universidade de aveiro  deti departamento de electrónica, telecomunicações e informática

Sistemas Operativos

Teóricas

Resumos

Gonçalo Matos | MEC 92972

Licenciatura em Engenharia Informática

1.º Semestre | 2.º Ano | Ano letivo 2019/2020

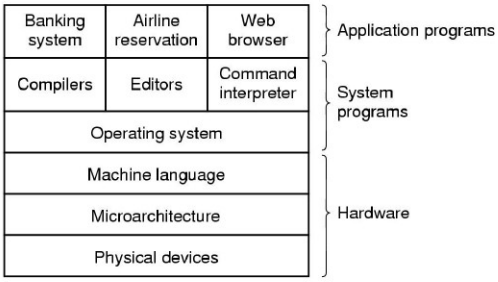
Última atualização a 15 de janeiro de 2020

Índice

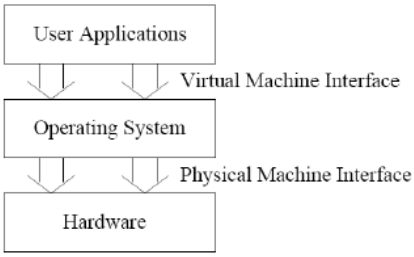
1. Conceitos introdutórios

Operating System Concepts, página 3

Um sistema operativo é um programa base que estabelece a interface entre os programas de aplicação e o hardware. É o único programa que permanece em execução todo o tempo em que um computador está ligado, designando-se por isso de **kernel**.



Tem como **objetivos** executar os programas de aplicação, tornar o hardware mais fácil de usar (criando um nível de abstração que esconde a utilização de dispositivos específicos) e usá-lo de uma forma eficiente (gestão de forma a utilização ser justa e segura). Por vezes torna-se difícil combinar os dois últimos.



Fornece serviços standart que são implementados pelo hardware (sistemas de ficheiros, redes); coordenação de várias aplicações e utilizadores garantido a sua segurança, eficiência e justiça na utilização dos recursos e controlo da execução dos programas, prevenindo erros.

Tem assim os **papéis** de árbitro, gerindo os recursos partilhados (memória, dispositivos externos); ilusionista, fornecendo às aplicações a abstração de recursos com capacidades superiores às existentes (memória infinita, uso exclusivo da CPU) e adaptador, separando as aplicações dos dispositivos de entrada/saída.

Existem vários tipos de sistemas operativos, cada um para um determinado dispositivo com funcionalidades específicas, por exemplo uma mainframe[1], um dispotivo móvel, um servidor, um *smart card*, etc.

Funcionalidades

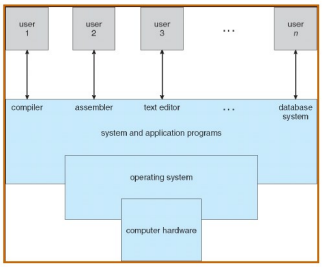
Um sistema operativo tem muitas funcionalidades, das quais se destacam:

- Estabelecimento do ambiente de base de interação com o utilizador, permitindo vários utilizadores em simultâneo;
- Mecanismos de execução controlada de programas, permitindo concorrência entre programas (execução em simultâneo);
- Mecanismos de comunicação entre programas e respetiva sincronização;
- Disponibilização de facilidades para o desenvolvimento, teste e depuração de programas, designadas por **middleware**, que suportam bancos de dados, ambientes multimédia, elementos gráficos, entre outros...
- Espaço de endereçamento dos programas independente das limitações da memória física e gestão das alocações de memória e transferências de dados entre memória e disco;
- Organização do espaço em disco num sistema de ficheiros, permitindo armazenar vários ficheiros de tamanho variável;
- Modelo de acesso a dispositivos de I/O (CPU continua a trabalhar enquanto dispositivo não responde);
- Deteção de situações de erro;
- Sistemas distribuidos, permitindo que um grupo de computadores trabalhe em conjunto para resolver um problema.

1 Tem como único objetivo otimizar a utilização do hardware

2. Sistemas de computação

Operating System Concepts, página 3



Um sistema de computação consiste num sistema dividido em quatro componentes: **hardware** (CPU), **sistema operativo**, **programas de aplicação** e **utilizadores**.

Dentro do sistema operativo, há ainda os **programas do sistema** que estão associados a este, mas não fazem necessariamente parte do kernel.

Inicialização

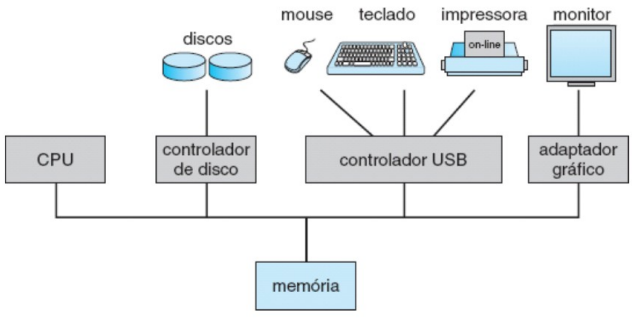
Operating System Concepts, página 6

Quando é **ligado**, executa um programa inicial, o **programa bootstrap**. Este é um programa de complexidade reduzida, normalmente armazenado na memória ROM ou EPROM[2] (firmware), responsável por iniciar vários dispositivos do sistema, localizando e carregando ainda na memória o núcleo (kernel) do sistema operativo e começando a sua execução.

Organização

Operating System Concepts, página 6

Um **sistema de computação** é composto por um ou mais CPU e vários controladores de dispositivos (um para cada tipo de dispositivo), que trabalham em paralelo, conectados por um bus comum que dá acesso à memória partilhada.

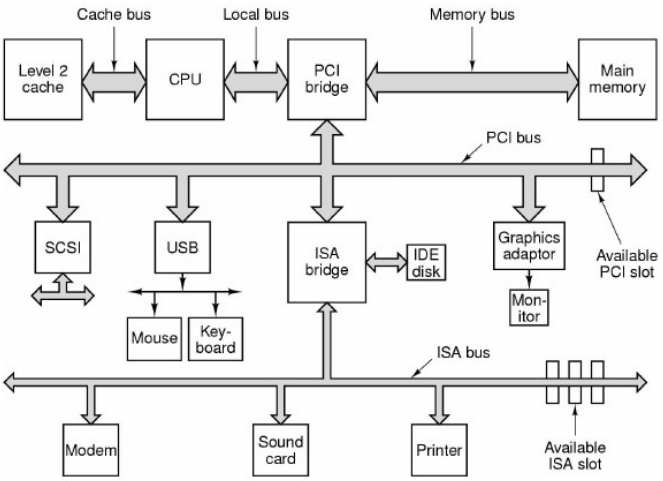


Os controladores de dispositivos têm um **buffer local**[3], para armazenarem os dados temporários quando não os conseguem transmitir de imediato.

2 Electrically Erasable Programmable Read-Only Memory

3 Uma região de memória temporária utilizada para escrita e leitura de dados

Os dados de I/O são assim transferidos do dispositivo para o buffer local, de onde posteriormente a CPU os vai mover para a memória. Quando termina a operação, o controlador de dispositivo informa a CPU através do envio de uma interrupção ±.



Organização do computador

Estrutura do armazenamento

Operating System Concepts, página 8

Para ser executado, um programa é carregado na **memória principal**, a RAM. Esta é a única a que o processador consegue aceder diretamente. Os acessos podem ser feitos para colocar informação nos registos (carregamento de *words*), ou para executar instruções diretamente.

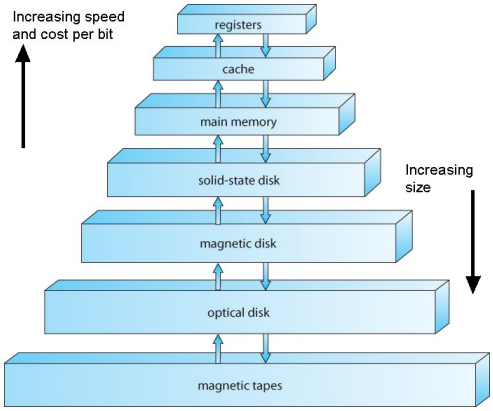
Uma execução normal num sistema de von Neuman consiste no carregamento das instruções para os registos de instruções, onde são depois descodificadas.

Apesar do seu acesso ser mais rápido, infelizmente e como sempre em informática, esta grande qualidade traz dois grandes defeitos, a memória é reduzida e volátil - quando desconectada da sua fonte de alimentação apagada.

Por isto existe sempre associada a ela uma **memória secundária**, de um tipo que tem maior capacidade de armazenamento permantente e apesar do acesso ser mais lento é mais económico.

É lá que os programas estão armazenados quando não estão em execução e durante a mesma há sempre comunicação entre estas e a memória secundária (para armazenar dados importantes). Normalmente esta memória é um disco magnético.

Existem para além destes vários tipos de memória, que são organizados numa hierarquia de acordo com a velocidade e custo por bit, que é inversamente proporcional à quantidade de informação que conseguem armazenar.



Os três tipos de memória no topo da hierarquia são voláteis.

Multiprogramação

Operating System Concepts, página 15

É comum um SO ter vários programas em execução. No entanto, na maioria das vezes, nem todos estão carregados em memória (esta não tem capacidade para tantos programas em simultâneo), sendo estes guardados na reserva de trabalhos (*job pool*) – na memória secundária.

Para que todos os programas possam ser executados, foi criada a multiprogramação. Assim, o SO mantém os programas na reserva de trabalhos, executando o máximo de trabalhos que consegue em simultâneo. No entanto, muitas vezes os programas pausam a sua execução à espera de alguma interação ou da resposta de um dispositivo e I/O.

A solução da multiprogramação consiste em, nestes momentos, começar a executar outro programa, utilizando assim os recursos do CPU que de outra forma estaria desocupado. Quando o primeiro estiver pronto para ser executado novamente, toma conta do CPU novamente.

A eficiência do CPU é assim exponenciada, sendo utilizados quase em permanência todos os recursos dos sistemas de computação.

Timesharing

Associada à multiprogramação está a partilha de tempo, fornecendo interação entre o sistema e o utilizador. A ideia base é a de partilha dos recursos (por vários utilizadores eventualmente), sendo a utilização do CPU alternada rapidamente entre os vários processos (programas em execução).

Este conceito baseia-se na premissa de que o CPU é muito mais rápido do que um dispositivo de I/O, cuja velocidade normal corresponde à velocidade de interação de um humano com uma máquina, que consegue processar dados a uma velocidade muito maior.

Assim, enquanto espera pela introdução de novos dados num programa, o CPU começa a trabalhar noutro processo. Como estas alterações no CPU são tão rápidas, os vários utilizadores não se apercebem de nenhuma perca de desempenho.

É importante destacar que mesmo que não tenha de esperar o CPU alterna a execução dos programas, sendo atribuído a cada um um tempo máximo de ocupação consecutiva.

O principal benefício é a redução da resposta de aplicações interativas.

A ter em conta

Esta utilização intensiva do CPU levanta algumas preocupações, nomeadamente no que toca ao agendamento da execução das aplicações que estão em espera no disco, o **job scheduling**, a escolha entre a execução de dois ou mais trabalhos que estejam prontos em simultâneo, o **CPU scheduling** e por fim a **partilha dos recursos** como os registos, a memória, etc.

Componente prática sobre este tema desenvolvida no final deste documento

Gestão de processos

Operating System Concepts, página 20

Um programa é uma entidade passiva. A sua execução é ativa, designa-se por **processo** e necessita de recursos que lhe são alocados no início ou durante a sua execução e eventuais dados de inicialização. No final, estes devem ser libertados.

Um processo pode correr em 1 *thread* ou em várias (*multi-thread*).

Uma **thread** é uma fila de execução de um mecanismo que pretende executar instruções Assembly. Existe um espaço de endereçamento comum a todo o processo, embora os registos e a pilha (stack) sejam individuais para cada uma. Há ainda um PC[4] para cada.

Tipicamente o sistema tem vários processos, inicializados pelo SO ou pelo utilizador, que correm em simultâneo em 1 ou mais CPU, sendo a distribuição de recursos feita através da multiplexagem do tempo dos CPU entre as várias *threads*.

É resposabilidade do SO fazer o planeamento de processos na CPU, criar, suspender, retomar e excluir processos e fornecer mecanismos de sincronização e comunicação de processos.

Componente prática sobre este tema desenvolvida no final deste documento

4 Indica qual a próxima instrução a executar

Gestão da memória

Operating System Concepts, página 21

A memória física é um recurso escasso e dispendioso.

A memória principal é o único dispositivo de armazenamento a que a CPU consegue aceder diretamente, pelo que as instruções a executar por esta unidade devem estar na memória. Esta é ainda utilizada como repositório de dados compartilhados pela CPU e pelos dispositivos de I/O.

A gestão da memória deve ter em conta a maximização da utilização da CPU, uma utilização transparente da memória, a segurança na utilização da memória e a partilha da mesma pelos vários processos.

Para fazer esta gestão, devem ser conhecidas as zonas de memória livres e a sua atribuição aos diferentes processos, decidindo que processos e dados vão ser movidos de e para a memória.

Gestão do armazenamento

Operating System Concepts, página 22

O SO disponibiliza uma vista lógica e uniforme do espaço de armazenamento, oferecendo uma abstração das propriedades físicas da unidade de armazenamento, o **ficheiro**[5], que mapeia para meios físicos e lhe acede através de controladores específicos para diferentes unidades (que têm diferentes capacidades, velocidades, etc.).

Os **sistemas de ficheiros** organizam os ficheiros em pastas, atribuindo-lhes permissões que garantem a segurança dos dados e permitem ao SO criar e apagar[6] ficheiros e pastas, gerir diferentes sistemas de ficheiros de forma integrada, *backups...*

Sistemas de I/O

Operating System Concepts, página 26

Um dos objetivos do SO é esconder os pormenores do dispositivos de hardware do utilizador, tendo como competências a gestão da memória de I/O, a interface geral dos *device drivers* e a disponibilização de *drivers* específicos para diferentes tipos de hardware.

5 Conjunto de dados

6 Permissões têm a ver com o diretório e não com o ficheiro em si!

Proteção e segurança

Operating System Concepts, página 26

Tendo vários usurários e processos em execução, é fundamental regular o acesso aos dados, criando mecanismos que assegurem que determinados ficheiros, segmentos da memória, CPU e outros recursos são executados apenas por quem tem autorização para tal.

Por exemplo o hardware de endereçamento garante que nenhum processo é executado fora do seu espaço de endereçamento; o timer assegura que nenhum processo utiliza o CPU sem abandonar o seu controlo periodicamente.

