Sistemas Operativos I

LESI + LMCC

Grupo de Sistemas Distribuídos
http://gsd.di.uminho.pt

As más notícias

- É uma cadeira de "engenharia": temos de
  - Perceber os compromissos => usar a "massa cinzenta"
  - Sujar as mãos na "massa" => programar, configurar
- Muitos alunos encravam em deficiências passadas, por exemplo dificuldade de raciocinar, concepção de algoritmos, programação, como funciona um computador...
- Esta cadeira engana muito: usar/"manusear" um SO não chega

Boas notícias

- Apesar da "mística" que rodeia os sistemas operativos, basta um pouco de estudo e bom senso para fazer a cadeira
- Quem fizer a cadeira certamente terá ultrapassado grande parte das deficiências anteriores!

Enquadramento

- Agora:
  - CSI → SO1 → SO II → SOD1 → SOD2 → AS1 → AS2 → SED
- Em breve:
  - LMCC foi restruturada
  - LESI em aprovação

Equipa Docente

- Responsável pela disciplina + aulas teóricas
  - Francisco Soares de Moura (fsm@di.uminho.pt)
- Aulas práticas
  - Rui Oliveira, José Orlando Pereira, Vitor Fonte, José Pedro Oliveira, ...
- Horário de atendimento a definir

Programa

- Recapitulação de conceitos de programação de sistemas
- Noções de programação concorrente
- Gestão de processos, memória, periféricos
- Mãos na massa:
  - Aulas práticas em laboratório: Linux
  - Programação de "baixo nível": C, syscalls, libs…

Bibliografia recomendada

- Sebenta de Sistemas Operativos (em construção…)
- A. Silberschatz et al., *Applied Operating System Concepts*, John Wiley & Sons, 2000.

OU

- A. S. Tanenbaum, *Modern Operating Systems*, 2nd edition, Prentice Hall, 2001.

Bibliografia recomendada

- fsm 2004, ***Vou fazer Sistemas Operativos***
- www.google.com
  - Introduction to operating systems

  …
- www.gildot.org, www.slashdot.org …

Bibliografia Adicional

- R. Stevens, *Advanced Programming in the Unix Environment*, Addison Wesley, 1990.
- Diversos artigos sobre sistemas operativos, a disponibilizar na página da cadeira ou a pesquisar na Internet.
- Manuais do sistema operativo, FAQs, código fonte do Linux, …

Transparências

- (Progressivamente) disponíveis em:

  http://gsd.di.uminho.pt/SOI/5305O3.html

- Baseadas nas transparências originais correspondentes aos livros recomendados
- Servem apenas como "âncora" ao estudo

Aulas Práticas

- No laboratório de SO, edifício DI sala 0.05
- Inscrições na 1ª aula teórica e no início de cada aula prática
- Em qualquer turno dos 2 cursos
- Frequência obrigatória para alunos inscritos pela primeira vez (*sem frequência => não admitido a exame*)

Avaliação

- Exame final + pequena parte de avaliação nas práticas
- Exame cobre matéria teórica e prática
  - ⇒ Código, pequenos programas
  - ⇒ Valoriza-se a capacidade de raciocínio e a concepção de algoritmos (por oposição à utilização de "padrões" de soluções)
- Ninguém faz a disciplina apenas com a parte teórica
- Dificilmente a fará só com a prática

Programa

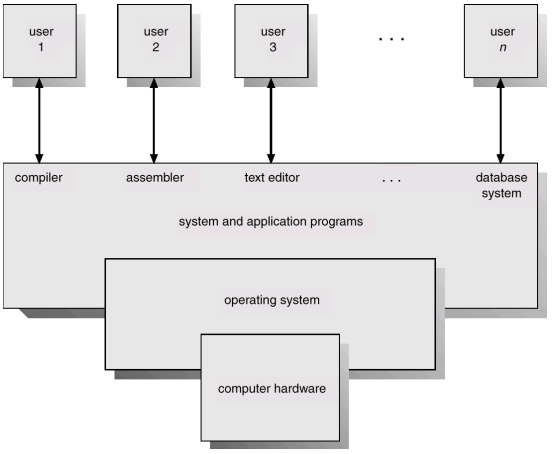
- Introdução
- Gestão de processos
- Noções de programação concorrente
- Gestão de memória
- Gestão de periféricos

Para que serve um computador?

- Para facilitar a vida aos utilizadores
- Para executar programas (aplicações)

O que é um Sistema Operativo?

- Programa que actua como **intermediário** entre os utilizadores e o hardware



Portanto...

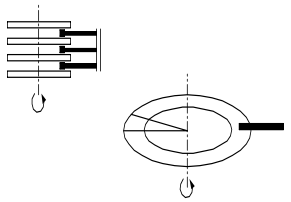
- SO deve colocar o hardware à disposição dos programas e utilizadores, mas de uma forma
  - **conveniente,**
  - **justa,**
  - **protegida,**
  - **eficiente**,
  - . . .

O Sistema Operativo pode ser visto como…

- Extensão da máquina, fornecedor de *máquina virtual*

open() , read(), write()...

