Sistemas Operativos B

MIEBIOM

Grupo de Sistemas Distribuídos

http://gsd.di.uminho.pt

Who am I ?

Surpresa...

- Esta UCE engana: não chega "saber usar" o Windows, Linux ou OSX
- Estamos em engenharia: temos de
  - Perceber compromissos => usar a "massa cinzenta"
  - "Sujar as mãos" => programar, ver manuais, corrigir erros, configurar, avaliar desempenho...
- Vamos falar de
  - Algoritmos, estruturas de dados, linguagens de programação
  - Como funciona um computador, como é constituído, como se gerem recursos
  - Como se conseguem executar tantas aplicações "ao mesmo tempo"
  - Eficiência, desempenho, SEGURANÇA...

As boas notícias

- Até hoje, ninguém reprovou...
- Basta um pouco de estudo e bom senso para se ficar a perceber como funcionam os sistemas actuais. Incluindo…
  - Se eu carregar numa tecla, o que se passa nos bastidores até o caracter correspondente chegar ao programa que alguém escreveu há meses?
  - O sistema está a ficar lento. Que fazer para melhorar o desempenho?
  - Como escrever uma aplicação concorrente, tipo gestão de stocks, com muita gente a querer actualizar stocks ao mesmo tempo?
  - Como executar pequenas tarefas de administração de sistemas?
  - etc, etc.

Organização das aulas

- Componente teórica:
  - "Systems Programming": ligação hw/so/sw
  - Arquitectura interna dos SOs, estratégias de gestão
  - Interessa perceber os "porquês", não interessa decorar..
- Componente prática em Unix (Linux, Mac OSX)
  - Programação em C e Bash
- Tentativa de parceria com Sistemas Distribuídos e Criptografia

Bibliografia recomendada

Alves Marques et al., *Sistemas Operativos*, FCA Editora de Informática,2009

OU

A. Silberschatz et al., *Applied Operating System Concepts*, John Wiley & Sons, 2000.

Bibliografia recomendada

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

  …
- www.slashdot.org …

Bibliografia Adicional

- R. Stevens, *Advanced Programming in the Unix Environment*, Addison Wesley, 1990.
- A. S. Tanenbaum, *Modern Operating Systems*, 2nd edition, Prentice Hall, 2001
- 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, …

Slides

- Disponíveis em:

  http://gsd.di.uminho.pt/teaching/BIOSO/2011/

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

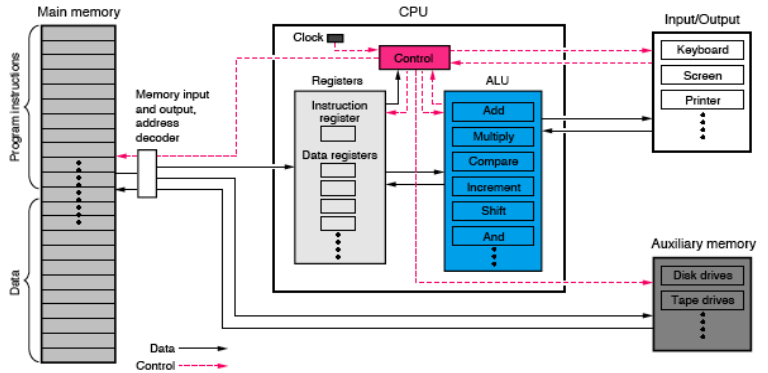
Avaliação

- 3 componentes
  - Prova escrita (50%) no final do semestre + trabalho prático (40%) + mini-teste de bash (10%)
  - Trabalho prático é obrigatório e tem de ter positiva para ser admitido a exame de recurso.
- Prova escrita e exame de recurso cobrem toda a matéria teórica e prática. Precisa de ter à-vontade na parte prática para garantir o sucesso nesta unidade curricular
  - ⇒ Código, pequenos programas em C, scripts bash
  - ⇒ Valoriza-se a capacidade de raciocínio e a concepção de algoritmos (por oposição à utilização de "padrões" de soluções)

Programa

- Introdução aos sistemas operativos, linguagem C, Linux
- Gestão de processos, memória, ficheiros e periféricos
- Noções de programação concorrente em Unix

O que é um computador?

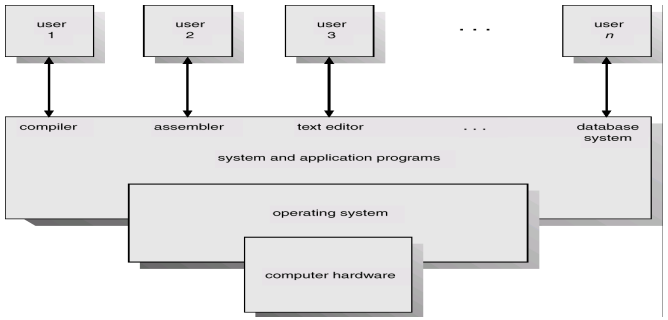


O que é uma instrução (máquina) ?

