Sebenta de SO - Conceitos Teóricos

Conceitos Introdutórios em sistemas de Operação, Process Management, Interprocess Communicattion & Memory Management

PEDRO MARTINS

February 1, 2018

Contents

List of Tables

List of Figures

1 Sistemas de Computação

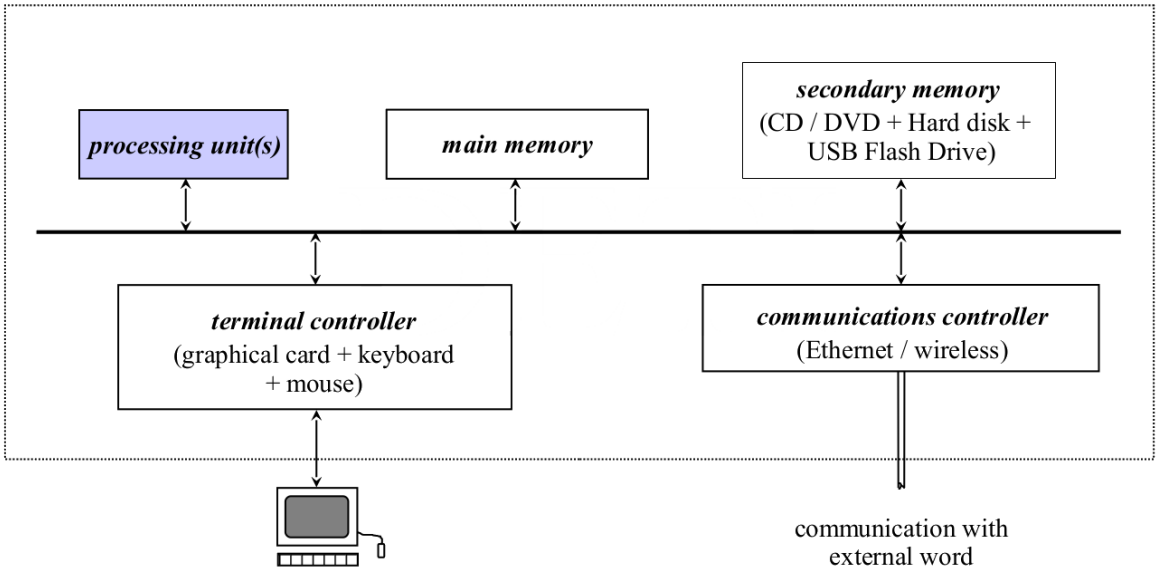


Figure 1: Esquema típico de um sistema de computação

1.1 Vista simplificada de um sistema de computação

Um sistema operativo é um sistema/programa base que é executado pelo sistema computacional.

- Controla diretamente o `hardware`
- Providencia uma **camada de abstração** para que os restantes **programas** possam **interagir com o `hardware`** de forma indireta

Podem ser classificados em dois tipos:

1. **gráficos:**
    - utilizam um contexto de **janelas** num ambiente gráfico
    - os elementos principais de interação são os **ícones** e os **menus**
    - a principal ferramente de `input` da interação humana é o rato
2. **textuais *(shell)*:**

- baseado em comandos introduzidos através do teclado
- uma linguagem de scripting/comandos[1]

Os dois tipos não são mutualmente exclusivas.

- Windows: sistema operativo gráfico que pode lançar uma aplicação para ambiente textual
- Linux: sistema operativo textual que pode lançar ambiente gráfico

1.2 Vista geral

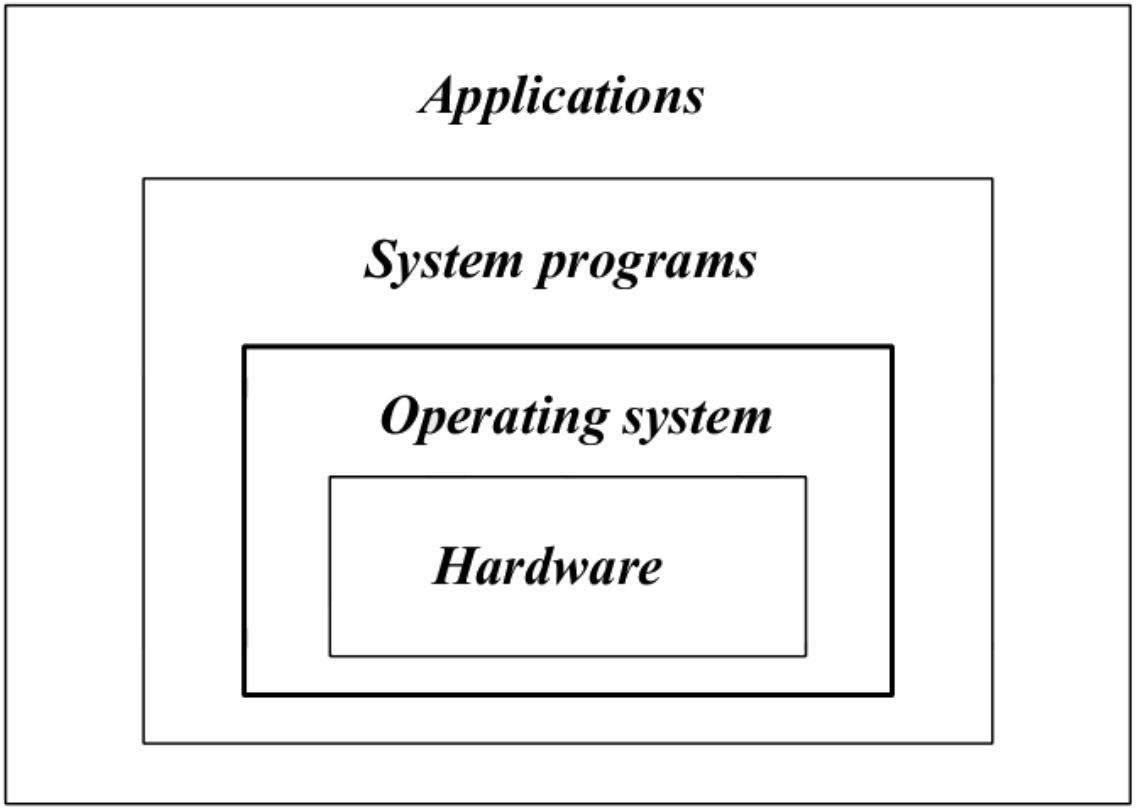


Figure 2: Diagrama em camadas de um sistema de operação

Os sistemas de operação podem ser vistos segundo duas perspectivas:

1. `Extended Machines`
2. `Resource Manager`

[1] ficheiro em código fonte de compilação separada

1.2.1 Extended Machine

O sistema operativo fornece **níveis de abstração** (APIs) para que os programas possam aceder a partes físicas do sistema, criando uma **"máquina virtual":**

- Os programas e programadores têm uma visão virtual do computador, um **modelo funcional**
  - Liberta os programadores de serem obrigados a saber os detalhes do hardware
- Acesso a componentes do sistema mediado através de `system calls`
  - Executa o core da sua função em root (com permissões de super user)
  - Existem funções que só podem correr em super user
  - Todas as chamadas ao sistema são interrupções
  - Interface uniforme com o `hardware`
  - Permite as aplicações serem **portáteis** entre sistemas de computação **estruturalmente diferentes**
- O sistema operativo controla o **espaço de endereçamento físico** criando uma camada de abstração **(memória virtual)**

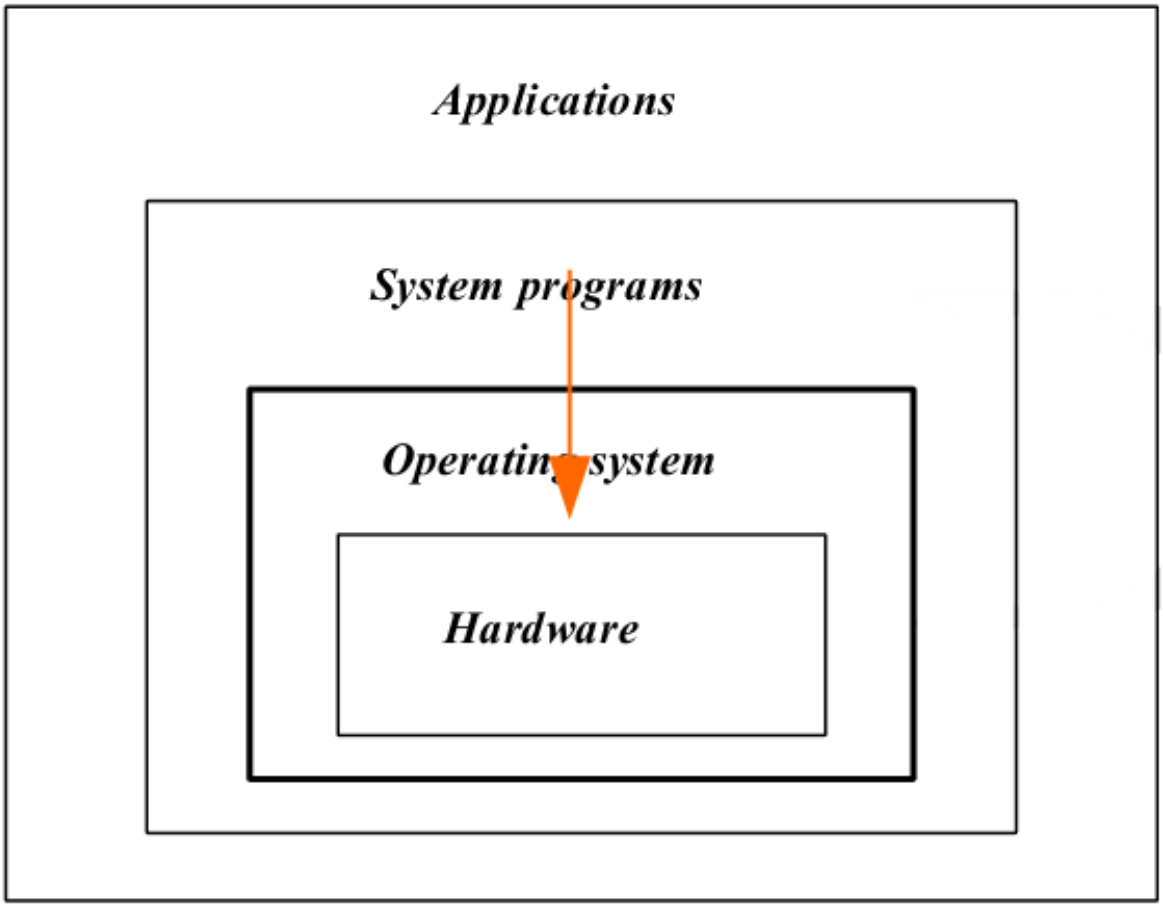


Figure 3: Visão de um sistema operativo do tipo Extended Machine

Tipos de funções da extended machine

- Criar um ambiente interativo que sirva de interface máquina-utilizador
- Disponibilizar mecanismos para desenvolver, testar e validar programas
- Disponibilizar mecanismos que controlem e monitorizem a execução de programas, incluindo a sua intercomunicação e e sincronização
- Isolar os espaços de endereçamento de cada programa e gerir o espaço de cada um deles tendo em conta as limitações físicas da memória principal do sistema
- Organizar a memória secundária [2] em sistema de ficheiros
- Definir um modelo geral de acesso aos dispositivos de I/O, independentemente das suas características individuais
- Detetar situações de erros e estabelecer protocolos para lidar com essas situações

1.2.2 Resource Manager

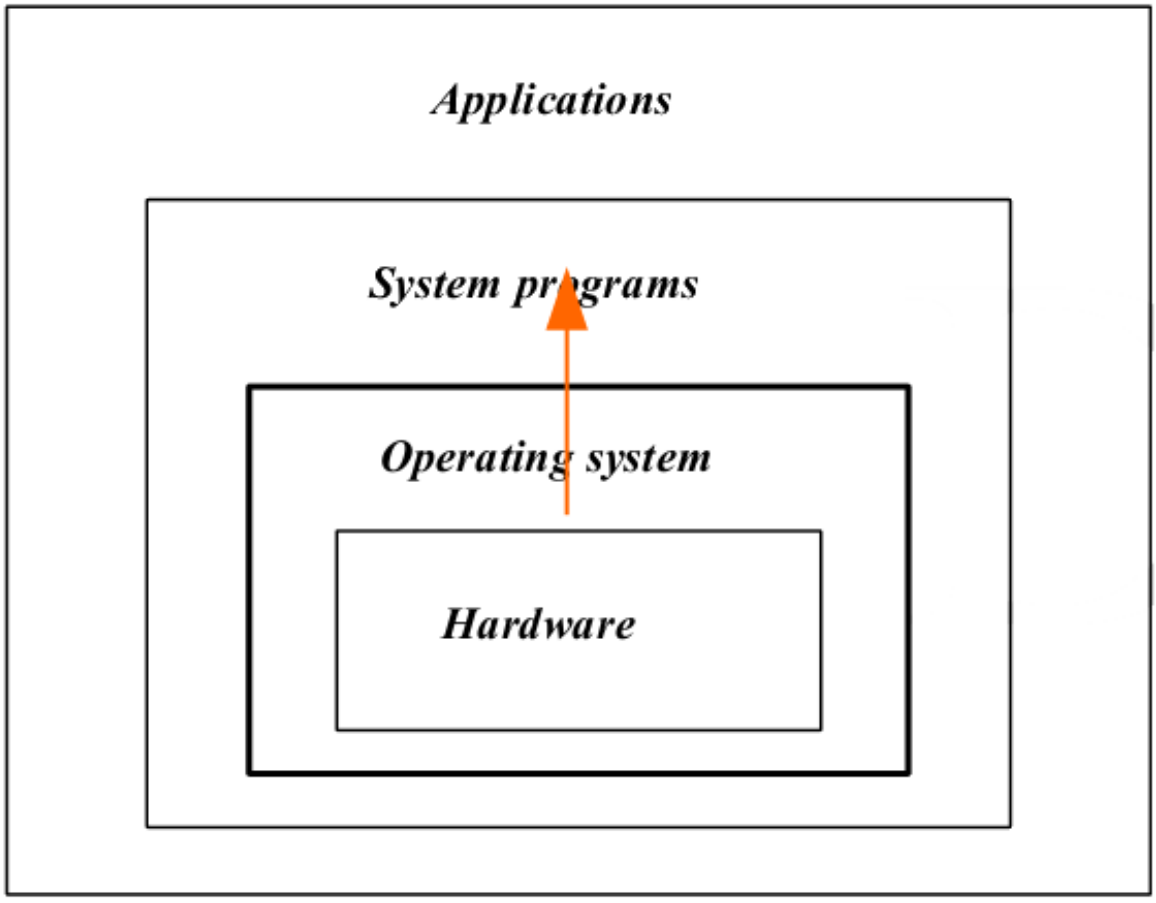


Figure 4: Visão de um sistema operativo do tipo Extended Machine

[2] De um para um

Sistema computacional composto por um conjunto de recursos:

- processador(es)
- memória
  - principal
  - secundária
- dispositivos de I/O e respetivos controladores

O sistema operativo é visto como um programa que gere todos estes recursos, efetuando uma gestão controlada e ordenada dos recursos pelos diferentes programas que tentam aceder a estes. O seu objetivo é **maximizar a performance** do sistema, tentando garantir a maior eficiência no uso dos recursos, que são **multiplexados no tempo e no espaço**.

1.3 Evolução dos Sistemas Operativos

Primórdios : Sistema Electromecânico

- 1ª Geração: 1945 - 1955
  - Vacuum tubes
  - electromechanical relays -No operating system -programed in system
  - Program has full control of the machine
  - Cartões perfurada (ENIAC)
- 2ª geração: Transistores individuais
- 4ª Geração (1980 - presente)

Technology	Notes
LSI/VLSI	Standard Operation systems (MS-DOS, Macintosh, Windows, Unix)
personal computers (microcomputers)	Network operation systems
network	

- 5ª Geração (1990 - presente)

Technology	Notes
Broadband, wireless	mobile operation systems (Symbian, iOS, Android)
system on chip	cloud computing
smartphone	ubiquitous computing

2 Taxonomia de Sistemas Operativos

2.1 Classificação com base no tipo de processamento

- Processamento em série
- Batch Processing
  - Single
  - Multiprogrammed batch
- Time-sharing System
- Real-time system
- Network system
- Distributed System

2.1.1 Multiprogrammed batch

- **Propósito:** Otimizar a utilização do processador
- **Método de Otimização:** Enquanto um programa está à espera pela conclusão de uma operação de I/O, outro programa usa o processador

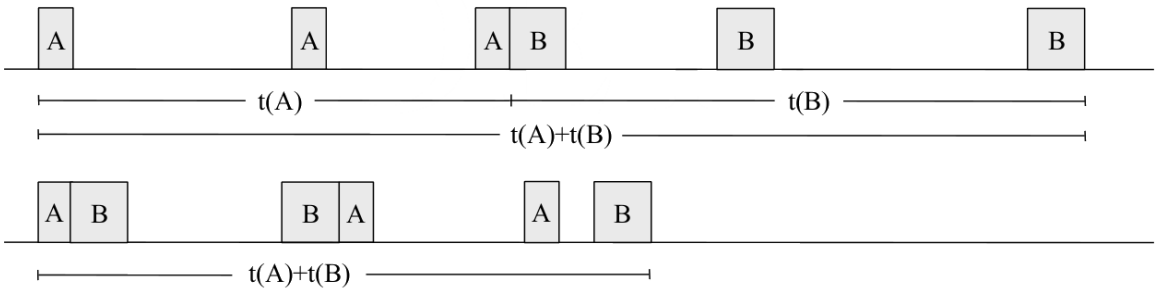


Figure 5: Multiprogrammed batch

2.1.2 Interactive System (Time-Sharing)

- **Propósito:**
  - Proporcionar uma interface `user-friendly`
  - Minimizar o tempo de resposta a pedidos externos
- **Método:**
  - Vários utilizadores mas cada um no seu terminal
  - Todos os terminais têm comunicação direta e em simultâneo com o sistema

- Usando multiprogramação, o uso do processador é multiplexado no tempo, sendo atribuído um time-quantum a cada utilizador
- No `macrotempo` é criada a ilusão ao utilizador que possui o sistema só para si

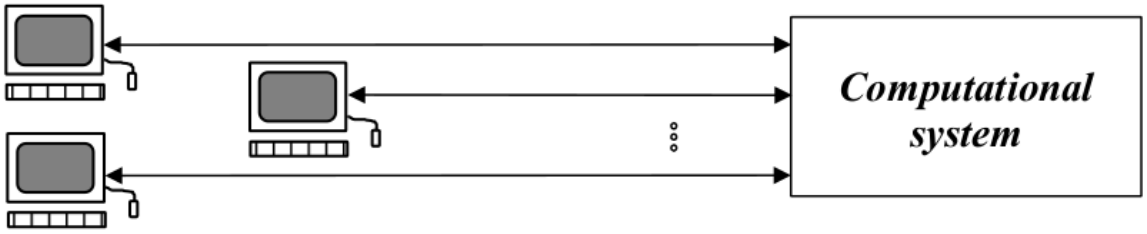


Figure 6: Interactive system (Time-Sharing)

2.1.3 Real Time System

- **Propósito:** Monitorizar e (re)agir processo físicos
- **Método:** Variante do Sistema Interativo que permite import limites máximos aos tempos de resposta para diferentes classes de eventos externos

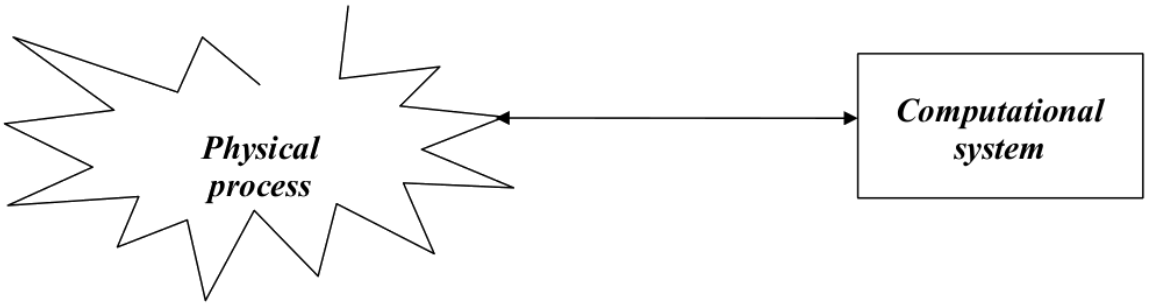


Figure 7: Real Time System

2.1.4 Network Operating System

- **Propósito:** Obter vantagem com as interconexões de `hardware` existentes de sistemas computacionais para estabelecer um conjunto de serviços comuns a uma comunidade.

A máquina é mantêm a sua individualidade mas está dotada de um conjunto de primitivas que permite a comunicação com outras máquinas da mesma rede:

- partilha de ficheiros (ftp)
- acesso a sistemas de ficheiros remotos (NFS)
- Partilha de recursos (e.g. impressoras)

- Acesso a sistemas computacionais remotos:
  - telnet
  - remote login
  - ssh
- servidores de email
- Acesso à internet e/ou Intranet

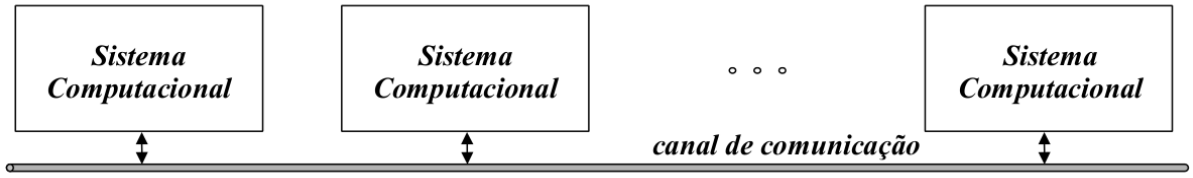


Figure 8: Networking Operating System

2.1.5 Distributed Operating System

- **Propósito:** Criar uma rede de computadores para explorar as vantagens de usar **sistemas multiprocessador**, estabelecendo uma cada de abstração onde o utilizador vê a computação paralela distribuída por todos os computadores da rede como uma única entidade
- **Metodologia:** Tem de garantir uma completa **transparência ao utilizador** no acesso ao processador e outros recursos partilhados (e.g. memória, dados) e permitir:
  - distribuição da carga de `jobs` (programas a executar) de forma dinâmica e estática
  - automaticamente aumentar a sua capacidade de processamento de forma dinâmica se
    * um novo computador se ligar à rede
    * forem incorporados novos processadores/computadores na rede
  - a paralelização de operações
  - implementação de mecanismos tolerantes a falhas

2.2 Classificação com base no propósito

- Mainframe
- Servidor
- Multiprocessador
- Computador Pessoal
- Real time
- Handheld
- Sistemas Embutidos
- Nós de sensores
- Smart Card

3 Multiprocessing vs Multiprogramming

3.1 Paralelismo

- Habilidade de um computador **executar simultaneamente** um ou mais programas
- Necessita de possuir uma estrutura multicore
  - Ou processadores com mais que um core
  - Ou múltiplos processadores por máquina
  - Ou uma estrutura distribuída
  - Ou uma combinação das anteriores

Se um sistema suporta este tipo de arquitectura, suporta **multiprocessamento**

O **multiprocessamento** pode ser feito com diferentes arquitecturas:

- **SMTP** - symmetric processing (SMP)
  - Computadores de uso pessoal
  - Vários processadores
  - A memória principal é partilhada por todos os processadores
  - Cada core possui cache própria
  - Tem de existir **mecanismos de exclusão mútua** para o hardware de suporte ao multiprocessamento
  - Cada processador vê toda a memória (como memória virtual) apesar de ter o acesso limitado
- **Planar Mesh**
  - Cada processador liga a 4 memória adjacentes

3.2 Concorrência

- Ilusão criada por um sistema computacional de "aparentemente" ser capaz de executar mais programas em simultâneo do que o seu número de processadores
- Os processador(es) devem ser atribuídos a diferentes programas de forma multiplexada no tempo

Se um sistema suporta este tipo de arquitectura suporta **multiprogramação**

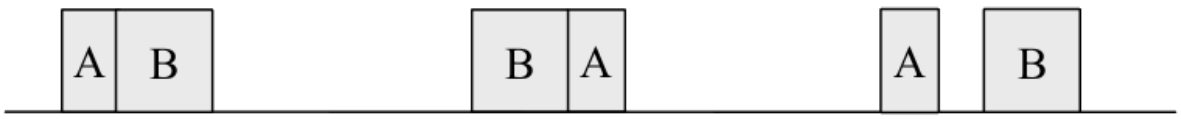


Figure 9: Exemplo de multiplexing temporal: Os programas A e B estão a ser executados de forma concorrente num sistema single processor

4 Estrutura Interna de um Sistema Operativo

Um sistema operativo deve:

- Implementar um ambiente gráfico para interagir com o utilizador
- Permitir mais do que um utilizador
  - Tanto simultânea como separadamente
- Ter capacidade de ser `multitasking`, i.e., executar vários programas ao mesmo tempo
- Implementar memória virtual
- Permitir o acesso, de forma transparente ao utilizador, a:
  - sistemas de ficheiros locais e/ou remotos (i.e., na rede)
  - dispositivos de I/O, independentemente da sua funcionalidade
- Permitir a ligação da máquina por rede a outras máquinas
- Conter um bom conjunto de `device drivers`
- Permitir a ligação de dispositivos `plug and play` [3]

4.1 Design de um sistema operativo

Por estas razões, um sistema operativo é **complexo**, com milhões de linhas de código. O design e implementação do seu `kernel`pode seguir as seguintes filosofias:

- Monolithic
- Layered (por camada)
- Microkernels
- Client-Server Model
- Virtual Machines
- Exokernels

4.1.1 Monolithic system

- A perspectiva mais utilizada
- Só existe um **único programa** a ser executado em `kernel mode`
- Um **único entry point**
  - Todos os pedidos ao sistema são feitos usando este `entry-point`
- Comunicação com o sistema através de `syscall` [4]
  - Implementadas por um conjunto de rotinas
  - Existe ainda outro conjunto de funções auxiliares para a system call
- Qualquer parte do sistema (aka `kernel`) pode "ver" qualquer outra parte do sistema

[3] ficheiro em código fonte de compilação separada
[4] De um para um

- **Vantagem:** eficiência no acesso a informação e dados
- **Desvantagem:** Sistema difícil de testar e modificar

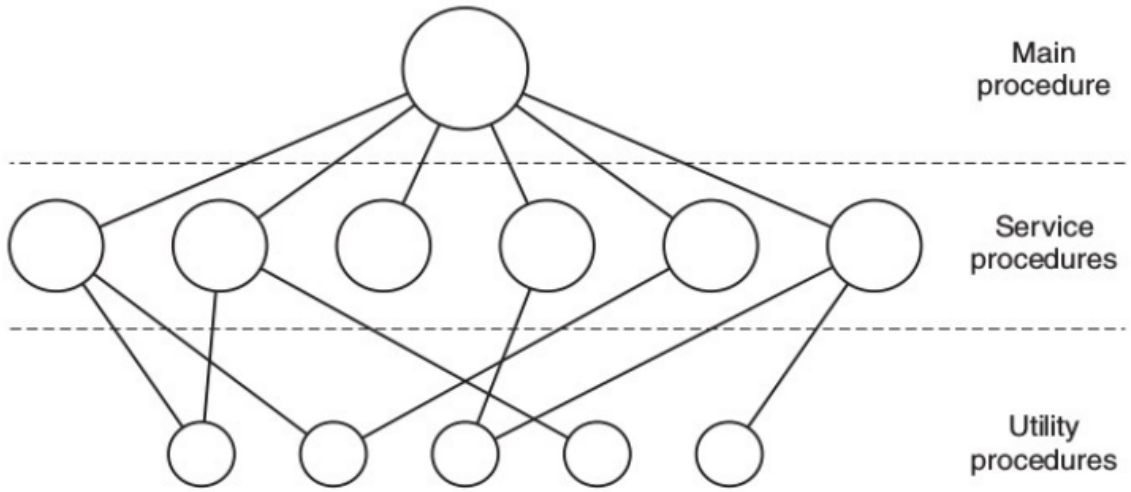


Figure 10: Diagrama de um `kernel` monolítico - imagem retirada do livro *Modern Operating Systems, Andrew Tanenbaum & Herbert Bos*

4.1.2 Layered Approach: Divisão por camadas

- Perspetiva modular
  - O sistema operativo é constituído por um conjunto de camadas, com diferentes níveis hierárquicos
- A interação **só é possível entre camadas adjacentes**
  - Uma função da camada mais superior não pode chamar uma função da camada mais abaixo
  - Tem de chamar uma função da camada imediatamente abaixo que irá tratar de chamar funções das camadas mais abaixo (estilo `sofs`)
- Não é simples de projetar
  - É preciso definir claramente que funcionalidades em que camada, o que pode ser difícil de decidir
- **Fácil de testar e modificar**, mas uma **grande perda de eficiência**
  - A eficiência pode piorar se a divisão de funções não for bem feita
  - Existe um `overhead` adicional causado pelo chamada de funções entre as várias camadas
- Facilita a divisão de funções entre o modo de utilizador e o modo de `kernel`

Table 3: Estrutura de um sistema operativo por camadas - Retirada do livro *Modern Operating Systems, Andrew Tanenbaum & Herbert Bos*

Layer	Function
5	Operador
4	Programas do Utilizador
3	Gestão de dispositivos de I/O
2	Comunicação Operator- Process
1	Memory and drum management
0	Alocação do processador e gestão do ambiente multiprogramado