- Gestor de recursos



Objectivos (1)

- Conveniência
  - SO esconde os detalhes do hardware

e.g. dimensão e organização da memória

  - Simula máquina virtual com valor acrescentado

e.g. cada *processo* executa numa "máquina" protegida

  - Fornece API mais fácil de usar do que o hardware

e.g. ficheiros vs. blocos em disco

Na prática...

- É o Sistema Operativo quem define a "personalidade " de um computador
- Como se comporta o mesmo computador (hardware) após ter arrancado
  - MSDOS?
  - Windows 95?
  - WindowsXP?
  - Linux, Knopix...?

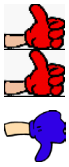
Objectivos (2)

- Eficiência
  - SO controla a alocação de recursos
    - Se 3 programas usarem a impressora ao mesmo tempo → sai lixo?
    - Programa em ciclo infinito → computador bloqueia?
    - Processo corrompe a memória dos outros → programas morrem?
  - Multiplexação:
    - Tempo: cada processo usa o recurso à vez (impressora, CPU)
    - Espaço: recurso é partilhado (memória central, disco)

Objectivos (3)

- Recapitulemos então os objectivos gerais de um SO
  - Conveniência
  - Eficiência

- Os nossos critérios de avaliação serão portanto…



Dá jeito?

É eficiente ou aumenta a eficiência geral do sistema?

Nem uma nem outra?

Evolução

- Sistemas de Computação
  - 1ª geração (1945/1955) – Válvulas e placas programáveis
  - 2ª geração (1955/1965) – Transistores e sistemas "batch"
  - 3ª geração (1965/1980) – ICs, Time-Sharing
  - 4ª geração (1980/) – PCs, Workstations, Servidores
  - ?? – PDAs, smartphones, GRID…

No início era assim...

- Acesso livre ao computador
  - Utilizador podia fazer tudo
  - Utilizador tinha de fazer tudo...
- Eficiência era baixa
  - Elevado tempo de preparação
  - Tempo "desperdiçado" com debug

E para aumentar a eficiência...

- Introduziu-se um operador especializado
  - Utilizador entrega fita perfurada ou cartões
  - Operador carrega o programa, executa-o e devolve os resultados
- Ganhou-se em eficiência, perdeu-se em conveniência
  - Operador é especialista em operação, não em programação
  - Pode haver escalonamento (i.e. alteração da ordem de execução)
  - Utilizador deixou de interagir com o seu programa

Melhor do que um operador...

- Só com um programa!
  - Controla a operação do computador
  - Encadeia "jobs", operador apenas carrega e descarrega
- Utilizadores devem usar rotinas de IO do sistema (embora ainda possam escrever as suas)

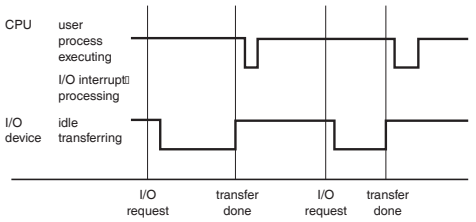
Embrião de um sistema operativo?

Mas havia o risco de...

- Se perder eficiência devido a erros de programação
  - Ciclos infinitos
  - Erros na leitura ou escrita de periféricos
  - Programa do utilizador destruir o "programa de controle"
  - Espera por periféricos lentos

Soluções (hardware)

- Interrupções



- Relógio de Tempo Virtual
- Instruções privilegiadas, 2 ou mais modos de execução
- Protecção de memória

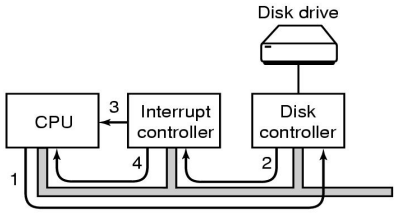
Exemplo: Polling IO

- Disk_IO()
  - Carrega o controlador de disco com parâmetros adequados (pista, sector, endereço de memória, direcção...)
  - While (NOT IO_done) /* do nothing*/

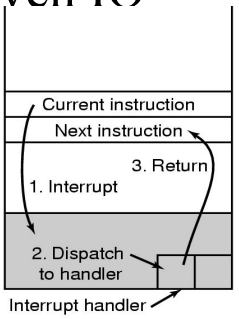
    (Equivalente a Já acabaste? Já acabaste? Já acabaste? Já acabaste? Já acabaste? Já acabaste? Já acabaste? Já acabaste? Já acabaste? Já acabaste? Já acabaste? Já acabaste? Já acabaste?...)
  - OK, regressa de disk_io()

Resulta em *desperdício de tempo de CPU*

Exemplo: Interrupt-driven IO



(a)



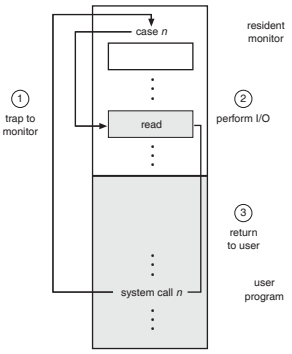
(b)

(a) OS inicia operação de IO e prepara-se para receber a interrupção

(b) No fim da operação de IO, o programa em execução é interrompido momentaneamente, trata-se o evento, e continua a execução

Soluções (software)

