
Sebenta de SO

Programação concorrente, Process switching e
Processor Scheduling

PEDRO MARTINS

January 9, 2018

Contents

1	Conceitos Introdutórios	5
1.1	Exclusão Mútua	5
2	Acesso a um Recurso	6
3	Acesso a Memória Partilhada	6
3.1	Relação Produtor-Consumidor	7
3.1.1	Produtor	7
3.1.2	Consumidor	8
4	Acesso a uma Zona Crítica	8
4.1	Tipos de Soluções	9
4.2	Alternância Estrita (<i>Strict Alternation</i>)	9
4.3	Eliminar a Alternância Estrita	10
4.4	Garantir a exclusão mútua	10
4.5	Garantir que não ocorre deadlock	11
4.6	Mediar os acessos de forma determinística: <i>Dekker algorithm</i>	12
4.7	Dijkstra algorithm (1966)	13
4.8	Peterson Algorithm (1981)	14
4.9	Generalized Peterson Algorithm (1981)	15
5	Soluções de Hardware	16
5.1	Desativar as interrupções	16
5.2	Instruções Especiais em Hardware	17
5.2.1	Test and Set (TAS primitive)	17
5.2.2	Compare and Swap	17
5.3	Busy Waiting	18
5.4	Block and wake-up	19
6	Semáforos	20
6.1	Implementação	21
6.1.1	Operações	21
6.1.2	Solução típica de sistemas <i>uniprocessor</i>	21
6.2	Bounded Buffer Problem	22
6.2.1	Como Implementar usando semáforos?	23
6.3	Análise de Semáforos	25
6.3.1	Vantagens	25
6.3.2	Desvantagens	26
6.3.3	Problemas do uso de semáforos	26
6.4	Semáforos em Unix/Linux	26
7	Monitores	27
7.1	Implementação	27

7.2	Tipos de Monitores	28
7.2.1	Hoare Monitor	28
7.2.2	Brinch Hansen Monitor	29
7.2.3	Lampson/Redell Monitors	30
7.3	Bounded-Buffer Problem usando Monitores	30
7.4	POSIX support for monitors	32
8	Message-passing	32
8.1	Direct vs Indirect	33
8.1.1	Symmetric direct communication	33
8.2	Assymetric direct communications	33
8.3	Comunicação Indireta	33
8.4	Implementação	34
8.5	Buffering	34
8.6	Bound-Buffer Problem usando mensagens	35
8.7	Message Passing in Unix/Linux	35
9	Shared Memory in Unix/Linux	36
9.1	POSIX Shared Memory	36
9.2	System V Shared Memory	37
10	Deadlock	37
10.1	Condições necessárias para a ocorrência de deadlock	38
10.1.1	O Problema da Exclusão Mútua	39
10.2	Jantar dos Filósofos	39
10.3	Prevenção de Deadlock	40
10.3.1	Negar a exclusão mútua	41
10.3.2	Negar <i>hold-and-wait</i>	41
10.3.3	Negar <i>no preemption</i>	42
10.3.4	Negar a espera circular	43
10.4	Deadlock Avoidance	44
10.4.1	Condições para lançar um novo processo	44
10.4.2	Algoritmo dos Banqueiros	45
	Algoritmo dos banqueiros aplicado ao Jantar dos filósofos	46
10.5	Deadlock Detection	46
11	Processes and Threads	48
11.1	Arquitetura típica de um computador	48
11.2	Programa vs Processo	48
11.3	Execução num ambiente multiprogramado	49
11.4	Modelo de Processos	49
11.5	Diagrama de Estados de um Processo	50
11.5.1	Swap Area	52
11.5.2	Temporalidade na vida dos processos	53
11.6	State Diagram of a Unix Process	55

11.7	Supervisor preempting	56
11.8	Unix – traditional login	56
11.9	Criação de Processos	57
11.10	Execução de um programa em C/C++	61
11.11	Argumentos passados pela linha de comandos e variáveis de ambiente	61
11.12	Espaço de Endereçamento de um Processo em Linux	62
11.12.1	Process Control Table	63
12	Threads	64
12.1	Diagrama de Estados de uma thread	66
12.2	Vantagens de Multithreading	66
12.3	Estrutura de um programa multithreaded	67
12.4	Implementação de Multithreading	67
12.4.1	Libreria pthread	68
12.5	Threads em Linux	69
13	Process Switching	70
13.1	Exception Handling	72
13.2	Processing a process switching	73
14	Processor Scheduling	73
14.1	Scheduler	74
14.1.1	Long-Term Scheduling	74
14.1.2	Medium Term Scheduling	74
14.1.3	Short-Term Scheduling	75
14.2	CrITÉrios de Scheduling	75
14.2.1	User oriented	75
14.2.2	System oriented	76
14.3	Preemption & Non-Preemption	76
14.4	Scheduling	77
14.4.1	Favouring Fearness	77
14.4.2	Priorities	78
	Prioridades Estáticas	78
	Prioridades Dinâmicas	79
	Shortest job first (SJF) / Shortest process next (SPN)	80
14.5	Scheduling Policies	81
14.5.1	First Come, First Serve (FCFS)	81
14.5.2	Round-Robin	81
14.5.3	Shortest Process Next (SPN) ou Shortest Job First (SJF)	82
14.5.4	Linux	82
	Algoritmo Tradicional	83
14.6	Novo Algoritmo	83

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 batch, 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.

```
1  /* processes competing for a resource - p = 0, 1, ..., N-1 */
2  void main (unsigned int p)
3  {
4      forever
5      {
6          do_something();
7          access_resource(p);
8          do_something_else();
9      }
10 }
11
12 void access_resource(unsigned int p)
13 {
14     enter_critical_section(p);
15     use_resource();      // critical section
16     leave_critical_section(p);
17 }
```

3 Acesso a Memória Partilhada

O acesso à memória partilhada é muito semelhante ao acesso 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, **re-ativar o acesso a memória partilhada** pelos outros processos.

```
1  /* shared data structure */
2  shared DATA d;
3
4  /* processes sharing data - p = 0, 1, ..., N-1 */
5  void main (unsigned int p)
6  {
7      forever
8      {
9          do_something();
10         access_shared_area(p);
11         do_something_else();
12     }
```

```
13 }
14
15 void access_shared_area(unsigned int p)
16 {
17     enter_critical_section(p);
18     manipulate_shared_area(); // critical section
19     leave_critical_section(p);
20 }
```

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

```
1  /* communicating data structure: FIFO of fixed size */
2  shared FIFO fifo;
3
4  /* producer processes - p = 0, 1, ..., N-1 */
5  void main (unsigned int p)
6  {
7      DATA val;
8      bool done;
9
10
11     forever
12     {
13         produce_data(&val);
14         done = false;
15         do
16         {
17             // Beginning of Critical Section
18             enter_critical_section(p);
19             if (fifo.notFull())
20             {
21                 fifo.insert(val);
22                 done = true;
23             }
24             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 poder **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:

- **Effective 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 **atômica**

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 `enter_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)
```

```
7 {
8     unsigned int other_pid_ = 1 - own_pid;
9
10    want_enter[own_pid] = true;
11    while (want_enter[other_pid]);
12 }
13
14 void leave_critical_section(unsigned int own_pid)
15 {
16     want_enter[own_pid] = false;
17 }
```

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

```
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)
7 {
8     unsigned int other_pid_ = 1 - own_pid;
9
10    want_enter[own_pid] = true;
11    while (want_enter[other_pid])
12    {
13        want_enter[own_pid] = false;    // go back
14        random_dealy();
15        want_enter[own_pid] = true;     // attempt a to go to the critical
16        section
17    }
```

```
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 algorithm*

```
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
15             priority process
16         {
17             want_enter[own_pid] = false;       // go back
18             while (own_pid != p_w_priority);   // waits to access to his
19             critical section while
20
21             // its is not the priority
22             // process
23             want_enter[own_pid] = true;        // attempt to go to his
24             critical section
25         }
26     }
27 }
```

```
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 atômica, 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.

```
1  shared int value = 0;
2
3  int compare_and_swap(int * value, int expected, int new_value)
4  {
5      int v = *value;
6      if (*value == expected)
7          *value = new_value;
8      return v;
9  }
10
11 void lock(int * flag)
12 {
13     while (compare_and_swap(&flag, 0, 1) != 0);
14 }
15
16 void unlock(bool * flag)
17 {
18     *flag = 0;
19 }
```

5.3 Busy Waiting

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

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 **uniprocessador**, 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 quantum* 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.

```
1  #define R ... /* process id = 0, 1, ..., R-1 */
2
3  shared unsigned int access = 1;    // Note that access is an integer, not a
    boolena
4
5  void enter_critical_section(unsigned int own_pid)
6  {
7      // Beginning of atomic operation
8      if (access == 0)
9          block(own_pid);
10
11     else access -= 1;
12     // Ending of atomic operation
13 }
14
15 void leave_critical_section(unsigned int own_pid)
16 {
17     // Beginning of atomic operation
18     if (there_are_blocked_processes)
19         wake_up_one();
20     else access += 1;
21     // Ending of atomic operation
22 }
```

```
1  /* producers - p = 0, 1, ..., N-1 */
2  void producer(unsigned int p)
3  {
4      DATA data;
5      forever
6      {
7          produce_data(&data);
8          bool done = false;
9          do
10         {
```

```
11         lock(p);
12         if (fifo.notFull())
13         {
14             fifo.insert(data);
15             done = true;
16         }
17         unlock(p);
18     } while (!done);
19     do_something_else();
20 }
21 }
```

```
1  /* consumers - c = 0, 1, ..., M-1 */
2  void consumer(unsigned int c)
3  {
4      DATA data;
5      forever
6      {
7          bool done = false;
8          do
9          {
10             lock(c);
11             if (fifo.notEmpty())
12             {
13                 fifo.retrieve(&data);
14                 done = true;
15             }
16             unlock(c);
17         } while (!done);
18         consume_data(data);
19         do_something_else();
20     }
21 }
```

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)

```
1 typedef struct
2 {
3     unsigned int val;    /* can not be negative */
4     PROCESS *queue;     /* queue of waiting blocked processes */
5 } 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*

```
1 /* array of semaphores defined in kernel */
2 #define R /* semid = 0, 1, ..., R-1 */
3
4 static SEMAPHORE sem[R];
5
6 void sem_down(unsigned int semid)
7 {
8     disable_interruptions;
9     if (sem[semid].val == 0)
10         block_on_sem(getpid(), semid);
11     else
12         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 multi-processador 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 multi-processador.

Uma solução alternativa seria a extensão do **disable_interruptions** a todos os processadores. No entanto, iríamos 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:

- **nitems:** 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 registoções 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