4.1.3 Microkernel

- Posso ter **modularidade** sem ser obrigado a usar camadas em níveis hierárquicos diferentes
- Defino um conjunto de módulos de "pequena dimensão", com funcionalidades bem definidas
  - apenas o `microkernel` é executado em `kernel space`, com permissões de `root`
  - todos os outros módulos são executados em `user space` e comunicam entre si usando os mecanismos de comunicação providenciados pelo `microkernel`
  - Os módulos que são executados em `user space` podem ser lançados no startup ou dinamicamente à medida que são precisos (dispositivos `plug-and-play` [5])
- O `microkernel` é responsável por:
  - Gestão de Processos
  - Gestão da Memória
  - Implementar sistemas simples de comunicação interprocess
  - Escalonamento do Processador (Processor Scheduling)
  - Tratar das interrupções
- Sistema robusto
  - Manipulação de um filesystem é feita em `user space`. Se houver problemas a integridade do sistema físico não é afetada

[5] ficheiro em código fonte de compilação separada

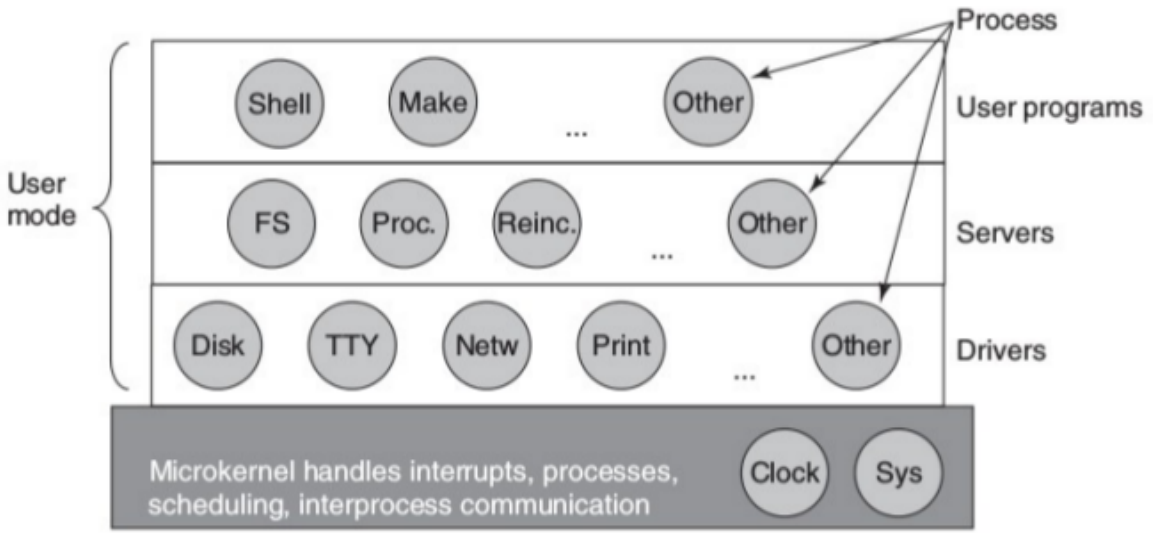


Figure 11: Estrutura de um sistema operativo que usa microkernel - Retirada do livro *Modern Operating Systems, Andrew Tanenbaum & Herbert Bos*

4.1.4 Virtual machine (hypervisors)

- Criam plataformas virtuais onde podem ser instalados `guess OSs`
- Existem dois tipos de hypervisors
  - Type-1 (**`native`** `hypervisor`): executa o `guest OS` **diretamente** no `hardware` da máquina `host` (máquina física onde a máquina virtual vai ser executada). Exemplos:
    * z/VM
    * Xen
    * Hyper-V
    * VMware ESX
  - Type-2 (`hosted supervisor`): executa o `guest OS` **indiretamente** no `hardware` da máquina, sendo a máquina virtual executada "em cima" do sistema operativo do `host`. Exemplos:
    * VirtualBox
    * VMware Workstation
    * Parallels
- Existem exemplos de `hypervisors` híbridos, que tanto podem ser executar o `guest OS` indiretamente (por cima do sistema operativo) ou diretamente no `hardware` da máquina:
  - KVM
  - bhyve

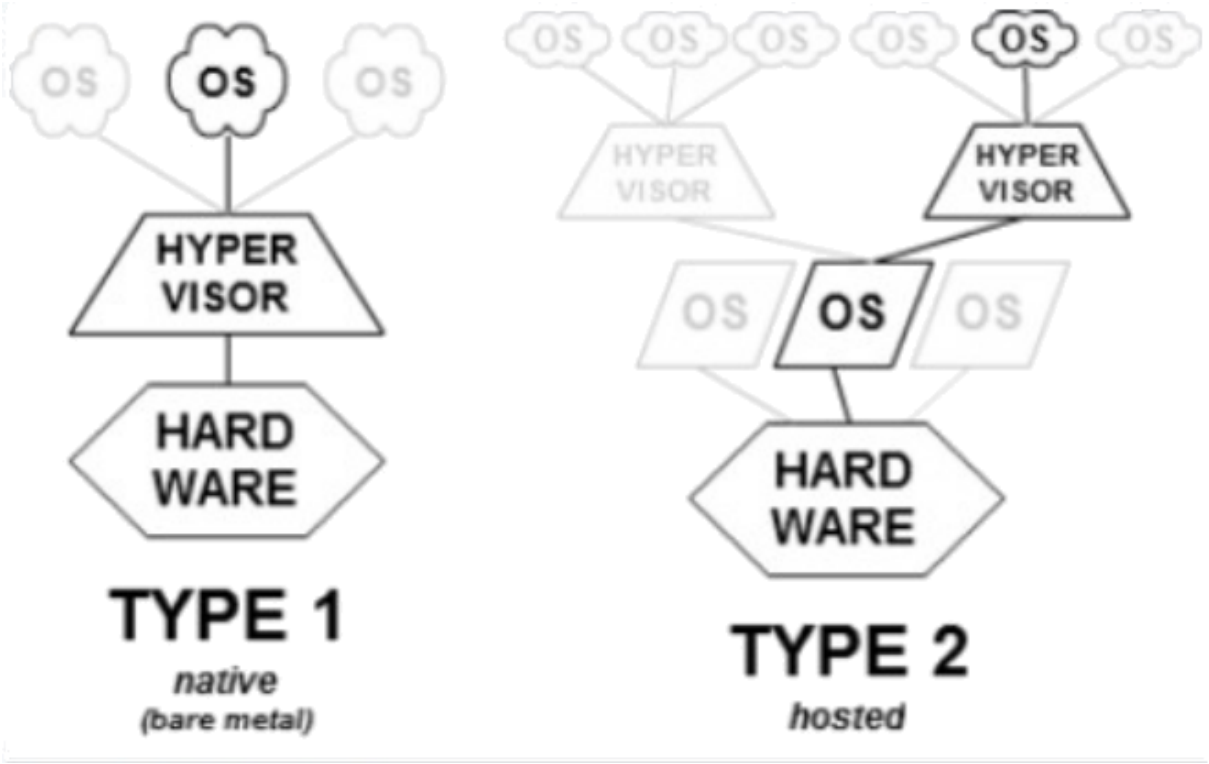


Figure 12: Estrutura de uma virtual machine - Imagem retirada da Wikipedia

4.1.5 Client-Server

- Implementação modular, baseada na relação cliente-servidor
  - A comunicação é feita através de **pedidos e respostas**
  - Para isso é usada `message-passing`
- Pode estar presente um `microkernel` que manipula operações de baixo nível
- Pode ser generalizado e usado em sistemas `multimachine`

4.1.6 Exokernels

- Usa um `kernel` com funcionalidades reduzidas
  - Apenas providencia algumas abstrações de `hardware`
- Segue a filosofia de *"Em vez de clonar a máquina virtual, divido-a"*
  - Os recursos são **divididos em partições**, em vez de clonados
  - Os recursos são alocados às `virtual machines` e a sua utilização é controlada pelo `microkernel`
- Permite a implementação de **camadas de abstração personalizadas** consoante as necessidades
- **Eficiente:** Poupa uma camada destinada a efetuar o mapeamento

4.2 Estruturas Internas do Unix/Linux e Windows

4.2.1 Estrutura Interna do Unix (tradicional)

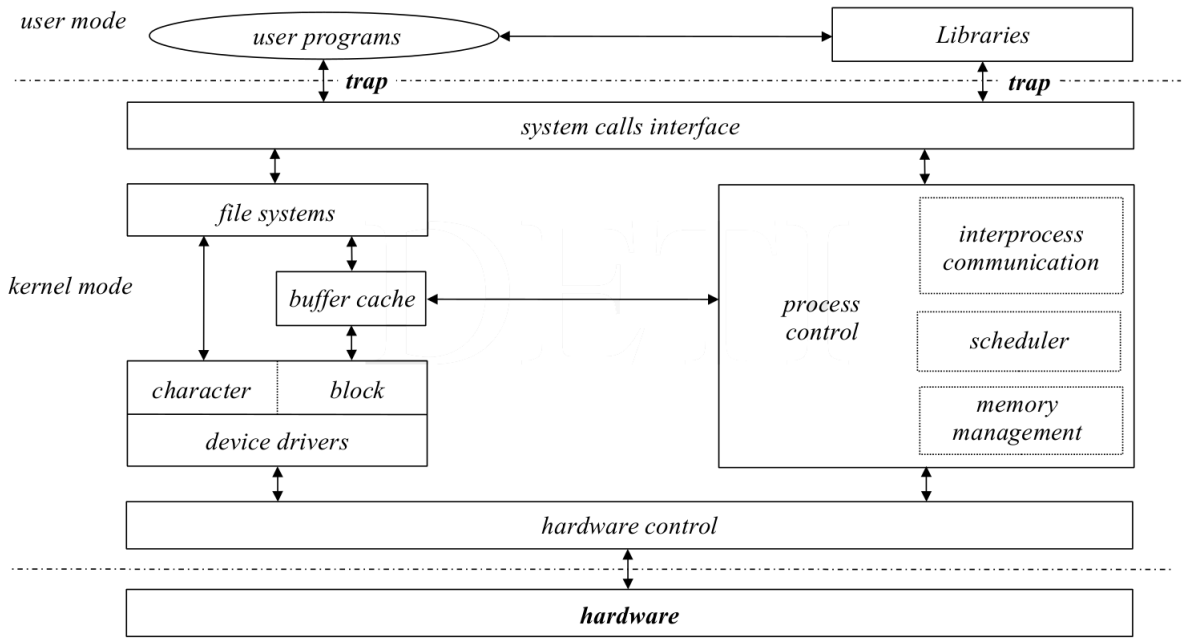


Figure 13: Estrutura Interna do Unix - Tradicional

Legenda:

- `trap`: interrupção por software (única instrução que muda o modo de execução)
- `buffercache`: espaço do disco onde são mantidos todos os ficheiros em cache (aka abertos)
  - **desmontar uma pen:** forçar a escrita da buffer cache para a pen

Unix considera tudo como sendo ficheiros: - ou blocos (buffer cache) - ou bytes

`open`, `close`, `fork` **não são system calls**. São funções de biblioteca que acedem às `system call` (implementadas no `kernel`). São um interface amigável para o utilizador ter acesso a estas funcionalidades.

4.2.2 Estrutura Global do Unix

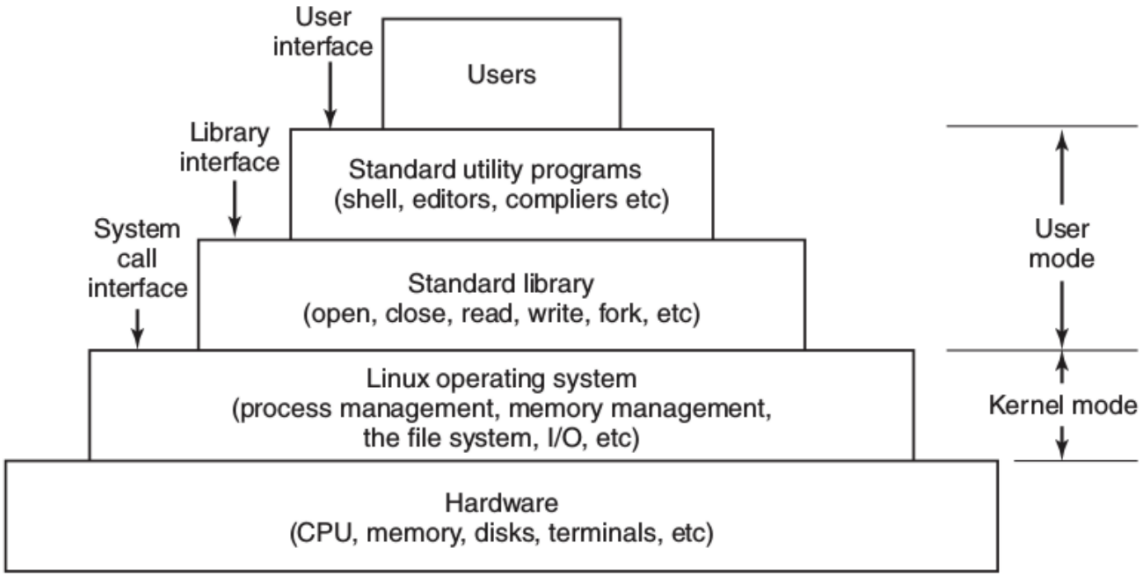


Figure 14: Estrutura Global do Sistema Linux - Retirada do livro *Modern Operating Systems, Andrew Tanenbaum & Herbert Bos*

4.2.3 Estrutra do Kernel Unix

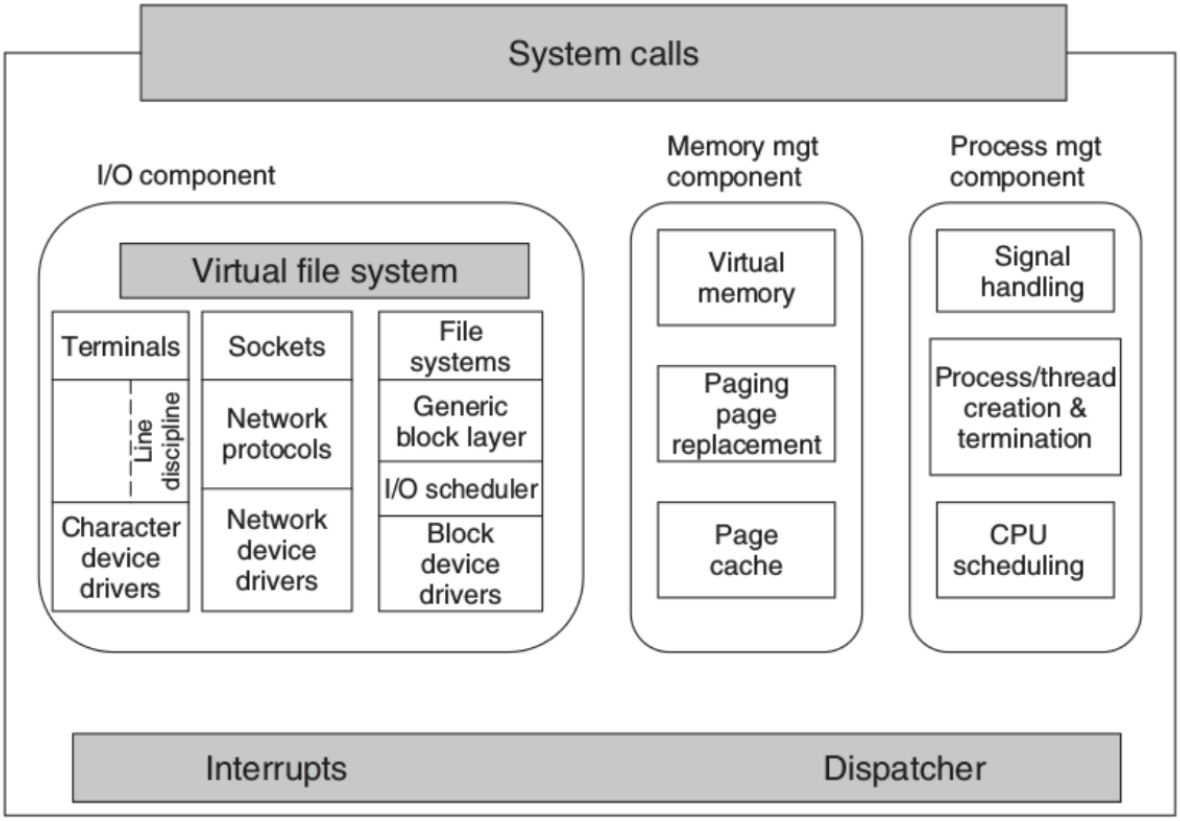


Figure 15: Estrutura do Kernel do Linux - Retirada do livro *Modern Operating Systems, Andrew Tanenbaum & Herbert Bos*

4.2.4 Estrutura do Kernel Windows

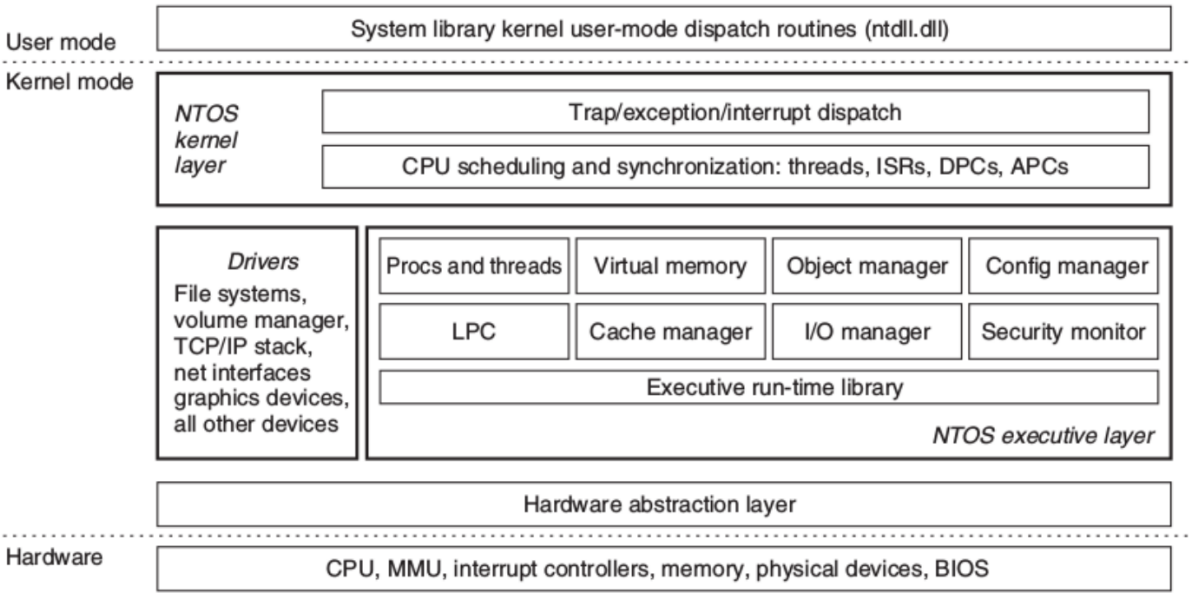


Figure 16: Estrutura Interna do Kernel do Windows - Retirada do livro *Modern Operating Systems, Andrew Tanenbaum & Herbert Bos*

5 Conceitos Introdutórios

Num ambiente multiprogramado, os processos podem ser:

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

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

5.1 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

6 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 acess_resource(unsigned int p)
13 {
14     enter_critical_section(p);
15     use_resource();     //  critical section
16     leave_critical_section(p);
17 }
```

7 Acesso a Memória Partilhada

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

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

```
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     }
```

```
}

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

7.1 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)*

7.1.1 Produtor

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

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

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


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

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

7.1.2 Consumidor

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

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

8 Acesso a uma Zona Crítica

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

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

8.1 Tipos de Soluções

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

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

8.2 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  }
```

8.3 Eliminar a Alternância Estrita

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

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

Assume que:

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

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

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

8.4 Garantir a exclusão mútua

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

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

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

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

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

Assume que:

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

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

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

8.5 Garantir que não ocorre deadlock

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

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

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

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

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

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

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

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

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

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

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

É uma **solução válida**:

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

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

8.7 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

8.8 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)

8.9 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*

9 Soluções de Hardware

9.1 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.

9.2 Instruções Especiais em Hardware

9.2.1 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 }
```

9.2.2 Compare and Swap

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

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

```
shared int value = 0;

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

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

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

9.3 Busy Waiting

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

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

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

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

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

- guardar o estado do processo atual
  - variáveis

  - stack
  - $PC

- copiar para memória o código do novo processo

9.4 Block and wake-up

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

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

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

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

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

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

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

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

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

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

10 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**.

10.1 Implementação

Um semáforo é implementado através de:

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

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

10.1.1 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

10.1.2 Solução típica de sistemas *uniprocessor*

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

static SEMAPHORE sem[R];

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

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

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

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

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

10.2 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;
```

```
            }
            unlock(p);
        } while (!done);
    do_something_else();
    }
}

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

10.2.1 Como Implementar usando semáforos?

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

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

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


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

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

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

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

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

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

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

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

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

10.3 Análise de Semáforos

10.3.1 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

10.3.2 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**

10.3.3 Problemas do uso de semáforos

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

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

10.4 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

11 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"

11.1 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      ...
```

11.2 Tipos de Monitores

11.2.1 Hoare Monitor

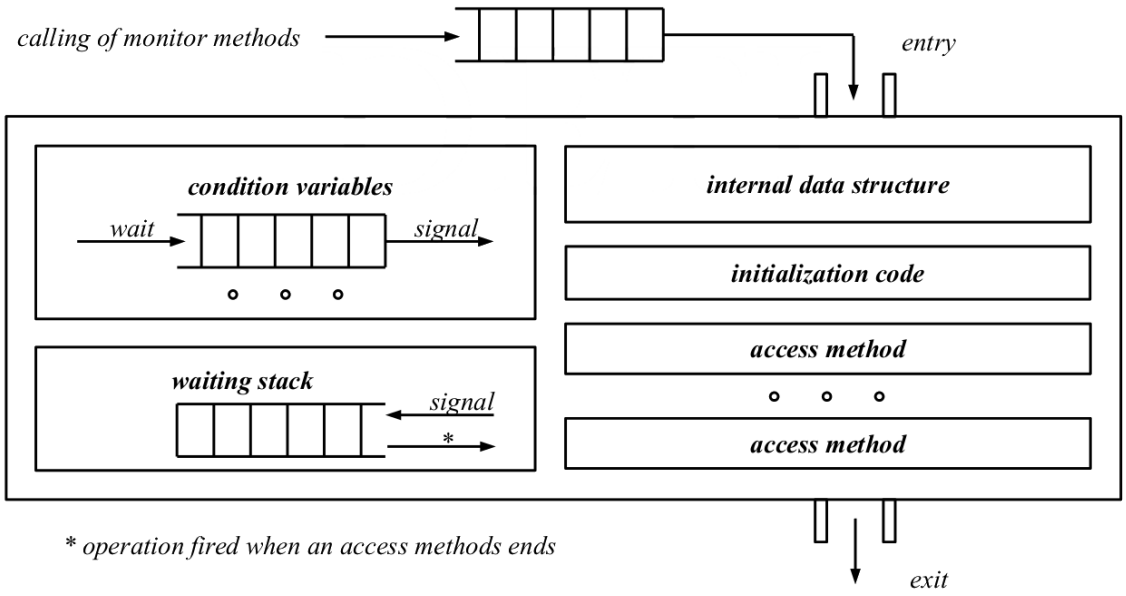


Figure 17: Diagrama da estrutura interna de um Monitor de Hoare

- 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

11.2.2 Brinch Hansen Monitor

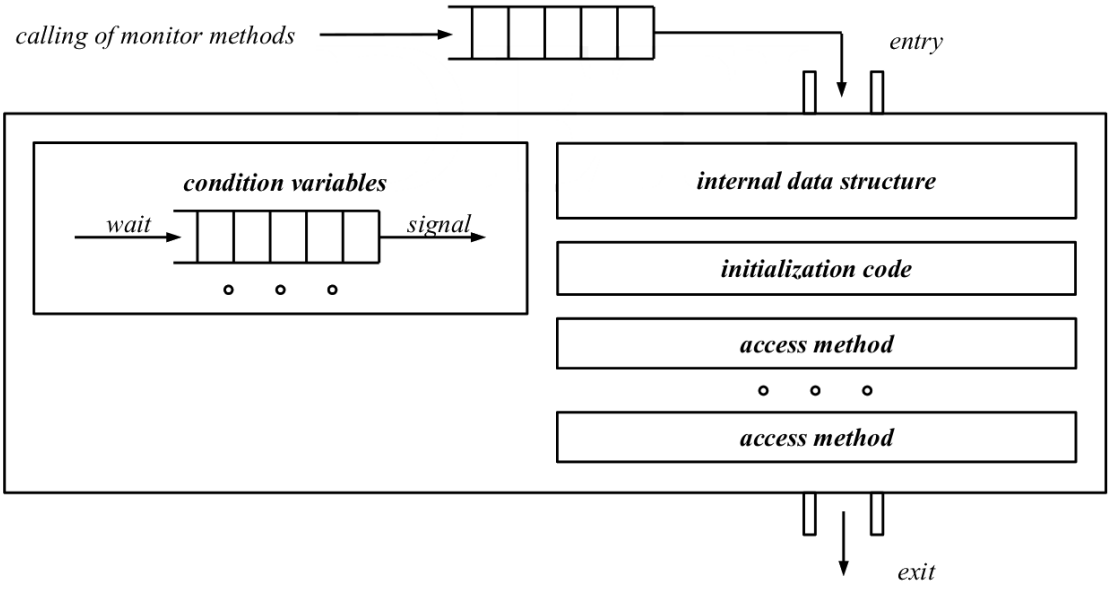


Figure 18: Diagrama da estrutura interna de um Monitor de Brinch Hansen

- 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`

11.2.3 Lampson/Redell Monitors

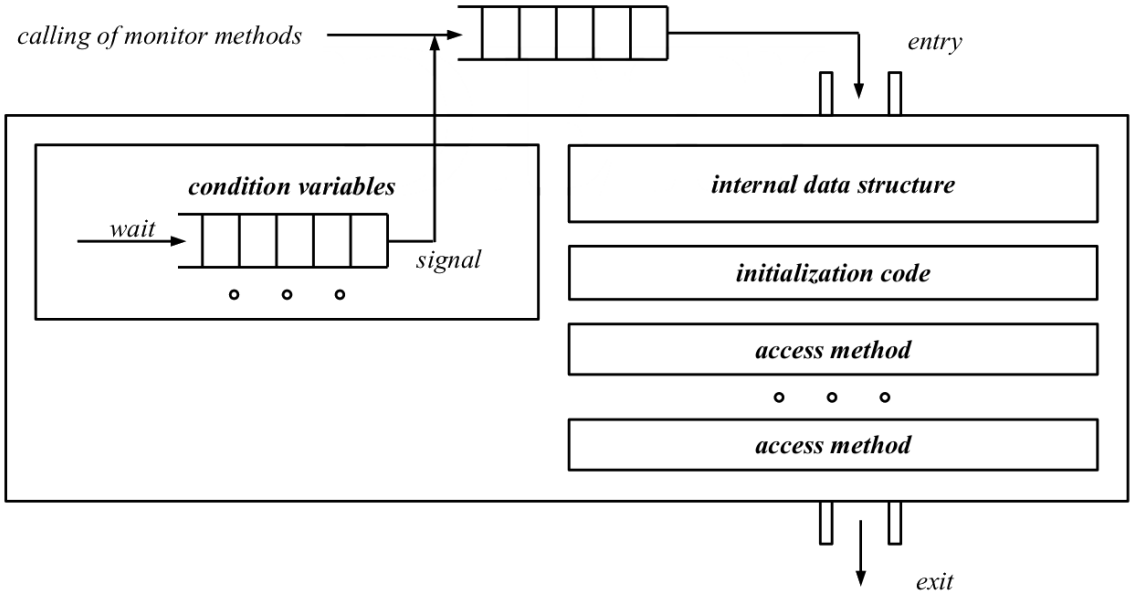


Figure 19: Diagrama da estrutura interna de um Monitor de Lampson/Redell

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

11.3 Bounded-Buffer Problem usando Monitores

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

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

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

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

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

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

O valor inicial do **mutex** é 0.

11.4 POSIX support for monitors

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

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

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

- `mutexes`
- `variáveis de condição`

As funções disponíveis são:

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

12 Message-passing

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

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

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

- `send`

- `receive`

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

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

12.1 Direct vs Indirect

12.1.1 Symmetric direct communication

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

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

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

12.2 Asymmetric direct communications

Só o `sender` tem de explicitar o destinatário:

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

12.3 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?