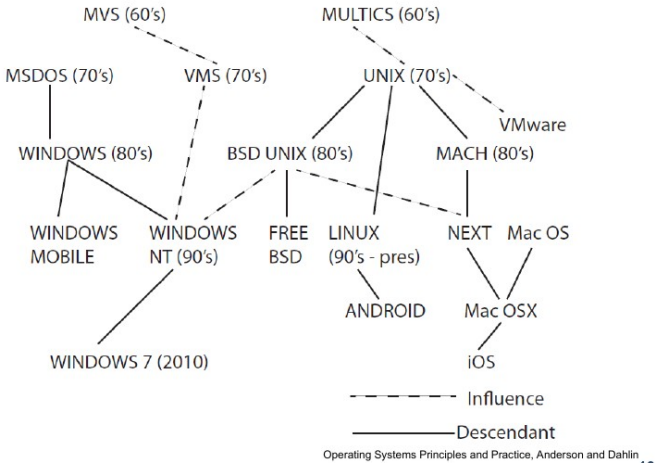
A **proteção** é então fundamental para controlar o acesso de processos e utilizadores aos recursos.

A **segurança** consiste na defesa do sistema contra ataques internos e externos.

Estes dois conceitos exigem que um SO seja capaz de identificar todos os seus utilizadores, associando a ficheiros e processos quem os controla. Para facilitar esta gestão, é ainda possível criar grupos de utilizadores, permitindo assim atribuições de permissão mais genéricas.

3. Sistemas Operativos

A maioria dos SO atuais oferecem um ambiente gráfico, e caracterizam-se por ser **multiutilizador** e **multitarefa**. Têm uma memória virtual e fazem o acesso indistinto a ficheiros/dispositivos locais ou em rede[7]. Oferecem ainda uma vasta gama de *device drivers* que permitem uma dinâmica de dispositivos *plug & play*.



Interpretador de comandos (CLI)

O *Command Line Interpreter* pode estar incluído no kernel ou funcionar como um programa de sistema, tendo como principal função ler e executar comandos do utilizador.

Nos comandos para gerir ficheiros, pode ser utilizado código integrado no interpretador ou programas independentes deste.

Interface gráfica (GUI)

A *Graphical User Interface* é um sistema baseado em janelas e menus preparado para ser manipulado através do rato.

Windows é GUI mas tem CLI como shell de comandos
Mac OS X é GUI mas disponibiliza várias shell
UNIX é em geral baseado em CLI, mas com várias GUI disponíveis (KNE, Gnome...)

7 Aplicações de login remoto, correio eletrónico, navegação na internet...

Modos de operação

De modo a garantir a segurança do sistema, a maioria dos SO podem ser executados em dois modos:

Utilizador Tem restrições de segurança

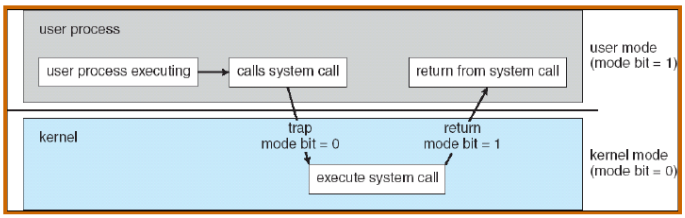
Acesso interdito a certas zonas de memória e dispositivos

Kernel Sem restrições de segurança

Pode executar todas as instruções e acessos

Instruções priveligiadas

As chamadas ao sistema permitem uma forma segura de alternar entre os dois modos.



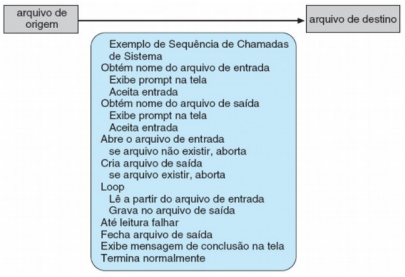
Componente prática sobre este tema desenvolvida no final deste documento

Chamadas ao sistema

Operating System Concepts, página 43

As chamadas ao sistema fornecem uma interface para acesso aos serviços do SO e tipicamente são escritas numa linguagem de alto nível.

Tipicamente, para fazer chamadas ao sistema os programas utilizam APIs em vez de utilização direta. Exemplos são a Win32API (Windows), POSIXAPI (UNIX, MacOS X) e Java API (JVM). Esta abstração permite a simplificação de um programa que vai fazer milhares de chamadas de sistema por segundo.



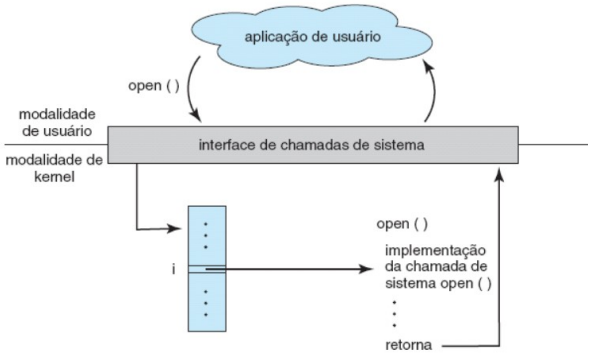
Mas porquê utilizar uma API em vez de fazer chamadas diretas ao sistema? Há várias razões. Uma delas é a portabilidade do programa. Todos os sistemas que tenham suporte para a API em que o programa é desenvolvido devem ser capazes de o compilar e executar. Outra é o nível de detalhe e a dificuldade de fazer chamadas de sistema em comparação com a API. Há no entanto, uma semelhança entre as duas.

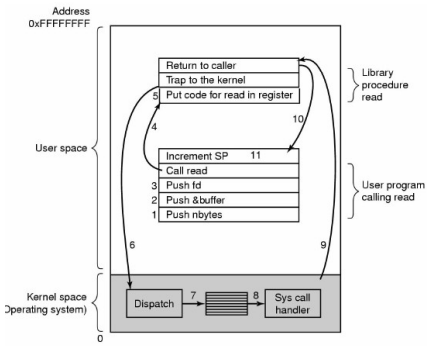
Na maioria das linguagens de programação, o sistema de suporte ao tempo de execução fornece uma **interface de chamadas de sistema**, que serve como uma ponte para as chamadas de sistema disponiblizadas pelo SO. Esta interface interseta as chamadas de função da API e invoca as chamadas de sistema necessárias dentro do SO.

Neste processo, normalmente, é associado um número a cada chamada de sistema, sendo mantida uma tabela indexada com esses números na interface. A interface invica então a chamada de sistema desejada no kernel do SO e retorna o *status* da chamada e eventuais valores de retorno.

O invocador da chamada de sistema fica assim liberto de saber como fazer a chamada, ou como esta é implementada, focando-se em seguir as diretrizes da API e em saber qual o resultado da execução dessa chamada no sistema.

Assim, a maioria dos detalhes da interface do SO é ocultada ao programador pela API e gerida pela biblioteca de suporte ao tempo de execução.





UNIX	Win32	Description
fork	CreateProcess	Create a new process
waitpid	WaitForSingleObject	Can wait for a process to exit
execve	(none)	CreateProcess = fork + execve
exit	ExitProcess	Terminate execution
open	CreateFile	Create a file or open an existing file
close	CloseHandle	Close a file
read	ReadFile	Read data from a file
write	WriteFile	Write data to a file
lseek	SetFilePointer	Move the file pointer
stat	GetFileAttributesEx	Get various file attributes
mkdir	CreateDirectory	Create a new directory
rmdir	RemoveDirectory	Remove an empty directory
link	(none)	Win32 does not support links
unlink	DeleteFile	Destroy an existing file
mount	(none)	Win32 does not support mount
umount	(none)	Win32 does not support mount
chdir	SetCurrentDirectory	Change the current working directory
chmod	(none)	Win32 does not support security (although NT does)
kill	(none)	Win32 does not support signals
time	GetLocalTime	Get the current time

Miscellaneous

Call	Description
s = chdir(dirname)	Change the working directory
s = chmod(name, mode)	Change a file's protection bits
s = kill(pid, signal)	Send a signal to a process
seconds = time(&seconds)	Get the elapsed time since Jan. 1, 1970

Process management

Call	Description
pid = fork( )	Create a child process identical to the parent
pid = waitpid(pid, &statloc, options)	Wait for a child to terminate
s = execve(name, argv, environp)	Replace a process' core image
exit(status)	Terminate process execution and return status

File management

Call	Description
fd = open(file, how, ...)	Open a file for reading, writing or both
s = close(fd)	Close an open file
n = read(fd, buffer, nbytes)	Read data from a file into a buffer
n = write(fd, buffer, nbytes)	Write data from a buffer into a file
position = lseek(fd, offset, whence)	Move the file pointer
s = stat(name, &buf)	Get a file's status information

Directory and file system management

Call	Description
s = mkdir(name, mode)	Create a new directory
s = rmdir(name)	Remove an empty directory
s = link(name1, name2)	Create a new entry, name2, pointing to name1
s = unlink(name)	Remove a directory entry
s = mount(special, name, flag)	Mount a file system
s = umount(special)	Unmount a file system

Projeção do SO

Para criar um SO de ter sida em conta a **política** do mesmo, ou seja, o que o sistema irá realizar e o **mecanismo**, a forma como será realizado.

Ao integrar mecanismos sem política, o sistema torna-se facilmente adaptável.

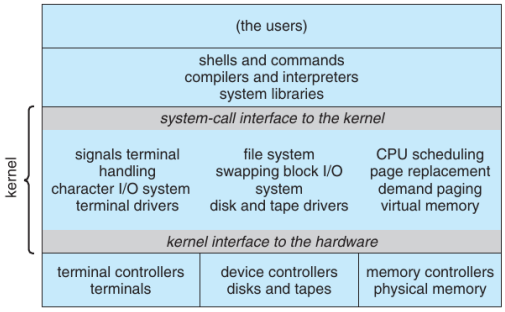
Arquiteturas do SO

Operating System Concepts, página 58

Um sistema operativo pode ser encarado como um todo, desenhado com funcionalidades estáticas, com código otimizado. Este sistema é designado por **monolítico** e caracteriza-se por ser pouco flexível e ocupar mais memória.

A sua estrutura simples geralmente implementa poucas funções, distinguindo muito levemente a interface e as funcionalidades, sendo difícil de implementar e manter. A fraca distinção leva a que muitas rotinas de I/O escrevam diretamente nos discos, o que deixa o sistema vulnerável a erros e a quebras de segurança.

O *Unix* (figura abaixo) é um exemplo de um sistema monolítico, diferenciando apenas o núcleo dos programas do sistema. Tudo o que fica abaixo das chamadas ao sistema e acima do *hardware* é o núcleo, sendo assim a maioria das funcionalidades (demasiadas) concentradas neste nível. Outro exemplo é o MS-DOS.



A maioria dos sistemas operativos modernos, no entanto, são desenhados por forma a serem utilizados e personalizados facilmente. Para o conseguir, as suas várias funções são abordadas em tarefas mais pequenas. Esta abordagem designa-se por **modular**, segundo a qual é possível adicionar e configurar funcionalidades através da inserção de módulos, tornando assim o SO mais flexível e mais pequeno (em termos de memória).

A **organização em camadas** é uma abordagem modular que consiste em dividir o SO em camadas, sendo a zero o *hardware* e a última a interface do utilizador. Uma camada consiste na implementação de um objeto com atributos e operações para o manipular. Uma camada é invocada apenas por outra camada superior e só pode invocar camadas inferiores, havendo uma abstração das camadas superiores em relação à implementação das inferiores.

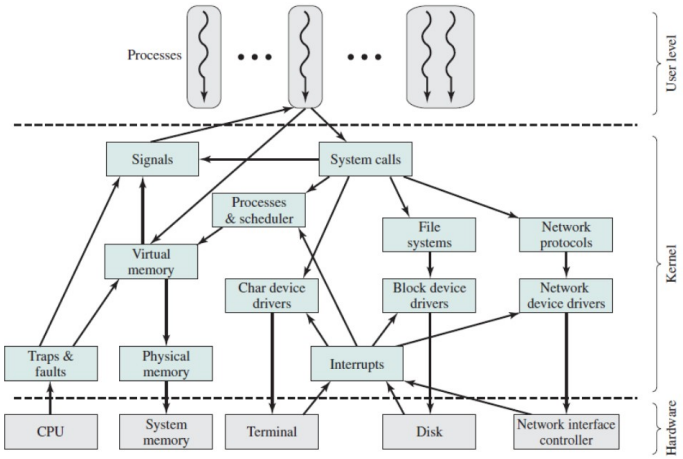
Esta organização permite uma depuração (*debugging*) mais eficiente, pois ao analisar uma cada, pressupõe-se que as camadas inferiores estejam a funcionar corretamente.

Outra organização possível é o **microkernel**, que consiste em remover do núcleo todas as funcionalidades não essenciais, que passam a ser executadas como programas ao nível do utilizador. O núcleo passa assim a funcionar como um intermediário entre as aplicações, agindo como um facilitador entre elas.

Por exemplo, um programa do utilizador que queira aceder a um ficheiro precisa de aceder ao servidor de ficheiros. Num núcleo "normal" o programa iria comunicar com o servidor, mas com esta organização a comunicação entre os dois é feita por via indireta, através do núcleo, que atua como intermediário.

O desempenho destes sistemas é penalizado, devido à sobrecarga do núcleo

Por fim existem ainda os sistemas **modulares**, que se focam na programação orientada a objetos. O *Linux* (figura abaixo) é um exemplo destes sistemas, cujos núcleos são compostos por alguns elementos base e ligações dinâmicas a serviços adicionais que podem correr durante a execução normal do SO ou aquando da inicialização da máquina. Outro exemplo desta organização é o sistema Solaris.



Assim, o SO não dá suporte por defeito a algumas funcionalidades, mas estas podem ser adicionadas dinâmicamente. Por exemplo os *drivers* para utilizar determinado dispositivo de I/O terão de ser adicionados ao núcleo, mas não vêm por defeito.

Apesar de ser semelhante à organização por camadas, uma vez que cada secção do núcleo é bem definida e detem interfaces protegidas, a modularidade permite a comunicação entre cada secção. Também existem algumas semelhanças com o *microkernel*, uma vez que cada secção do núcleo adicionada dinâmicamente pode ser considerado um programa do núcleo, mas aqui deixa de haver a necessidade de um moderador para as comunicações entre ambas as partes.

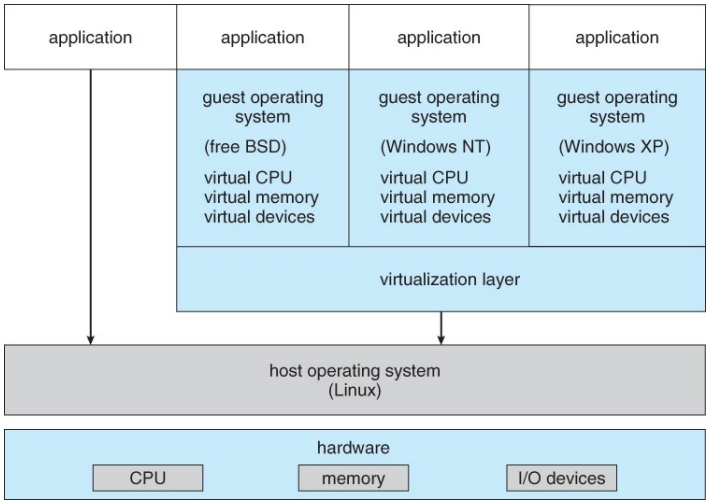
Componente prática sobre este tema desenvolvida no final deste documento

Máquinas virtuais

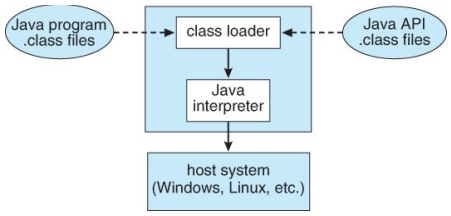
Operating System Concepts, página 64

O conceito de máquina virtual é utilizar uma abordagem em camadas para criar a abstração do *hardware* de uma máquina em vários ambientes de execução.