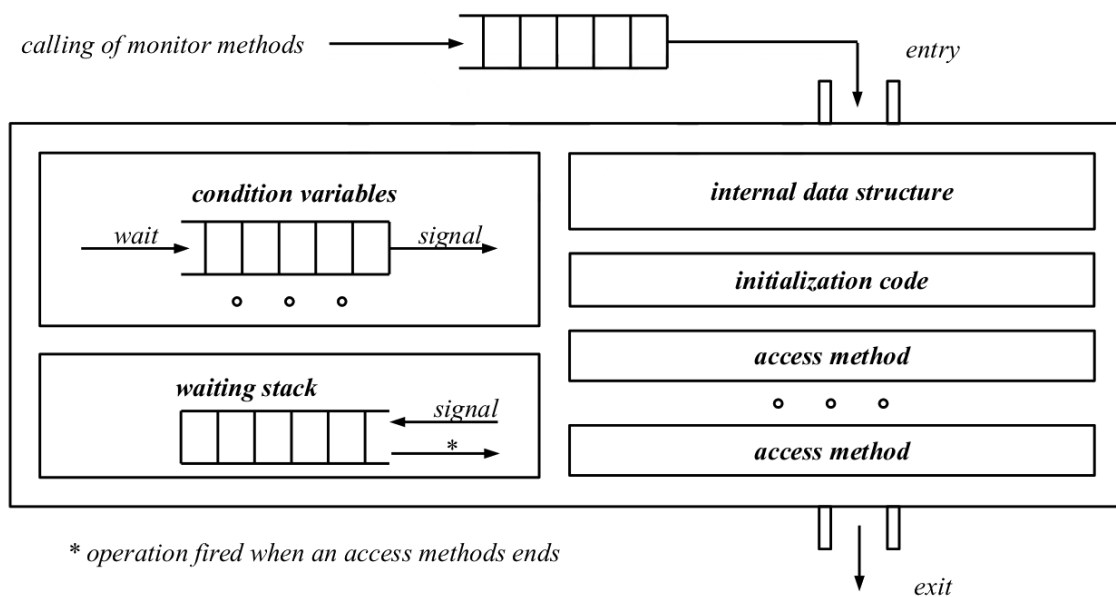


Figure 1: Diagrama da estrutura interna de um Monitor de Hoare

- 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

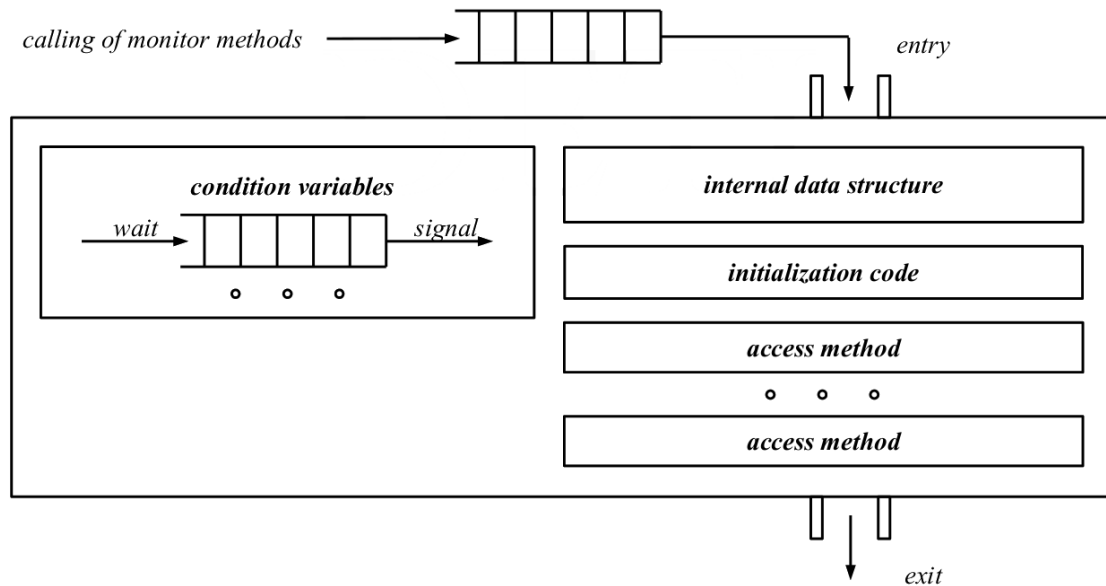


Figure 2: Diagrama da estrutura interna de um Monitor de Brinch Hansen

- 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

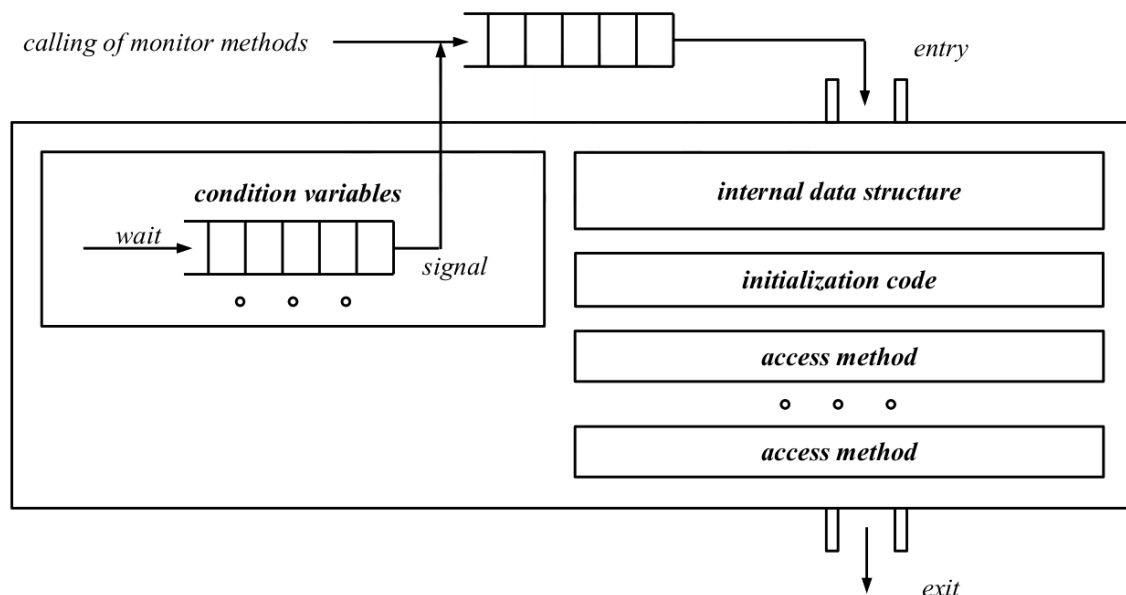


Figure 3: Diagrama da estrutura interna de um Monitor de Lampson/Redell

- 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 ti nha 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(. . .)** seguid 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 *Lamport/Redell* em C/C++, usando:

- `mutexes`
- `variáveis de condição`

As funções disponíveis são:

- `pthread_create`: **cria** uma nova *thread* (similar ao *fork*)
- `pthread_exit`: equivalente à `exit`
- `pthread_join`: equivalente à `waitpid`
- `pthread_self`: equivalente à `getpid`
- `pthread_mutex_*`: manipulação de **mutexes**
- `pthread_cond_*`: manipulação de **variáveis condicionais**
- `pthread_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 **uniprocessador** 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 expliciting 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 um conjunto de processos comunicantes, só existindo **um canal de comunicação**

8.2 Assymetric direct communications

Só o `sender` tem de explicitar o destinatário:

- `send(P, message:`
- `receive(id, message)`: recebe 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

```
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     MESSAGE msg;
11
12     forever
13     {
14         produce_data(&val);
15         make_message(msg, data);
16         send(msg);
17         do_something_else();
18     }
19 }
20
21 /* consumers - c = 0, 1, ..., M-1 */
22 void consumer(unsigned int c)
23 {
24     DATA data;
25     MESSAGE msg;
26
27     forever
28     {
29         receive(msg);
30         extract_data(data, msg);
31         consume_data(data);
32         do_something_else();
33     }
34 }
```

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 receção 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 receção 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 poder 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 poder 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 mantém 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. ($A \Rightarrow B$)

Se **uma delas não se verifica**, não há deadlock. ($\sim B \Rightarrow \sim A$)

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 a tentar entrar na sua zona crítica, então eventualmente esse processo entra na sua zona crítica
4. **First-In-First-Out:** Nenhum processo que 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:

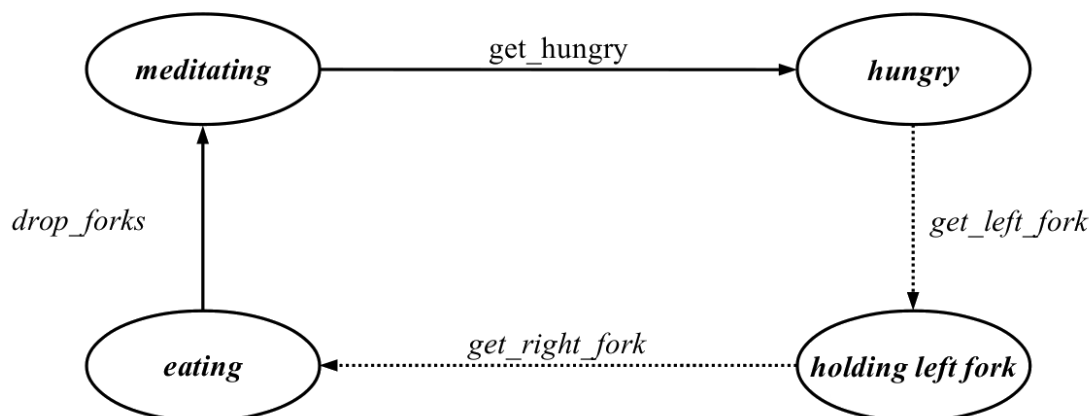


Figure 4: Ciclo de Vida de um filósofo

```
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 re-aquisição) 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**

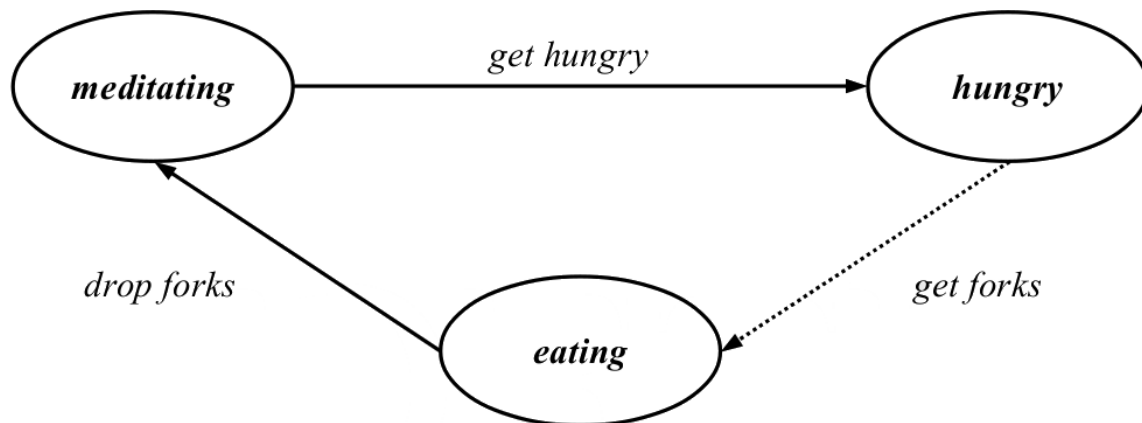


Figure 5: Negar *hold-and-wait*

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`

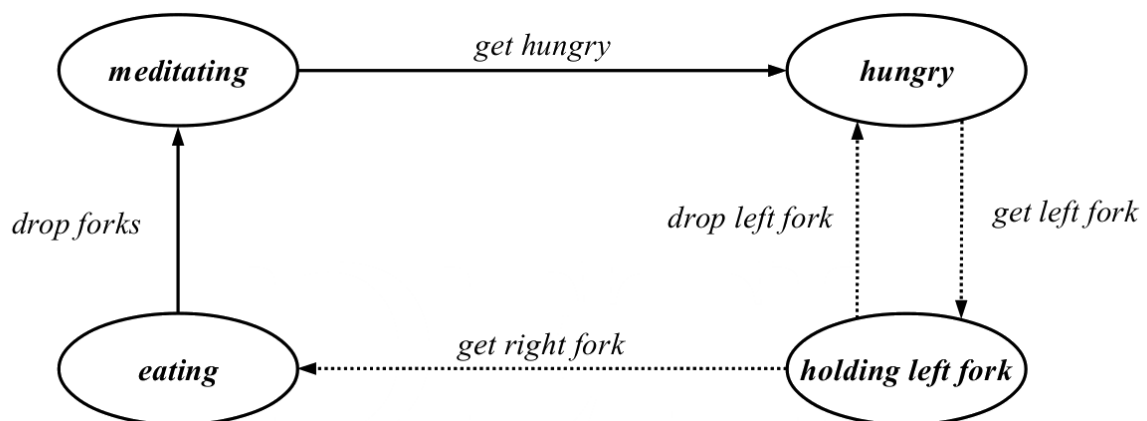


Figure 6: Negar a condição de *no preemption* dos recursos

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 adquire primeiro o *right_fork*

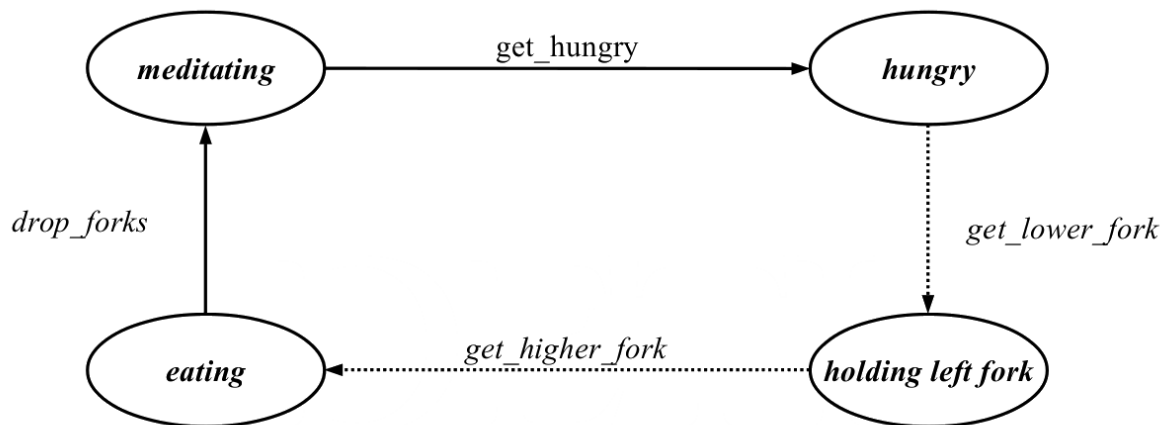


Figure 7: Negar a condição de espera circular no acesso aos recursos

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 ($i = 0, 1, \dots, N-1$)
- $R_{i,j}$: o número de recursos do tipo i requeridos pelo processo j , ($i=0, 1, \dots, N-1$ e $j=0, 1, \dots, M-1$)

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:

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

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:

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

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 **dead-lock**

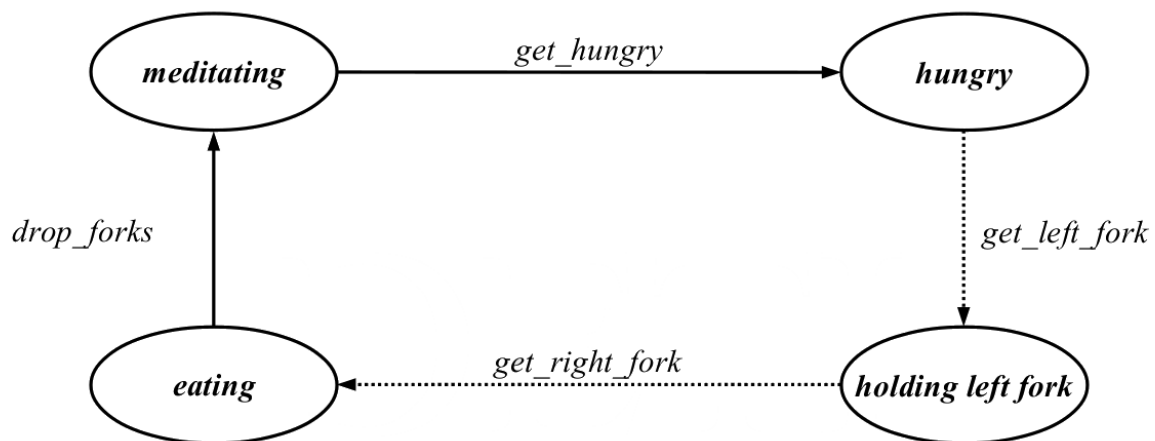


Figure 8: Algoritmo dos banqueiros aplicado ao Jantar dos filósofos

10.5 Deadlock Detection

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

- O estado do 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`

11 Processes and Threads

11.1 Arquitectura típica de um computador

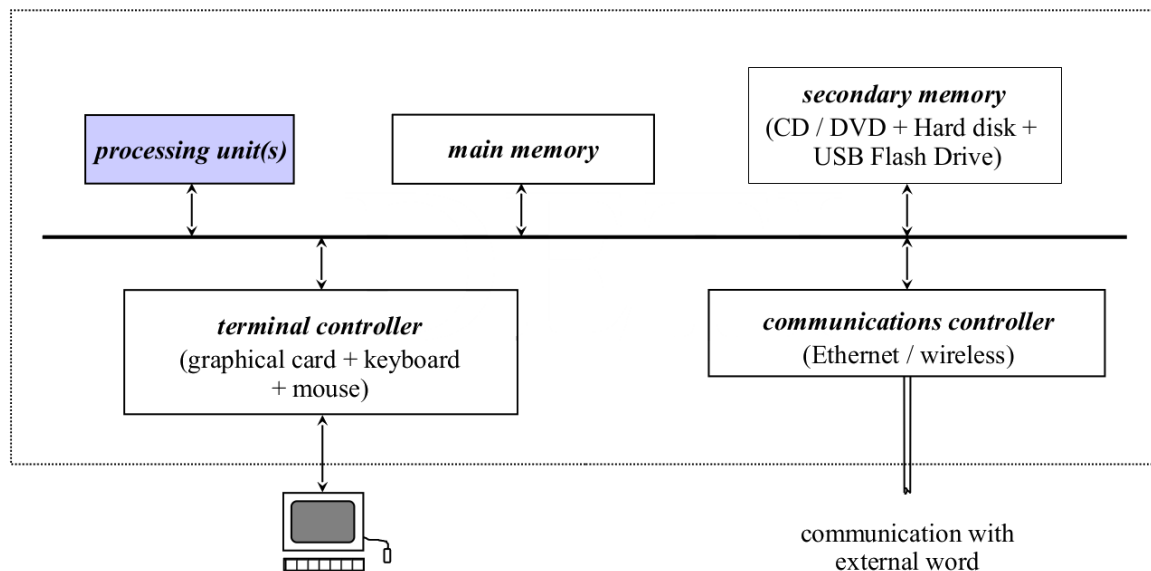


Figure 9: Arquitectura típica de um computador

11.2 Programa vs Processo

- **programa:** conjunto de instruções que definem como uma tarefa é executada num computador
 - É apenas um **conjunto de instruções** (código máquina), nada mais
 - Para realizar essas **funções/instruções/tarefas** o código (ou a versão compilada dele) tem de ser executado(a)
- **processo:** Entidade que representa a **execução de um programa**
 - Representa a sua atividade
 - Tem associado a si:
 - * código que ao contrário do programa está armazenado num endereço de memória (addressing space)
 - * **dados** (valores das diferentes variáveis) da execução corrente
 - * valores atuais dos registos internos do processador
 - * dados dos I/Os, ou seja, dados que estão a ser transferidos entre dispositivos de input e output
 - * Estado da execução do programa, ou seja, qual a próxima execução a ser executada (registo PC)
 - Podem existir diferentes processos do mesmo programa
 - * Ambiente **multiprogramado** - mais processos que processadores

11.3 Execução num ambiente multiprogramado

O sistema assume que o processo que está na posse do processador irá **ser interrompido**, podendo assim executar outro processo e dar a “sensação” em **macro tempo** de **simultaneidade**. Nestas situações, o OS é responsável por:

- tratar da **mudança do contexto de execução**, guardando
 - o valor dos registos internos
 - o valor das variáveis
 - o endereço da próxima instrução a ser executada
- chamar o novo processo que vai ocupar agora o CPU e:
 - Esperar que o novo processo termine a realização das suas operações **ou**
 - Interromper o processo, **parando a sua execução no** processador quando este esgotar o seu **time quantum**

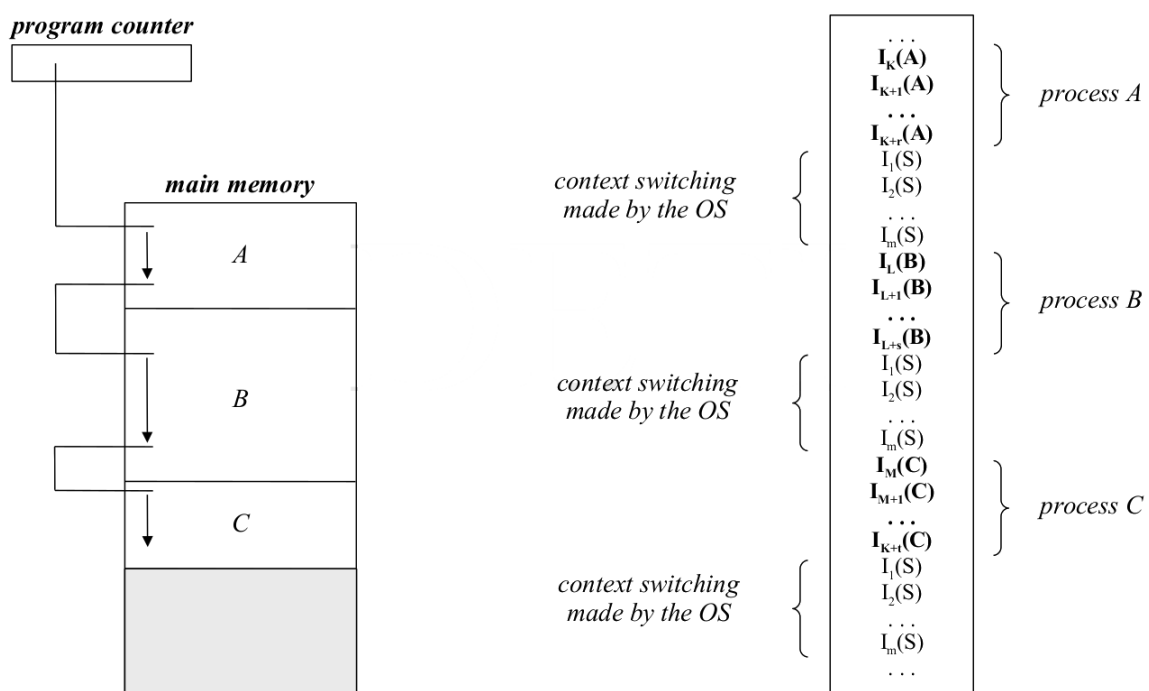


Figure 10: Exemplo de execução num ambiente multiprogramado

11.4 Modelo de Processos

Num ambiente **multiprogramado**, devido à constante **troca de processos**, é difícil expressar uma modelo para o processador. Devido ao elevado numero de processo e ao multiprogramming, torna-se difícil de saber qual o processo que está a ser executado e qual a fila de processos as ser executada.

É mais fácil assumir que o ambiente multiprogramado pode ser representado por um **conjunto de processadores virtuais**, estando um processo atribuído a cada um.

O processador virtual está: - **ON**: se o processo que lhe está atribuído está a ser executado - **OFF**: se o processo que lhe está atribuído não está a ser executado

Para este modelo temos ainda de assumir que: - Só um dos processadores é que pode estar ativo num dado período de tempo - O número de **processadores virtuais ativos é menor** (ou igual, se for um ambiente **single processor**) ao número de **processadores reais** - A execução de um processo **não é afetada** pelo instante temporal nem a localização no código em que o processo é interrompido e é efetuado o switching - Não existem restrições do número de vezes que qualquer processo pode ser interrompido, quer seja executado total ou parcialmente

A operação de **switching entre processos** e consequentemente entre processadores virtuais ocorre de forma não **controlada** pelo programa a correr no CPU

Uma **operação de switching** é equivalente a efetuar o **Turning Off** de um processo virtual e o **Turning On** de outro processo virtual.

- **Turning Off** implica **guardar** todo o **contexto de execução**
- **Turning On** implica carregar todo o contexto de execução, **restaurando o estado do programa** quando foi interrompido

11.5 Diagrama de Estados de um Processo

Durante a sua existência, um processo pode assumir diferentes estados, dependendo das condições em que se encontra:

- **run**: O processo está em execução, tendo a posse do processador
- **blocked**: O processo está bloqueado à **espera de um evento externo** para estar em condições retomar a sua execução. Esse evento externo pode ser:
 - Acesso a um recurso da máquina
 - Fim de uma operação de I/O
 - ...
- **ready**: O processo está pronto a ser executado, mas está à espera que o processador lhe dê a ordem de **start/resume** para poder retomar a sua execução.

As **transições entre estados** normalmente resultam de **intervenções externas ao processo**, mas podem depender de situações em que o processo força uma transição: - termina a sua execução antes de terminar o seu **time quantum** - Leitura/Escrita em I/O (scanf/printf)

Mesmo que um processo **não abandone o processador por sua iniciativa**, o **scheduler** é responsável por **planear o uso do processador pelos diferentes processos**.

O (**Process**) **Scheduler** é um módulo do kernel que **monitoriza e gere as transições entre processos**. Assim, um **while**(1) não é executado *ad eternum*. Um processador **multiprocess** só permite que o ciclo infinito seja executado quando é atribuído **CPU time** ao processo.

Existem diferentes políticas que permitem controlar a execução destas transições

