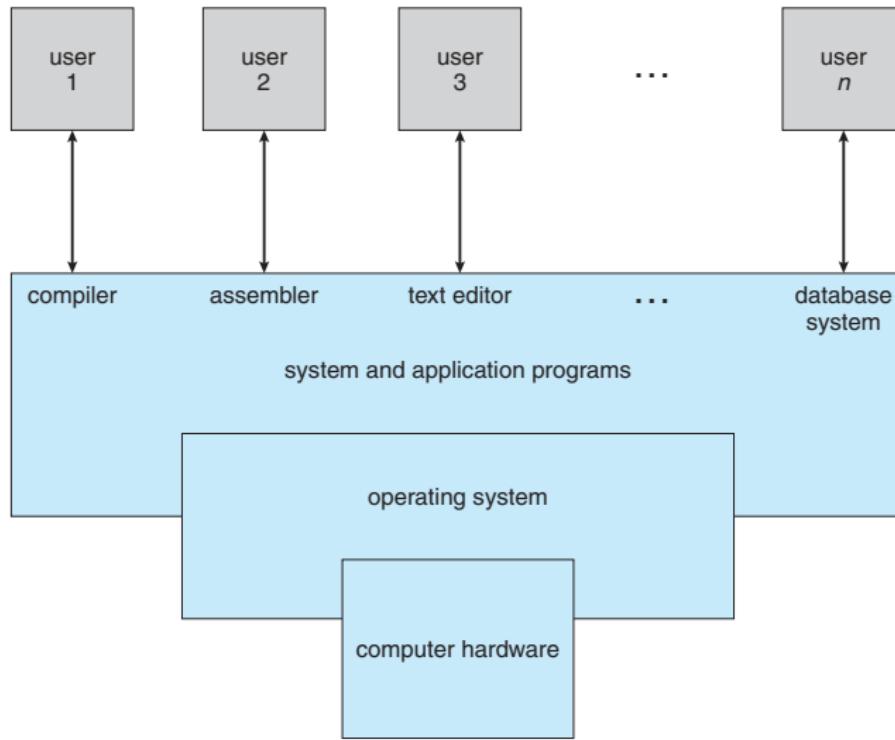


Unidade 1 - Introdução

- 1 Estrutura do Sistema de Computação
- 2 Organização e Operação do Hardware
- 3 Conceito de Sistema Operativo
- 4 Componentes do Sistema Operativo
- 5 Programas do Sistema e Aplicações
- 6 Proteção do Hardware
- 7 Estrutura do Sistema Operativo
- 8 Desenvolvimento do Sistema Operativo
- 9 Depuração e Monitorização do Sistema
- 10 Slides Suplementares

1.1 Estrutura do Sistema de Computação

Estrutura do Sistema de Computação (1/2)



Estrutura do Sistema de Computação (2/2)

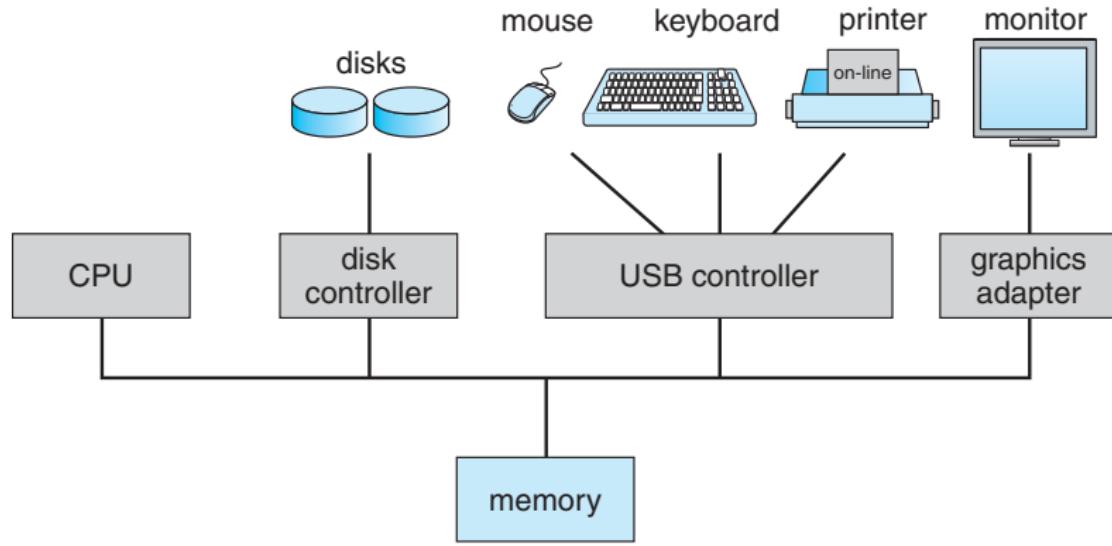
- ① Hardware - fornece os recursos de computação básicos (CPU, memória, dispositivos de Entrada/Saída (E/S))
- ② Sistema Operativo - controla e coordena a utilização do hardware pelos diversos programas
- ③ Aplicações - definem formas através das quais os recursos do sistema são usados para resolver os problemas dos utilizadores (compiladores, bases de dados, editores de texto, folhas de cálculo, browsers, etc.)
- ④ Utilizadores (pessoas, máquinas, outros computadores)

1.2 Organização e Operação do Hardware

Organização e Operação do Hardware (1/12)

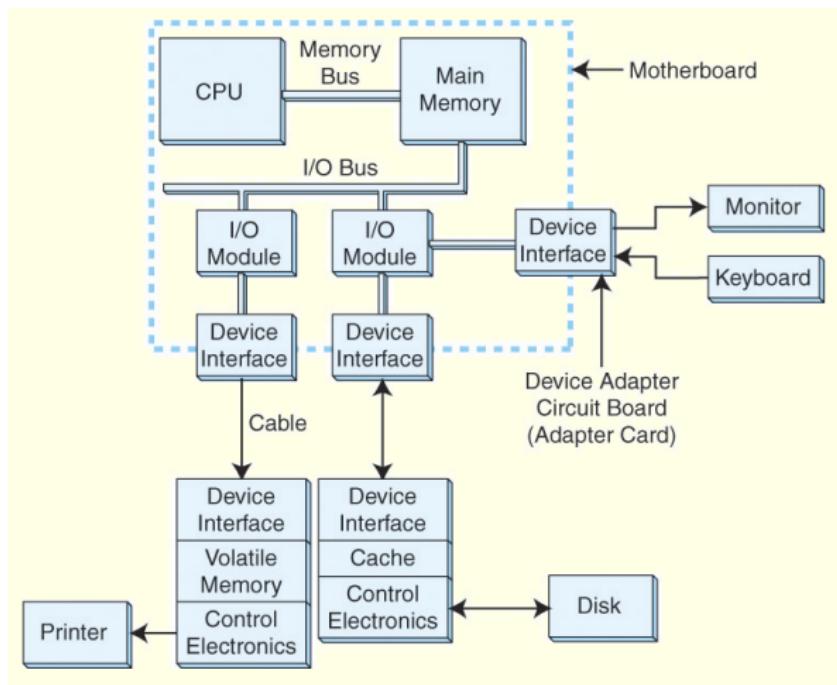
Visão Simplificada

- CPU, memória, controladores de dispositivos, e dispositivos, interconectados (a representação abaixo, com 1 só barramento, partilhado, é simplificada)
- CPU e dispositivos competem (operação concorrente) pelo acesso à memória



Organização e Operação do Hardware (2/12)

Visão Realista



Gestão dos Periféricos

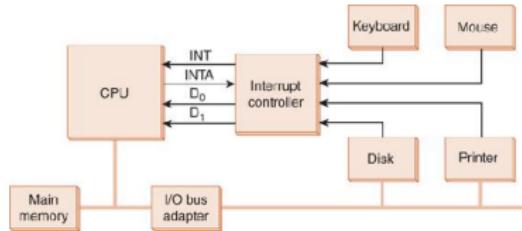
- CPU(s), dispositivos E/S e RAM, interligados por barramentos
- os dispositivos E/S são geridos por controladores específicos
 - um controlador pode controlar vários dispositivos (e.g., SATA, USB)
 - um controlador dispõe de memória local e de um microprocessador
 - um controlador move dados entre a memória local e o(s) dispositivo(s)
- a CPU e os controladores E/S trabalham em simultâneo (nalguns casos - DMA - competindo por ciclos de acesso à memória RAM)
- controlo da atividade E/S: polling, interrupções, DMA
- *device driver*: módulo do SO que interage com um dispositivo

Interrupções

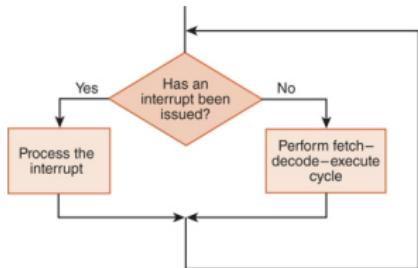
- as interrupções ocorrem de forma **assíncrona** (a qualquer momento)
- internas (CPU), do HW (periféricos), ou de SW (*system calls*)
- assinalam um evento que carece de tratamento pelo sistema operativo
- interrupção ⇒ a CPU suspende o processamento em curso, executa uma *rotina de serviço*, após o que retoma o processamento anterior
- como determinar que rotina de serviço executar ?
 - *indirectamente*, através de uma rotina genérica (*switch*); lento
 - *diretamente*, através de um *vetor de interrupções*; muito rápido
 - pressupõe um id único (número) por cada interrupção diferente
 - o id único da interrupção, i , é um índice do vetor de interrupções
 - a célula i do vetor aponta para a rotina de serviço da interrupção i
- antes da rotina executar, é feita uma cópia do PC; a própria rotina salvaguarda outros registos que irá modificar, recuperando o seu valor no fim da execução, após o que o PC original é também recuperado

Organização e Operação do Hardware (5/12)

Interrupções (cont.)

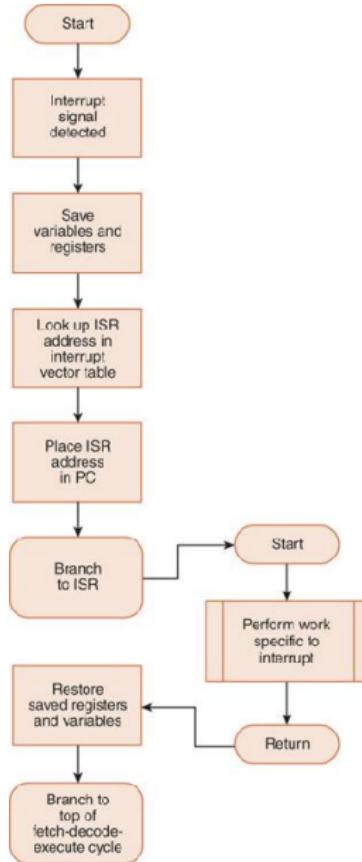


Geração e Identificação



Detecção

Atendimento →

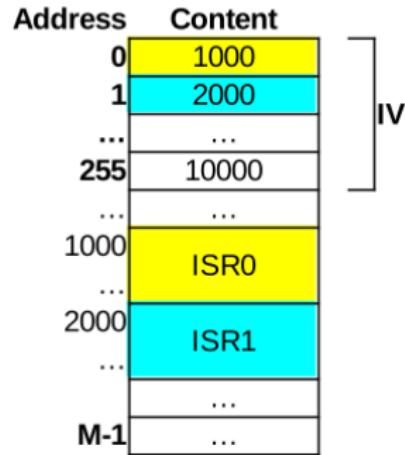


Organização e Operação do Hardware (6/12)

Interrupções (cont.) - vetor de interrupções na arquitetura x86:

vector number	description
0	divide error
1	debug exception
2	null interrupt
3	breakpoint
4	INTO-detected overflow
5	bound range exception
6	invalid opcode
7	device not available
8	double fault
9	coprocessor segment overrun (reserved)
10	invalid task state segment
11	segment not present
12	stack fault
13	general protection
14	page fault
15	(Intel reserved, do not use)
16	floating-point error
17	alignment check
18	machine check
19–31	(Intel reserved, do not use)
32–255	maskable interrupts

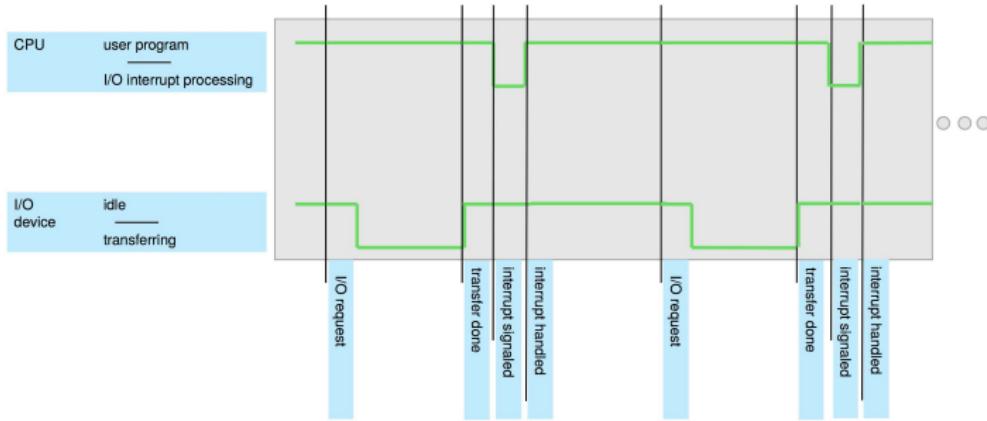
Main Memory (0 .. M-1)



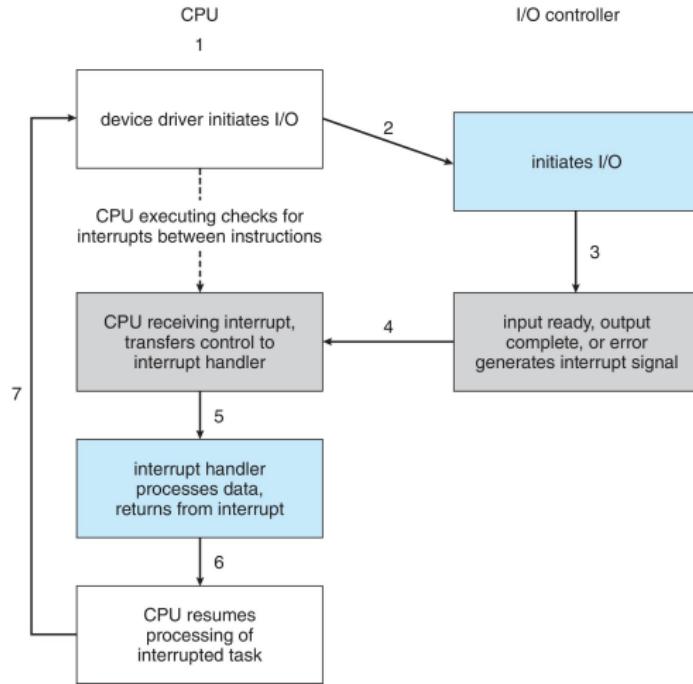
Organização e Operação do Hardware (7/12)

E/S controlada por Interrupções

- um programa pede ao SO uma operação E/S associada a um dispositivo
- um *device driver* programa o microprocessador do controlador do dispositivo
- o controlador despoleta a transferência de dados de/para a memória local
- o controlador avisa o *device driver* da conclusão da operação via **interrupção**
- o *device driver* transfere dados do dispositivo para a RAM (pedido foi de leitura), ou fica apenas a saber da conclusão do pedido (se foi de escrita)



E/S controlada por Interrupções

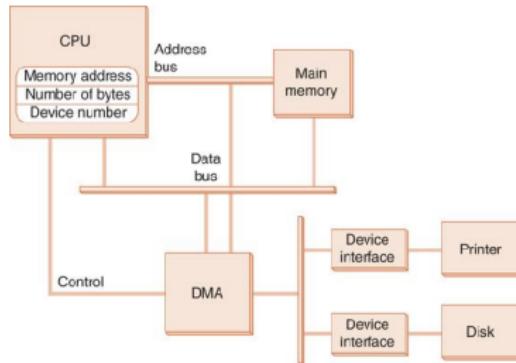


Acesso Direto à Memória (DMA)

- alternativa à *E/S guiada por interrupções*, para minimizar as interrupções
- usada por dispositivos E/S que movimentam grandes quantidades de dados
- o controlador do dispositivo E/S transfere blocos de dados entre a sua memória local e a RAM, com intervenção mínima da CPU
 - um processo cliente solicita ao SO uma transferência de dados
 - o SO programa os registos do controlador de DMA com a origem, destino e dimensão dos dados, e pede-lhe para iniciar a transferência
 - entretanto, a CPU prossegue a execução de outros processos, competindo com o controlador de DMA no acesso à memória
 - finalizada a transferência, o controlador de DMA notifica a CPU

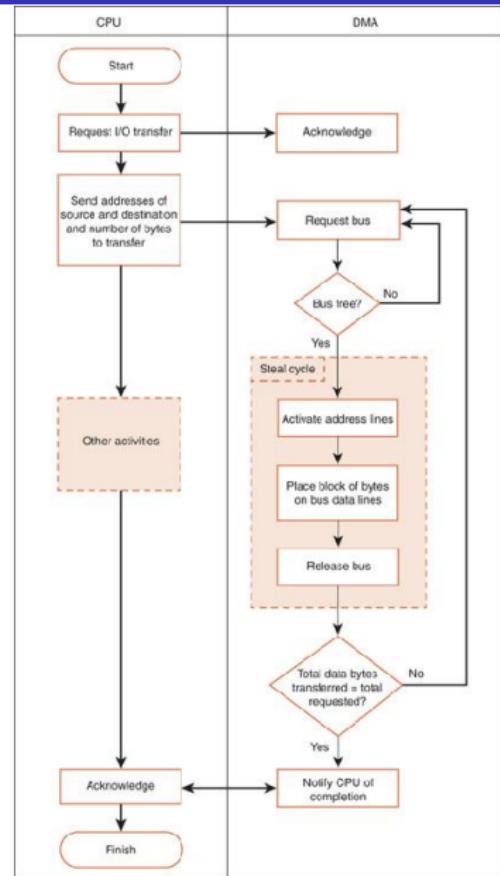
Organização e Operação do Hardware (10/12)

Acesso Direto à Memória (DMA)

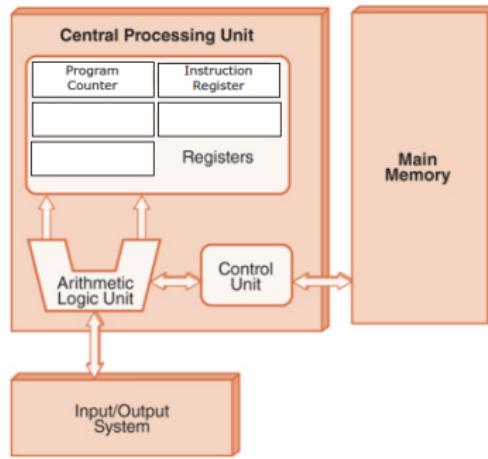


Controlador DMA

Workflow do DMA →

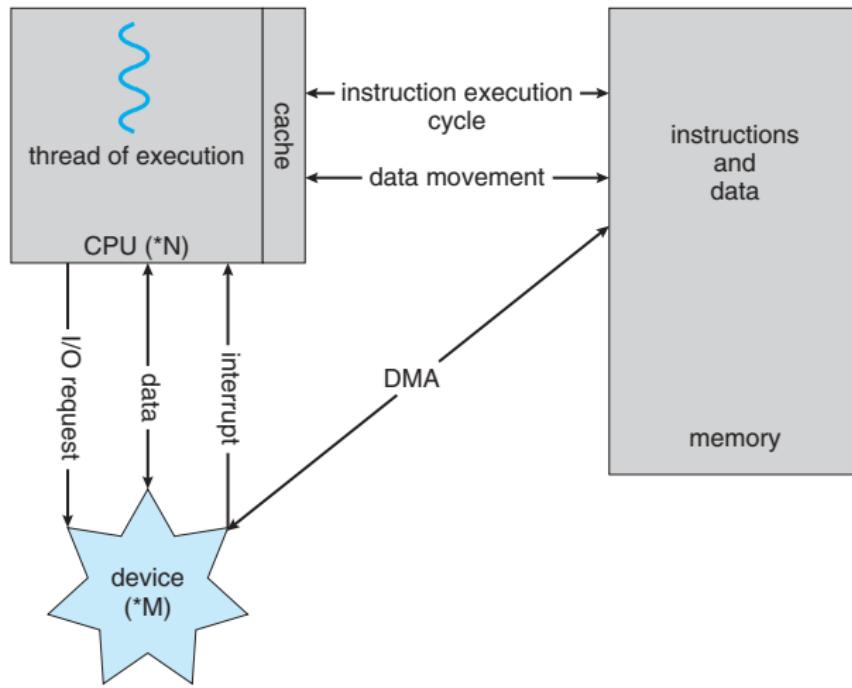


Execução de Instruções



- **modelo de Von Neumann:** programas residem na memória principal, e são executados, instrução após instrução; cada instrução passa por **3 fases**
- **fetch:** a UC traz da memória, para o IR, a instrução apontada pelo PC
- **decode:** a UC descodifica a instrução (podendo ler parâmetros da memória)
- **execute:** a instrução é executada (podendo ler/escrever de/na memória)

Operação do Sistema – Resumo



1.3 Conceito de Sistema Operativo

Conceito de Sistema Operativo (1/2)

• Sistema Operativo

- **intermediário** entre o utilizador e o hardware do computador
- fornece o **ambiente** adequado à execução de programas

• Objetivos de um Sistema Operativo

- executar programas dos utilizadores, ajudando a resolver problemas
- tornar o Sistema de Computação *conveniente* de usar
- utilizar o hardware de forma *eficiente*
- objetivos relativamente antagónicos ...
- objetivos com pesos diferentes consoante o tipo de sistema de computação (servidores, desktops, disp. móveis, sist. embebidos, ...)

Conceito de Sistema Operativo (2/2)

Perspetivas do Sistema Operativo

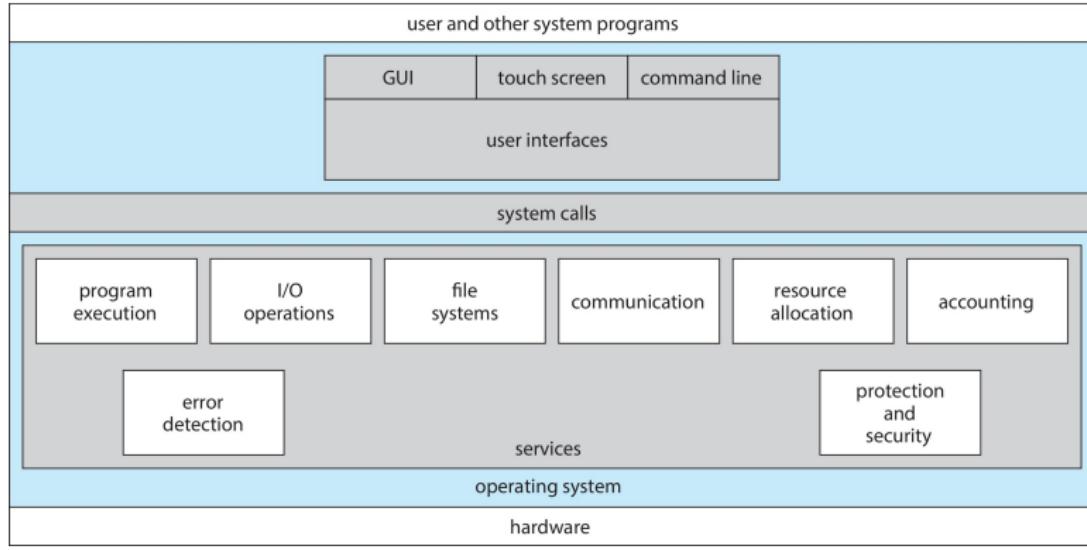
- Perspetiva do **Utilizador**: o SO deve fornecer um **interface conveniente** (GUIs, acesso remoto, partilha de periféricos, ...) e **bom desempenho** na exploração do hardware
- Perspetiva do **Sistema de Computação**:
 - **alocador de recursos** – gere e reserve recursos (conflitos, otimizações)
 - **programa de controlo** – controla a execução dos programas dos utilizadores e a operação dos dispositivos E/S (operações legais, ...)