Podem assim ser executados na mesma máquina vários SO, sem que nenhum tenha a noção de que existe outro a correr em simultâneo. Cria-se assim a ilusão de que os recursos do computador são exclusivos, apesar de serem partilhados.

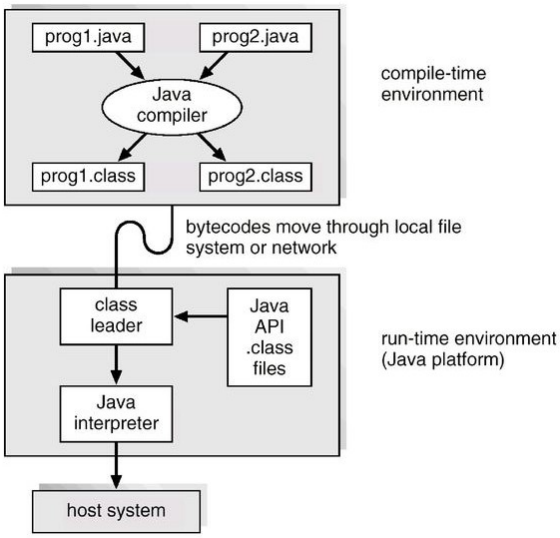


Um exemplo da utilização comum de máquinas virtuais é a linguagem Java. Esta caracteriza-se por ser multiplataforma graças à implementação da *Java Virtual Machine* (JVM) e da Java API. A máquina virtual recebe os ficheiros *.class* em *bytecode* do programa e da API do Java e converte-os em linguagem nativa do SO onde está a ser executada.



A cada programa em execução corresponde uma máquina virtual. Cada máquina virtual, **Java *Platform***, é composta pela JVM e API.

O ambiente de desenvolvimento em Java divide-se então em duas etapas: a compilação, que gera o ficheiro *bytecode .class* e a execução, na Java *platform*.



Programas de sistema

Operating System Concepts, página 77

Os programas de sistema fornecem um ambiente confortável para o desenvolvimento e execução de programas. Para a maioria dos utilizadores, o SO funciona através de programas de aplicação e sistema (sem verem o nível das chamadas ao sistema).[8]

Podem ser divididos em:

Manipulação de ficheiros

Criação, eliminação, cópia, renomeação, impressão, descarregamento, listagem e manipulação de ficheiros e diretórios.

Informação de estado

Fornecem informações como a data, hora, memória disponível ou espaço em disco, desempenho, depuração, etc.

Modificação de ficheiros

Criar e modificar o conteúdo de ficheiros em disco, busca de conteúdos...

Suporte às linguagens de programação

Compiladores, depuradores...

Carregamento e execução de programas

Depois de compilado, deve ser carregado em memória para ser executado

Comunicações

Fornecem mecanismo para a criação de conexões virtuais entre processos, utilizadores e sistemas de computação.

8 Por exemplo, interação através da GUI vs. CLI. São efetuadas as mesmas chamadas de sistema, mas com interfaces diferentes. O dual boot permite ter duas interfaces completamente diferentes, utilizando os mesmos recursos físicos.

Sistemas de ficheiros

Não é possível guardar dados numa unidade de memória de forma a manter as informações acessíveis e organizadas sem um tipo de estrutura que indique como os ficheirs devem ser gravados e lidos pelo SO.

Os sistemas de ficheiros tornam-se assim parte fundamental do SO e sem eles os dados armazenados seriam apenas um conjunto de zeros e uns.

Existem vários sistemas de ficheiros, cada um com as suas limitações ao nível do tamanho, número e comprimento do nome de ficheiros e de volumes.

Criteria	NTFS5	NTFS	exFAT	FAT32	FAT16	FAT12
Operating System	Windows 2000 Windows XP Windows 2003 Server Windows 2008 Windows Vista Windows 7	Windows NT Windows 2000 Windows XP Windows 2003 Server Windows 2008 Windows Vista Windows 7	Windows CE 6.0 Windows Vista SP1 Windows 7 WinXP+KB955704	DOS v7 and higher Windows 98 Windows ME Windows 2000 Windows XP Windows 2003 Server Windows Vista Windows 7	DOS All versions of Microsoft Windows	DOS All versions of Microsoft Windows
Limitations						
Max Volume Size	2 ^ 64 clusters – 1 cluster	2 ^ 32 clusters – 1 cluster	128PB	32GB for all OS. 2TB for some OS	2GB for all OS. 4GB for some OS	16MB
Max Files on Volume	4,294,967,295 2 ^ 32 -1	4,294,967,295 2 ^ 32 -1	Nearly Unlimited	268173300 ≈ 2^28	65536 = 2^16	
Max File Size	2 ^ 64 bytes (16 ExaBytes) minus 1KB	2 ^ 44 bytes (16 TeraBytes) minus 64KB	16EB	4GB minus 2 Bytes	2GB (Limit Only by Volume Size)	16MB (Limit Only by Volume Size)
Max Clusters Number	2 ^ 64 clusters – 1 cluster	2 ^ 32 clusters – 1 cluster	4294967295	268435444 = 2^28	65520	4080 ≈ 2^12
Max File Name Length	Up to 255	Up to 255	Up to 255	Up to 255	Standard - 8.3 Extended - up to 255	Up to 254
File System Features						
Unicode File Names	Unicode Character Set	Unicode Character Set	Unicode Character Set	System Character Set	System Character Set	System Character Set
System Records Mirror	MFT Mirror File	MFT Mirror File	No	Second Copy of FAT	Second Copy of FAT	Second Copy of FAT
Boot Sector Location	First and Last Sectors	First and Last Sectors	Sectors 0 to 11 Copy in 12 to 23	First Sector and Copy in Sector #6	First Sector	First Sector
File Attributes	Standard and Custom	Standard and Custom	Standard Set	Standard Set	Standard Set	Standard Set
Alternate Streams	Yes	Yes	No	No	No	No
Compression	Yes	Yes	No	No	No	No
Encryption	Yes	No	No	No	No	No
Object Permissions	Yes	Yes	Yes	No	No	No
Disk Quotas	Yes	No	No	No	No	No
Sparse Files	Yes	No	No	No	No	No
Reparse Points	Yes	No	No	No	No	No
Volume Mount Points	Yes	No	No	No	No	No
Overall Performance						
Built-in Security	Yes	Yes	Yes minimal ACL only	No	No	No
Recoverability	Yes	Yes	Yes if TFAT activated	No	No	No
Performance	Low on small volumes High on Large	Low on small volumes High on Large	High	High on small volumes Low on large	Highest on small volumes Low on large	High
Disk Space Economy	Max	Max	Max	Average	Minimal on large volumes	Max
Fault Tolerance	Max	Max	Yes if TFAT activated	Minimal	Average	Average

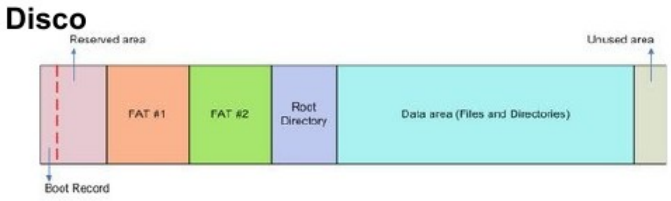
Sistema FAT32

Operating System Concepts, página 457

https://www.youtube.com/watch?v=V2Gxqv3bJCk

Um dos sistemas mais estudados como introdução a este tópico é o FAT32, que embora seja antigo (1966), representa o funcionamento geral da maioria dos sistemas de ficheiros utilizados atualmente. Denominado de **File Allocation Table**, foi criado pela Microsoft.

Funciona com base numa tabela que indica onde estão os dados de cada arquivo na memória, que está divida em blocos, podendo cada ficheiro ocupar mais do que um destes e não necessariamente de forma sequencial. No entanto, um bloco não pode ser ocupado por mais do que um ficheiro.



O disco é assim dividido em várias componentes:

Boot Tamanho da unidade, quantos setores tem cada cluster, onde começa fat1, fat2, raiz e área de gravação

FAT Tabela de alocação de ficheiros

Raiz e área de dados

Início da gravação do mapa dos arquivos (nome do arquivo, timestamp e cluster inicial)

A FAT é formado por clusters, que contém a informação do cluster seguinte que contém informação do ficheiro. Os clusters não são sequenciais ou contínuos, mas contíguos (próximos), podendo haver saltos entre clusters para definir um ficheiro. O último cluster da sequência é FF FF FF FF.

FAT

xxxxxxxx	xxxxxxxx	00000009	00000004
00000005	00000007	00000000	00000008
FFFFFFFF	0000000A	0000000B	00000011
0000000D	0000000E	FFFFFFFF	00000010
00000012	FFFFFFFF	00000013	00000014
00000015	00000016	FFFFFFFF	00000000
00000000	00000000	00000000	00000000
00000000	00000000	00000000	00000000
00000000	00000000	00000000	00000000
00000000	00000000	00000000	00000000
00000000	00000000	00000000	00000000
00000000	00000000	00000000	00000000
00000000	00000000	00000000	00000000
00000000	00000000	00000000	00000000
00000000	00000000	00000000	00000000
00000000	00000000	00000000	00000000

Root Directory:
2, 9, A, B, 11

File #1:
3, 4, 5, 7, 8

File #2:
C, D, E

File #3:
F, 10, 12, 13, 14, 15, 16

Entrada de diretoria

Root Directory SFN Entry Data Structure	
Bytes	Purpose
0	First character of file name (ASCII) or allocation status (0x00=unallocated, 0xe5=deleted)
1-10	Characters 2-11 of the file name (ASCII); the "." is implied between bytes 7 and 8
11	File attributes (see File Attributes table)
12	Reserved
13	File creation time (in tenths of seconds)*
14-15	Creation time (hours, minutes, seconds)*
16-17	Creation date*
18-19	Access date*
20-21	High-order 2 bytes of address of first cluster (0 for FAT12/16)*
22-23	Modified time (hours, minutes, seconds)
24-25	Modified date
26-27	Low-order 2 bytes of address of first cluster
28-31	File size (0 for directories)

Apesar de ser simples, a leitura de um ficheiro pode ser lenta se a tabela FAT não estiver em memória, a cabeça do disco volta sempre ao início para ler a tabela e para cada cluster para ler o conteúdo do seu endereço em disco.

O benefício é no acesso aleatório, porque a cabeça do disco consegue encontrar a localização de cada bloco pela leitura da informação na FAT.

Fazendo alguma abstração, este sistema de ficheiros relaciona-se com as listas ligadas.

Um ficheiro é nada mais nada menos do que um conjunto de nós, mas que em vez de serem seguidos, estão dispersos numa tabela. Para além dos endereços dos nós, a tabela mantém também o registo se essa linha está livre, ou atribuída.

Assim, sempre que queremos ler um ficheiro precisamos de saber qual o nó inicial (bloco da tabela – cluster) e a partir daí ler os blocos sequencialmente (cada um aponta para o próximo) até chegarmos ao bloco final (FFFFFFFF).

Quando quisermos escrever um ficheiro, só temos de procurar pela primeira linha da tabela que não esteja atribuída.

Componente prática sobre este tema desenvolvida no final deste documento

4. Sistemas de I/O

Operating System Concepts, página 535

Muitas das vezes que recorremos a um sistema operativo, fazêmo-lo para executar operações de I/O, sendo o processamento irrelevante, por exemplo, quando editamos um ficheiro ou consultamos uma página na internet.

Apesar de ter sido abordado anteriormente, este tópico desenvolve-se em muitos subtópicos.

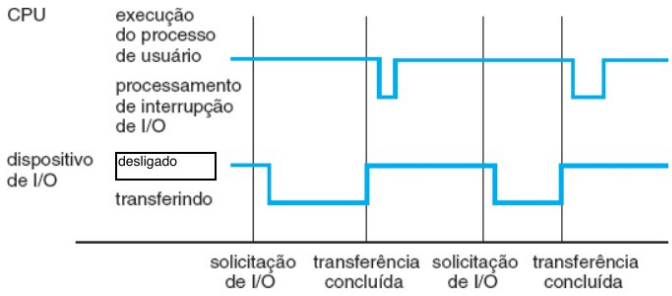
Interrupções

Operating System Concepts, páginas 7 e 539

As interrupções são fundamentais para o funcionamento de um SO.

Por exemplo, numa máquina com um único CPU a executar um ciclo infinito, se não existissem interrupções, seria impossível interagir com a mesma.

Esta interrupção pode ser proveniente do *hardware* (através do *bus*[9]) ou do *software* (através da operação especial chamada de sistema ou de monitor).



Quando a CPU é interrompida, para de imediato o processo atual e transfere a execução para uma localização fixa, normalmente contendo o endereço inicial no qual se encontra a rotina de serviço da interrupção. Quando completa a execução, a CPU retoma a computação interrompida.

Em alternativa a CPU podia fazer pedidos periódicos para saber se o processo já terminou, mas seria menos eficiente.

Quando são originadas pelo *software*, as interrupções são designadas também por ***traps*** ou **exceções**.

Exemplos são *breakpoints*, *overflow*, *acess violation*, *divide by 0*, *illegal instruction*...

9 Um *bus* é um sistema de comunicação eletrónico, que conecta computadores ou componentes dos mesmos.

Métodos (as)síncronos

Operating System Concepts, página 550

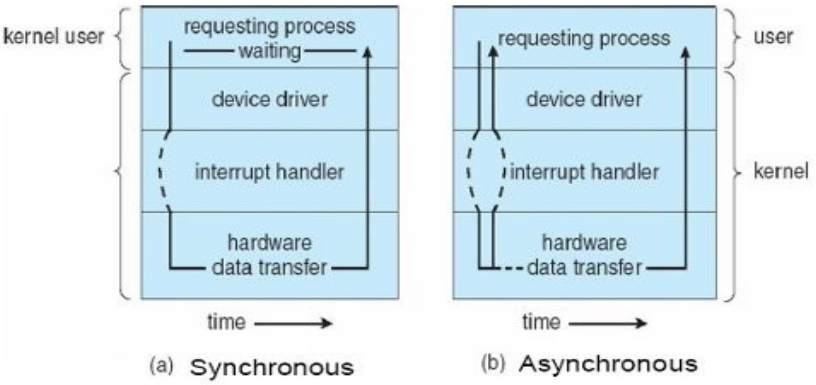
Existem duas abordagens para a realização de interrupções. As chamadas ao sistema **assíncronas**[10] suspendem a execução da aplicação até que o dispositivo de I/O responda.

