para criação e sincronização de *threads*. Os sistemas de operação da família Unix fornecem habitualmente uma biblioteca que o implementa – esta biblioteca designa-se *pthread*.
- A sua utilização reveste-se de grande importância, já que permite a construção de *monitores* em Linguagem C. Como o C não é uma linguagem concorrente, o conceito de *monitor* não é suportado diretamente pela linguagem, mas pode ser implementado a partir de *mutexes* e *variáveis de condição* fornecidos pela biblioteca. Note-se que os monitores construídos são de tipo Lampson / Redell.
- Quando se faz a *linkagem* de programas com módulos que usam funções desta biblioteca, é necessário fazer uma referência explícita a ela no comando que processa a *linkagem*

```
gcc -lpthread ...
```

Threads e monitores em Unix - 2

- Entre as funções fornecidas, destacam-se
 - *pthread_create* – lançamento de um *thread*, a execução da função passada em argumento é objecto de execução separada;
 - *pthread_exit* – equivalente a *exit* no caso de um processo;
 - *pthread_once*, *pthread_mutex_**, *pthread_cond_** – necessárias para a construção de *monitores*;
 - *pthread_join* – equivalente a *waitpid*
 - *pthread_self* – equivalente a *getpid* no caso de um processo.

Semáforos em Unix

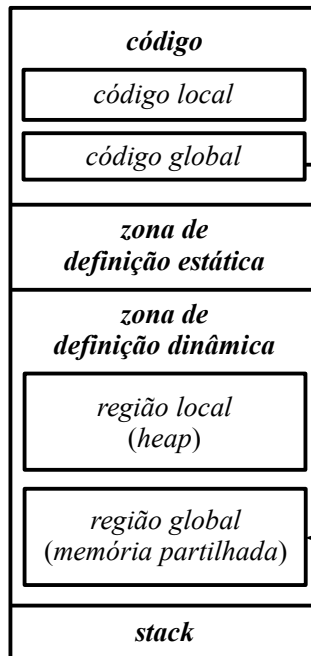
- ♦ A implementação de semáforos em Unix ultrapassa em muitos aspetos a definição original de Dijkstra. As principais diferenças são
 - ♦ *tratamento em bloco de grupos de semáforos* – é possível criar, destruir e manipular atomicamente mais do que um semáforo de cada vez;
 - ♦ *operações básicas diferentes* – as chamadas ao sistema são *semget*, *semop* e *semctl*
 - ♦ além da sincronização convencional (os processos bloqueiam se o campo *val* for zero) é também possível bloquear processos se o campo *val* não for zero;
 - ♦ as operações de *down* e *up* podem ser também realizadas com decrementos e incrementos do campo *val* diferentes de um

Memória partilhada em Unix - 1

- ♦ Uma *região de memória partilhada* é estabelecida no espaço de endereçamento do processo apenas em *runtime*.
- ♦ Está localizada na
 - ♦ *zona de definição dinâmica* – quando representa partilha de dados, o acesso à informação aí armazenada é feito tipicamente pela dereferenciação de um ponteiro para uma estrutura de dados cujos campos constituem os diferentes componentes de interesse;
 - ♦ *zona de código* – quando representa partilha de bibliotecas de sistema, o acesso à informação aí armazenada é feito tipicamente pela dereferenciação de um ponteiro para uma estrutura de dados cujos campos constituem ponteiros para funções específicas.

Memória partilhada em Unix - 2

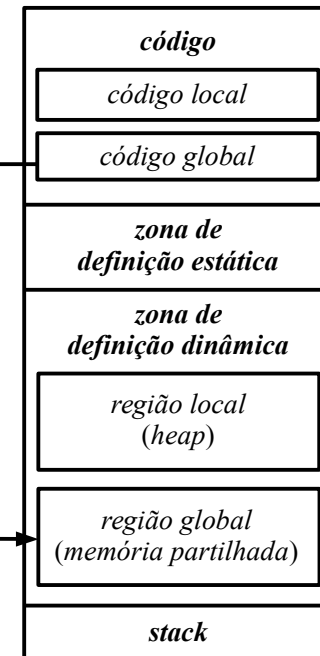
*espaço de endereçamento
do processo A*



*área isolada em
memória principal
(código)*

*área isolada em
memória principal
(dados)*

*espaço de endereçamento
do processo B*



Memória partilhada em Unix - 3

- O estabelecimento de uma *região de memória partilhada* no espaço de endereçamento de um processo exige as operações seguintes
- *criação* ou *ligação* a uma região previamente criada – é aí definido o tipo de acesso *read/write* ou *readonly*;
- *anexação* da região ao espaço de endereçamento do processo – obtém-se aqui um ponteiro para a sua localização;
- *desanexação* da região do espaço de endereçamento do processo – operação complementar da anterior;
- *destruição* de uma região existente – operação complementar da criação.
- Estas operações são construídas a partir das chamadas ao sistema *shmget*, *shmat*, *shmdt* e *shmctl*.

Mensagens em Unix - 1

- A implementação de mensagens em Unix é bastante restritiva. O modelo assume a existência de filas de mensagens onde são armazenadas mensagens de diferentes tipos, descritos por números inteiros positivos.
- O formato de cada mensagem é

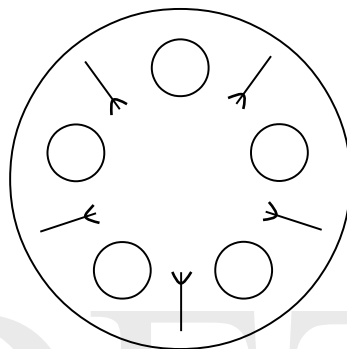
```
struct msgbuf
{ unsigned long mtype;          /* tipo da mensagem, > 0 */
  DATA info;                  /* conteúdo informativo */
}
```

- Um aspecto a ter em conta no estabelecimento dos argumentos das primitivas de *envio* e *receção* de mensagens é que se passa um ponteiro para uma variável de tipo **struct msgbuf**, que representa a *mensagem*, mas se indica como seu comprimento apenas o comprimento do conteúdo informativo. Isto significa que é possível enviar e receber mensagens de comprimento nulo (mensagens de sincronização)

Mensagens em Unix - 2

- O *envio* de mensagens é, em princípio, não bloqueante. A primitiva só bloqueia se não houver espaço suficiente na fila para armazenamento da mensagem. A *receção* pode ser bloqueante, ou não bloqueante, e permite alternativamente recolher a primeira mensagem de um dado tipo, de qualquer tipo, de um tipo diferente do tipo especificado ou de um tipo menor ou igual ao tipo especificado, existente na fila.
- Estas operações são construídas a partir das *chamadas ao sistema* *msgget*, *msgsnd*, *msgrcv* e *msgctl*.
- Uma fila de mensagens pode ser vista como um apartado, sendo o tipo é usado para identificação do processo .