- LOAD A addr
- LOAD B addr
- SAVE A addr
- SAVE B addr
- LOADI A, val
- LOADI B, val
- ADD A, B
- SUB A, B
- MUL A, B
- CMP A, B
- JMP addr
- JE addr
- JNE addr
- JG addr
- ...

Sistema Operativo

- Actua como **intermediário** entre os utilizadores e o hardware

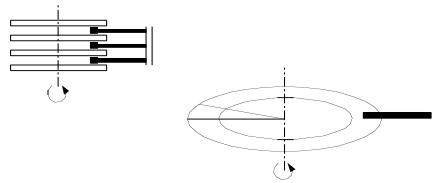


Portanto...

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

Sistema Operativo

- Extensão da máquina real, fornece "instruções" de alto nível:
  - 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...

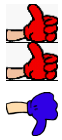
- É o Sistema Operativo quem define a "personalidade " de um computador
- Como se comporta o mesmo computador (hardware) após ter arrancado
  - MSDOS?
  - Windows 95?
  - Windows 7, Linux, Mac OSX?

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, Cloud…

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)

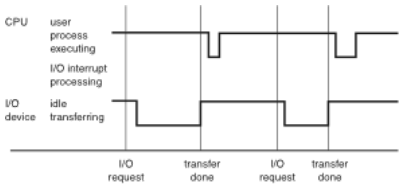
Embrião de um sistema operativo?

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

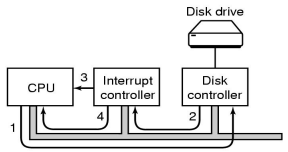
Exemplo: Polling IO

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

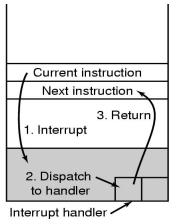
    (Equivalente a Já acabaste? Já acabaste? Já acabaste? Já acabaste? Já acabaste? Já acabaste? Já acabaste? Já acabaste? Já acabaste? Já acabaste? Já acabaste? Já acabaste? Já acabaste?...)
  - OK, regressa de disk_io()

Resulta em *desperdício de tempo de CPU*

Exemplo: Interrupt-driven IO



(a)



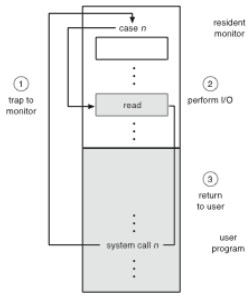
(b)

(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)

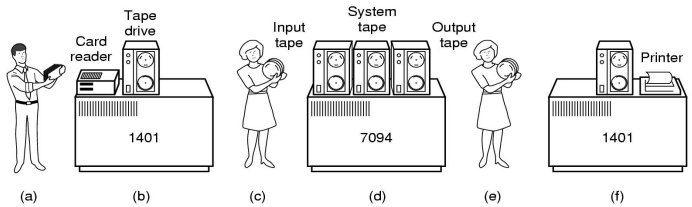
- Chamadas ao Sistema
- Virtualização de periféricos
- Multiprogramação



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? E as interrupções?
  - modo de execução é hardware ou software?
  - E multiprogramação? Multiprocessamento? O que é o tempo virtual?

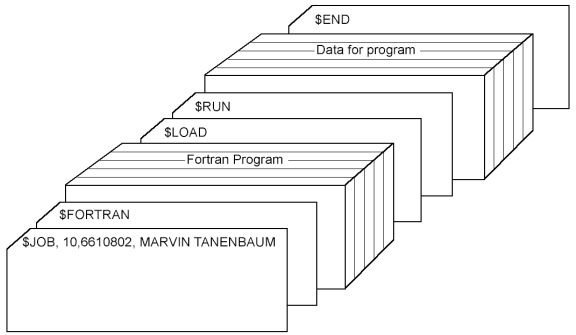
Primeiros sistemas de batch



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

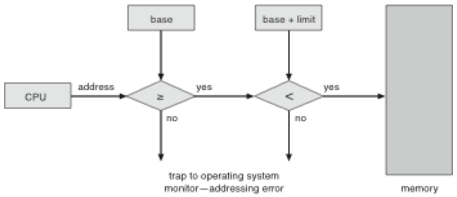
Exemplo de um "job"



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

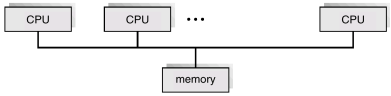
- Terminais (consolas) ligados ao computador central permitem que os utilizadores voltem a interagir directamente
- Sistema Operativo reparte o tempo de CPU pelos vários programas prontos a executar

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?