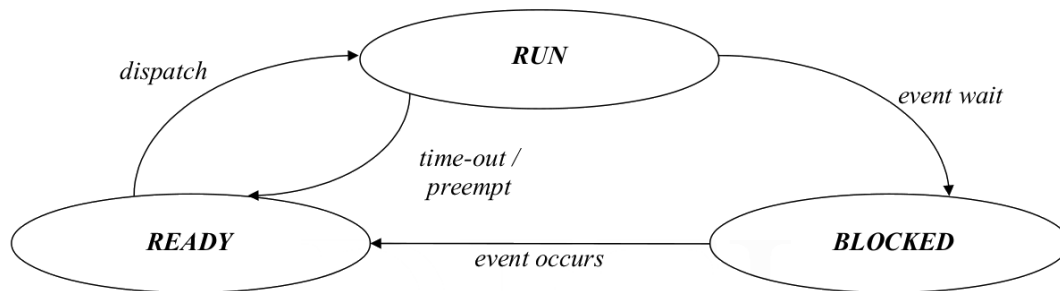


Figure 11: Diagrama de Estados do Processador - Básico

Triggers das transições entre estados:

- **dispatch:**

- O processo que estava em modo **run** perdeu o acesso ao processador.
- Do conjunto de processos prontos a serem executados, tem de ser escolhido **um** para ser executado, sendo-lhe atribuído o processador.
- A escolha feita pelo **dispatcher** pode basear-se em:
 - * um sistema de prioridades
 - * requisitos temporais
 - * aleatoriedade
 - * divisão igual do CPU

- **event wait:**

- O processo que estava a ser executado sai do estado **run**, não estando em execução no processador.
 - * Ou porque é impedido de continuar pelo scheduler
 - * Ou por iniciativa do próprio processo.
 - `scanf`
 - `printf`
- O CPU guarda o estado de execução do processo
- O processo fica em estado **blocked** à **espera da ocorrência de um evento externo**, **event occurs**

- **event occurs:**

- Ocorreu o evento que o processo estava à espera
- O processo transita do estado **blocked** para o estado **ready**, ficando em fila de espera para que lhe seja atribuído o processador

- **time_out:**

- O processo esgotou a sua janela temporal, **time quantum**
- Através de uma interrupção em **hardware**, o sistema operativo vai forçar a saída do processo do processador

- Transita para o estado *ready* até lhe ser atribuído um novo *time-quantum* do CPU
- A transição por time-out ocorre em qualquer momento do código.
- Os sistemas podem ter *time quantum* diferentes e os *time slots* alocados não têm de ser necessariamente iguais entre dois sistemas.

- **preempt:**

- O processo que possui a posse do processador tem uma prioridade mais baixa do que um processo que acordou e está pronto a correr (estado *ready*)
- O processo que está a correr no processador é **removido** e transita para o estado *ready*
- Passa a ser **executado** o processo de **maior prioridade**

11.5.1 Swap Area

O diagram de estados apresentado não leva em consideração que a **memória principal** (RAM) é **finita**. Isto implica que o número de **processos coexistentes em memória é limitado**.

É necessário usar a **memória secundária** (Disco Rígido) para **estender a memória principal** e aumentar a capacidade de armazenamento dos estados dos processos.

A **memória swap** pode ser:

- uma partição de um disco
- um ficheiro

Qualquer processo que **não esteja a correr** por ser *swapped out*, libertando memória principal para outros processos

Qualquer processo *swapped out* pode ser *swapped in*, **quando existir memória principal disponível**

Ao diagrama de estados tem de ser adicionados: - dois novos estados: - **suspended-ready**: Um processo no estado *ready* foi *swapped-out* - **suspended-blocked**: O processo no estado *blocked* foi *swapped-out* - dois novos tipos de transições: - **suspend**: O processo é *swapped out* - **activate**: O processo é *swapped in*

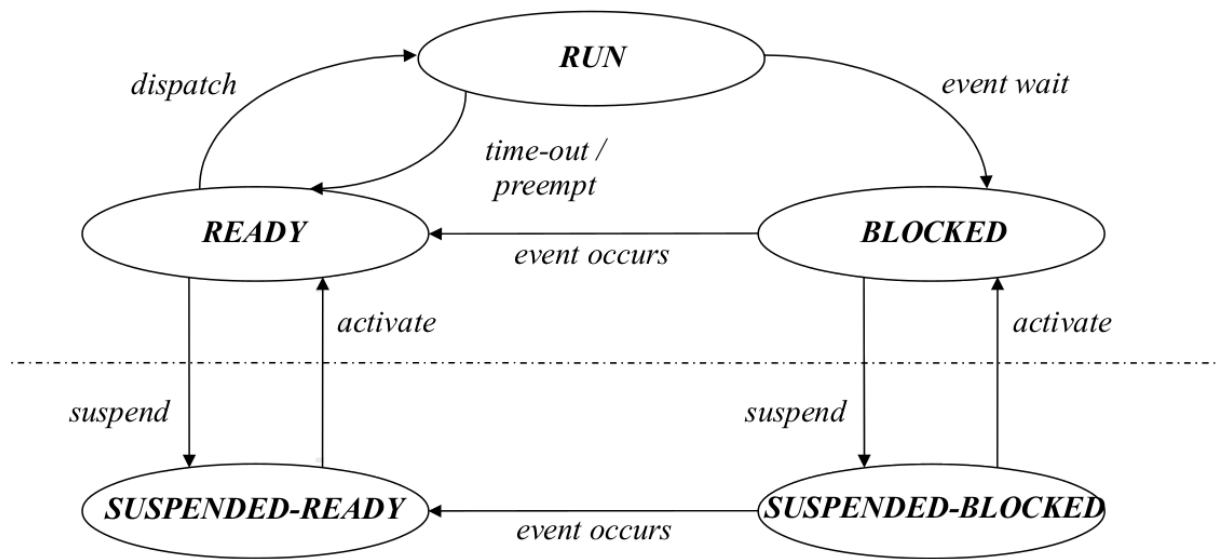


Figure 12: Diagrama de Estados do Processador - Com Memória de Swap

11.5.2 Temporalidade na vida dos processos

O diagrama assume que os processos são **intemporais**. Excluindo alguns processos de sistema, todos os processos são **temporais**, i.e.:

1. Nascem/São criados
2. Existem (por algum tempo)
3. Morrem/Terminam

Para introduzi a temporalidade no diagrama de estados, são necessários dois novos estados: - **new**: - O processo foi criado - Ainda não foi atribuído à **pool** de processos a serem executados - A estrutura de dados associado ao processo é inicializada - **terminated**: - O processo foi descartado da fila de processos executáveis - Antes de ser descartado, existem ações que tem de tomar (*needs clarification*)

Em consequência dos novos estados, passam a existir três novas transições: - **admit**: O processo é admitido pelo OS para a **pool** de processos executáveis - **exit**: O processo informa o SO que terminou a sua execução - **abort**: Um processo é forçado a terminar. - Ocorreu um **fatal error** - Um processo autorizado abortou a sua execução

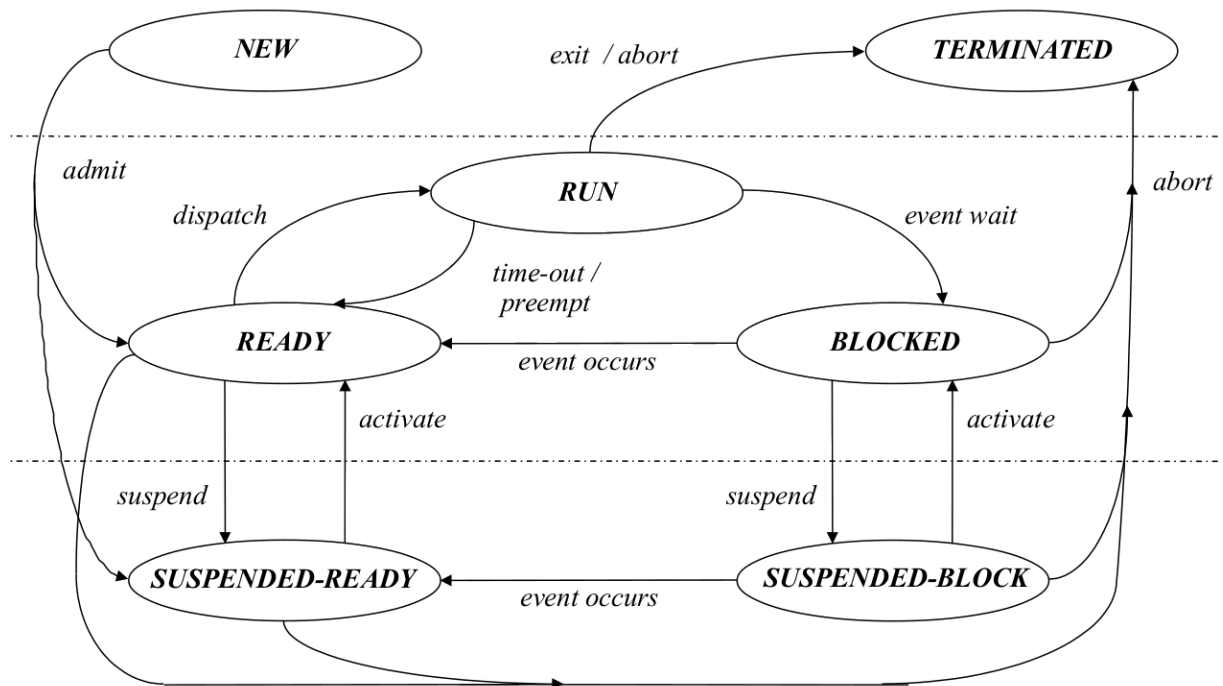


Figure 13: Diagrama de Estados do Processador - Com Processos Temporalmente Finitos

11.6 State Diagram of a Unix Process

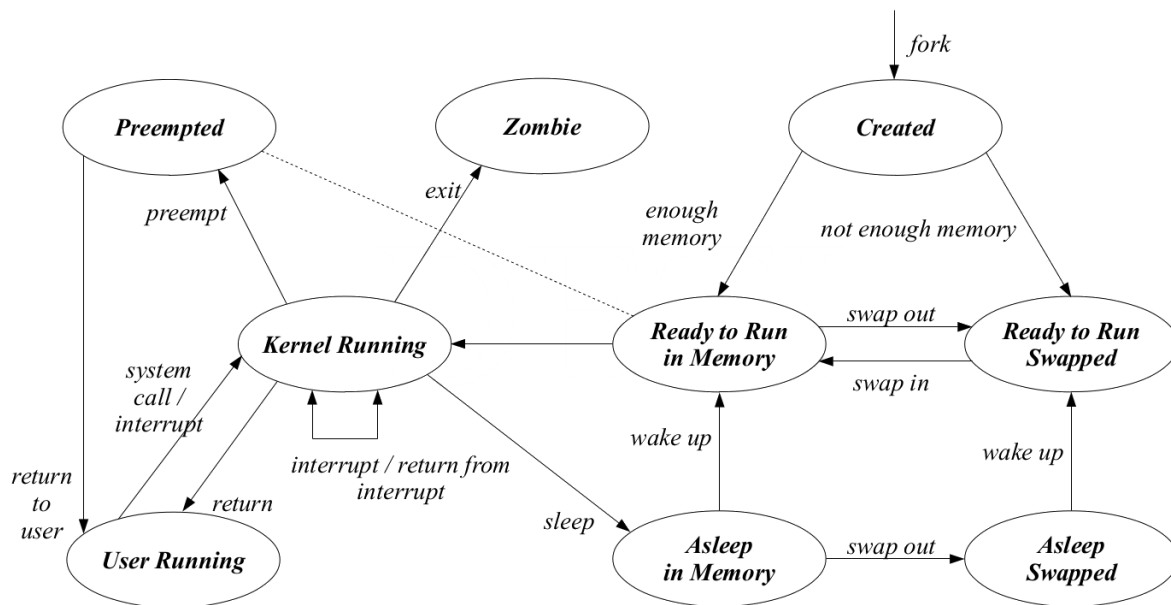


Figure 14: Diagrama de Estados do Processador - Com Memória de Swap

As três diferenças entre o diagrama de estados de um processo e o diagrama de estados do sistema Unix são

1. Existem **dois estados run**

1. **kernel running**
2. **user running**

- Diferem **no modo** como o processador **executa o código máquina**, existindo **mais instruções e diretivas disponíveis** no modo supervisor (**root**)

2. O estado **ready** é dividido em dois estados:

1. **ready to run in memory:** O processo está pronto para ser executado/continuar a execução, estando guardado o seu estado em memória
2. **preempted:** O processo que estava a ser executado foi **forçado a sair do processador** porque **existe um processo mais prioritário para ser executado**
 - Estes **estados são equivalentes** porque:
 - estão ambos **armazenado na memória principal**
 - quando um processo é **preempted** continua pronto a ser executado (não precisando de nenhuma informação de I/O)
 - Partilham a mesma fila (**queue**) de processos, logo são tratados de forma idêntica pelo OS
 - Quando um **processo do utilizador abandona o modo de supervisor** (corre com permissões **root**), **pode ser preempted**

3. A transição de `time-out` que existe no diagrama dos estados de um processo em UNIX é coberta pela transição `preempted`

11.7 Supervisor preempting

Tradicionalmente, a **execução** de um processo **em modo supervisor** (`root`) implicava que a execução do processo **não pudesse ser** interrompida, ou seja, o processo não pudesse ser **preempted**. Ou seja, o UNIX não permitia **real-time processing**

Nas novas versões o código está dividido em **regiões atômicas**, onde a **execução não pode ser interrompida** para garantir a **preservação de informação das estruturas de dados a manipular**. Fora das regiões atômicas é seguro interromper a execução do código

Cria-se assim uma nova transição, **return to kernel** entre os estados `preempted` e `kernel running`.

11.8 Unix – traditional login

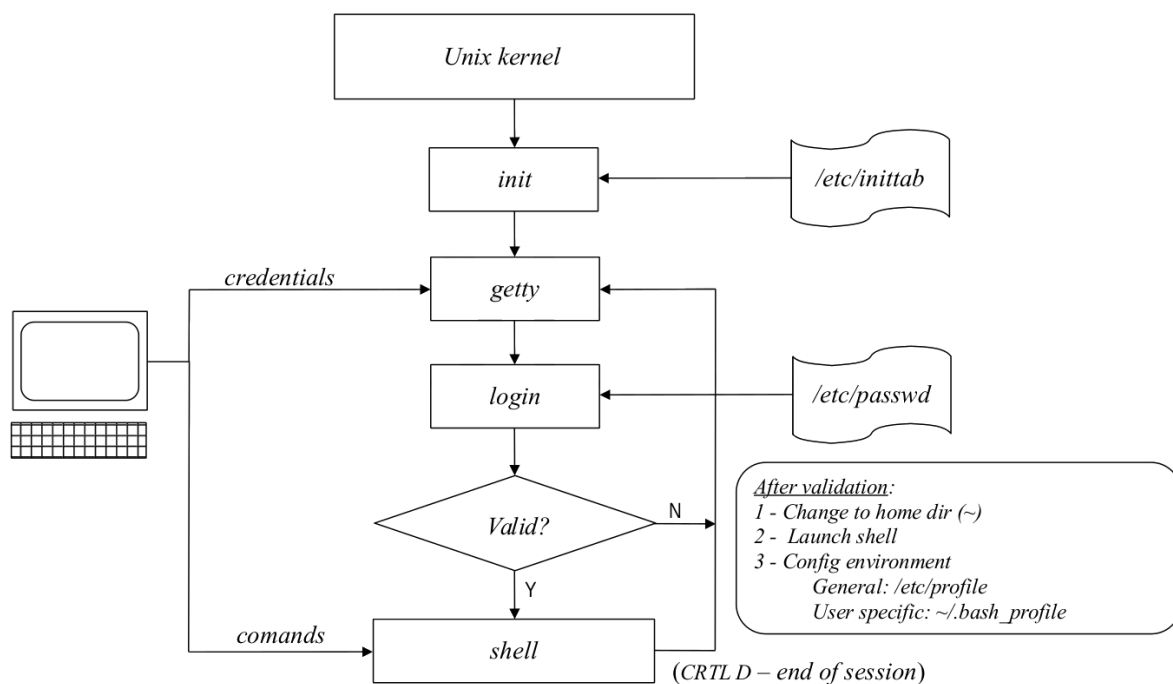


Figure 15: Diagrama do Login em Linux

11.9 Criação de Processos

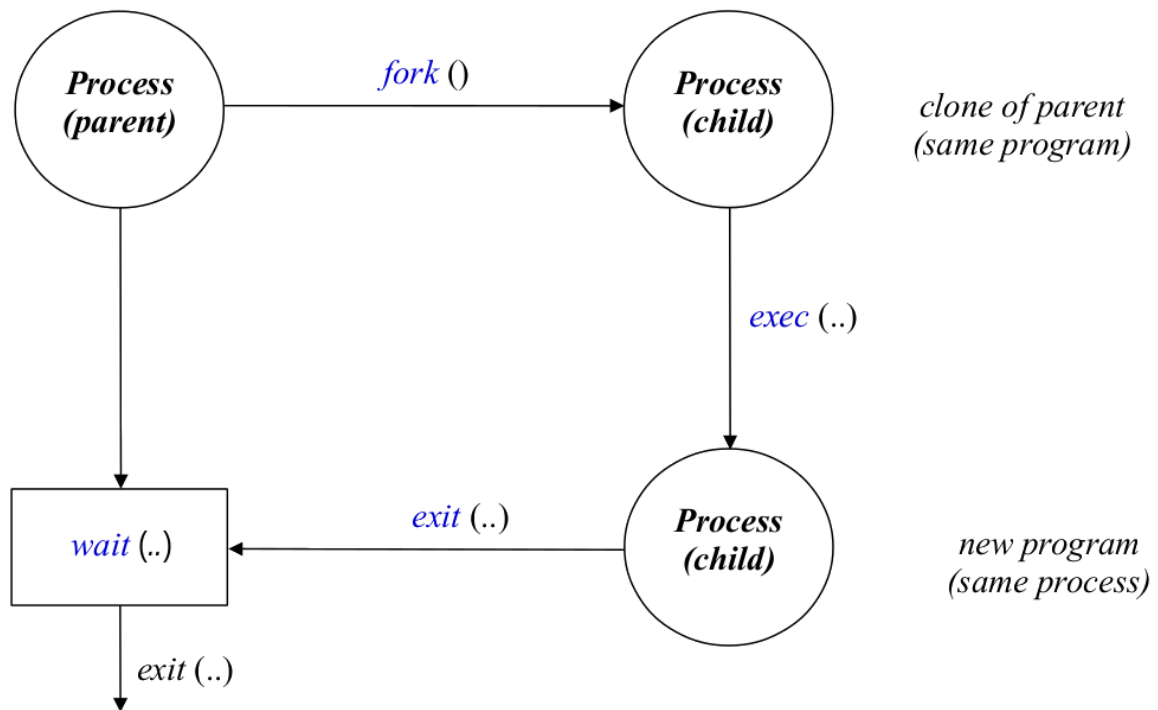


Figure 16: Criação de Processos