12.4 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

12.5 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**

12.6 Bound-Buffer Problem usando mensagens

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

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

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

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

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

12.7 Message Passing in Unix/Linux

System V:

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

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

POSIX

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

13 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**

13.1 POSIX Shared Memory

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

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

13.2 System V Shared Memory

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

14 Deadlock

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

Os recursos podem ser:

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

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

O caso mais simples de deadlock ocorre quando:

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

14.1 Condições necessárias para a ocorrência de deadlock

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

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

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

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

14.1.1 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.[6]

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

14.2 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:

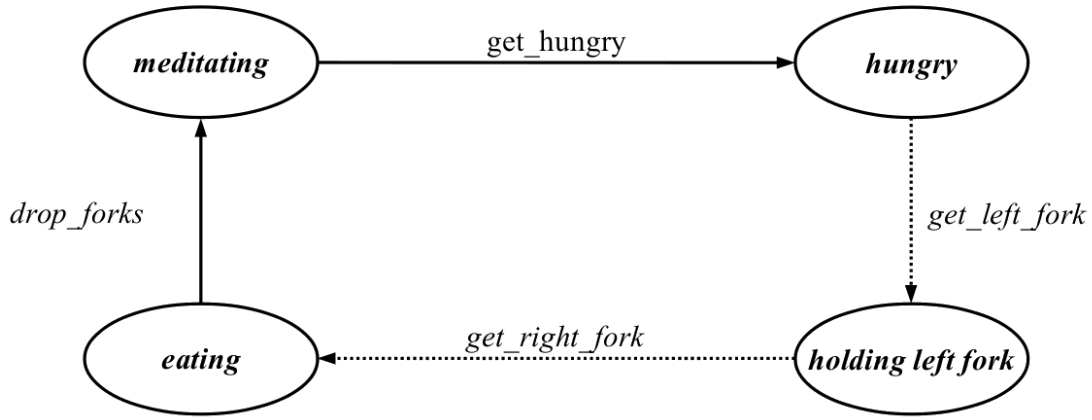


Figure 20: Ciclo de Vida de um filósofo

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

[6] ficheiro em código fonte de compilação separada

```
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**.

14.3 Prevenção de Deadlock

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

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

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

14.3.1 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*

14.3.2 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**

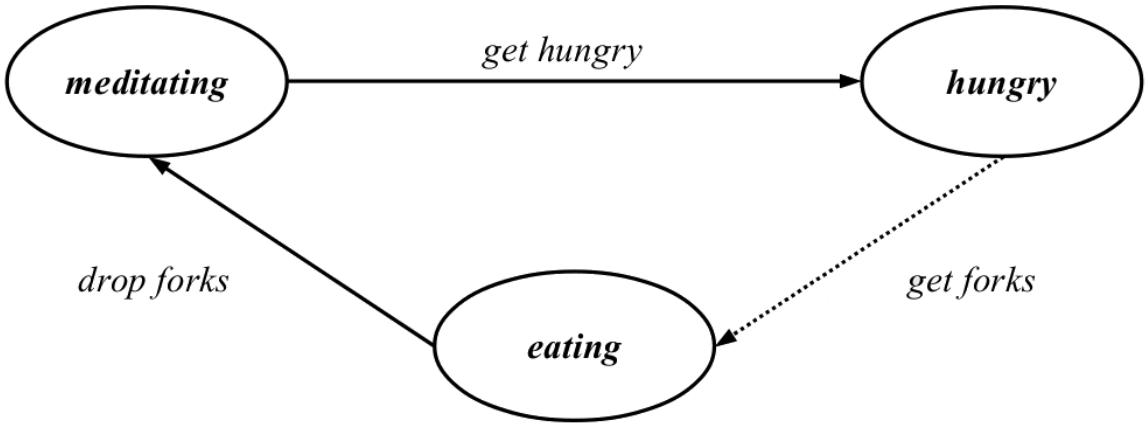


Figure 21: Negar *hold and wait*

Solução equivalente à proposta por Dijkstra.

14.3.3 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`

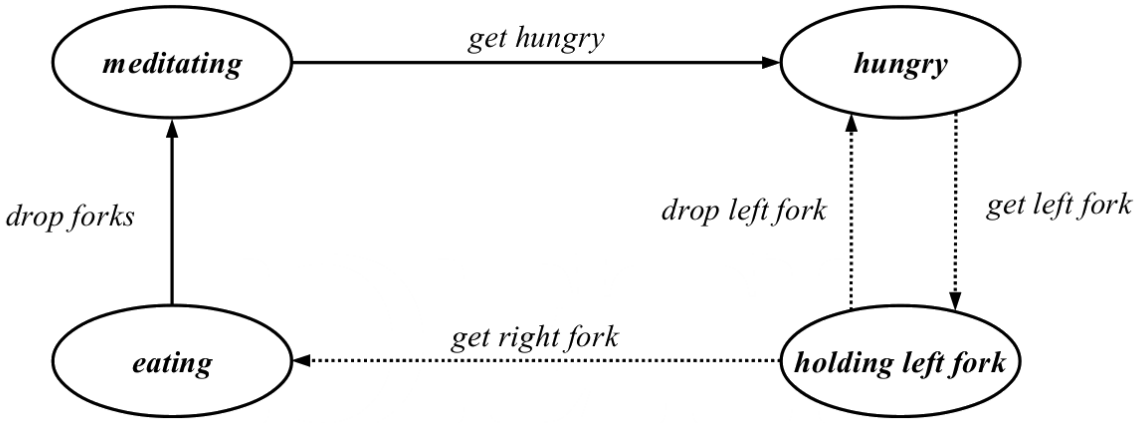


Figure 22: Negar a condição de *no preemption* dos recursos

14.3.4 Negar a espera circular

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

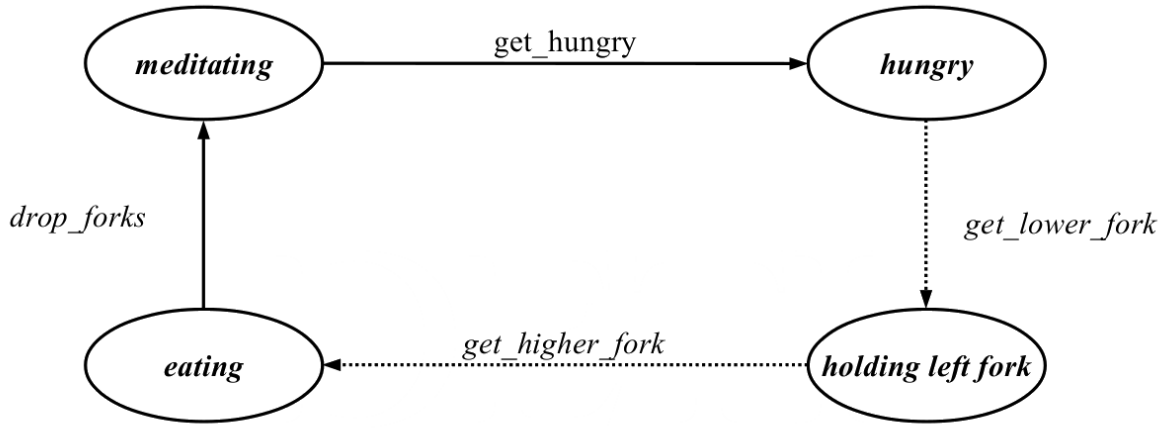


Figure 23: Negar a condição de espera circular no acesso aos recursos

14.4 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

14.4.1 Condições para lançar um novo processo

Considerando:

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

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

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

14.4.2 Algoritmo dos Banqueiros

Considerando:

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

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

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

Table 4: Banker's Algorithm Example

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

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

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

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

Algoritmo dos banqueiros aplicado ao Jantar dos filósofos

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

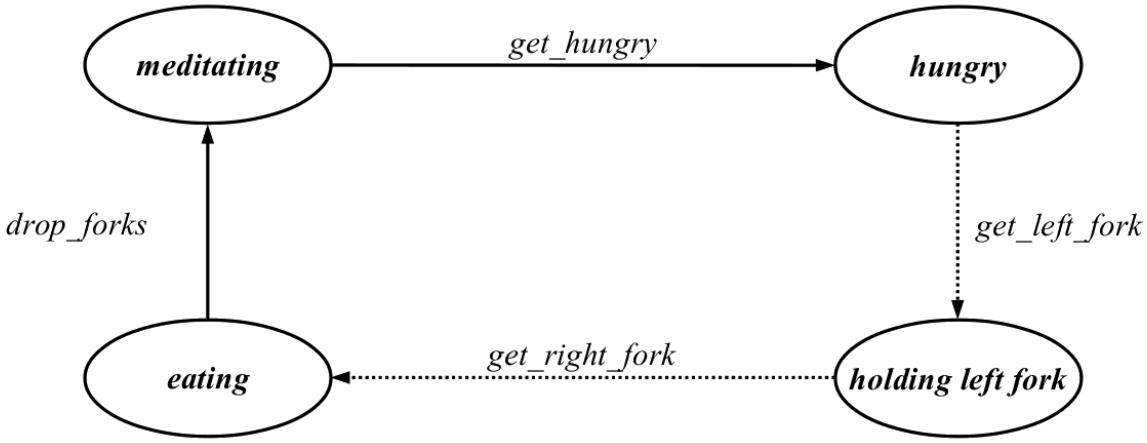


Figure 24: Algoritmo dos banqueiros aplicado ao Jantar dos filósofos

14.5 Deadlock Detection

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

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

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

Existem três métodos para recuperar de deadlock:

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

  - Método eficiente

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

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

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

15 Processes and Threads

15.1 Arquitectura típica de um computador

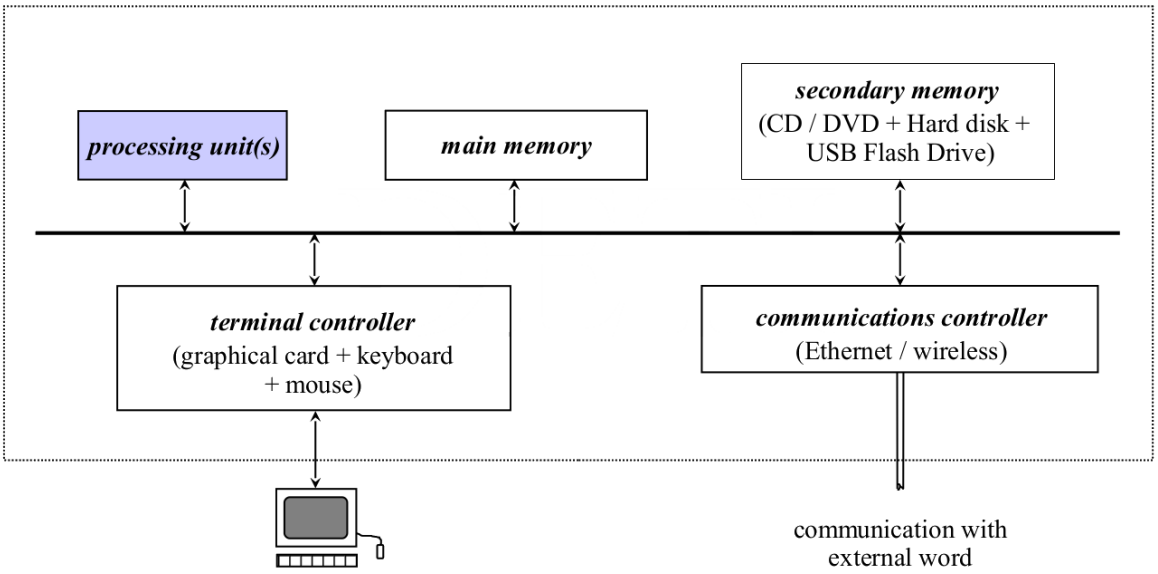


Figure 25: Arquitectura típica de um computador

15.2 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

15.3 Execução num ambiente multiprogramado

O sistema assume que o processo que está na posse do processador irá **ser interrompido**, pudendo 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`

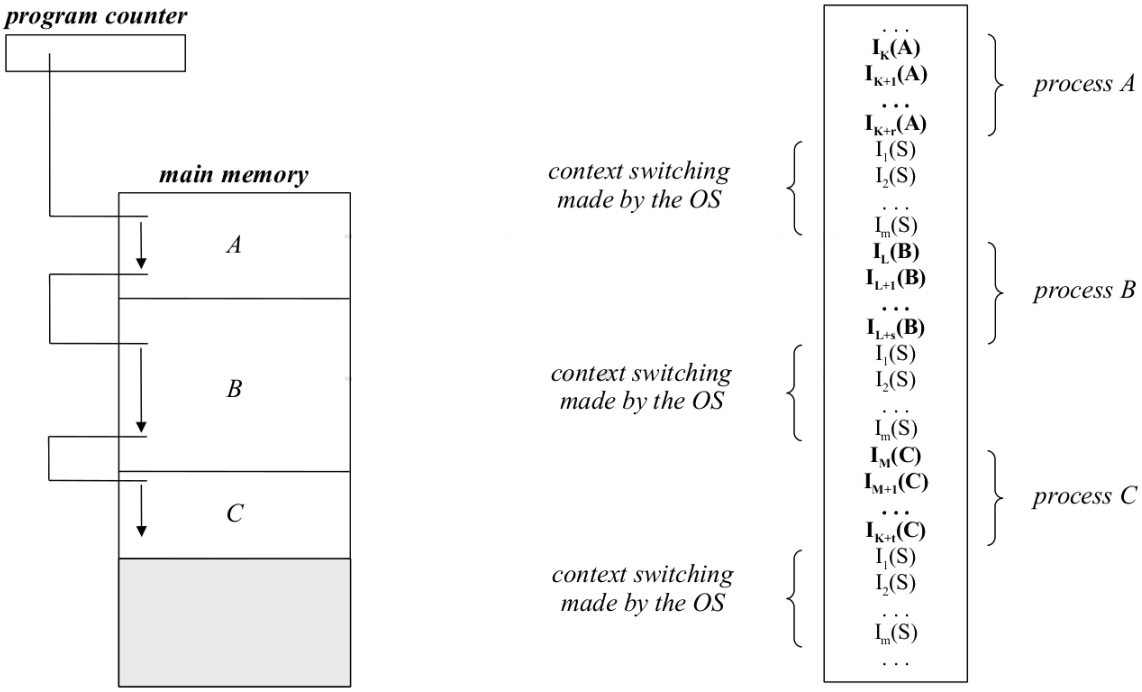


Figure 26: Exemplo de execução num ambiente multiprogramado

15.4 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

15.5 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 puder 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

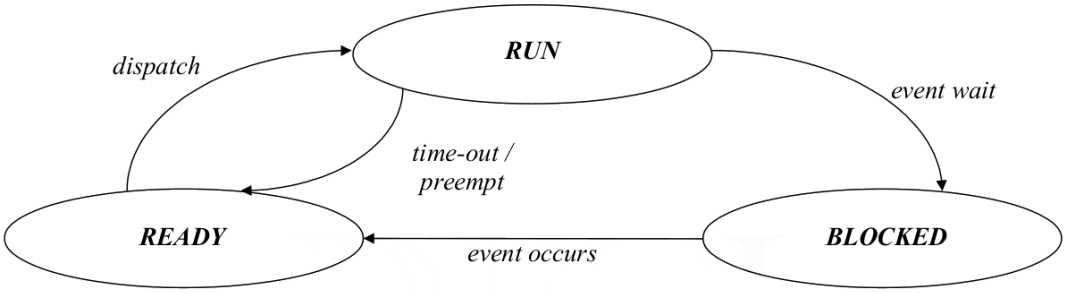


Figure 27: Diagrama de Estados do Processador - Básico

`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**

15.5.1 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 **extender 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`

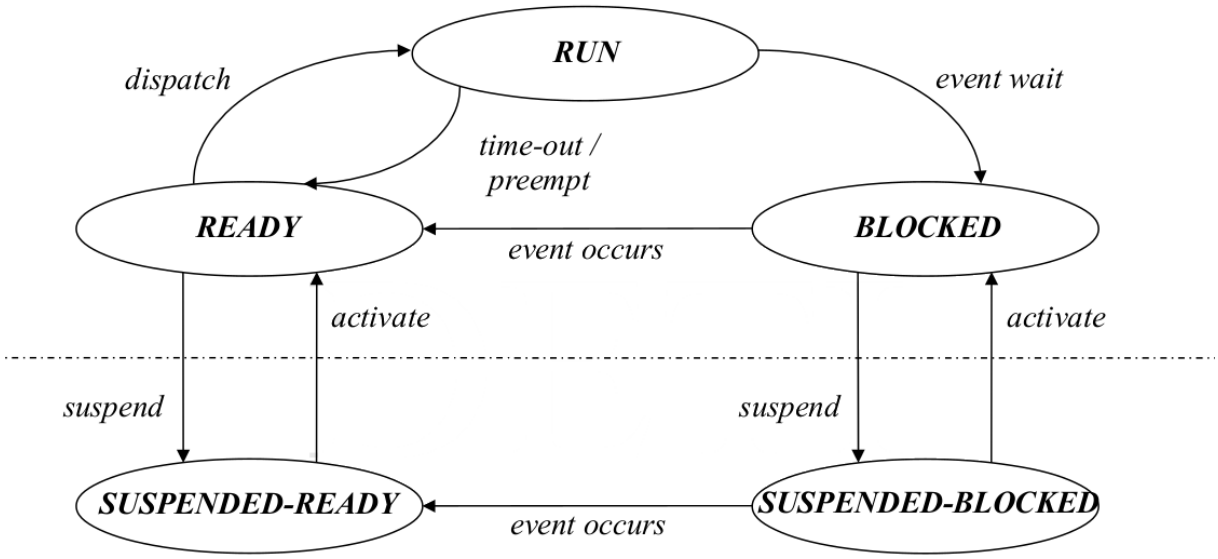


Figure 28: Diagrama de Estados do Processador - Com Memória de Swap

15.5.2 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

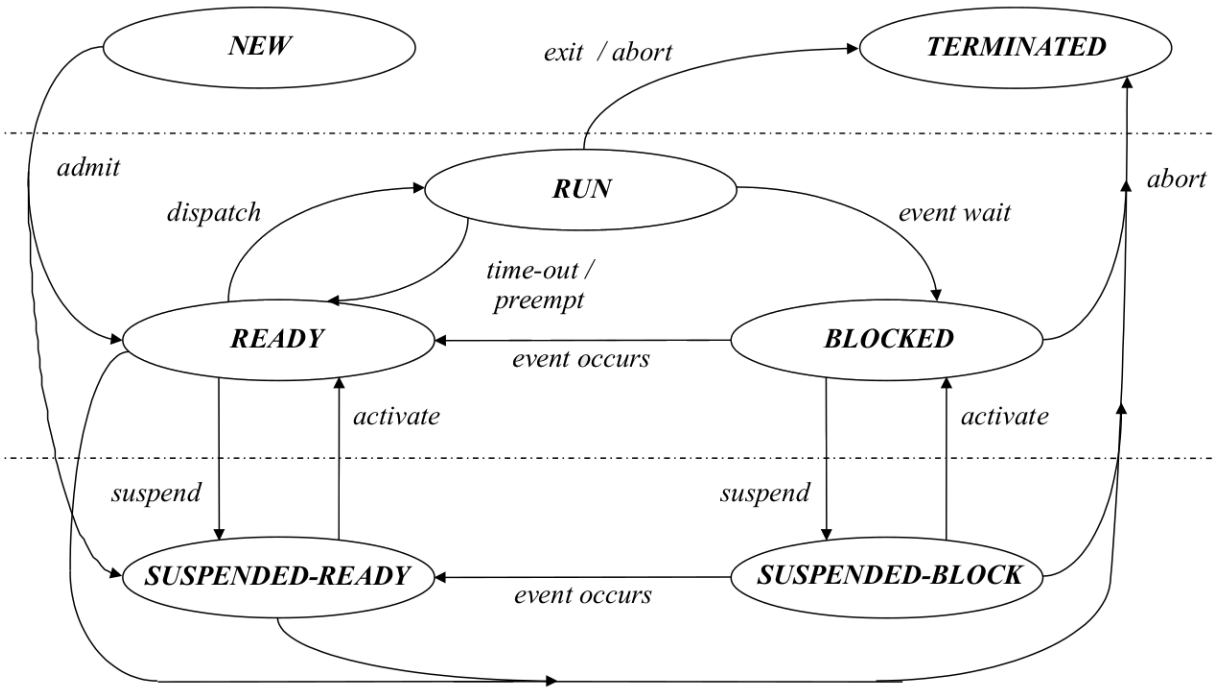


Figure 29: Diagrama de Estados do Processador - Com Processos Temporalmente Finitos

15.6 State Diagram of a Unix Process

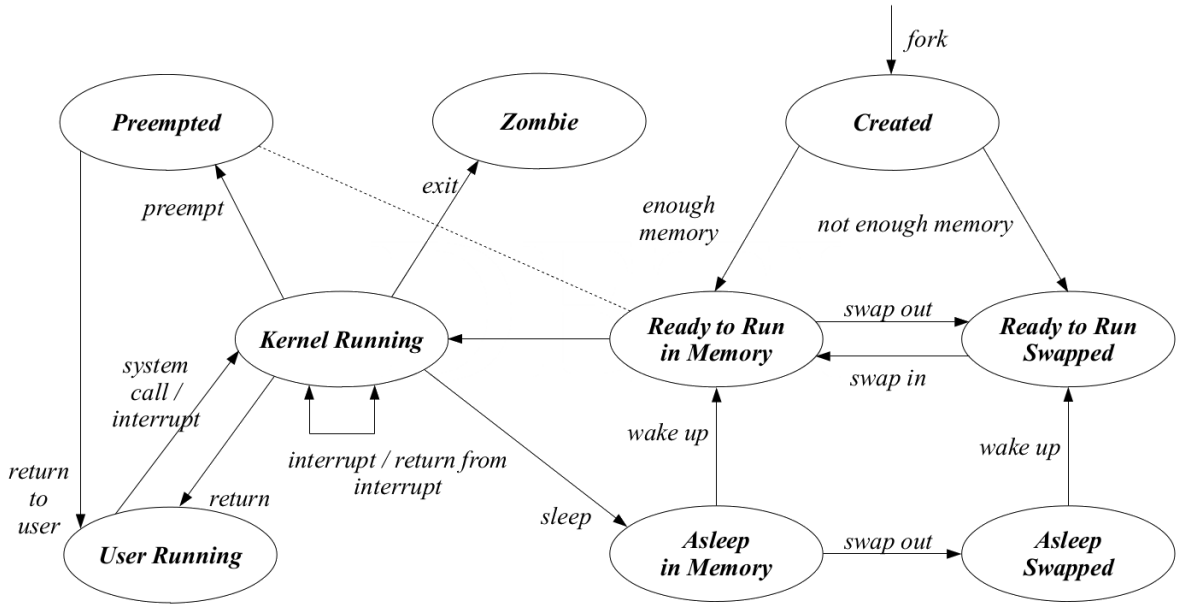


Figure 30: Diagrama de Estados do Processador - Com Memória de Swap

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`

15.7 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 pode ser **`preempted`**. Ou seja, o UNIX não permitia **real-time processing**

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

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

15.8 Unix – traditional login

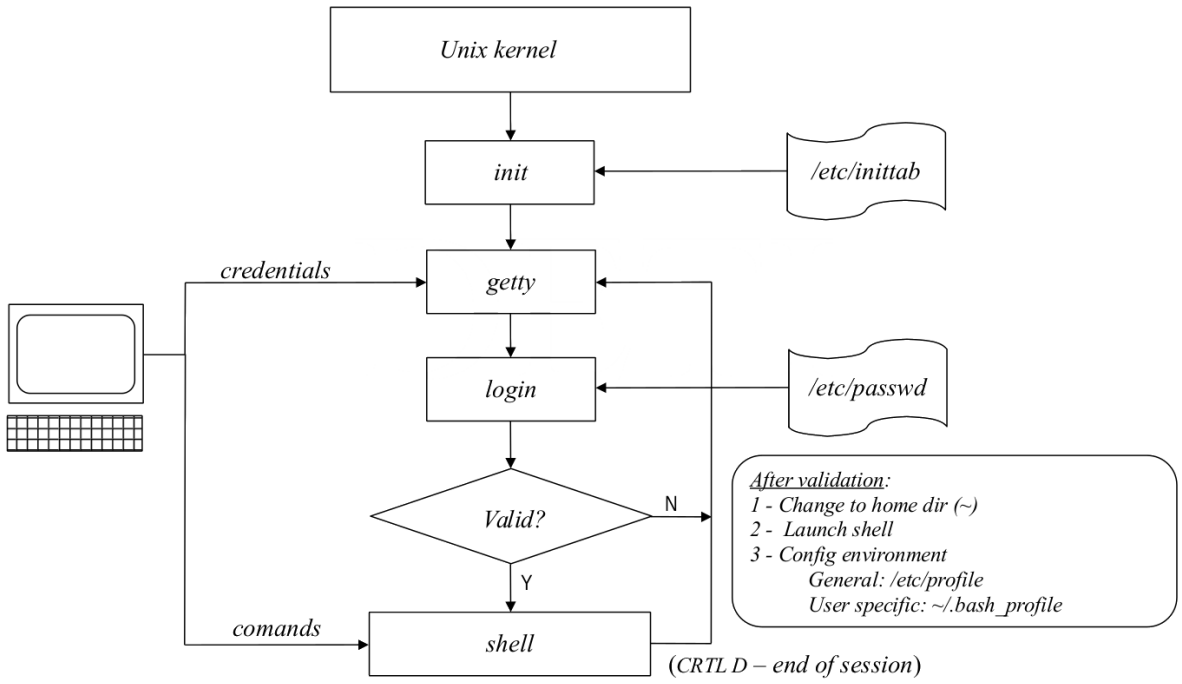


Figure 31: Diagrama do Login em Linux

15.9 Criação de Processos

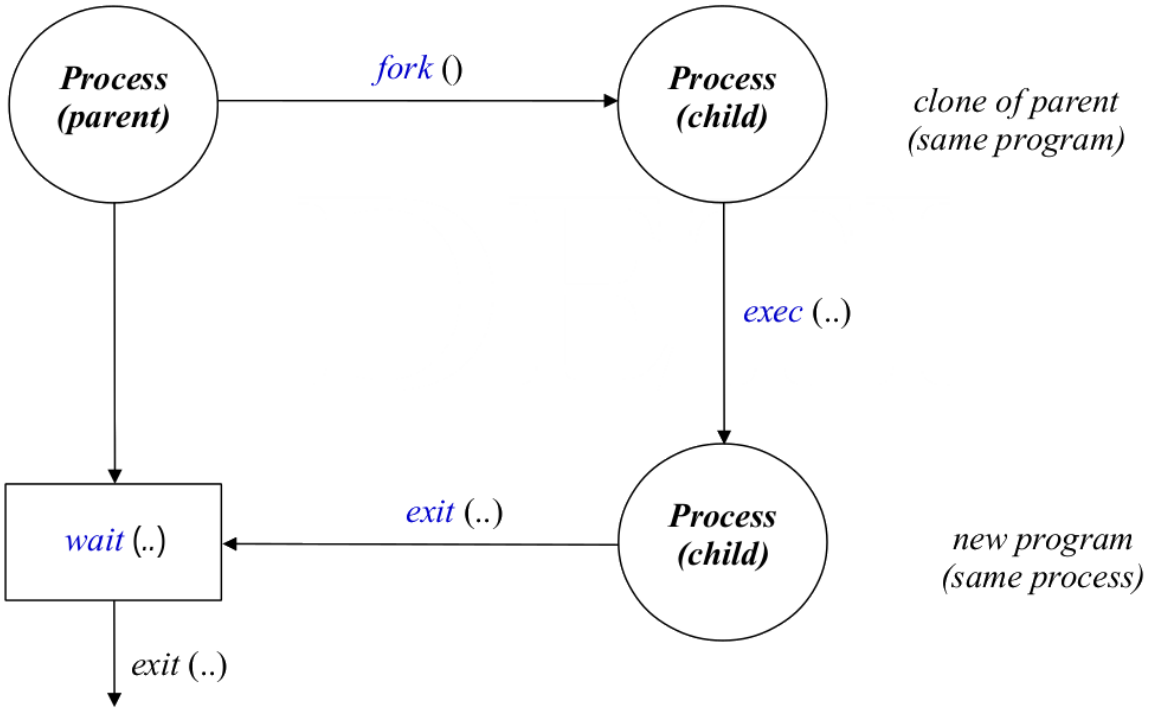


Figure 32: Criação de Processos

```c
#include <stdio.h>
#include <stdlib.h>
#include <sys/types.h>
#include <unistd.h>

int main(void)
{
    printf("Before the fork:\n");
    printf(" PID = %d, PPID = %d.\n",
    getpid(), getppid());

    fork();

    printf("After the fork:\n");
    printf(" PID = %d, PPID = %d. Who am I?\n", getpid(), getppid());

    return EXIT_SUCCESS;
}
```

- **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

```c
#include <stdio.h>
#include <stdlib.h>
#include <sys/types.h>
#include <unistd.h>

int main(void)
{
    printf("Before the fork:\n");
    printf(" PID = %d, PPID = %d.\n",
    getpid(), getppid());

    int ret = fork();

    printf("After the fork:\n");
    printf(" PID = %d, PPID = %d, ret = %d\n", getpid(), getppid(), ret);

    return EXIT_SUCCESS;
}
```

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**

```c
#include <stdio.h>
#include <stdlib.h>
#include <sys/types.h>
#include <unistd.h>

```

```c
int main(void)
{
    printf("Before the fork:\n");
    printf(" PID = %d, PPID = %d.\n", getpid(), getppid());

    int ret = fork();

    if (ret == 0)
    {
        printf("I'm the child:\n");
        printf(" PID = %d, PPID = %d\n", getpid(), getppid());
    }
    else
    {
        printf("I'm the parent:\n");
        printf(" PID = %d, PPID = %d\n", getpid(), getppid());
    }

    printf("After the fork:\n");
    printf(" PID = %d, PPID = %d, ret = %d\n", getpid(), getppid(), ret);

    return EXIT_SUCCESS;
}
```

O `fork` por si só não possui grande interesse. O interesse é puder 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

```c
#include <stdio.h>
#include <stdlib.h>
#include <sys/types.h>
#include <unistd.h>

int main(void)
{
    /* check arguments */
    if (argc != 2)
    {
        fprintf(stderr, "spawn <path to file>\n");
        exit(EXIT_FAILURE);
    }
    char *aplic = argv[1];

    printf("================================\n");

    /* clone phase */
    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`.