- 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)

- Simétrico, qualquer CPU pode executar código do SO
  - 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 ao CPU que executa o SO
  - Não há *races,* mas os outros CPUs podem estar parados porque esse não "despacha" depressa; nesse caso o *throughput* diminui
- Hoje em dia os CPUs possuem vários núcleos ("cores", em inglês)

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)

- Em pouco mais do que uma década
  - passou-se dos *network aware OSs* para sistemas vocacionados para o trabalho em rede
  - as aplicações podem localizar e aceder recursos remotos de uma forma transparente
- E chegou-se à Web... no telemóvel...

E ainda...

- SOs para *mainframes*:
  - IBM MVS, IBM VM/CMS.
  - desenvolvidos nos anos 60 e ainda em operação (z/VM)!

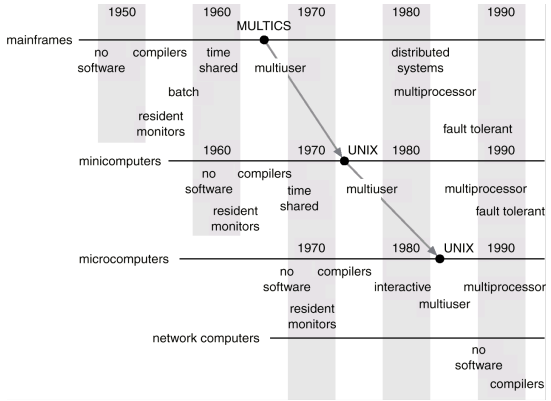
- Em anos recentes renovou-se o interesse pela virtualização: Xen, Vmware, Hyper V…

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, sensores...
  - Smartphones já não são assim tão restritos!

Evolução de conceitos de SO

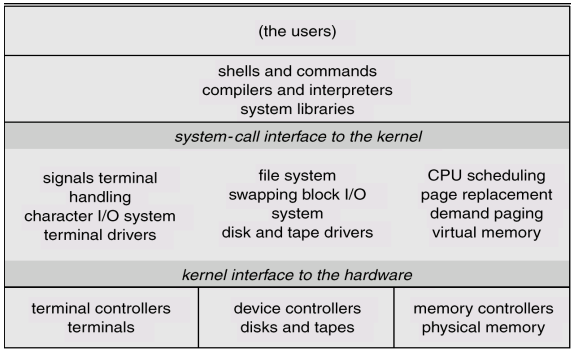


Arquitectura de Sistemas Operativos

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

  …

UNIX System Structure



Nas nossas aulas?

- O SO é bastante modular e "horizontal", à-la Unix
  - Módulo de gestão de processos
  - Módulo de gestão de memória
  - Módulo de gestão de periféricos
  - Módulo de gestão de ficheiros
- 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?)
- Começamos a perceber as razões para o hardware e software de sistemas serem como são!

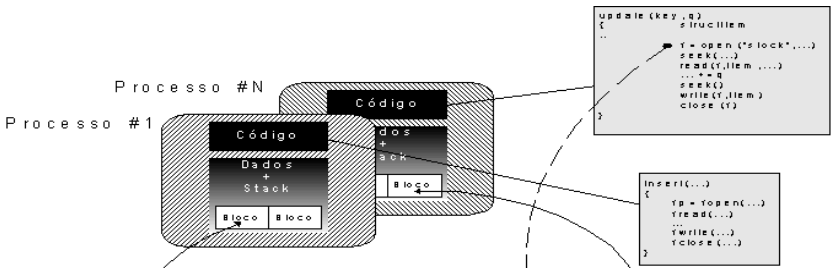
O que é/como é um programa/processo?

- Programa executável:
  - Resultado da compilação, ligação, (re)colocação em memória
  - Normalmente dependerá de módulos externos, bibliotecas
- 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...)

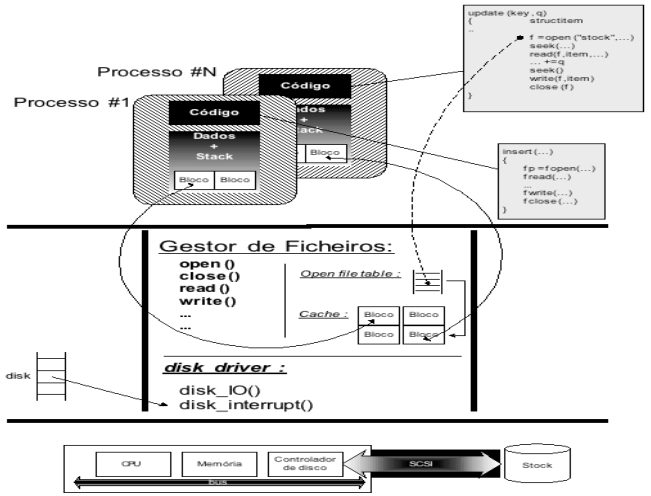
O que é/como é um computador?