Constituintes do Sistema Operativo

- a definição do software que constitui o SO não é consensual ...
- Núcleo (*Kernel*) – o único programa em execução “contínua”
- Núcleo + Aplicações – Windows*, Mac OS, Linux, UNIXes, ...
- Núcleo + Middleware (frameworks de desenvolvimento de apps, com suporte a bases de dados, multimédia, gráficos, ...) – iOS, Android, ...

1.4 Componentes do Sistema Operativo

Componentes do Sistema Operativo (1/16)



Serviços orientados à **conveniência** do Utilizador

- ① **Interface com o Utilizador:** i) interface de linha de comando (CLI), incluindo suporte a *batch files (scripts)*, ii) interface gráfico (GUI), iii) ecrã sensível ao toque (*touch screen*), reconhecimento de voz
- ② **Execução de Programas:** carregamento de um programa em memória, sua execução e sua terminação (normal ou anormal)
- ③ **Operações E/S:** leitura/escrita de dados, em periféricos e ficheiros
- ④ **Manuseamento do Sistema de Ficheiros** – criar, remover, escrever, ler, ligar, proteger, caracterizar e procurar ficheiros e diretórios
- ⑤ **Comunicações:** entre processos na mesma máquina, ou em máquinas diferentes, através de *memória partilhada* ou *passagem de mensagens*
- ⑥ **Deteção de Erros:** deteção e tratamento de erros no hardware (CPU, memória, dispositivos E/S) ou nos programas dos utilizadores

Serviços orientados à **eficiência** do Sistema

- ① **Reserva de Recursos**: para vários utilizadores ou jobs concorrentes (CPU, memória, armazenamento, acesso s dispositivos E/S)
- ② **Contabilização (Accounting)**: da espécie e quantidade de recursos consumidos pelos utilizadores, com fins estatísticos e/ou de taxação
- ③ **Proteção e Segurança**: assegura o controlo do acesso aos recursos do sistema (autenticação de utilizadores, validação de acessos E/S)

Interpretador de Comandos (*shell*)

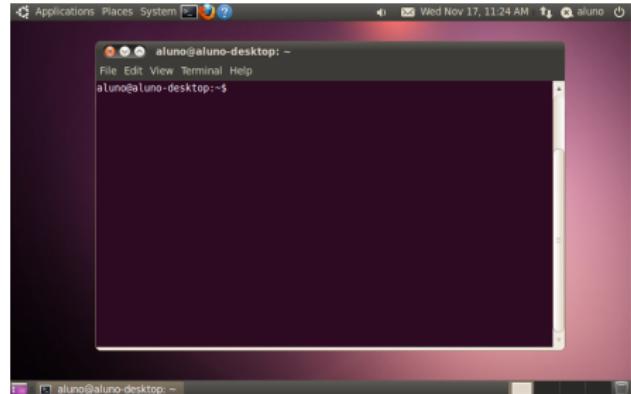
- permite emitir comandos ao SO, através de um interface CLI
- algoritmo simples: 1) ler sequência de comandos e seus argumentos, 2) interpretar sequência, 3) executar comandos, 4) voltar ao passo 1)
- UNIX/Linux:
 - Bourne shell, Bourne-Again Shell (bash), C shell, K shell , ...
 - execução automática após autenticação numa consola de texto
- MS-DOS: command.com; Windows: cmd.exe, powershell
- *shell scripts*: interpretadas pela *shell*, permitem automatizar a execução de sequências de comandos ou executar tarefas mais complexas (e.g., com sintaxe ≈ à das linguagens de programação)
- *comandos internos*: implementados no código do interpretador
- *comandos externos*: invocados a partir do código do interpretador

Componentes do SO - Interfaces de Utilizador (5/16)

Interpretador de Comandos (cont.) – Linux BASH

```
fso@fso-desktop:~$ date  
Sat Oct 8 23:07:03 WEST 2011  
fso@fso-desktop:~$  
fso@fso-desktop:~$ ls -la *.c  
-rw-r--r-- 1 fso fso 76 2011-10-08 23:05 ola_mundo.c  
fso@fso-desktop:~$  
fso@fso-desktop:~$ gcc ola_mundo.c -o ola_mundo.exe  
fso@fso-desktop:~$  
fso@fso-desktop:~$ ola_mundo.exe  
hello world  
fso@fso-desktop:~$ echo $PATH  
/usr/local/sbin:/usr/local/bin:/usr/sbin:/usr/bin:/sbin:/usr/games:  
fso@fso-desktop:~$  
fso@fso-desktop:~$ cp ola_mundo.c ola_mundo.c.bak  
fso@fso-desktop:~$  
fso@fso-desktop:~$ rm ola_mundo.exe  
fso@fso-desktop:~$  
fso@fso-desktop:~$ alias  
alias egrep="egrep --color=auto"  
alias fgrep="fgrep --color=auto"  
alias grep="grep --color=auto"  
alias ls="ls -CF"  
alias la="ls -a"  
alias ll="ls -alF"  
alias ls="ls --color=auto"  
fso@fso-desktop:~$
```

sessão BASH



sessão BASH em GUI

```
#!/bin/bash  
for jpg in "$@" ; do  
    png="${jpg%.jpg}.png"  
    echo converting "$jpg" ...  
    if convert "$jpg" jpg.to.png ; then  
        mv jpg.to.png "$png"  
    else  
        echo 'error: failed output saved in "jpg.to.png".' 1>&2  
        exit 1  
    fi  
done  
echo all conversions successful
```

`# use $jpg in place of each filename given, in turn
find the PNG version of the filename by replacing .jpg with .png
output status info to the user running the script
use the convert program (common in Linux) to create the PNG in a temp file
if it worked, rename the temporary PNG image to the correct name
...otherwise complain and exit from the script`
`# the end of the "if" test construct
the end of the "for" loop
tell the user the good news`

script em BASH

Componentes do SO - Interfaces de Utilizador (6/16)

Interpretador de Comandos (cont.) – MSDOS

```
Starting MS-DOS...  
  
HIMEM is testing extended memory...done.  
C:\>C:\DOS\SMARTDRV.EXE /X  
C:\>command  
  
Microsoft(R) MS-DOS(R) Version 6.22  
 (C)Copyright Microsoft Corp 1981-1994.  
C:\>_
```

sessão MSDOS 6.22

```
@echo off  
prompt $PSG  
PATH=C:\DOS;C:\WINDOWS  
set TEMP=C:\TEMP  
set BLASTER=A220 I7 D1 T2  
goto %CONFIG%  
:WIN  
    lh smartdrv.exe  
    lh mouse.com /Y  
    win  
goto END  
:XMS  
    lh smartdrv.exe  
    lh doskey  
    goto END  
:END
```

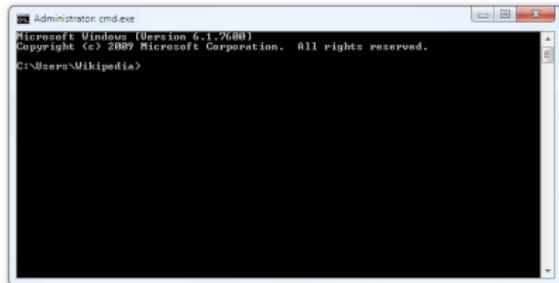
autoexec.bat

```
[menu]  
menuitem=WIN, Windows  
menuitem=XMS, DOS with only Extended Memory  
menudefault=WIN, 10  
[common]  
device=c:\dos\himem.sys  
dos=high,umb  
shell=c:\dos\command.com c:\dos /e:512 /p  
country=44,437,c:\dos\country.sys  
[WIN]  
device=c:\dos\emm386.exe ram  
devicehigh=c:\windows\mouse.sys  
devicehigh=c:\dos\setver.exe  
[XMS]  
device=c:\dos\emm386.exe noems
```

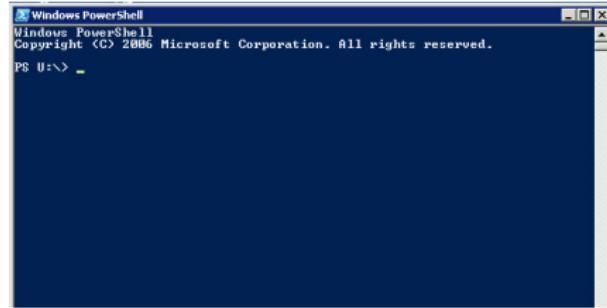
config.sys

Componentes do SO - Interfaces de Utilizador (7/16)

Interpretador de Comandos (cont.) – Windows 7



cmd.exe



powershell

```
PS C:\> Get-ChildItem 'MediaCenter:\Music' -rec |  
>>     where { -not $_.PSIsContainer -and $_.Extension -match '\w+mp3' } |  
>>     Measure-Object -property length -sum -min -max -ave  
>>  
  
Count : 1307  
Average : 5491276.09563087  
Sum : 7177097857  
Maximum : 22995267  
Minimum : 3235  
Property : Length  
  
PS C:\> Get-WmiObject CIM_BIOSElement | select biosu*, man*, ser* | Format-List  
  
BIOSVersion : <TOSCPL = 6040000, Ver 1.00PARTIBL>  
Manufacturer : TOSHIBA  
SerialNumber : M821116H  
  
PS C:\> (IumiSearcherJob'  
>> SELECT * FROM CIM_Job  
>> WHERE Priority > 1  
>> '$@>.get() | Format-Custom  
>>  
class ManagementObject#root\cimv2\Win32_PrintJob  
{  
    Document = Monad Manifesto - Public  
    JobId = 6  
    JobStatus =  
    Owner = User  
    Priority = 12  
    Title = 027088  
    Name = Epson Stylus COLOR 740 ESC/P 2, 6  
>  
  
PS C:\> $rssUrl = 'http://blogs.msdn.com/powershell/rss.aspx'  
PS C:\> $blog = [xml](New-Object System.Net.WebClient).DownloadString($rssUrl)  
PS C:\> $blog.rss.channel.item | select title -first 3  
  
title  
MMS: What's Coming In PowerShell V2  
PowerShell Presence at MMS  
MMS Talk: System Center Foundation Technologies  
  
PS C:\> $host.version.ToString().Insert(0, 'Windows PowerShell: ')  
Windows PowerShell: 1.0.0.0  
PS C:\>
```

exemplo de sessão em powershell

Interface Gráfico (GUI)

- sistema de janelas e menus, operados com rato e teclado
 - desktop, ícones, pastas, menus contextuais, drag-n-drop, seleções
- interface amigável, mas de potencialidades limitadas ...
 - as *shells* de linha de comando são tipicamente mais versáteis
- invenção: década de 70, Laboratório da Xerox, em Palo Alto
- popularização: década de 80, Apple Macintosh + Mac OS
 - hoje em dia, a GUI da linha Mac OS X é a GUI de referência
- UNIX: Common Desktop Environment (CDE), X-Windows
- Linux: LXDE, Xfce, K Desktop Environment (KDE), GNOME
- Windows: 1.x/2.x/3.x, 95/98/Me/NT, 2000, XP, Vista, 7, 8.x, 10
- outro exemplos de GUIs são as adequadas a dispositivos móveis
 - primeiros exemplos: ecrãs limitados, teclados e/ou canetas
 - atual/: ecrãs de alta resolução, toques / gestos, reco. de voz

Componentes do SO - Interfaces de Utilizador (9/16)

Interface Gráfico - Evolução das GUIs (Apple + Microsoft)



Xerox Alto (1973)



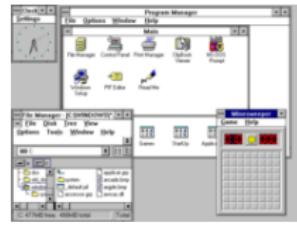
System 1.0 (1984)



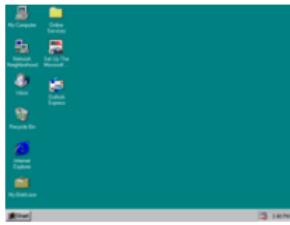
Mac OS 9 (1999)



Mac OS X 10.5 (2007)



Windows 3.11 (1993)



Windows 95 (1995)



Windows 7 (2009)



Windows 10 (2015)

Componentes do SO - Interfaces de Utilizador (10/16)

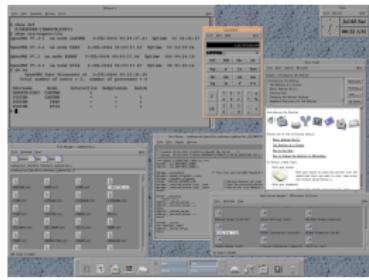
Interface Gráfico - Evolução das GUIs (Unix + Linux)



X Windows (>1984)



NeXTSTEP (1990)



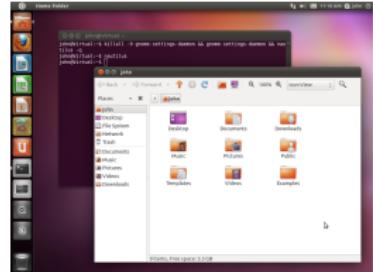
CDE (1995)



KDE 4.3 (2010)



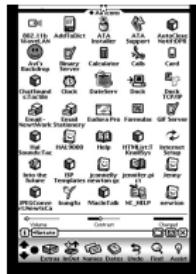
Gnome 3.2 (2011)



Unity (2011)

Componentes do SO - Interfaces de Utilizador (11/16)

Interface Gráfico - Evolução das GUIs (Sistemas Handheld)



Apple Newton OS (1993)

Psion EPOC32 (1997)

Palm OS 5 (2002)

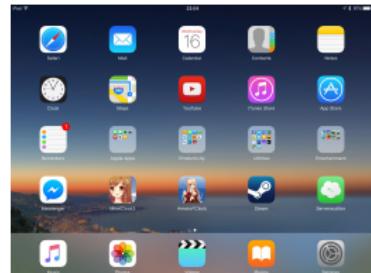
Nokia Symbian (2005)



Windows Mobile 6.1 (2008) + 10 (2015)



Android 5 "Lollipop" (2014)



Apple iOS 9 (2015)

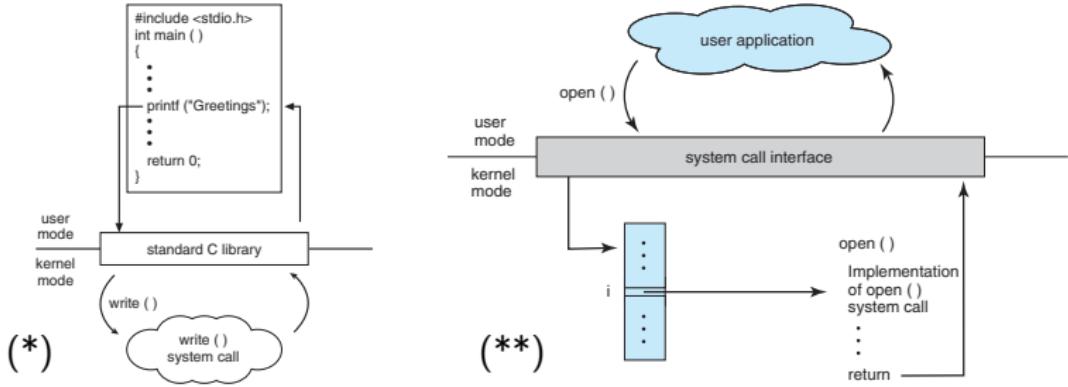
Componentes do SO - Chamadas ao Sistema (12/16)

- *chamadas ao sistema (syscalls)* são funções disponibilizadas pelo SO, que fornecem às aplicações um interface de acesso aos seus serviços
- em Linux, o comando **strace** mostra as chamadas ao sistema realizadas por um programa; exemplo: **strace cp x.c y.c**:

```
execve("/bin/cp", ["cp", "x.c", "y.c"], /* 38 vars */) = 0
execve("/bin/cp", ["cp", "x.c", "y.c"], /* 38 vars */) = 0
...
stat64("y.c", {st_mode=S_IFREG|0644, st_size=41, ...}) = 0
stat64("x.c", {st_mode=S_IFREG|0644, st_size=53, ...}) = 0
stat64("y.c", {st_mode=S_IFREG|0644, st_size=41, ...}) = 0
open("x.c", O_RDONLY|O_LARGEFILE)      = 3
fstat64(3, {st_mode=S_IFREG|0644, st_size=53, ...}) = 0
open("y.c", O_WRONLY|O_TRUNC|O_LARGEFILE) = 4
fstat64(4, {st_mode=S_IFREG|0644, st_size=0, ...}) = 0
read(3, "#include <stdio.h> main() { prin"..., 32768) = 53
write(4, "#include <stdio.h> main() { prin"..., 53) = 53
read(3, "", 32768)                  = 0
close(4)                           = 0
close(3)                           = 0
close(0)                           = 0
close(1)                           = 0
```

Componentes do SO - Chamadas ao Sistema (13/16)

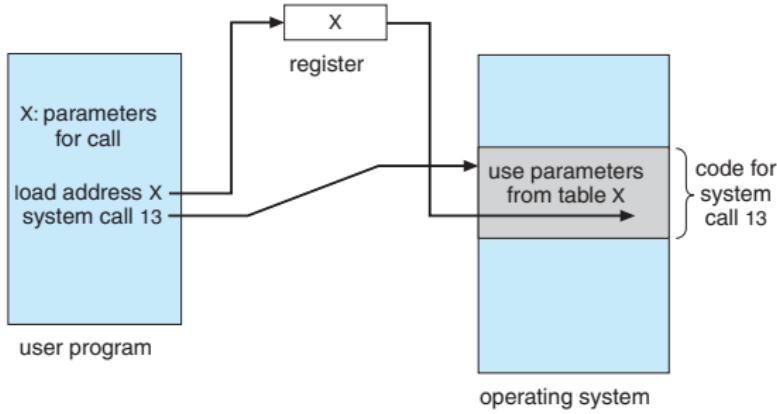
- a maioria dos programadores não invoca diretamente *chamadas ao sistema*, usando APIs de mais alto nível (POSIX API, Win API, Java API, etc.)
 - vantagens: interface mais simples, aplicações mais portáveis
 - as funções destas APIs invocam as *syscalls* necessárias (*)
- a invocação de uma *syscall* é interceptada pelo *system call interface* (**)
 - este invoca a *syscall* no *kernel*, com base num id único por *syscall*
 - no *kernel*, esse id permite localizar o código específico da *syscall*
 - ver exemplos para o sistema operativo Linux nos slides [104 a 109](#)



Componentes do SO - Chamadas ao Sistema (14/16)

Passagem de Parâmetros:

- ① através de *registos*; viável apenas se o número de parâmetros é reduzido; exemplo: o Linux usa esta estratégia para 5 ou menos parâmetros
- ② através de uma tabela ou bloco de memória, com o endereço respetivo em registo; exemplo: o Linux segue esta opção para mais de 5 parâmetros



- ③ através da *stack* do programa do utilizador, de onde são extraídos pelo SO

Os métodos 2 e 3 permitem qualquer número e dimensão dos parâmetros.

Tipos de Chamadas ao Sistema:

- Process control
 - end, abort
 - load, execute
 - create process, terminate process
 - get process attributes, set process attributes
 - wait for time
 - wait event, signal event
 - allocate and free memory
- File management
 - create file, delete file
 - open, close file
 - read, write, reposition
 - get and set file attributes
- Device management
 - request device, release device
 - read, write, reposition
 - get device attributes, set device attributes
 - logically attach or detach devices
- Information maintenance
 - get time or date, set time or date
 - get system data, set system data
 - get and set process, file, or device attributes
- Communications
 - create, delete communication connection
 - send, receive messages
 - transfer status information
 - attach and detach remote devices

Componentes do SO - Chamadas ao Sistema (16/16)

Exemplos de Chamadas ao Sistema:

	Windows	Unix
Process Control	CreateProcess() ExitProcess() WaitForSingleObject()	fork() exit() wait()
File Manipulation	CreateFile() ReadFile() WriteFile() CloseHandle()	open() read() write() close()
Device Manipulation	SetConsoleMode() ReadConsole() WriteConsole()	ioctl() read() write()
Information Maintenance	GetCurrentProcessID() SetTimer() Sleep()	getpid() alarm() sleep()
Communication	CreatePipe() CreateFileMapping() MapViewOfFile()	pipe() shmget() mmap()
Protection	SetFileSecurity() InitializeSecurityDescriptor() SetSecurityDescriptorGroup()	chmod() umask() chown()

1.5 Programas do Sistema e Aplicações

Programas do Sistema e Aplicações (1/1)

- proporcionam funcionalidades úteis à exploração do sistema
- acompanham o SO, ou são instalados pelos utilizadores
- **comandos e utilitários de sistema:**
 - Manuseamento de Ficheiros e Pastas: criação/remoção, movimentação /cópia, edição, filtragem, transformação, pesquisa, compressão. ...
 - Informação de Estado: data e hora, ocupação de recursos (cpu, memória, disco, rede), utilizadores ligados, registos de eventos (logs)
 - Desenvolvimento de Software: compiladores, ligadores, carregadores, depuradores, interpretadores, bibliotecas, gestão de versões, ...
 - Comunicações: gestão de canais entre processos, utilizadores, hosts

Programas do Sistema e Aplicações (1/1)

- **serviços de sistema (daemons):** iniciam no arranque do SO, terminando rapidamente, ou persistindo em segundo plano
 - são serviços que executam acima do kernel (em modo utilizador)
 - rede, impressão, indexação, updates, logs, exec. periódica (cron), ...
- **aplicativos:** respondem a necessidades específicas dos utilizadores
 - browsers, office, comunicações, imagem/vídeo, jogos, virtualização, ...

Na prática, a visão que os utilizadores têm do sistema operativo é definida pelos seus comandos e pelos aplicativos suportados (e não pelas syscalls oferecidas).