15.10 Execução de um programa em C/C++

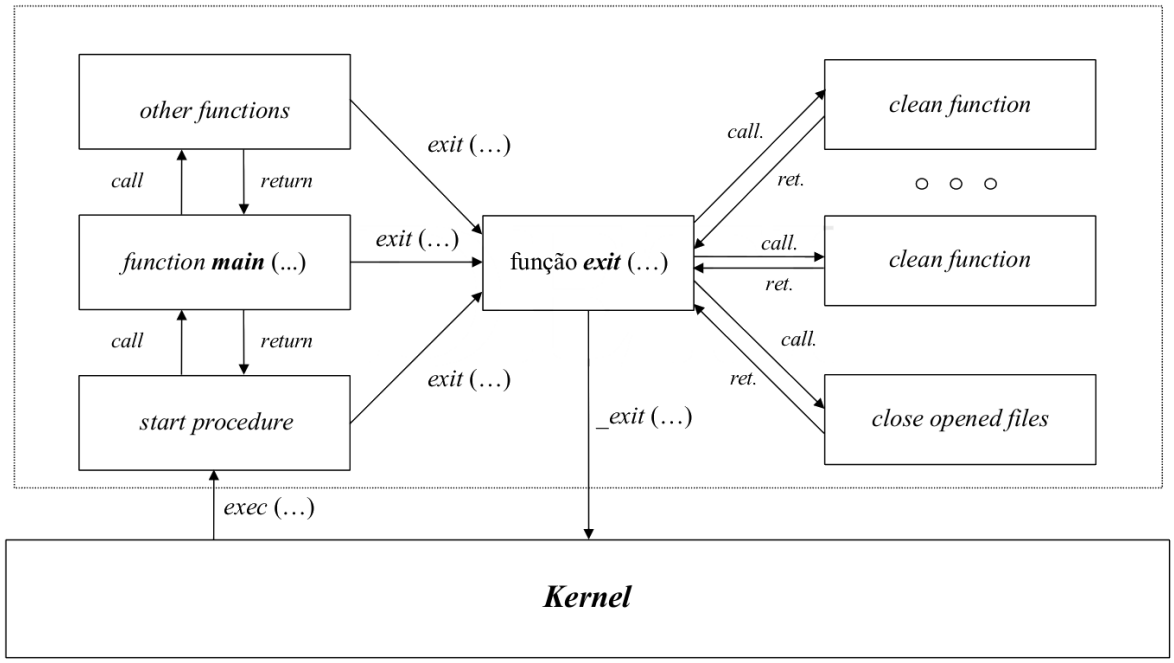


Figure 33: Execução de um programa em C/C++

- 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

15.11 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**

15.12 Espaço de Endereçamento de um Processo em Linux

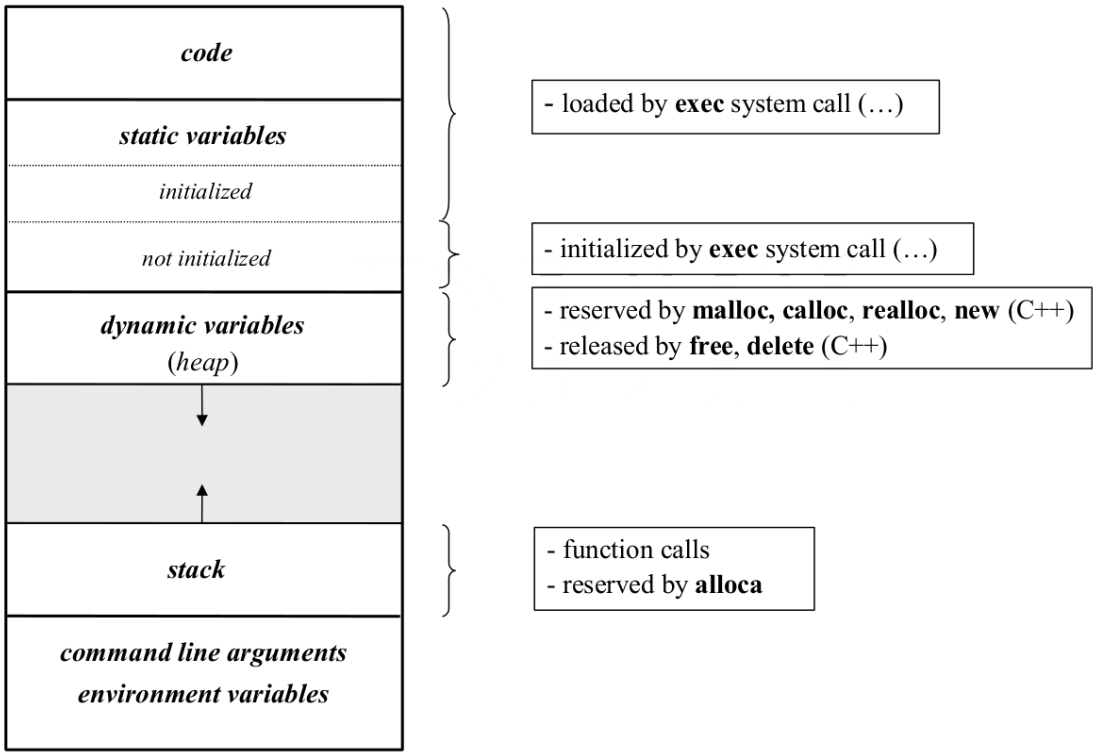


Figure 34: Espaço de endereçamento de um processo em Linux

- 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

15.12.1 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*

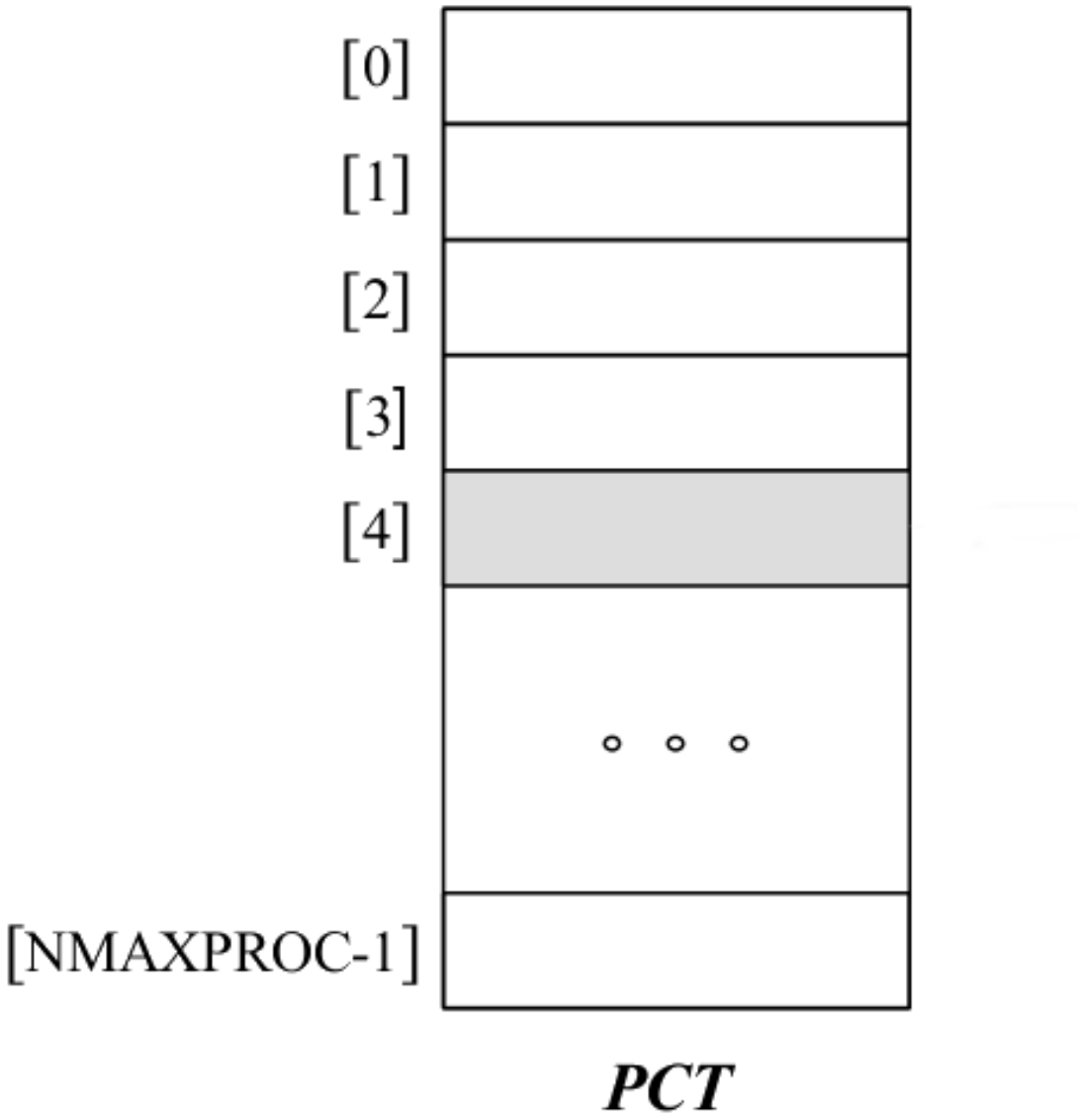


Figure 35: Process Control Table

16 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*

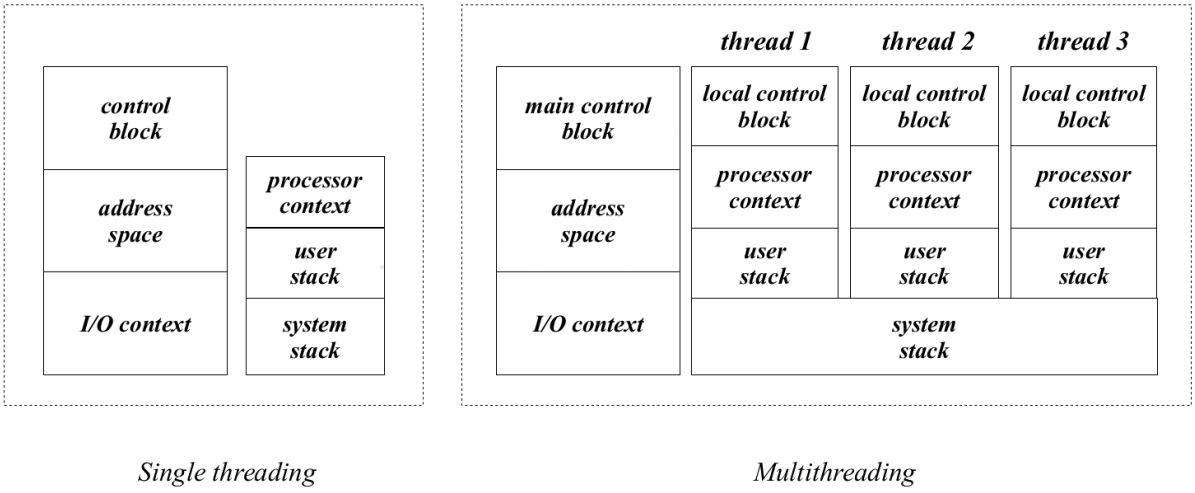


Figure 36: Single threading vs Multithreading

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`.

16.1 Diagrama de Estados de uma thread

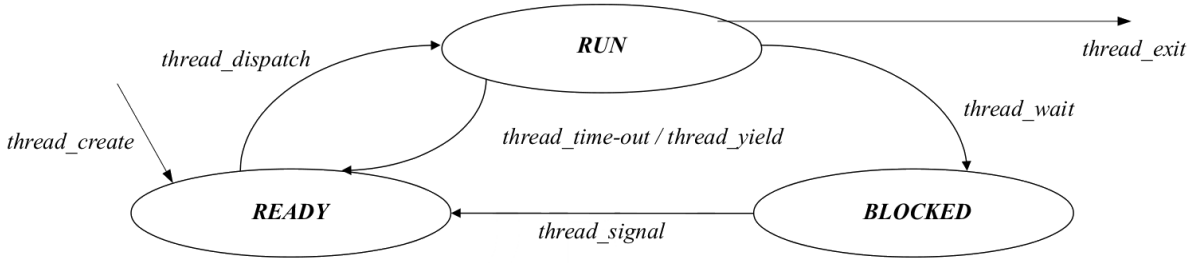


Figure 37: Diagrama de estados de uma thread

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**:

```
- 'run'
- 'ready'
- '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 estado **`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

16.2 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**

16.3 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

16.4 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 portável
- 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**

16.4.1 Libraria pthread

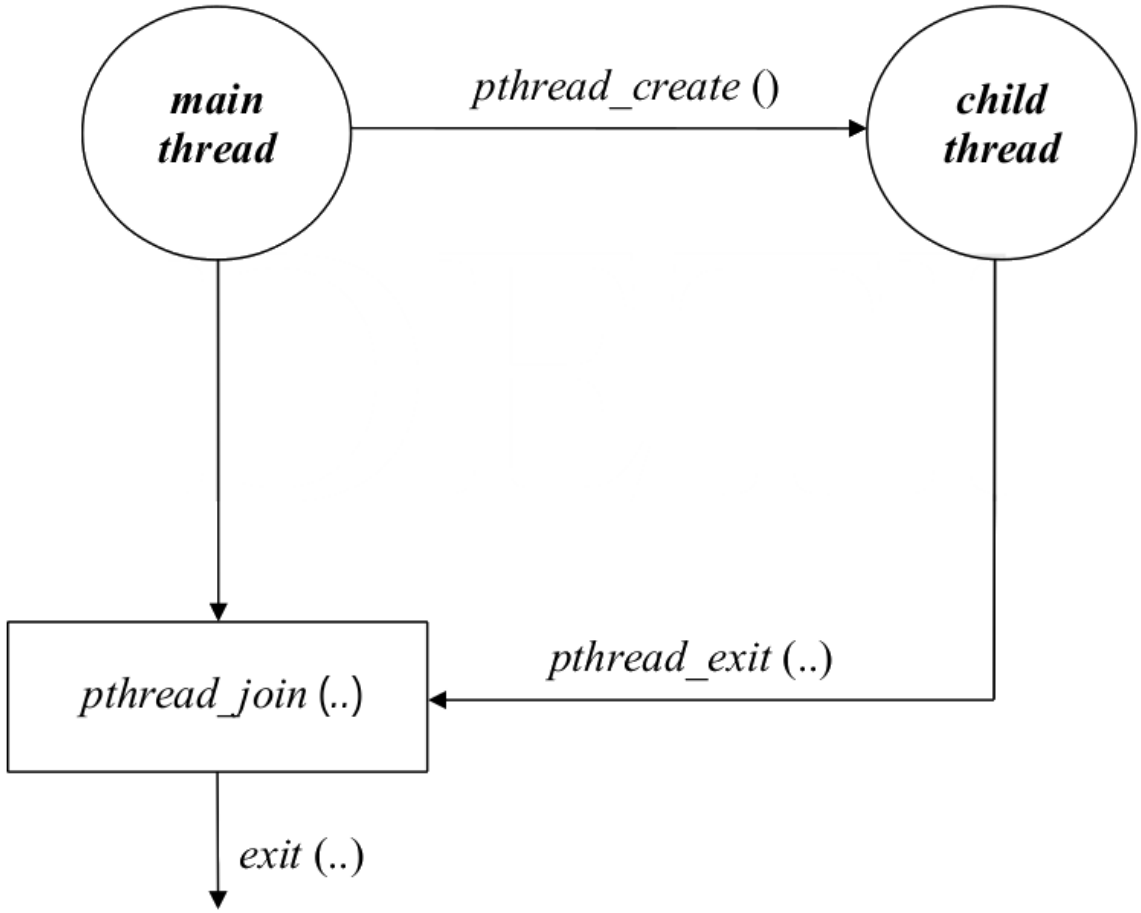


Figure 38: Exemplo do uso da libraria pthread

```c
#include <stdio.h>
#include <stdlib.h>
#include <pthread.h>

/* return status */
int status;

/* child thread */
void *threadChild (void *par)
{
    printf (”I’m the child thread!\n”);
    status = EXIT_SUCCESS;
    pthread_exit (&status);
}
```

```
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 }
```

16.5 Threads em Linux

2 system calls para criar processos filhos:

- fork:
  - cria uma novo processo que é uma **cópia integral** do processo atual
  - o address space é 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** foma 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()

```c
#include <stdio.h>
#include <stdlib.h>
#include <pthread.h>
#include <unistd.h>
#include <unistd.h>
#include <sys/types.h>

pid_t gettid()
{
    return syscall(SYS_gettid);
}

/* child thread */
int status;
void *threadChild (void *par)
{
    /* There is no glibc wrapper, so it was to be called
     * indirectly through a system call
     */