- 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

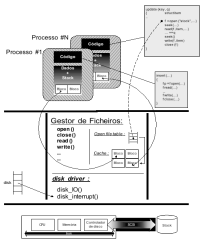


Como coordenar a execução simultânea de dois programas que pretendem aceder ao mesmo ficheiro ou base de dados?



Assegure-se que percebeu

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



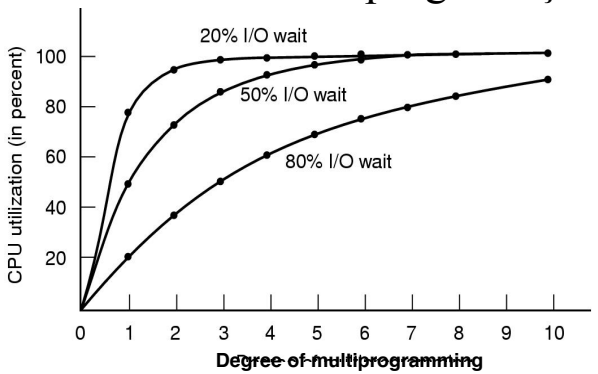
Estamos a falar de caches por software, neste caso são várias cópias em memória dos mesmos dados. Parece pouco razoável mas são contextos diferentes: sistema operativo, protegido, buffers da biblioteca (user-level) e finalmente os dados no programa.

Gestão de Processos

Porquê criar vários processos?

- Dá jeito... ✓ conveniência
  - Estruturação dos programas
  - Para não estar à espera (spooling, background...)
  - Múltiplas actividades / janelas
- 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 executar os programas, os processos precisam de tempo de CPU, memória, diversos 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 perante determinada carga

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.

Alguns 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...

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

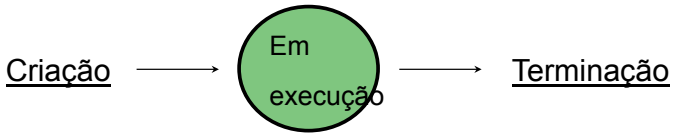
Relação entre processos

- 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/wait

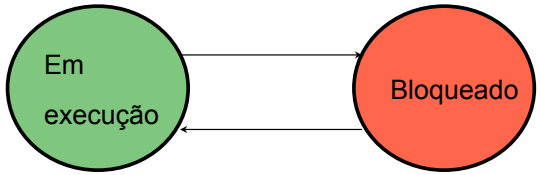
```
pid = fork()
if (pid == 0) {
        /* Sou o filho */
        exec( novo programa )
else {
        /* Sou o pai, faço qualquer coisa e depois espero que o filho termine */
        … = wait (…);
}
```


Estados de um processo (i)



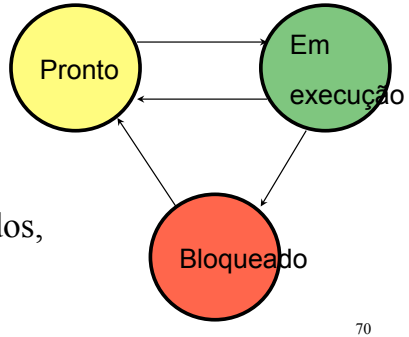
Estados de um processo (ii)

Na prática, raros serão os processos que conseguem executar continuamente do princípio ao fim. Muito provavelmente terão de aguardar por eventos que ainda não ocorreram (e.g. bloco lido do disco, caracter do teclado...). Assim, durante a sua "vida" os processos passam por 2 estados:



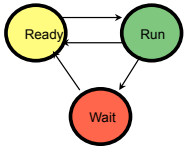
Estados de um processo (iii)

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

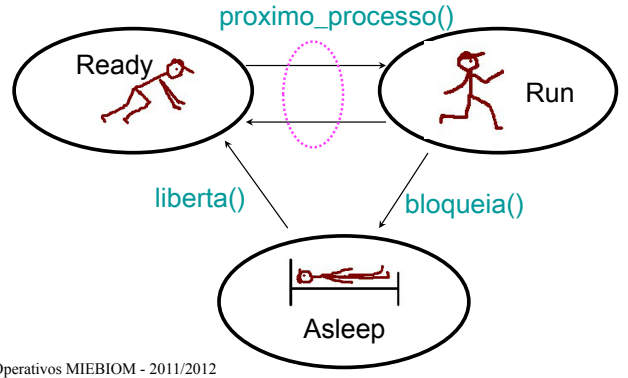


Estados de um processo (iv)

- Em execução
  - Foi-lhe atribuído o/um CPU/núcleo, corre 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

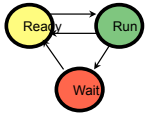


Primitivas de despacho (i)



Primitivas de despacho (ii)

- Bloqueia (evento)
  - Coloca processo corrente na fila de processos parados à espera deste "evento"
  - Invoca próximo_processo()
- Liberta (evento)
  - Assinala a ocorrência do evento. Os processos à espera desse evento são colocados 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?