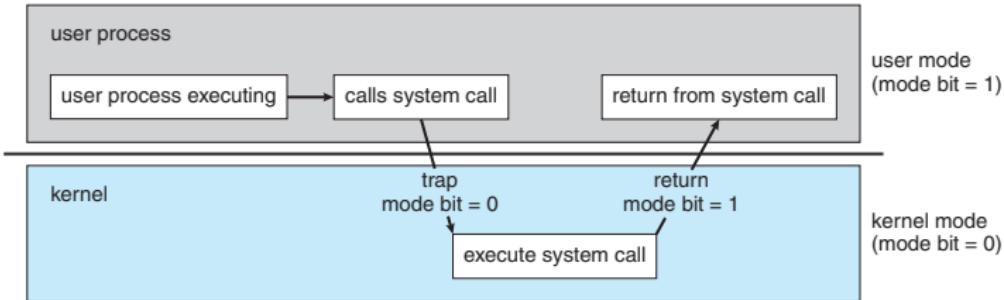
1.6 Proteção do Hardware

Operação Dual-Mode / Multimode

- a operação **dual-mode** garante que só o SO executa certas operações:
 - modo **kernel/supervisor/monitor/sistema/privilegiado** – ISA 100% disponível; acesso ao HW sem restrições; todas as operações disponíveis
 - modo **utilizador/não-privilegiado** – ISA parcialmente disponível; acesso a operações privilegiadas apenas através de syscalls
- modo atual de execução definido pelo valor do **bit de modo** na CPU
- no arranque do sistema, o modo kernel é ativado automaticamente; e sempre que o sistema operativo executa, é também esse o modo ativo
- antes de o SO atribuir CPU a um processo normal, ativa o modo utilizador; regresso ao modo kernel: invocação de syscall, ou outra interrupção
- com virtualização, são necessários vários bits (operação **multimode**): um SO virtualizado não pode ter os mesmos privilégios do hypervisor

Protecção do Hardware (2/4)

Instruções Privilegiadas



- um processo normal solicita uma operação privilegiada invocando uma **chamada ao sistema** (*system call*), o que provoca uma interrupção (*trap* – interrupção gerada por software), e consequente mudança para modo kernel
- **operações privilegiadas:** i) interação com periféricos (E/S); ii) gestão de interrupções; iii) mudança do bit de modo; iv) gestão do *timer* da CPU (ver a seguir); v) gestão da memória (ver a seguir); vi) gestão das *caches*; vii) etc.
- **pergunta:**
 - e se um processo normal pudesse modificar o vetor de interrupções ?

Protecção do Hardware (3/4)

Protecção E/S

- os processos normais só conseguem executar operações E/S indiretamente, através da invocação de chamadas ao sistema

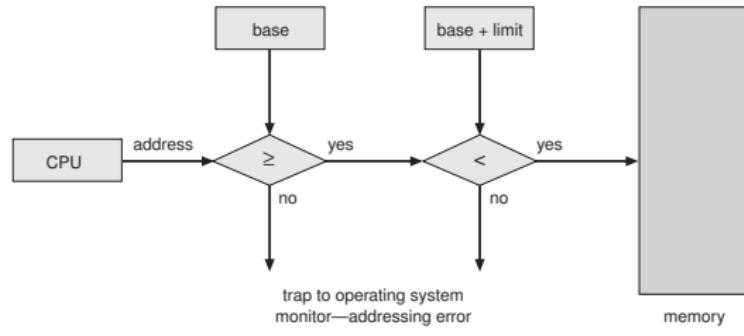
Protecção da CPU

- evita que um processo normal monopolize para si o tempo de CPU
- mecanismo básico: assente num **timer** (dispositivo temporizador), que interrompe a CPU após um intervalo de tempo configurável
 - o *timer* dispõe de um contador, decrementado a cada pulso do relógio
 - o SO inicializa o contador antes de atribuir a CPU a um processo
 - quando o contador atinge o valor 0, o *timer* gera uma interrupção
 - a programação do *timer* assenta em instruções privilegiadas ...
 - **pergunta:** se assim não fosse, o que poderia acontecer ?
- o *timer* é tipicamente usado para concretizar *time-sharing*
- o *timer* pode também ser usado para contagem de tempo

Protecção do Hardware (4/4)

Protecção de Memória

- objetivo: proteger as zonas de memória afetas ao SO e aos utilizadores
- mecanismo básico: endereços acessíveis definidos com base em registos
 - registo base – contém o menor endereço físico legal
 - registo limite – contém a dimensão da gama
 - as instruções de carregamento dos registos são *privilegiadas*
- qualquer acesso à memória deve ser intercetado e validado pela CPU
- o SO executa em *modo kernel* \Rightarrow acesso ilimitado a toda a memória



1.7 Estrutura do Sistema Operativo

Estrutura do Sistema Operativo (1/13)

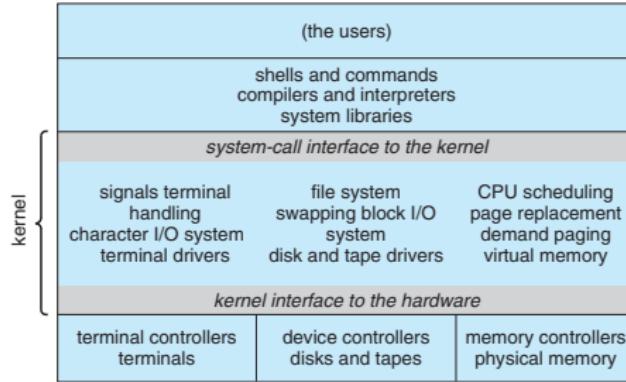
- um SO de uso genérico é uma peça de software grande e complexa (SOs mais especializados, são mais pequenos e mais simples)
- há várias abordagens à forma como o software do SO é estruturado
 - estrutura **monolítica**
 - estrutura **por camadas**
 - estrutura baseada em **microkernel**
 - estrutura **modular**
 - estrutura **híbrida**

Estrutura do Sistema Operativo (2/13)

Estrutura Monolítica:

- o *kernel* é um só binário estático, há um só espaço de endereçamento

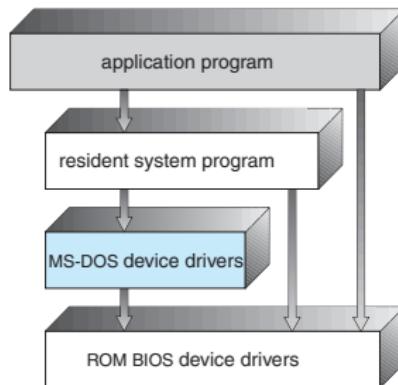
Estrutura Monolítica de Referência: UNIX



- estrutura simples, limitada pelo hardware (limitado) então disponível
 - núcleo (*kernel*): tudo o que está “acima” do hardware e “abaixo” da interface oferecida pelas *chamadas ao sistema (system calls)*
 - programas e *daemons*: acedem às funções do SO pelas *system calls*
 - porém, a camada do núcleo concentra demasiada funcionalidade

Estrutura do Sistema Operativo (3/13)

Estrutura Monolítica Simples: MS-DOS

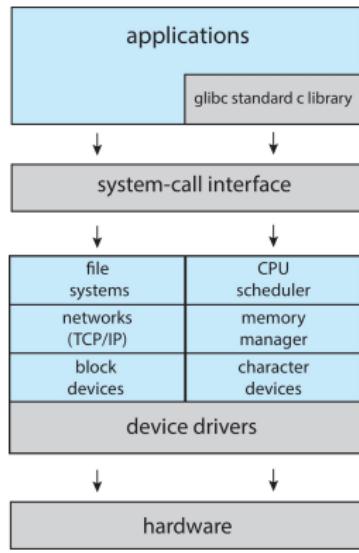


- estrutura também limitada pelo hardware ao qual se destinava
 - CPU 8088 não suportava modo dual, RAM limitada a 640 KB
- tenta fornecer a maior funcionalidade possível com esses recursos
 - os diferentes níveis de funcionalidade não estão isolados
 - o ambiente de execução era muito pouco robusto ...

Estrutura do Sistema Operativo (4/13)

Estrutura Monolítica Atual: Linux

- monolítico porque é similar ao UNIX (kernel = 1 ficheiro = /boot/vmlinuz)
- aplicações podem invocar *syscalls* diretamente, ou através da glibc
- também **modular** porque suporta modificação do kernel em run-time (inserção/remoção de módulos)

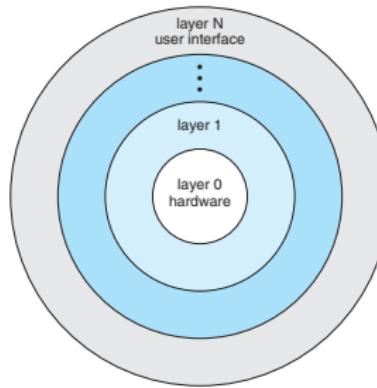


Estrutura Monolítica: (Des)Vantagens

- aparentemente mais simples, mas de difícil implementação e extensão
- bom desempenho: pouca sobrecarga no interface das primitivas e na comunicação intra-kernel; abordagem ainda mt presente nos SOs atuais

Estrutura do Sistema Operativo (5/13)

Estrutura por Camadas:



- o SO é dividido em múltiplas camadas ou níveis, mais específicos
- cada camada assenta em (usa funcionalidades de) camadas inferiores
- o interface oferecido por cada camada é padronizado e conhecido
- a camada mais inferior (nível 0) é o hardware
- a camada mais superior (nível N) é a interface dos utilizadores

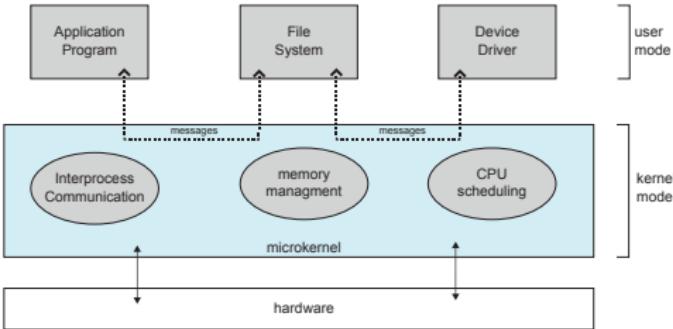
Estrutura por Camadas (cont.):

- vantagens
 - simplicidade de desenho, implementação e depuração
- desvantagens
 - dificuldade na definição correta das camadas (dependências mútuas)
 - menor desempenho potencial: um pedido atravessa várias camadas até que seja concretizado; cada camada incorre numa penalização temporal
- tendências
 - menos camadas, com mais funcionalidade por camada

Estrutura do Sistema Operativo (7/13)

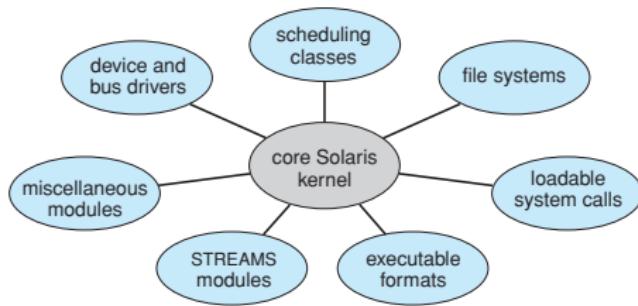
Microkernel:

- passar a maior quantidade possível de serviços do *núcleo* para *user space*
- *núcleo*: gestão de memória e processos, comunicação inter-processos
- comunicação inter-processos: *passagem de mensagens* através do núcleo
- vantagens
 - boa expansibilidade: novos serviços são criados em *user space*, sem modificações ao microkernel (ou com muito poucas modificações)
 - melhor portabilidade entre arquiteturas diferentes
 - maior robustez e segurança (menos código em *kernel mode*)
- desvantagem: < desempenho pela necessidade de comunicação via núcleo
- exemplos: Mach, Tru64 UNIX, Windows NT 3.x, QNX (RTOS)



Estrutura do Sistema Operativo (8/13)

Módulos:



- suportados pela maior parte dos SOs modernos
- cada componente do núcleo é implementado como um módulo
- módulos comunicam entre si através de interfaces bem conhecidos
- podem ser carregados no arranque ou em plena operação do sistema
- ≈ Estrutura por Camadas (modularidade), mas + flexível:
 - os módulos podem comunicar direta/ sem obedecer a uma hierarquia rígida
- ≈ Microkernels (um módulo principal), mas + flexível:
 - os módulos podem comunicar direta/, sem recurso ao módulo principal

Sistemas Híbridos:

- A maioria dos SOs modernos segue uma abordagem híbrida ...
- ... a fim de melhorar o desempenho, segurança e usabilidade
- **Linux:** **monolítico**, porque o kernel reside num só espaço de endereçamento, o que beneficia o desempenho; mas também **modular**, permitindo carregamento dinâmico de funcionalidades
 - sugestão: explorar os comandos `lsmod`, `modinfo`, `insmod`, `rmmmod` e os ficheiros de configuração `/etc/modules`, `/etc/modprobe.d/*`
- **Windows:** maioritariamente **monolítico** (por causa do desempenho); mas com características dos sistemas **microkernel**, suportando subsistemas (*personalities*) que correm como processos *user-level*
 - Win 10/11: inclui o **Windows Subsystem for Linux (WSL)**; permite executar aplicações Linux via translação de syscalls (v1), ou com base num kernel de Linux (v2) que corre lado-a-lado (numa VM) com o kernel do Windows

Estrutura do Sistema Operativo (10/13)

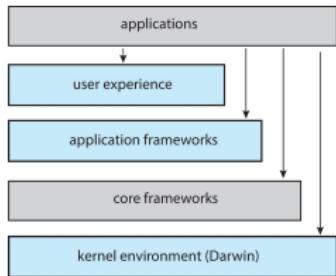
Sistemas Híbridos: macOS e iOS – Arquitetura Geral



macOS:



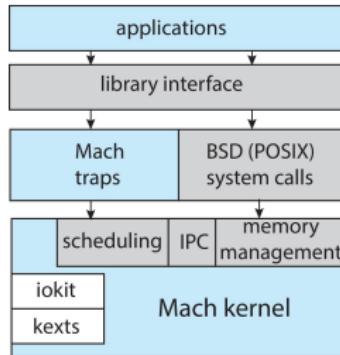
iOS:



- **user experience:** mac OS usa o interface *Aqua* (desenhado para rato e teclado); iOS usa o interface *Springboard*, para dispositivos com ecrãs sensíveis ao toque
- **application frameworks:** inclui o *Cocoa* (macOS) e o *Cocoa Touch* (iOS), que oferecem uma API para desenvolvimento de aplicações em Objective-C e Swift
- **core frameworks:** suportam gráficos (OpenGL) e multimédia (Quicktime)
- **kernel Darwin: híbrido** – inclui o **microkernel** Mach e um kernel BSD UNIX
 - código fonte do Darwin é aberto, ao contrário do das camadas acima
- as **aplicações** podem aceder diretamente a qualquer uma destas camadas

Estrutura do Sistema Operativo (11/13)

Sistemas Híbridos: macOS e iOS – kernel Darwin:

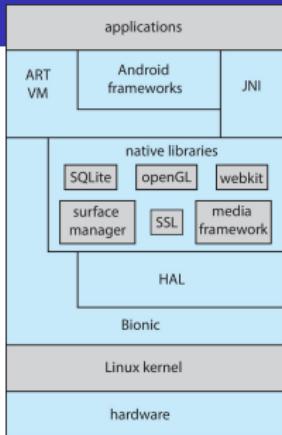


- **system call interface:** dois interfaces; primitivas Mach (*traps*) e primitivas BSD
- **Mach microkernel:**

- gestão de memória e de processos, comunicação inter-processos (IPC), ...
- suporta **kernel abstractions**: e.g., um *processo* criado pela primitiva BSD *fork* será representado/suportado no kernel Mach por uma *tarefa* deste
- oferece um **iokit** para desenvolvimento de *device drivers*
- suporta **módulos** carregáveis dinamicamente (**kexts**: kernels extensions)
- não é um microkernel puro: os seus vários subsistemas partilham o mesmo espaço de endereçamento, o que acelera a passagem de mensagens entre eles

Estrutura do Sistema Operativo (12/13)

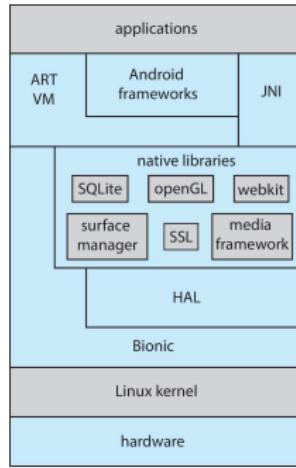
Sistemas Híbridos: Android



- SO de **código aberto**, da Google, para sistemas móveis **multi-arquitetura**
- estrutura por camadas, com várias frameworks e bibliotecas acima do kernel
- sistema **híbrido**: tal como no iOS, frameworks e bibliotecas são parte do SO
- aplicações codificadas em **Java**
 - Android API for Java: apps compiladas para executar numa **ART VM** (Android RunTime virtual machine), otimizada para pouca RAM e CPUs modestos; apps compiladas para código máquina nativo durante a instalação
 - Java Native Interface (**JNI**): acesso mais direto ao HW; apps não portáveis

Estrutura do Sistema Operativo (13/13)

Sistemas Híbridos: Android



- **bibliotecas nativas**: bases de dados (SQLite), browsing (webkit), multimedia, ...
- **Hardware Abstraction Layer (HAL)**: facilita a execução do Android em diferente HW; oferece vista consistente dos vários dispositivos (sensores, GPS, câmaras, ...); permite o desenvolvimento de aplicações móveis com elevada **portabilidade**
- **Bionic**: equivalente à *GNU C library* (glibc), mas mais leve (em RAM e CPU)
- **Linux kernel**: modificado para dispositivos móveis (gestão de energia, etc.)

1.8 Desenvolvimento do Sistema Operativo

Implementação do Sistema (1/2)

- inicialmente: sistema operativo escrito totalmente em assembly
- posteriormente: 1as linguagens de alto nível para programação de sistemas (Algol, PL/1); desenvolvimento mais eficiente; portabilidade ainda limitada
- posteriormente: adoção da linguagem C; grande eficiência e portabilidade
- atualmente:
 - kernel - níveis mais baixos em assembly, restante código em C (e Rust)
 - utilitários de sistema - desenvolvidos em C, C++, Perl, Python, Bash
 - Android - bibliotecas de sistema em C/C++; frameworks de suporte às aplicações (que fornecem o interface de acesso às bibliotecas) em Java
- vantagens da codificação em linguagens de alto nível:
 - desenvolvimento mais rápido
 - maior portabilidade
 - maior facilidade de compreensão
 - maior facilidade de depuração
 - maior facilidade de configuração e afinação
 - beneficia-se da evolução dos compiladores

Geração do Sistema (2/2)

- quando se compra/prepara um computador, o sistema operativo instalado está num formato binário (executável), e pré-configurado de forma genérica
 - exemplo: Linux pré-compilado para x86 (32 bits) vs x86-64 (64 bits)
- por vezes, é necessário re-compilar o sistema operativo, a fim de adicionar/retirar/modificar funcionalidades; a recompilação só é viável se o código fonte estiver disponível (Linux, *BSD, Minix, ReactOS, FreeDOS, etc.)
 - exemplo: retirar/adicionar/modularizar o suporte a IPv6 em Linux (modularizar permite inserir/retirar o módulo respetivo em run-time)
- pode também ter interesse re-compilar ou agregar de formas diferentes os utilitários e aplicações que acompanham o kernel (conceito de distribuição)
 - exemplo: distribuições Linux personalizadas (e.g., Linux From Scratch)
 - este processo de personalização é moroso e pode demorar muito tempo
 - e envolvendo re-compilação, depende da disponibilidade do código fonte

Sistemas de Código Aberto (3/4)

- sistemas cujo código fonte está disponível para consulta e modificação
 - exemplo paradigmático: Linux (kernel e a maioria das aplicações);
 - contra-exemplo (*closed-source*): SOs da família Microsoft Windows (componentes do SO e aplicações distribuídas em formato binário)
 - abordagem híbrida: Apple Mac OS X e iOS (só o kernel Darwin aberto)
- até ao início dos anos 80, o código dos sistemas UNIX era aberto
- em 1985, Richard Stallman cria a Free Software Foundation (FSF), com o objetivo de desenvolver e proteger o *software livre*; este caracteriza-se por:
 - poder ser executado para qualquer propósito
 - poder ser estudado e modificado sem restrições
 - poder ser redistribuído (copiado) sem limites
- atenção: software livre (*free speech*) \neq software grátis (*free beer*)
 - ver <https://moqod-software.medium.com/understanding-open-source-and-free-software-licensing-c0fa600106c9>

Sistemas de Código Aberto (4/4)

- para proteger legalmente o software livre, foi criada a *General Public License* (GPL); esta licença inclui um conjunto de 3 restrições aplicadas ao software:
 - ➊ pode ser livremente distribuído / comercializado, mas o distribuidor tem de avisar os utilizadores / clientes dos termos da GPL
 - ➋ o software eventualmente derivado de software protegido pela GPL terá de estar também abrangido pela GPL
 - ➌ o código fonte de todo o software protegido pela GPL deve ser anexado ao software distribuído / comercializado ou ser de acesso público
- a GPL foi criada em 1989 para proteger o software criado no âmbito do projeto GNU (GNU is Not Unix); este pretendia desenvolver uma alternativa livre ao kernel do UNIX e apps satélite; ao longo dos anos surgiram muitas apps GNU, mas o kernel acabou por ser o Linux (1992, v0.12, GPLv2)
- hoje em dia, é comum falar em sistemas GNU/Linux (distribuições Linux)
- porém, nem todas as entidades respeitam os termos da licença GPL ...