    printf ("Child: PPID: %d, PID: %d, TID: %d\n", getppid(), getpid(), gettid
        ());
    status = EXIT_SUCCESS;
    pthread_exit (&status);
}
```

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:

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

17 Process Switching

Revisitando a o diagrama de estados de um processador `multithreading`

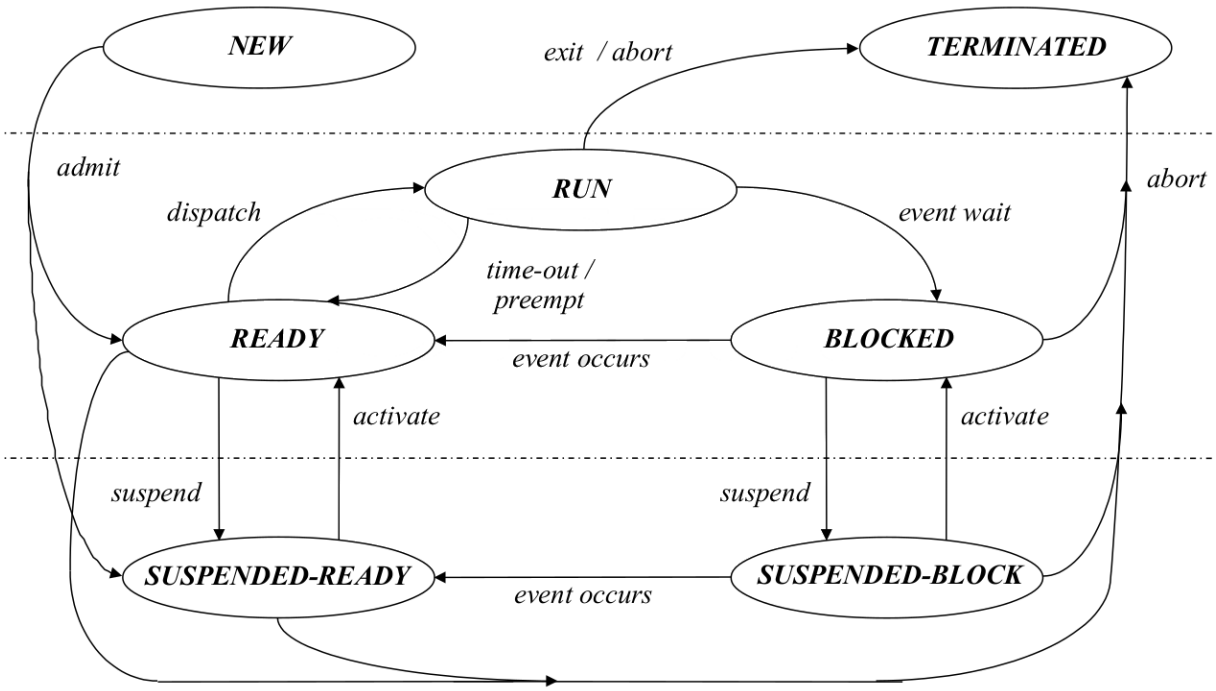


Figure 39: Diagrama de estados completo para um processador `multithreading`

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

1. `supervisor mode`
   - Todas as instruções podem ser executadas
   - É um modo **privilegiado**, **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** $\Longrightarrow$ `interrupção`
- **traps** $\Longrightarrow$ `intreeupção por software`

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

17.1 Exception Handling

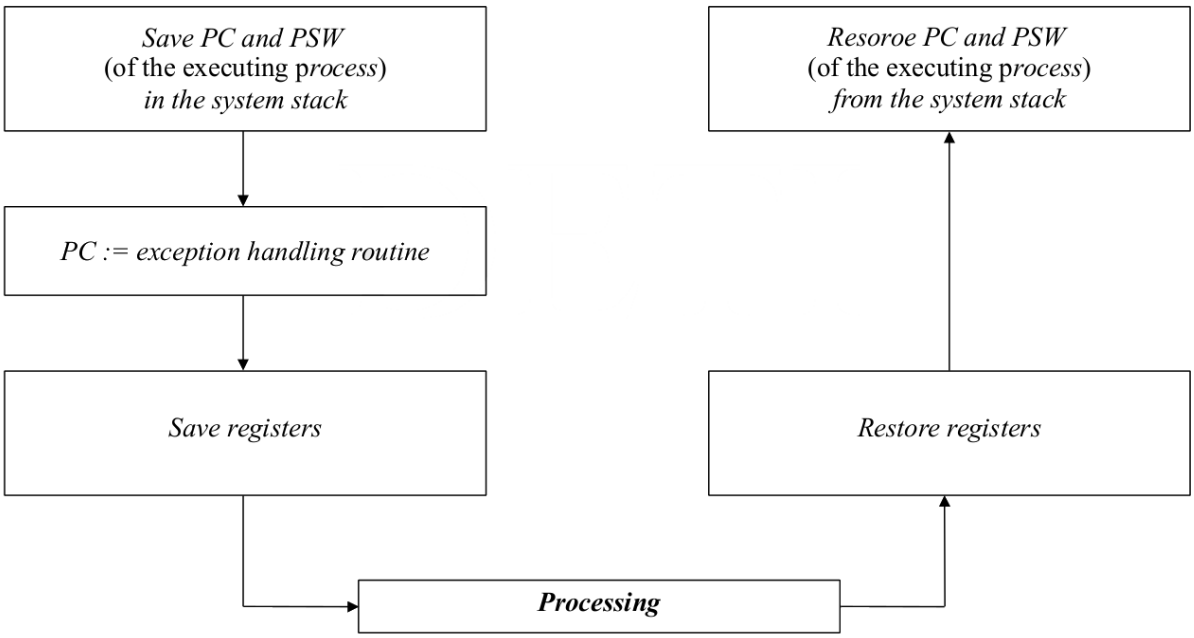


Figure 40: Algoritmo a seguir para tratar de exceções normais

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

17.2 Processing a process switching

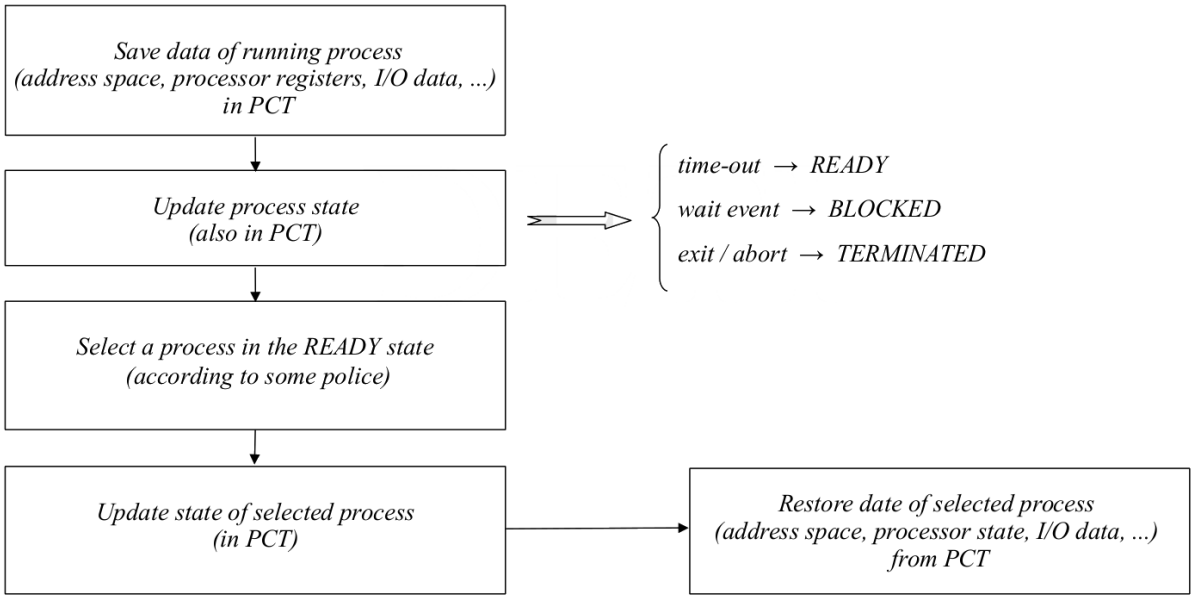


Figure 41: Algoritmo a seguir para efetuar uma process switching

O algoritmo é bastante parecido com o tratamento de exceções:

1. Salvaguardar 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

18 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?

18.1 Scheduler

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

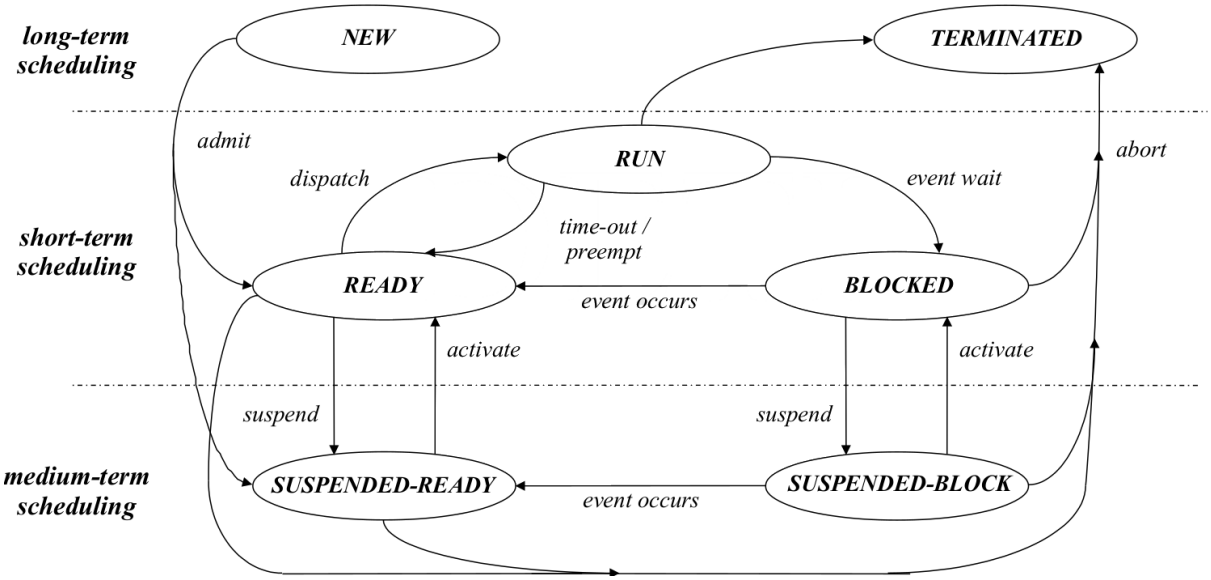


Figure 42: Identificação dos diferentes tipos de schedulers no diagrama de estados dos processos

18.1.1 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

18.1.2 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**

18.1.3 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)

18.2 Critérios de Scheduling

18.2.1 User oriented

Turnaround Time:

- Intervalo de 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 outras objetivos/parâmetros do sistema

Predictability:

- Quantiza o impacto da carga (de processos) no tempo de resposta dos 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

18.2.2 System oriented

Fairness:

- Igualdade de tratamento entre todos os processos
- Na ausência de diretivas que condicionem os processos a atender, deve ser efetuada um 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**

18.3 Preemption & Non-Preemption

Non-preemptive scheduling:

- O processo mantêm 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`
  - Tipico 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

18.4 Scheduling

18.4.1 Favouring Fearness

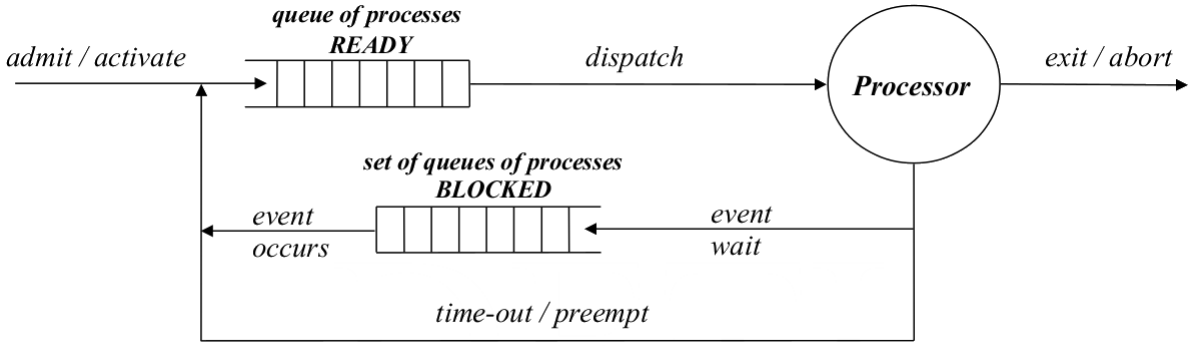


Figure 43: Espaço de endereçamento de um processo em Linux

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` $\Longrightarrow$ `first come, first-served` (FCFS)
- `preemptive scheduling` $\Longrightarrow$ `round robin`

18.4.2 Priorities

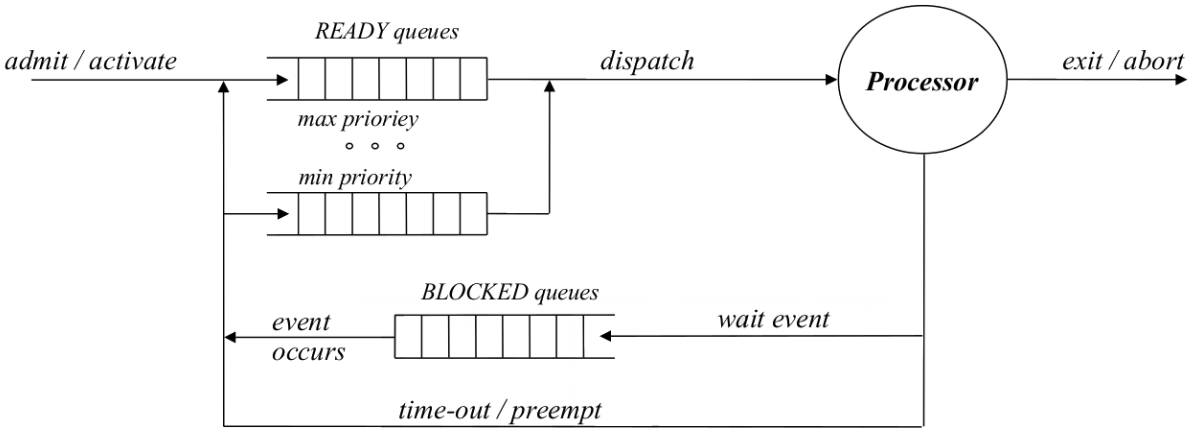


Figure 44: Espaço de endereçamento de um processo em Linux

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 $n = 1, 2, ..., N$. O average turnaround time é:

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

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 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

18.5 Scheduling Policies

18.5.1 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**

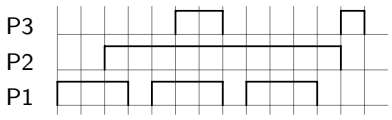


Figure 45: Problema de Scheduling

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

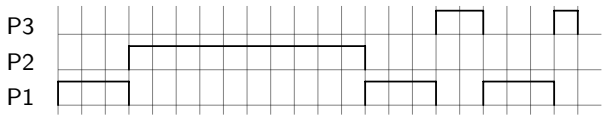


Figure 46: Política FCFS

- 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 puder terminar executando o seu último ciclo

18.5.2 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:

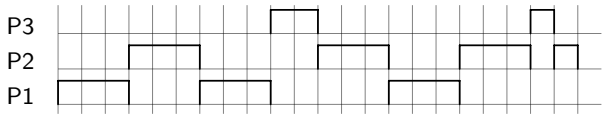


Figure 47: Política Round-Robin

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

18.5.3 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 logo prazo, **`long`**`-term scheduling` em sistemas `batch`, porque os utilizadores esperam estimar com precisão o tempo máximo que o processo necessita para ser executado

18.5.4 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_yeld`
- 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

18.6 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

19 Introdução à Gestão de Memória

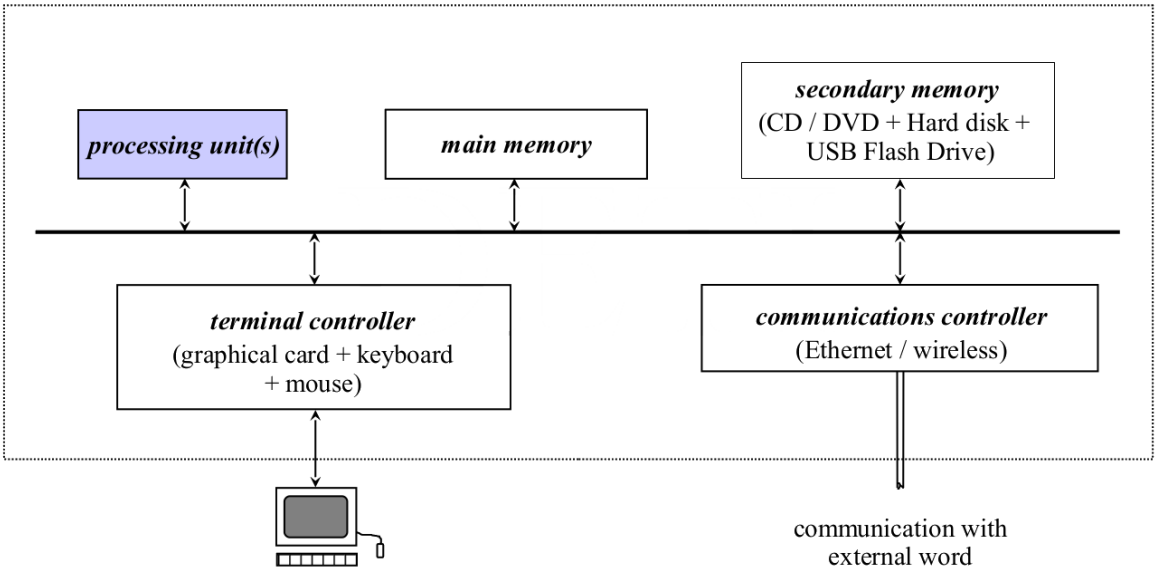


Figure 48: Relembrando o diagrama de um sistema Computacional

19.1 Porquê a gestão de memória

- Para puder executar um programa este tem de residir em **memória principal**
  - As varáveis, instruções, etc. tem de estar na memória principal, pelo menos de forma parcial
- É necessário maximizar a ocupação do processador e minimizar o tempo de resposta (`turn-around time`)
  - Ambiente **multiprogramado**
  - Ocorre **comutação de processos**
  - Existem **vários processos em memória**

> **Lei de Parkinson** *"Os programas tendem a expandir-se ocupando toda a memória disponível"*

Ou seja, apesar de o espaço disponível em memória principal ter aumentado ao longo dos anos, os mesmos problemas mantêm-se.

Supondo que a **fração de ocupação do processador** pode ser modelada de forma simplificada pela expressão

$$\%_{ocupaoCPU} = 1 - p^n$$

onde:

- p: fração de tempo em que um processo está **bloqueado à espera** que as operações de I/O, sincronização, etc terminem
- n: **número de processos** que **coexistem** de forma concorrente e a competir por recursos em memória principal

Supondo $p = 0.8$, temos:

Nº de processos em Memória Principal	% de ocupação do Processador
4	59
8	83
12	93
16	97

De forma mais geral:

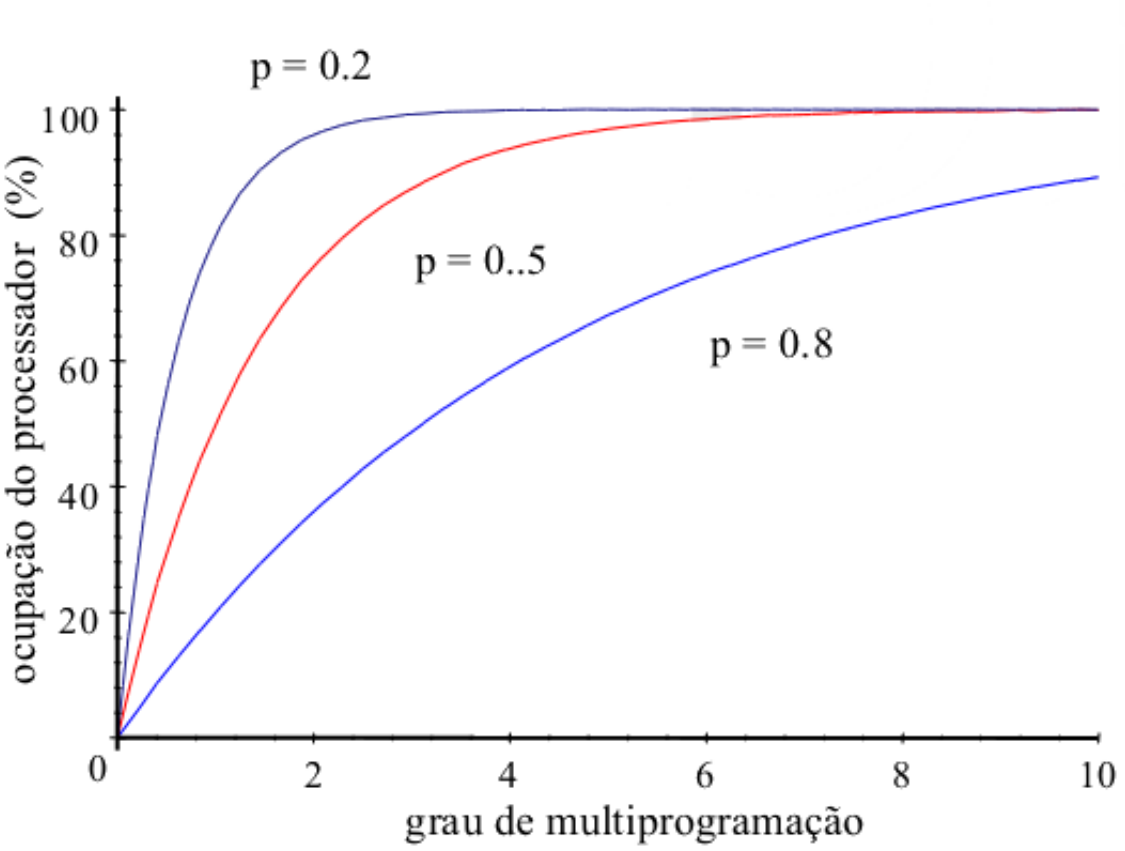


Figure 49: Grau de ocupação do processador em função do número de processos concorrentes residentes em memória principal em simultâneo

O número de processos que devem estar em memória têm de ser otimizados. O número de processos em memória depende do número de processos I/O intensivos (`I/O-bound`) ou CPU intensivos (`CPU-bound`)

19.2 Hierarquia da memória

Para melhorar a eficiência do sistema e reduzir os custos, as memórias devem ser otimizadas para as funções que vão desempenhar:

	Cache	Principal	Secundária
tamanho	pequena (dezenas de KB ou unidades de MB)	tamanho médio (centenas de MB ou unidades de MB)	Grande (dezenas, centenas ou milhares de GB)
velocidade	muito rápida	rápida	lenta
preço	cara	razoável	barata
volátil	✓	✓	x

Table 6: Comparação entre os diferentes tipos de memórias de um sistema computacional

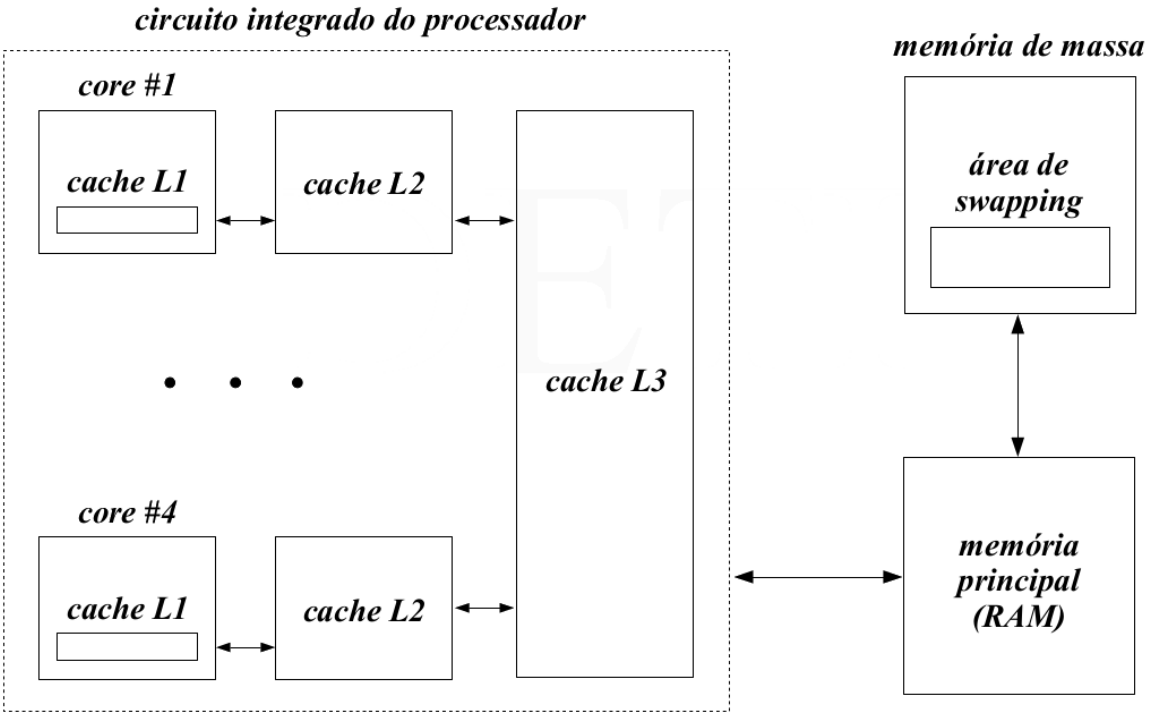


Figure 50: Hierarquia da Memória num sistema de computação

19.2.1 Memória Cache

- Contém uma cópia das **posições** e **operandos** mais frequentemente referenciadas pelo processador num passador próximo
- Existem 3 tipos de `memória cache`
  - **L1:** localizada no **IC[7] do processador**
  - **L2** e **L3:** localizadas num **IC autónomo** mas no mesmo substrato que L1
- O controlo da transferência de dados de/para a memória principal é feito de modo quase completamente **transparente** ao programador
- É útil devido ao **princípio da localidade de referência**

19.2.2 Memória Secundária

Duas funções principais:

- Armazenar de forma **não volátil** a informação (dados e programas), através de um **sistema de ficheiros** implementado no dispositivo
- **extender a memória principal** para que o tamanho desta não seja limitativo ao número de processos que podem coexistir em memória - `área de wapping`

19.2.3 Princípio da Localidade da Referência

Temporal: > Quando é acedido um endereço de memória (quer seja para r/w uma variável ou para ler uma instrução), a probabilidade de voltar a aceder a esse mesmo endereço de memória é mais elevada do que aceder a outros endereços de memória

Espacial: > Quando é acedido um endereço de memória (quer seja +ara r/w de uma variável ou para ler uma instrução), a probabilidade de aceder a offsets do endereço de memória é mais elevada do que aceder a endereços muito distantes

Estes princípios baseiam-se no facto de que quanto **mais afastada** uma instrução/operando está do endereço atual que o processador está a executar, **menos vezes será referenciado**.

O uso destes princípios no design de software e hardware tem como objetivo diminuir o tempo médio de acesso à referência.

Estes princípios foram derivados da **constatação heurística** do comportamento de um programa em execução. Conclui-se que as referências à memória durante a sua execução tendem a **concentrar-se** em **frações bem definidas do seu espaço de endereçamento** em intervalos de tempo mais ou menos longos

19.3 Gestão da memória num ambiente multiprogramado

- **objetivo:** Tirar partido dos princípios anteriores

[7] ficheiro em código fonte de compilação separada

A função principal é **controlar a transferência de dados** entre a **memória principal** e a **memória secundária**, garantindo:

- Manter o *track* das partes da **memória principal** que estão **ocupadas** e as partes que estão **livres**
- Reservar secções da memória para as necessidades dos processos
- Libertar secções da memória quando os processos terminam
- Transferir para a `área de swapping` a totalidade/parte do **espaço de endereçamento** de um processo quando a memória principal não consegue guardar todos os processos que coexistem em memória

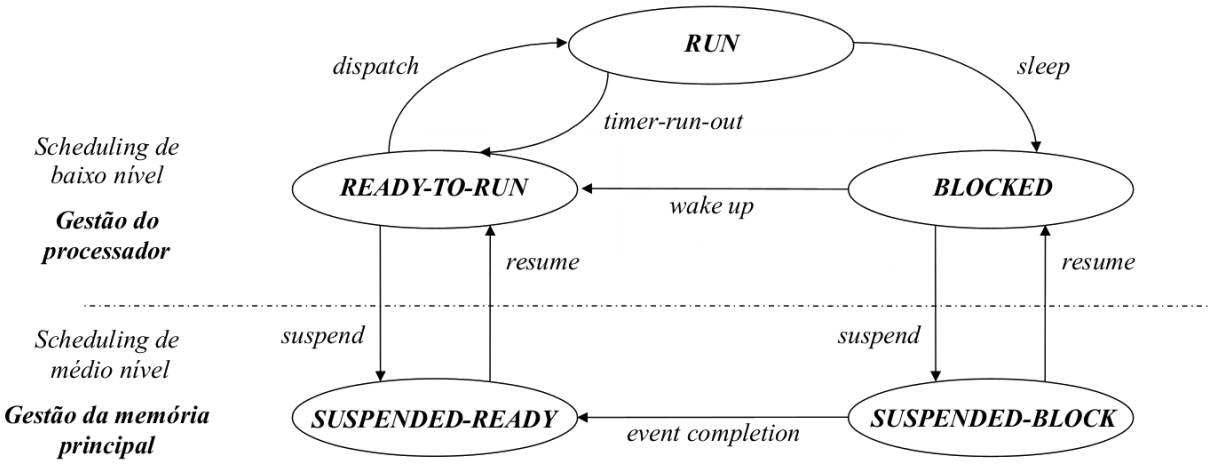


Figure 51: Diagrama da inclusão da gestão de memória com o scheduling de baixo nível do processador

19.4 Espaço de Endereçamento

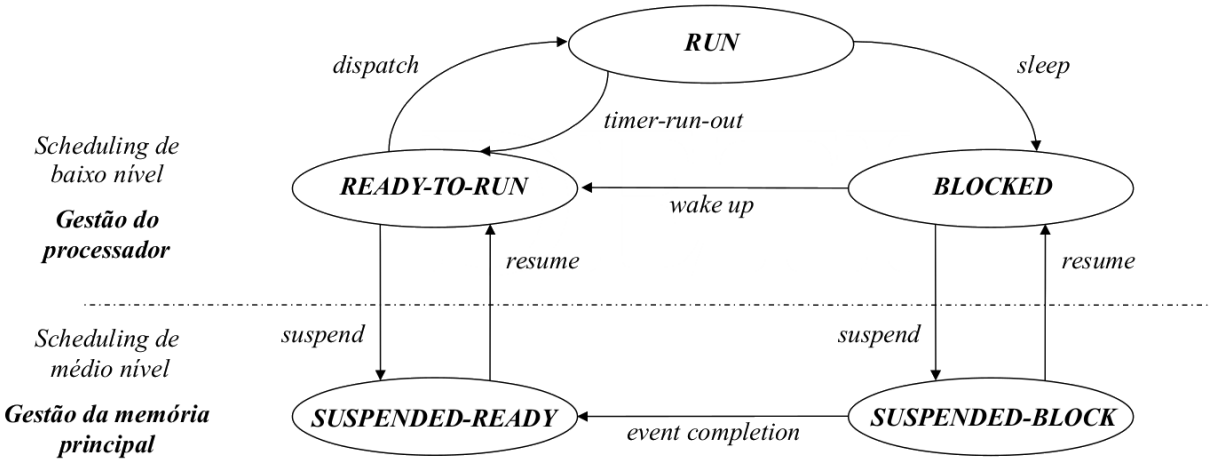


Figure 52: Construção do espaço de endereçamento de um programa após compilação e linkagem

- Os ficheiros `object` (resultantes da compilação), possuem todos os seus endereços das diversas instruções, constantes e variáveis calculados a partir do endereço 0 (início dos endereços do módulo)

Se a linkagem for **estática:**

- Após linkagem, os diferentes ficheiros objeto são reunidos num único ficheiro executável
  - São resolvidas as várias referências externas
  - As bibliotecas de sistema podem não estar incluídas na linkagem para minimizar o tamanho do ficheiro
- O `loader` constrói a **imagem binária do espaço de endereçamento do processo**
  - ficheiro executável + bibliotecas de sistema
  - Resolve todas as dependências externas que não foram incluídas no ficheiro executável no processo de linkagem

Se a linkagem for **dinâmica**, cada referência no código do processo é substituída por um `stub`:

- `stub`: pequeno conjunto de instruções que determina a localização de uma rotina
  - se a rotina estiver em memória principal, executa-a
  - se não estiver, força o seu carregamento para memória principal (e depois executa-a)
- Quando um `stub` é executado **pela primeira vez** obtém a referência para o endereço de memória da rotina
  - Substitui no **código do processo** o seu endereço pelo endereço da rotina
  - Executa a rotina

- Quando a secção de código onde era executado o `stub` é novamente atingida, a **rotina do sistema** é executada **diretamente**

Como vários processos independentes podem executar a mesma biblioteca do sistema, ao todos executarem uma cópia do código **minimiza-se** a ocupação da **memória principal**

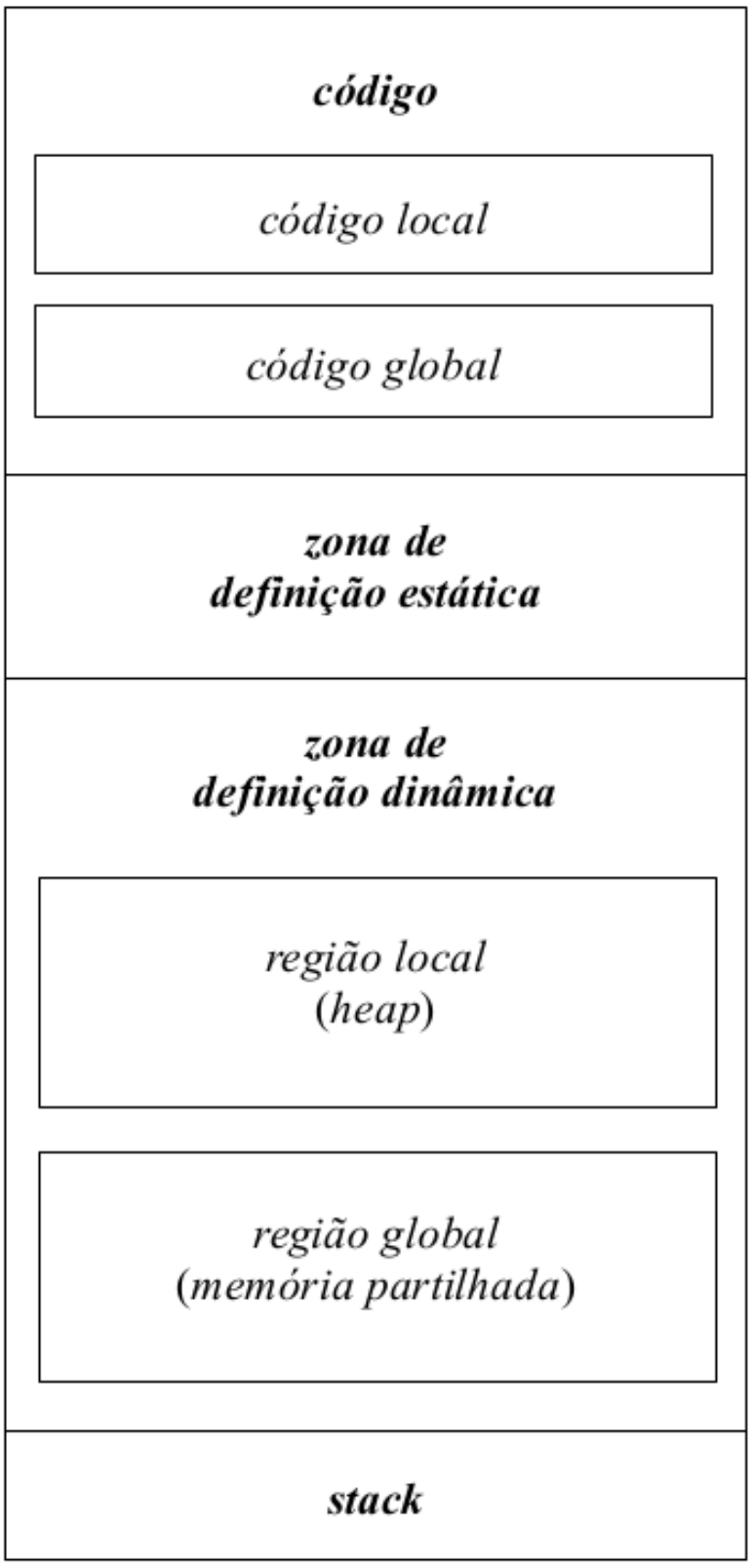


Figure 53: Diagrama da divisão do espaço de endereçamento de um programa

- As zonas de de código e definição estática de variáveis têm um **tamanho fixo**
  - Determinado pelo `loader`
- As `zonas de definição dinâmica` e a `stack` podem variar de tamanho ao longo da execução do programa
  - O espaço de memória referente à `zona de definição dinâmica` e à `stack` pode ser usada alternativamente entre eles
  - Quando o espaço para a `stack` cresce e o espaço disponível é esgotado pelo **lado da `stack`**, ocorre um `stack overflow`

19.4.1 Exemplo

Considerando o seguinte código fonte

```
1 //ficheiro fonte
2 #include <stdio.h>
3 #include <stdlib.h>
4
5 int main (void)
6 {
7     printf (”hello, world!\n”);
8     exit (EXIT_SUCCESS);
9 }
```

A produção do ficheiro objeto pode ser feita com:

```
1 gcc -Wall -c hello.c
```

E o executável gerado através de:

```
1 gcc -o hello hello.o
```

Antes da linkagem temos:

```
1 >> file hello.o
2 hello.o: ELF 32-bit LSB relocatable, Intel 80386, version 1 (SYSV), not
      stripped
```

```
1 >> objdump -fstr hello.o
2
3 hello.o:      file format elf32-i386
4 architecture: i386, flags 0x00000011:
5 HAS_RELOC, HAS_SYMS
6 start address 0x00000000
7 SYMBOL TABLE:
8 00000000 l     df *ABS* 00000000 hello.c
```

```
00000000 l     d .text            00000000
00000000 l     d .data            00000000
00000000 l     d .bss             00000000
00000000 l     d .rodata          00000000
00000000 l     d .note.GNU-stack 00000000
00000000 l     d .comment         00000000
00000000 g      F .text 0000002a main
00000000          *UND* 00000000 printf
00000000          *UND* 00000000 exit

RELOCATION RECORDS FOR [.text]:
OFFSET   TYPE              VALUE
00000014 R_386_32          .rodata
00000019 R_386_PC32        printf
00000026 R_386_PC32        exit

Contents of section .rodata:
 0000 68656c6c 6f20776f 726c6421 0a000000   hello world!....

Contents of section .comment:
 0000 00474343 3a202847 4e552920 332e332e   .GCC: (GNU) 3.3.
 0010 3120284d 616e6472 616b6520 4c696e75   1 (Mandrake Linu
 0020 7820392e 3220332e 332e312d 326d646b   x 9.2 3.3.1-2mdk
 0030 2900                                  ).
$
```

Após a linkagem temos:

```
>> file hello
hello: ELF 32-bit LSB executable, Intel 80386, version 1 (SYSV), for GNU/Linux
2.2.5, dynamically linked (uses shared libs), not stripped
```

```
>> objdump -fTR hello

hello:     file format elf32-i386
architecture: i386, flags 0x00000112:
EXEC_P, HAS_SYMS, D_PAGED
start address 0x080482d0
DYNAMIC SYMBOL TABLE:
0804829c      DF *UND* 000000e6 GLIBC_2.0      __libc_start_main
080482ac      DF *UND* 0000002d GLIBC_2.0      printf
080482bc      DF *UND* 000000c8 GLIBC_2.0      exit
080484b4 g    DO .rodata         00000004 Base         _IO_stdin_used
00000000 w    D *UND* 00000000                 __gmon_start__
DYNAMIC RELOCATION RECORDS
OFFSET   TYPE              VALUE
080495cc R_386_GLOB_DAT    __gmon_start__
```