- Chamadas ao Sistema
- Virtualização de periféricos
- Multiprogramação

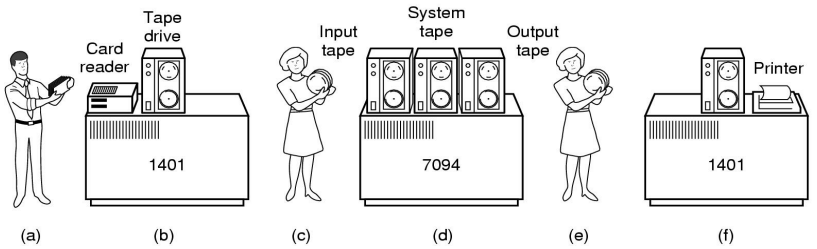


Antes de continuar…

- Assegure-se que percebeu os conceitos anteriores, e que entendeu os problemas que as soluções indicadas procuram resolver…

- Por exemplo,
  - sabe mesmo o que são e para que servem os 2 modos de execução?
  - modo de execução é hardware ou software?
  - e multiprogramação? Multiprocessamento?
  - o que é o tempo virtual?

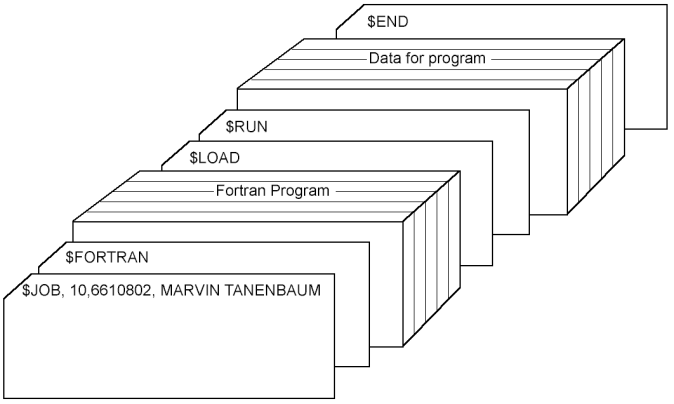
Primeiros sistemas de batch



Processador auxiliar faz IO de periféricos lentos (virtuais)

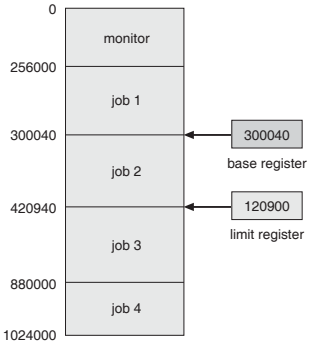
- Carregar cartões no 1401, que os copia para banda magnética
- Colocar banda no 7094 e executar os programas
- Recolher banda com resultados e colocá-la no 1401, que os envia para a impressora

Exemplo de um "job"

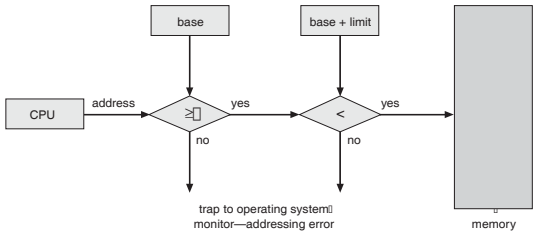


Multiprogramação

Vários jobs são carregados para memória central, e o tempo de CPU é repartido por eles.



Protecção de memória



- Note que estes testes têm de ser feitos sempre que há um acesso à memória...
- 2, 3 ou mesmo 4 vezes por instrução?

E a conveniência?

- Teve de esperar pelos sistemas de

  Time-Sharing

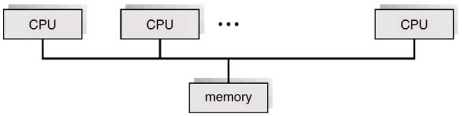
- Terminais (consolas) ligados ao computador central permitem que os utilizadores voltem a interagir directamente
- Sistema Operativo reparte o tempo de CPU pelos vários programas prontos a executar

E desde aí?

- Com o computador pessoal volta tudo ao início...
  - Control Program for Microcomputers
  - Monoprogramação, baixa eficiência...
- Mas...
  - É muito conveniente para o utilizador
  - É barato, logo eficiência não é a prioridade

Multiprocessamento (1)

- Vantagens
  - *throughput*
  - economia
  - *graceful degradation*
  - ...



Exemplo: com 2 CPUs

- A ideia é executar o dobro da carga no mesmo intervalo de tempo (i.e. maior throughput)
- não é executar um programa mais depressa (i.e. baixar tempo de resposta). Para isso necessitaria de paralelizar a aplicação, dividi-la em vários processos

Multiprocessamento (2)

- Arquitectura
  - Simétrico
    - Qualquer CPU pode executar código do SO, mas
      - cuidado com *race conditions*, (e.g. tabela de blocos de memória livres)
      - hardware mais sofisticado (e.g disco interrompe todos os CPUs?)
  - Assimétrico
    - Periféricos associados a um só CPU, o que executa o SO
      - Não há *races,* mas os outros CPUs podem estar parados porque esse não "despacha" depressa,
      - nesse caso o *throughput* diminui

Sistemas Distribuídos (1)

- Nos anos 80 apareceram as redes locais para partilha de
  - recursos caros (e.g. impressoras) ou
  - inconvenientes de replicar (e.g. sistemas de ficheiros)
  - redirecionamento de IO

Exemplo: cat fich.txt | rsh print_server lpr

- Questões
  - protocolos de comunicação, modelo cliente-servidor?
  - como saber o estado de recursos remotos?