1.9 Depuração e Monitorização do Sistema

Depuração do Sistema e Aplicações (1/6)

- **depuração** (*debugging*): encontrar a causa de erros e corrigi-los
 - problemas de desempenho podem considerar-se erros, donde a **afinação do desempenho** (*performance tuning*) é um aspeto da depuração
- quando um processo falha:
 - a falha pode ficar registada num ficheiro de **log** (usual para processos de sistema, menos usual para processos normais)
 - a imagem da memória do processo quando falhou pode ficar registada num ficheiro de **core dump** para posterior análise por um *debugger*
- a depuração de processos normais e do *kernel* usam técnicas diferentes
 - exemplo: quando o *kernel* falha, a sua memória é gravada em formato raw numa zona separada do disco, pois é arriscado gravar num ficheiro, já que a falha pode ter sido no sistema de ficheiros; durante o *reboot*, o **crash dump** é recolhido e gravado num ficheiro para posterior análise

Monitorização e Afinação do Desempenho (2/6)

- a **afinação** de desempenho tem como objetivo remover **constrangimentos** (*bottlenecks*); para tal, é necessário identificá-los através de **monitorização**
- tipos de ferramentas de monitorização:
 - incidem sobre processos específicos (**per-process**) ou sobre o sistema como um todo (**system-wide**)
 - reúnem informação através de **contadores** (estatísticas mantidas pelo kernel) ou **traçagem** (acompanhamento de uma sequência de eventos; por exemplo: sequência de instruções de um programa, chamadas ao sistema invocadas, passos na execução de uma chamada ao sistema)

Monitorização e Afinação do Desempenho (3/6)

- ferramentas **Linux** que exploram **contadores**
 - per-process / selection of processes: ps, top, htop
 - system-wide: vmstat (uso da memória), netstat (uso da rede), iostat (uso dos discos/SSDs)
 - muitas destas ferramentas consultam o pseudo-sistema de ficheiros /proc, onde o kernel expõe métricas gerais e específicas por processo
 - cat /proc/uptime: 1st value - total number of seconds the system has been up; 2nd value - sum of how much time each core has spent idle, in seconds; cat /proc/12345/statm: the status of the 12345 process memory (total program size (KB), ..., number of dirty pages)
- ferramenta **Windows** que explora **contadores**: Task Manager (taskmgr.exe)
- ferramentas **Linux** que exploram **traçagem**
 - per-process: strace - traces system calls invoked by a process; gdb - a source-level debugger
 - system-wide: perf - a collection of Linux performance tools; tcpdump - collects network packets

BCC Tracing Tools (4/6)

- a depuração das interacções entre código user-level e kernel-level necessita de ferramentas especiais, que não ponham em causa a estabilidade do sistema
- adicionalmente, o impacto no desempenho deve ser mínimo
- o toolkit BCC (BPF Compiler Collection) cumpre estas condições
 - o BCC é um front-end para o eBPF (extended Berkeley Packet Filter)
 - o BPF foi inicialmente (1990s) desenvolvido para filtrar tráfego de rede
 - o eBPF estendeu o BPF para capturar no kernel eventos de vários tipos (invocação de primitivas específicas, tempo de uma operação IO, etc.)
 - as sondas eBPF são escritas num sub-conjunto do C e compiladas em "instruções eBPF" que podem ser inseridas dinamicamente no kernel
 - escrever sondas eBPF em C é complexo ... o BCC permite usar Python
 - uma ferramenta BCC escreve-se em Python e inclui código C embbebido para interface com a instrumentação eBPF que interage com o kernel
 - o código C dá origem a instruções eBPF que são inseridas no kernel utilizando duas técnicas possíveis: **sondas ou pontos de traçagem**

BCC Tracing Tools (5/6)

Exemplos de Ferramentas BCC

- `disksniff.py`: monitoriza a atividade IO dos discos (system-wide trace)

TIME(s)	T	BYTES	LAT(ms)
1946.29186700	R	8	0.27
1946.33965000	R	8	0.26
1948.34585000	W	8192	0.96
1950.43251000	R	4096	0.56
1951.74121000	R	4096	0.35

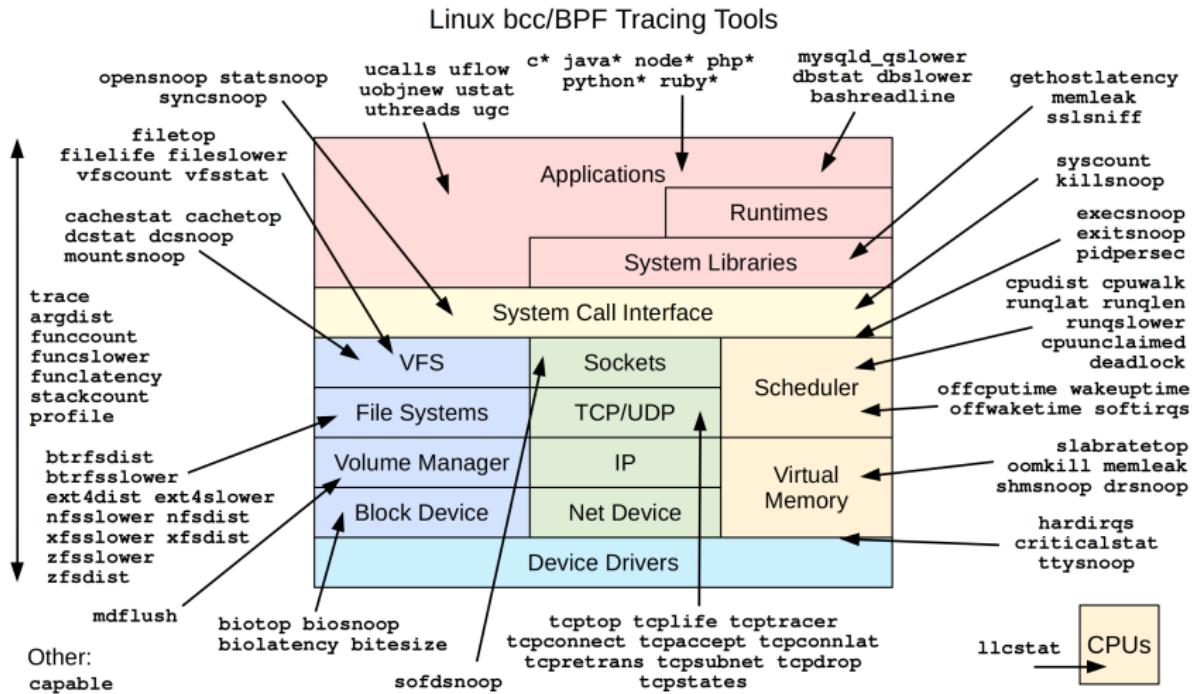
- TIME(s): timestamp da operação; T: tipo da operação (R = Read, W = Write); BYTES: nº de bytes envolvidos; LAT(ms): latência (duração) da operação
- `opensnoop -Tp 1956`: monitoriza as invocações à primitiva open pelo processo de PID 1956 (-p) mostrando o instante (-T) em que acontecem

TIME(s)	PID	COMM	FD	ERR	PATH
0.000000000	1956	supervise	9	0	supervise/status.new
0.000289999	1956	supervise	9	0	supervise/status.new
1.023068000	1956	supervise	9	0	supervise/status.new
1.023381997	1956	supervise	9	0	supervise/status.new

- TIME(s): timestamp (instante de tempo)) da operação; PID: pid do processo; COMM: programa do processo; PATH: ficheiro aberto (o mesmo, 2 vezes/s)

BCC Tracing Tools (6/6)

Ferramentas BCC disponíveis

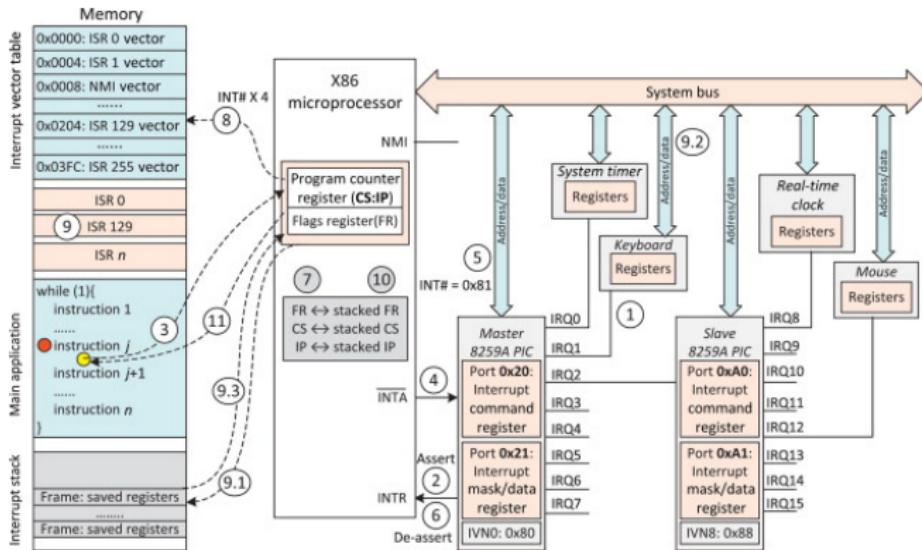


<https://github.com/iovisor/bcc#tools> 2019

1.10 Slides Suplementares

- **prioridades**: interrupções mais prioritárias podem interromper o tratamento em curso de interrupções menos prioritárias
- **inibição**: *nonmaskable interrupts* - interrupções que nunca podem ser ignoradas; *maskable interrupts* - interrupções ignoráveis (desligáveis)
- **chaining**: cada célula do vetor de interrupções aponta para uma lista de rotinas; útil quando há mais interrupções que células no vetor
- tratamento de uma interrupção em CPUs x86: slides [74](#) a [77](#)

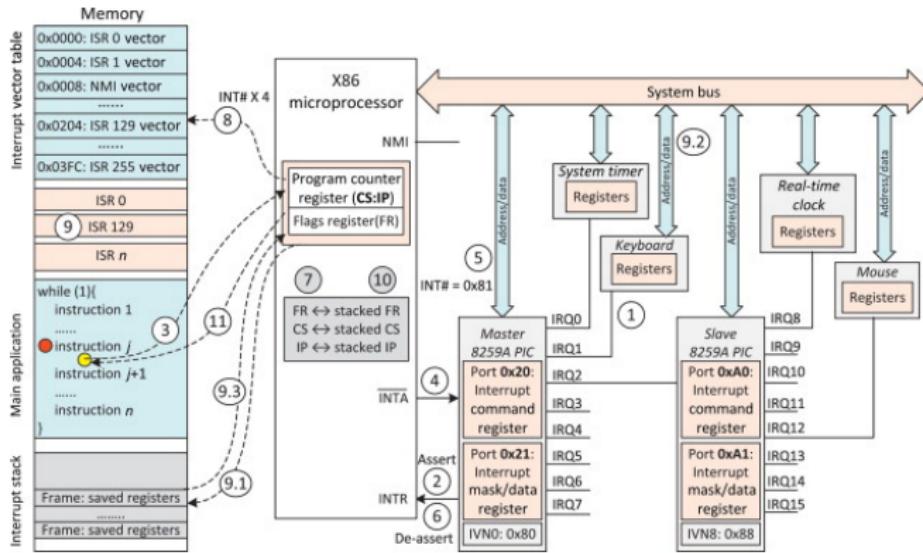
Processamento de Interrupções x86 (1/4)



<https://www.sciencedirect.com/topics/engineering/interrupt-vector>

1. A user presses a key on the keyboard, which drives an IRQ on the IRQ1 line of the 8259 master PIC
2. The PIC detects the IRQ on IRQ1 [...] and asserts the INTR line to inform the processor.
3. While the instruction j of the current program is being executed, the processor samples INT and detects an asserted line. After the execution of the instruction j has been completed, the current program is suspended. At this point, the value contained by CS:IP is the location of the next instruction of the current program.
4. The processor examines the interrupt flag within the flags register. If the interrupt flag is set, the processor acknowledges the IRQ by asserting the INTA line to the PIC, expecting an interrupt vector number from the PIC.

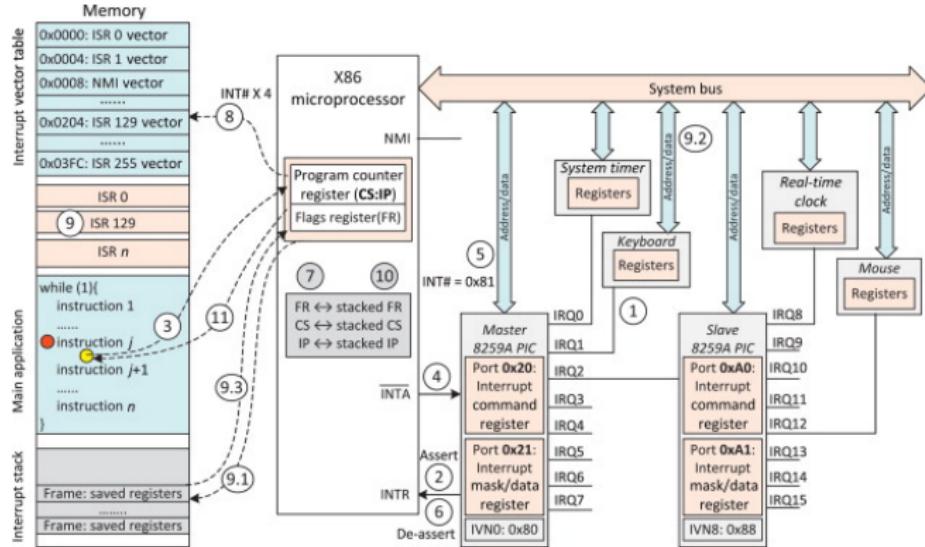
Processamento de Interrupções x86 (2/4)



<https://www.sciencedirect.com/topics/engineering/interrupt-vector>

5. The PIC drives the interrupt vector number 0x81 to the system bus [...]
6. The PIC then deasserts the INTR line (so that a new IRQ can be asserted).
7. [...] The values of CS and IP are pushed onto the interrupt stack, so that the original program can be resumed later
8. To protect context switching, the "interrupt enable" flag of the flags register is cleared to disable further interrupts. The processor comes to the keyboard interrupt vector cell, which has 4 bytes, located at $4 * 0x81 = 0x0204$.
9. The keyboard interrupt vector cell contains the address of the keyboard ISR. By loading IP with the 16-bit data at 0x0204 and loading CS with the 16-bit data at 0x0206, the processor jumps to the start location of the keyboard ISR

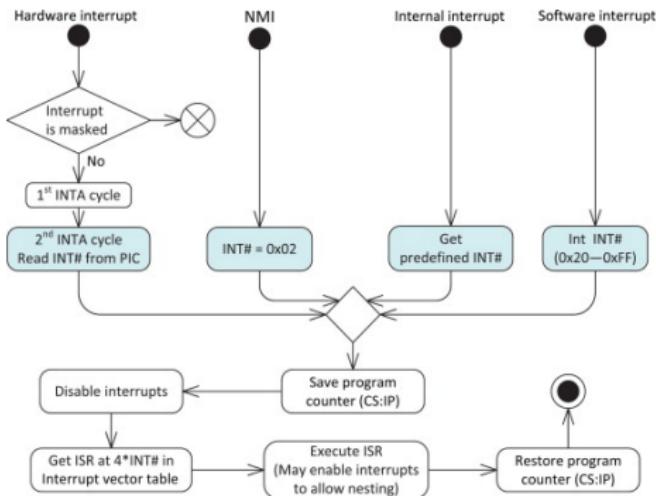
Processamento de Interrupções x86 (3/4)



<https://www.sciencedirect.com/topics/engineering/interrupt-vector>

- 9.1 Inside the ISR, the commonly used registers are first saved onto the interrupt stack. At this point, the processor has switched its context from the last task to the current ISR. The "interrupt enable" flag of the flags register is set to enable further interrupts, if interrupt nesting is desired.
- 9.2 The portion of code pertinent to the requesting device is executed. In this case, the code of the key being pressed is stored in a memory area called the keyboard buffer.
- 9.3 At the end of the ISR, interrupts are again disabled for context switching. [...] The ISR then performs an instruction to pop up the top frame of the interrupt stack.
10. The context of the original task is restored. Especially, CS:IP now refers to the next instruction of the original program.
11. The processor is ready to resume the execution of the original program.

Processamento de Interrupções x86 (4/4)

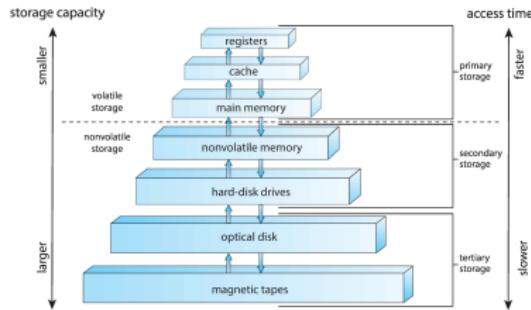


<https://www.sciencedirect.com/topics/engineering/interrupt-vector>

- **hardware interrupt:** the vector number is typically read from the PIC or the hardware device itself
- **nonmaskable hardware interrupt:** the vector number is fixed to 0x02.
- **internal interrupt (exception):** depending on the type of the exception that occurred, the vector number is also fixed. In the case of an error, an x86 processor may raise an exception prior to completing the current instruction (which is re-executed upon exception return), or may raise an exception after completing the current instruction (which is not re-executed upon exception return). For serious errors, the processor might abort the misbehaving program.
- **software interrupt:** the vector number is simply part of the instruction.

Organização do Armazenamento (1/6)

Hierarquia de Armazenamento



Level	1	2	3	4	5
Name	registers	cache	main memory	solid state disk	magnetic disk
Typical size	< 1 KB	< 16MB	< 64GB	< 1 TB	< 10 TB
Implementation technology	custom memory with multiple ports CMOS	on-chip or off-chip CMOS SRAM	CMOS SRAM	flash memory	magnetic disk
Access time (ns)	0.25 - 0.5	0.5 - 25	80 - 250	25,000 - 50,000	5,000,000
Bandwidth (MB/sec)	20,000 - 100,000	5,000 - 10,000	1,000 - 5,000	500	20 - 150
Managed by	compiler	hardware	operating system	operating system	operating system
Backed by	cache	main memory	disk	disk	disk or tape

- os vários sistemas de armazenamento formam uma hierarquia, definida segundo a *velocidade, custo, volatilidade e capacidade* desses sistemas

Organização do Armazenamento (2/6)

Estruturas de Armazenamento:

- **Registos**: unidades básicas de armazenamento, localizadas na CPU; número e capacidade limitados; acesso instantâneo (típica / 1 ciclo)
- **Caches**: memórias intermédias, concebidas para minimizar acessos à memória principal (tiram partido da localidade de referências)
 - o acesso à RAM pode consumir vários ciclos de CPU ⇒ utilização de cache(s) para "alimentar" a CPU, em paralelo com o acesso à RAM
- **Memória Principal (RAM)**:
 - array de bytes, ou palavras (grupos de bytes), direta/ endereçáveis
 - a única memória de "grande" dimensão à qual a CPU acede direta/
 - acesso explícito: instruções do tipo **LOAD/STORE**
 - acesso implícito: leitura da próxima instrução e seus operandos

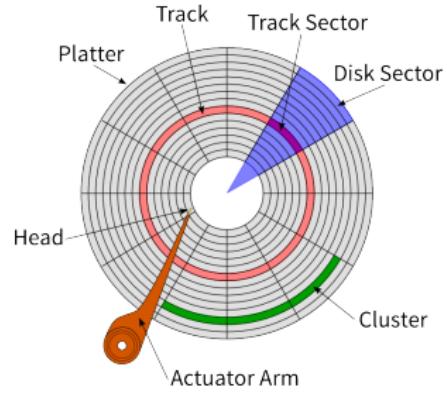
todas estas memórias são **voláteis** (perdem o seu conteúdo na ausência de corrente elétrica) e insuficientes para albergar a totalidade de um programa (registos e caches) ou até mesmo todos os programas pretendidos (RAM)

Organização do Armazenamento (3/6)

Estruturas de Armazenamento (cont.)

- Memória Secundária – **Discos Rígidos**

- não-volátil (persistente); elevada capacidade (atualmente: TBs)
- conjunto de pratos rígidos de metal/vidro, cobertos por um material magnético regravável, suspensos num eixo comum, em rotação
- cabeças de leitura/escrita suspensas a distâncias mínimas da superfície
- as superfícies dos pratos dividem-se em *pistas* e estas em *sectores*
- acesso aleatório (direto e independente) a qualquer sector



Organização do Armazenamento (4/6)

Estruturas de Armazenamento (cont.)

- Memória Secundária – **Discos de Estado Sólido (SSDs)**
 - não-volátil (persistente); de *média* capacidade de armazenamento
 - maioritariamente baseados em memórias de tipo FLASH ou similares
 - (+) > rapidez, < ruído e < consumo energético, face aos discos rígidos
 - (+) s/ partes móveis: "imunes" a falhas mecânicas (não às eletrónicas)
 - (-): relação custo/capacidade menos favorável, face aos discos rígidos
 - (-): durabilidade/fiabilidade incerta/limitada, face aos discos rígidos
 - (-): desempenho assimétrico: velocidade leitura > escrita >> remoção



Organização do Armazenamento (5/6)

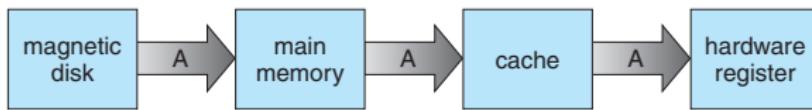
Caching

- cópia temporária dos dados, de um suporte mais lento (mas de maior capacidade), para um outro mais rápido (mas de menor capacidade)
- as *caches* são alimentadas por instruções ou dados de acesso recente ou expectável (*i.e.*, tiram partido do *princípio da localidade de referências*)
- o correto dimensionamento das *caches* e a escolha adequada do algoritmo de substituição podem assegurar *taxis de acerto* entre 80% a 99%
- na hierarquia de armazenamento, um nível funciona como *cache* do anterior
 - o acesso aos registos da CPU precede o acesso às *caches L1/L2/L3*
 - o acesso às *caches L1/L2/L3* precede o acesso à RAM
 - o acesso à RAM precede o acesso ao Disco, etc.
- movimentação de dados entre os níveis da hierarquia de armazenamento:
 - *implícita*: diretamente controlada pelo hardware, sem intervenção do SO (*e.g.*, de uma *cache L1* para os registos da CPU);
 - *explícita*: com intervenção do SO (*e.g.*, pedido do SO ao controlador do Disco para transferir dados para a RAM através de DMA)

Organização do Armazenamento (6/6)

Caching (cont.)

- numa hierarquia de armazenamento há replicação de dados em vários níveis



- num cenário com apenas um processo este esquema não levanta dificuldades: será sempre acedida a cópia no mais alto nível da hierarquia
- num ambiente multitarefa, é preciso garantir que todos os processos interessados em A tenham acesso à cópia mais recente
- num sistema multiprocessador o cenário agrava-se porque o nível mais alto da hierarquia (registos da CPU e memórias cache) encontra-se representado em cada processador ⇒ necessidade de *algoritmos de coerência das caches*
- num sistema distribuído a situação é ainda mais complexa (e.g., replicação do sistema de ficheiros ⇒ necessidade de *algoritmos de gestão de réplicas*)

Organização do Processamento (1/2)

Sistemas Monoprocessadores

- sistemas com 1 CPU (e 1 núcleo), de uso genérico

Sistemas Multiprocessadores

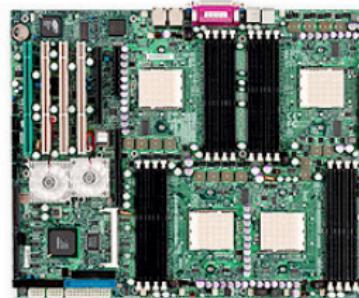
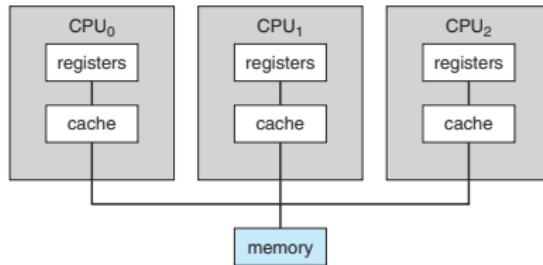
- sistemas com ≥ 1 CPU (e ≥ 1 núcleos/CPU), de uso genérico
- são sistemas estreitamente agregados (*tightly coupled*) – as CPUs partilham a RAM e o relógio; comunicação entre CPUs através da memória partilhada
- maior *throughput* (nº de processos que completam a sua execução por unidade de tempo); sobrecarga de gestão, competição por recursos ...
- boa razão custo / benefício face a múltiplos sistemas monoCPU
- maior fiabilidade: *degradação graciosa, tolerância a falhas*

Sistemas Multinúcleo (Multicore)

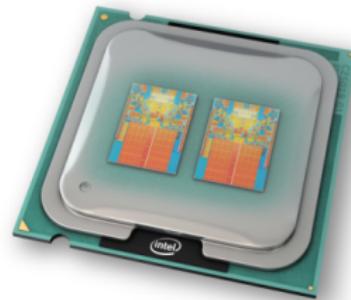
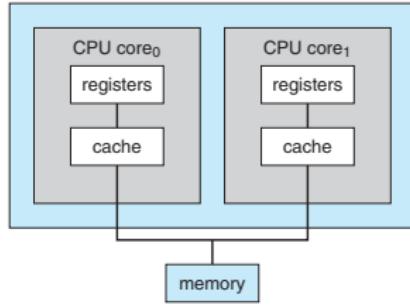
- integram, num só chip, ≥ 2 núcleos de execução (*cores*)
- $>$ proximidade entre núcleos $\Rightarrow >$ desempenho sem necessidade de frequências mt. elevadas $\Rightarrow <$ consumo e $<$ custo de produção

Organização do Processamento (2/2)

Sistema Multiprocessador



CPU Multinúcleo



Sistemas e Ambientes Especializados (1/8)

Sistemas Embebidos

- eletrónica automóvel, eletrodomésticos, domótica, robótica, etc.
- executam funcionalidades muito específicas e têm interfaces limitadas
- os SOs usados são minimalistas, ou versões mais leves de SOs genéricos

Sistemas de Tempo Real (usados, por vezes, em sistemas embebidos)

- *hard real-time systems*: i) garantem a realização das tarefas em intervalos de tempo bem definidos; ii) armazenamento secundário limitado ou ausente
- *soft real-time systems*: i) toleram (pequenos) atrasos na execução das suas tarefas; ii) exploração de prioridades / extensões *real-time* em SOs genéricos