```
1 #include <stdio.h>
2 #include <stdlib.h>
3 #include <sys/types.h>
4 #include <unistd.h>
5
6 int main(void)
7 {
8     printf("Before the fork:\n");
9     printf(" PID = %d, PPID = %d.\n",
10         getpid(), getppid());
11
12     fork();
13
14     printf("After the fork:\n");
15     printf(" PID = %d, PPID = %d. Who am I?\n", getpid(), getppid());
16
17     return EXIT_SUCCESS;
18 }
```

- **fork: clona** o processo existente, criando uma **réplica**
 - O estado de execução é igual, incluindo o PC (*Program Counter*)
 - O **mesmo programa** é executado em **processos diferentes**
 - Não existem garantias que o pai execute primeiro que o filho
 - * Tudo depende do **time quantum** que o processo pai ocupa antes do **fork**
 - * É um ambiente multiprogramado: os dois programas correm em **simultâneo no micro tempo**
- O **espaço de endereçamento** dos dois processos é **igual**
 - É seguida uma filosofia **copy-on-write**. Só é efetuada a cópia da página de memória se o **processo filho modificar** os conteúdos das variáveis

Existem variáveis diferentes:

- **PPID:** Parent Process ID
- **PID:** Process ID

```
1 #include <stdio.h>
2 #include <stdlib.h>
3 #include <sys/types.h>
4 #include <unistd.h>
5
6 int main(void)
7 {
8     printf("Before the fork:\n");
9     printf(" PID = %d, PPID = %d.\n",
10         getpid(), getppid());
11
12     int ret = fork();
13
14     printf("After the fork:\n");
15     printf(" PID = %d, PPID = %d, ret = %d\n", getpid(), getppid(), ret);
16
17     return EXIT_SUCCESS;
18 }
```

O retorno da instrução **fork** é diferente entre o processo pai e filho:

- pai: **PID** do filho
- filho: 0

O retorno do **fork** pode ser usado como variável booleana, de modo a **distinguir o código a correr no filho e no pai**

```
1 #include <stdio.h>
2 #include <stdlib.h>
3 #include <sys/types.h>
4 #include <unistd.h>
5
```