Para que isto seja possível, a aplicação é movida da fila de execução para a fila de espera, até que a chamada ao sistema termine, retomando depois a normal execução, agora com os valores retornados pela chamada.

Em contraste existem as chamadas **síncronas**, que não bloqueiam a execução da aplicação enquanto fazem pedidos ao dispositivo de I/O. Eventualmente, num futuro próximo o dispositivo irá completar a sua execução e o seu retorno comunicado à aplicação, seja através de variáveis no espaço de endereçamento da aplicação, ou mesmo de um sinal.

Um exemplo de um método assíncrono é a reprodução de um vídeo em memória, durante o qual tem de ser feito um acesso à mesma em simultâneo com um processo de descompressão das imagens.

Outro cenário é a execução de uma aplicação que recebe constantes introduções de teclas ou cliques e reproduz em simultâneo conteúdo no ecrã.



10 Assíncrono, adj | Que não se realiza ao mesmo tempo que o outro.

5. Processos

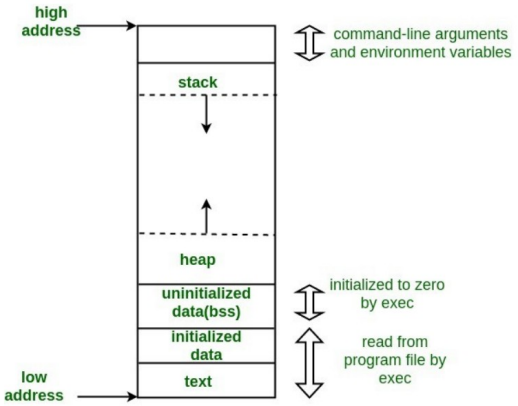
Operating System Concepts, página 85

Um processo é um programa em execução, que pode ser criado aquando da inicialização do sistema, de uma chamada ao sistema por um processo ou pelo pedido de um utilizador.

Estes podem ocorrer em ***foreground***[11], se interagirem com o utilizador, ou em ***background***, se forem executados sem interação, também designados por processos *daemon*[12].

Apesar de muitas vezes a execução de um programa ser associada ao código das instruções que o compõem, um processo é mais do que isso, constituindo para além do código, no *program counter* e nos registos associados ao mesmo.

Pode ainda fazer parte do processo a *stack,* a secção dos dados, que contém variáveis globais e a *heap*, memória alocada durante a execução.



O processo consiste assim numa cópia na memória física da imagem executável do programa, que tem associada uma *stack, heap*, secção de dados e instruções (texto). A imagem é criada através da compilação de um código fonte.

Componente prática sobre este tema desenvolvida no final deste documento

11 Expressão inglesa para primeiro plano.

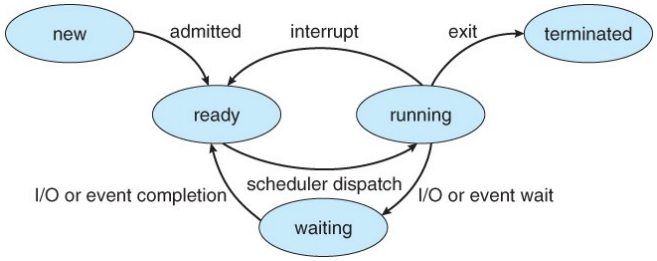
12 Expressão inglesa para demónio.

Estados de um processo

Operating System Concepts, página 87

Ao longo da sua execução, um processo assume vários estados.

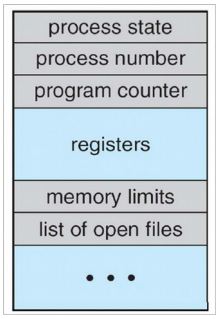
Novo	Se está a ser criado
Em execução	Instruções em execução
Em espera	O processo está à espera de um evento (I/O p.e.)
Pronto	O processo está à espera de ir para o processador
Terminado	O processo terminou a sua execução

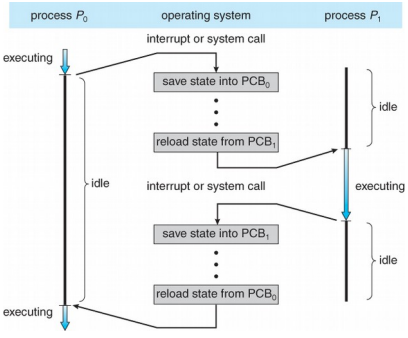


Bloco de controlo dos processos (PCB)

Operating System Concepts, página 87

Cada processo é representado pelo SO através de blocos de controlo de processos (ou tarefas). Estes são responsáveis por armazenar a informação associada com cada processo, nomeadamente o seu estado, PC, registos do CPU, tipo de escalonamento, informação sobre a memória do processo, sobre o I/O e informação estatística.





A troca de processos em execução no CPU materializa-se então na troca dos PCB.

Job scheduling

Operating System Concepts, página 89

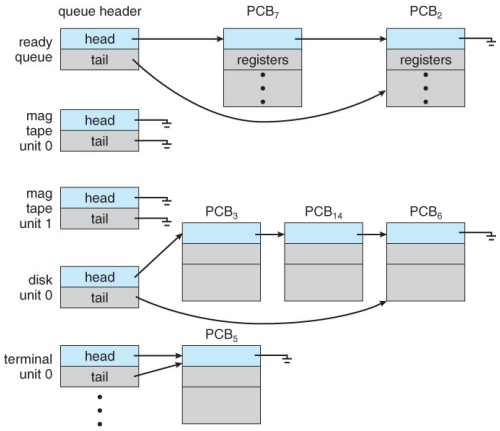
Quando foi abordada a multiprogramação, levantou-se a questão do agendamento da execução das aplicações que estão em espera no disco.

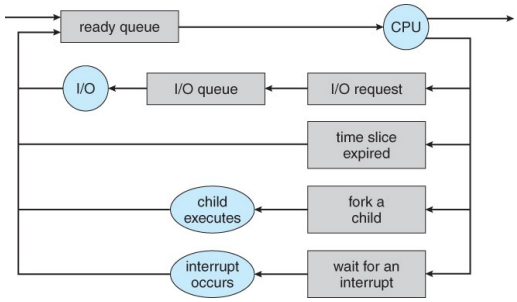
Para fazer o escalonamento dos processos, existem listas ligadas, que representam filas. A mais abrangente é a **job queue**, onde estão todos os processos do sistema.

Os que estão na RAM e prontos para serem executados estão na **ready queue**, sendo esta a primeira fila por onde qualquer programa passa e onde pode voltar, depois de uma pausa na sua execução (para esperar pela resposta de um dispositivo I/O, por exemplo).

Se um processo quiser aceder a um dispositivo de I/O, há outra fila correspondente ao mesmo, denominada de **device queue**, para o qual será transferido, evitando assim que multiplos processos acedam em simultâneo ao mesmo recurso.

Depois de realizarem algumas transições entre as várias filas e completarem a sua tarefa, os processos são terminados, sendo removidos de todas as filas e os seus recursos desalocados.





CPU scheduling

Operating System Concepts, página 92

O mecanismos de planeamento anterior responde ao problema de escalonamento dos processos que estão à espera em disco. No entanto, também é preciso decidir qual dos processos na RAM em estado *ready* devem ser executados.

Agendamento a curto (,médio) e longo prazo

Operating System Concepts, página 93

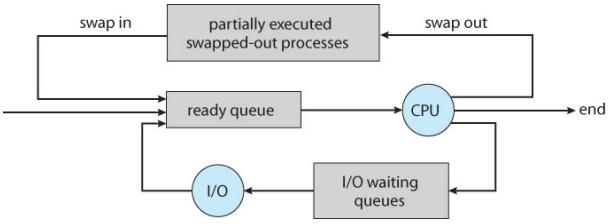
O mecanismo de agendamento mais frequente é o CPU scheduling, que também é executado mais rapidamente.

No lado oposto temos o job scheduling, que é um agendamento a longo prazo, decorrendo com pouca frequência (minutos de distância), mas representando um papel de extrema importância.

Existem programas que necessitam de grandes capacidades de computação e outros que necessitam de muita interação com dispositivos de I/O. Assim, é fundamental o job scheduling colocar em execução um misto entre os dois processos, para que o processador nunca fique desocupado por estar à espera de vários processos que estão a manipular dispositivos de I/O, nem os processos estejam muito tempo à espera no estado *ready*, por o processador estar ocupado com processos pesados que demoram algum tempo a executar.

No entanto, mesmo com um bom planeamento a longo prazo, por vezes é necessário melhorar a relação entre processos maioritariamente computacionais e de I/O ou suspender um processo devido a uma alteração na quantidade de memória disponível.

Para responder a este problema criou-se o **planemaneto a médio prazo**. Este atua nas situações descritas acima, removendo do CPU processos em execução, que são temporariamente suspensos. A esta suspensão, chamamos **swapping**.



Mais info sobre o agendamento do CPU no capítulo 8

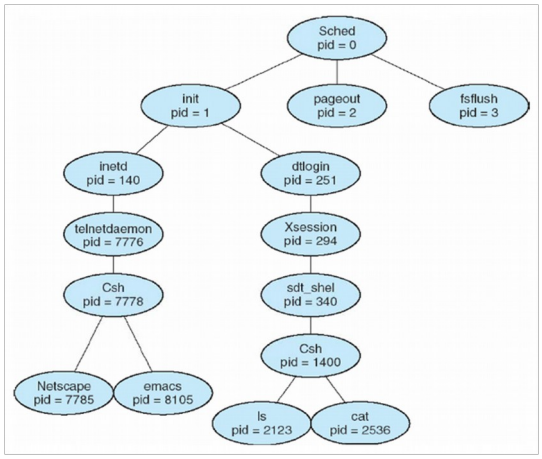
Criação de processos

Operating System Concepts, página 94

Um processo pode dar origem a vários processos, sendo o primeiro considerado o processo pai e os segundos, processos filhos. Nos SO, cada processo é identificado por um número inteiro (pid).

Estabelece-se assim uma hierarquia de processos, que pode ser representada atraés de uma **árvore de processos**.

Cada processo filho pode saber o pid do pai, que recebe o sinal *SIGCHLD* quando um filho morre, recolhendo o *exit code* dos filhos. Quando um pai morre antes do filho, este é herdado pelo processo init (1).



Esta árvore representa o SO Solaris. O processo pai é denominado por *Sched*, que por sua vez inicia três subprocessos. Um destes é o ***init***, processo pai de todos os processos de utilizador, que por sua vez inicia os processos *inetd*, responsável pelos serviços da rede (ftp, por exemplo) e *dtlogin*, o ecrã de início de sessão do utilizador, que depois de se entrar no sistema inicia uma nova sessão (um novo subprocesso), que cria o processo *sdt_shell*.

Eventualmente o utilizador abriu a linha de comandos (processo *Csh*), onde invocou os comandos *ls* e *cat*. Estes por sua vez podem ter recebido argumentos passados pelo processo pai, caso o utilizador tenha executado o comando *ls* para mostrar o conteúdo de uma pasta dando o caminho para a mesma ou o *cat* para mostrar o conteúdo de um ficheiro, por exemplo.

É de destacar que no processo *inetd*, eventualmente foi feito um acesso à máquina através do protocolo telnet, sendo este feito através de um terminal criado através do processo *Csh*, através do qual foi iniciado o navegador da internet *Netscape* e o editor *emacs*.

A forma como os subprocessos são implementados pode no entanto variar, podendo este fornecer novos recursos ao novo processo ou haver uma partilha ou uma partição dos recursos do processo pai pelos filhos. Há também a possibilidade do processo pai passar dados de inicialização aos filhos.

O processo pai pode ou não esperar pela execução do processo filho e pode ou não suspender a sua execução até que o filho termine.

O processo filho pode duplicar o processo pai (manter o mesmo programa e informação), ou ser gerado a partir de um novo programa.

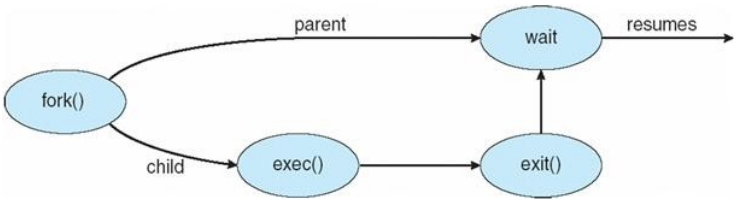
Todas estas hipóteses foram abordadas nas aulas práticas utilizando a linguagem C.

Ao executar a função *fork()*, criamos um subprocesso do atual, duplicando o espaço de endereçamento do processo pai, com a única diferença no valor de retorno da função, que é 0 para o processo filho e >0 para o pai.

Eventualmente, podemos chamar a função *execl()* no processo filho, que vai carregar para a memória um ficheiro binário que vai executar, eliminado a imagem em memória do programa que invocou a função. Por fim, terminamos o processo filho com a função *exit()*.

O processo pai podia estar durante o tempo de execução do filho em espera, recorrendo à função *wait()*, que o iria remover da fila de execução até que o filho termine (assumindo o estado *waiting*), voltando nesta altura ao estado *ready*.

O esquema abaixo representa este programa.

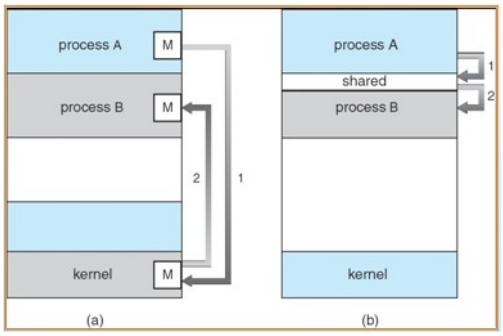


Componente prática sobre este tema desenvolvida no final deste documento

Comunicação entre processos

Operating System Concepts, página 101

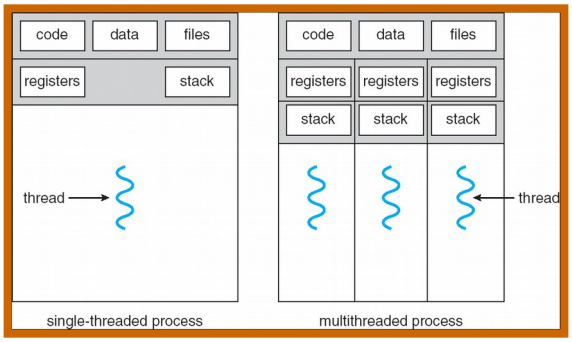
A comunicação etre processos/*threads* pode ser feita através da memória partilhada (b), ou através da troca de mensagens (a), mediada pelo núcleo.



6. Threads

Operating System Concepts, página 133

Uma *thread* é um caminho de execução de um processo, podendo existir mais do que uma para cada processo. Ou seja, um processo pode ser dividido em várias *threads*, que na prática são partes do processo que decorrem em paralelo, partilhando uma memória única (a do processo).

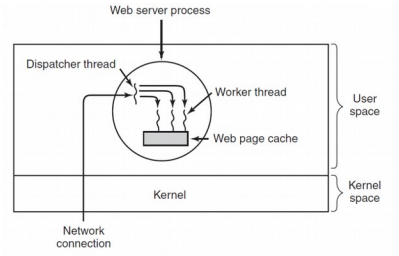


Threads diferentes não são independentes, ao contrários dos processos, partilhando a maioria dos os recursos: código, memória, registos, espaço de endereçamento, variáveis globais, ficheiros abertos, processos filho, alarmes e recursos do SO (*flags* e *signals*).