Sistemas Distribuídos (2)

- Actualmente
  - passou-se dos *network aware OSs* para sistemas que estão vocacionados para o trabalho em rede
  - as aplicações podem localizar e aceder recursos remotos de uma forma transparente
- E chegou-se à Web...

E ainda...

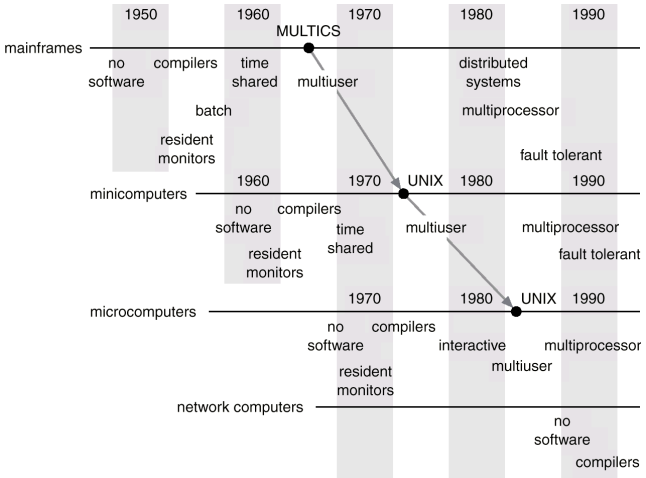
- SOs para *mainframes*:
  - IBM MVS, IBM VM/CMS.
  - desenvolvidos nos anos 60 e ainda em operação (z/VM)!
- Actualmente a virtualização é *HOT TOPIC* (vmware, …)

E ainda...

- SO de Tempo Real
  - controlo de processos industriais, sistemas de vôo, automóveis, máquinas de lavar, etc.
  - SO normais não conseguem dar garantias de tempo de resposta.

- SOs para computadores "restritos":
  - smartcards, PDAs, telemóveis, sensores...

Evolução de conceitos de SO

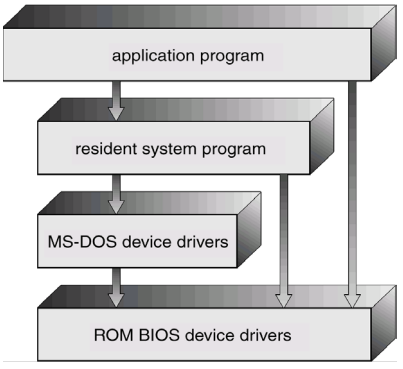


Arquitectura de Sistemas Operativos

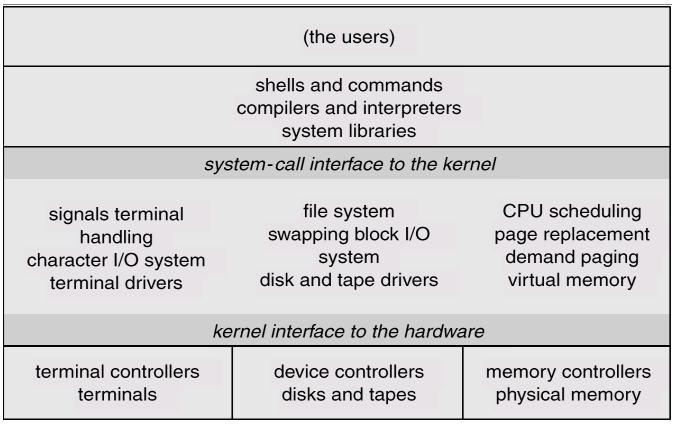
- Alguns exemplos
  - Sistemas monolíticos
  - Sistemas em camadas, hierárquicos
  - Modelo cliente-servidor
  - Máquinas virtuais

    …

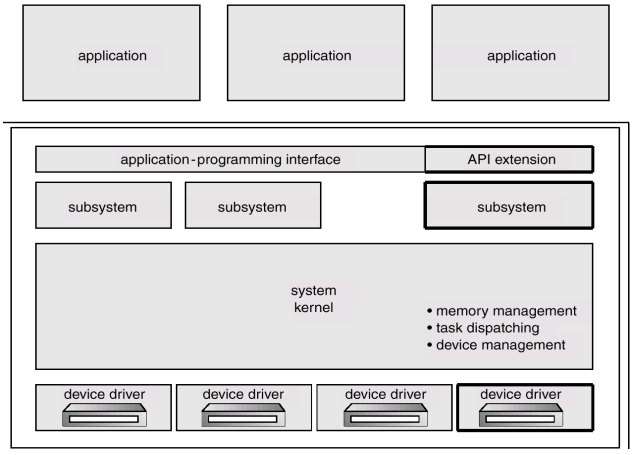
MS-DOS Layer Structure



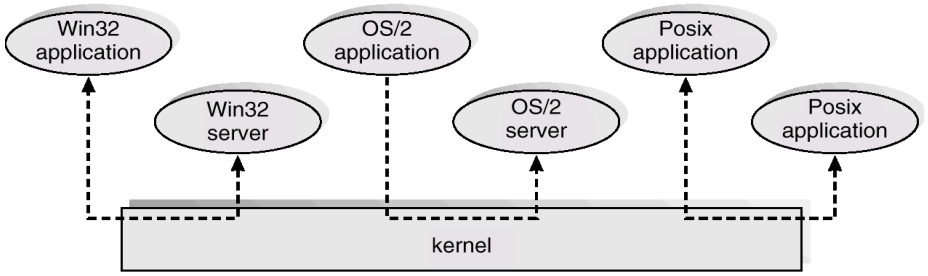
UNIX System Structure



OS/2 Layer Structure



Windows NT (cliente-servidor)