Sistemas Multimédia

- orientados ao processamento de áudio e vídeo (*video-on-demand, streaming*)

Sistemas Handheld

- adequados às especificidades de dispositivos móveis (PDAs, telemóveis) - CPUs lentos, ecrans reduzidos, *input* limitado, largura de banda limitada

Sistemas Distribuídos

- conjunto de sistemas fisicamente separados, possivelmente heterogéneos, interligados em rede, com o propósito de elevar os níveis de funcionalidade, desempenho, fiabilidade e disponibilidade (face a um sistema isolado)
- os membros têm a noção de que estão interligados de forma cooperativa, criando a ilusão de que existe um só sistema controlando a/operando na rede
- "sistemas que falham por causa de um membro de que nunca se ouviu falar"
- exemplos: o conjunto dos servidores DNS, o sistema de ficheiros Google FS

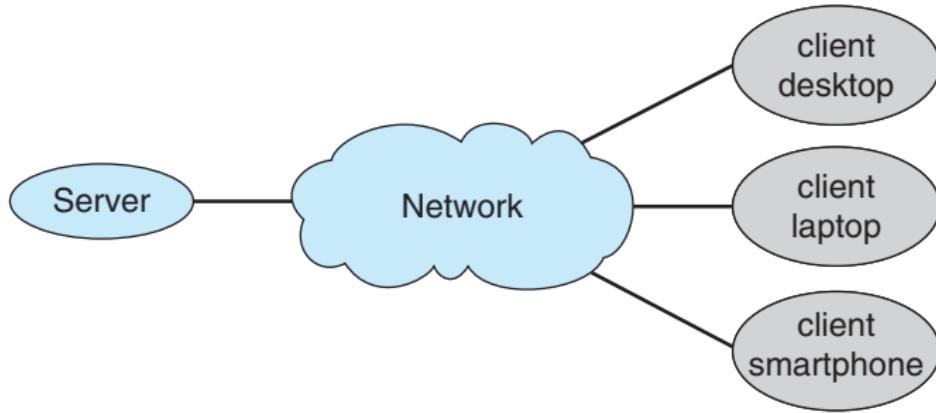
Sistemas Operativos de Rede

- capazes de trocar dados com outros sistemas ligados à mesma rede, mas agindo de forma + autónoma em comparação com os Sistemas Distribuídos
- no passado: Novell Netware, Windows for Workgroups, Windows NT; atualmente: qualquer sistema operativo de linhas desktop ou servidor

Sistemas e Ambientes Especializados (3/8)

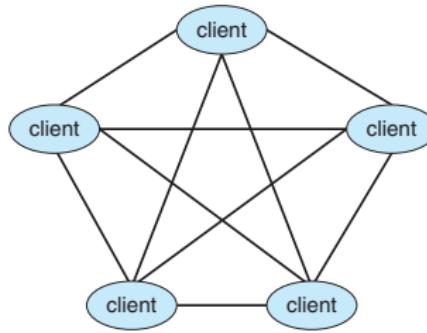
Sistemas Cliente-Servidor

- forma de Sistema Distribuído em que sistemas *clientes* acedem, através da rede, a recursos – **dados** ou **aplicações** – de um ou mais sistemas *servidores*
- cenários típicos: servidores de bases de dados, servidores web, servidores de ficheiros, servidores de aplicações/terminais/VDI, servidores de jogos on-line



Sistemas Peer-to-Peer (P2P)

- forma de Sistema Distribuído em que os membros são pares entre si (todos são clientes e servidores, em simultâneo) e a composição assume-se volátil

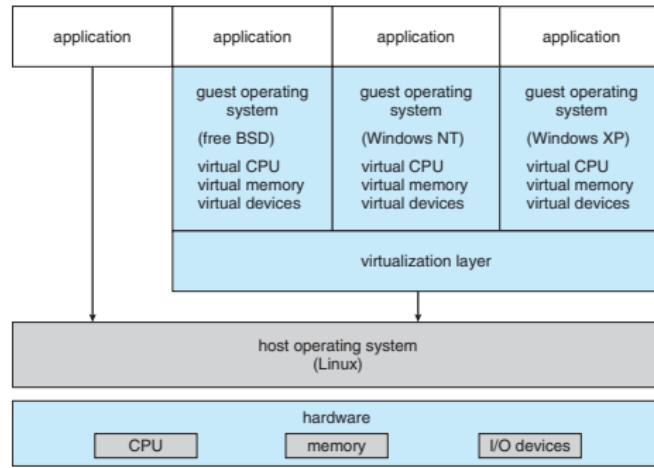


- nos primeiros sistemas P2P (*), o registo da composição e dos recursos oferecidos por cada membro era centralizado, limitando a escalabilidade
- posteriormente: pesquisa por difusão (**); pesquisa distribuída (***)
 - Distributed Hash Tables (DHTs): paradigma de localização distribuída
 - exemplos: Napster (*), Gnutella (**), BitTorrent (***)

Sistemas e Ambientes Especializados (5/8)

Virtualização

- Virtualização: técnica que permite a um SO (*guest*) executar no seio de outro SO (*host*), como se fosse mais uma aplicação deste em execução

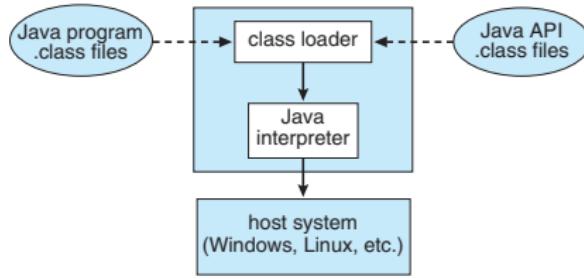


- explora vários modos de operação disponíveis nas CPUs modernas
- vantagens: > rentabilização do HW, isolamento, testes/simulações ...
- desvantagens: < desempenho, HW + exigente, ponto único de falha ...

Sistemas e Ambientes Especializados (6/8)

Virtualização (cont.)

- Emulação: variante em que o código original foi gerado para uma arquitetura diferente daquela em que vai ser executado (e.g., PowerPC vs Intel x86)
 - o impacto da tradução em plena execução pode ser considerável ...
- Interpretação: variante onde o código fonte não chega a ser compilado para código máquina de uma arquitetura real, mas sim de uma "máquina virtual"
 - exemplo: Java compilado em byte-code, executado numa JVM;
vantagem: portabilidade das aplicações (*write once, run anywhere*)



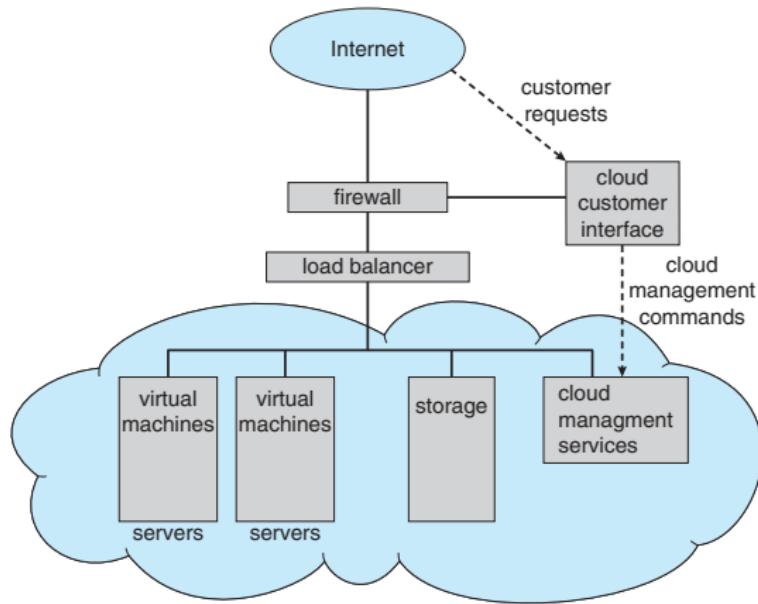
- (informação complementar nos slides 94 a 103)

Computação em Nuvem

- extensão lógica da virtualização, que é usada como tecnologia de base
- agrupa e fornece computação, armazenamento, plataformas/ambientes de desenvolvimento de aplicações, aplicações/serviços, on-demand, self-service, acessíveis via rede; recursos elásticos; utilização contabilizável (paga/grátis)
- tipos de nuvens:
 - *public cloud*: acessível via Internet, paga; free-tier limitado
 - *private cloud*: operada por uma instituição, p/ uso interno
 - *hybrid cloud*: inclui ambas as componentes pública e privada
- tipos de serviços prestados:
 - *Software as a Service (SaaS)*: aplicações/serviços end-user/turnkey; exemplos: Google Apps, Office 365, Dropbox, Salesforce CRM, ...
 - *Platform as a Service (PaaS)*: ambientes de desenvolvimento (e.g., BDs, web services); exemplo: Google App Engine, Amazon Heroku
 - *Infrastructure as a Service (IaaS)*: acesso a servidores de computação (virtuais) e a armazenamento, escaláveis/elásticos; exemplos: Amazon EC2 e S3, Google Compute Engine, Microsoft Azure, Rackspace

Computação em Nuvem (cont.)

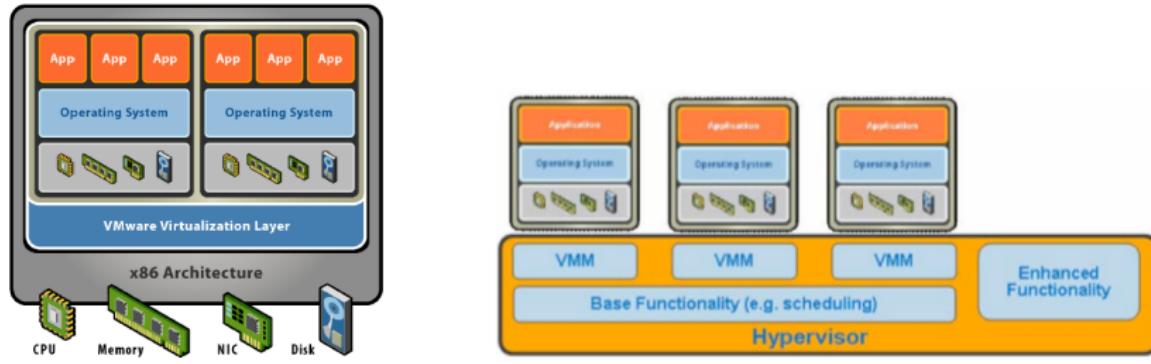
- *Infrastructure as a service (IaaS) de uma Public Cloud - exemplo:*



Virtualização de Sistemas (1/10)

Conceito

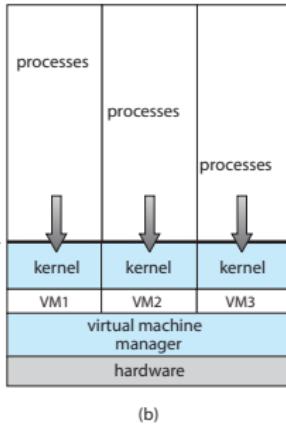
- suporta vários ambientes de execução estanques - máquinas virtuais (MVs) - na mesma máquina física, cada MV funcionando como uma máquina física
- uma MV oferece HW virtualizado com interface similar ao HW físico, permitindo instalar nela instalar um SO (*guest*) e respetivas aplicações
- um SO *hypervisor* (*host*) partilha o HW real pelas várias MVs



Virtualização de Sistemas (2/10)

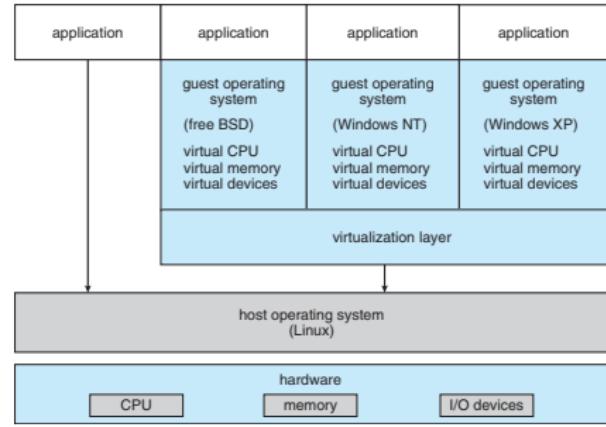
Tipos de Hypervisors

- **Tipo 1** – corre por cima do HW físico; suporta **VMs nativas**
- **Tipo 2** – corre alojado num SO normal; suporta **VMs hospedadas**



a) Sem Virt.

b) Virt. **Tipo 1**



c) Virt. **Tipo 2**

Vantagens / Potencialidades

- maior rentabilização do HW, pela sua partilha por várias VMs
- consolidação de muitos serviços em poucos servidores
 - maior taxa de utilização de equipamento oneroso
 - diminuição de custos energéticos e ambientais
- isolamento e protecção: cada VM é isolada/protegida das outras
 - falhas ou quebras de segurança naturalmente “auto-contidas” ...
 - ... mas, na prática, esse isolamento pode ser virtual/fictício: as VMs partilham, frequente/, volumes de armazenamento e redes de dados
- portabilidade, disponibilidade, balanceamento de carga
 - VMs são facilmente movidas/copiadas
 - migração/replicação de VMs em tempo real
- desenvolvimento/teste de software, incluindo multiplataforma
- simulação de cenários de deployment de novas aplicações ou updates
- experimentação/investigação de sistemas operativos; suporte a formação

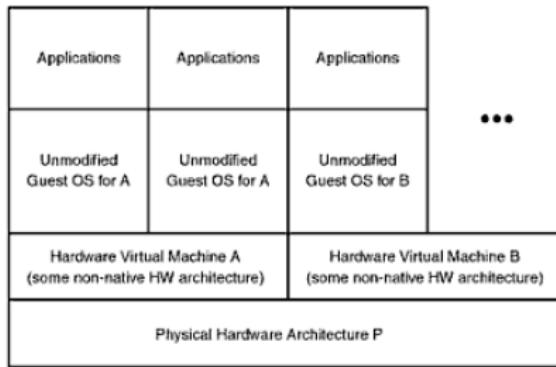
Desvantagens / Limitações

- oferecem, necessariamente, < desempenho que uma máquina real
 - processos *CPU-bound*: desempenho próximo (>95%) do nativo
 - processos *IO-bound*: penalizações mais notórias, em especial no acesso a armazenamento secundário (mitigação com SSDs, *passthrough*, etc.)
- certas plataformas de virtualização exigem hardware certificado e ferramentas sofisticadas de gestão, logo maior investimento inicial
- maior dependência de um conjunto restrito de meios: falha no HW de suporte pode implicar indisponibilidade simultânea de múltiplas VMs
- custos colaterais em cenários exigentes: reforço do desempenho e da disponibilidade (dos servidores, armazenamento e redes de dados)

Como parece evidente, as vantagens são mais que as desvantagens !!

Virtualização de Sistemas (5/10)

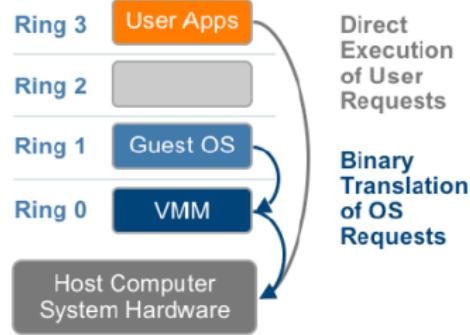
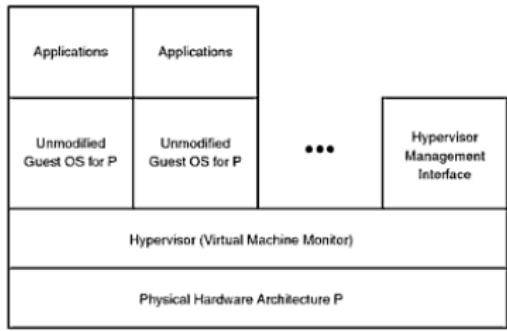
Técnicas de Virtualização - Emulação de Arquiteturas Não-Nativas



- a VM simula a arquitetura de hardware necessária para que um *guest*, concebido originalmente para essa arquitetura, execute sem modificações
- a arquitetura simulada (A, B) pode ser (e tipicamente é) diferente da real (P)
- permite continuar a executar aplicações de arquiteturas descontinuadas
- mais lenta, em uma ordem de magnitude (10x) que a execução nativa
- implica implementar, por software, toda a operação de um CPU

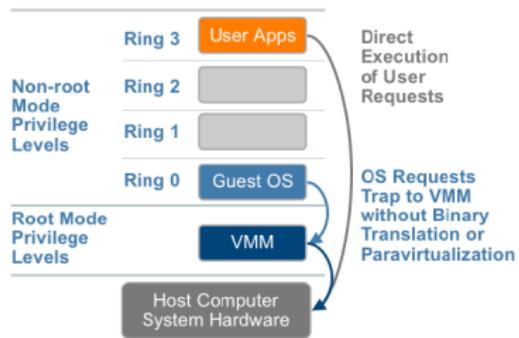
Virtualização de Sistemas (6/10)

Virtualização Total ou Nativa



- é semelhante à emulação, mas
 - o HW simulado (P) é coincidente com o HW real (P)
 - permite execução direta de instruções user-level de P
 - instruções privilegiadas não-virtualizáveis são traduzidas em sequências de outras, de resultado final equivalente
 - tira partido do modo de operação **multimode** dos CPUs, e de várias extensões à ISA para assistir (auxiliar) a virtualização (ver a seguir)

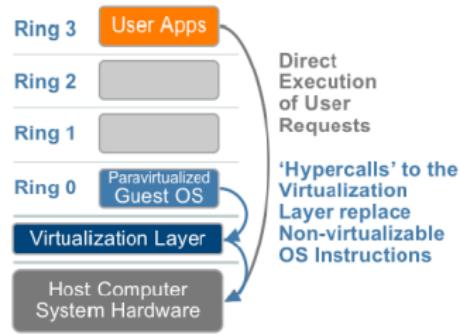
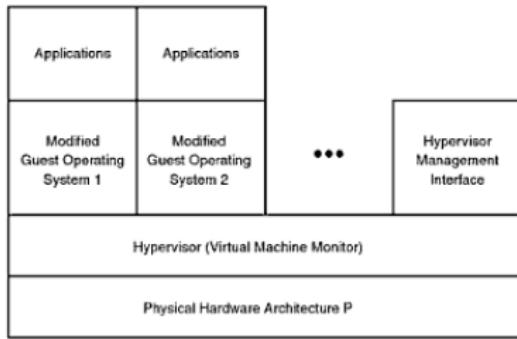
Técnicas de Virtualização - Virtualização Total ou Nativa (cont.)



- Virtualização Assistida por Hardware (arq. x86, 2005, 2006)
 - CPU virtualization (Intel VT-x e AMD-V)
 - otimiza a execução de instruções privilegiadas
 - recentemente: Nested Page Tables (AMD); Extended Page Tables (Intel); virtualização de tabelas de páginas (para memória virtual)
 - I/O MMU Virtualization (AMD-Vi e Intel VT-d)
 - permite acesso direto das VMs a placas de rede, gráficas, discos, etc.
 - técnica também conhecida por PCI passthrough

Virtualização de Sistemas (8/10)

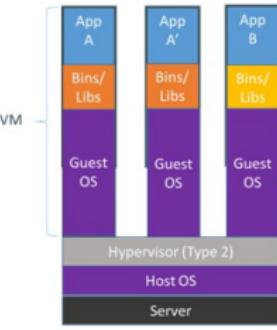
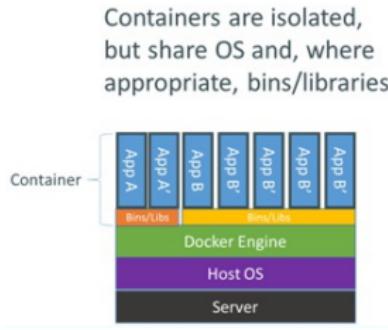
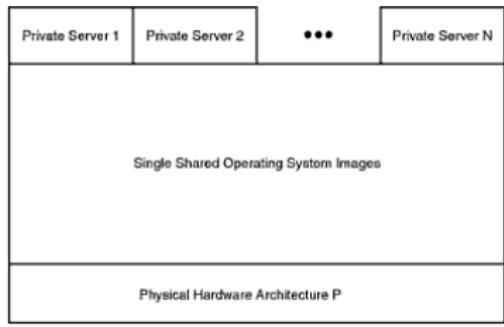
Técnicas de Virtualização - Para-Virtualização / Virtualização Assistida pelo SO



- em vez de apresentar aos *guests* uma cópia fiel da arquitetura física de base (P), o *hypervisor* fornece uma versão modificada da mesma
- o SO das VMs é modificado para executar chamadas ao *hypervisor* (*hypercalls*), em vez de invocar funcionalidades da arquitetura P
- traduz-se em níveis de desempenho melhorados face a outras técnicas

Virtualização de Sistemas (9/10)

Técnicas de Virtualização - Virtualização de Nível SO



- técnica também conhecida por *containerization* (de *containers*)
- não existem VMMs; em vez disso, um SO genérico aloja várias instâncias de SOs do mesmo tipo, mas executando em *user-level*
- na RAM, os *guests* partilham o kernel, bibliotecas e programas
- vantagem: elevado desempenho e escalabilidade (no nº de *guests*)
- desvantagem: menor grau de isolamento dos vários *guests* entre si
- não confundir c/ Virtualização Assistida pelo SO (Para-Virtualização)

Virtualização de Sistemas (10/10)

Plataformas de Virtualização

Implementation	Virtualization Type	Installation Type	License
Bochs	Emulation	Hosted	LGPL
QEMU	Emulation	Hosted	LGPL/GPL
VMware	Full Virtualization & Paravirtualization	Hosted and bare-metal	Proprietary
User Mode Linux (UML)	Paravirtualization	Hosted	GPL
Open VZ	OS Level	Bare-metal	GPL
Linux VServer	OS Level	Bare-metal	GPL
Xen	Paravirtualization or Full when using hardware extensions	Bare-metal	GPL
Parallels	Full Virtualization	Hosted	Proprietary
Microsoft	Full Virtualization	Hosted	Proprietary
z/VM	Full Virtualization	Hosted and bare-metal	Proprietary
KVM	Full Virtualization	Bare-metal	GPL
Solaris Containers	OS Level	Hosted	CDDL
BSD Jails	OS Level	Hosted	BSD

Implementação de uma syscall em Linux (exemplo) (1/2)

Implementação de uma primitiva em Linux: USER-LAND

- test program (compiled with “gcc test.c -I<path to absolute.h folder> -o test.exe”)

```
// test.c
#include <stdio.h>
#include <absolute.h>

main() {
    int a;
    scanf("%d", &a);
    printf("%d\n", absolute(a));
}
```

- header file absolute.h

```
// src/linux-4.4.0/absolute/absolute.h
#include <linux/kernel.h>
#include <sys/syscall.h>
#include <unistd.h>

unsigned int absolute(int i) {
    return syscall(XXX, i);
// XXX is the unique syscall integer id
}
```

Implementação de uma syscall em Linux (exemplo) (2/2)

Implementação de uma primitiva em Linux: KERNEL-LAND

- definition of the unique id of the primitive

```
// src/linux-4.4.0/arch/x86/entry/syscalls/syscall_64.tbl
...
326      common      absolute
...
...
```