Cada *thread* tem o seu PC (*program counter*), *set* de registos, estado e *stack*.

Em dispositivos com multiprocessadores, o *multithreading* permite dividir a execução de um processo pelos vários processadores, aumentando assim a sua velocidade de execução.

Um exemplo prático é um servidor *web*. Este consiste num processo único, que para responder rapidamente aos vários pedidos que lhe são feitos em simultâneo, recorre a *threads*. Existe uma *thread* que recebe todos os pedidos (*dispatcher*) e os distribui pelas (*working*) *threads*.



Vantagens das threads

Operating System Concepts, página 135

As *threads* permitem tornar um programa modular, resposivo, e com melhor desempenho, tirando partido da partilha de recursos e máximo uso das arquiteturas multi-processador.

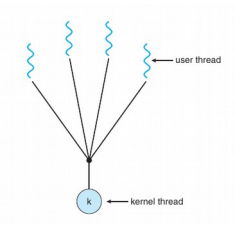
Modelos de multithreading

Operating System Concepts, página 135

As *threads* podem ser implementadas ao **nível do utilizador**, sendo a sua gestão feita por uma biblioteca que corre em modo de utilizador ou ao **nível do núcleo**, sendo a gestão feita diretamente por este.

Apesar de terem implementações distintas, é necessário estabelecer uma relação entre as duas, que pode ser feita de várias maneiras.

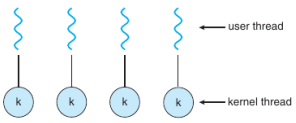
Many-to-one



Várias *threads* do utilizador são mapeadas numa *thread* do núcleo, sendo o acesso a este alternado, fazendo com que não seja tirado partido de multiprocessadores.

Se uma *thread* fizer uma chamada de sistema para bloquear, todo o processo bloqueia.

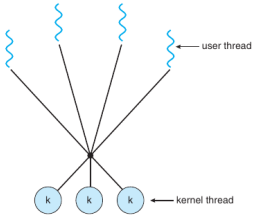
One-to-one



Cada *thread* do utilizador é mapeada numa *thread* nuclear. No entanto, existe um número máximo de *threads*, para evitar perdas de performance.

Se uma *thread* bloquear, as restantes continuam em execução.

Many-to-many



Várias *threads* do utilizador mapeadas em várias *threads* do núcleo, sendo no entanto o número de *threads* nucleares menor do que as de utilizador, que pode variar de acordo com a aplicação e sistema.

Quando uma *thread* bloqueia, o núcleo pode agendar outra para execução.

Bibliotecas de threads

Operating System Concepts, página 137

Fornecem ao programador uma API para criar e gerir *threads*. A sua implementação pode ser ao nível do utilizador ou do núcleo.

Pthreads

Define o comportamento, mas não implementação das *threads*, sendo a última responsabilidade do programador. É uma norma do IEEE para a compatibilidade dos SO e está implementado nos sistemas *Linux*. Implementado através do POSIXAPI.

Thread call	Description
Pthread_create	Create a new thread
Pthread_exit	Terminate the calling thread
Pthread_join	Wait for a specific thread to exit
Pthread_yield	Release the CPU to let another thread run
Pthread_attr_init	Create and initialize a thread's attribute structure
Pthread_attr_destroy	Remove a thread's attribute structure

Java Threads

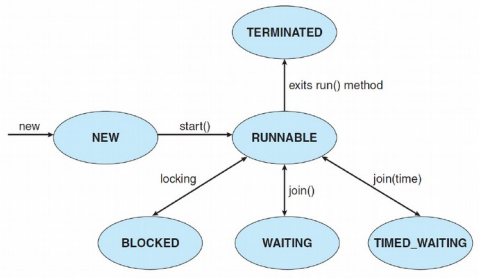
Geridas pela JVM, as Java *Threads* recorrem com frequência às bibliotecas do sistema, de forma transparente (Pthreads no *Linux* e *Solaris* e Win32 no *Windows*).

```
public class RunHello implements Runnable {
 public void run() {
    System.out.println("Hello!");
 }
}

public class mainThread {
  public static void main(String args[]) {
     Thread th =new Thread(new RunHello());
     th.start();
  }
}
```

```
public class ThHello extends Thread {
 public void run() {
    System.out.println("Hello!");
 }
}

public class mainThread {
  public static void main(String args[]) {
     ThHello th = new ThHello();
     th.start();
  }
}
```

Estas podem ser criadas através de classes com suporte para a interface *Runnable*, ou derivadas de *Thread.*

A classe *Thread* define seis estados para as *threads*, sendo o primeiro **New**, estado seguinte à criação, mas antes do início da execução (função *start()*), seguido do **Runnable**, que é o estado de execução. A partir deste estado é alocada memória para a *thread* e podem haver transições para os restantes quatro, havendo a possibilidade de bloqueio (por um pedido de I/O, p.e.) e entrar no estado **Blocking**, um *join()* e entrar em **Waiting** ou em **Timed_Waiting**, caso tenha sido passado um tempo como argumento. Por fim, para terminar a *thread* executa-se a função *exit()* e passa para o estado **Terminated**.



Problemas das Threads

Operating System Concepts, página 147

fork() e exec()

Apesar de terem sido abordadas anteriormente, o modo de funcionamento destas funções é diferente quando em programas *multithread*.

No que toca ao *fork()*, os SO implementam duas abordagens. A versão normal duplica apenas a *thread* que invocou a função e criou-se uma nova versão, a *forkall()*, que duplica todas as *threads*.

É importante que após a utilização do *fork()* se utilizem apenas funções *async-safe*. Podem ainda ser gerados problemas de memória (inconsistente) e semáforos (bloqueados).

Em relação ao *exec()*, em geral substitui todo o processo, incluindo todas as *threads*.

Cancelamento das Threads

Por vezes é necessário cancelar a execução de uma *thread* antes que esta termine a sua execução normal. Para tal, existem duas abordagens, a **síncrona**, quando uma *thread* verifica periodicamente se deve terminar e a **assíncrona**, quando a *thread* é terminada de imediato por outra.

O cancelamento de *threads* pode ser importante quando por exemplo tiverem a ser utilizadas várias numa pesquisa a uma base de dados e uma encontra o que procuram, num *browser*, que utiliza *multithreading* para carregar os vários elementos de uma página, quando o utilizador cancela o carregamento da mesma.

Em ambos os cenários, é fundamental cancelar a execução das *threads* antes que estas terminem de correr os seus processos. No primeiro cenário, a *thread* que encontrou o que procuravam deve cancelar as outras e no caso do navegador, as *threads* que estão a carregar componentes da página no momento do cancelamento não precisam de continuar a carregar, pois a página já não vai ser mostrada.

O cancelamento assíncrono não é recomendado, pois uma *thread* ser cancelada sem aviso prévio pode interromper processos de escrita, em que lhe estão alocados determinados recursos, que apesar de parte serem recuperados pelo SO de forma automática, podem haver alguns meios que lhe fiquem atribuídos.

Apesar de implementado nas *threads* do Java, é *deprecated* (desaconselhado).

Atendimento de sinais

Para notificar um processo de que um determinado evento ocorreu, em *Unix*, utilizam-se sinais.

Estes podem ser **síncronos**, quando são gerados pelo próprio processo que os recebe, em tempo real, ou **assíncronos**, quando gerados por um evento externo ao processo que o recebe.

Um exemplo de um sinal síncrono é a divisão por zero ou acesso ilegal à memória. Sinais assícronos são por exemplo o cancelamento da execução de um programa através das teclas (CTRL+C).

Para cada sinal existe um gestor de sinal por defeito, que é corrido pelo núcleo do SO. No entanto, estes gestores podem ser substituídos por gestores definidos pelo utilizador.

O modo como o sinal gerado é entregue é bastante simples quando um programa é *single thread*. Já no que toca a *multi thread*, esta pode variar, podendo o sinal ser entregue...

1. ... à *thread* à qual o sinal se aplica;
2. ... a todas as *threads* do processo;
3. ... a algumas *threads**;
4. ... a uma *thread* pré-definida para receber todos os sinais do processo.

*As *threads* podem especificar quais os sinais que aceitam, sendo neste caso os sinais entregues apenas às que aceitam.

Thread pools

A coexistência de um número descontrolado de *threads* pode levar ao desgaste dos recursos disponibilizados pelo SO. Para evitar esta situação e responder ao problema do tempo necessário para criar uma *thread*, criaram-se ass *thread pools*.

Esta solução consiste em criar um número pré-definido de *threads* quando o processo é iniciado, que ficam à espera que lhes seja atribuída alguma função numa *pool*. Quando há trabalho a ser feito, este é atribuído a uma *thread* disponível, caso contrário, o trabalho espera que haja uma *thread* disponível.

Assim, deixa de haver necessidade de estar constantemente a criar *threads* e a libertá-las e consequentemente aos recursos do SO sempre que há um trabalho a ser feito.

Cria-se também um limite ao número de *threads*, o que garante uma maior eficiência e aumenta-se a rapidez da resposta aos pedidos feitos ao processo.

Em Java as *thread pools* podem ser geridas pela interface *Executor*. Esta disponibiliza métodos para criar uma única *thread* – *newSingleThreadExecutor()* - ou várias,

havendo um número fixo – *newFixedThreadPoll(int size)* – ou dinâmico – *newCachedThreadPool()*, que não tem limite na criação, mas reutiliza as existentes sempre que possível (passado algum tempo sem serem reutilizadas, são descartadas).

+ ver programa disponibilizado nas aulas teóricas

Threads nos sistemas operativos

Operating System Concepts, página 195

No **windows**, as *threads* são geridas pela Win32API, que implementa um mapeamento *one-to-one*. No entanto, tem ainda suporte para a biblioteca *fiber*, que permite a criação de *threads many-to-many*, através da qual, cada *thread* tem um id, conjunto de registos, *stacks* do utilizador e do núcleo (dependendo do modo em que a *thread* está a correr) e uma área privada.

Em **linux**, as *threads* são interpretadas como *tasks*, sendo criadas através da chamada ao sistema *clone()*, que, através da passagem de determinadas *flags* como argumento permite criar *threads* que partilham os recursos do processo.

7. Sincronização de processos

Operating System Concepts, página 209

Condição de corrida e região crítica

Operating System Concepts, página 211

Quando várias *threads* acedem a dados partilhados, por vezes o resultado final depende da forma inesperada da ordem de execução (aleatório). Assim, é fundamental a criação de **regiões críticas**: zonas do código que manipula dados partilhados e que não pode ser executada concorrentemente por mais do que uma *thread*.

Para i=5, quando é feito i++ e i-- por duas *threads* em simultâneo, i pode ficar com o valor 4, 5 ou 6.

Para manipular estas regiões é fundamental a implementação de código que faz o pedido de acesso à região crítica, a **região de entrada** e o código que é executado após a saída da região crítica, "libertando" a sua utilização, a **região de saída**.

É assim criada a **condição de corrida**, sendo fundamental assegurar a exclusividade mútua (*mutex*) no acesso à região, o progresso, pois o acesso a um processo em espera não pode ser adiado indefinidamente, sendo assegurado pela espera limitada, um limite ao número de vezes que é concedido o acesso à região crítica após um determinado processo ter pedido esse acesso até que esse pedido seja satisfeito.

Implementação de soluções de software

Operating System Concepts, página 213

Alternância estrita

Consiste em ter uma variável partilhada que pode assumir n valores para n *threads* acedem à região crítica quando a variável assume o seu valor correspondente.

```
while(turn==j)
      //criticalRegion
```

Entre duas *threads*, turn varia entre 0 e 1. Quando 0 termina o acesso muda turn para 1 e vice-versa.

Algoritmo de Peterson

Para dois processos, consiste em criar duas variáveis partilhadas, uma para fazer a alternância estrita e um *array* com dois elementos, que indica no índice correspondente ao id da *thread* se esta pretende aceder à região crítica.

```
while(flag[j] && turn==j)
```

```
        //criticalRegion
```

Existe ainda o algoritmo de Dekker, que é similar ao de Petterson e embora seja apenas aplicável à alternância entre duas *threads*, é mais eficiente.

Implementação de soluções de hardware

Operating System Concepts, página 214

Todas as soluções para a sincronização têm por base o (des)bloqueio do acesso a uma determinada região. Como vimos anteriormente, este pode ser feito através do *software*. No entanto, existem também soluções com base em *hardware*.

Em **uniprocessadores** podemos desativar as interrupções quando está a ser feita uma alteração numa variávelm partilhada.

No entanto, este método não é eficiente para **multiprocessadores**, sendo neste caso necessário criar intruções atómicas especiais, que testam valor na memória e alteram esse valor. Um exemplo é a utilização do método *getAndSet()* que devolve o valor atual da variável e altera-o para o valor passado como argumento.

```
// lock is shared by all threads
HardwareData lock = new HardwareData(false);

while (true) {
   while (lock.getAndSet(true))
     Thread.yield();

   criticalSection();
   lock.set(false);
   remainderSection();
}
```

O programa só sai do ciclo *while* quando o retorno do *getAndSet()* for *false*, ou seja, quando o *lock* estiver livre.

Apesar de funcionarem, estas soluções são complicadas de utilizar. Por isso, foi criada uma nova ferramenta para a sincronização de processos em *hardware*, os **semáforos**.

7.1. Semáforos

Operating System Concepts, página 217

https://www.youtube.com/playlist?list=PLWi7UcbOD_0vs6Q32c5j22BTFsEVU344X

Semáforos são processos de sincronização que permite a comunicação eficiente entre processos ou *threads*.

Consistem numa alternativa aos mecanismos apresentados como solução para as condições de corrida nas *threads*, pois não necessitam de *busy waiting*, ou seja, deixam de necessitar de uma verificação constante e repetida no tempo de uma determinada condição para aceder às regiões críticas.

São caracterizados por uma variável inteira, que pode tomar qualquer valor inteiro (*counting*), ou ser binária (*mutex*) e à qual estão associadas as funções *acquire()* e *release(),* sendo a primeira realisada antes e a última no final do acesso a uma região crítica.

```
acquire() {
    while value <= 0
       ; // no-op
    value--;
}
```

```
release() {
    value++;
}
```

A solução mostrada acima não é a melhor, pois continua a estar assente em *busy waiting*. Assim, quando um processo tem de esperar por um semáforo é colocado numa fila de espera própria, através do processo *block()*, sendo libertado quando um dos processos em execução faça *release()*, onde executa a função *wakeup()*. Estas funções constituem regiões críticas.

```
acquire(){
    value--;
    if (value < 0) {
       add this process to list
       block;
    }
}
```

```
release(){
    value++;
    if (value <= 0) {
       remove a process P from list
       wakeup(P);
    }
}
```

O número inteiro no qual o semáforo é inicializado representa o número de processos que podem ser executados em simultâneo.