E nas nossas aulas?

- O nosso SO é bastante modular
  - Módulo de gestão de processos
  - Módulo de gestão de memória
  - Módulo de gestão de periféricos
  - Módulo de gestão de ficheiros (2º semestre)
- Mas há hierarquia / interdependência:
  - e.g. Memória virtual / memória real / disco / processos

Agora que já sabemos

- Para que serve um sistema operativo
- Quais os objectivos de um sistema operativo
- E começamos a saber:
  - como é um sistema operativo → estrutura interna, algoritmos, ...
  - e os porquês de ser assim
    → que benefícios/objectivos se pretendem alcançar com determinadas estratégias
    → em que circunstâncias não se pode fazer melhor

Convinha garantir que...

- Sabemos de facto
  - "Como é" um programa (e porquê?)
  - "Como é" um computador (e porquê?)
- Ou seja,
  - perceber as razões para o hardware e software de sistemas serem como são

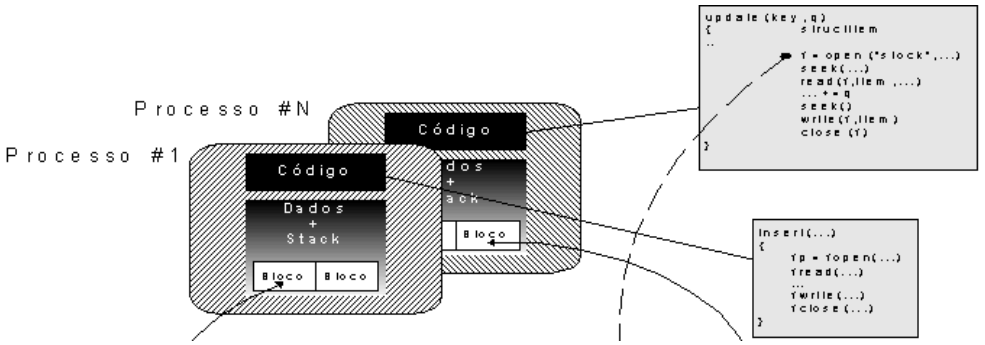
O que é/como é um programa/processo?

- Programa executável:
  - Resultado da compilação, ligação, (re)colocação em memória
  - Normalmente dependerá de módulos externos, libs
- Processo em execução:
  - código já (re)colocado em memória central + dados +stack
  - Estruturas de gestão:
    - Processo: contexto, recursos HW e SO em uso (registos, ficheiros abertos...)
    - Utilizador (uid, gid, account...)

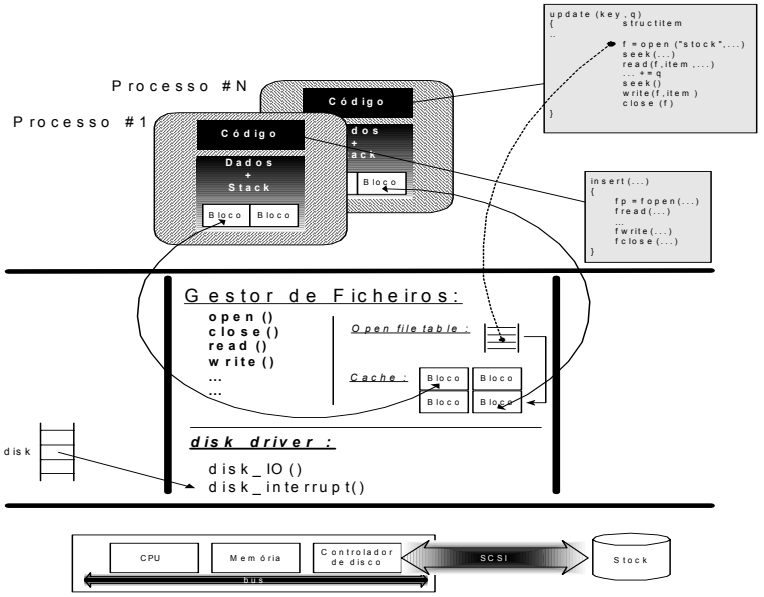
O que é/como é um computador?

- CPU
  - Registos (PC, SP, BP, CS, DS...) → "contexto volátil"
  - Instruções privilegiadas → só podem ser executadas em modo "protegido"; a forma de um programa do utilizador solicitar serviços ao SO é através das chamadas ao sistema (syscalls)
- Memória (mas o que é um endereço? E modos de endereçamento?)
- Periféricos + formas de dialogar com eles
- Interrupções (já agora, recordemos traps e excepções!)