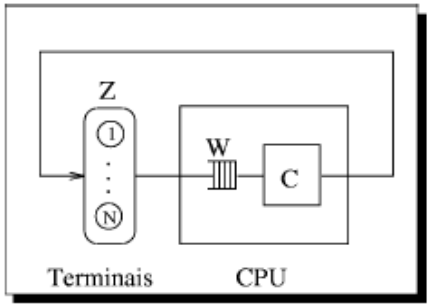
Escalonamento de processos

- 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 diz-se que o escalonamento é com **desafectação forçada** (preemptive). Pode ser devido ao fim da *fatia de tempo* ou porque chegou um processo de maior pritoridade.

Escalonamento de processos

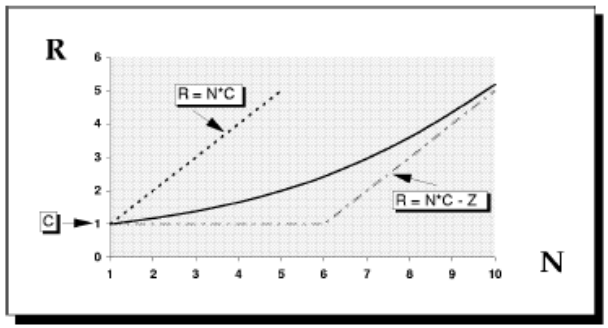
- Escalonamento **cooperativo** (non-preemptive).
  - "poor man´s approach to multitasking" ?
  - Sensível às variações de carga
- Escalonamento com **desafectação forçada**
  - Sistema "responde" melhor
  - Mas a comutação de contexto tem overhead

Modelo de sistema interactivo

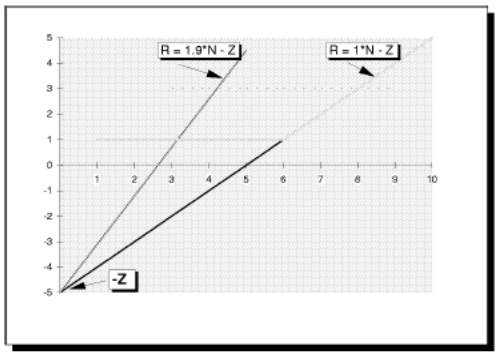


- Z = Think time
- C = Service time
- W = Wait time
- N = Number of users

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 deve usar-se desafectação forçada.
- Neste caso deve atribuir-se um quantum (ou time slice) para permitir a troca rápida de processos:
  - Interacções curtas terminam dentro dessa fatia de tempo, logo não são afectadas pela política de desafectação.
  - Interacções longas executam durante uma fatia de tempo 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.

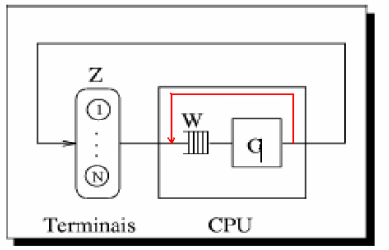
Duração da fatia de tempo

- Maioria das interacções deve "caber" num quantum

  $R = W + C$

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

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



Escalonamento de processos

- Escalonadores de longo-prazo (segundos, minutos) e de curto-prazo (milisegundos)
- Processo CPU-bound: processo que faz pouco I/O mas que requer muito processamento
- Processo I/O-bound: processo que está frequentemente à espera de I/O.

Escalonamento de processos

- Os processos prontos são seriados numa fila (*ready list*)
- A lista é uma lista ligada de apontadores para PCB's
- A lista poderá estar ordenada por prioridades de forma a dar um tratamento preferencial aos processos com maior prioridade

Escalonamento de processos

- Quando um processo é escalonado, é retirado da *ready list* e posto a executar
- O processo pode "perder" o CPU por várias razões:
  - Aparece um processo com maior prioridade
  - Pedido de I/O (passa ao estado de bloqueado)
  - O *quantum* expira (passa ao estado de pronto)

Escalonamento de processos

- Pretende-se maximizar a utilização do CPU tendo em atenção outros aspectos importantes:
  - Tempo de resposta para aplicações interactivas
  - Utilização de dispositivos de I/O
  - Justiça na distribuição do tempo de CPU