```
080495c0 R_386_JUMP_SLOT   __libc_start_main
080495c4 R_386_JUMP_SLOT   printf
080495c8 R_386_JUMP_SLOT   exit
$
```

Podemos verificar que passam a existir mais instruções no ficheiro `object`:

- A função `main` tem de ser executada por alguém…
- É preciso construir o `argv` e o `argc` para passar à `main`
- Implica código adicional

19.4.2 Espaço de endereçamento lógico vs físico

- **espaço de endereçamento lógico:** espaço de endereçamento realocável
- **espaço de endereçamento físico:** região da memória principal onde o processo é carregado para ser executado

Uma vez que a **Imagem binária do espaço de endereçamento de um processo** é mapeada no **espaço lógico** do processo, em sistemas multiprogramados é necessário garantir:

- **mapeamento dinâmico:** capacidade de conversão em `run-time` de um **endereço lógico** num **endereço físico**
  - Passo intermédio para permitir o armazenamento do espaço de endereçamento de um processo em qualquer região da memória principal (incluindo a sua mudança em `run-time`)
- **proteção dinâmica:** impedimento em `run-time` de referenciar endereços que estão localizados fora do espaço de endereçamento do processo (aka, `core dumped`)

20 Arquitecturas de Memória Particionadas

20.1 Arquitectura de partições fixas

- A memória principal (restante) é dividida num **conjunto fixo de partições**
  - Mutualmente exclusivas
  - Não necessariamente iguais
- Cada uma das partições contém o espaço de endereçamento físico de um processo

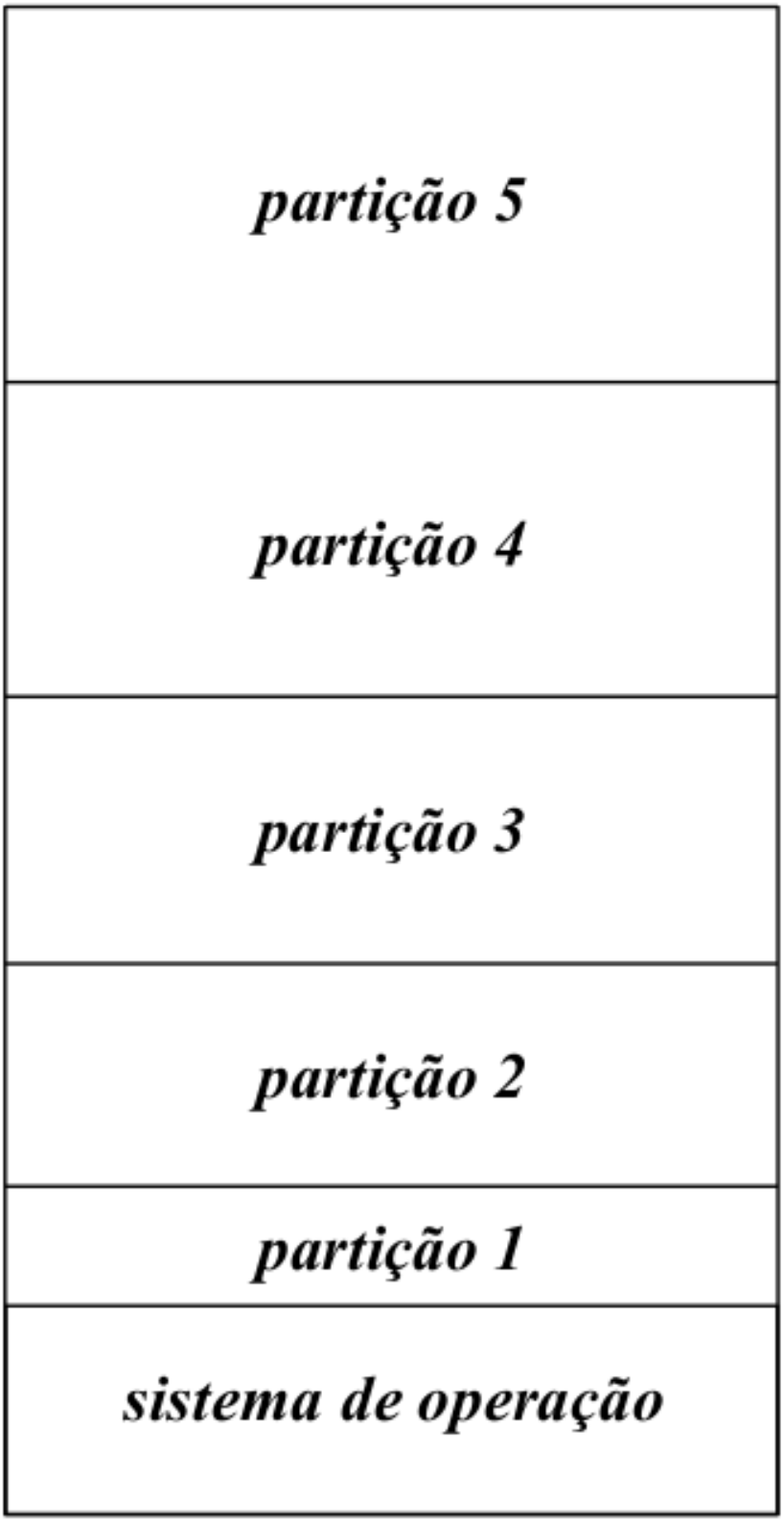


Figure 54: Divisão em partições fixas mutuamente exclusivas com diferentes tamanhos

- A memória principal vai sendo dividida à medida que vai recebendo solicitações
- Podem ser utilizadas diferentes filosofias de escalonamento
  - **Valorização do critério de justiça**
    * Escolher o primeiro processo da fila de espera dos processos `Suspended-Ready` cujo **espaço de endereçamento cabe na partição**
  - **Valorização da ocupação da memória principal**
    * Escolher o primeiro processo da fila de espera dos processos `Suspended-Ready` com o **espaço de endereçamento de tamanho maior** que caiba na partição
    * Corre-se o risco de adiamento indefinido de processos com espaço de endereçamento pequeno (`starvation`)
    * Por isso associa-se um contador a cada processo
      · o contador é incrementado a cada passagem
      · contador > valor pré-definido $\implies$ **processo já não pode ser descartado**
      · Passa-se a aplicar a 1ª regra

20.1.1 Vantagens e Desvantagens

Vantagens:

- `simples de implementar`: não exige hardware ou estruturas de dados especiais para gerir a memória
- `eficiente`: a seleção pode ser feita rapidamente com qualquer das políticas acima

Desvantagens:

- `fragmentação interna da memória principal`: o espaço restante de cada partição que não é alocado pelo processo é desperdiçado
- `política direcionada para certos tipos de aplicações`:
  - O tamanho das partições é fixo
  - A única maneira de ser evita o desperdício de memória é através da adequação do tamanho das partições ao tipo de processos a utilizar
    * número de processos
    * tipo de processos
    * tamanho do seu espaço de endereçamento
  - torna a solução pouco generalizável

20.2 Arquitectura de posições variáveis

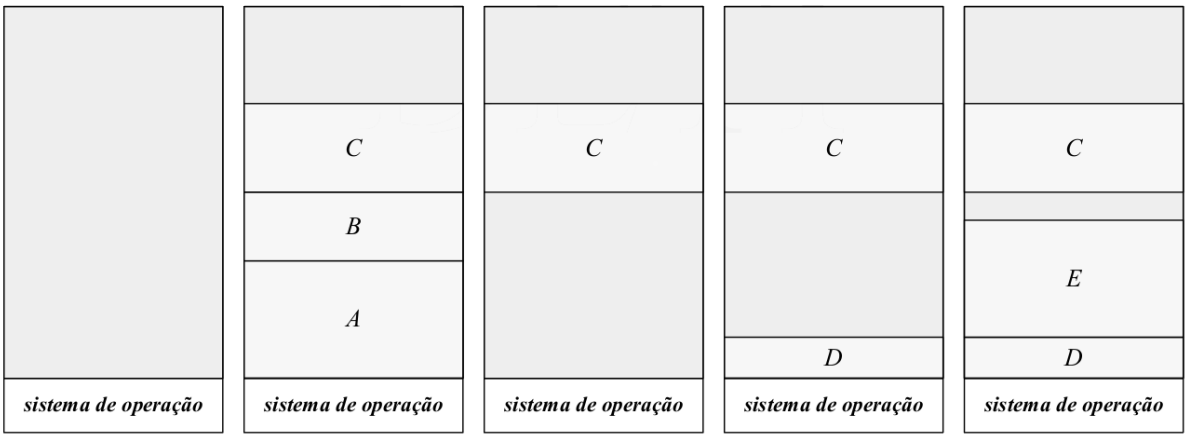


Figure 55: Divisão da memória em partições de tamanho variável

- Toda a parte disponível da memória constitui um bloco único
  - Sucessivamente são alocadas alocadas/atribuídas/reservadas regiões de tamanho suficiente para conter o espaço de endereçamento dos processos que vão sendo criados/`swapped-in`
  - Posteriormente ao processo terminar, os espaços de endereçamento deixam de ser usados e são libertados

20.2.1 Gestão do espaço

Como a memória é reservada dinamicamente, o sistema de operação tem de manter um registo atualizado de:

- **Regiões livres:**
  - regiões ainda disponíveis na memória para armazenar o espaço de endereçamento dos novos processos:
    * criados
    * transferidos da á`rea de swapping`
- **Regiões Ocupadas:**
  - localiza as regiões que foram reservadas para armazenamento do espaço de endereçamento dos processos que residem em memória principal

Usando uma lista biligada (ou simplesmente ligada) para cada região. Estas listas:

- Não vão sempre indicar os espaços livres
- Podem ser alargadas
- A sua gestão em feita em blocos
  - Posso adicionar blocos à lista medida que vou libertando blocos da memória principal

- Se o bloco a libertar estiver contíguo a um (ou dois) bloco(s) livre(s) tenho de modificar a lista (e não simplesmente introduzir um novo bloco)

Problema: Se a região de memória reservada for exatamente a suficiente para o espaço de armazenamento do processo, existe o risco de `fragmentar` o disco em regiões de memória tão pequenas que não podem ser utilizadas. Para complicar, estas partições seriam introduzidas na lista de regiões livres tornando a lista mais complexa e aumentando o seu custo de processamento

Solução: A memória principal é dividida em múltiplos de **blocos de tamanho fixo** que constituem a unidade de trabalho para a alocação de partições

20.2.2 Exemplo

Considerando o seguinte diagrama e tabela que mostra a distribuição de 3 processos em memória, juntamente com o espaço ocupado pelo sistema operativo e, temos:

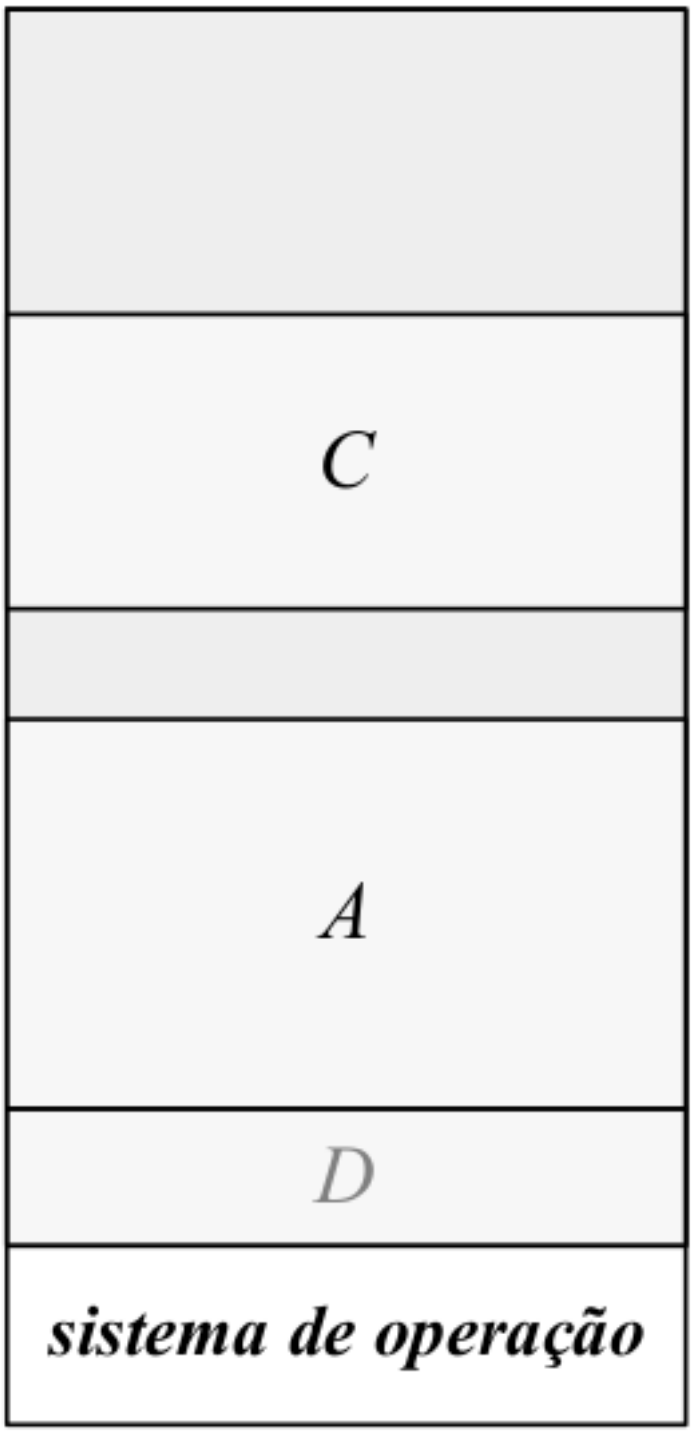


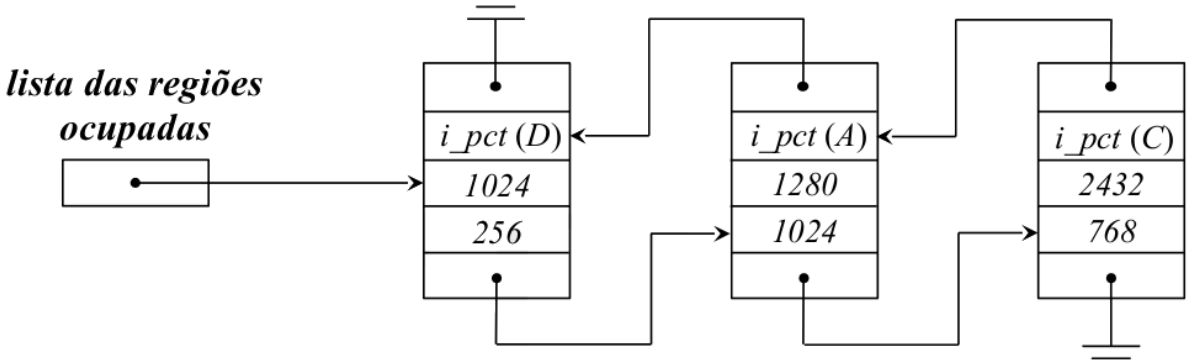
Figure 56: Diagrama da Memória Particionada

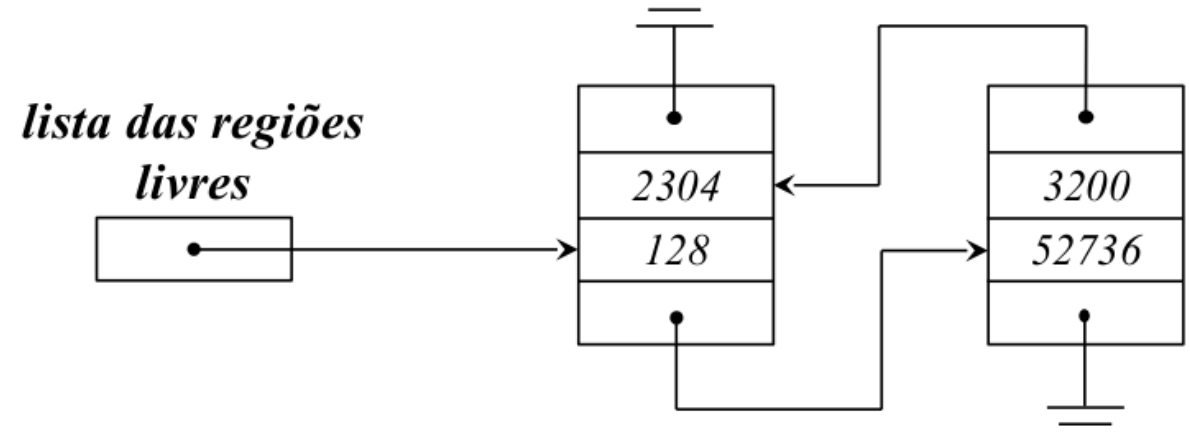
Table 7: Distribuição da ocupação da memória

	Tamanho (bytes)
Memória Principal	256 M
Sistema de Operação	4M
Unidade de reserva [^3]	4K
Processo A	4M
Processo C	3M
Processo D	1M

Tamanho mínimo de uma partição. Todos os endereços em memória são múltiplos da unidade de reserva

A obtenção da `lista das regiões ocupadas` e da `lista das regiões livres` pode ser determinada de forma trivial:





20.2.3 Políticas de Escalonamento

A **valorização do critério de justiça** é a disciplina de escalonamento mais adotada.

- É escolhido o **primeiro** processo da **fila de espera** dos processos `Suspended-Ready` cujo espaço de endereçamento pode ser colocado em memória principal
- O **principal problema** de uma arquitectura de partições variáveis é o `grau de fragmentação externa` que é produzido na memória principal
  - sucessivas **alocações** e **libertações** das partições do espaço de endereçamento dos processos
  - em casos críticos, podemos ter situações em que apesar de **haver memória livre em quantidade suficiente**, ela **não é continua** e não é possível alocar espaço em memória para um novo processo
- A solução passa por efetuar `garbage collection`
  - agrupar todas as posições/partições de memória livres num dos extremos da memória
  - obriga a mudança em memória de todos os processos realocados
  - exige a paragem de todo o processamento
  - se a memória for grande, tem um tempo de execução elevado

Para a escolha da região de memória principal a ser reservada para o armazenamento do processo, dominam as seguintes políticas: 1.`first fit` - A lista de regiões livres é pesquisada desde o princípio - A região de memória a ocupar é a primeira região com tamanho suficiente encontrada 2. `next fit` - Variante do `first fit`, com os mesmos princípios de decisão - No entanto, a pesquisa é iniciada do ponto de paragem da pesquisa anterior 3. `best fit` - A lista de regiões livres é pesquisada na sua totalidade - Escolhe-se a região mais pequena de tamanho igual ou maior ao espaço de endereçamento do processo 4. `worst fit` - A lista de regiões livres é pesquisada na sua totalidade - A região escolhida é a maior região existente

Algumas considerações:

- Uma política que seja boa/rápida a inserir elementos na fila será má/lenta a removê-la
- É difícil encontrar soluções que sejam tão rápidas a inserir como a remover da fila de partições
- Os diferentes métodos possuem diferentes desempenhos:
  - grau e tipo de fragmentação causado
  - eficiência na reserva/libertação do espaço

20.2.4 Vantagens vs Desvantagens

Vantagens:

- `Geral`: O âmbito da sua aplicação é independente do:
  - tipo de processos que vão ser executados
  - número
  - e tamanho do seu espaço de endereçamento (com algumas limitações)
- `pouco complexo`
  - não exige `hardware` especial

  - as estruturas reduzem-se a listas biligadas

Desvantagens:

- `Grande fragmentação externa da memória principal`
  - Uma fração da memória principal acaba por ser desperdiçada
  - É dividida em regiões reduzidas que não são úteis
  - O desperdício de memória pode chegar a um terço (regra dos 50%)
- `Pouco Eficiente`
  - Não é possível desenvolver algoritmos que sejam eficientes quer a:
    * **reservar/alocar espaço**
    * **libertar espaço**

21 Organização da memória real

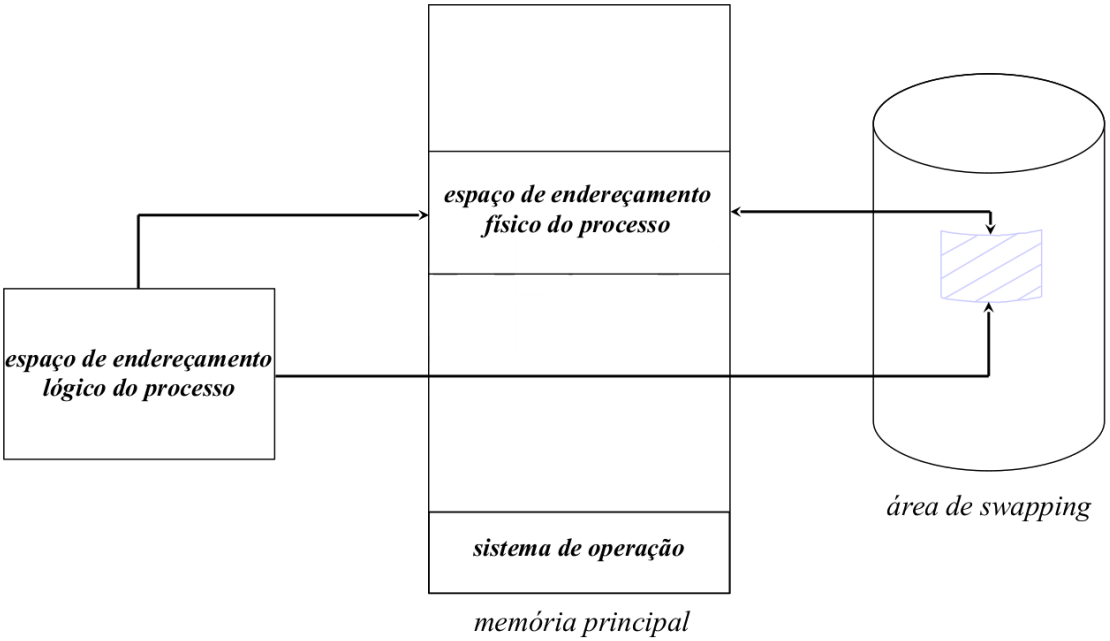


Figure 57: Espaço de endereçamento real de um processo

Existe uma correspondência biunívoca [8] entre o **espaço de endereçamento lógico** de um processo e o **espaço de endereçamento físico** de um processo. Isto implica

- O **espaço de endereçamento de um processo é limitado**

[8] De um para um

  - O espaço de endereçamento de um processo nunca pode ser superior ao tamanho de memória principal disponível
  - Os mecanismos que o tentem fazer devem ser bloqueados

- O **espaço de endereçamento físico de um processo deve ser contíguo**
  - Não e uma condição estritamente necessária
  - Simplifica e torna mais eficiente se o espaço de endereçamento de um processo for obrigado a ser contíguo

- A **existência de uma área de swapping**
  - Serve como extensão da memória principal
  - Armazena espaços de endereçamentos de processos que não podem residir em memória principal por falta de espaço

21.1 Tradução de um endereço lógico num endereço físico

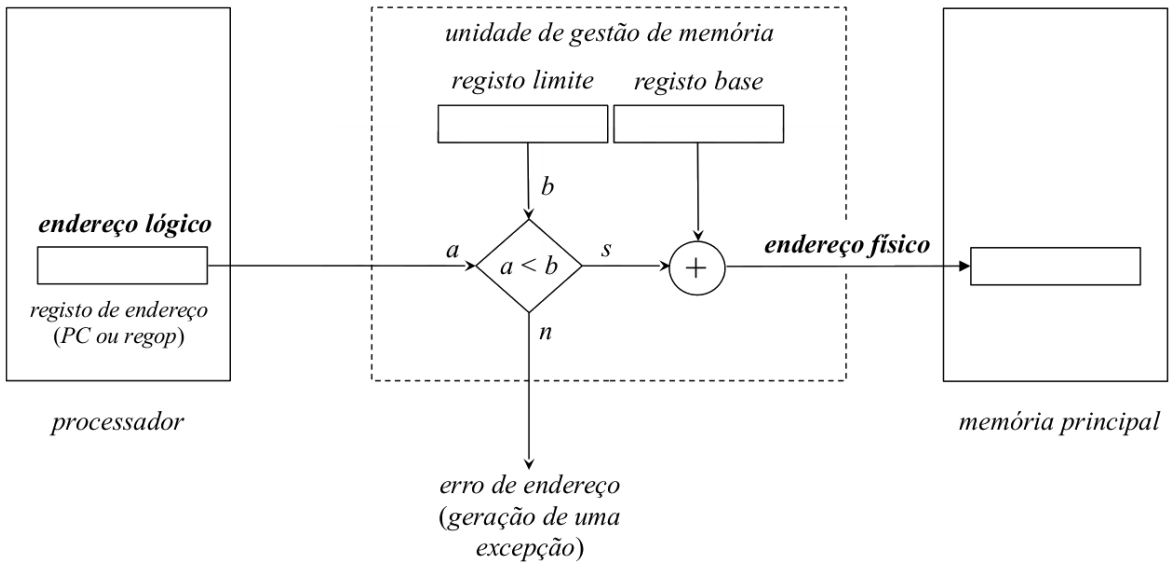


Figure 58: Tradução de um endereço lógico num endereço físico

- `registo base`: endereço do início da região de memória principal onde está alojado o espaço de endereçamento físico do processo
- `registo limite`: tamanho em *bytes* do espaço de endereçamento

Na comutação de processos:

- `dispatch` carrega o registo base e o registo limite da tabela de controlo de processos do processo que vai ser calendarizado para discussão

Sempre que há uma referência à memória, o **endereço lógico** comparado com o **registo limite** e:

- Se for **maior** $\Longrightarrow$ **referência inválida**
  - Acesso à memória nulo é executado (aka `dummy cycle`)
  - É gerada uma exceção por erro de endereço
- Se for **menor** $\Longrightarrow$ **referência válida**
  - a referência aponta para dentro do espaço de endereçamento do processo
  - o conteúdo do registo base é adicionado ao endereço lógico para produzir o endereço físico

21.2 Memória real e o ciclo de vida de um processo

Depois de carregado o Sistema Operativo, o que resta da memória principal é usado para conter o espaço de endereçamento dos diferentes processos

21.2.1 Criação de um processo

- O processo está no estado `CREADTED`
- São inicializadas as estruturas de dados destinadas a geri-lo
  - A imagem binária do seu espaço de endereçamento é construída
  - O valor do campo `registo limite` da entrada da tabela de controlo de processos é determinado
- Se houver espaço em memória
  - o espaço de endereçamento do processo é carregado
  - o campo `registo base` é atualizado com o endereço inicial da região reservada
  - o processo transita para o estado `Ready-to-Run` e é colocado na respetiva fila de espera
- Se não houver espaço em memória
  - O processo transita para o estado `Suspended-Ready`
  - O processo é colocado na respetiva fila de espera
  - O seu espaço de endereçamento é armazenado temporariamente na `área de swap`

21.2.2 Ciclo de Vida do processo

- Ao longo da sua execução, o espaço de endereçamento do processo pode ser deslocado temporariamente para a `área de wapping`.
  - `Ready-to-Run` $->$ `Suspended-Ready`
  - `Blocked` $->$ `Suspended-Blocked`
- Sempre que há espaço em memória
  - Um dos processos presentes na fila de espera dos processos `Suspended-Ready` é selecionado
  - O seu espaço de endereçamento é carregado
  - O campo `registo base` da entrada da **tabela de controlo de processos** é atualizada com o endereço inicial da região reservada

  - O processo é colocado na fila de espera `Ready-to-Run`, transitando para esse estado

- Caso a lista de espera `Suspended-Ready` estiver vazia e existirem processos na fila de espera dos processos `Suspended-block`, um desses processos pode ser selecionado
  - À semelhança da transição `Suspended-Ready` para `Ready`, na transição `Suspended-Blocked` para `Blocked` as mesmas inicializações são feitas

21.2.3 Fim de Vida do processo

- O processo transita para o estado `Terminated`
- O seu espaço de endereçamento é transmitido para a `área de swapping` (se não estiver lá), para aguardar o fim das operações

22 Organização da memória virtual

- Num sistema com memória virtual, o `espaço de endereçamento lógico` e o `espaço de endereçamento físico` de um processo estão **totalmente dissociados**

Como consequência:

- `O espaço de endereçamento de um processo não estálimitado àmemória física`
  - O espaço de endereçamento virtual é “ilimitado”
  - Podem criar-se mecanismos que permitam a um processo ocupar mais do que a memória principal disponível
- `Não continuidade` **`do`** `espaço de endereçamento físico`
  - O espaço de endereçamento dos processos podem estar dispersos por toda a memória
    * quer os blocos sejam de tamanho fixo ou variável
  - Garante-se uma ocupação mais eficiente do espaço disponível
- `Área de swapping`
  - Serve como extensão da memória principal
  - Guarda uma imagem atualizada dos espaços de endereçamento dos processos que coexistem de forma concorrente
  - guarda também as variáveis dinamicamente alocadas:
    * stack
    * zona de definição estática

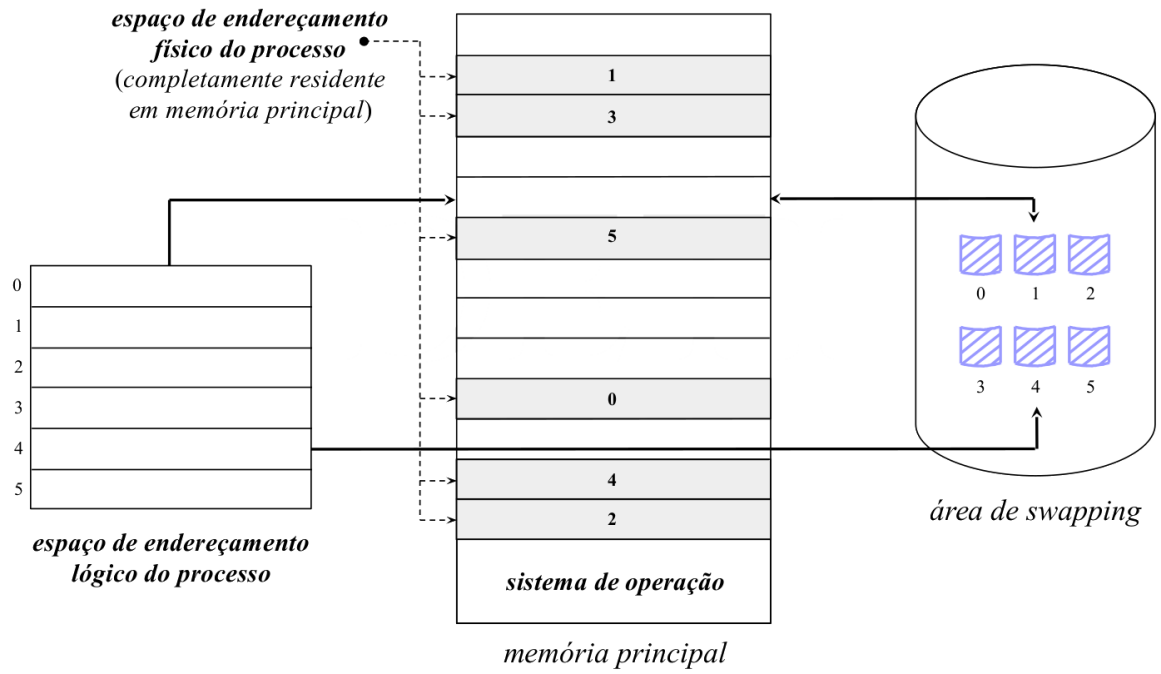


Figure 59: Espaço de endereçamento completamente em memória virtual

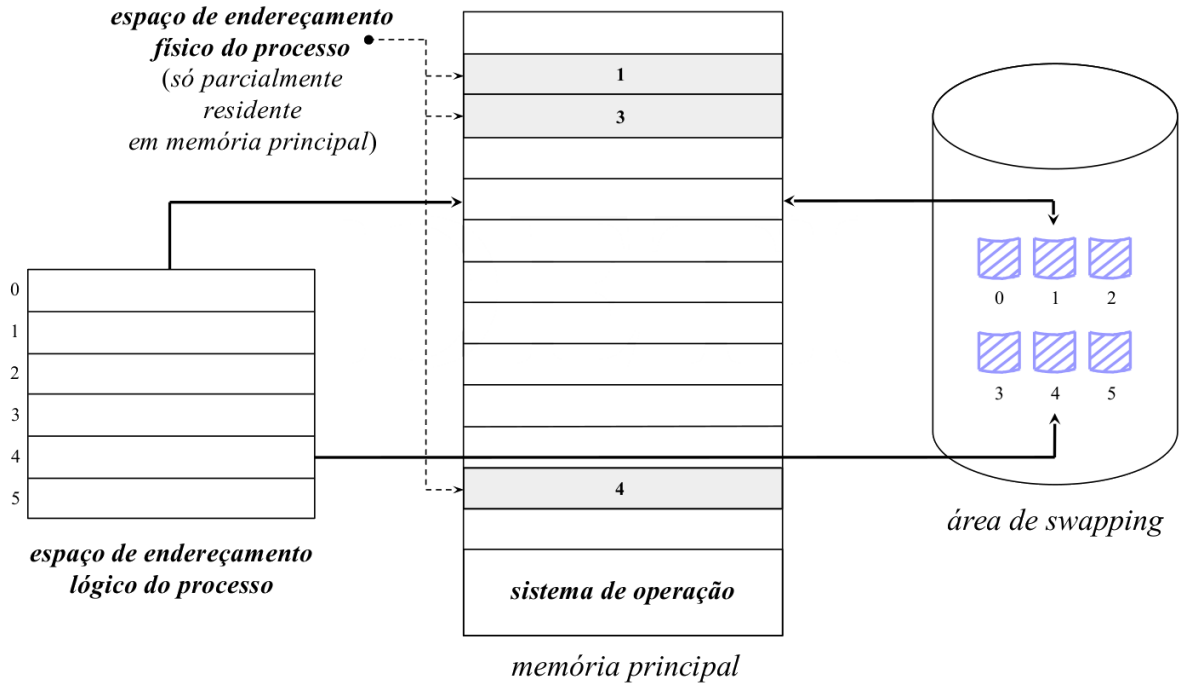


Figure 60: Espaço de endereçamento apenas parcialmente em memória virtual

22.1 Tradução de um endereço lógico num endereço físico

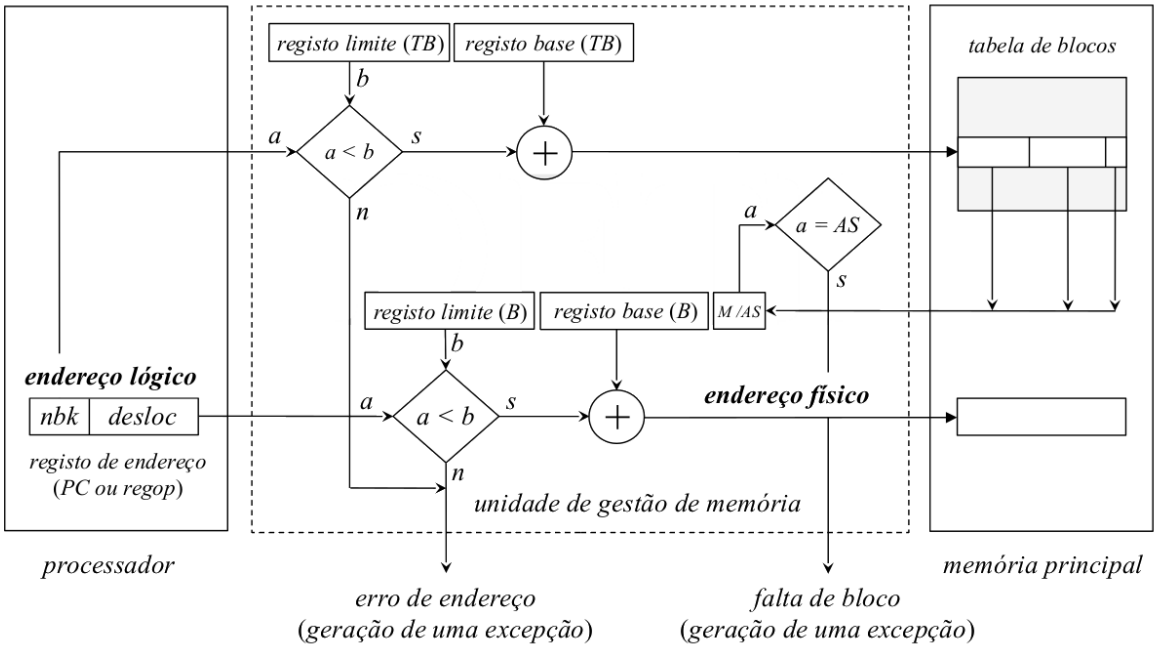


Figure 61: Diagrama de blocos da decomposição de um endereço lógico num endereço físico

O endereço lógico é formado por dois campos:

- **nbk:** identificador de um bloco específico
- **desloc:** localiza uma posição de memória concreta dento do bloco, através do cálculo da distância ao seu erro

A **unidade de gestão de memória** contém dois pares de `registos base` e `limite`

1. Associado com a **tabela de blocos do processo**:
    - descreve a localização dos vários blocos me que o espaço de endereçamento do processo está dividido
2. Descrição de um bloco particular

Quando ocorre uma **comutação de processos** a operação de `dispatch` carrega o `registo base` e o `registo limite` da tabela de blocos com os valores na tabela de controlo de processos (associada com o processo que vai ser calendarizado para execução)