Os *mutexers* são muito utilizador para acessos à região crítica de um processo, por exemplo. Os *counting*, permitem múltiplos acessos, mas limitados e são utilizados quando um recurso admite um número máximo de instâncias.

Deadlock e starvation

Operating System Concepts, página 222

Quando um processo X está a utilizar recurso A e um processo Y em simultâneo está a usar o recurso B e para terminar o processo X precisa de aceder ao B e o Y ao A, dá-se um **deadlock.**

Este cenário pode ser descrito de forma abstrata por um casal de namorados que teve uma discussão, em que cada um dos elementos do casal quer pedir desculpa ao outro, mas se e só se o outro pedir desculpa primeiro. Como ambos estão à espera que o outro tome a iniciativa, nada se resolverá.

Fonte: https://stackoverflow.com/a/35640575/10735382

Para evitar este cenário, deve-se ter especial atenção à utilização de bloqueios de acessos e se estes forem necessários, fazê-los sempre na mesma ordem!

Outra possibilidade é a imposição da libertação de recursos, ou seja, se não for concedido acesso a um deles, libertar o que conseguiu.

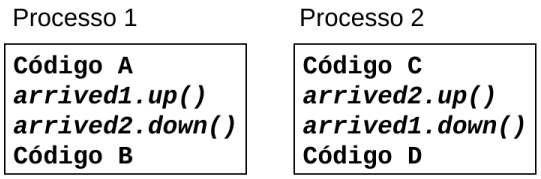
Outro problema que pode surgir é a **starvation**, que ocorre quando a fila de espera do semáforo segue a filosofia LIFO (*last in first out*) e existem muitos processos a aceder ao mesmo, podendo algum esperar infinitamente pelo acesso.

Mecanismos básicos de sincronização

Nos *mutexers*, as funções *acquire()* e *release()* são abstraídas nas funções *down()* e *up()*.

Estes podem ser utilizados para além do controlo do acesso a regiões críticas, para sicronizar código em diferentes *threads* ou processos, impondo assim uma ordem na sua execução. A este processo chama-se **signaling**.

Outro mecanismo básico de sincronização é denominado por **rendezvous**, que consiste em que determinado código de dois processos só seja executado quando algum código antes desse seja executado, também em ambos os processos.



Este pode ser generalizado a vários processos, com a introdução de um novo semáforo, a **barreira** e um inteiro partilhado.

```
mutex.down()
count++;
if(count == N) {
   for(i=1..N) barrier.up()
   count=0;
}
mutex.up ()
barrier.down()
```

Ainda assim esta solução tem problemas, pois apenas funciona se só for executada uma vez. Se for cíclica, temos de considerar a barreira como um *array* de semáforos.

```
mutex.down()
count++;
if(count == N) {
   for(i=1..N) barrier[i].up()
   count=0;
}
mutex.up ()
barrier[j].down()
```

Outra opção válida é a utilização de dois semáforos barreira e em vez de um, dois inteiros partilhados.

```
mutex.down()
count++; localt=turn;
if(count == N) {
   for(i=1..N) barrier[localt].up()
   count=0; turn=1-turn;
}
mutex.up ()
barrier[localt].down()
```

Problemas de sincronização

Operating System Concepts, página 222

Bounded-buffer (produtor-consumidor)

Também designado por problema do produtor-consumidor, este consiste num problema clássico de sincronização, onde um produtor coloca um determinado "produto" num *buffer*, que vai ser consumido por um consumidor, que deve ser notificado quando o produto está disponível.

Este problema resolve-se com a implementação de três semáforos. Um *mutex* para controlar o acesso à região crítica e dois outros semáforos: o *full()*, inicializado a 0 e utilizado pelo produtor para notificar o consumidor de que há produto disponível e o *empty()*, inicializado com o tamanho do *buffer()* e utilizado pelo consumidor para notificar o produtor que consumiu um produto (que precisa de ser reposto).

Em ambas as entidades o primeiro *down()* a fazer, neste caso, *acquire()* é respetivamente para o produtor e para o consumidor, dos semáforos *empty* e do *full*, seguido em ambos do *mutex*. Depois de atualizarem o *buffer*, fazem *release()* do *mutex* e por fim dos primeiros semáforos que bloquearam.

```
public void insert(Object item) {
   empty.acquire();
   mutex.acquire();

   // add an item to the buffer
   buffer[in] = item;
   in = (in + 1) % BUFFER_SIZE;

   mutex.release();
   full.release();
}
```

```
public Object remove() {
   full.acquire();
   mutex.acquire();

   // remove an item from the buffer
   Object item = buffer[out];
   out = (out + 1) % BUFFER_SIZE;

   mutex.release();
   empty.release();

   return item;
}
```

Escritores e leitores

Este é um problema muito comum em bases de dados, onde podem ser feitas operações de leitura ou de escrita (que pressupõe a leitura).

A primeira pode ser feita em simultâneo por vários processos, sem qualquer problema. No entanto, a escrita na base de dados de mais do que um processo, ou a escrita de um processo em simultâneo com a leitura de outro podem dar problemas.

É assim fundamental garantirmos que a escrita é um processo de acesso exclusivo, o que não tem de acontecer com a leitura.

Existem duas implementações que podem parecer óbvias. Uma em que os leitores têm prioridade (se um estiver a ler e houver um escritor em espera, o leitor não precisa de esperar que o atual termine de ler, então passa à frente do escritor e começa a ler também) e outra em que os escritores têm prioridade. Em ambos os cenários pode haver *starvation*, respetivamente dos escritores e dos leitores.

A melhor solução consiste em ter um inteiro para contar o número de leitores e três semáforos, um *mutex* para aceder à região crítica (atualizar o inteiro), um *nobody* para garantir acesso exclusivo à base de dados (utilizado pelos escritores) e um *turnstile* para alternar entre os escritores e os leitores.

```
readStart()
  turnstile.down()
  turnstile.up()
  mutex.down()
  if(readers==0)
    nobody.down()
  readers++
  mutex.up()
read access
readEnd()
  mutex.down()
  readers--
  if(readers==0)
    nobody.up()
  mutex.up()
```

```
writeStart()
  turnstile.down()
  nobody.down()
write access
writeEnd()
  turnstile.up()
  nobody.up()
```

Jantar de filósofos

Neste problema, vários filósofos vão jantar numa mesa redonda, tendo à sua frente uma taça de comida e entre cada taça um garfo. Cada filósofo só pode comer quando tiver dois garfos! Supondo que todos os filósofos agarram no garfo à esquerda, deixam de ter o garfo da direita disponível, pelo qual vão ficar à espera. Gera-se um deadlock.

Para resolver este problema introduzimos um empregado de mesa no cenário, que restringe a utilização dos garfos a um filósofo de cada vez (que pode agarrar em um ou dois de cada vez), devendo antes disto pedir permissão a esta entidade.

Assim, quando um filósofo tem fome, pede permissão que recebe uma resposta afirmativa e agarra nos dois garfos à sua esquerda e direita. De seguida, quando o filósofo ao seu lado pede permissão, que recebe resposta afirmativa, este tenta agarrar nos dois garfos, sem sucesso, dizendo ao empregado que afinal não vai comer nesse momento. Quano um outro filósofo com pelo menos outro de intervalo entre si e o que está a comer tem fome, pede para comer e ao receber autorização e verificar que há dois garfos ao seu lado, começa a comer.

O empregado de mesa atua como um **monitor**, que quando está a ser acedido por alguma entidade, mais ninguém lhe pode aceder. Quando um filósofo está a falar com o empregado, este está bloqueado para si apenas e mais nenhum pode comunicar com ele, até que o filósofo o desbloqueie (quando agarra nos talheres, verifica que não consegue agarrar ou quando os pousa depois de comer).

No entanto há ainda problemas por resolver. Se os primeiros a comer forem os filósofos 1 e 3 e os seguintes forem o 2 e o 4, o seguinte deve ser o 5º, mas pode acontecer voltar a ser o 1 e o 3. Precisamos ainda de introduzir um mecanismo de agendamento para priorizar a alimentação dos filósofos. Uma possibilidade é avaliar o número de minutos desde a última vez em que comeu e assim o 5 teria prioridade sobre o 1 e o 3.

Fonte: https://www.youtube.com/watch?v=NbwbQQB7xNQ

Na prática, este problema pode ser implementado com um *array* de semáforos para os garfos e outro *limit4()*, que impede que os cinco filósofos tentem pegar nos garfos ao mesmo tempo (apenas permite 4).

```
while(true)
  think()
  getForks()
    limit4.down()
    forks[left(f)].down()
    forks[right(f)].down()
  eat()
  putForks()
    forks[left(f)].up()
    forks[right(f)].up()
    limit4.up()
```

Existem ainda outras alternativas como permitir que os filósofos pares peguem primeiro no da esquerda e os ímpares no da direira.

Outra é a utilização de varíaveis de estado, complementares ao *mutex* e ao *array* de semáforos para os garfos. Solução de Tanembaum.

```
getForks()
  mutex.down()
  st[f]=HG
  test(f)
  mutex.up()
  filos[f].down()
```

```
putForks()
  mutex.down()
  st[f]=TK
  test(left(f))
  test(right(f))
  mutex.up()
```

```
test(f)
  if(st[f]==HG)
    and st[left(f)]!=ET
    and st[right(f)]!=ET
    st[f]=ET
    filos[f].up()
```

7.2. Monitores

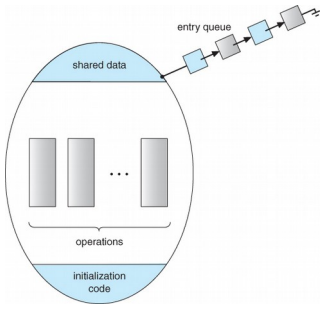
Operating System Concepts, página 231

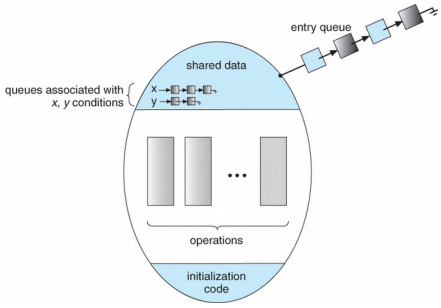
Nos pontos anteriores foi abordada a utilização de semáforos como resposta às condições de corrida sem *busy waiting*. No entanto, existem cenários em que os semáforos não são 100% eficientes, gerando *deadlocks* ou violando os princípios da exclusão mútua.

Em resposta foram criados os **monitores**, uma abstração de alto nível utilizada na sincronização de processos. Um monitor é então um tipo de dados abstrato que armazena dados privados manipuláveis através de métodos públicos, que são acedidos em exclusão mútua.

Se uma *thread* estiver a executar uma função deste objeto e outra tentar também executar um função do mesmo objeto terá de esperar que a primeira termine: apenas um processo pode estar ativo dentro do monitor de cada vez.

Os dados declarados dentro do monitor são apenas acessíveis aos seus métodos, tal como os seus métodos não têm acesso aos dados exteriores ao mesmo.





Os monitores disponibilizam assim mecanismos de exclusão mútua. No entanto, pode ainda ser preciso suportar a sincronização. Para isso foram criadas as **variáveis condicionais**. Sobre estas podem ser aplicadas apenas duas ações: *wait()* e *signal()*.

A primeira faz com que o processo que a invoca fique bloqueado até que a segunda seja executada. Se na execução da *signal()*, não existirem processos em espera, nada acontece, mesmo que no futuro hava algum em espera, não o desbloqueia.

Isto contrasta com os semáforos, que quando lhes é feito *up()*, se não existir nenhum em espera, se no futuro algum processo fizer *down()*, não terá de esperar.

Resolução do sinal

Operating System Concepts, página 234

A forma como o sinal é interpretado pode variar, existindo três abordagens distintas:

- **Signal and continue** quando o processo que é invocado espera que o processo invocador saia do monitor para executar;

Pode gerar *starvation* (espera infinita).

- **Signal and wait** quando o processo invocador pausa a sua execução para que o invocado execute e retoma no final;

É necessária uma *stack* para colocar os processos pausados.
Argumento a favor é que não faz sentido atender um *signal()* no final da execução do processo invocador, porque nessa altura o que motivou à invocação pode já não se verificar.

- **Signal and leave** quando o processo invocador termina a sua execução quando faz o sinal, começando de seguida a execução do processo invocado, não sendo o primeiro retomado.

Problemas de sincronização

Operating System Concepts, página 235

Jantar de filósofos

A solução com recurso a um monitor implica a criação de um vetor de estados. Em 5 filósofos, antes de começar a comer, cada filósofo tem de verificar se o filósofo i+1%5 e i+4%5 está no estado *thinking* e apenas nesse caso pode começar a comer. No final de comer pousa os talheres, fazendo *signal()* para que os filósofos ao seu lado possam comer, caso hajam talheres disponíveis para tal.

```
monitor DiningPhilosophers
{
    enum State {THINKING, HUNGRY, EATING};
    State[] states = new State[5];
    Condition[] self = new Condition[5];

    public DiningPhilosophers {
       for (int i = 0; i < 5; i++)
          state[i] = State.THINKING;
    }

    public void takeForks(int i) {
       state[i] = State.HUNGRY;
       test(i);
       if (state[i] != State.EATING)
          self[i].wait;
    }

    public void returnForks(int i) {
       state[i] = State.THINKING;
       // test left and right neighbors
       test((i + 4) % 5);
       test((i + 1) % 5);
    }

    private void test(int i) {
       if ( (state[(i + 4) % 5] != State.EATING) &&
          (state[i] == State.HUNGRY) &&
          (state[(i + 1) % 5] != State.EATING) ) {
             state[i] = State.EATING;
             self[i].signal;
       }
    }
}
```

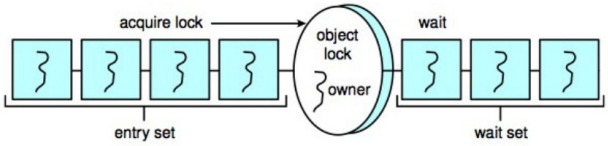
Sincronização em Java

Operating System Concepts, página 236

A implementação dos monitores em Java é feito através da definição dos métodos como ***synchronized***.

Um mecanismo semelhante ao de *busy waiting* em Java é o *Thread.yield()*[13], que apesar de não alterar o estado de *running* do mesmo, permite ao CPU executar processos concorrentes. No entanto, nos monitores isto não é possível, uma vez que apenas um processo pode estar neste estado de cada vez no monitor. A utilização deste método pode levar ao **livelock**.

A forma mais correta de gerir a sincronização é então através dos métodos *wait()* e *notify()*, que colocam bloqueiam e desbloqueiam os processos, respetivamente. Quando bloqueados, os processos são movidos para o *wait set* e quando desbloqueados devolvidos ao *entry set* (colocados no estado *ready*), onde a sua execução é retomada (estado *running* - a partir de onde parou).