Escalonamento de processos

- A decisão de escalonar um processo pode ser tomada em diversas alturas:
  - Qdo um processo passa de a-executar a bloqueado
  - Qdo um processo passa de a-executar a pronto
  - Qdo se completa uma operação de I/O
  - Qdo um processo termina

Escalonamento de processos

- Alguns algoritmos de escalonamento:
  - FCFS (First Come, First Served)
  - SJF (Shortest Job First)
  - SRTF (Shortest Remaining Time First)
  - Preemptive Priority Scheduling
  - RR (Round Robin)
  - MLQ (multi-level Queues)

First Come, First Served (FCFS)

- A *ready list* é uma fila FIFO
- O processos são colocados no fim da fila e é selecionado o que se encontra na cabeçã da lista.
- Método cooperativo
- Nada apropriado para ambientes interactivos

SJF (Shortest Job First)

- A ideia é escalonar primeiro o processo mais curto
- Possibilidades:
  - Desafectação forçada (SRTF) - interrompe o processo em execução se aparecer um mais curto
  - Cooperativo – aguardar que o processo corrente termine
- Como determinar o tempo que cada processo demora?

Preemptive Priority

- Associa uma *prioridade* (geralmente um inteiro) a cada processo.
- A *ready queue* é uma fila seriada por prioridades.
- Escalona-se sempre o processo na frente da fila.
- Se aparecer um processo com maior prioridade do que o que está a executar faz a troca dos processos

Preemptive Priority

- Problema: starvation
- Uma solução: envelhecimento – aumenta a prioridade dos processos pouco a pouco de forma a que inevitavelmente executem e terminem.

RR (Round Robin)

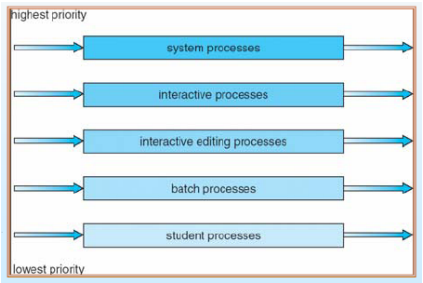
- De cada vez que um processo é escolhido para execução é-lhe atribuído um intervalo de tempo fixo de CPU
- Quando um processo esgota o seu quantum sai do CPU e regressa para o fim da fila Ready
- Ignorando os overheads do escalonamento, cada um dos *n* processos CPU-bound terá (1/*n*) do tempo disponível de CPU

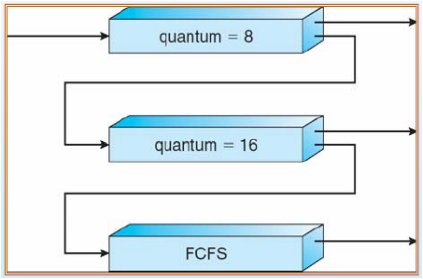
RR

- Se o quantum for (muito) grande o RR tende a comportar-se como FCFS
- Se o quantum for (muito) pequeno então o overhead de mudanças de contexto tende a dominar degradando os níveis de utilização de CPU
- Tem um tempo de resposta melhor que o SJF (o quantum "é" normalmente o SJ)
- Pode ser optimizada ajustando o local da fila ready onde inserem os processos.

Multi-level Queues (MLQ)

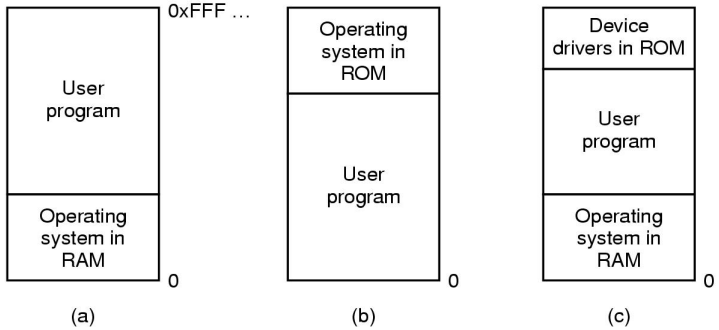
MLQ sem feedback **MLQ com feedback**





Gestão de Memória

Monoprogramação

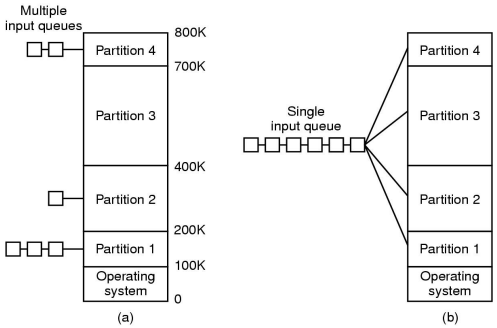


Três formas de organizar a memória com SO e apenas um processo

Multiprogramação

- Recorde que a multiprogramação visa aumentar a eficiência
- Quando o processo em execução não pode prosseguir, comuta-se de imediato para outro processo já residente em memória central.
- CPU não pára à espera do disco.