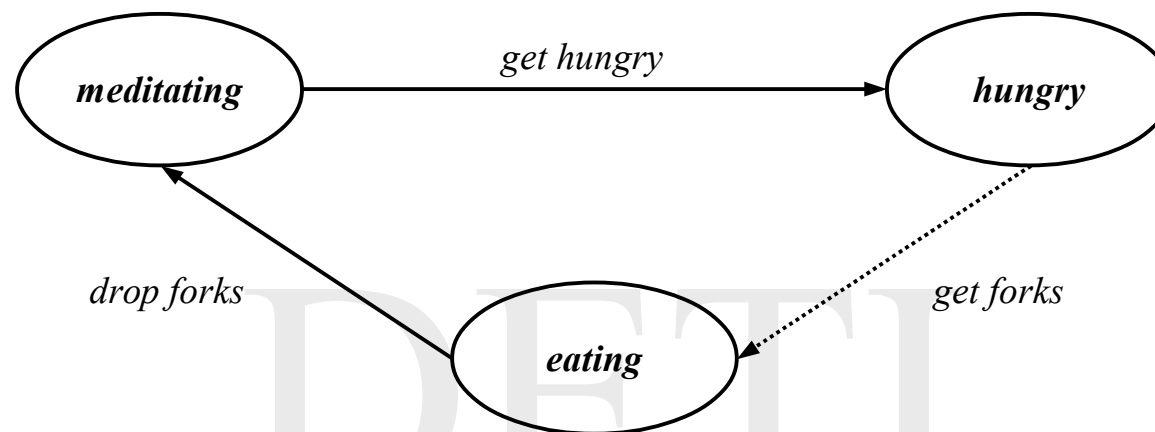
Jantar dos filósofos - 1



Formulação do problema

- 5 filósofos estão sentados à volta de uma mesa. Cada filósofo tem um prato de esparguete à frente e um garfo de cada lado, como é ilustrado na figura. Os filósofos alternam períodos em que meditam, com períodos em que comem. Para comer, um filósofo tem primeiro que pegar nos garfos colocados à sua esquerda e à sua direita. Um só não é suficiente porque o esparguete é muito escorregadio.
- Escrever um programa que modele a situação e que permita a todos os filósofos presentes proceder como indicado. Nomeadamente, é preciso garantir que não há o risco de qualquer deles passar períodos infinitos de tempo sem conseguir comer, o que conduziria inevitavelmente à sua morte.

Jantar dos filósofos - 2



- a solução apresentada é equivalente à solução original de Dijkstra;
- supõe que cada filósofo, quando pretende os garfos, pega em simultâneo nos dois; o que significa que se ambos não estiverem disponíveis, o filósofo aguarda por eles de mãos vazias (estado *hungry*);
- não se trata, contudo, de uma solução geral (o *adiamento indefinido* não está completamente resolvido).

Jantar dos filósofos - 3

Solução implementada

```
enum {MEDITATING, HUNGRY, WAITING, EATING};
```

```
typedef struct TablePlace  
{  
    int state;  
} TablePlace;
```

```
typedef struct Table  
{  
    int nplaces;  
    TablePlace place[0];  
} Table;
```

```
int set_table(unsigned int n, FILE *logp);
```

```
int get_hungry(unsigned int f);
```

```
int get_forks(unsigned int f);
```

```
int drop_forks(unsigned int f);
```


Sinais em Unix - 1

- Um *sinal* constitui uma interrupção produzida no contexto de um processo onde lhe é comunicada a ocorrência de um acontecimento especial.
- Pode ser despoletado pelo *kernel*, em resultado de situações de erro ao nível hardware ou de condições específicas ao nível software, pelo próprio processo, por outro processo, ou pelo utilizador através do dispositivo *standard* de entrada.
- Tal como o processador no tratamento de exceções, o processo assume uma de três atitudes possíveis relativamente a um sinal
 - *ignorá-lo* – não fazer nada face à sua ocorrência
 - *bloqueá-lo* – impedir que interrompa o processo durante intervalos de processamento bem definidos;
 - *executar uma acção associada* – pode ser a acção estabelecida por defeito quando o processo é criado (conduz habitualmente à sua terminação ou suspensão de execução), ou uma acção específica que é introduzida (*registada*) pelo próprio processo em *runtime*.

Sinais em Unix - 2

- ♦ Existem em Unix um conjunto variado de *chamadas ao sistema* que possibilitam o processamento de sinais.
- ♦ Destacam-se entre elas
 - ♦ *kill* – um sinal é enviado a um processo ou grupo de processos
 - ♦ *raise* – um sinal é enviado ao próprio processo
 - ♦ *sigaction* – registo de um função para serviço a um sinal específico;
 - ♦ *sigprocmask* – manipulação da máscara que inibe ou ativa o processamento de sinais específicos
 - ♦ *sigpending* – determinação dos sinais pendentes
 - ♦ *sigsuspend* – suspensão da execução do processo até à chegada de um sinal.