The big picture



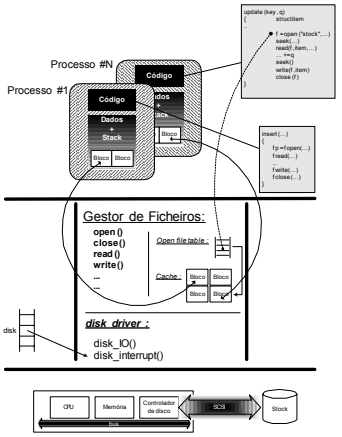
Dois programas a acederem simultaneamente ao mesmo ficheiro ou base de dados



The big picture revisited

- Assegure-se que percebeu
  - Como surgem as "race conditions"
    - entre processos
    - dentro do SO
  - Vantagens/desvantagens do uso de caches

Note que estamos a falar de caches por software, cópias de dados em memória mas acessíveis em contextos diferentes



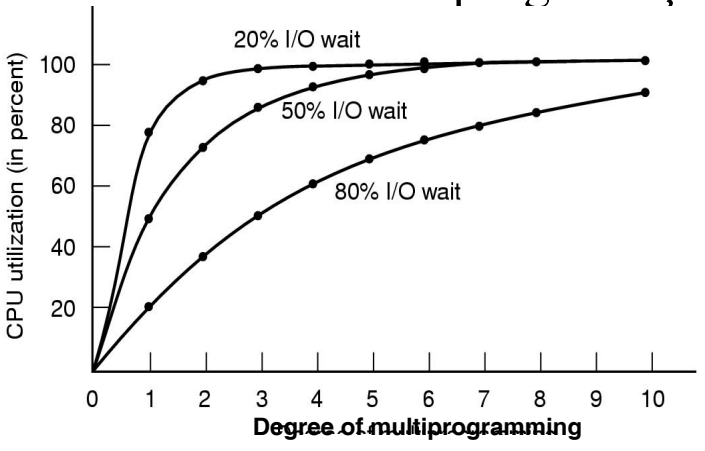
Programa

- Introdução
- Gestão de processos
- Noções de programação concorrente
- Gestão de memória
- Gestão de periféricos

Porquê criar vários processos?

- Porque dá jeito... ☞ conveniência
  - Estruturação dos programas
  - Para não estar à espera (spooling, background...)
  - Multiplas actividades / janelas
- Porque é melhor ☞ eficiência
  - Múltiplos CPUs
  - Aumenta a utilização de recursos (e.g multiprogramação)

Benefícios da multiprogramação



Processos

- Processo: um programa em execução, tem actividade própria
- Programa: entidade *estática*, Processo: entidade *dinâmica*
- Duas invocações do mesmo programa resultam em dois processos diferentes (e.g. vários utilizadores a usarem cada um a sua shell, o vi, browser, etc.)

Processos

- O contexto de execução de um processo (i.e. o seu estado) compreende:
  - código
  - dados (variáveis globais, *heap*, *stack*)
  - estado do processador (registos)
  - ficheiros abertos,
  - tempo de CPU consumido, …

Exemplo de informação sobre um processo

Process management	**Memory management**	**File management**
Registers	Pointer to text segment	Root directory
Program counter	Pointer to data segment	Working directory
Program status word	Pointer to stack segment	File descriptors
Stack pointer		User ID
Process state		Group ID
Priority		
Scheduling parameters		
Process ID		
Parent process		
Process group		
Signals		
Time when process started		
CPU time used		
Children's CPU time		
Time of next alarm		

Processos

- O SO deverá ser capaz de:
  - Criar, suspender e reiniciar a execução de processos
  - Suportar a comunicação entre processos
- O próprio SO tem muitos processos "do sistema"

Processos

- Para poderem executar os seus programas, os processos requerem tempo de CPU, memória, utilização de dispositivos…
- Por outras palavras, os processos

COMPETEM POR RECURSOS

- E cabe ao sistema operativo fazer o escalonamento dos processos, i.e. atribuir os recursos pela ordem correspondente às políticas de escalonamento

Políticas de escalonamento

- Qual a melhor?
- E a resposta é...
  - Depende!
    - De quem responde, utilizador ou administrador?
- É preciso definir OBJECTIVOS

Objectivos

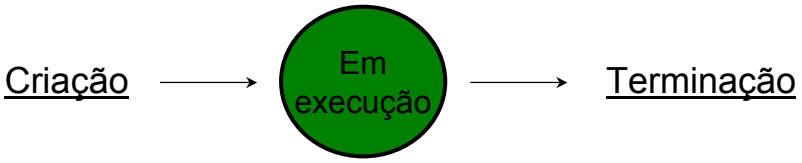
- Conveniência
  - Justiça
  - Redução dos tempos de resposta
  - Previsibilidade

    ...
- Eficiência
  - Débito (*throughput*), transacções por segundo, ...
  - Maximização da utilização de CPU e outros recursos
  - Favorecer processos "bem comportados", etc.