- declaration of the primitive

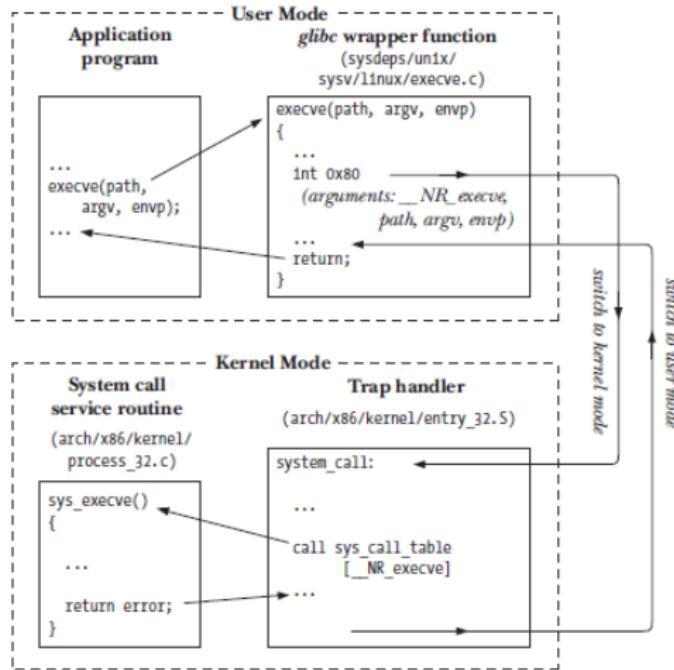
```
// src/linux-4.4.0/include/linux/syscalls.h
...
asmlinkage unsigned int sys_absolute(int);
...
```

- code of the primitive

```
// src/linux-4.4.0/absolute/absolute.c
#include <linux/kernel.h>

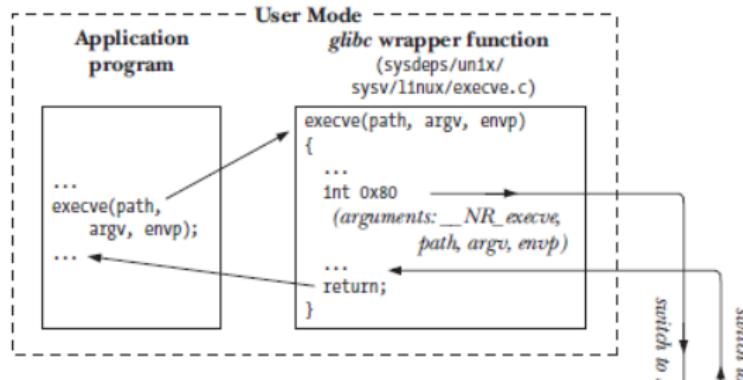
asmlinkage unsigned int sys_absolute(int i) {
//printk(KERN_DEBUG, "sys_absolute: received %d\n", i);
return(i>=0 ? i:(-i));
}
```

Anatomia da syscall execve em Linux (1/4)



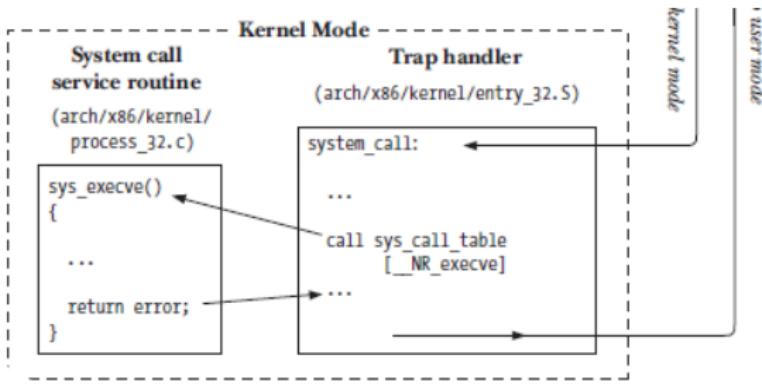
[Fonte: "The Linux Programming Interface - Section 3.1", Michael Kerrisk, No Starch, 2010]

Anatomia da syscall execve em Linux (2/4)



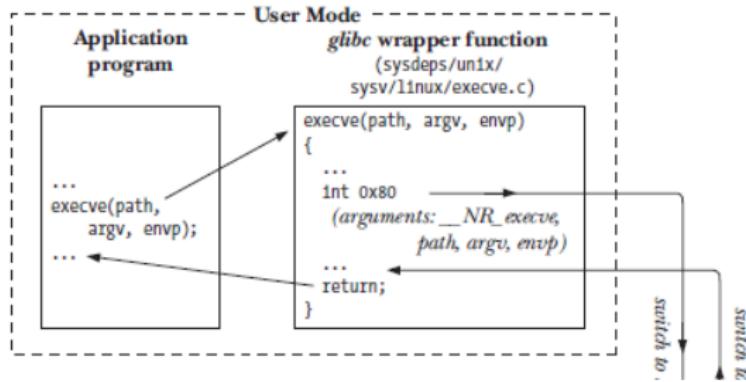
- 1- The application program makes a system call by invoking a wrapper function in the C library (e.g., `execve` in `glibc`).
- 2- The wrapper must make all of the system call arguments available to the system call trap-handling routine (which belongs to the kernel – see below). The arguments are passed to the wrapper via the stack, but the kernel expects them in specific registers. The wrapper copies the arguments to these registers.
- 3- Since all system calls enter the kernel in the same way, the kernel needs some method of identifying the system call. To permit this, the wrapper function copies the system call number (`__NR_execve = 11`) into a specific CPU register (`%eax`).
- 4- The wrapper function executes a *trap* machine instruction (`int 0x80`), which causes the processor to switch from user mode to kernel mode and execute code pointed to by location `0x80` (128 decimal) of the system's trap vector.

Anatomia da syscall execve em Linux (3/4)



- 5- In response to the trap to location 0x80, the kernel invokes its `system_call()` routine (located in the assembler file `arch/i386/entry.S`) to handle the trap. This handler:
- Saves register values onto the kernel stack;
 - Checks the validity of the system call number (`__NR_execve`);
 - Invokes the appropriate system call service routine, which is found by using the system call number (`__NR_execve`) to index a table of all system call service routines (the kernel variable `sys_call_table`). If the system call service routine has any arguments, it first checks their validity. Then the service routine performs the required task. Finally, the service routine returns a result status to the `system_call()` routine.
 - Restores register values from the kernel stack and places the system call return value on the stack.
 - Returns to the wrapper function, simultaneously returning the processor to user mode.

Anatomia da syscall execve em Linux (4/4)



- 6- If the return value of the system call service routine is an error, the wrapper function sets the global variable `errno` using this value. The wrapper function then returns to the caller, providing an integer return value indicating the success or failure of the system call.

Arranque do Sistema (1/2)

- Fases principais do processo de arranque (*booting*):
 - 1) localizar, carregar p/ memória e iniciar a execução do *kernel*; 2) este inicializa o hardware, 3) monta o sistema de ficheiros principal (*root*) e outros, 4) arranca diversos serviços e 5) lança o ecrã de início de sessão
- *bootstrap program(s) / boot loader(s)*: responsável(eis) pela fase 1
- *BIOS-based multistage boot process*:
 - BIOS = Basic Input Output System (firmware em EEPROM);
 - quando o computador é ligado, é executado código da BIOS; este verifica se os componentes essenciais do sistemas estão operacionais;
 - de seguida, carrega código do bloco inicial do disco de arranque e executa-o; este pode carregar e executar o *kernel* ou um menu de boot;
 - um menu de boot permite escolher um de entre vários SOs p/ arrancar
- *UEFI-based single stage boot process*
 - UEFI = Unified Extensible Firmware Interface
 - melhor suporte p/ sistemas de 64 bits, discos maiores, um só *boot loader/manager* (oferece um processo de arranque mais rápido)

Arranque do Sistema (2/2)

- Arranque de Sistemas Linux:
 - GRUB = Grand Unified Boot Loader; suporta um *multi-boot menu*; permite escolher diferentes *kernels*, afinar os seus parâmetros, arrancar outro SO (e.g., Windows instalado no mesmo computador que o Linux)
 - para poupar espaço na *partição de boot* e acelerar o arranque, é comum o *kernel* estar **compactado** (e.g., /boot/vmlinuz-4.4.0-59-generic)
 - durante o arranque, o GRUB cria um sistema de ficheiros temporário em RAM (initramfs), copiando para ele apenas os módulos e *drivers* do *kernel* necessários para poder suportar o sistema de ficheiros raíz (*root*)
 - mal o *kernel* arranca, passa a usar o sistema de ficheiros raíz, e localiza e carrega os restantes drivers e módulos necessários ao HW presente
 - depois, o *kernel* lança o processo systemd / init (com PID 1), e a partir deste, como descendentes, processos de vários serviços (web, BDs, etc.)
 - por fim, é apresentado o ecrã de início de sessão (CLI ou GUI), e o SO fica em "repouso", aguardando a ocorrência de eventos, que tratará
 - *single-user/recovery mode*: permite arrancar o SO numa configuração simplificada, para diagnosticar/corrigir erros, etc. (e.g., passwd reset)

REFERÊNCIAS

- "Operating System Concepts, 10th Ed.", Silberschatz & Galvin, Addison-Wesley, 2018: Cap. 1+2
- Interrupt Vector:
<https://www.sciencedirect.com/topics/engineering/interrupt-vector>
- Cpu Rings, Privilege and Protection:
<https://web.archive.org/web/20180713151101/https://manybutfinite.com/post/cpu-rings-privilege-and-protection/>
- System Calls Make the World Go Round:
<https://manybutfinite.com/post/system-calls/>
- "The Linux Programming Interface", Michael Kerrisk, No Starch, 2010: Section 3.1
- How Computers Boot Up:
<https://manybutfinite.com/post/how-computers-boot-up/>
- The Kernel Boot Process:
<https://web.archive.org/web/20180501083929/https://manybutfinite.com/post/kernel-boot-process/>
- "BCC Tracing Tools", <https://github.com/iovisor/bcc>

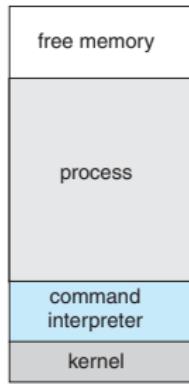
Unidade 2 - Processos e Threads

- 1 Modalidades de Exploração
- 2 Conceito de Processo
- 3 Operações sobre Processos
- 4 Escalonamento de Processos
- 5 Comunicação Inter-Processos
- 6 Comunicação Cliente-Servidor
- 7 Conceito de Thread
- 8 Modelos de Threading
- 9 Bibliotecas de Threading
- 10 Tópicos Extra

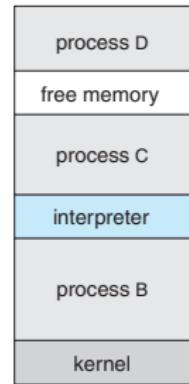
2.1 Modalidades de Exploração

Modalidades de Exploração (1/2)

- os primeiros sistemas de computação eram **monoprogramados**
 - **monoprogramação**: um só programa na memória (para além do SO)
 - subutilização do sistema: não permite sobreposição de computação e E/S
- posteriormente, surgiram os sistemas **multiprogramados**
 - **multiprogramação**: vários programas na memória (para além do SO)
 - rentabilização do sistema: permite sobreposição de computação e E/S
 - mas o SO tem de gerir a execução concorrente de vários programas



sist. monoprogramado



sist. multiprogramado

Modalidades de Exploração (2/2)

- sistema **mono-utilizador**: executa programas de um só utilizador
 - o sistema pode ser monoprogramado ou multiprogramado ...
- sistema **multi-utilizador**: executa programas de vários utilizadores
 - o sistema é multiprogramado (monoprogramação não faz sentido)
- **time-sharing**: comutação rápida entre programas de \neq s utilizadores
 - cria a ilusão de que cada utilizador dispõe da máquina só para si
 - permite a interação direta (local ou remota) com o sistema
- sistema **mono-núcleo**: tem um só núcleo de execução (núcleo = conjunto das unidades funcionais mínimas para executar uma sequência de instruções)
 - suporta mono/multi-programação, e mono/multi-utilização; mas o desempenho da multi-programação/utilização é relativamente limitado
- sistema **multi-núcleo**: tem vários núcleos de execução
 - o mais adequado a sistemas multi-programados e multi-utilizador

2.2 Conceito de Processo

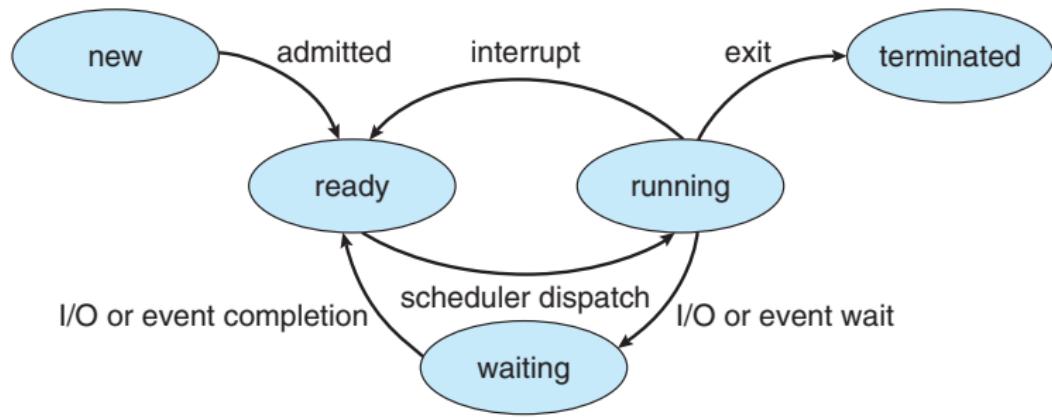
Conceito de Processo (1/6)

- a noção de **processo** deriva da necessidade em compartmentalizar e gerir os recursos necessários à execução concorrente de múltiplos programas
 - conceito que emerge naturalmente nos sistemas **multi-programados**
- um **programa** é uma sequência de instruções (com dados eventualmente pré-definidos), armazenada num ficheiro **executável**; a natureza das instruções e o formato do executável são específicos da arquitetura hospedeira e do SO alvo
 - um programa é uma entidade *passiva*
- um **processo** é um “*programa em execução*”: inclui a execução sequencial do programa, instrução a instrução, e todos os recursos necessários a essa execução
 - um processo é uma entidade *ativa*
 - um programa passa a processo quando é despoletada a sua execução
- alguns sistemas permitem que um processo i) dê origem a outros (e.g., via `fork`, ii) execute programas diferentes durante a sua vida (e.g., via `exec*`, iii) forneça um ambiente de execução para outro tipo de código (e.g., Java Virtual Machine)

Conceito de Processo (2/6)

Ciclo de Vida de um Processo

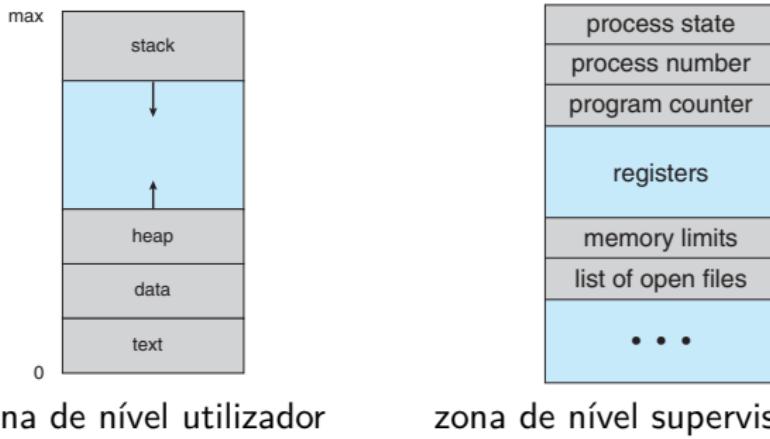
- o ciclo de vida (genérico) de um processo inclui vários estados
 - *novo*: o processo está a ser criado
 - *em execução*: as instruções do programa estão a ser executadas
 - *bloqueado*: o processo aguarda por um evento ou conclusão de E/S
 - *pronto*: o processo aguarda atribuição de CPU (escalonamento)
 - *concluído*: o processo terminou e libertou os seus recursos



Conceito de Processo (3/6)

Espaço de Memória de um Processo

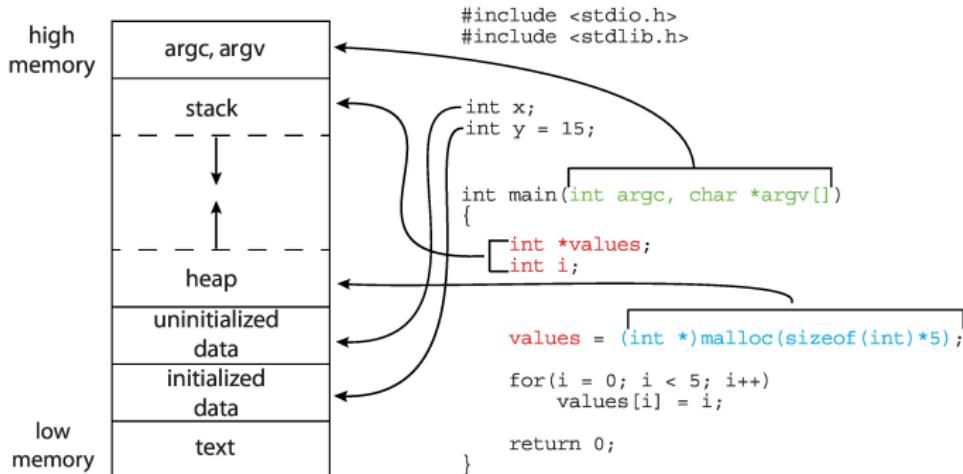
- a memória ocupada por um processo inclui duas zonas principais: i) uma acessível em *modo utilizador*, e ii) outra acessível só em *modo supervisor*



- complemento: ver <https://www.youtube.com/watch?v=7dLZRMDcY6c>
- papel da *stack*: quando se invoca uma função, guarda o seu *activation record* (registo de ativação), com parâmetros, variáveis locais e endereço de retorno
 - complemento: ver https://www.youtube.com/watch?v=XbZQ-EonR_I

Conceito de Processo (4/6)

Zona de Nível Utilizador - layout da memória de um programa em C



- o comando Linux `size` permite saber o espaço (em bytes) destas secções:

```
$ size exemplo.exe
```

text	data	bss	dec	hex	filename
1158	284	8	1450	5aa	exemplo.exe

- `text` = código; `data` = dados não inicializados; `bss (block started by symbol)` = dados inicializados; `dec, hex` = `text + data + bss`

Conceito de Processo (5/6)

Zona de Nível Supervisor - Bloco de Controlo de um Processo (PCB)

- a informação de cada processo, gerida em *modo supervisor*, concentra-se no *Bloco de Controlo do Processo*, compreendendo:
 - o estado do processo (pronto, em execução, bloqueado, ...)
 - o conteúdo dos registos da CPU (PC, SP, registos da ALU/FPU, ...)
 - informação de escalonamento da CPU (prioridade, fatia de tempo, ...)
 - inf. de gestão de memória (reg. base/limite, tab. de páginas/seg., ...)
 - inf. de *accounting* (id. do processo/job, tempo de CPU consumido, ...)
 - estado E/S (listas de disp. com pedidos pendentes, fichs. abertos, ...)

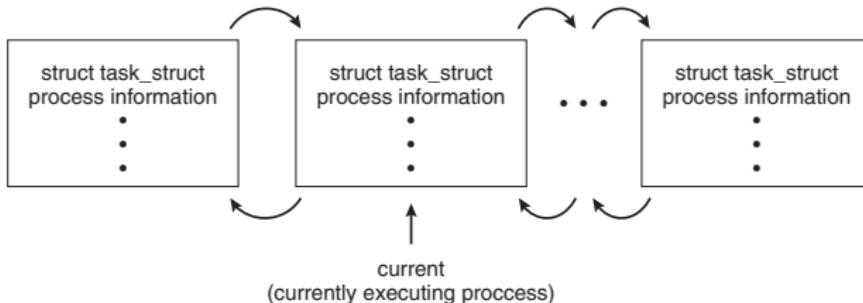
Conceito de Processo (6/6)

Bloco de Controlo de um Processo – Linux

- é uma estrutura C, task_struct, definida em <linux/sched.h>
(ver /usr/src/linux-headers-*/include/linux/sched.h)

```
...
char comm[TASK_COMM_LEN];      /* task command (executable name, excluding path) */
pid_t pid;                    /* process identifier */
long state;                   /* process state */
unsigned int time_slice        /* scheduling information */
struct task_struct *parent;   /* process parent */
struct list_head children;   /* process children */
struct files_struct *files;   /* open files information */
struct mm_struct *mm;         /* process address space */
void *stack;                  /* process stack */
int exit_code;                /* process exit code */
...
```

- lista de processos ativos: lista duplamente ligada de estruturas task_struct

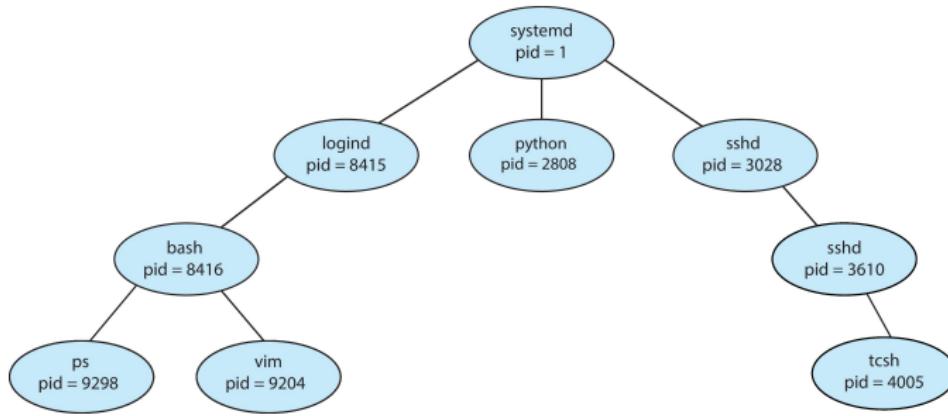


2.3 Operações sobre Processos

Operações sobre Processos (1/6)

Criação de Processos

- um processo *pai* cria um ou mais *filhos* que, por sua vez, poderão criar novos processos, formando assim uma árvore de processos



Exemplo de uma árvore de processos de um sistema Linux

- processo no topo da hierarquia (pid=1): init ou systemd
- comandos para listar os processos atuais em Linux: **ps**, **pstree**

Criação de Processos (cont.)

- **partilha de recursos**

- pai e filhos não partilham recursos
 - risco de sobrecarga do sistema ...
- pai e filhos partilham todos ou alguns dos recursos
 - riscos do acesso concorrente ...

- **execução**

- pai e filhos executam concorrentemente
- o pai aguarda que um ou mais filhos terminem

- **espaço de endereçamento**

- o filho duplica o do pai (mesmo programa e dados)
- o filho carrega um novo programa nesse espaço

Criação de Processos (UNIX)

- **chamadas ao sistema**

- através de fork, um processo pai cria um filho, que prosseguirá a execução do mesmo programa ou executará um programa diferente
- através de exec, um processo pode substituir o seu programa por outro
- através de system, um processo pode pedir a execução de um programa num processo separado, aguardando pela sua terminação

- **partilha de recursos**

- após fork, o filho herda do pai: a diretoria de trabalho, o estado do vetor de sinais, a máscara de criação de ficheiros, o segmento de código (partilhado) e de dados (partilhado enquanto possível - *copy-on-write*)
- após fork, o filho difere do pai: no PID, no segmento de dados (separado após *copy-on-write*), os contadores para alarmes sofrem *reset*
- após exec, o processo conserva: o PID, a diretoria de trabalho, a máscara de criação de ficheiros, trincos sobre ficheiros, os mesmos sinais já armadilhados, tempo que resta até à entrega de SIGALRM, ...