```
- O valor do ‘registo base da tabela de blocos‘ representa o **endereço do início da região de memória principal** onde está alojada a tabela de blocos do processo
- O valor do ‘registo limite‘ está relacionado com o número de entradas na tabela
```

22.1.1 Acesso à memória

- Decomposto em 3 fases

1. Campo `ndk` do `endereço lógico` é comparado com o valor do **registo limite da tabela de blocos**
   - $ nbk < registo limite (TB) $
     - nbk é adicionado ao conteúdo do `regiso base da tabela de blocos` para produzir o endereço da entrada da tabela de blocos
   - $ ndk >= registo limite (TB) $
     - A referência é inválida e não é efetuado nenhuma acesso à memória
     - A instrução é interrompida por uma instrução de acesso à memória nulo *(dummy cycle)*
     - Gera-se uma exceção por erro de endereço: `segmentation fault`
2. É efetuada a avaliação do registo `M`/`AS`(*Memória/Área de Swap*)
   - se `M`: os **campos** da entrada da **tabela de blocos** referenciada são transferidos para os **registos respetivos** da unidade de gestão de memória.
   - se `AS`:
     - o bloco não está **atualmente em memória**
       * tem de ser transferido para a instrução puder continuar
     - Instrução finalizada com um acesso à memória nulo
     - Gera-se uma exceção por falta de bloco
       * será responsável por iniciar a transferência dos blocos da swap para a memória principal
     - Processo transita para o estado `blocked`
3. o campo `desloc` (deslocamento) do `endereço lógico` é comparado com o valor do `registo limite (B)` (do bloco)
   - $ desloc < registo limite (B) $ **referência válida
     - A referência é efetuada para dentro do espaço de endereçamento do bloco
     - `desloc` é adicionado ao conteúdo do registo base do bloco para produzir o endereço físico
   - $ desloc >= registo limite (B) $ **referência inválida**
     - é efetuado um acesso à memória nulo *(dummy cycle)*
     - gera-se uma exceção por erro de endereço `segmentation fault`

A gestão do espaço de memória principal usando uma organização de memória virtual possui a vantagem de permitir maior versatilidade, mas também possui o custo associado de que **cada pedido de acesso à memória (r/w) requer dois acessos para puder ser executado**

- **1ª Acesso:**
  - Referencia a entrada da tabela de blocos do processo, usando o campo `nbk` do endereço lógico como endereço do bloco em memória que contém o endereço da posição de memória que se quer ler/escrever
- **2º Acesso:** É feita referência à posição de memória específica (que se deseja efetivamente aceder)

  - O cálculo do seu endereço é efetuado adicionando o campo `desloc` do endereço lógico ao endereço que corresponde ao início do bloco em memória

- A organização em memória virtual causa um **fracionamento do espaço de endereçamento lógico** do processo.
- Os blocos/frações são tratadas dinamicamente como sub espaços de endereçamento **autónomos** numa organização de memória real
  - A memória real pode estar organizada em partições físicas ou em partições variáveis
- A diferença entre uma organização de memória virtual vs uma organização de memória real é que passas a existir a possibilidade de ocorrer o acesso a um bloco que atualmente não reside em memória principal
  - Nestas condições o sistema é capaz de anular a instrução de acesso atual
  - Meter em marcha a sequência de instruções que permite carregar esse bloco para memória principal
  - repetir a instrução assim que o bloco for carregado

Evidentemente, a necessidade do duplo acesso à memória pode ser minimizada tirando partido do **Princípio da localidade da referência:**

- Os acessos tenderão a estar concentrados num conjunto bem definido de blocos durante grandes intervalos de tempo de execução do processo
- A MMU faz caching do conteúdo das entradas da tabela de blocos que forma ultimamente referenciadas, usando uma memória associativa *(`translation lookaside buffer (TLB)`)*:
  - Cada acesso passa assim a ser um:
    * `hit`:
      · a entrada está armazenada no processador
      · o acesso é interno
      · não é referenciada memória na 1ª fase
    * `miss`:
      · a entrada não está armazenada na TLB
      · é preciso um acesso externo à memória principal na 1ª fase
- O tempo médio de acesso a uma instrução/operado tende para:
  - Acesso ao TLB + acesso à memória principal

22.2 Ciclo de vida de um processo

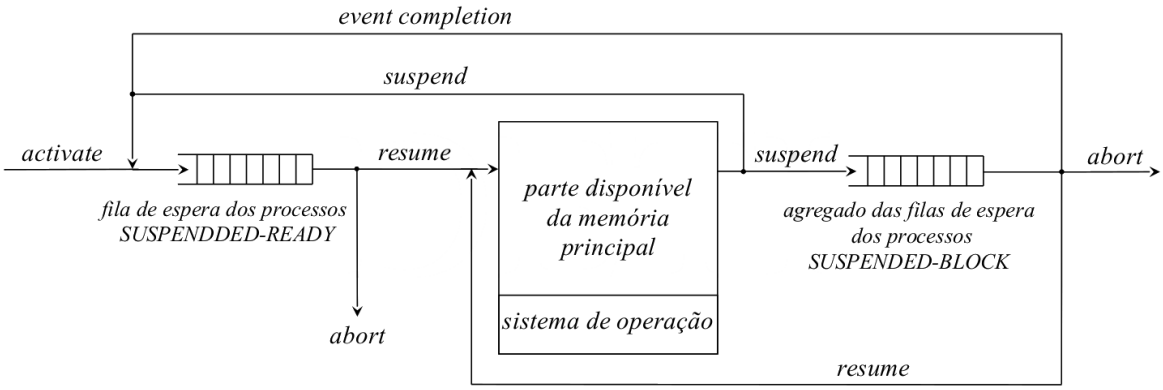


Figure 62: Diagrama de eventos e estrutura de uma organização em memória virtual

22.2.1 Criação de um processo

- estado: `CREATED`
- são inicializadas a estruturas de dados destinadas a geri-lo
  - É construída a imagem binária do seu espaço de endereçamento
  - É transferida para a área de swapping a sua parte variável
  - A tabela de blocos associada é organizada
    * Se existir espaço livre em memória é carregado em memória principal:
      · o 1º bloco de código do processo
      · o bloco da stack
      · as entradas correspondentes da tabela de blocos são atualizadas
      · estado: `READY-TO-RUN`
      · colocado na fila de espera de processos prontos a serem executados
    * Se não existir:
      · estado: `SUSPENDED-READY`
      · colocado na fila de espera de processos suspensos mas prontos a serem transferidos para memória principal e executados
- Os escalonadores tentam sempre garantir que o registo PC, a stack estão em memória e o primeiro bloco de instruções de estão em memória - São as condições principais para que um programa possa ser executado

22.2.2 Ao longo da execução

- Se ocorrer um acesso a um bloco não residente em memória principal:

  - estado passa a `BLOCKED`
    * permanece `BLOCKED` enquanto ocorre a transferência do bloco para memória principal
    * quando a transferência terminar, passa a `READY-TO-RUN`
    * é colocado na fila de espera de processos `READY-TO-RUN`

- Os blocos residentes na memória principal pertencentes a um mesmo processo podem ser `swappeed-in`
  - os seus blocos são "movidos" para a área de swap
  - o estado passa de:
    * `READY-TO-RUN` -> `SUSPENDED-READY`
    * `BLOCKED` -> `SUSPENDED-BLOCK`

- Sempre que há espaço memória:
  - um dos processo presentes na fila de espera `SUSPENDED-READY` é selecionado
  - A tabela de blocos e um grupo de blocos do seu espaço de endereçamento são carregados
  - As entradas correspondentes na tabela são atualizadas com os endereços iniciais das regiões reservadas
  - o processo é colocado na fila de espera de processos `READY-TO-RUN`

- Se a lista `SUSPENDED-READY` estiver vazia e **houver processos na fila de espera** dos processos `SUSPENDED-BLOCK`
  - Pode ser selecionado um destes processos
  - Passa para o estado `BLOCKED` e é inserido na respetiva lista de espera

22.2.3 Término de um processo

- estado: `TERMINATED`
- A imagem do seu espaço de endereçamento residente na área de swapping (ou pelo menos a sua parte variável) é atualizada
  - libertação de todos os blocos existentes em memória principal
  - Aguarda pelo fim das operações

22.3 Exceção por falta de bloco

- A rotina de serviço a esta interrupção/exceção é responsável por em marcha as ações que permitam:
  - a transferência desse bloco da área de swapping para a memória principal
  - a repetição da instrução que produziu a referência
- Estas operações são realizadas de forma totalmente transparente ao utilizador

Se um processo estivesse continuamente a gerar exceções por falta de bloco:

- o ritmo de processamento seria muito lento
- o `throughput` do sistema computacional seria mais baixo

Isto não acontece (pelo menos com tanta frequência devido à hierarquia da memória e principalmente há `Translation Lookaside Bffer` da unidade de memória, usando o princípio da localidade de referência.

Apesar da fração do espaço de endereçamento variar com o tempo, o custo em carregar vários blocos pontualmente para memória, relativamente a outras soluções é reduzido, podendo progredir a execução do processo praticamente sem ocorrerem faltas de bloco.

Situações de falta de página degridem muito a qualidade do sistema

22.3.1 Sequência de instruções

1. Salvaguardar o contexto do processo na entrada correspondente na tabela de controlo de processos
   - estado: `BLOCKED`
   - atualizar o seu $PC [9] para o endereço que produziu a falta de bloco
2. Determinar se existe espaço em memória para carregar o bloco em falta
   - Caso **exista**: selecionar uma região livre
   - Caso **não exista**: selecionar uma região cujo bloco vai ser substituído
     - se tiver sido modificado -> transferi-lo para a `área de swapping`
     - atualizar a entrada da tabela de blocos do processo a que o bloco pertence
       * indicar que o bloco já não se encontra em memória (registo `M/AS`)
3. Transferir o bloco em falta da área de swapping para a região selecionada
4. Invocar o escalonador para calendarizar a execução de um dos processos da fila de espera `READY-TO-RUN`
5. Quando a transferência estiver concluída
   - Atualizar a entrada da tabela de blocos do processo
     - indicar que o processo está residente em memória
     - indicar a sua localização
   - estado: `READY-TO-RUN`
   - colocar o processo na fila de espera `READY-TO-RUN`

[9] ficheiro em código fonte de compilação separada

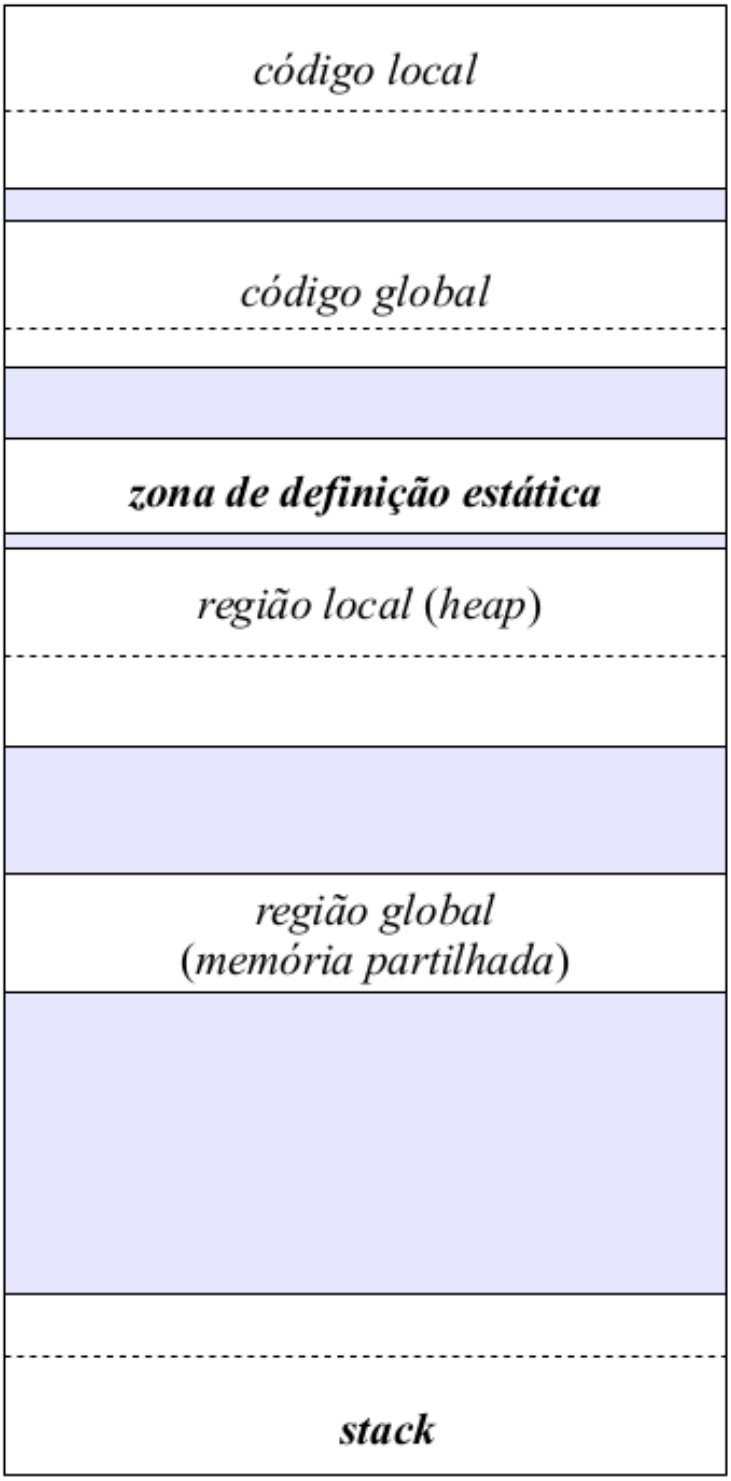


Figure 63: Estrutura de uma organização de memória em arquitectura paginada

Os blocos do espaço de endereçamento do processo passam a ser designados de **páginas**.

- São todos iguais
- Tamanho múltiplo de uma potência de 2
  - Tipicamente 4 ou 8 KB
-

O espaço de endereçamento lógico é endereçado usando:

- **bits mais significativos:** número da página
- **bits menos significativos:** deslocamento

A memória principal é dividida em blocos da **mesma dimensão** que as **páginas**. A estes blocos chamamos **frames**

O `linker` organiza o espaço de endereçamento lógico do processo atribuindo o início de uma nova página a cada uma das regiões funcionalmente distintas:

- código local
- código global
- zona de definição estática
- região local *(heap)*
- região global *(shared memory)*
- `stack` (neste caso, atribuí o fim da página)

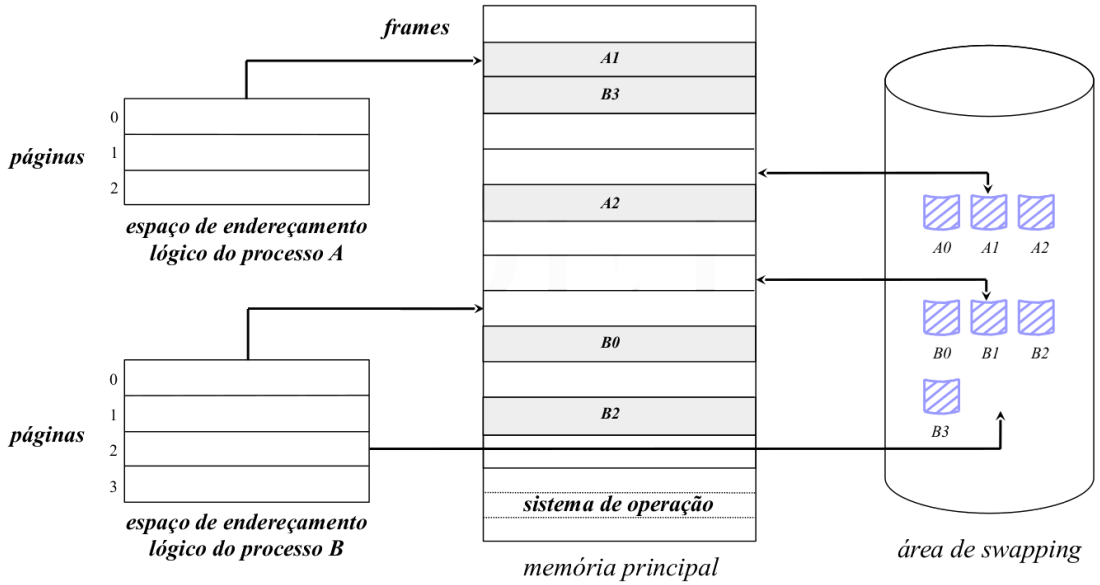


Figure 64: Exemplo da ocupação da memória principal e swap num arquitectura paginada

22.4 Acesso à memória

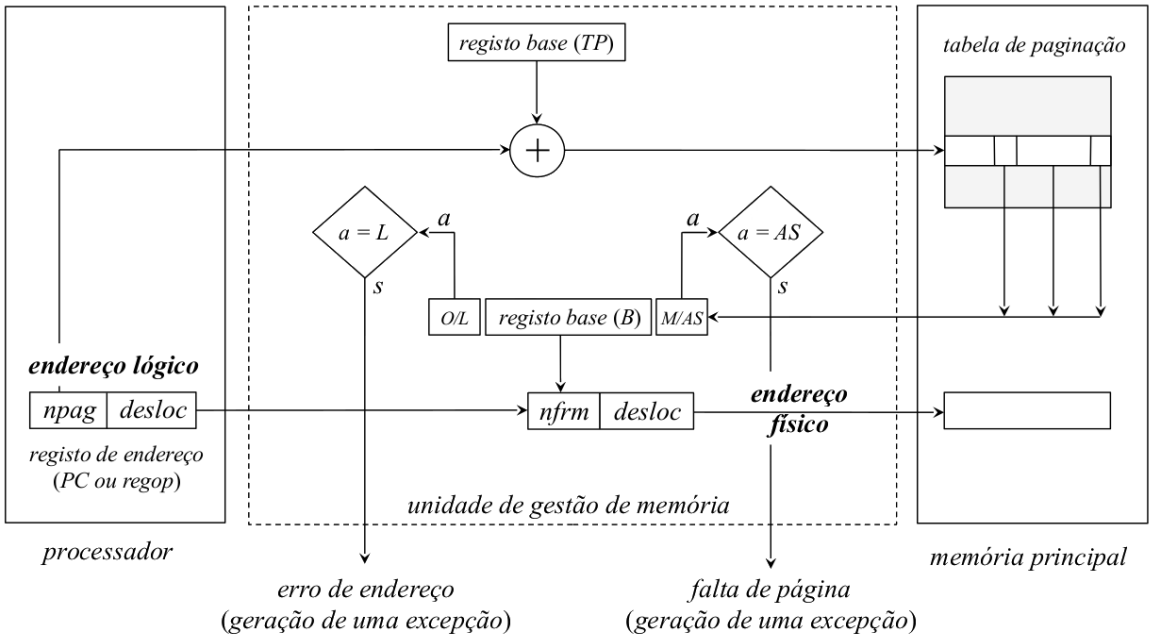


Figure 65: Diagrama de blocos para efetuar o acesso

- Deixa de ser necessário o registo limite na tabela de paginação do processo
  - A Tabela de paginação do processo só precisa do registo base
  - Não é necessário o endereço limite de uma página articular
  - O endereço físico é formado pela concatenação entre os campos:
    * `nfrm` (identifica o frame da memória principal onde a página está localizada)
    * `desloc` (identifica o deslocamento dentro da página/frame
    * endereço físico = nfrm | desloc
  - O endereço lógico é a concatenação de (32 bits):
    * bits MSB: número da página (20 bits)
    * bits LSB: offset dentro da página (12 bits)
  - É possível estruturar o espaço de endereçamento lógico do processo de modo a mapear a totalidade (ou pelo menos uma fração) do espaço de endereçamento do processador
  - Estas frações de espaço podem ser maiores ou iguais ao tamanho da memória principal existente
- Diretamente resultam duas consequências:
  - É possível reservar espaço na zona de definição dinâmica
  - A stack pode atingir a máxima amplitude possível
- As páginas correspondentes à zona de **definição dinâmica** (heap) e à **stack** só são criadas **quando necessário**

  - Permite poupar área de swapping
  - Origina um erro de endereço sempre que se tenta aceder a uma página que ainda não existe
    * Erro de segmento != Falta de página
      · Acesso a um endereço invalido != Acesso a um endereço válido mas que não existe em memória de swap ### Conteúdo da entrada da tabela de paginação

O/L	*M/AS*	*Ref*	*Mod*	*Perm*	*N. do fra*

- **O/L** (Ocupada/Livre): bit que sinaliza a ocupação ou não desta entrada (a não ocupação significa que ainda não foi reservado espaço na área de swapping para esta página)
- **M/AS** (Memória/Área de Swap): bit que sinaliza se a página está ou não `residente em memória principal`
- **Ref** (Referenciada): bit que sinaliza se a página foi ou não `referenciada para leitura e/ou escrita`
- **Mod** (Modificada): bit que sinaliza se a página foi ou não `referenciada para escrita`
- **Perm** (Permissões): indicação do tipo de acesso permitido
  - `ronly` (read-only)
  - `read`/`write`
  - `rwx` (ler/escrever operandos, executar instruções)
- **Número do frame em memória (nfrm):** localização da página, se residente em memória principal
- **Número do bloco na área de swapping:** localização da página na área de swapping, se lhe foi atribuído espaço

22.5 Vantagens e Desvantagens

22.5.1 Vantagens

- **geral:** o âmbito da aplicação é **independente do tipo de processos** que vão ser executados (número e tamanho do espaço de endereçamento)
- **grande aproveitamento da memória principal:**
  - não conduz à fragmentação externa
  - desfragmentação interna desprezável
- **não exige requisitos especiais de hardware:** a unidade de gestão de memória (MMU) existente nos processadores atuais já vem preparada para a sua implementação
  - Gera as exceções, os processadores é que têm de tratar delas
- As páginas só vão sendo atribuídas ao processos à medida das necessidades

22.5.2 Desvantagens

- **acesso à memória mais longo:**

  - cada acesso à memória transforma-se num duplo acesso devido à consulta prévia da tabela de paginação
  - pode ser minimizado usando a TLB, `translation lookaside buffer` para armazenar as entradas da tabela de paginação recentemente mais referenciadas

- **operacionalidade muito exigente:**
  - a sua implementação exige que o SO possua um conjunto de operações de apoio complexas
  - essas operações têm de ser cuidadosamente estabelecidas para que não existam perdas grandes de eficiência
  - As entradas das tabelas de paginação são mais completas

23 Arquitectura Segmentada

- Divide o espaço de endereçamento lógico do processo em segmentos
  - **Divisão cega**.
  - Não leva em consideração qualquer informação sobre a estrutura do programa
  - Apenas efetua a divisão tendo em contas as regiões do código funcionalmente distintas (distinguidas atrás)
  - Não é possível trabalhar com grupos de segmentos
  - Na prática cada segmento é tratado de forma independente

Tem como consequência:

- A estrutura modular que está na base do desenvolvimento de software complexo **não é tida em conta**
  - **Não é possível usar o princípio da localidade da referência** para minimizar o número de páginas que tenham de estar residentes em memória principal em cada etapa de execução do processo
- A **gestão do espaço disponível** entre a zona de **definição dinâmica** e **stack** torna-se difícil e pouco eficiente
  - É agravada no caso de surgirem em run-time múltiplas regiões de dados partilhados de tamanho variável ou estruturas de dados de crescimento contínuo

Uma solução consiste em desdobrar o espaço de endereçamento lógico do processo. Passamos de um espaço de endereçamento linear único (como na arquitectura paginada) para uma multiplicidade de espaços de endereçamento lineares autónomos definidos na fase de linkagem

- Cada módulo [10] da aplicação irá originar dois espaços de endereçamento autónomos:
  1. código
  2. zona de definição estática:
     - variáveis globais á aplicação (definidas localmente)
     - variáveis localmente globais (internas ao módulo)