Critérios de escalonamento

- IO-bound ou CPU-bound
- Interactivo ou não (batch, background)
- Urgência de resposta (e.g. tempo real)
- Comportamento recente (utilização de memória, CPU)
- Necessidade de periféricos especiais
- PAGOU para ir à frente dos outros...

Estados de um processo (i)



Processos em Unix

- Para criar um novo processo:
  - **fork**: cria um novo processo (a chamada ao sistema retorna "duas vezes", uma para o pai e outra para o filho)
  - A partir daqui, ambos executam o mesmo programa
- Para executar outro programa
  - **exec**: substitui o programa do processo corrente por um novo programa
- Para terminar a execução
  - **exit**

Compare o **exec** com a invocação de uma função: são muito diferentes

Relação entre processos

- Possibilidades na execução dos filhos:
  - Pai e filho executam concorrentemente
  - Pai aguarda pelo fim da execução do filho para continuar
- Possibilidades no espaço de endereçamento:
  - O do filho é uma duplicação do do pai
  - O do filho é um programa diferente desde a criação

fork/exec

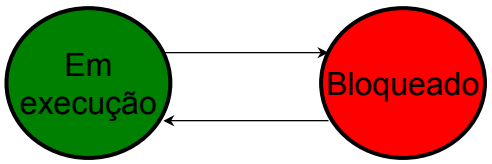
```
pid = fork()
if (pid == 0) {
        /* Sou o filho */
        exec( novo programa )
else {
        /* Sou o pai
           A Identificação do meu filho é colocada na variavel  pid
        */
}
```

fork'ing e exec'ing

- O padrão fork/exec é muito frequente (e.g. shell)
- Optimizações (a rever no capítulo de gestão de memória):
  - **copy on write**
  - Variante: **vfork**, não duplica o espaço de endereçamento; ambos os processos partilham o espaço de endereçamento e o pai é bloqueado até o filho terminar ou invocar o exec.

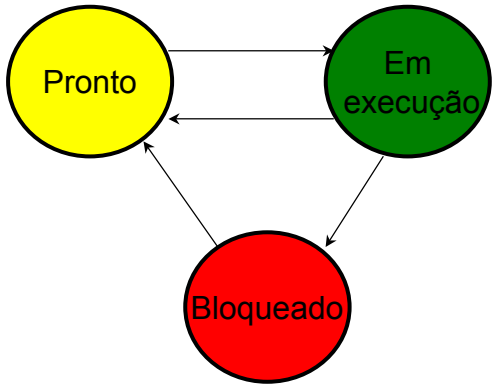
Estados de um processo (ii)

Podemos para já admitir que durante a sua "vida" os processos passam por 2 estados:



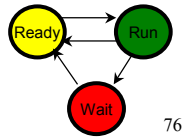
Estados de um processo (iii)

- Na prática, há mais processos não bloqueados do que CPUs
- Surge uma fila de espera com processos **Prontos a executar**
- Processos em execução podem ser desafectados

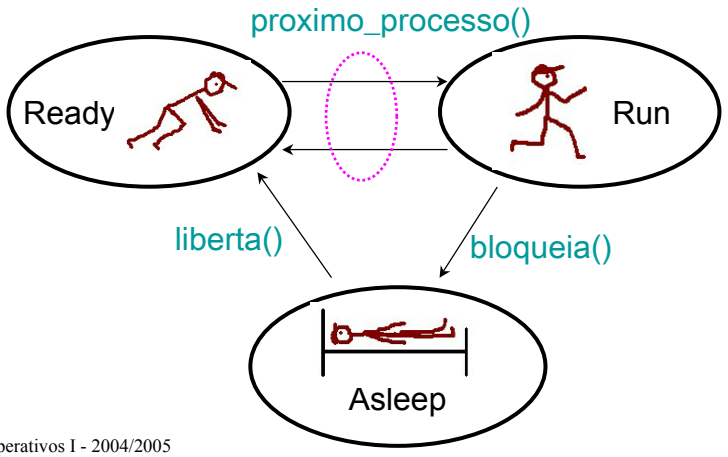


Estados de um processo (iv)

- Em execução
  - Foi-lhe atribuído o/um CPU, executa o programa correspondente
- Bloqueado
  - O processo está logicamente impedido de prosseguir, e.g. porque lhe falta um recurso ou espera por evento
  - Do ponto de vista do SO, é uma transição VOLUNTÁRIA!
- Pronto a executar, aguarda escalonamento

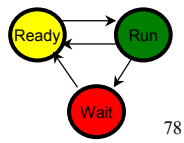


Primitivas de despacho (i)



Primitivas de despacho (ii)

- Bloqueia(evento)
  - Coloca processo corrente na fila de processos parados à espera deste "evento"
  - Invoca próximo_processo()
- Liberta(evento) ou liberta(processo,evento)
  - Se o outro processo não está à espera de mais nenhum evento, então coloca-o na lista de processos prontos a executar
  - Nesta altura pode invocar ou não próximo_processo()



Primitivas de despacho (iii)

- Proximo_processo()
  - Selecciona um dos processos existentes na lista de processos prontos a executar, de acordo com a política de escalonamento
  - Executa a comutação de contexto
    - Salvaguarda contexto volátil do processo corrente
    - Carrega contexto do processo escolhido e regressa (executa o return)

Como o Stack Pointer foi mudado, ”regressa” para o processo escolhido!