Por exemplo no problema do produtor-consumidor, se o produtor chamar o método *Thread.yield()* por causa do *buffer* estar cheio, quando o consumidor tentar aceder ao monitor não vai conseguir, porque este ainda está atribuido ao produtor. Gera-se uma incompatibilidade infinita no tempo, semelhante ao *deadlock*, mas neste caso em vez de termos duas *threads* a esperarem uma pela outra, temos um processo a tentar uma ação continuamente que falha.

A alternativa é a utilização dos métodos *wait()* e *notify()*.

```
public synchronized void insert(Object item) {
    while (count == BUFFER_SIZE) {
       try {
          wait();
       }
       catch (InterruptedException e) { }
    }

    ++count;
    buffer[in] = item;
    in = (in + 1) % BUFFER_SIZE;

    notify();
}
```

```
public synchronized Object remove() {
    Object item;

    while (count == 0) {
       try {
          wait();
       }
       catch (InterruptedException e) { }
    }

    --count;
    item = buffer[out];
    out = (out + 1) % BUFFER_SIZE;

    notify();

    return item;
}
```

---

13 Expressão inglesa para “dar passagem”

É ainda possível sincronizar apenas partes internas de um método.

```
Object mutexLock = new Object();
. . .
public void someMethod() {
    nonCriticalSection();

    synchronized(mutexLock) {
      criticalSection();
    }

    remainderSection();
}
```

Qual a diferença entre monitores e semáforos?

Embora um semáforo possa ser utilizado para implementar um monitor, este é um objeto de baixo nível.

Uma analogia ao mundo real de um monitor é uma cabine de uma casa de banho pública, onde só pode entrar uma pessoa de cada vez. O utilizador entra, tranca a porta para não entrar mais ninguém e apenas quando termina o que foi lá fazer é que a destranca e sai.

Já um semáforo é como uma loja de aluguer de bicicletas, que tem um determinado número de bicicletas para alugar enquanto houverem bicicletas podem ser alugadas. Se não houverem, quem quiser alugar tem de esperar que alguém devolva a que alugou. É ainda importante referir que esta loja não se importa com quem devolve a bicicileta ao final do dia, desde que seja devolvida, esta pode ser passada entre várias pessoas, tendo apenas uma feito o aluguer.

Fonte: https://stackoverflow.com/a/7336516/10735382

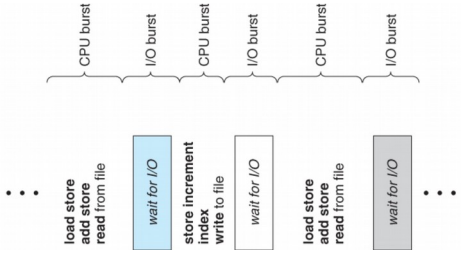
---

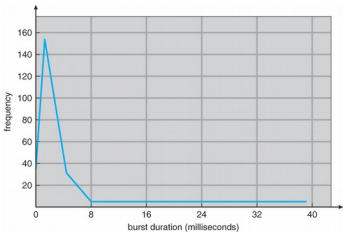
8. Escalonamento do CPU

Operating System Concepts, página 167

Para que a multiprogramação seja possível, é fundamental haver um escalonamento dos processos que vão ser executados, de entre os que se encontram no estado *ready*.

O acesso à CPU caracteriza-se por ser cíclico e na maioria dos casos por um curto acesso intenso (*burst*) seguido de um acesso com menos frequência ao longo de algum tempo.





A seleção do processo a executar é feito pelo escalonamento a curto prazo, podendo o método de seleção variar (FIFO, LIFO…), no entanto, é comum a todos que os processos sejam armazenados como PCB.

A defição dos PCB é feita no capítulo 5
O escalonamento do CPU a médio/longo prazo é descrito no capítulo 5

O escalonamento é invocado cada vez que um proceso muda de estado.

Quando só é invocado quando são feitas as transições *running>waiting* e *running>terminated*, diz-se que o escalonamento é **cooperativo** ou **não preemptivo**[14].

Quando para além destas, é invocado nas transições *running>ready* e *waiting>ready*, diz-se **preemptivo**.

O escalonamento cooperativo necessita de menos recursos, pois nos cenários em que o escalonamento ocorre não há nada a fazer, senão selecionar um novo processo da fila para execução. No preemptivo já não é assim, sendo fundamental a sua associação a um *timer*.

Há ainda algumas questões que se levantam sobre o escalonamento preemptivo, nomeadamente o acesso a regiões críticas interrompido que pode levar a que outro processo aceda a dados corrompidos. Para resolver este problema foi criada a sincronização de processos, abordada no capítulo anterior.

14 Que prevê ou antecipa uma situação (ex.: ataque preemptivo). = PREVENTIVO

Dispatcher

Operating System Concepts, página 171

O componente responsável por colocar o processo selecionado pelo escalonamento em execução chama-se ***dispatcher*** e é responsável pela mudança de contexto, pela alteração do CPU para modo de utilizador e o salto para a instrução do programa que permite continuar a sua execução.

O tempo que o *dispatcher* demora a trocar processos no CPU designa-se por ***dispatch latency***.

Critérios de escalonamento

Operating System Concepts, página 171

Existem vários algoritmos de escalonamento que implementam diferentes formas de o fazer, sendo as propriedades que os permitem distinguir a utilização do CPU, o débito (programas que terminam por unidade de tempo), o tempo de execução do processo (*turnaround time*), o tempo de espera (no estado *ready*) e por fim o tempo de resposta (a pedidos).

O tempo de resposta é fundamental na avaliação do escalonamento de sistemas interativos (por exemplo a *bash*).

Algoritmos de escalonamento

Operating System Concepts, página 172

FCFS (First-Come, First-Served)

Consiste numa filosofia FIFO e é **nonpreensive**. O tempo médio de espera tem tendência a ser irregular e muitas vezes elevado.

Quando o primeiro processo é *cpu-bounded* (requer algum tempo na CPU) os processos restantes têm de esperar muito. No entanto, se este processo for o último e os restantes sejam rápidos a executar, todos irão esperar pouco e o tempo médio de espera será muito baixo.

Por este motivo este algoritmo é um pouco imprevisível e facilmente deixa o CPU sem trabalho (*idle*).

Isto acontece por exemplo quando o primeiro processo é *cpu-bounded* e os restantes são processos de I/O. O primeiro faz os outros esperarem muito tempo, acabando depois, por exemplo, deixando os outros executarem no CPU, passando rapidamente para o estado *waiting* enquanto esperam pela resposta do dispositivo de I/O, deixando o CPU sem nada para fazer.

Componente prática sobre este tema desenvolvida no final deste documento

SJF (Shortest-Job-First)

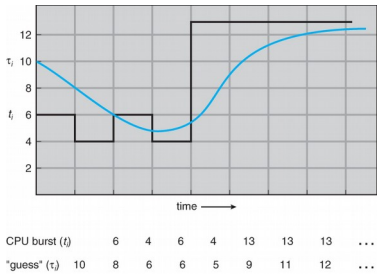
A filosofia agora é *priority queueing*, tendo prioridade os processos que requerem menos tempo de execução na CPU. Em caso de empate aplica-se o algoritmo FCFS. É **nonpreensive**.

A avaliação dos recursos que cada trabalho em espera vai requerer não é fácil, sendo por isso este algoritmo mais comum no planeamento a longo prazo.

Existe no entanto uma variação que está assente em métodos probabilísticos, que analisam o histórico do processo para antever a sua execução futura.

$$\tau_{n+1} = \alpha\, t_n + (1 - \alpha)\tau_n.$$

t_n é o tempo do último CPU *Burst*, τ é a estimativa do CPU burst



SRTF (Shortest Remaining Time First)

Este algoritmo é uma pequena variação do SJF, mas **preensive**, uma vez que se um processo entrar na fila (mudar para o estado *ready*) e o seu tempo de execução for menor do que o tempo restante do trabalho em execução, este é suspenso e susbtituído pelo mais curto.

Escalonamento por prioridade

Neste algoritmo é atribuída uma prioridade a cada processo, sendo primeiro executados os mais prioritários. Podem ser **preensive** ou **non preensive**.

O SJF é um caso particular deste algoritmo.

Para evitar *starvation* dos processos menos prioritários, a prioridade pode ser aumentada proporcionalmente ao tempo de espera.

RR (Round Robin)

Este algoritmo tem por base o conceito de *timesharing*, estabelecendo um limite de tempo que cada trabalho pode ocupar o CPU (*time quantum*) e caso não bloqueie antes do prazo, é retirado da execução e colocado no final da lista de espera dos trabalhos em estado *ready*, que é tratada como uma fila circular.

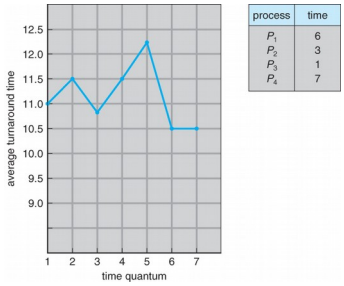
Um processo só é alocado ao CPU em mais do que um intervalo de tempo seguido se não existirem processos no estado *ready* em espera.

Se o *time quantum* for elevado, este algoritmo torna-se semelhante ao FCFS. No entanto, se for reduzido, começa a aplicar o *timesharing* e cada processo tem a ilusão de uma velocidade de execução de 1/n da velocidade do processador (para n processos *ready*).

Nenhum processo espera mais do que (1-n)*q unidades de tempo.

É necessário haver alguma ponderação na definição do *time quantum*, uma vez que a cada alteração de processo no CPU requer tempo. Deve assim ser assumido um compromisso para que haja tempo suficiente para os trabalhos mais rápidos possam ser executados num *quantum* e ao mesmo tempo não exista um tempo de espera muito elevado.

O gráfico ao lado mostra a relação entre o tempo médio de execução dos trabalhos e o *quantum*.



process	time
P_1	6
P_2	3
P_3	1
P_4	7

FIFO Multilevel

Esta abordagem consiste em dividir a fila dos processos *ready* em duas, uma para os processos em *foreground* (interativos) e outra para os de *background* (*batch*).

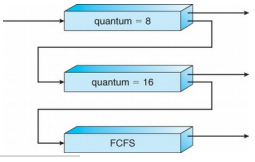
O escalonamento pode ser baseado em prioridade estrita, sendo a máxima dada aos processos em *foreground*, podendo os de *background* executar apenas quando a primeira fila está vazia, ou, para evitar *starvation* dos últimos, pode ser feito com base em *time slicing*, atribuindo à fila prioritária mais tempo do CPU, mas não evitando que a última execute em algum intervalo.

Em cada fila é também importante definir um escalonamento, sendo o mais comum para os processos em *foreground* o RR e para os de *background* o FCFS.

FIFO Multilevel com alimentação

Tal como o anterior cria várias filas que oferecem diferentes prioridades na execução dos processos. É no entanto mais genérico, permitindo para além de ter um número variável de filas e um algoritmo de escalonamento em cada uma, de definir algoritmos para atribuir a prioridade inicial de um processo e elevar ou descer a prioridade dos mesmos ao longo da sua execução.

Um exemplo é a criação de três filas, uma com um *time quantum* de 8ms, outra com 16ms e a última com um escalonamento FCFS. Estas foram descritas por ordem decrescente de prioridade. A segunda só pode ser executada quando a primeira está vazia e a terceira quando as duas anteriores o estiverem também.



Um novo processo começa na fila dos 8ms, se esgotar este intervalo de tempo antes de terminar a sua execução, passa para a fila doa 16ms e se o mesmo ocorrer para para a última, a do FCFS.

Escalonamento Linux

Operating System Concepts, página 193

Em Linux são implementadas três classes de *scheduling*, que ordenadas por ordem decrescente de prioridade são:

SCHED_FIFO

Neste classe o mais prioritário é executado e só é interrompido se for atribuído um mais prioritário do que este à classe.

SCHED_RR

A atribuição do processador a estes processos está condicionada a uma janela de execução, sendo substituídos no CPU pelos processos com maior prioridade da mesma classe ou por qualquer processo pronto a executar da classe *FIFO*.

SCHED_OTHER

Os restantes processos funcionam de acordo com o melhor esforço, sendo executados apenas quando não há processos das classes anteriores prontos a serem executados.

Os processos do utilizador pertencem à classe *Other*, sendo as *FIFO* e *RR* atribuídas ao processamento em tempo real e aos processos do sistema.

Recentemente foram criadas novas classes:

SCHED_DEADLINE

Para *threads* em tempo real, cada uma com um período, uma *deadline* e um tempo de computação, escalonadas de acordo com o algoritmo ***Global Earliest Deadline First***.

SCHED_BATCH

Para processos *cpu-bound*, usa *time slices* maiores.

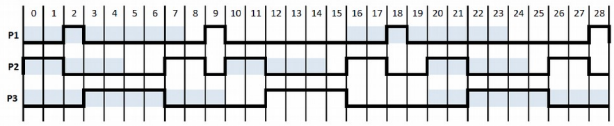
SCHED_IDLE

Para processos de prioridade muito reduzida.

Earliest Deadlien First

Neste algoritmo os processos são escalonados dando prioridade ao processo com a *deadline* mais próxima, permitindo correr um conjunto de processos de forma a todos cumprirem as *deadlines*.

Process	Execution Time	Period
P1	1	8
P2	2	5
P3	4	10



Escalonamento dos processos de utilizador

Como vimos anteriormente os processos de utilizador são processados na classe *SCHED_OTHER*.

Algoritmo tradicional

Antigamente o seu escalonamento era feito de acordo com a prioridade de cada processo, sendo este calculado a cada recreditação da seguinte maneira:

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

A proridade do processo j no instante i é igual à soma de metade dos créditos não utilizados no instante anterior com a prioridade base desse processo e o valor da alteração da prioridade dependente do utilizador, *nice* (varia entre -20 e 19).

A cada interrupção do RTC (*real time clock*) o processo perde um crédito, perdendo a posse do CPU quando tiver 0 créditos. Quando a fila de espera de processo *ready* com créditos fica vazia, é feita uma recreditação a todos os processos da classe.

Este algoritmo combina assim a história de execução e a prioridade.

No entanto é pouco escalável – O(n) –, dando prioridade aos processos de I/O, com um *timeslice* elevado e comportamento *real time* (não *preemptive*).

Novo algoritmo (Justiça Total)

Para tornar o escalonamento mais justo foi desenvolvido um novo algoritmo para a classse *SHCED_OTHER*, que assume um processador ideal, que é capaz de executar

todos os processos em paralelo, distribuindo a potência de cálculo por todos de modo uniforme (velocidade de processamento 1/N).

Como o processador é “imaginário”, é preciso transpor a ideia para o processador real. Define-se assim o tempo de execução virtual, o ideal e o tempo de espera.

Os processos são depois organizados numa fila, sendo dada prioridade aos processos que têm uma maior diferença entre o tempo de execução virtual e o de espera.

Sempre que um processo é calandarizado, o seu tempo de espera é decrementado no valor da sua janela de execução, tempo que é incrementado aos tempos de espera de todos os processos que permanecem na fila, que têm também incrementado o tempo de execução virtual neste tempo (hipotético, caso fossem executados). Os processos bloqueados mantém os valores inalterados.

O elemento central deste algoritmo é a história da execução do processo, não distinguindo os processos I/O ou CPU intensivos.