Tabela dos sinais mais comuns (standard Posix.1)

| <i>Sinal</i> | <i>Valor</i> | <i>Acção</i> | <i>Causa</i> |
|--------------|--------------|--------------|---|
| SIGHUP | 1 | Term | Hangup detected on controlling terminal or death of controlling process |
| SIGINT | 2 | Term | Interrupt from keyboard |
| SIGQUIT | 3 | Core | Quit from keyboard |
| SIGILL | 4 | Core | Illegal Instruction |
| SIGABRT | 6 | Core | Abort signal from abort (3) |
| SIGFPE | 8 | Core | Floating point exception |
| SIGKILL | 9 | Term | Kill signal |
| SIGSEGV | 11 | Core | Invalid memory reference |
| SIGPIPE | 13 | Term | Broken pipe: write to pipe with no readers |
| SIGALRM | 14 | Term | Timer signal from alarm (2) |
| SIGTERM | 15 | Term | Termination signal |
| SIGUSR1 | 10 | Term | User-defined signal 1 |
| SIGUSR2 | 12 | Term | User-defined signal 2 |
| SIGCHLD | 17 | Ign | Child stopped or terminated |
| SIGCONT | 18 | | Continue if stopped |
| SIGSTOP | 19 | Stop | Stop process |
| SIGTSTP | 20 | Stop | Stop typed at tty |
| SIGTTIN | 21 | Stop | tty input for background process |
| SIGTTOU | 22 | Stop | tty output for background process |

NOTA – The signals SIGKILL and SIGSTOP cannot be caught, blocked, or ignored.

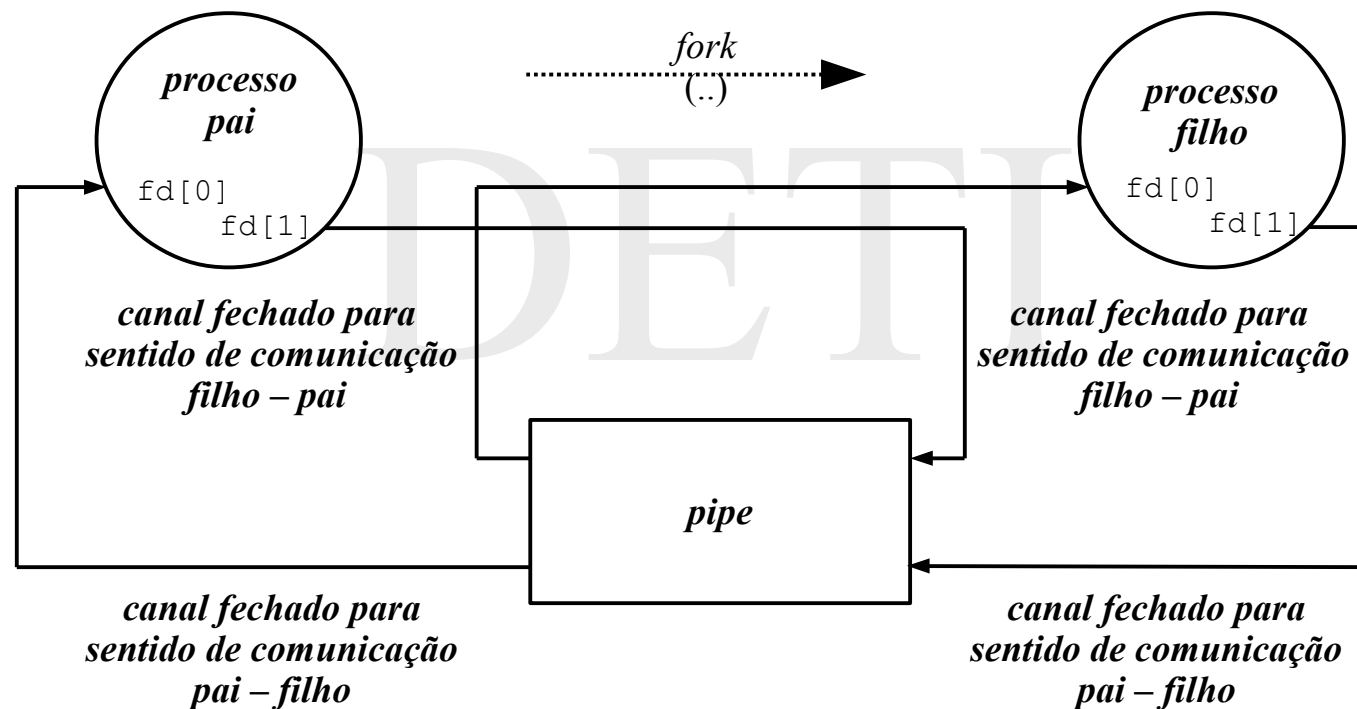
Exemplo de processamento de um sinal - 1

```
...  
  
int main(int argc, char* argv[])  
{  
    ...  
  
    /* installing interrupt handler for ^C */  
    struct sigaction sigact;  
    struct sigaction prev_sigact;  
    sigact.sa_handler = interruptHandler;  
    sigact.sa_flags = 0;  
    if (sigaction(SIGINT, &sigact, &prev_sigact) == -1)  
    {  
        perror("Fail installing interrupt handler");  
        return EXIT_FAILURE;  
    }  
    ...  
}  
  
static void interruptHandler(int signum)  
{  
    printf("\nPlease, let me finish\n");  
}
```

Pipes em Unix - 1

- Um *pipe* é um canal de comunicação elementar que é estabelecido entre dois ou mais processos. O mecanismo de *piping* constitui a forma tradicional de comunicação entre processos em Unix, tendo sido inclusivamente introduzido por ele.
- Apresenta, contudo, algumas limitações
 - *o canal de comunicação é half-duplex* – só permite comunicação num sentido;
 - *só pode ser usado entre processos que estão relacionados entre si* – um *pipe* é criado por um processo, que a seguir se duplica uma ou mais vezes e a comunicação é então estabelecida entre o *pai* e o *filho*, ou entre dois *filhos*.
 - A sua principal utilização é na composição de comandos complexos a partir de uma organização em cadeia de comandos mais simples em que o *standard output* de um dado comando é redireccionado para a entrada de um *pipe* e o *standard input* do comando seguinte é redireccionado para a saída do mesmo *pipe*.