Principais decisões

- Qual o próximo processo?
- Quando começa a executar?
- Durante quanto tempo?

- Por outras palavras,

Há desafectação forçada ou não?

Escalonamento de processos

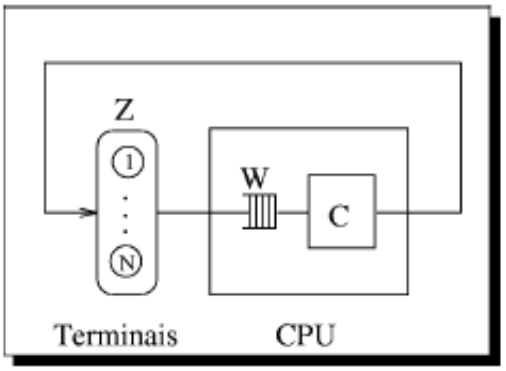
- Quando, uma vez atribuído a um processo, o CPU nunca lhe é retirado então diz-se que o escalonamento é **cooperativo** (non-preemptive). Exemplos:
  - windows 3.1,
  - co-rotinas,
  - thread_yield()
- Quando o CPU pode ser retirado a um processo porque surgiu outro de maior prioridade ou a fatia de tempo acabou diz-se que o escalonamento tem **desafectação forçada** (preemptive)

Universidade do Minho
Departamento de Informática
GSD

Escalonamento de processos

- Escalonamento **cooperativo** (non-preemptive).
  - "poor man´s approach to multitasking"
    - obrigatório "chamar o sistema" periodicamente, para dar oportunidade de escalonar outra actividade?
  - Sensível às variações de carga
- Escalonamento com **desafectação forçada**
  - Sistema "responde" melhor
  - Mas a comutação de contexto tem overhead

Modelo de sistema interactivo



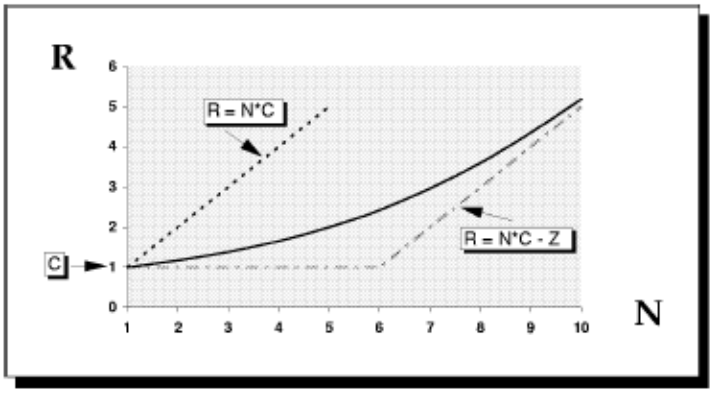
Z = Think time

C = Service time

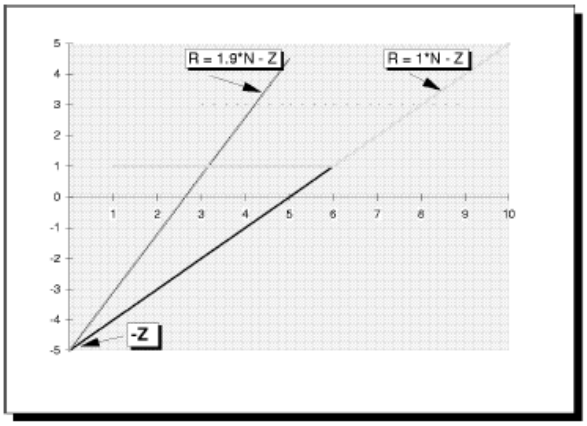
W = Wait time

N = Number of users

Tempo de Resposta (carga homogénea)



Tempo de Resposta (carga heterogénea)



- Assuma-se agora que uma em cada 10 interacções é muito longa, 10 vezes maior.
- Veja-se a degradação de tempos de resposta

Tempo de Resposta (carga heterogénea)

- Para evitar que as interações longas monopolizem o CPU e aumentem o tempo de resposta das restantes, temos várias alternativas:
  - Uma vez que, no exemplo apresentado, a carga média quase duplicou, podemos pensar em fazer um upgrade ao CPU. Por exemplo, trocá-lo por outro com o dobro da frequência de clock.
  - Usar multiprocessamento, 2 CPUs iguais ao que lá está
  - Usar desafectação forçada

Qual prefere?

Tempo de Resposta (carga heterogénea)

- Para permitir a troca rápida de processos, deve atribuir-se um quantum (ou time slice) :
  - Interacções curtas terminam dentro dessa fatia de tempo, logo não são afectadas pela política de desafectação.
  - Interacções longas executam durante um quantum e a seguir o processo correspondente regressa ao estado de Pronto a Executar, dando a vez a outros processos. Mais tarde ser-lhe-á atribuído nova fatia de tempo, e sucessivamente até a interacção terminar.

Duração da fatia de tempo

- Maioria das interacções deve "caber" num quantum

  $R = W + C$

- Se precisar de 2 passagens pelo CPU, $T_{Resposta}$ é quase o dobro!

  $R = W + q + W + c'$