Multiprogramação c/ partições dim. fixa



Filas separadas para cada partição Fila única

Partições

- Em princípio, a fila única será mais eficiente porque não mantém processos à espera de serem carregados para memória quando há partições disponíveis, mas...
- Embora para efeitos de protecção baste mudar os registos que marcam os limites inferior e superior da partição,
- Os endereços terão de mudar se o programa for “swapped out” e swapped in” para outra partição.

Recolocação e Protecção

- Incerteza sobre o endereço de carregamento do programa
  - Endereços de variáveis e funções não pode ser absoluto
  - Um processo não se pode sobrepor a outro processo
- Solução: uso de valores de base e limite
  - Endereços adicionados à base para obter endereços físicos
  - Endereços superiores ao limite são erros

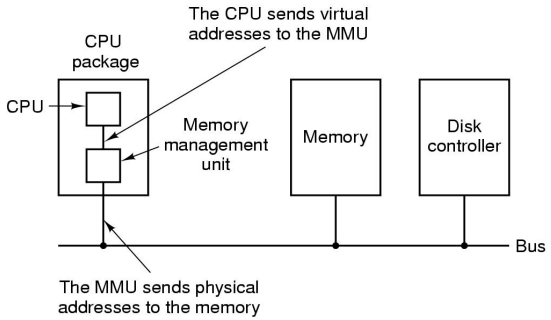
Partições de dimensão variável

- Havendo suporte de hardware para recolocação dinâmica, pode-se passar para um número (variável) de partições de memória com dimensão variável
- A dimensão da partição é estabelecida quando o programa é carregado
- Conduz a algoritmos de "alocação": first-fit, best-fit, worst-fit, buddy-system...

Alguns problemas com partições

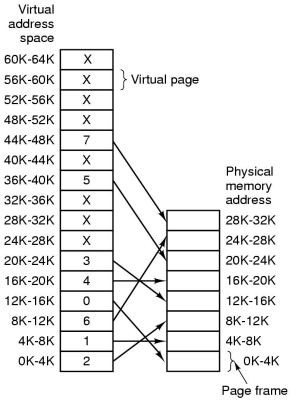
- Dimensão máxima dos programas diminui; Pode obrigar a overlays
- Desperdício com fragmentação interna e/ou externa
- Desperdício devido à dispersão de referências, estática e dinâmica
- Desperdício porque não consegue partilhar (porque não sabe onde começa/acaba o código)
- Desperdício de CPU devido a algoritmos de gestão complicados
- Protecção "tudo ou nada"; não distingue código, dados e stack

Memória Virtual

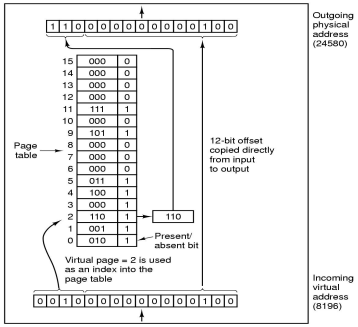


Memória Virtual

A relação entre endereços virtuais e físicos é dada por uma tabela



Memória Virtual



Operação da MMU com 16 páginas de 4 KB

Rejeição de páginas

- Um *page fault* leva:
  - A decidir que página em memória rejeitar
  - A criar espaço para uma nova página
- Uma página modificada tem que ser escrita
  - Uma não modificada é logo libertada
- Convém não rejeitar uma página frequentemente usada
  - Pois provavelmente terá de ser carregada a seguir

Rejeição de páginas FIFO

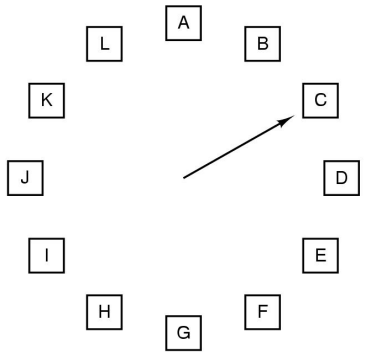
- Mantém uma lista das páginas em memória
  - Segundo a ordem em que foram carregadas
- A página no topo da lista é rejeitada
- Desvantagem
  - A página há mais tempo em memória poderá ser a mais usada

Rejeição de páginas NRU

- Cada página tem 1 bit de acesso e 1 de escrita
- As páginas são assim classificadas:
  1. Não acedida, não modificada
  2. Não acedida, modificada
  3. Acedida, não modificada
  4. Acedida, modificada