```
6  int main(void)
7  {
8      printf("Before the fork:\n");
9      printf(" PID = %d, PPID = %d.\n", getpid(), getppid());
10
11     int ret = fork();
12
13     if (ret == 0)
14     {
15         printf("I'm the child:\n");
16         printf(" PID = %d, PPID = %d\n", getpid(), getppid());
17     }
18     else
19     {
20         printf("I'm the parent:\n");
21         printf(" PID = %d, PPID = %d\n", getpid(), getppid());
22     }
23
24     printf("After the fork:\n");
25     printf(" PID = %d, PPID = %d, ret = %d\n", getpid(), getppid(), ret);
26
27     return EXIT_SUCCESS;
28 }
```

O `fork` por si só não possui grande interesse. O interesse é poder executar um programa diferente no filho.

- **exec system call:** Executar um programa diferente no processo filho
- **wait system call:** O pai esperar pela conclusão do programa a correr nos filhos

```
1  #include <stdio.h>
2  #include <stdlib.h>
3  #include <sys/types.h>
4  #include <unistd.h>
5
6  int main(void)
7  {
8      /* check arguments */
9      if (argc != 2)
10     {
11         fprintf(stderr, "spawn <path to file>\n");
12         exit(EXIT_FAILURE);
13     }
14     char *aplic = argv[1];
15
16     printf("=====\n");
17
18     /* clone phase */
19     int pid;
```

```
20     if ((pid = fork()) < 0)
21     {
22         perror("Fail cloning process");
23         exit(EXIT_FAILURE);
24     }
25
26     /* exec and wait phases */
27     if (pid != 0) // only runs in parent process
28     {
29         int status;
30         while (wait(&status) == -1);
31         printf("=====\n");
32         printf("Process %d (child of %d)"
33             "ends with status %d\n",
34             pid, getpid(), WEXITSTATUS(status));
35     }
36     else // this only runs in the child process
37     {
38         execl(aplic, aplic, NULL);
39         perror("Fail launching program");
40         exit(EXIT_FAILURE);
41     }
42 }
```

O `fork` pode **não ser bem sucedido**, ocorrendo um `fork failure`.

11.10 Execução de um programa em C/C++

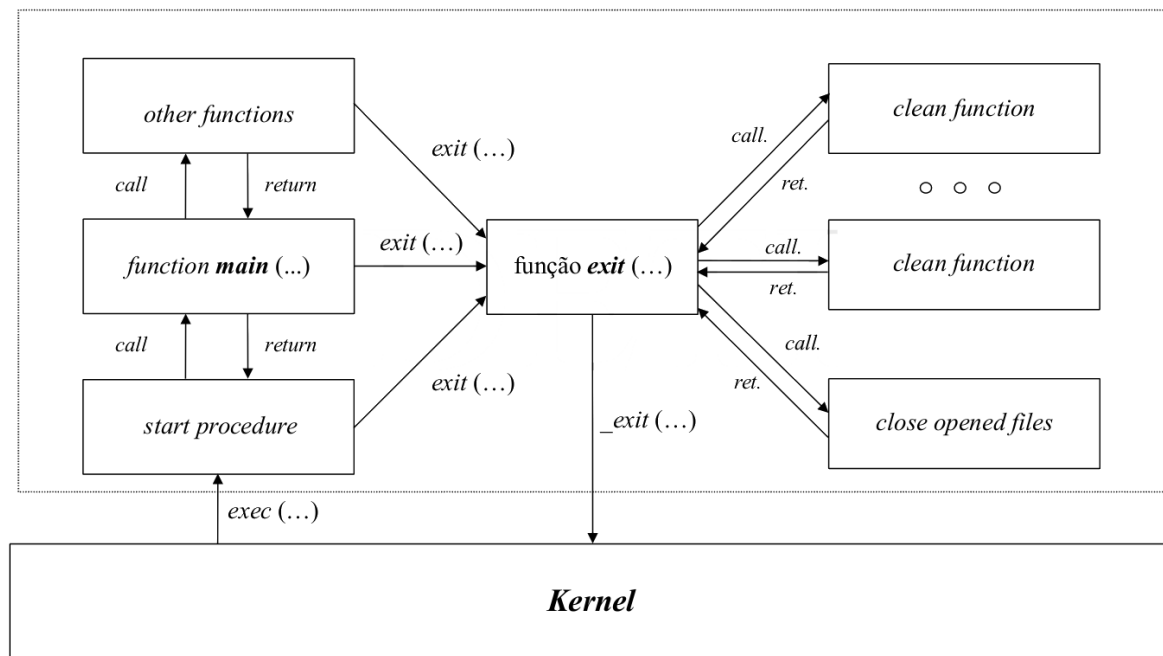


Figure 17: Execução de um programa em C/C++

- Em C/C++ o nome de uma função é um ponteiro para a sua função.
- Em C/C++ um include não inclui a biblioteca
 - Indica ao programa que vou usar uma função que tem esta assinatura
- **atexit:** Regista uma função para ser chamada no fim da execução normal do programa
 - São chamadas em ordem inversa ao seu registo

11.11 Argumentos passados pela linha de comandos e variáveis de ambiente

- **argv:** array de strings que representa um conjunto de parâmetros passados ao programa
 - **argv[0]** é a referência do programa
- **env** é um array de strings onde cada string representa uma variável do tipo: `name=value`
- **getenv** devolve o valor de uma variável definida no array **env**

11.12 Espaço de Endereçamento de um Processo em Linux

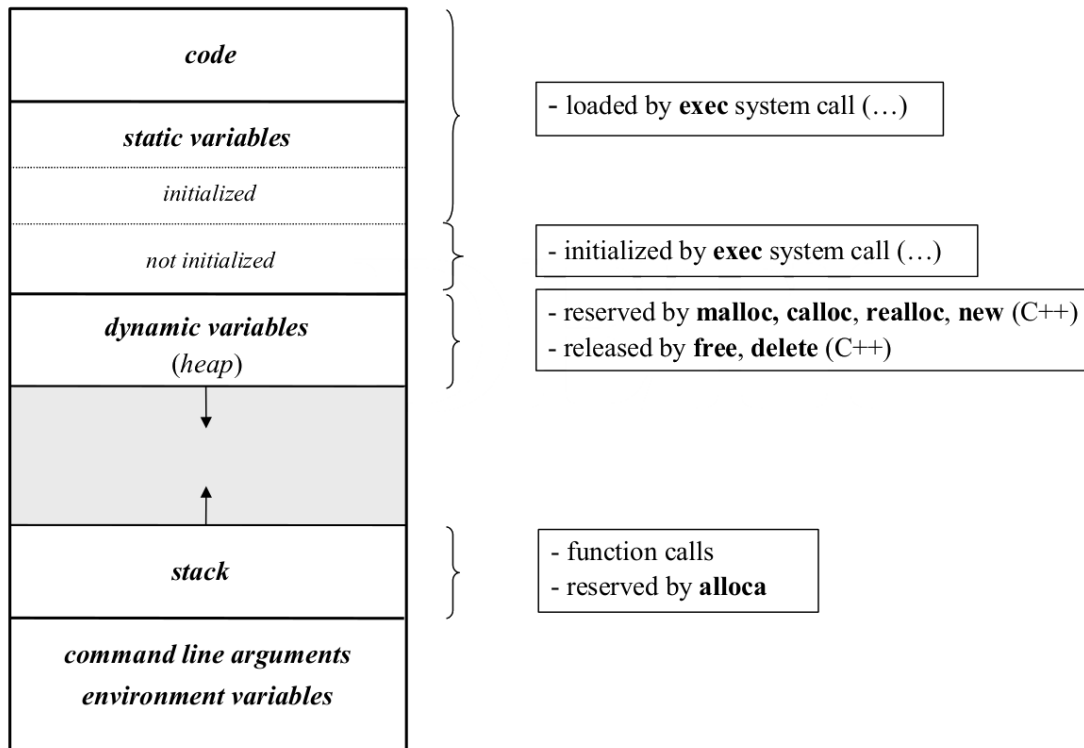


Figure 18: Espaço de endereçamento de um processo em Linux

- As variáveis que existem no processo pai também existem no processo filho (clonadas)
- As variáveis globais são comuns aos dois processos
- Os endereços das variáveis são todos iguais porque o espaço de endereçamento é igual (memória virtual)
- Cada processo tem as suas variáveis, residindo numa página de memória física diferente
- Quando o processo é clonado, o espaço de dados só é clonado quando um processo escreve numa variável, ou seja, após a modificação é que são efetuadas as cópias dos dados
- O programa acede a um endereço de memória virtual e depois existe hardware que trata de alocar esse endereço de memória de virtual num endereço de memória física
- Posso ter dois processos com memórias virtuais distintas mas fisicamente estarem ligados *ao mesmo endereço de memória*
- Quando faço um **fork** não posso assumir que existem variáveis partilhadas entre os processos

11.12.1 Process Control Table

É um array de `process control block`, uma estrutura de dados mantida pelo sistema operativo para guardar a informação relativa todos os processos.

O `process controlo block` é usado para guardar a informação relativa a apenas um processo, possuindo os campos:

- `identification`: id do processo, processo-pai, utilizador e grupo de utilizadores a que pertence
- `address space`: endereço de memória/swap onde se encontra:
 - código
 - dados
 - stack
- `processo state`: valor dos registos internos (incluindo o PC e o `stack pointer`) quando ocorre o switching entre processos
- `I/O context`: canais de comunicação e respetivos buffers que o processo tem associados a si
- `state`: estado de execução, prioridade e eventos
- `statistic`: *start time, CPU time*

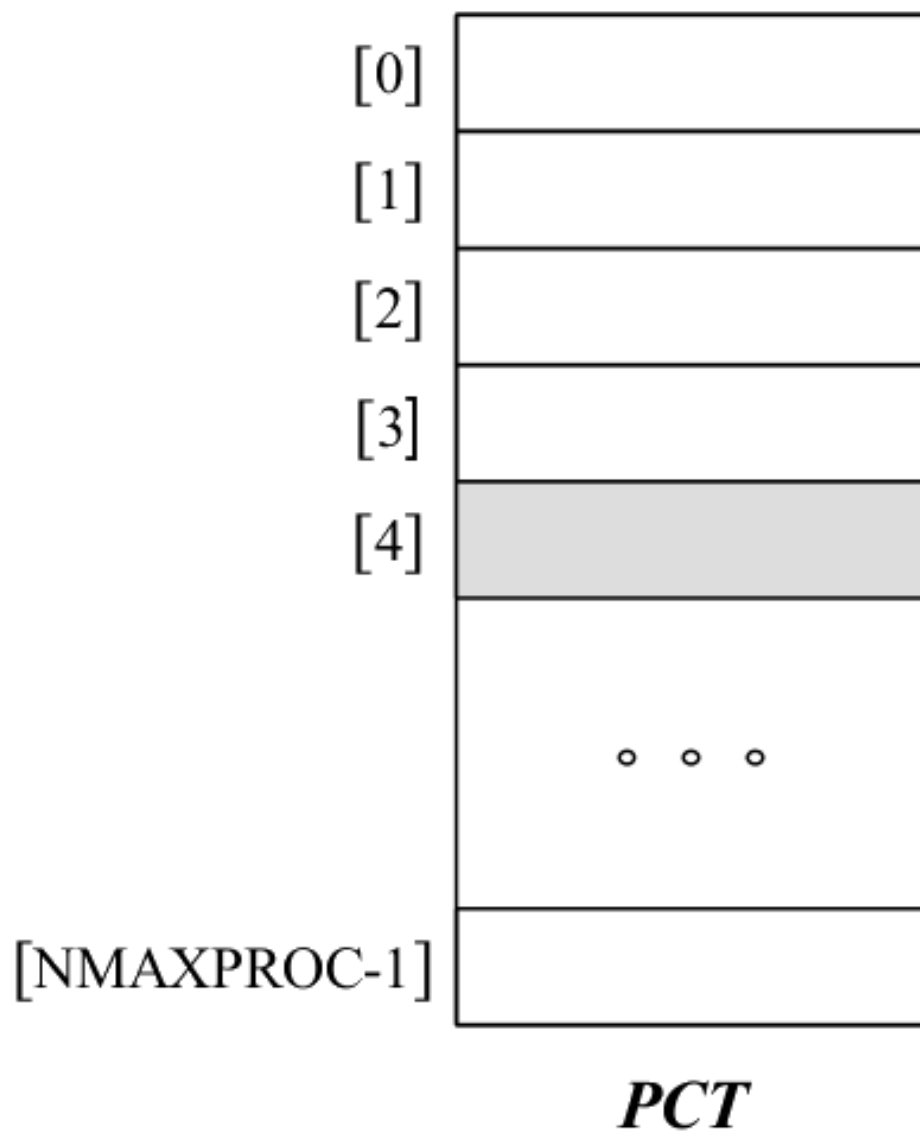


Figure 19: Process Control Table

12 Threads

Num sistema operativo tradicional, um **processo** inclui:

- um **espaço de endereçamento**
- um **conjunto de canais de comunicação** com dispositivos de I/O

- uma única **thread de controlo** que:
 - incorpora os **registos do processador** (incluindo o PC)
 - **stack**

Existem duas stacks no sistema operativo:

- **user stack**: cada **processo/thread** possui a sua (em memória virtual e corre com privilégios do **user**)
- **system stack**: global a todo o sistema operativo (no **kernel**)

Podendo estes dois componentes serem **geridos de forma independente**.

Visto que uma **thread** é apenas um **componente de execução** dentro de um processo, várias **threads independentes** podem coexistir no mesmo processo, **partilhando** o mesmo **espaço de endereçamento** e o mesmo contexto de **acesso aos dispositivos de I/O**. Isto é **multithreading**.

Na prática, as **threads** podem ser vistas como *light weight processes*

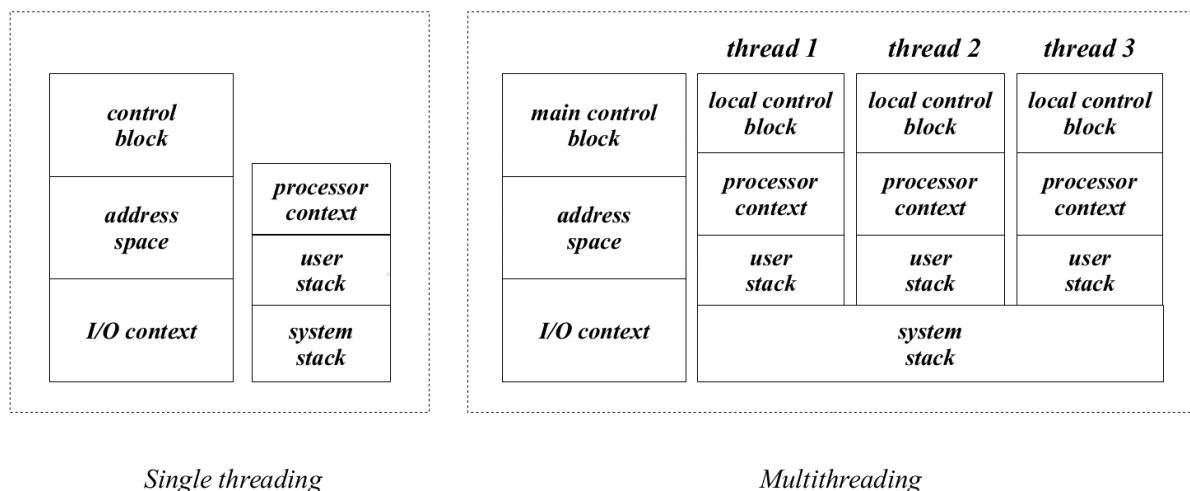


Figure 20: Single threading vs Multithreading

O controlo passa a ser centralizado na **thread** principal que gere o processo. A **user stack**, o **contexto de execução do processador** passa a ser dividido por todas as **threads**.

12.1 Diagrama de Estados de uma thread

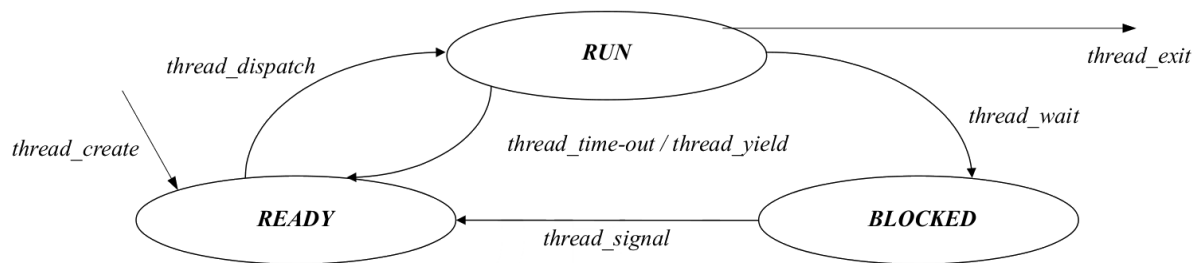


Figure 21: Diagrama de estados de uma thread

O diagrama de estados de um **thread** é mais simplificado do que o de um processo, porque só são “necessários” os estados que interagem **diretamente com o processador**:

- 1 - ‘run’
- 2 - ‘ready’
- 3 - ‘blocked’

Os estados **suspend-ready** e **suspended-blocked** estão relacionados com o **espaço de endereçamento** do **processo** e com a zona onde estes dados estão guardados, dizendo respeito ao **processo e não à thread**

Os estados **new** e **terminated** não estão presentes, porque a gestão do ambiente multiprogramado prende-se com a restrição do número de **threads** que um processo pode ter, logo dizem respeito ao processo

12.2 Vantagens de Multithreading

- **facilita a implementação** (em certas aplicações):
 - Existem aplicações em que **decompor a solução** num conjunto de **threads** que correm paralelamente facilita a implementação
 - Como o **address space** e o **I/O context** são partilhados por todas as **threads**, **multithreading** favorece esta decomposição
- **melhor utilização dos recursos**
 - A criação, destruição e **switching** é mais eficiente usando **threads** em vez de processos
- **melhor performance**
 - Em aplicações **I/O driven**, **multithreading** permite que **várias atividades se sobreponham**, **aumentando a rapidez** da sua execução
- **multiprocessing**
 - É possível **paralelismo em tempo real** se o processador possuir **múltiplos CPUs**

12.3 Estrutura de um programa multithreaded

- Cada `thread` está tipicamente associada com a execução de uma função que implementa alguma atividade em específico
- A **comunicação entre threads** é efetuada através da estrutura de dados do **processo**, que é vista pelas `threads` como uma estrutura de dados global
- o **programa principal** também é uma `thread`
 - A 1ª a ser criada
 - Por norma a última a ser destruída

12.4 Implementação de Multithreading

`user level threads`:

- Implementadas por uma biblioteca
 - Suporta a criação e gestão das `threads` sem intervenção do `kernel`
- Correm com permissões do utilizador
- Solução versátil e portátil
- Quando uma `thread` executa uma `system call` bloqueante, **todo o processo bloqueia** (o `kernel` só “vê” o processo)
- Quando passo variáveis a `threads`, elas têm de ser estáticas ou dinâmicas

`kernel level threads`

- As `threads` são implementadas diretamente ao nível do kernel
- Menos versáteis e portáteis
- Quando uma `thread` executa uma `system call` bloqueante, **outra thread pode entrar em execução**

12.4.1 Biblioteca pthread

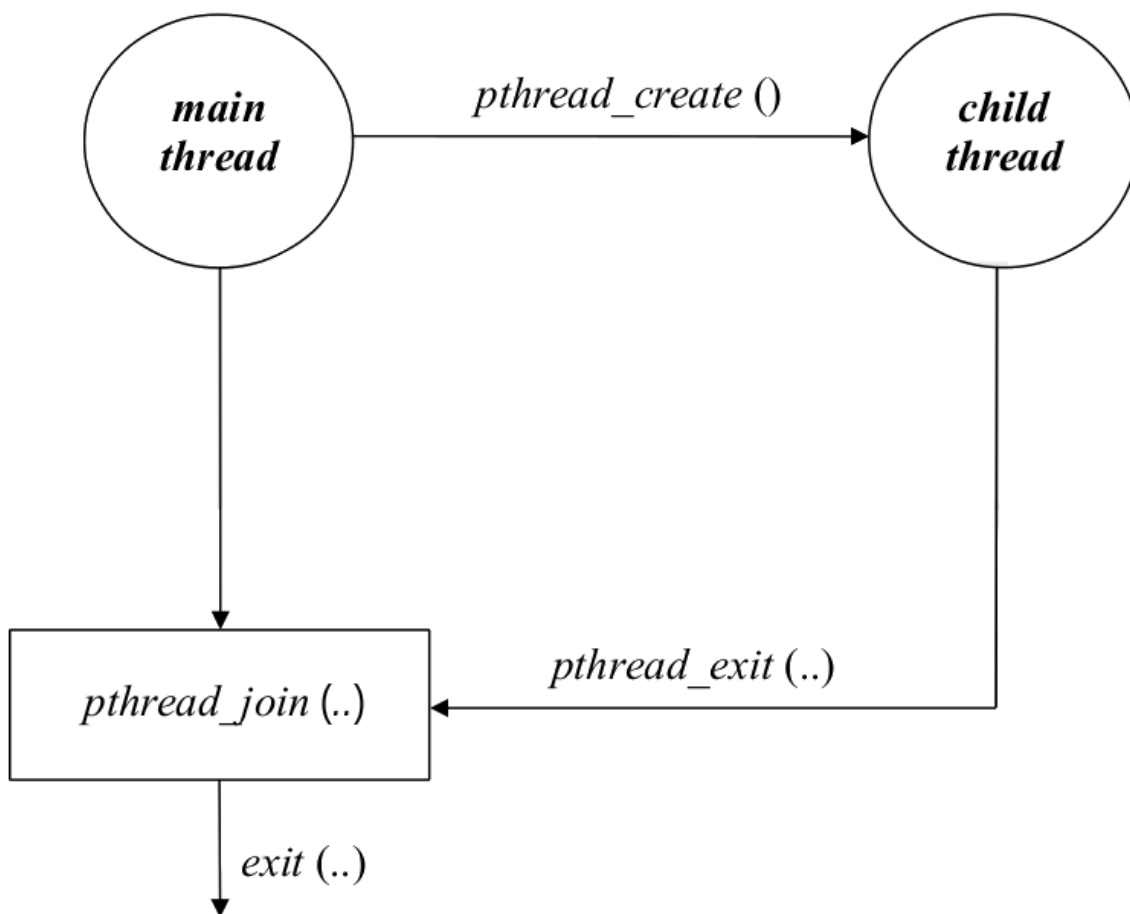


Figure 22: Exemplo do uso da biblioteca pthread