Pipes em Unix - 2



Exemplo de comunicação pai – filho - 1

```
int main(void)
{
    /* create a pipe */
    int fd[2];
    if (pipe(fd) == -1)
    {
        perror("Fail creating a pipe: ");
        abort();
    }

    /* forking current process */
    switch (fork())
    {
        case -1: /* in this case fork fails */
            perror("Fail forking: ");
            Abort();

        case 0: /* this case is only executed by the child process */
            /* redirect, close, and launch program */

        default: /* this case is only executed by the parent process */
            /* redirect, close, and launch program */
    }

    /* never get here, parent or child */
    return 0;
}
```

Exemplo de comunicação pai – filho - 2

```
/* child side */

case 0:    /* this case is only executed by the child process */

    /* redirect stdout to pipe */
    if (dup2(fd[1], STDOUT_FILENO) == -1)
    {
        perror("Fail redirecting stdout: ");
        abort();
    }

    /* close unnecessary file descriptors; it must be done. */
    close(fd[0]);
    close(fd[1]);

    /* substitute process program */
    execl("/bin/ls", "ls", "-l", NULL);
    perror("Fail in exec (ls): ");
    abort();
```


Exemplo de comunicação pai – filho - 3

```
/* parent side */

default: /* this case is only executed by the parent process */
/* redirect stdin to pipe */
if (dup2(fd[0], STDIN_FILENO) == -1)
{
    perror("Falhou o redirecting stdin: ");
    abort();
}

/* close unnecessary file descriptors; it must be done. */
close(fd[0]);
close(fd[1]);

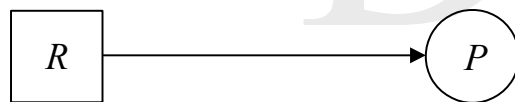
/* substitute process program */
execl("/usr/bin/sort", "sort", "-n", "-k", "5", NULL);
perror("Fail in exec (sort): ");
abort();
```

Deadlock

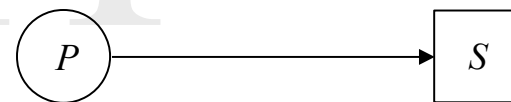
- ♦ Genericamente, um *recurso* é algo que um processo precisa para a sua execução. Pode ser
 - ♦ *componente físicos do sistema computacional* (processadores, regiões de memória principal ou de memória de massa, dispositivos concretos de entrada / saída, etc)
 - ♦ *estrutura de dados comuns* definidas ao nível do sistema de operação (tabela de controlo de processos, canais de comunicação, etc), ou entre processos de uma mesma aplicação.
- ♦ Uma propriedade essencial dos recursos é o tipo de apropriação que os processos fazem deles. Nestes termos, os recursos dividem-se em
 - ♦ *recursos preemptable* – quando podem ser retirados aos processos que os detêm, sem que daí resulte qualquer consequência irreparável à boa execução dos processos; são, por exemplo, em ambientes multiprogramados, o processador, ou as regiões de memória principal onde o espaço de endereçamento de um processo está alojado;
 - ♦ *recursos non-preemptable* – em caso contrário; são, por exemplo, a impressora, ou uma estrutura de dados partilhada que exige exclusão mútua para a sua manipulação.

Caracterização esquemática de deadlock

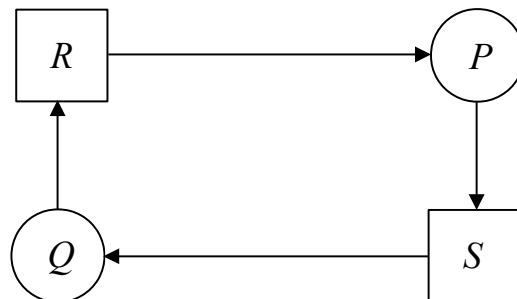
- Numa situação de *deadlock*, só os recursos *non-preemptable* são relevantes. Os restantes podem ser sempre retirados, se tal for necessário, ao(s) processo(s) que o(s) detêm e atribuídos a outros para garantir o prosseguimento da execução destes últimos.
- Assim, usando este tipo de classificação, torna-se possível desenvolver uma notação esquemática que representa graficamente situações de *deadlock*.