- Cada um destes espaços de endereçamento autónomo designa-se por `segmento`

[10] ficheiro em código fonte de compilação separada

  - possui uma organização em memória virtual

- Os blocos/segmentos podem ser de comprimento variável

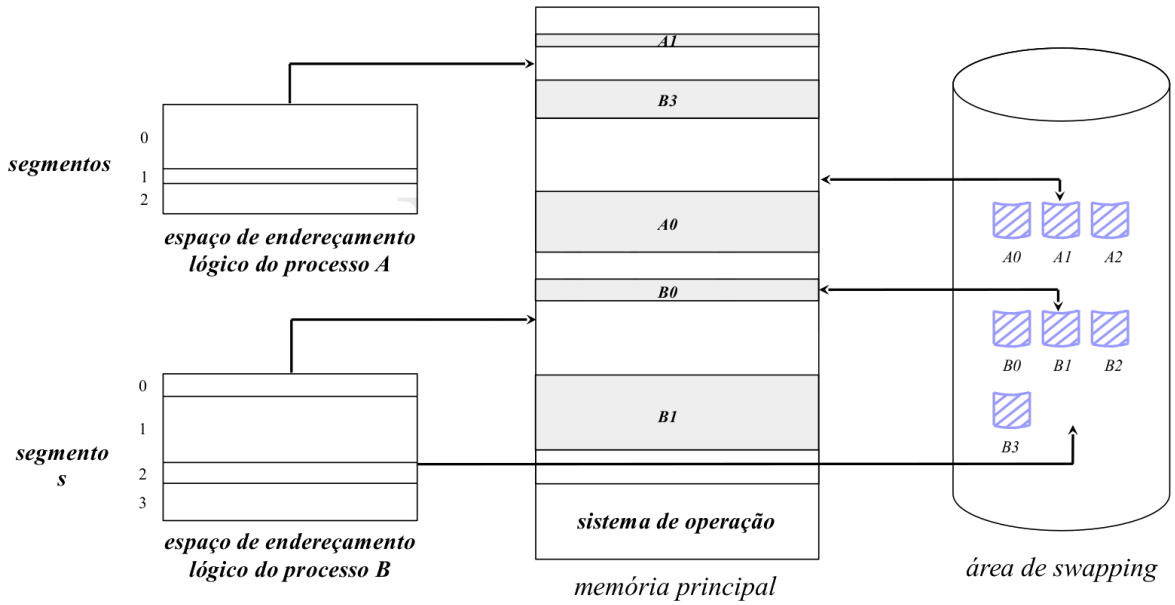


Figure 66: Exemplo de memória segmentada

23.1 Tipos de Segmentos:

- **região de código:** um segmento por cada módulo que contenha código (global ou local)
- **região de definição estática:** um segmento por cada módulo que contenha a definição de variáveis globais à aplicação ou módulo
- **zona de definição dinâmica local (heap):** um segmento
- **zona de definição dinâmica global:** um segmento por cada região de memória de partilha de dados
- **stack:** um segmento

23.2 Tradução de um endereço lógico num endereço físico

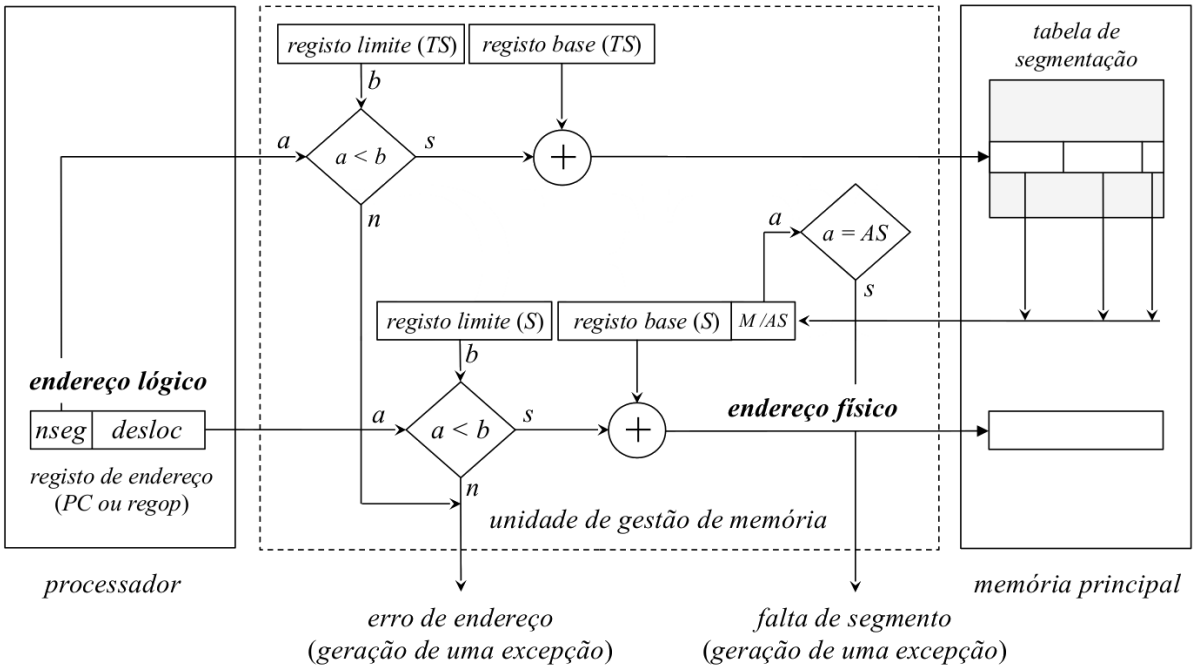


Figure 67: Diagrama de blocos da operação de tradução de um endereço lógico num endereço físico

23.3 Conclusão

A arquitectura segmentada, na sua versão pura, possui pouco interesse prático. Ao tratar a memória principal como um espaço contínuo, exige que sejam aplicadas técnicas de reserva de espaço para carregamento de um segmento de memória.

Estas técnicas assemelham-se ao que acontece numa estrutura de memória real com partições variáveis

Como consequência, existe uma grande **desfragmentação externa da memória principal**, resultando no **desperdício de espaço**

Coloca-se outro problema referente a segmentos de dados de crescimento contínuo:

- pode ser necessário efetuar um acréscimo de espaço ao tamanho do segmento mas este poderá não ser realizado na sua localização presente
  - obriga à transferência total para outra região de endereçamento de memória
  - no caso limite não existe memória disponível para essa expansão
    * o processo é bloqueado/suspenso
    * o seu segmento ou a totalidade do espaço de endereçamento são movidos para a área de swapping

24 Arquitectura Segmentada/Paginada

- Arquitectura mista que combina as características desejáveis das duas arquitecturas anteriores:
  - O **espaço de endereçamento lógico** é dividido em segmentos, com a atribuição na fa se de linkagem de múltiplos espaços de endereçamento autónomos
  - Cada um dos **segmentos** (espaços de endereçamento lineares) é divido em **páginas**
    * Origina um mecanismo de carregamento de blocos em memória principal com todas as características da arquitectura paginada

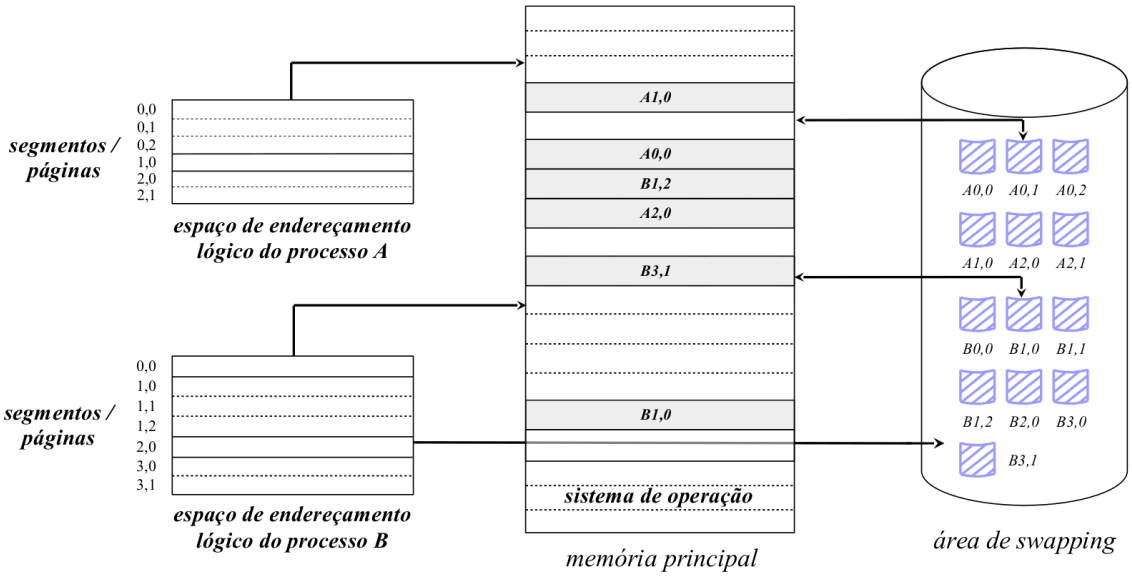


Figure 68: Estrutura de uma arquitectura segmento-paginada

24.1 Tradução de um endereço lógico num endereço físico numa arquitectura segmento-paginada

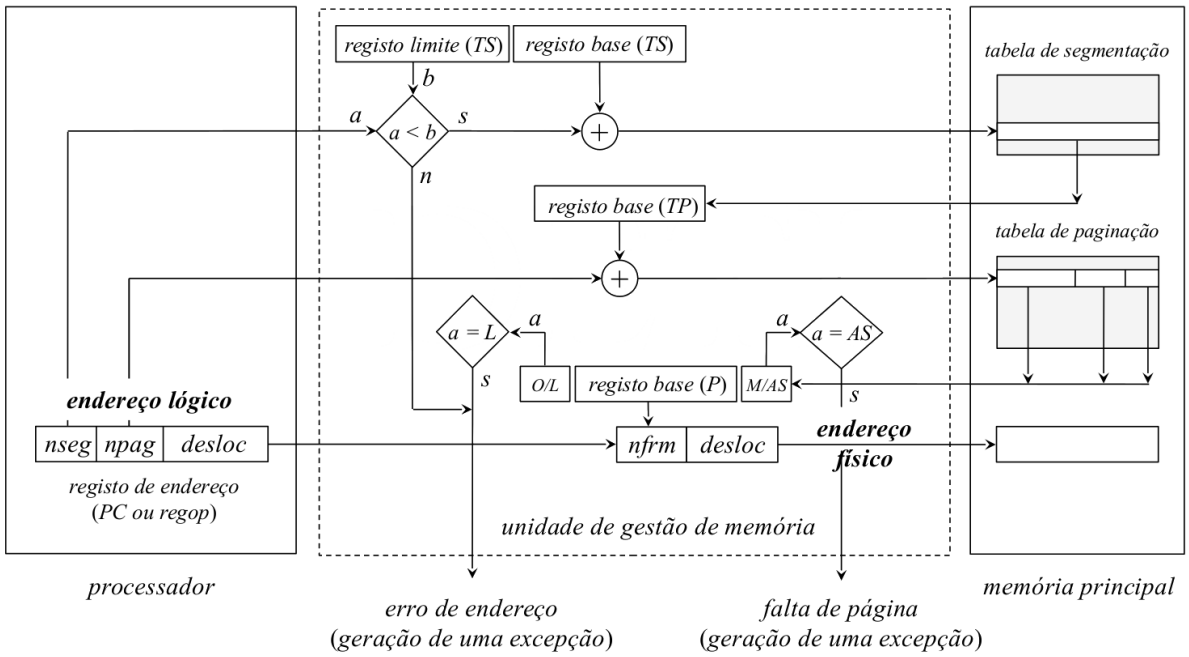


Figure 69: Tradução de um endereço lógico num endereço físico numa arquitectura segmento-paginada

- O endereço lógico passa a possuir 3 campos:
  1. `nseg`: número do segmento
  2. `npah`: identifica a página no segmento
  3. `desloc`: localiza uma posição de memória concreta dentro da página (offset)
- A unidade de gestão de memória contém três registos base e um registo limite associados:
  - **endereço da tabela de segmentação** do processo: registo base (TS)
  - **número de entradas na tabela de segmentação**: registo limite
  - **endereço da tabela de paginação do segmento** que está a ser referenciado: registo base (TP)
  - **frame da memória principal** onde está localizada a página: registo base (P)
- Cada acesso à memória transforma-se em **3 acessos**:
  1. Referencio a **tabela de segmentação do processo** associada com o segmento descrito no campo `nseg` do endereço lógico para obter o **endereço da tabela de paginação do segmento**
  2. Referencio a entrada da **tabela de paginação do segmento** associada com a página descrita no campo `npag` do endereço lógico para obter o `frame` da memória principal onde está localizada a página
  3. Referencio a posição de memória pretendida, concatenando o `nfrm` com o campo `desloc`

Perm	*Endereço em memória da tabela de paginação do segmento*

Figure 70: Conteúdo de cada entrada da tabela de segmentação

O/L	*M/AS*	*Ref*	*Mod*	*N. do frame em memória*	*N. do bloco na área de swapping*

Figure 71: Conteúdo de cada entrada da tabela de paginação de cada segmento

O campo `Perm` é deslocado para a entrada que descreve o segmento. O aceso pode ser tratado de maneira global, sendo as permissões aplicadas ao segmento.

Do ponto de vista do processo passam a ser precisas várias tabelas de paginação e várias tabelas de segmentação, sendo que cada tabela de segmentação pode ter mais que uma tabela de paginação.

24.2 Vantagens vs Desvantagens

24.2.1 Vantagens

- **geral:** pode ser aplicado independentemente do tipo de processos que vão ser executados. quer em número como em tamanho do seu espaço de endereçamento
- **grande aproveitamento da memória principal**:
  - não conduz à fragmentação externa da memória
  - fragmentação interna +e desprezável
- **gestão mais eficiente da memória no que respeita a regiões de crescimento dinâmico**
- minimização do número de páginas que têm de estar residentes em memória principal em cada execução do processo

24.2.2 Desvantagens

- **Exige requisitos especiais de hardware**
  - Nem todos os processadores atuais de uso geral estão preparados para a sua implementação
- **Acesso à memória mais longo**
  - Cada acesso à memória é um **triplo acesso**
  - Pode ser minimizado se a unidade de gestão de memória contiver um **TLB** `translation lookaside buffer`

        * Seria usado para armazenamento das entradas da tabela de paginação recentemente referenciadas no segmento

- **Operacionalmente muito exigente**
  - A sua implementação por parte do SO é mais exigente do que a arquitectura paginada

25 Políticas de Substituição de páginas em memória

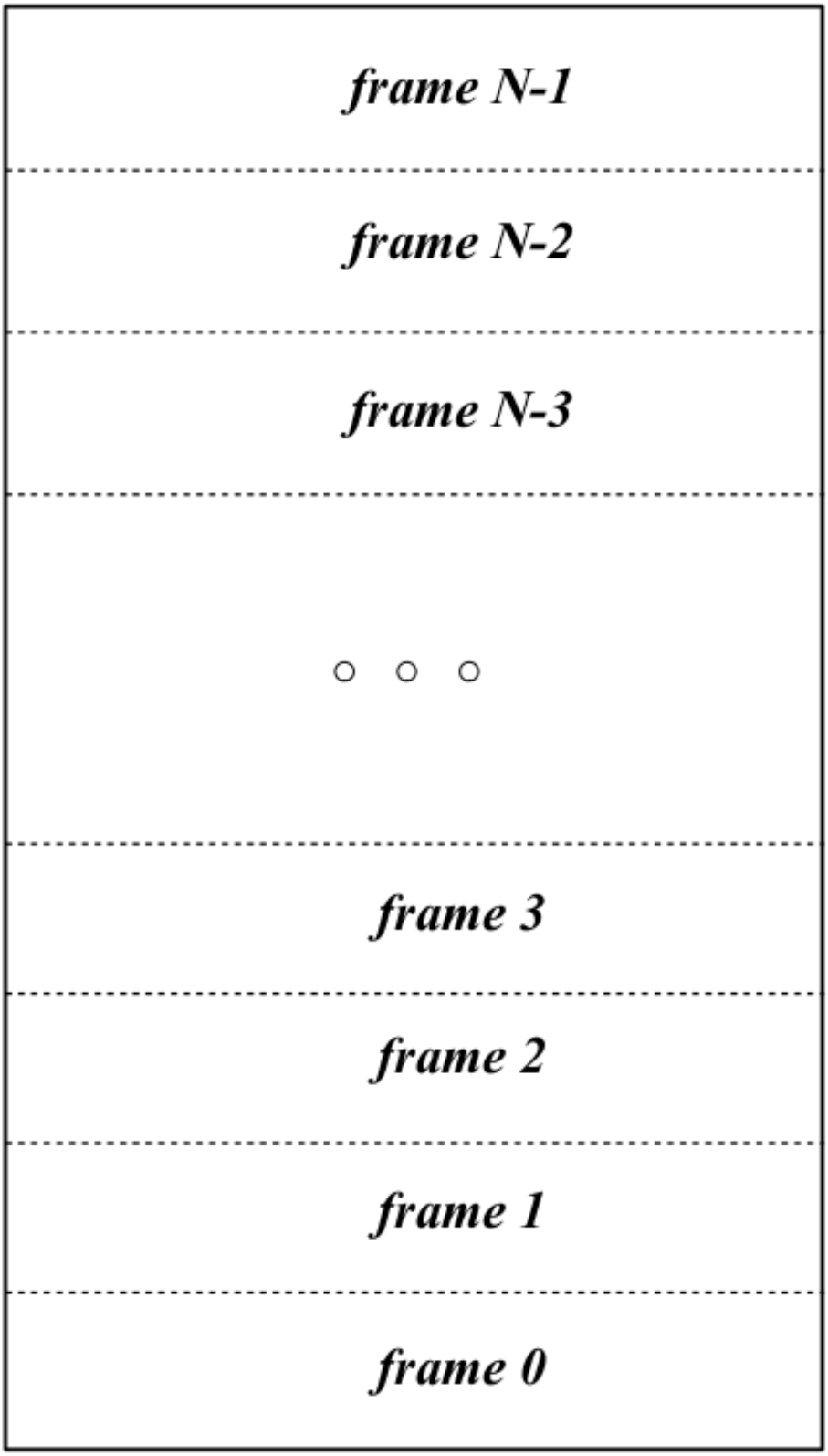


Figure 72: Divisão da memória em frames

Numa arquitectura paginada ou segmentada/paginada a memória principal é vista como dividida operacionalmente me frames do tamanho de cada página

- Cada `frame` vai permitir o armazenamento do conteúdo de uma página do espaço de endereçamento lógico de um processo
- As páginas podem estar em dois estados diferentes:
  - `locked`: **Não podems er removidas de memória**
    * páginas associadas com o *kernel* do SO
    * *buffer cache* do sistema de ficheiros
    * *memory mapped file*
    * *memory mapped variables*
    * *memory mapped IO*
  - `unlocked`: **podem ser removidas de memória**
    * páginas associadas aos processos convencionais

Os `frames` estão associados em listas biligadas:

- se ocupados e associados a páginas `unlocked` $\implies$ `frames` passíveis de substituição.
- `frames` livres

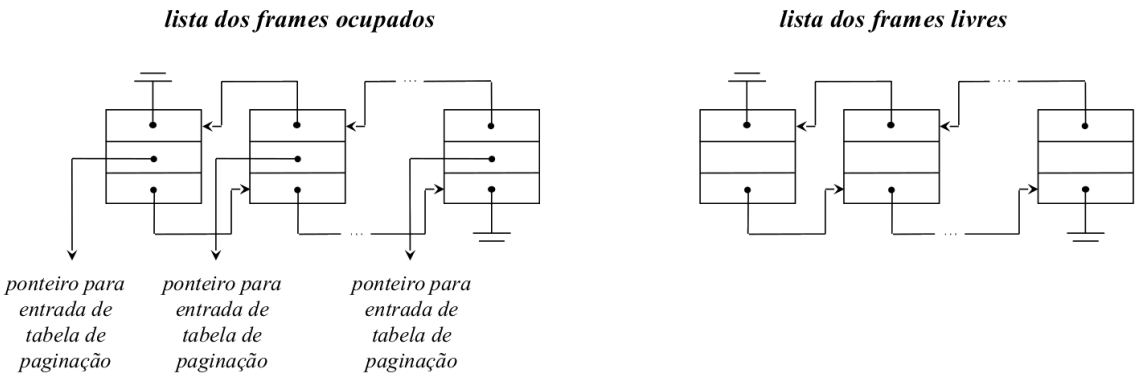


Figure 73: Exemplos do estado das listas biligadas

O tipo de memória implementado pela **lista dos frames** ocupados depende do **algoritmo de substituição utilizado**

Quando ocorre uma **falta de página** (programa tenta aceder a uma dada página que não está em memória):

- a situação mais provável é a lista dos `frames livres` estar vazia
  - torna-se necessário selecionar um `frame` para substituição da lista dos `frames ocupados`
  - alternativamente, pode manter-se sempre na `lista dos frames livres` alguns `frames`
    * usa-se um deles para carregar a página em falta
    * procede-se de seguida **substituição de um `frame ocupado`**
    * é o **método mais eficiente** (as operações decorrem em paralelo)

É necessário ir mudando dinamixamente as páginas dos vários processos que vão existindo em memória. Se assumirmos ainda que os processos em execução ocupam toda a memória disponível, se um dos processos que está em execução precisar de aceder a um bloco que ainda não está em memória, como faço?

O problema que se coloca é: **Que frame escolher para a substituição?**. Em teoria deve ser um `frame` que:

- não irá ser mais referenciado
- ou a sê-lo, sê-lo-á o mais tarde possível

A condição anterior enuncia o **Princípio da Otimalidade**. Ao aplicar estes critérios na escolha da página **Minimiza-se** a ocorrência de outras **faltas de página**

PROBLEMA: o princípio da otimalidade é **não-causal**. Não pode ser diretamente implementado.

Objetivo: Encontrar estratégias de substituição que sejam realizáveis e que ao mesmo tempo, se aproximem tanto quanto possível do princípio da otimalidade

25.1 Algoritmo LRU - Least Recently Used

- Visa encontrar o `frame` que não é referenciado à mais tempo
- Assumo que cada processo vai usar às paginas que usou à menos tempo
- Partindo do **princípio da localidade de referência**, se um `frame` não é referenciado há muito tempo, é fortemente provável que não venha a ser referenciado num futuro próximo
- Cada referência à memória precisa de ser sinalizada com o instante da sua ocorrência (conteúdo de um `timer` ou um contador)
  - Preciso de ordenar cronologicamente quantas páginas possuo em memória
  - Como é pouco provável que a unidade de gestão de memória possua a capacidade de o fazer, será necessário *hardware* especializado
  - Ou então tenho de ir à memória ler a lista em cada pedido de acesso à memória
- Sempre que ocorre uma **falta de página**, a **lista biligada** dos `frames` ocupados tem de ser percorrida para determinar qual o `frame` que foi acedido à mais tempo
  - HEAD: página acedida à menos tempo
    - tem de ser atualizada a cada acesso à memória
  - TAIL: página acedida à mais tempo

Possui um **custo de implementação elevado e pouco eficiente**

25.1.1 Algoritmo NRU - Not Recently used

- Aproximação menos exigente e relativamente eficiente do LRU.
- Usa os bits `Ref` e `Mod` que são processados tipicamente por uma unidade de gestão de memória convencional:
  - Sempre que uma página é acedida para leitura, o campo Ref é colocado a 1

  - Sempre que uma página é acedida para escrita, o campo Ref e Mod são colocados a 1

- Periodicamente o SO percorre a `lista dos frames ocupados` e coloca a **zero o bit `Ref`**
- Quando ocorre uma falta de página os `frames ocupados` enquadram-se numa das classes seguintes

Classes	Ref	Mod
classe 0	0	0
classe 1	0	1
classe 2	1	0
classe 3	1	1

A seleção da página a substituir será feita entre aquelas pertencentes à classe de ordem mais baixa existente atualmente na lista dos frames ocupados

25.2 Algoritmo FIFO - First In, First Out

- Critério baseado no tempo de estadia das páginas em memória principal
- Baseia-se no pressuposto que **quanto mais tempo as páginas residirem em memória, menos provável será que elas sejam referenciadas a seguir**
- A `lista dos frames` ocupados está organizada num FIFO que espelha a **ordem de carregamento** das páginas correspondentes em memória principal

Quando ocorre uma **falta de página:**

- retira-se do FIFO o elemento correspondente à **página há mais tempo em memória**

Algoritmo extremamente falível! Por exemplo:

- Páginas associadas com o código de um editor de texto
- compilador
- bibliotecas do sistema

25.3 Algoritmo da Segunda Oportunidade

- A lista dos `frames ocupados` está organizada num FIFO que espelha a ordem de carregamento das páginas correspondentes em memória principal
- Quando ocorre uma **falta de página:**
  - retira-se do FIFO o **elemento correspondente à página há mais tempo em memória**
  - se o seu bit `Ref` estiver a **zero** $\implies$ a página é escolhida para **substituição**
  - caso contrário, coloca-se o seu bit `Ref` a **zero**
    * o nó é reintroduzido no fim da FIFO
    * o processo repete-se

25.4 Algoritmo do relógio

- Segue a estratégia subjacente ao algoritmo da segunda oportunidade
- Torna a mais eficiente implementando a FIFO numa lista circular
- As operações `fifoIn` e `fifoOut` correspondem a incrementos de um ponteiro

Quando ocorre uma **falta de página:**

- Enquanto o bit `Ref` do `frame` pelo ponteiro **não for zero**:
  - O bit `Ref` é colocado a zero
  - O ponteiro avança uma posição
- A página apontada é escolhida para substituição
- O ponteiro avança uma posição

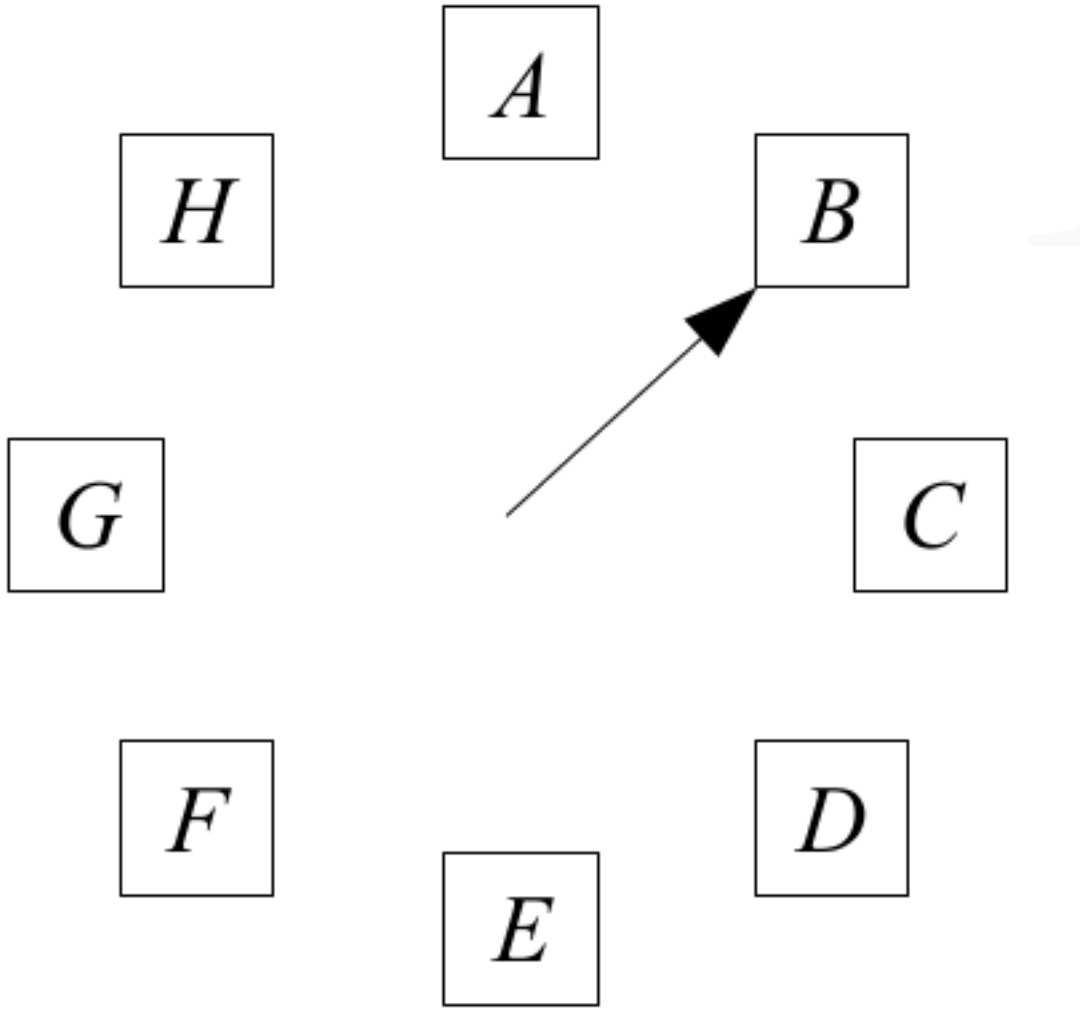


Figure 74: Algoritmo do Relógio

26 Working set

- Quando um processo é colocado pela primeira vez na fila de espera dos processos `READY-TO-RUN`, só a 1ª e última página do seu espaço de endereçamento (início do código e *stack*, respetivamente) é que são carregadas em memória
- Quando o processador for atribuído ao processo, suceder-se-ão inicialmente várias **faltas de página** a um ritmo rápido, porque não possui as páginas necessárias à sua execução em memória principal
- De seguida o número de flatas de página diminui e o processo entra numa fase da execução sem faltas de página
- Dado o princípio da localidade da referência, um processo vai aceder às mesmas variáveis e instruções.

- Assim, todas as páginas associadas à fração do espaço de endereçamento que o processo está atualmente a referenciar já estão todas presentes em memória principal

working set: conjunto de páginas é designado o working set do processo

Ao longo do tempo o `working set` do processo vai variar, não só no que respeita ao número, mas também às páginas concretas que o definem

Se o working set não consegue estar todo em memória vão ocorrer muitas **faltas de página** e o ritmo de execução será muito lento. Ocorre `trashning`:

- `frames` do working set a passarem para da memória para a swap
- `frames` do `working set` na swap a serem passados para a memória

Se não correr `trashing`, o processo alterna entre períodos curtos que sofrerá muitas **faltas de páginas** e períodos longos quase sem **faltas de página**

O **objetivo prioritário** de qualquer política de substituição é garantir que mantem sempre o `working set` do processo em memória principal

Uma estratégia consiste em atribuir novos `frames`ao process sempre que este se encontre num período elevado de **faltas de página** e retirar-lhe `frames` quando as ocorrência de faltas de página baixar.

27 Demand paging vs prepaging

Quando um processo é introduzido na fila de espera dos processos `READY-TO-RUN` pela 1ª vez ou em resultado de uma suspensão, é preciso decidir que páginas colocar em memória principal.

- `Demand paging`: Estratégia minimalista e menos eficiente
  - nenhuma página é colocada
  - o mecanismos de geração de **faltas de página** é que é responsável por formar o `working set` do processo
- `prepaging`: estratégia mais eficiente
  - procura-se adivinhar o `working set` do processo para minimizar a geração de faltas de página
  - na 1ª vez são colocadas as primeiras duas páginas atrás referidas
  - nas vezes seguintes, são colocadas o conjunto de páginas que residiam em memória no **momento em que o processo foi suspenso**

27.1 Substituição global vs substituição local

Qual o âmbito da aplicação dos algoritmos de substituição?

- **local**: a escolha é efetuada entre o conjunto de `frames` de um processo
- **global**: a escolha é efetuada entre o conjunto de todos os `frames` que continuem a lista de `frames` ocupados

É preferível o âmbito de aplicação global:

- Penso em cada processo globalmente
  - É mais fácil gerir `working sets` que mudam de dimensão
  - Permite-me suportar grandes variações de `working set` dos processos sem resultar desperdício de memória ou `trashing`
  - Desde que os processos não for em demasia, é possível minimizar o `thrashing`
    * Se a soma dos `working sets` de todos os processos é superior ao número de `frames unlocked` disponíveis em memória principal, entro em `thrashing`
    * A solução passa por ir suspendendo processos até que o `trashing` desapareça