Operações sobre Processos (4/6)

Terminação de Processos

- **terminação normal**

- o processo pede explicitamente ao SO para removê-lo (`exit`)
- os recursos consumidos pelo processo são libertados pelo SO
- o pai pode receber dados de um filho que terminou (`wait`)

- **terminação anormal**

- processos do mesmo grupo podem terminar-se (`kill`)
- um processo pai pode terminar um filho porque
 - o filho excede os recursos alocados
 - a tarefa atribuída ao filho já não é necessária
 - o próprio pai está a terminar

- **variantes**

- UNIX: quando um pai termina, os filhos que subsistem não podem ficar órfãos ⇒ *adoção pelo init/systemd ou delegados (subreapers)*
- VMS: quando um pai termina, todos os filhos são automaticamente terminados ⇒ *terminação em cascata*

Terminação de Processos (UNIX)

- **um processo termina antes do seu pai**
 - se o pai executou wait, é notificado, via SIGCHLD, que o filho terminou, recuperando o estado de saída devolvido por exit
 - se o pai não executou wait, o filho é marcado como *zombie* ⇒ os seus recursos são libertados mas o estado de saída é conservado
- **um processo termina antes dos seus filhos**
 - os filhos ficarão órfãos (sem pai), logo com um PPID inválido ...
 - identificam-se todos os órfãos (ativos ou *zombies*) com aquele PPID
 - o processo *init/systemd* ou um *subreaper* “adota” esses filhos
 - os *zombies* serão terminados (periódica/, o pai adotivo invoca wait)
- **prevenção de filhos zombies**
 - ignorar explicitamente o sinal SIGCHLD antes de criar quaisquer filhos

Operações sobre Processos (6/6)

Windows vs Linux

```
#include <stdio.h>
#include <windows.h>

int main(VOID)
{
STARTUPINFO si;
PROCESS_INFORMATION pi;

/* allocate memory */
ZeroMemory(&si, sizeof(si));
si.cb = sizeof(si);
ZeroMemory(&pi, sizeof(pi));

/* create child process */
if (!CreateProcess(NULL, /* use command line */
"C:\\WINDOWS\\system32\\mspaint.exe", /* command */
NULL, /* don't inherit process handle */
NULL, /* don't inherit thread handle */
FALSE, /* disable handle inheritance */
0, /* no creation flags */
NULL, /* use parent's environment block */
NULL, /* use parent's existing directory */
&si,
&pi))
{
    fprintf(stderr, "Create Process Failed");
    return -1;
}
/* parent will wait for the child to complete */
WaitForSingleObject(pi.hProcess, INFINITE);
printf("Child Complete");

/* close handles */
CloseHandle(pi.hProcess);
CloseHandle(pi.hThread);
}
```

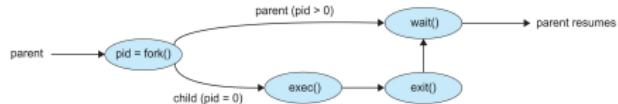
```
#include <sys/types.h>
#include <stdio.h>
#include <unistd.h>

int main()
{
pid_t pid;

/* fork a child process */
pid = fork();

if (pid < 0) { /* error occurred */
    fprintf(stderr, "Fork Failed");
    return 1;
}
else if (pid == 0) { /* child process */
    execp("/bin/ls","ls",NULL);
}
else { /* parent process */
    /* parent will wait for the child to complete */
    wait(NULL);
    printf("Child Complete");
}

return 0;
}
```



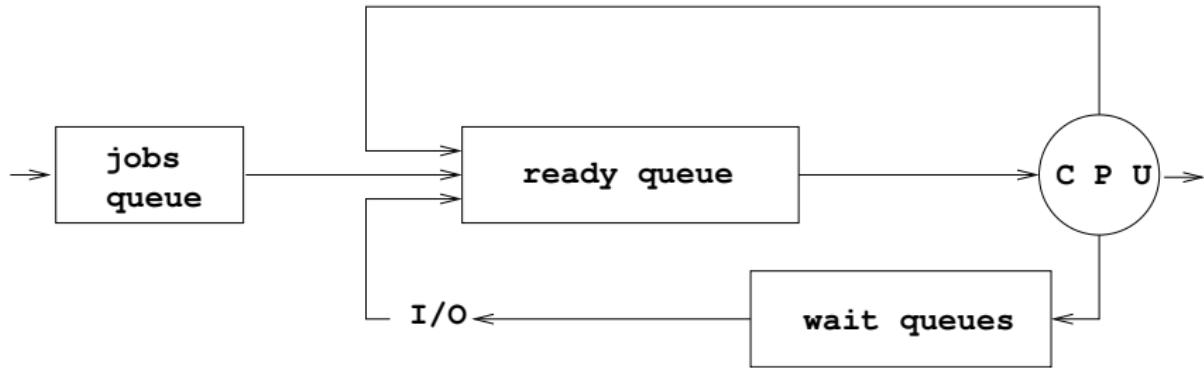
2.4 Escalonamento de Processos

Escalonamento de Processos (1/8)

- objetivo da **multiprogramação**: maximizar a utilização da CPU
 - e, simultaneamente, maximizar a utilização dos vários periféricos
 - um sistema **time-sharing** acrescenta ainda outro requisito: permitir o uso interativo, fornecendo a sensação de uso exclusivo do sistema
- para cumprir estes objetivos, o sistema operativo contempla:
 - **filas de escalonamento**: estruturas de dados que organizam processos no mesmo estado
 - **escalonadores de processos**: entidades que fazem transitar os processos entre filas (estados)

Escalonamento de Processos (2/8)

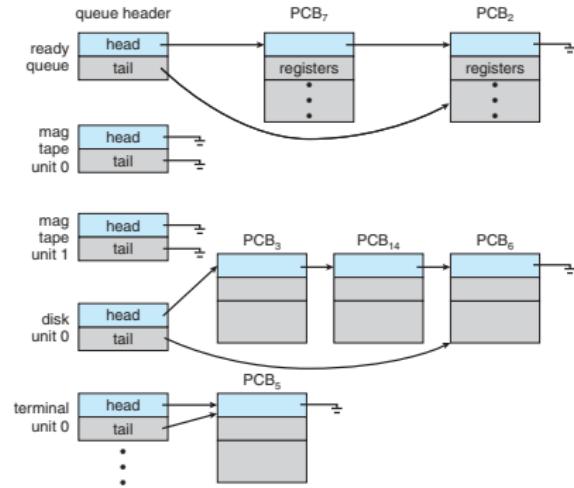
Filas de Escalonamento



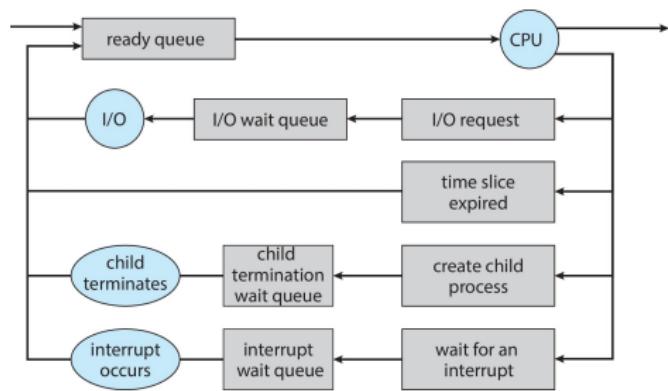
- **jobs queue** – processos *novos* e ainda não admitidos em RAM (obsoleto; apenas relevante de um ponto de vista histórico ...)
- **ready queue(s)** – fila(s) dos processos em RAM, *prontos* a executar
- **wait queue(s)** – fila(s) dos processos *bloqueados*, à espera de eventos E/S ou da execução de serviços privilegiados pelo SO

Escalonamento de Processos (3/8)

Filas de Escalonamento (cont.)



a) estrutura/implementação



b) transição entre diferentes filas

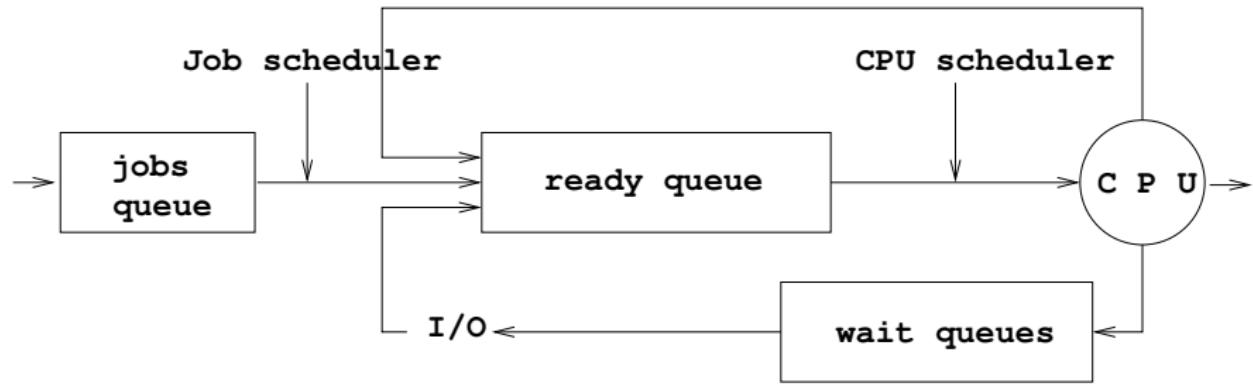
Escalonamento de Processos (4/8)

Escalonadores

- componentes do SO associados à gestão das filas de escalonamento
- **escalonador de longo termo** (*Job scheduler; obsoleto*) – de entre os novos processos, que se encontram temporariamente em memória secundária (*job queue*), seleciona aqueles a transitar para a memória principal (*ready queue*)
 - invocado pouco frequentemente (ordem dos segundos, minutos)
 - pode ser mais lento / ponderado a tomar as suas decisões
 - controla o *grau de multiprogramação* (m)
- **escalonador de curto termo** (*CPU scheduler*) – de entre os processos prontos a executar (*ready queue*), seleciona o próximo a quem atribuir CPU
 - invocado muito frequentemente (ordem dos milissegundos)
 - convém que seja muito rápido/eficiente a tomar as suas decisões

Escalonamento de Processos (5/8)

Escalonadores (cont.)



Escalonamento de Processos (6/8)

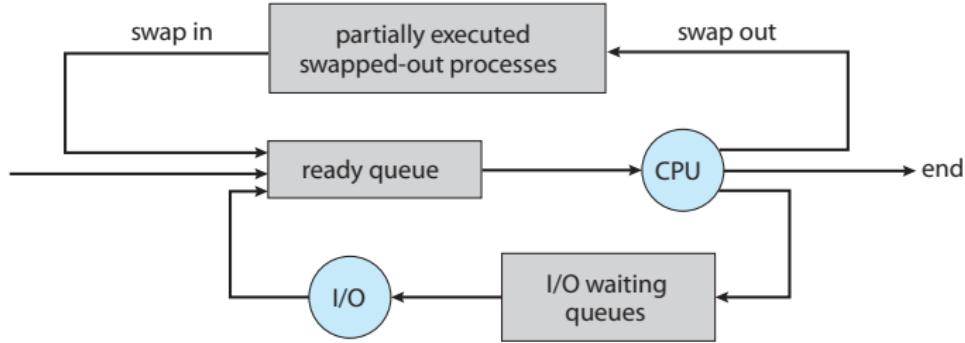
Escalonadores (cont.)

- **processos IO-bound**
 - consomem mais tempo em operações E/S do que processamento
 - *surtos de CPU* (períodos de ocupação) numerosos e reduzidos
- **processos CPU-bound**
 - consomem mais tempo em processamento do que em operações E/S
 - surtos de CPU longos, mas em número reduzido
- o escalonador de longo termo deve escolher um mix equilibrado de processos
 - maioria de processos *IO-bound*
⇒ *ready queue* vazia ⇒ baixa utilização da CPU
 - maioria de processos *CPU-bound*
⇒ *device queues* vazias ⇒ baixa utilização dos periféricos

Escalonamento de Processos (7/8)

Escalonadores (cont.)

- **escalonador de médio termo:** reduz o grau de multiprogramação via **swapping**
 - retirar um processo da memória principal para a secundária (*swap out*) pode ser necessário para 1) assegurar o equilíbrio de processos *IO/CPU-bound*, 2) libertar memória ou 3) reduzir a competição pela CPU
 - posteriormente, o processo pode ser migrado novamente para memória principal (*swap in*), e a sua execução retomada, quando re-escalonado



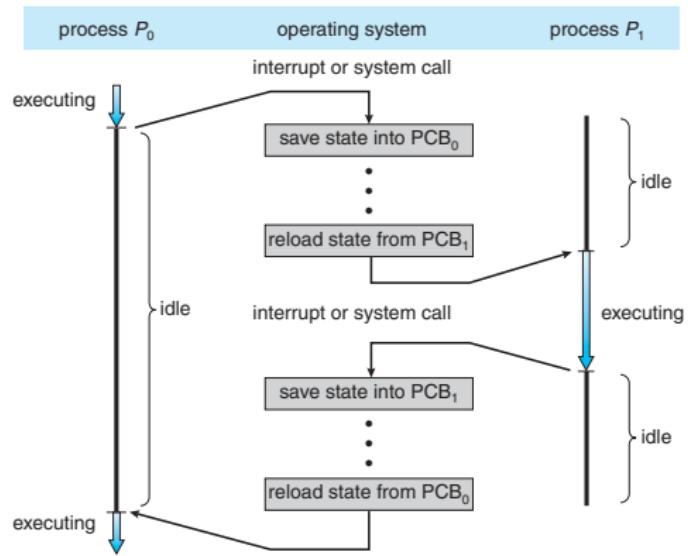
Escalonamento de Processos (8/8)

Comutação de Contexto

- quando a CPU comuta para outro processo, o sistema tem de guardar o estado do processo atual e carregar o estado do próximo processo

- o tempo consumido na comutação de processos é em si uma sobrecarga: o sistema não realiza trabalho "útil" enquanto se realiza a comutação

- o tempo consumido nesta atividade depende do suporte específico fornecido pelo hardware (e.g. guardar/carregar um conjunto de registo, janelas de registo)



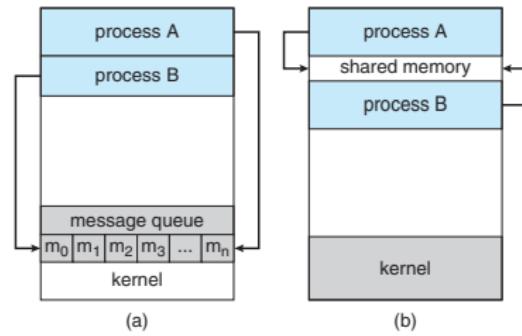
2.5 Comunicação Inter-Processos

Comunicação Inter-Processos (1/13)

Processos Cooperantes

- processos *independentes* não interferem entre si
- processos *cooperantes* podem afetar ou ser afetados por outros
- razões para a cooperação entre processos
 - **partilha de informação** (e.g., mecanismo copy+paste, acesso a BDs)
 - **aumento do desempenho** (e.g., execução paralela por várias CPUs)
 - **modularidade** (e.g., módulos organizados em *pipeline* ou grafo)
- mecanismos IPC: permitem *comunicação* e *sincronização* entre processos

- a) **Passagem de Mensagens**



- b) **Memória Partilhada**

Comunicação Inter-Processos (2/13)

Passagem de Mensagens vs Memória Partilhada

- ambos os modelos habitualmente disponíveis, mas adequados a ≠s casos
- **passagem de mensagens:**
 - comunicação através de *syscalls* de tipo `send(...)` e `receive(...)`
 - mais adequado (+ fácil de implementar) a ambientes distribuídos
 - mais adequado a pequenas trocas de dados, uma vez que implica invocações frequentes das *syscalls* envolvidas (menor eficiência)
- **memória partilhada:**
 - mais rápido (*) que passagem de mensagens: apenas são necessárias *syscalls* para criar a região partilhada; a partir daí, o acesso faz-se como a qualquer outra zona de memória (e.g., com apontadores)
 - (*) exceção – sistemas *multicore*: vários *cores* acedendo a memória partilhada provocarão a invalidação das *caches* uns dos outros ...
 - exige a programação explícita de toda a sincronização (ver slide 31)

Comunicação Inter-Processos (3/13)

Memória Partilhada

- tipicamente, um dos processos *cria* a zona partilhada, e outro(s) associa(m)-se à zona; cabe aos processos coordenarem-se para garantir a consistência dos dados
- exemplo com dois processos: **problema produtor-consumidor**
 - um processo produz informação consumida **concorrentemente** por outro
 - variante *bounded-buffer*: define um tamanho fixo para o *buffer* partilhado

```
// shared data:  
#define N ...  
typedef ... item_t; item_t buffer[N]; int in = out = 0;
```

```
// producer:  
item_t nextProduced;  
while (1) {  
    produceItem(&nextProduced);  
    while (((in + 1) % N) == out);  
    buffer[in] = nextProduced;  
    in = (in + 1) % N;  
}
```

```
// consumer:  
item_t nextConsumed;  
while (1) {  
    while (in == out);  
    nextConsumed = buffer[out];  
    out = (out + 1) % N;  
    consumeItem(&nextConsumed);  
}
```

Comunicação Inter-Processos (4/13)

Memória Partilhada - SysV Shared Memory

- exemplo de memória partilhada não-privada: processo criador+escritor

```
#define MAXSIZE 128
main() {
    int shmid; key_t key = 0x12345678;
    char *buffer;

    shmid = shmget(key, MAXSIZE*2, IPC_CREAT | 0666 );
    buffer = (char*)shmat(shmid, NULL, 0);

    strcpy(buffer, "first sentence");
    strcpy(buffer+MAXSIZE, "second sentence");
}
```

- **sincronização:** neste caso, o criador da zona é também o único escritor; mas logo após a criação, outros processos podem associar-se e aceder (leitura/escrita) à zona

Comunicação Inter-Processos (5/13)

Memória Partilhada - SysV Shared Memory (cont.)

- exemplo de memória partilhada não-privada: processo leitor+destruidor

```
#define MAXSIZE 128
main() {
    int shmid; key_t key = 0x12345678;
    char *buffer;

    shmid = shmget(key, MAXSIZE*2, 0666);
    buffer = (char*)shmat(shmid, NULL, 0);

    printf("%s\n", buffer);

    printf("%s\n", buffer+MAXSIZE);

    shmctl(shmid, IPC_RMID, NULL);
}
```

– sincronização: como é que o leitor “sabe” que é seguro aceder à memória partilhada, ou seja, que o escritor já lá depositou conteúdo ? neste caso, o problema resolve-se executando o leitor depois do escritor ... e outros casos ?

Passagem de Mensagens

- comunicação através de *syscalls* adequadas (e.g., `send(mensagem)`, `receive(mensagem)`), sem recurso explícito a variáveis partilhadas ...
- adequado a ambientes distribuídos, mas também pode ser usado entre processos na mesma máquina (rentabiliza o esforço de programação !)
- **dimensão das mensagens:** tamanho fixo (variável) simplifica (complica) a implementação, mas dificulta (simplifica) a utilização / programação
- **canal de comunicação:**
 - se P e Q pretendem trocar mensagens, é necessário um **canal**
 - *implementação física:* memória partilhada, barramento, rede, ...
 - *implementação lógica / propriedades lógicas:*
 - comunicação direta ou indireta ?
 - comunicação síncrona ou assíncrona ?
 - bufferização automática ou explícita ?

Comunicação Inter-Processos (7/13)

Passagem de Mensagens - Nomeação: Comunicação Direta

- processos que querem comunicar têm de ser capazes de se identificar
- **variante simétrica:** os processos têm que se nomear explicitamente
 - $\text{send}(P, \text{mensagem})$ – enviar uma mensagem ao processo P
 - $\text{receive}(Q, \text{mensagem})$ – receber uma mensagem do processo Q
- exemplo:
 - produtor–consumidor (com comunicação **blockante** – ver slide 39)

```
// producer:  
item nextProduced;  
while (1) {  
    produceItem(&nextProduced);  
    send(consumer,&nextProduced);  
}
```

```
// consumer:  
item nextConsumed;  
while (1) {  
    receive(producer,&nextConsumed);  
    consumeItem(&nextConsumed);  
}
```

Passagem de Mensagens - Nomeação: Comunicação Direta (cont.)

- **variante assimétrica:** só o remetente nomeia o destinatário
 - $\text{send}(P, \text{mensagem})$ – enviar uma mensagem ao processo P
 - $\text{receive}(id, \text{mensagem})$ – receber uma mensagem de qualquer processo, guardando em id o identificador desse processo
- propriedades do canal de comunicação
 - é estabelecida automaticamente uma ligação entre cada par de processos comunicantes (apenas necessitam da identificação)
 - uma ligação é associada a exatamente um par de processos
 - entre cada par existe exatamente uma ligação
- desvantagem comum a ambas as variantes: fraca modularidade (mudar os identificadores P e/ou Q implica recompilar o código)

Passagem de Mensagens - Nomeação: Comunicação Indireta

- as mensagens são trocadas através de *mailboxes* (*caixas de correio/portos*)
 - *send (A, mensagem)* - enviar uma mensagem para a *mailbox A*
 - *receive (A, mensagem)* - receber uma mensagem da *mailbox A*
 - cada *mailbox* tem um identificador único
 - os processos podem comunicar apenas se partilharem uma *mailbox*
 - os processos podem comunicar através de várias *mailboxes*
- propriedades do canal de comunicação
 - a ligação é estabelecida apenas se os processos partilham uma *mailbox*
 - uma ligação / *mailbox* pode ser associada a mais de dois processos
 - dois processos podem partilhar mais de uma ligação / *mailbox*

Passagem de Mensagens - Nomeação: Comunicação Indireta (cont.)