processo P mantém o recurso R na sua posse



processo P requer o recurso S

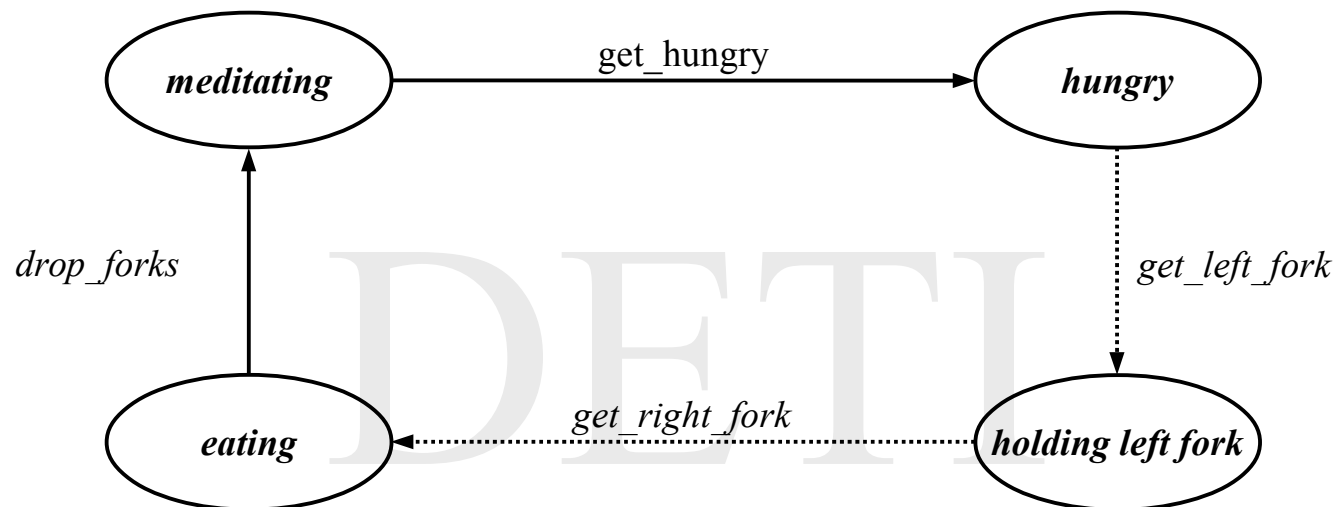


*situação típica de deadlock
(a mais simples possível)*

Condições necessárias à ocorrência de deadlock - 1

- ♦ Pode demonstrar-se que, sempre que ocorre *deadlock*, há quatro condições que ocorrem necessariamente. São elas
 - ♦ *condição de exclusão mútua* – cada recurso existente, ou está livre, ou foi atribuído a um e um só processo (a sua posse não pode ser partilhada);
 - ♦ *condição de espera com retenção* – cada processo, ao requerer um novo recurso, mantém na sua posse todos os recursos anteriormente solicitados;
 - ♦ *condição de não libertação* - ninguém, a não ser o próprio processo, pode decidir da libertação de um recurso que lhe tenha sido previamente atribuído;
 - ♦ *condição de espera circular* (ou *ciclo vicioso*) - formou-se uma cadeia circular de processos e recursos, em que cada processo requer um recurso que está na posse do processo seguinte na cadeia.

Condições necessárias à ocorrência de deadlock - 2



- ♦ a solução apresentada é a solução intuitiva;
- ♦ supõe que cada filósofo, quando pretende os garfos, pega primeiro no garfo da esquerda e, só depois, no da direita; o que significa que se algum não estiver disponível, o filósofo aguarda por ele de mãos vazias (estado *hungry*), ou segurando o garfo da esquerda (estado *holding_left_fork*).

Condições necessárias à ocorrência de deadlock - 3

Solução implementada

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

```
typedef struct TablePlace  
{  
    int state;  
} TablePlace;
```

```
typedef struct Table  
{  
    Int semid;  
    int nplaces;  
    TablePlace place[0];  
} Table;
```

```
int set_table(unsigned int n, FILE *logp);  
int get_hungry(unsigned int f);  
int get_left_fork(unsigned int f);  
int get_right_fork(unsigned int f);  
int drop_forks(unsigned int f);
```

♦ Let's look at the code!

Condições necessárias à ocorrência de deadlock - 4

Análise Crítica

- ♦ A solução anterior funciona bem quase sempre! Contudo, se houver mudança de contexto de todos os processos filósofo entre o pegar no garfo da esquerda e pegar no da direita, o sistema entrará em deadlock.
- ♦ Todas as condições referidas anteriormente são verificadas
 - ♦ *exclusão mútua* – cada garfo está na posse de um só filósofo de cada vez;
 - ♦ *espera com retenção* – cada filósofo, ao tentar pegar no garfo da direita, mantém na sua posse o garfo da esquerda;
 - ♦ *não liberação* – desde que tome posse de um garfo, cada filósofo conserva-o até ter acabado de comer;
 - ♦ *espera circular* (ou *ciclo vicioso*) – formou-se uma cadeia circular de filósofos e garfos em que cada filósofo conserva o seu garfo da esquerda enquanto espera tomar posse do da direita.

Prevenção de deadlock no sentido estrito - 1

- ♦ As condições necessárias à ocorrência de *deadlock* conduzem à proposição
há deadlock \Rightarrow *há exclusão mútua no acesso a um recurso* **and**
há espera com retenção **and**
há não libertação de recursos **and**
há espera circular
- ♦ que é equivalente a
não há exclusão mútua no acesso a um recurso **or**
não há espera com retenção **or**
há libertação de recursos **or**
não há espera circular \Rightarrow *não há deadlock* .
- ♦ Assim, desde que uma das condições necessárias à ocorrência de *deadlock* seja negada pelo algoritmo de acesso aos recursos, o *deadlock* torna-se impossível.
- ♦ Políticas com esta característica designam-se de *políticas de prevenção de deadlock no sentido estrito* (*deadlock prevention*, em inglês).

Prevenção de deadlock no sentido estrito - 2

- ♦ A primeira delas, *há exclusão mútua no acesso a um recurso*, é bastante restritiva porque só pode ser negada tratando-se de um recurso passível de partilha em simultâneo. Caso contrário, são introduzidas *condições de corrida* que conduzem, ou podem conduzir, a inconsistência de informação.
- ♦ O acesso para leitura por parte de múltiplos processos a um dado ficheiro é um exemplo típico da negação desta condição. Note-se que, neste caso, é comum permitir também um acesso para escrita por parte de um processo de cada vez.
 - ♦ Quando tal acontece, porém, não se pode impedir completamente a existência de *condições de corrida*, com a consequente inconsistência de informação. *Porquê?*
- ♦ É, por isso, que só as três últimas condições são em geral objeto de negação.

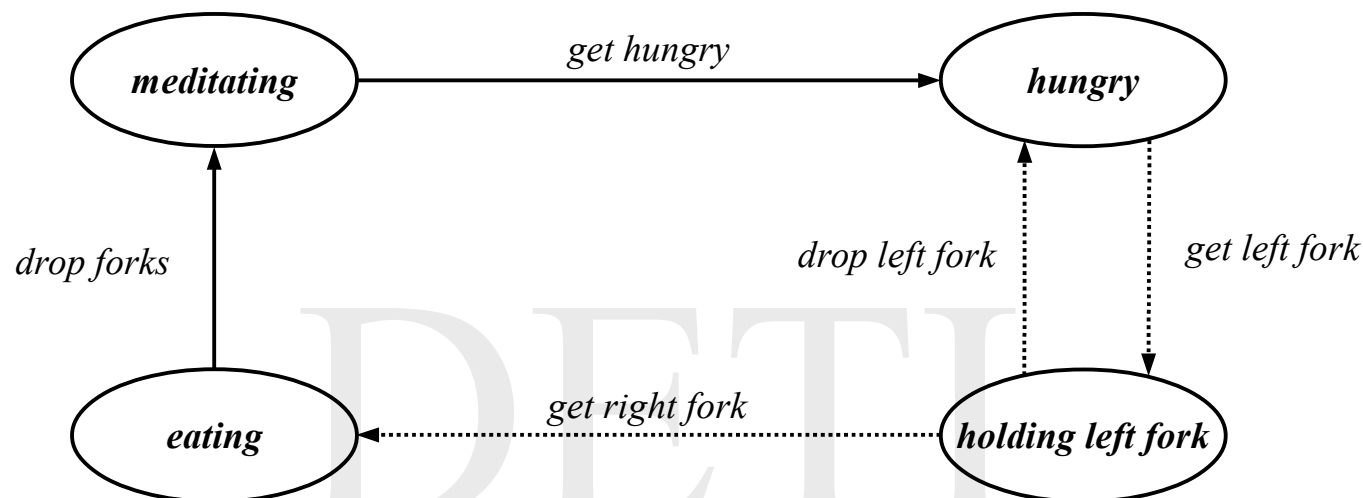
Negando a condição de espera com retenção - 1