9. Memória virtual

Operating System Concepts, página 347

Muitas vezes a execução de programas mais pesados e/ou de vários em simultâneo (*multiprogramming*) cria algumas limitações do ponto de vista físico, se tivermos de os ter todos carregados em memória RAM, tendo em conta o seu tamanho, que normalmente é reduzido face à memória do disco, dado o seu custo elevado.

Criou-se assim a necessidade de abstração das limitações físicas da memória, permitindo a execução de programas que estão apenas parcialmente carregados em memória. Isto é possível porque raramente ao utilizarmos um programa estamos a explorar todas as suas funcionalidades.

A memória virtual oferece assim **eficiência na utilização da memória**, que é partilhada pelos processos, utilizando apenas o necessário e utilizando endereços de memória virtuais; **segurança**, pois mesmo com a partilha da memória, um processo não pode alterar a memória de outro e **transparência** porque o processo tem acesso a muita memória.

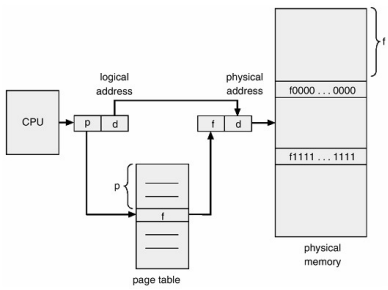
É assim fundamental criar um espaço de endereçamento virtual, que é utilizado pelos programas, oferecendo ao programador memória quase "ilimitada" em comparação com a disponível efetivamente. O mesmo endereço virtual de dois processos distintos pode corresponder a endereços físicos distintos e alguns endereços virtuais podem ser armazenados em disco.

Paginação da memória

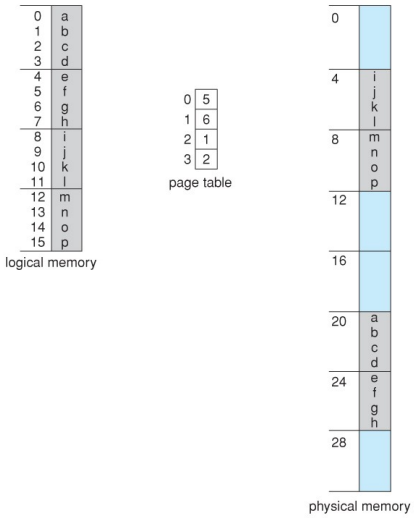
Operating System Concepts, página 320

Antes da sua implementação, a maioria dos gestores de memória tentava guardar espaço contíguo em memória suficiente para cad programa, sendo esta tarefa bastante complexa e por vezes imprevisível. A paginação da memória permitiu resolver este problema, permitindo a dispersão do programa em memória.

Para o fazer, cada programa é dividido em blocos de uma determinada dimensão (fixa), as páginas, que são armazenadas em memória não necessáriamente de forma sequencial. Para lhes aceder, é utilizada uma ***page table***, que para cada processo relaciona o número das suas páginas com a sua posição na memória.

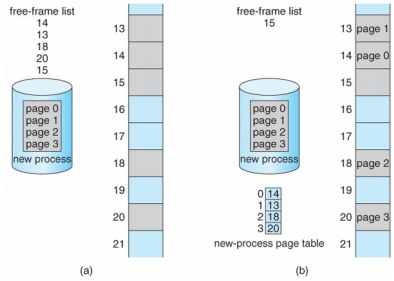


Cada endereço gerado no CPU é assim composto pelo *page number* (p) e pelo *page offset* (d). Sendo o primeiro o índice da página na *page table* e o segundo o deslocamento dentro da mesma. Abaixo podemos ver um exemplo da paginação com 4 *bytes* numa memória de 32 *bytes*. Este mapeamento é feito pela MMU (*memory management unit*).



O endereço lógico 3 é paginado na página 0 (índice 5 pela *page table*) com *offset* 3, logo em memória física será 4*5+3=23.

A alocação dos *frames* de memória livres aos processos é feita pelo SO, que tem conhecimento da topologia da memória física, mantendo a informação dos livres numa estrutura denominada *free-frame list*.



Para além desta informação, o SO mantém uma cópia da *page table* de todos os processos, para que em caso de pedidos de I/O, p.e., este possa fazer a tradução para o endereço físico correspondente.

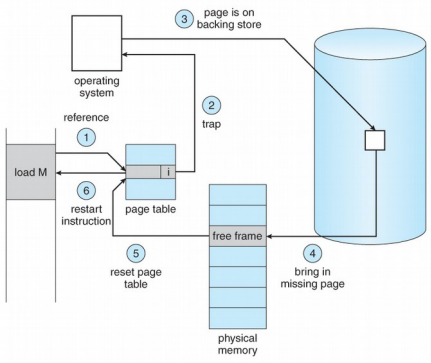
Pagination fault

Operating System Concepts, página 353

Como vimos na introdução deste capítulo, raramente utilizamos todas as funcionalidades de um programa em simultâneo, razão pela qual implementamos a memória virtual e carregamos apenas parte das páginas para cada processo.

Antes de ser executado, cada processo é avaliado por um *pager*, que prevê as páginas que vão ser utilizadas durante a mesma e carrega-las em memória. Na *page table*, é acrescentado um bit *valid*, que nas páginas carregadas em memória está a 1 e nas restantes a 0.

No entanto, pode haver necessidade de aceder a uma página que não foi carregada. A isto chamamos **page-fault trap** e acontece quando se tenta aceder a uma página na tabela cujo bit *valid* está a 0 (i), sendo necessário carregá-la do disco.



Page Replacement

Operating System Concepts, página 368

Por vezes, um *page fault* não vem só e traz consigo outro problema: a inexistência de *frames* livres. Nestes casos é necessário libertar algum *frame* que não seja importante, uma vez que o carregamento em memória da página é fundamental para a execução do processo que gerou o primeiro erro.

Um dos mecanismos de substituição é o **LRU (least-recently-used)**, substituindo em caso de necessidade a página que foi utilizada pela última vez há mais tempo.

No entanto, nem todos os sistemas o suportam, mas sima a uma implementação alternativa, o **LRU-aproximation**, que consiste em ter um conjunto de bits de referência, que são periodicamente colocados a zero (feito *shift right*), sendo estes mudados para 1 caso a página seja acedida.

Por exemplo se utilizarmos 8 bits para cada processo, começam todos a 00000000. Se um é acedido muda para 10000000, que no período seguinte será 01000000 (10000000>>1). Um processo com indicativo 11000100 foi utilizado mais recentemente que um a 01110111.

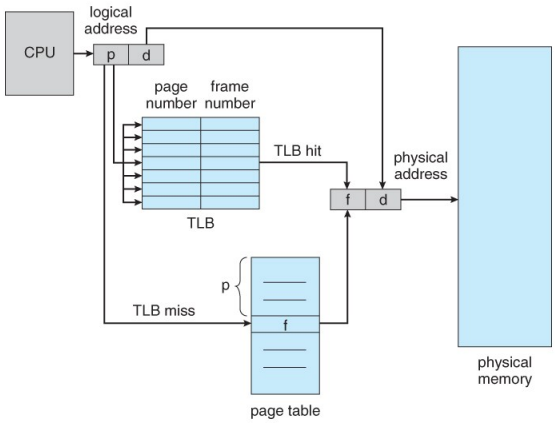
Existem outras formas de implementar esta aproximação ao LRU.

TLB (Translation Look-aside Buffer)

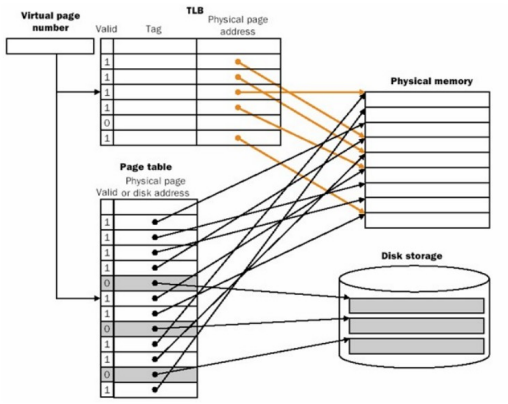
Operating System Concepts, página 324

Os computadores modernos permitem aos processos que tenham *page tables* muito grandes, o que cria um problema. Apesar de evitar o carregamento total do programa, o carregamento necessário da *page table* torna-se agora impossível e neste cenário o processo vai demorar imenso tempo a executar, tendo em conta as operações de acesso ao disco constantes.

Foi por isto criado o TLB, que funciona como uma *cache* das entradas da *page table,* guardando as últimas páginas acedidas. Quando cheia, as suas entradas podem ser substituídas de forma aleatória ou de acordo com o algoritmo LRU.



Este *buffer*, também implementa a memória virtual.



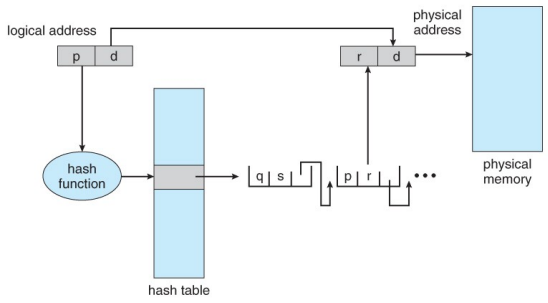
Tamanho das tabelas de página

Operating System Concepts, página 329

Como vimos, as tabelas de página de cada processo podem ser imensas. Como soluções, podemos limitar o tamanho da tabela, utilizar *hashing*, tabelas com vários níveis ou até recorrer à memória virtual :)

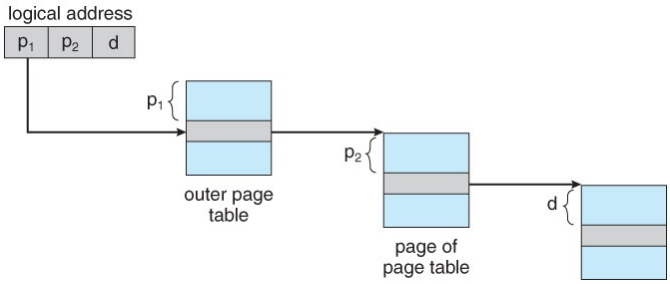
Tabelas de página com hashing

Segundo este processo, cada endereço virtual é mapeado por uma *hash function* num índice de uma *hash table*, onde estará uma lista ligada, com os vários endereços virtuais que foram mapeados nesse endereço. É feita uma análise nó a nó, até encontrar um nó cujo primeiro elemento corresponda ao endereço virtual (o segundo será o endereço real), sendo o segundo devolvido quando for encontrada a correspondência.



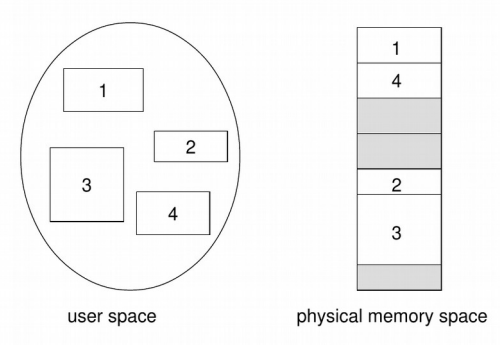
Tabelas de página com vários níveis

Outra abordagem consiste em paginar a tabela de páginas.



Segmentação da memória

A alternativa à paginação da memória é a segmentação, que consiste em dividir a memória em segmentos, de tamanho variável, sendo necessário para cada processo uma zona contígua de memória livre para cada segmento.



Segmentação interna vs. externa

Como vimos, há duas formas de segmentar a memória. A segmentação **interna**, ocorre quando a memória é dividida em blocos de tamanho fixo e a **externa** quando estes são dinâmicos, tendo em conta o tamanho necessário ao processo.

10. Em prática

Neste capítulo serão descritos algumas interações com a bash que permitem pôr em prática os conceitos teóricos abordados nos capítulos anteriores. Todos os temas que aqui constem têm no final do seu conteúdo a seguinte referência:

> Componente prática sobre este tema desenvolvida no final deste documento

2. Multiprogramação

```
$ cat /proc/interrupts
```

Configuração de interrupções

2. Gestão de processos

```
$ ps
```

Lista dos processos a correr atualmente no SO

```
$ top
```

Processos Linux

```
$ htop
```

Mesmo que anterior, mas de forma interativa

3. Modos de operação

```
$ passwd -S -a
```

Base de dados dos utilizadores no PC

```
$ group
```

Base de dados de grupos (armazenada em */etc/group*)

```
$ shadow / gshadow
```

Base de dados de palavras-passe encriptadas

3. Arquiteturas do SO

```
$ lsmod
```

Estado dos módulos no núcleo do *Linux*

```
$ modinfo <modName>
```

Informação acerca do módulo passado como argumento

```
$ modprobe <modName>
```

Adiciona módulo ao núcleo

```
$ rmmod <modName>
```

Remove módulo do núcleo

3. Sistemas de ficheiros

```
$ find (-iname <regexExpression>)
```

Lista de forma recursiva todos os diretórios e conteúdos a partir do diretório atual, podendo filtrar pelo nome do ficheiro através da opção *-iname*.

5. Processos

```
$ ls -l /proc
```

O diretório proc é um diretório virtual e contém informações sobre o sistema e os seus processos, tendo em vez de ficheiros, na sua forma informações em tempo de execução.

O comando lsmod visto na secção anterior é nada mais nada menos que $ cat /proc/modules
Outros ficheiros cuja consulta pode ser útil são cpuinfo, meminfo...

Dentro deste, existe um diretório para cada processo denominado do seu id, havendo dentro de cada uma ficheiros que apontam para onde a informação do processo está armazenada.

+ info http://www.tldp.org/LDP/Linux-Filesystem-Hierarchy/html/proc.html

```
$ <progName> &
```

Escrever o caracter & no final de um comando faz com que este seja executado no *background* da *shell* onde o comando foi executado.

+ info https://unix.stackexchange.com/questions/3886/difference-between-nohup-disown-and

```
$ fg <pid>
```

“Puxa” processo em *background* para primeiro plano (*foreground*)

```
$ jobs
```

Estados dos trabalhos em execução na sessão atual da *shell*.

5. Criação de processos

```
$ ps (-el)
```

Permite consultar os processos em execução. Com o *-el* é mostrada informação detalhada para todos os processos ativos no sistema.

```
$ kill <pid>
```

Termina o processo.

8. Escalonamento de prioridade dos processos

```
$ nice -20 program
```

Manipulação da prioridade de um processo.

Extra. Login remoto

O acesso remoto a um computador pode ser feito de forma segura através do protocolo SSH (*Secure Shell*), que pode ser configurado em *Linux* com os seguintes comandos:

```
$ ssh-keygen
$ ssh-copy-id
```

Extra. Redes

```
$ ifconfig / ip addr
```

Informações da interface de rede

```
$ ping <ip>
```

Verificar ligação da rede

```
$ route / ip route
```

Tabelas de encaminhamento

```
$ traceroute <ip>
```

Traça percurso dos pacotes até ao destino (*router hops*)

```
$ nmap -sn 192.168.1.*
```

Mostra mapa da rede (dispositivos conectados)