- partilha de *mailboxes*
 - problema
 - P_1, P_2 e P_3 partilham a *mailbox A*
 - P_1 executa `send(A, mensagem)`
 - P_2 e P_3 executam ambos `receive(A, mensagem)`
 - quem fica com a mensagem?
 - soluções (alternativas)
 - uma ligação só poderá ser associada a dois processos no máximo
 - apenas um processo pode executar, de cada vez, `receive`
 - o sistema seleciona um receptor e notifica o emissor dessa escolha
- operações sobre *mailboxes*
 - criar/reservar uma *mailbox* (o criador é o *dono*)
 - enviar e receber mensagens através da *mailbox*
 - remover/libertar a *mailbox*

Passagem de Mensagens - Sincronização

- envio bloqueante/síncrono:
 - o emissor espera (bloqueia) até que a mensagem seja entregue no recetor (processo ou mailbox)
- envio não-bloqueante/assíncrono:
 - o emissor submete o pedido de envio e prossegue a execução sem esperar (bloquear) pela confirmação de entrega
- receção bloqueante/síncrona:
 - o recetor bloqueia até que esteja disponível uma mensagem
- receção não-bloqueante/assíncrona:
 - o recetor recupera uma mensagem válida ou vazia
- **rendezvous** entre emissor e recetor:
 - quando o envio e a receção são ambas bloqueantes

Passagem de Mensagens - Bufferização

- uma ligação comporta um *número máximo de mensagens* ...
- ... correspondente à *capacidade da fila* associada à ligação
- *implementação da fila de mensagens*:
 - capacidade zero (0 mensagens) – o emissor tem de esperar que o recetor receba a mensagem (processo designado por *rendezvous*)
 - capacidade limitada (*bounded*) – dimensão finita de n mensagens em espera; o emissor tem de esperar se a ligação estiver cheia
 - capacidade ilimitada – tamanho infinito; o emissor nunca espera
- para filas de capacidade não-zero, a confirmação da receção da mensagem envolve comunicações adicionais; por exemplo:
 - o emissor P executa
“send (Q , *mensagem*); receive (Q , *mensagem*)”
 - o recetor Q executa
“receive (P , *mensagem*); send (P , *ACKNOWLEDGE*)”

Comunicação Inter-Processos (13/13)

Passagem de Mensagens – Exemplo: Pipes sem Nome

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

#define BUFFER_SIZE 25
#define READ_END 0
#define WRITE_END 1

int main(void)
{
    char write_msg[BUFFER_SIZE] = "Greetings";
    char read_msg[BUFFER_SIZE];
    int fd[2];
    pid_t pid;

    /* create the pipe */
    if (pipe(fd) == -1) {
        fprintf(stderr, "Pipe failed");
        return 1;
    }

    /* fork a child process */
    pid = fork();

    if (pid < 0) { /* error occurred */
        fprintf(stderr, "Fork Failed");
        return 1;
    }

    if (pid > 0) { /* parent process */
        /* close the unused end of the pipe */
        close(fd[READ_END]);

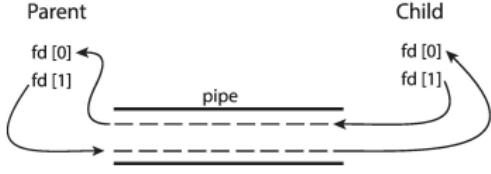
        /* write to the pipe */
        write(fd[WRITE_END], write_msg, strlen(write_msg)+1);

        /* close the write end of the pipe */
        close(fd[WRITE_END]);
    } else { /* child process */
        /* close the unused end of the pipe */
        close(fd[WRITE_END]);

        /* read from the pipe */
        read(fd[READ_END], read_msg, BUFFER_SIZE);
        printf("read %s", read_msg);

        /* close the read end of the pipe */
        close(fd[READ_END]);
    }
}

return 0;
```

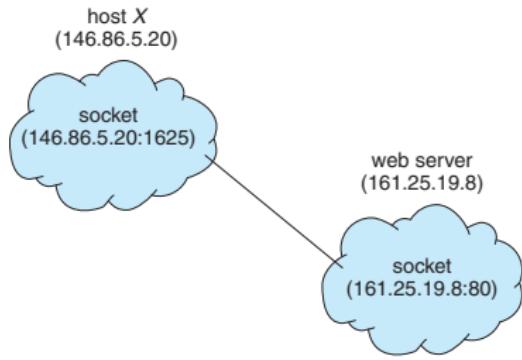


- canal = kernel buffer; comunicação indireta; receção síncrona (ler de um pipe vazio bloqueia o leitor); emissão assíncrona exceto se buffer cheio (escrever num pipe cheio bloqueia o escritor)

2.6 Comunicação Cliente-Servidor

Comunicação Cliente-Servidor (1/3)

- a utilização dos mecanismos IPC é limitada à mesma máquina
- processos em máquinas diferentes comunicam através de outros mecanismos (que podem ser usados também entre processos situados na mesma máquina)
- *Sockets*
 - definem extremos (*endpoints*) de uma transação TCP/IP
 - representados, cada um, por um par \langle endereço IP, porto \rangle
 - suportam comunicação *connectionless* e *connection-oriented*
 - servidores: um processo por “conexão”, atendimento *event-driven*
 - comunicação eficiente, mas de baixo nível; dados não estruturados



Comunicação Cliente-Servidor (2/3)

Sockets BSD – Exemplo: Cliente-Servidor TCP

```
#include "inet.h"
int main() // a TCP server
{
    int sockfd, newsockfd; socklen_t clilen;
    struct sockaddr_in cli_addr, serv_addr;
    char buffer[MAXLINE];

    // create a TCP socket
    sockfd=socket(AF_INET, SOCK_STREAM, 0);

    // bind the socket to a local port
    bzero(&serv_addr, sizeof(serv_addr));
    serv_addr.sin_family = AF_INET;
    serv_addr.sin_addr.s_addr = htonl(INADDR_ANY);
    serv_addr.sin_port = htons(SERV_TCP_PORT);
    bind(sockfd, (struct sockaddr *)&serv_addr, sizeof(serv_addr));

    // set max pending connections (5)
    listen(sockfd, 5);

    while(1) {
        // wait for connection
        clilen=sizeof(cli_addr);
        newsockfd=accept(sockfd, (struct sockaddr*)&cli_addr, &clilen);

        // fork a child to deal with the request
        switch(fork()) {
            case 0: // CHILD
                close(sockfd);
                printf("Client Address: %s\n", inet_ntoa(cli_addr.sin_addr));
                printf("Client Port: %d\n", ntohs(cli_addr.sin_port));
                read(newsockfd, buffer, MAXLINE);
                printf("Client Data: %s\n", buffer);
                close(newsockfd);
                exit(0);
            default: // PARENT
                close(newsockfd);
        }
    }
}

// inet.h
#include <stdio.h>
#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <arpa/inet.h>
#include <unistd.h>
#include <string.h>
#include <stdlib.h>
#define SERV_TCP_PORT 7564
#define SERV_HOST_ADDR "127.0.0.1"
#define MAXLINE 512

#include "inet.h"
int main(int argc, char **argv) // a TCP client
{
    int sockfd;
    struct sockaddr_in serv_addr;

    // create a TCP socket
    sockfd=socket(AF_INET, SOCK_STREAM, 0);

    // set server address and port
    bzero(&serv_addr, sizeof(serv_addr));
    serv_addr.sin_family = AF_INET;
    serv_addr.sin_addr.s_addr = inet_addr(SERV_HOST_ADDR);
    serv_addr.sin_port = htons(SERV_TCP_PORT);

    // connect to server
    connect(sockfd, (struct sockaddr *)&serv_addr, sizeof(serv_addr));

    // send message
    write(sockfd, argv[1], strlen(argv[1])+1);

    // close connection
    close(sockfd);

    return(0);
}
```

- canal = rede; comunicação indireta (porta SERV_TCP_PORT); receção síncrona (servidor bloqueado esperando conexões); envio síncrono (cliente conecta ao servidor); cenário rendezvous

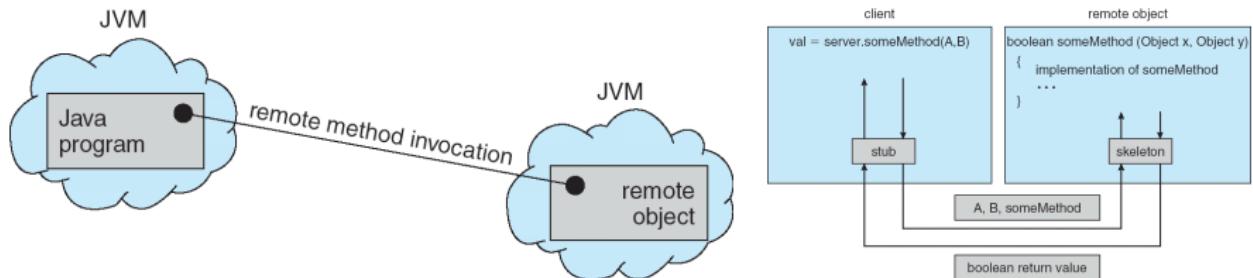
Comunicação Cliente-Servidor (3/3)

- *Remote Procedure Calls* (Sun RPCs)

- invocação remota ≈ local (maior abstração que os *sockets*)
- *stub* local: localiza servidor, constrói o pedido e envia-o
- *stub* remoto: interpreta o pedido e executa o procedimento
- pedidos e respostas são representados num formato neutro (XDR)
- exemplo de aplicação: Network File System (NFS), da Sun

- *Remote Method Invocation* (Java RMI)

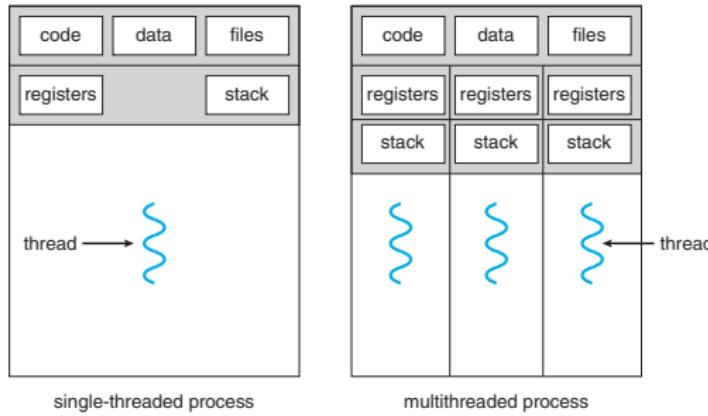
- ≈ Sun RPCs, mas para interações entre Java Virtual Machines
- Sun RPCs adequados a programação procedural; o Java é OO ...
- Java RMI permite invocar métodos de objetos remotos
- Java RMI permite passar objetos como parâmetros



2.7 Conceito de Thread

Conceito de Thread (1/4)

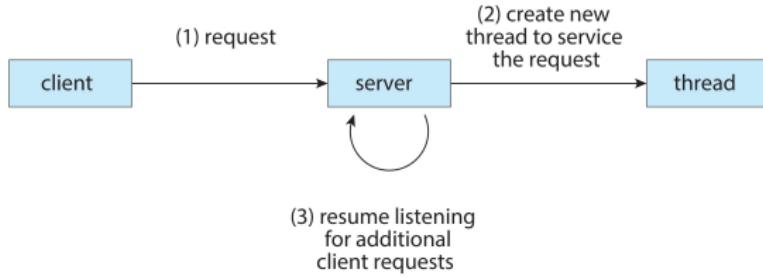
- **thread** (Fio-de-execução, Processo Leve): unidade básica de utilização de CPU
 - comprehende um trajeto no código e os recursos mínimos para o efetuar
 - recursos específicos de um *thread*: registos (PC, ...), pilha (*stack*), TID
- um processo tradicional (ou Processo Pesado) inclui pelo menos um *thread* – o *thread* principal; nos programas em C, este *thread* é o que executa a função `main`
 - um processo pode ter um ou vários *threads* (processo *single/multithreaded*)
 - *threads* do mesmo processo partilham o seu PID, segmento de código, dados globais e stack, tabela de ficheiros abertos, manuseadores de sinais, etc.



Conceito de Thread (2/4)

Motivação e Exemplos

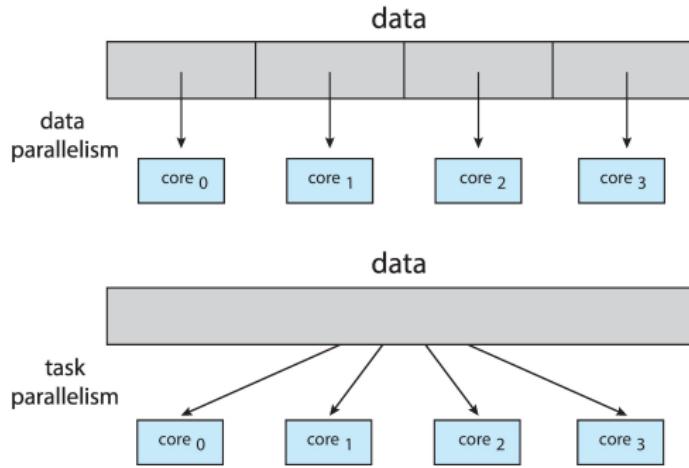
- hoje em dia, a maior parte das aplicações são *multithreaded* (incluindo o SO), contribuindo para tirar pleno partido das arquiteturas multi-núcleo dos CPUs
- uma aplicação de processamento de imagem (criar miniatura, aplicar um filtro, etc.) pode usar ≥ 1 *threads* por imagem, por conjunto de imagens, etc.
- um processador de texto pode ter *threads* separados para visualizar o documento, editar o documento, responder a toques de teclado e a acções do rato, verificar a ortografia e a gramática em tempo real, sugerir conteúdo (assistente IA), etc.
- um navegador web pode usar um *thread* separado por cada aba (ou, na mesma aba, um *thread* para mostrar a página, outro *thread* para aceder à rede, etc.)
- num servidor web um *thread* aguarda pedidos e cria um novo *thread* por pedido



Conceito de Thread (3/4)

Motivação e Exemplos

- um problema computacionalmente pesado pode-se resolver mais rapidamente partindo-o em sub-problemas independentes (ou pouco dependentes) e usando 1 (ou +) thread(s) para resolver cada sub-problema (**computação paralela**)



Conceito de Thread (4/4)

Benefícios/Vantagens

- + : menor consumo de recursos (menos estado) que os processos
(n threads do mesmo processo consomem menos recursos que n clones do processo)
- + : mais rápidos de criar (menos estado) que os processos tradicionais
- + : alternar a CPU entre *threads* é mais rápido (menos estado) que entre processos
- + : facilitam a partilha de memória (dispensam mecanismos de partilha explícita)
- + : mais responsividade (parte de uma aplicação pode continuar a funcionar, estando a outra parte bloqueada aguardando por eventos - importante em apps com GUI)
- + : mais escalabilidade (mecanismo mais eficiente para aproveitar os CPUs *multicore*)

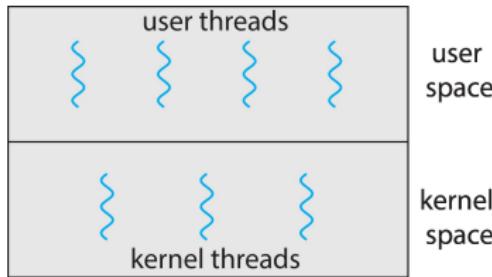
Problemas/Desvantagens

- : não se traduzem necessariamente em ganhos (globais) de desempenho ...
- : o acesso concorrente a dados partilhados pode originar bugs de difícil deteção
(mesmo utilizando mecanismos de sincronização e exclusão mútua)
- : nem todas as linguagens/ambientes/ferramentas suportam convenientemente *threads*
(e.g., depurar programas *multithreaded* carece de *debuggers* com suporte apropriado)

2.8 Modelos de Threading

Modelos de Threading (1/4)

- até agora assumiu-se que um *thread* é uma facilidade implementada pelo SO (**kernel thread**), exposta às aplicações através de *syscalls*
- no entanto, os *threads* normalmente utilizados pelas aplicações são abstrações criadas por bibliotecas acima do kernel (**user threads**)
- como mapear **user threads** (virtuais) em **kernel threads** (reais) ?

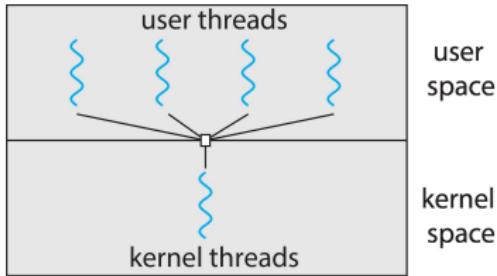


- modelo **muitos-para-um**
- modelo **um-para-um**
- modelo **muitos-para-muitos**
- modelo de **2 níveis**

Modelos de Threading (2/4)

Modelo Muitos-para-Um

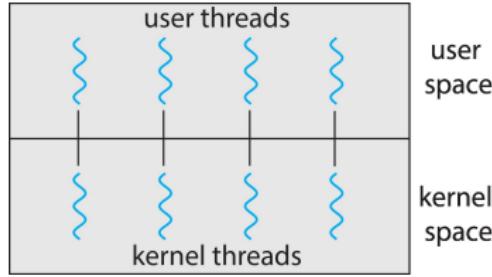
- mapeia muitos *user threads* em um *kernel thread*
- gestão eficiente dos *user threads* por biblioteca em *user space* (a alternância entre *user threads* não implica comutação de contexto)
- processo inteiro bloqueia se um *thread* invocar *syscall* bloqueante
- apenas há execução concorrente (não paralela) dos vários *threads*
- exemplo: biblioteca Green Threads usada pelo Java no SO Solaris



Modelos de Threading (3/4)

Modelo Um-para-Um

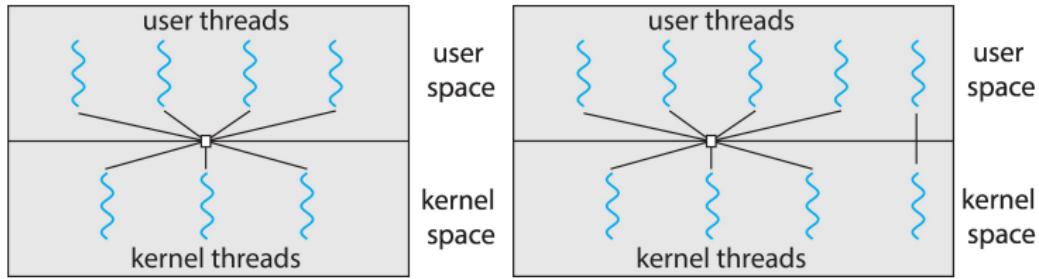
- mapeia cada *user thread* em um *kernel thread*
- vs modelo muitos-para-um:
 - suporta mais concorrência/paralelismo
 - implica criar mais *kernel threads*
(em demasia podem ser prejudiciais, devido a sobrecarga de recursos)
- exemplo: Windows e Linux, modelo acessível via biblioteca Pthreads



Modelos de Threading (4/4)

Modelo Muitos-para-Muitos

- mapeia N *user threads* em $M \leq N$ *kernel threads*
 - o valor de M pode variar em função da aplicação ou do CPU
- (+) mais flexível que os modelos anteriores:
 - as aplicações podem lançar o num. de threads que consideram desejável (N),
 - com a garantia de que algumas (até M) serão executadas em paralelo,
 - e de que a invocação de uma *syscall* bloqueante não bloqueia todo o processo
 - variante: **Modelo de 2 Níveis** (permite Um-para-Um para alguns *threads*)
- (-) mais difícil de implementar; com cada vez mais núcleos por CPU, é menos importante limitar o número de *kernel threads*



2.9 Bibliotecas de Threading

Bibliotecas de Threading (1/3)

- **biblioteca de threading**: expõe API para criação e gestão de threads

- **biblioteca user-level**: código e estruturas de dados em *user-space*; invocar uma função da API implica invocar uma função implementada em *user-space*
- **biblioteca kernel-level**: código e estruturas de dados no *kernel*; invocar uma função da API implica invocar uma chamada ao sistema
- principais bibliotecas de threading explícito: POSIX, Windows e Java threads

- **threading síncrono vs threading assíncrono**

- **threading síncrono**: logo após a criação de um *thread* "filho", o *thread* "pai" aguarda que esse "filho" termine, e só depois prossegue (se forem criados N *threads* "filhos", o *thread* "pai" aguarda que todos terminem)
- **threading assíncrono**: logo após a criação de um *thread* "filho", o *thread* "pai" prossegue a sua execução, sem esperar que o "filho" termine (ou, se forem criados N *threads* "filhos", o *thread* "pai" não aguarda por nenhum)

Bibliotecas de Threading (2/3)

POSIX threads (Pthreads)

- POSIX standard (IEEE 1003.1c) que especifica uma API de threads
- API de threads por omissão em sistemas UNIX, Linux e MacOS
- implementação disponível via biblioteca *user-level* ou *kernel-level*
- dados globais (fora de funções) são visíveis a todos os threads

Windows threads

- disponibilizadas por uma biblioteca *kernel-level*
- dados globais (fora de funções) são visíveis a todos os threads
- workflow e API tem algumas semelhanças com as Pthreads

Java threads

- disponibilizadas por uma biblioteca *user-level*
- implementadas com base na API preferencial do SO subjacente
- uma vez que o Java não suporta dados globais nativos, a partilha de dados entre diferentes threads tem de ser programada explicitamente

Bibliotecas de Threading (3/3)

POSIX Threads (exemplo)

```
#include <stdio.h>
#include <stdlib.h>
#include <pthread.h>
#define NUM_WORKERS 5

long GLOBAL_squares[NUM_WORKERS];

void *aThread(void *arg)
{
    long tid=(long)arg;

    GLOBAL_squares[tid]=tid*tid;
    printf("thread %ld: square = %lu\n", tid, GLOBAL_squares[tid]);
    pthread_exit(NULL);
}

int main()
{
    pthread_t threads[NUM_WORKERS];
    long t, sum=0;

    for(t=0;t<NUM_WORKERS;t++){
        printf("thread main: creating thread %ld\n", t);
        pthread_create(&threads[t], NULL, aThread, (void *)t);
    }

    for(t=0; t<NUM_WORKERS; t++) {
        pthread_join(threads[t], NULL);
        printf("thread main: joined with thread %ld\n", t);
        sum += GLOBAL_squares[t];
    }

    printf("thread main: sum = %lu: exiting\n", sum);
    pthread_exit(NULL);
}
```

thread main: creating thread 0
thread main: creating thread 1
thread 0: square = 0
thread 1: square = 1
thread main: creating thread 2
thread main: creating thread 3
thread 2: square = 4
thread main: creating thread 4
thread 3: square = 9
thread main: joined with thread 0
thread main: joined with thread 1
thread main: joined with thread 2
thread main: joined with thread 3
thread 4: square = 16
thread main: joined with thread 4
thread main: sum = 30: exiting

(compilar com: gcc pthreads-example.c -o pthreads-example.exe -lpthread)

Tópicos Extra

Comunicação Inter-Processos

Exemplos de Sistemas IPC - UNIX System V Message Queues

- exemplo de emissor

```
#define MAXSIZE 128
main() {
    int msqid; key_t key = 0x12345678; size_t bufferlen;
    struct { long mtype; char mtext[MAXSIZE]; } buffer;

    msqid = msgget(key, IPC_CREAT | 0666);

    buffer.mtype = 1;
    strcpy(buffer.mtext, "first message");
    bufferlen = strlen(buffer.mtext) + 1 ;
    msgsnd(msqid, &buffer, bufferlen, IPC_NOWAIT);

    buffer.mtype = 1;
    strcpy(buffer.mtext, "second message");
    bufferlen = strlen(buffer.mtext) + 1 ;
    msgsnd(msqid, &buffer, bufferlen, IPC_NOWAIT);
}
```

- sincronização: neste caso (IPC_NOWAIT), a emissão é não-bloqueante (assíncrona)

Comunicação Inter-Processos

Exemplos de Sistemas IPC - UNIX System V Message Queues (cont.)

- exemplo de receptor

```
#define MAXSIZE 128
main() {
    int msqid; key_t key = 0x12345678; long type = 1;
    struct { long mtype; char mtext[MAXSIZE]; } buffer;

    msqid = msgget(key, 0666);

    msgrcv(msqid, &buffer, MAXSIZE, type, 0);
    printf("%s\n", buffer.mtext);

    msgrcv(msqid, &buffer, MAXSIZE, type, 0);
    printf("%s\n", buffer.mtext);

    msgctl(msqid, IPC_RMID, NULL);
}
```

- **sincronização:** por omissão, a receção é bloqueante (síncrona)

REFERÊNCIAS

- "Operating System Concepts, 10th Ed.", Silberschatz & Galvin, Addison-Wesley, 2018: Capítulos 3+4
- "Fundamentos de Sistemas Operacionais, 9a Ed.", Silberschatz, Galvin & Gagne, LTC, 2015: Capítulos 3+4
- "Anatomy of a Program in Memory", Gustavo Duarte, 2009,
<https://web.archive.org/web/20180206141815/https://manybutfinite.com/post/anatomy-of-a-program-in-memory/>
- "Anatomy of Linux process management", M. Tim Jones, 2008,
<https://developer.ibm.com/tutorials/l-linux-process-management/>