- ♦ Significa que um processo *tem que solicitar de um só vez todos os recursos que vai precisar para a sua continuação*.
 - ♦ Se os obtém, o completamento da acção associada está garantido.
 - ♦ Se não os obtém, terá que aguardar.
- ♦ Note-se que a ocorrência de *adiamento indefinido* não está impedida.
 - ♦ A solução deve garantir para tal que os recursos necessários serão mais tarde ou mais cedo atribuídos ao processo.
 - ♦ A introdução de mecanismos de *aging* é uma solução muito usada nestas situações.
- ♦ No jantar dos filósofos, ao considerar-se os dois garfos como um único recurso, está-se a negar esta condição.
 - ♦ A solução de Dijkstra, apresentada em aulas anteriores, segue esta estratégia

Impondo a condição de libertação de recursos - 1

- ♦ Significa que um processo *quando não obtém os recursos que necessita para a sua continuação, tem que libertar todos os recursos na sua posse, tentando mais tarde a sua obtenção a partir do princípio.*
- ♦ Um cuidado a ter numa solução deste tipo é que o processo não entre em *busy waiting*. O processo deve bloquear e ser mais tarde acordado quando houver libertação de recursos.
- ♦ Note-se que a ocorrência de *adiamento indefinido* não está impedida.
 - ♦ A solução deve garantir para tal que os recursos necessários serão mais tarde ou mais cedo atribuídos ao processo.
 - ♦ A introdução de mecanismos de *aging* é uma solução muito usada nestas situações.
- ♦ No jantar dos filósofos, se um filósofo falha na tentativa de obter o garfo da direita deve pousar o da esquerda

Impondo a condição de libertação de recursos - 2



- cada filósofo, quando pretende os garfos, continua a pegar primeiro no garfo da esquerda e, só depois, no da direita, mas agora se o garfo da direita não estiver livre, o da esquerda é obrigatoriamente pousado;
- o *busy waiting* é evitado se o filósofo aguardar pela libertação do garfo da direita, após ter pousado o da esquerda;
- não se trata, contudo, de uma solução geral (o *adiamento indefinido* não está completamente resolvido).

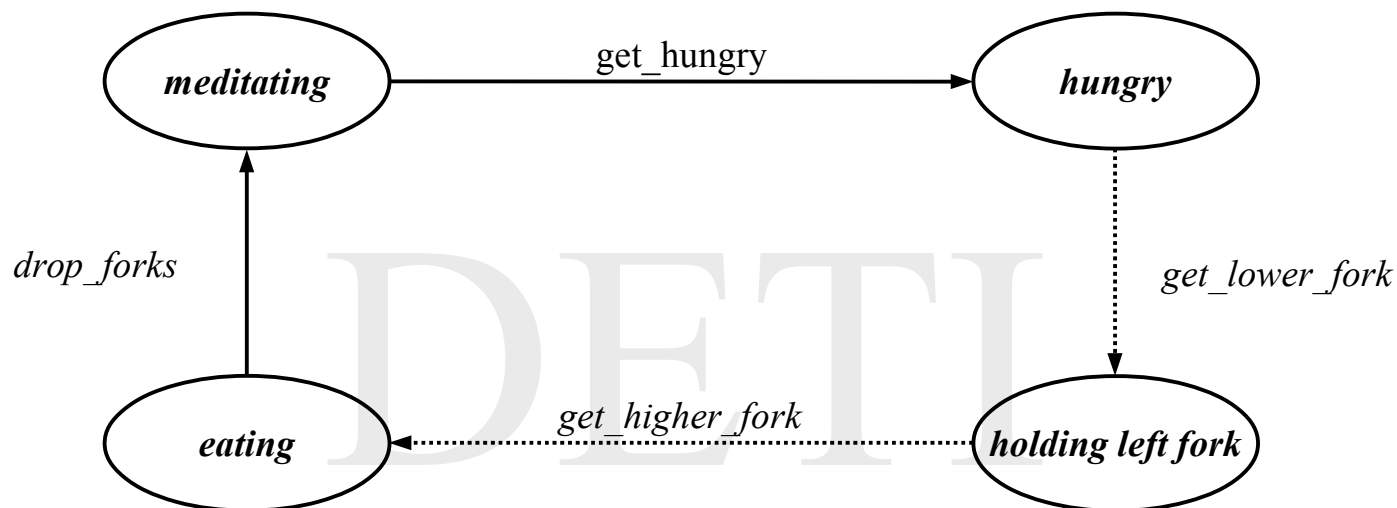
Negando a condição de espera circular - 1

- ♦ Significa *estabelecer uma ordenação linear dos recursos* e impor que um processo *quando procura obter os recursos que necessita para a sua continuação, o faça sempre por ordem crescente (decrescente) do número associado a cada um.*
- ♦ Desta maneira, a possibilidade de formação de uma cadeia circular de processos e recursos está posta de parte.
- ♦ Note-se que a ocorrência de *adiamento indefinido* não está impedida.
 - ♦ A solução deve garantir para tal que os recursos necessários serão mais tarde ou mais cedo atribuídos ao processo.
 - ♦ A introdução de mecanismos de *aging* é uma solução muito usada nestas situações.
- ♦ No jantar dos filósofos, um dos filósofos pega nos garfos pela ordem contrária aos outros.