```
1 #include <stdio.h>
2 #include <stdlib.h>
3 #include <pthread.h>
4
5 /* return status */
6 int status;
7
8 /* child thread */
9 void *threadChild (void *par)
10 {
11     printf ("I'm the child thread!\n");
12     status = EXIT_SUCCESS;
13     pthread_exit (&status);
14 }
```

```
15
16 /* main thread */
17 int main (int argc, char *argv[])
18 {
19     /* launching the child thread */
20     pthread_t thr;
21     if (pthread_create (&thr, NULL, threadChild, NULL) != 0)
22     {
23         perror ("Fail launching thread");
24         return EXIT_FAILURE;
25     }
26
27     /* waits for child termination */
28     if (pthread_join (thr, NULL) != 0)
29     {
30         perror ("Fail joining child");
31         return EXIT_FAILURE;
32     }
33
34     printf ("Child ends; status %d.\n", status);
35     return EXIT_SUCCESS;
36 }
```

12.5 Threads em Linux

2 `system calls` para criar processos filhos:

- `fork`:
 - cria um novo processo que é uma **cópia integral** do processo atual
 - o `address space` e `I/O context` é duplicado
- `clone`:
 - cria um novo processo que pode partilhar elementos com o pai
 - Podem ser partilhados
 - * espaço de endereçamento
 - * tabela de `file descriptors`
 - * tabela de `signal handlers`
 - O processo filho executa uma dada função

Do ponto de vista do `kernel`, `processos` e `threads` são **tratados de forma semelhante**

`Threads` do **mesmo processo** formam um `thread group` e possuem o **mesmo thread group identifier** (TGID). Este é o valor retornado pela função `getpid()` quando aplicada a um grupo de `threads`

As várias `threads` podem ser distinguidas dentro de um **grupo de threads** pelo seu `unique thread identifier` (TID). É o valor retornado pela função `gettid()`

```
1 #include <stdio.h>
2 #include <stdlib.h>
3 #include <pthread.h>
4 #include <unistd.h>
5 #include <unistd.h>
6 #include <sys/types.h>
7
8 pid_t gettid()
9 {
10     return syscall(SYS_gettid);
11 }
12
13 /* child thread */
14 int status;
15 void *threadChild (void *par)
16 {
17     /* There is no glibc wrapper, so it was to be called
18      * indirectly through a system call
19      */
20
21     printf ("Child: PPID: %d, PID: %d, TID: %d\n", getppid(), getpid(), gettid
22            ());
23     status = EXIT_SUCCESS;
24     pthread_exit (&status);
25 }
```

O **TID** da *main thread* é a mesma que o **PID** do processo, **porque são a mesma entidade**.

Para efetuar a compilação, tenho de indicar que a biblioteca pthread tem de ser usada na linkagem:

```
1 g++ -o x thread.cpp -pthread
```

13 Process Switching

Revisitando a o diagrama de estados de um processador [multithreading](#)

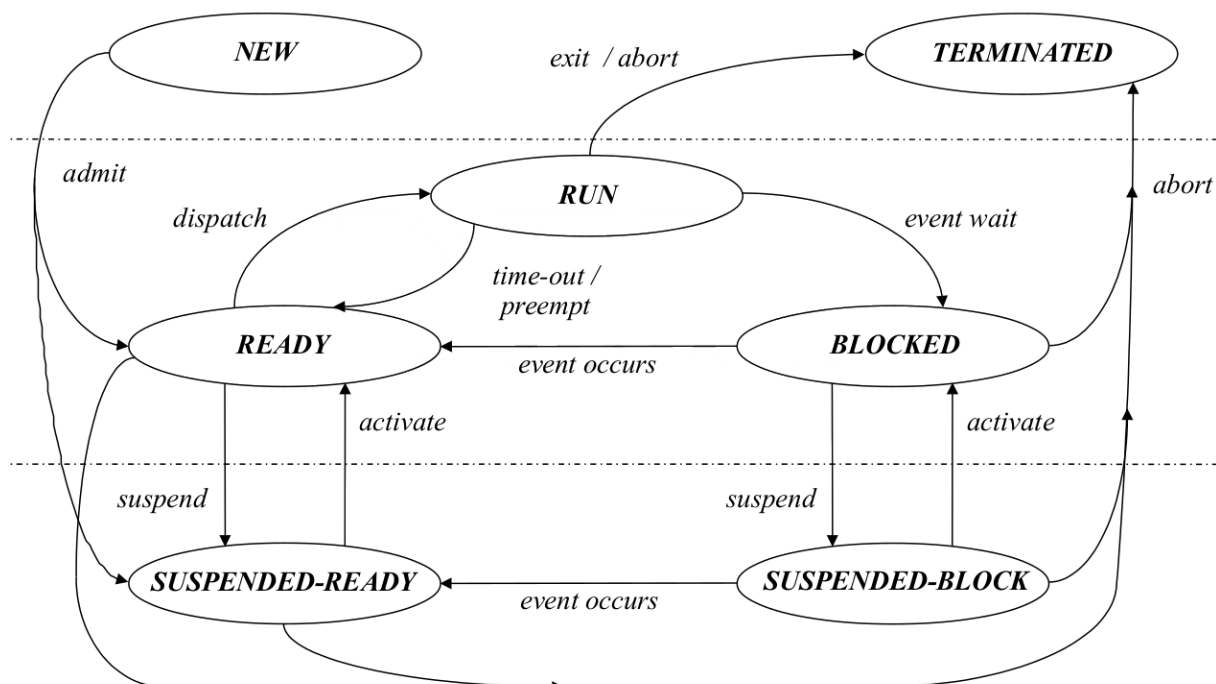


Figure 23: Diagrama de estados completo para um processador `multithreading`

Os processadores atuais possuem **dois modos de funcionamento**:

1. `supervisor mode`

- Todas as instruções podem ser executadas
- É um modo **priviligiado, reservado para o sistema operativo**
- O modo em que o **sistema operativo devia funcionar**, para garantir que pode aceder a todas as funcionalidades do processador

2. `user mode`

- Só uma **secção reduzida do instruction set** é que pode ser executada
- Instruções de I/O e outras que modifiquem os registos de controlo não são executadas em `user mode`
- É o **modo normal de operação**

A **troca entre os dois modos de operação**, `switching`, só é possível através de **exceções**. Uma **exceção** pode ser causada por:

- Interrupção de um dispositivo de I/O
- Instrução ilegal
 - divisão por zero
 - bus error
 - ...

- trap instruction (aka interrupção por *software*)

As **funções do kernel**, incluindo as *system calls* só podem ser lançadas por:

- **hardware** \Rightarrow *interrupção*
- **traps** \Rightarrow *interrupção por software*

O ambiente de operação nestas condições é denominado de *exception handling*

13.1 Exception Handling

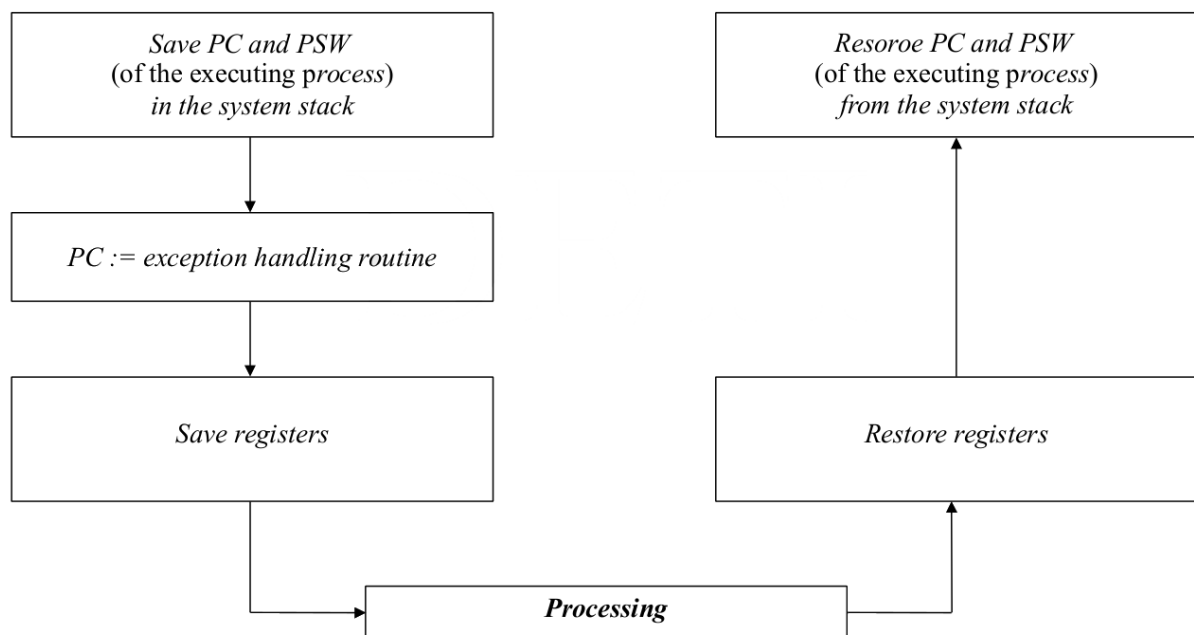


Figure 24: Algoritmo a seguir para tratar de exceções normais

A **troca do contexto de execução** é feita guardando o estado dos registos PC e PSW na stack do sistema, saltando para a rotina de interrupção e em seguida salvaguardando os registos que a rotina de tratamento da exceção vai precisar de modificar. No fim, os valores dos registos são restaurados e o programa resume a sua execução

13.2 Processing a process switching

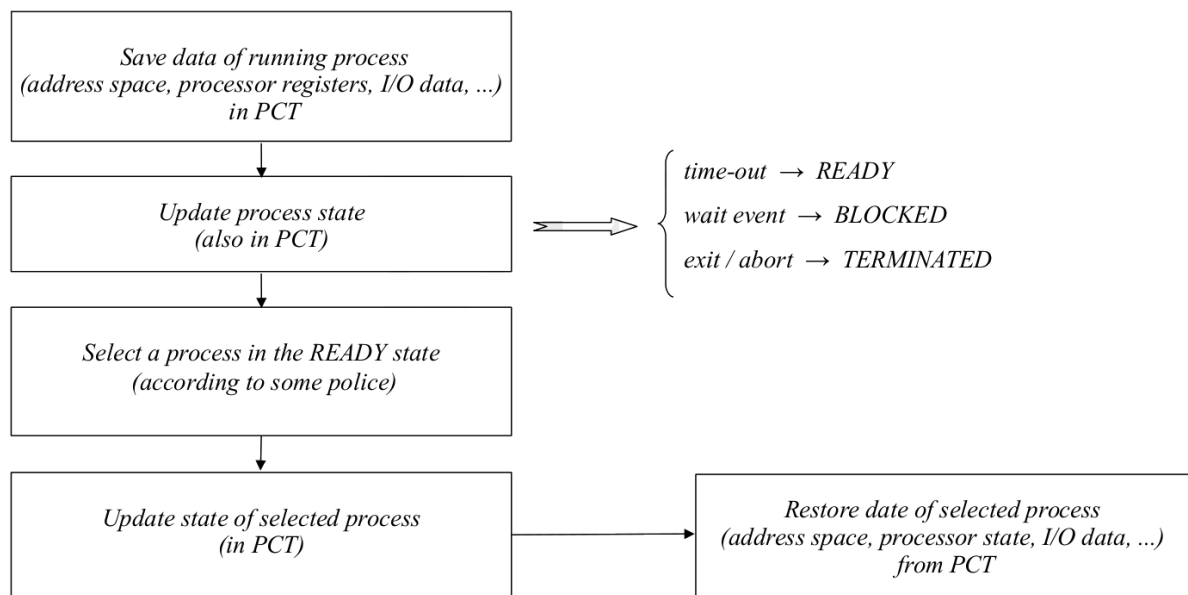


Figure 25: Algoritmo a seguir para efetuar uma process switching

O algoritmo é bastante parecido com o tratamento de exceções:

1. Salvar todos os dados relacionados com o processo atual
2. Efetuar a troca para um novo processo
3. Correr esse novo processo
4. Restaurar os dados e a execução do processo anterior

14 Processor Scheduling

A execução de um processo é uma sequência alternada de períodos de:

- **CPU burst**, causado pela execução de instruções do CPU
- **I/O burst**, causados pela espera do resultado de pedidos a dispositivos de I/O

O processo pode então ser classificado como:

- **I/O bound** se possuir muitos e curtos **CPU bursts**
- **CPU bound** se possuir poucos e longos **CPU bursts**

O objetivo da **multiprogramação** é obter vantagem dos períodos de **I/O burst** para permitir outros processos terem acesso ao processador. A componente do sistema responsável por esta gestão é o **scheduler**.

A funcionalidade principal do **scheduler** é decidir da **poll** de processos prontos para serem executados que coexistem no sistema:

- quais é que devem ser executados?
- quando?
- por quanto tempo?
- porque ordem?

14.1 Scheduler

Revisitando o diagrama de estados do processador, identificamos três *schedulers*

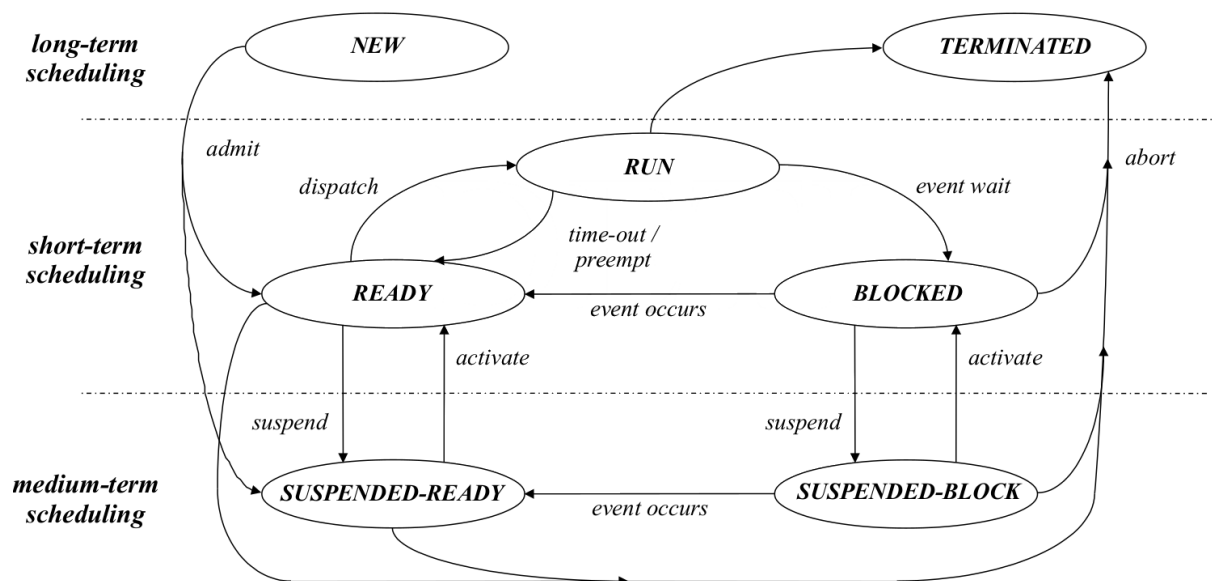


Figure 26: Identificação dos diferentes tipos de schedulers no diagrama de estados dos processos

14.1.1 Long-Term Scheduling

Determina que **programas são admitidos para serem processados**:

- Controla o **grau de multiprogramação** do sistema
- Se um programa do utilizador ou **job** for aceite, torna-se um **processo** e é adicionado à **queue de processos ready em fila de espera**
 - Em princípio é adicionado à **queue** do **short-term scheduler**
 - mas também é possível que seja adicionada à **queue** do **medium-term scheduler**
- Pode colocar processos em **suspended ready**, libertando quer a memória quer a fila de processos

14.1.2 Medium Term Scheduling

Gere a **swapping area**

- As decisões de `swap-in` são **controladas pelo grau de multiprogramação**
- As decisões de `swap-in` são **condicionadas pela gestão de memória**

14.1.3 Short-Term Scheduling

Decide qual o **próximo processo a executar**

- É invocado quando existe um evento que:
 - **bloqueia o processo atual**
 - **permite que este seja preempted**
- Eventos possíveis são:
 - interrupção de relógio
 - interrupção de I/O
 - `system calls`
 - signal (e.g. através de semáforos)

14.2 Critérios de Scheduling

14.2.1 User oriented

Turnaround Time:

- Intervalo de tempo entre a submissão de um processo até à sua conclusão
- Inclui:
 - Tempo de execução enquanto o processo tem a posse do CPU
 - Tempo dispendido à espera pelos recursos que precisa (inclui o processador)
- Deve ser minimizado em sistemas `batch`
- É a medida apropriada para um `batch job`

Waiting Time:

- Soma de todos os períodos de tempo em que o processo esteve à espera de ser colocado no estado `ready`
- Deve ser minimizado

Response Time:

- Intervalo de tempo que decorre desde a submissão de um pedido até a resposta começa a ser produzida
- Medida apropriada para sistemas/processos interativo
- Deve ser minimizada para este tipo de sistemas/processos
- O número de processos interativos deve ser maximizado desde que seja garantido um tempo de resposta aceitável

Deadlines:

- Tempo necessário para um processo terminar a sua execução

- Usado em sistemas de tempo real
- A percentagem de **deadlines** atingidas deve ser maximizada, mesmo que isso implique subordinar/reduzir a importância de outros objetivos/parâmetros do sistema

Predictability:

- Quantiza o impacto da carga (de processos) no tempo de resposta do sistema
- Idealmente, um **job** deve correr no **mesmo intervalo de tempo** e gastar os **mesmos recursos de sistema** independentemente da carga que o sistema possui

14.2.2 System oriented**Fairness:**

- Igualdade de tratamento entre todos os processos
- Na ausência de diretivas que condicionem os processos a atender, deve ser efetuada uma gestão e partilha justa dos recursos, onde todos os processos são tratados de forma equitativa
- Nenhum processo pode sofrer de **starvation**

Throughput:

- Medida do número de processos completados por unidade de tempo (“taxa de transferência” de processos)
- Mede a quantidade de trabalho a ser executada pelos processos
- Deve ser maximizado
- Depende do tamanho dos processos e da **política de escalonamento**

Processor Utilization:

- Mede a percentagem que o processador está ocupado
- Deve ser maximizada, especialmente em sistemas onde predomina a partilha do processador

Enforcing Priorities:

- Os processos de **maior prioridade** devem ser sempre favorecidos em detrimento de processos menos prioritários

É impossível favorecer todos os critérios em simultâneo

Os **critérios a favorecer** dependem da **aplicação específica**

14.3 Preemption & Non-Preemption**Non-preemptive scheduling:**

- O processo mantém o processador até este ser bloqueado ou terminar
- As **transições são sempre por time-out**
- Não existe **preempt**

- Típico de sistemas **batch**
 - Não existem deadlines nem limitações temporais restritas a cumprir

Preemptive Scheduling:

- O processo pode **perder o processador devido a eventos externos**
 - esgotou o seu **time-quantum**
 - um processo **mais prioritário** está **ready**
 - Típico de **sistemas interativos**
 - * É preciso garantir que a resposta ocorre em intervalos de tempo limitados
 - * É preciso “simular” a ideia de paralelismo no **macro-tempo**
 - Sistemas em tempo real são **preemptive** porque existem **deadlines restritas** que precisam de ser cumpridas
 - Nestas situações é importante que um **evento externo** tenha capacidade de libertar o processador

14.4 Scheduling

14.4.1 Favouring Fearness

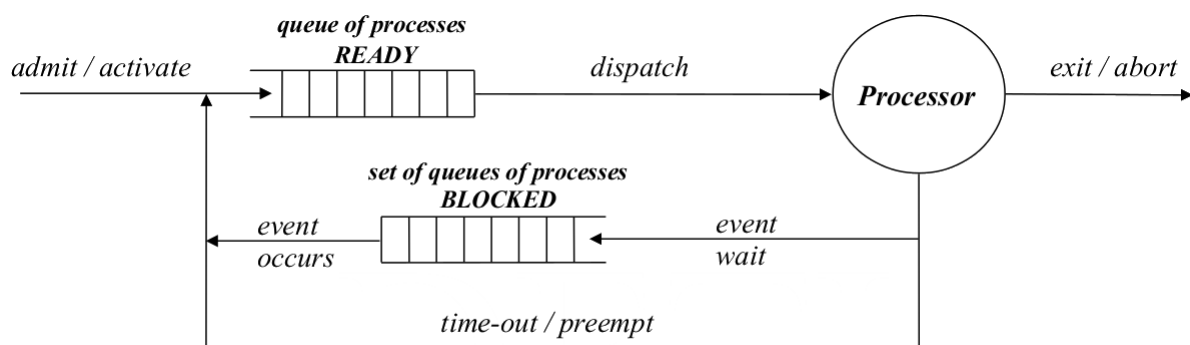


Figure 27: Espaço de endereçamento de um processo em Linux

Todos os processos **são iguais** e são atendidos por **ordem de chegada**

- É implementado usando **FIFOs**
- Pode existir mais do que um processo à espera de eventos externos
- Existe uma fila de espera para cada evento
- Fácil de implementar
- Favorece processos **CPU-bound** em detrimento de processos **I/O-bound**
 - Só necessitam de acesso ao processador, não de recursos externos
 - Se for a vez de um processo **I/O-bound** ser atendido e não possuir os recursos de I/O que precisa tem de voltar para a fila

- Em **sistemas interativos**, o **time-quantum** deve ser escolhido cuidadosamente para obter um bom compromisso entre **fairness** e **response time**

Em função do scheduling pode ser definido como:

- **non-preemptive scheduling** \Rightarrow **first come, first-served** (FCFS)
- **preemptive scheduling** \Rightarrow **round robin**

14.4.2 Priorities

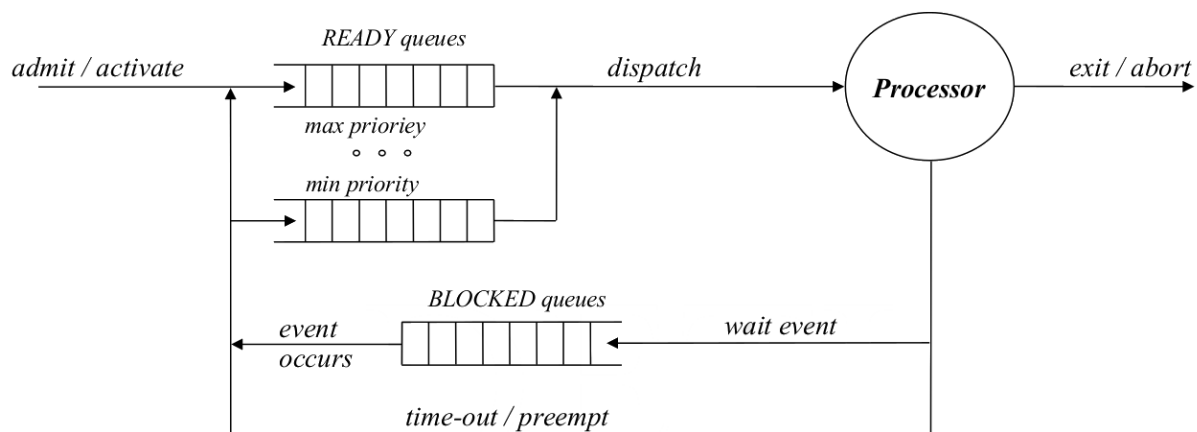


Figure 28: Espaço de endereçamento de um processo em Linux

Segue o princípio de que atribuir a mesma importância a todos os processos pode ser uma solução errada. Um sistema injusto *per se* não é necessariamente mau.

- A **minimização do tempo de resposta** (**response time**) exige que os processos **I/O-bound** sejam **privilegiados**
- Em **sistemas de tempo real**, os processos associados a **eventos/alarmes** e **ações do sistema operativo** sofrem de várias **limitações e exigências temporais**

Para resolver este problema os processos são **agrupados** em grupos de **diferentes prioridades**

- Processos de maior prioridade são executados primeiros
- Processos de menor prioridade podem sofrer **starvation**

Prioridades Estáticas

As prioridades a atribuir a cada processo são determinadas *a priori* de forma **determinística**

- Os processos são **agrupados em classes de prioridade fixa**, de acordo com a sua importância relativa
- Existe risco de os processos menos prioritários sofrerem **starvation**

- Mas se um **processo de baixa prioridade não é executado** é porque o sistema foi **mal dimensionado**
- É o sistema de `scheduling` mais injusto
- É usado em sistemas de tempo real, para garantir que os processos que são críticos são sempre executados

Alternativamente, pode se fazer:

1. Quando um processo é criado, é-lhe **atribuído um dado nível de prioridade**
2. Em `time-out` a prioridade do processo é **decrementada**
3. Na ocorrência de um `wait event` a prioridade é **incrementada**
4. Quando o valor de **prioridade atinge um mínimo**, o valor da prioridade sofre um `reset`
 - É colocada no valor inicial, garantindo que o processo é executado

Previnem-se as situações de `starvation` impedindo que o processo não acaba por ficar com uma prioridade tão baixa que nunca mais consegue ganhar acesso

Prioridades Dinâmicas

- As classes de prioridades estão definidas de forma funcional *a priori*
- A mudança de um processo de classe é efetuada com base na utilização última janela de execução temporal que foi atribuída ao processo

Por exemplo:

- **Prioridade 1:** `terminais`
 - Um processo entra nesta categoria quando se efetua a transição `event occurs` (evento de escrita/leitura de um periférico) quando estava à espera de dados do `standard input device`
- **Prioridade 2:** `generic I/O`
 - Um processo entra nesta categoria quando efetua a transição `event occurs` se estava à espera de dados de **outro tipo de input device** que não o `stdin`
- **Prioridade 3:** `small time quantum`
 - Um processo entra nesta classe quando ocorre um `time-out`
- **Prioridade 4:** `large time quantum`
 - Um processo entra nesta classe após um sucessivo número de `time-outs`
 - São claramente processos `CPU-bound` e o objetivo é atribuir-lhes janelas de execução com grande `time quantum`, mas menos vezes

Shortest job first (SJF) / Shortest process next (SPN)

Em sistemas **batch**, o **turnaround time** deve ser minimizado.

Se forem conhecidas **estimativas do tempo de execução a priori**, é possível estabelecer uma **ordem de execução** dos processos que **minimizam o tempo de turnaround médio** para um dado grupo de processos

Assumindo que temos N **jobs** e que o tempo de execução de cada um deles é te_n , com $n = 1, 2, \dots, N$. O **average turnaround time** é:

$$t_m = te_1 + \frac{N-1}{N} \cdot te_2 + \dots + \frac{1}{N} \cdot te_N$$

onde t_m é o **turnaround time** mínimo se os **jobs** forem sorteados por ordem ascendente de tempo de execução (estimado)

Para **sistemas interativos**, podemos usar um sistema semelhante:

- Estimamos a taxa de ocupação da próxima janela de execução baseada na taxa de ocupação das janelas temporais passadas
- Atribuímos o processador ao processo cuja estimativa for a **mais baixa**

Considerando fe_1 como sendo a **estimativa da taxa de ocupação** da primeira janela temporal atribuída a um processo e f_1 a fração de tempo efetivamente ocupada:

- A estimativa da segunda fração de tempo necessária é

$$fe_2 = a \cdot fe_1 + (1 - a) \cdot f_1$$

- A estimativa da e-nésima fração de tempo necessária é:

$$fe_N = a \cdot fe_{N-1} + (1 - a) \cdot f_{N-1}$$

Ou alternativamente:

$$a^{N-1} \cdot fe_1 + a^{N-2} \cdot (1 - a) \cdot fe_2 + a \cdot (1 - a) \cdot fe_{N-2} + (1 - a) \cdot fe_{N-1}$$

Com $a \in [0, 1]$, onde a é um coeficiente que representa o peso que a história passada de execução do processo influencia a estimativa do presente

Esta alternativa levanta o problema que processos **CPU-bound** podem sofrer de **starvation**. Este problema pode ser resolvido contabilizando o tempo que um processo está em espera (**aging**) enquanto está na fila de processos **ready**

Normalizando esse tempo em função do período de execução e denominando-o R , a **prioridade** de um processo pode ser dada por:

$$p = \frac{1 + b \cdot R}{fe_N}$$

onde b é o coeficiente que **controla o peso do aging** na fila de espera dos processos **ready**

14.5 Scheduling Policies

14.5.1 First Come, First Serve (FCFS)

Também conhecido como **First In First Out** (FIFO). O processo mais antigo na fila de espera dos processos **ready** é o primeiro a ser selecionado.

- **Non-preemptive** (em sentido estrito), podendo ser combinado com um esquema de prioridades baixo
- Favorece processos **CPU-bound** em detrimento de processos **I/O-bound**
- Pode resultar num **mau uso** do processador e dos dispositivos de I/O
- Pode ser utilizado com **low priority schemas**

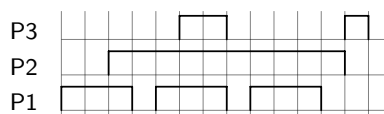


Figure 29: Problema de Scheduling

Usando uma política de **first come first serve**, o resultado do scheduling do processador é:

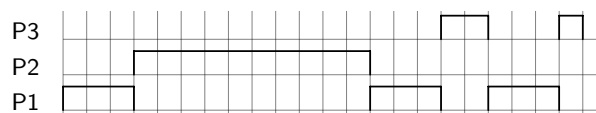


Figure 30: Política FCFS

- O P1 começa a usar o CPU.
- Como é um sistema FCFS, o processo 1 só larga o CPU passado 3 ciclos.
- O processo P2 é o processo seguinte na fila **ready**, e ocupa o CPU durante 10 ciclos.
- Quando P2 termina, P1 é o processo que está à mais tempo à espera, sendo ele que é executado
- Quando P2 abandona voluntariamente o CPU, o processo P1 corre os seus primeiros dois ciclos
- Quando P3 liberta o CPU, o processo P1 termina os últimos 3 ciclos que precisa
- Quando P3 liberta o CPU, o processo P1 como é **I/O-bound** e precisa de 5 ciclos para o dispositivo estar pronto fica mais dois ciclos à espera para poder terminar executando o seu último ciclo

14.5.2 Round-Robin

- **Preemptive**
 - O **scheduler** efetua a gestão baseado num **clock**
 - A cada processo é atribuído um **time-quantum** máximo antes de ser **preempted**
- O processo **mais antigo** em **ready** é o **primeiro a ser selecionado**
 - não são consideradas prioridades
- Efetivo em sistemas **time sharing** com objetivos globais e sistemas que processem transações

- **Favorece** CPU-bound em detrimento de processos I/O-bound
- Pode resultar num **mau uso de dispositivos I/O**

Na escolha/otimização do `time quantum` existe um **tradeoff**:

- **tempos muito curtos** favorecem a execução de **processos pequenos**
 - estes processos vão ser executados **rapidamente**
- **tempos muito curtos** obrigam a `processing overheads` devido ao `process switching` **intensivo**

Para os processos apresentados acima, o diagrama temporal de utilização do processador, para um `time-quantum` de 3 ciclos é:

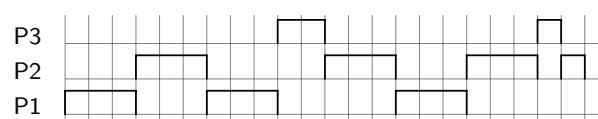


Figure 31: Política Round-Robin

A história de processos em `ready` em fila de espera é: 2, 1, 3, 2, 1, 2, 3, 1

14.5.3 Shortest Process Next (SPN) ou Shortest Job First (SJF)

- `Non-preemptive`
- O process com o `shortest CPU burst time` (menor tempo espectável de utilização do CPU) é o **próximo a ser selecionado**
 - Se vários processos tem o **mesmo tempo de execução** é usado FCFS para desempatar
- Existe um **risco de starvation** para grandes processos
 - o seu acesso ao CPU pode ser **sucessivamente adiado** se existir “forem existindo” processos com **tempo de execução menor**
- Normalmente é usado em escalonamento de longo prazo, `long-term scheduling` em sistemas `batch`, porque os utilizadores esperam estimar com precisão o tempo máximo que o processo necessita para ser executado

14.5.4 Linux

No Linux existem 3 classes de prioridades:

1. **FIFO**, `SCHED_FIFO`
 - `real-time threads`, com política de prioridades
 - uma `thread` em execução é `preempted` apenas se um processo de **mais alta prioridade da mesma classe** transita para o estado `ready`

- uma `thread` em execução pode **voluntariamente abandonar o processador**, executando a primitiva `sched_yield`
- dentro da mesma classe de prioridade a política escolhida é `First Come, First Serve` (FCFS)
- Só o `root` é que pode lançar processos em modo FIFO

2. Round-Robin real time threads, `SCHED_RR`

- `threads` com prioridades com necessidades de execução em tempo real
- Processos nesta classe de prioridades são `preempted` se o seu `time-quantum` termina

3. Non real time threads, `SCHED_OTHER`

- Só são executadas se não existir nenhuma `thread` com necessidades de execução em tempo real
- Está associada à processos do utilizador
- A política de escalonamento tem mudado à medida que a são lançadas novas versões do *kernel*

A escala de prioridades varia

- 0 a 99 para `real-time threads`
- 100 a 139 para as restantes

Para lançar uma `thread` (sem necessidades de execução em tempo real) com diferentes prioridades, pode ser usado comando `nice`.

Por *default*, o comando lança uma `thread` com prioridade 120. O comando aceita um `offset` de [-20, +19] para obter a prioridade mínima ou máxima.

Algoritmo Tradicional

- Na classe `SCHED_OTHER` as prioridades são baseadas em **créditos**
- Os créditos do processo em execução são **decrementados** à medida que ocorre uma interrupção do `real time clock`
- O processo é `preempted` quando são atingidos zero créditos
- Quando todos os processos `ready` têm zero créditos, os créditos de **todos os processos** (incluindo os que estão bloqueados) são **recalculados** segundo a fórmula:

$$CPU_j(i) = \frac{CPU_j(i-1)}{2} + PBase_j + nice_j$$

onde são tido em conta a **história passada de execução do processo** e as **prioridades**

- O `response time` de processos `I/O-bound` é minimizado
- A `starvation` de processos `CPU-bound` é evitada
- Solução **não adequada para múltiplos processadores** e é má se o número de processos é elevado

14.6 Novo Algoritmo

- Os processos na classe `SCHED_OTHER` passam a usar um `completely fair scheduler` (CFS)
- O scheduling é baseado no `vruntime`, *virtual run time*, que mede durante quanto tempo uma `thread` esteve em execução

- o `virtual run time` está relacionado quer com o **tempo de execução real** (`physical run time`) e a **prioridade** da `thread`
- Quanto maior a prioridade de um processo, menor o `physical run time`
- O `scheduler` seleciona as `threads` com menor `virtual run time`
 - Uma `thread` com prioridade mais elevada que fique pronta a ser executada pode “forçar” um `preempt` uma `thread` com menor prioridade
 - * Assim é possível que uma `thread I/O bound` “forçar” o processador a `preempt` um processo `CPU-bound`
- O algoritmo é implementado com base numa `red-black tree` do processador