Negando a condição de espera circular - 2

- ♦ Face ao modo utilizado para identificação dos filósofos e dos garfos, é fácil constatar que
 - ♦ os filósofos, 0 a N-2, ao pegarem primeiro no garfo da esquerda e, só depois, no garfo da direita, solicitam os garfos (recursos) por ordem crescente do seu número de ordem;
 - ♦ com o filósofo N-1 passa-se exatamente o contrário: o garfo da esquerda tem o número de ordem N-1 e o da direita o número de ordem 0.
- ♦ Assim, a imposição de uma ordenação crescente na atribuição de recursos exige apenas que o filósofo N-1 pegue nos garfos pela ordem inversa à dos outros: primeiro o da direita e, só depois, o da esquerda!

Negando a condição de espera circular - 3



- cada filósofo, quando pretende os garfos, pegar primeiro no garfo com id mais baixo e, só depois, no com id mais alto.
- não se trata, contudo, de uma solução geral (o *adiamento indefinido* não está completamente resolvido).

Prevenção de deadlock no sentido estrito - 3

Análise Crítica

- ♦ As políticas de *prevenção de deadlock no sentido estrito*, embora seguras, são muito restritivas, pouco eficientes e difíceis de aplicar em situações muito gerais.

Resumindo, tem-se que

- ♦ *negação da condição de exclusão mútua* – só pode ser aplicada a recursos passíveis de partilha em simultâneo;
- ♦ *negação da condição de espera com retenção* – exige o conhecimento prévio de todos os recursos que vão ser necessários e considera sempre o pior caso possível (uso de todos os recursos em simultâneo);
- ♦ *imposição da condição de não libertação* – ao supor a libertação de todos os recursos anteriores quando o próximo não puder ser atribuído, atrasa a execução do processo de modo muitas vezes substancial;
- ♦ *negação da condição de espera circular* – desaproveita recursos eventualmente disponíveis que poderiam ser usados na continuação do processo.

Prevenção de deadlock no sentido lato - 1

- Uma alternativa menos restritiva do que a *prevenção de deadlock no sentido estrito* é não negar *a priori* qualquer das condições necessárias à ocorrência de *deadlock*, mas monitorar continuamente o estado interno do sistema de modo a garantir que a sua evolução se faz apenas entre estados ditos *seguros*.
- Define-se-se neste contexto *estado seguro* como uma qualquer distribuição dos recursos do sistema, livres ou atribuídos aos processos que coexistem, que possibilita a terminação de todos eles.
 - Por oposição, um estado é *inseguro* se não for possível fazer-se uma tal afirmação sobre ele.
- Políticas com esta característica designam-se de *políticas de prevenção de deadlock no sentido lato* (*deadlock avoidance*, em inglês).
- Convém notar o seguinte
 - *é necessário o conhecimento completo de todos os recursos do sistema e cada processo tem que indicar à cabeça a lista de todos os recursos que vai precisar* – só assim se pode caracterizar um estado seguro;
 - *um estado inseguro não é sinónimo de deadlock* – vai, contudo, considerar-se sempre o pior caso possível para garantir a sua não ocorrência.

Prevenção de deadlock no sentido lato - 2

- ♦ Definindo

NTR_i – n.º total de recursos do tipo i (com $i = 0, 1, \dots, N-1$)

$R_{i,j}$ – n.º de recursos do tipo i requeridos pelo processo j (com $i = 0, 1, \dots, N-1$, $j = 0, 1, \dots, M-1$)

$A_{i,j}$ – n.º de recursos do tipo i já atribuídos ao processo j (com $i = 0, 1, \dots, N-1$, $j = 0, 1, \dots, M-1$) .

O problema pode ser abordado de duas maneiras diferentes

- ♦ *impedimento de lançamento de um novo processo* – quando não podem ser garantidas as condições da sua terminação
 - ♦ o processo P_M só é lançado se e só se

$$NTR_i \geq R_{i,M} + \sum_{j=0}^{M-1} R_{i,j} \quad (\text{com } i = 0, 1, \dots, N-1) .$$

Prevenção de deadlock no sentido lato - 3

- ♦ *impedimento de atribuição de um novo recurso a um dado processo* – quando daí resultar um estado inseguro

Algoritmo dos banqueiros de Dijkstra

- ♦ um novo recurso de tipo i só é atribuído ao processo P_m se e só se for possível ordenar os processos numa sucessão $j' = f(i, j)$ em que se tem sempre que

$$R_{i, j'} - A_{i, j'} < NTR_i - \sum_{k \geq j'}^{M-1} A_{i, k} \quad (\text{com } i = 0, 1, \dots, N-1, j' = 0, 1, \dots, M-1) ;$$

ou seja, quando for possível encontrar pelo menos uma sucessão de atribuição de recursos que conduza à terminação de todos os processos que coexistem.

Algoritmo dos banqueiros de Dijkstra - 1

| | | 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 |

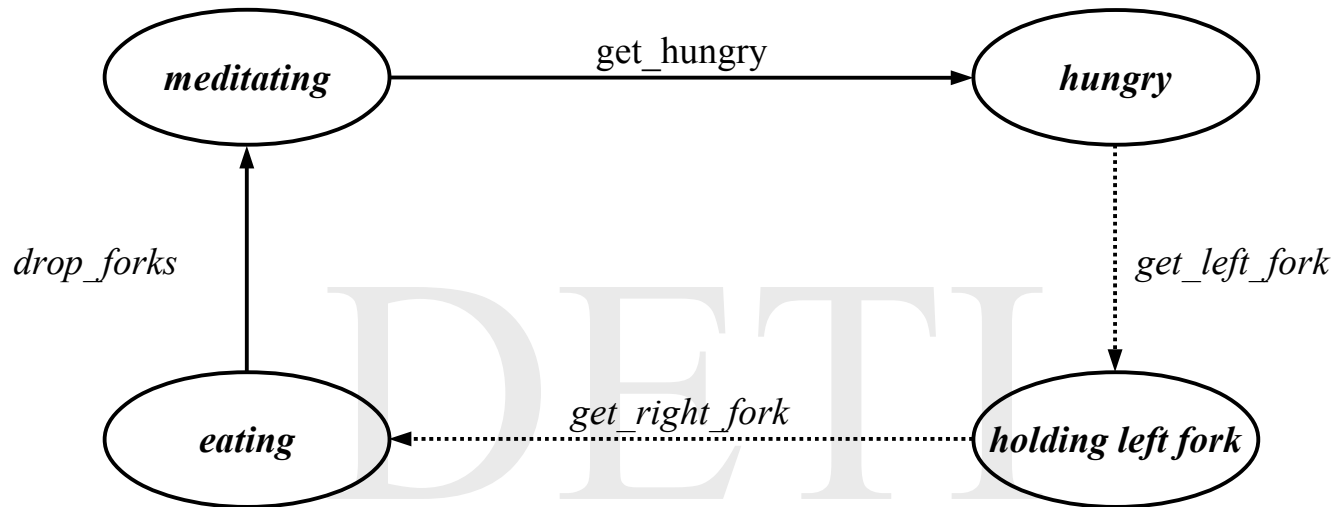
- Se p3 pede 2 recursos do tipo C, o pedido é recusado porque só há 1 disponível
- Se p3 pede 1 recurso do tipo B, que acontece? Porquê?

Algoritmo dos banqueiros de Dijkstra - 2

| | | A | B | C | D |
|-----------|-------|---|---|---|---|
| | total | 6 | 5 | 7 | 6 |
| | free | 3 | 0 | 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 | 0 | 4 | 0 |
| new Grant | p1 | 0 | 0 | 0 | 0 |
| | p2 | 0 | 0 | 0 | 0 |
| | p3 | 0 | 1 | 0 | 0 |

- Se p3 pede 2 recursos do tipo C, o pedido é recusado porque só há 1 disponível
- Se p3 pede 1 recurso do tipo B, que acontece? Porquê?

Algoritmo dos banqueiros de Dijkstra - 3



- cada filósofo, quando pretende os garfos, continua a pegar primeiro no garfo da esquerda e, só depois, no da direita, mas agora mesmo que o garfo da esquerda esteja sobre a mesa, o filósofo nem sempre pode pegar nele;
- só o poderá fazer se pelo menos um dos outros filósofos estiver a meditar; caso contrário, a atribuição origina uma *situação insegura*; *porquê?*
- não se trata, contudo, de uma solução geral (o *adiamento indefinido* não está completamente resolvido).

Prevenção de deadlock no sentido lato - 4

Análise Crítica

- ♦ As políticas de *prevenção de deadlock no sentido lato*, embora igualmente seguras, não são menos restritivas e ineficientes do que as políticas de *prevenção de deadlock no sentido estrito*. São, porém, mais versáteis e, por isso, mais fáceis de aplicar em situações gerais.
- ♦ Resumindo, tem-se que
 - ♦ *impedimento de lançamento de um novo processo* – a avaliação de estado seguro é feita *a priori*, o que tem como consequência uma política extremamente restritiva que leva a condição de pior caso possível ao limite;
 - ♦ *impedimento de atribuição de um novo recurso a um dado processo* – a avaliação de estado seguro é adiada para o momento de atribuição de um novo recurso, o que possibilita uma gestão mais eficiente dos recursos disponíveis e, portanto, um aumento de *throughput* do sistema.

Detecção e recuperação de deadlock - 1

- ♦ Uma última alternativa é pura e simplesmente atribuir os recursos sempre que eles estão disponíveis. A possibilidade de *deadlock* está então sempre presente e terá que ser tida em conta.
- ♦ Existem duas estratégias que podem ser seguidas
 - ♦ *estratégia da avestruz* – ignorar pura e simplesmente o problema e, quando o *deadlock* acontecer, matar os processos envolvidos ou, no caso limite, reiniciar o sistema (a mais comum);
 - ♦ *deteção de deadlock* – monitorar de tempos a tempos o estado dos diferentes processos, procurando determinar se existem cadeias circulares de processos e recursos (a *teoria de grafos* é comumente utilizada no desenvolvimento de algoritmos deste tipo).

Detecção e recuperação de deadlock - 2

- ♦ Após a detecção de *deadlock*, a cadeia circular de processos e recursos pode ser quebrada usando diversos métodos
 - ♦ *libertação forçada de um recurso* – em algumas situações, um recurso pode ser retirado a um processo, que é entretanto suspenso, permitindo o prosseguimento da execução dos restantes; mais tarde, o recurso volta a ser atribuído ao processo e este é recomeçado; trata-se do método mais eficaz, mas exige que o estado do recurso possa ser salvaguardado;
 - ♦ *rollback* – estabelecer um funcionamento do sistema que impõe o armazenamento periódico do estado dos diferentes processos; deste modo, quando ocorre *deadlock*, um recurso é libertado e o processo que o detinha vê a sua execução recuada até a um ponto anterior à atribuição desse recurso;
 - ♦ *morte de processos* – é o método mais radical e mais fácil de implementar.

Leituras sugeridas - 1

Operating Systems Concepts, Silberschatz, Galvin, Gagne, John Wiley & Sons, 8th Ed

- Capítulo 3: *Process concept* (Secção 3.4)
 - Mensagens
- Capítulo 6: *Synchronization* (Secções 6.1 a 6.8)
 - Condições de corrida e regiões críticas
 - Soluções do acesso com exclusão mútua a uma região crítica
 - Sincronização usando semáforos e monitores
- Capítulo 7: *Deadlocks*
 - Caracterização, prevenção, detecção e recuperação

Modern Operating Systems, Tanenbaum, Prentice-Hall International Editions, 3rd Ed

- Capítulo 2: *Processes and Threads*
 - Secções 2.3 e 2.5: Condições de corrida e regiões críticas, soluções do acesso com exclusão mútua a uma região crítica e sincronização usando semáforos, monitores e mensagens
- Capítulo 6: *Deadlocks*
 - Caracterização, prevenção, detecção e recuperação

Leituras sugeridas - 2

Operating Systems, Stallings, Prentice-Hall International Editions, 7th Ed

- Capítulo 5: *Concurrency: mutual exclusion and synchronization*
 - Condições de corrida e regiões críticas
 - Soluções do acesso com exclusão mútua a uma região crítica
 - Sincronização usando semáforos, monitores e mensagens
- Capítulo 6: *Concurrency: deadlock and starvation*
 - Caracterização, prevenção, detecção e recuperação