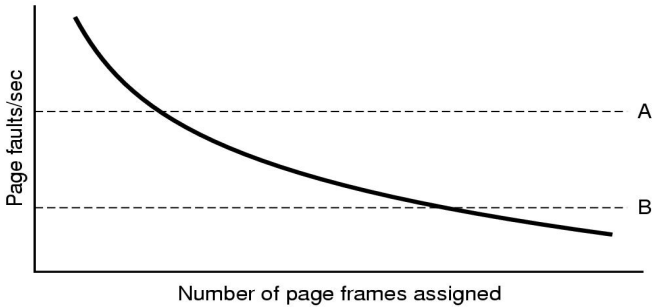
NRU remove a página com menor "ranking"

Rejeição de páginas: Relógio



When a page fault occurs, the page the hand is pointing to is inspected. The action taken depends on the R bit:
R = 0: Evict the page
R = 1: Clear R and advance hand

Alocação Local e Global



Controlo de carga / thrashing

- Apesar de um bom desenho, pode ainda ocorrer thrashing
- Quando a Frequência de Page Faults indica que:
  - Alguns processos precisam de mais memória
  - Mas nenhum pode ceder parte da que tem
- A solução é fazer Swap Out
  - Passar um ou mais processos para disco e dividir as páginas que lhes estavam atribuídas
  - Rever o grau de multiprogramação

Tamanho das páginas

Páginas pequenas

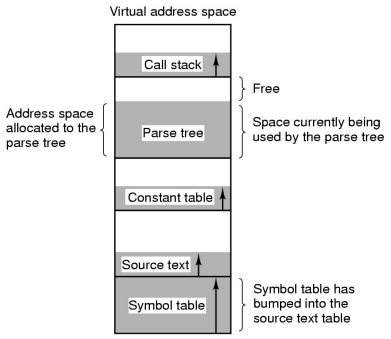
- Vantagens
  - Menos fragmentação interna
  - Melhor adequação a várias estruturas de dados e código
  - Menos partes de programas não usados em memória
- Desvantagens
  - Mais páginas, tabelas de páginas maiores

Aspectos de Implementação

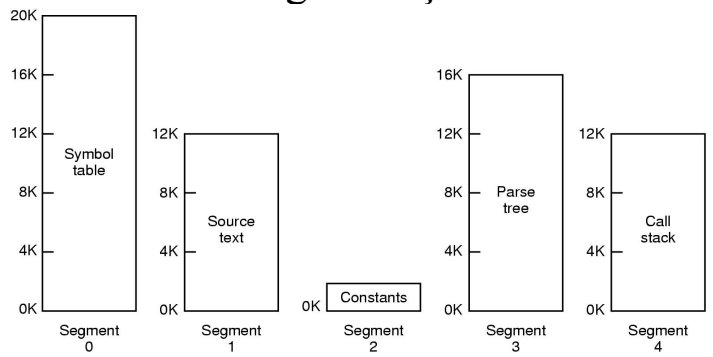
Tratamento da Page Fault

1. O hardware interrompe o kernel
2. São salvaguardados os registos
3. SO determina a página virtual necessária
4. SO valida endereço e procura page frame
5. Se a página foi alterada, escreve-a para disco

Segmentação



Segmentação



Cada tabela pode crescer ou encolher independentemente

Segmentação vs Paginação

	Paginação	Segmentação
Transparente para o programador	Sim	Não
Número de espaços de endereçamento	1	Vários
O espaço de endereçamento pode ultrapassar o tamanho da memória física	Sim	Sim
O código e dados podem ser distintos e protegidos separadamente	Não	Sim
Tabelas de tamanho variável podem são geridas facilmente	Não	Sim
A partilha de código é facilitada	Não	Sim

Gestão de Ficheiros

Gestão de Ficheiros

- Sistemas de ficheiros
  - Recapitulação de hw e sw de IO
    - Discos, partições, disk IO, device drivers, concorrência, caches, etc.
  - Requisitos, objectivos, estudo de casos
  - RAID, Log structured File Systems
  - Noções de sistemas de ficheiros distribuídos

Sistemas de ficheiros: requisitos

- Persistência
- Grande escala (quantidade de ficheiros + dimensão elevada)
- Rapidez de acesso (Tempo de acesso a disco >> TaccRAM)
- Concorrência
- Segurança
- ...

Objectivos (1)

- Armazenamento
  - Persistente (backup, undelete, RAID)
  - Eficiente
    - Espaço (=> aproveitar)
      - Dados (exemplos)
        - Alocação não contígua para eliminar fragmentação externa
        - Suporte para ficheiros "dispersos" (resultado de "hash", por exemplo)
      - Metadados, eg. estruturas para representar blocos livres/ocupados: FAT, i-nodes, ...
    - Tempo: algoritmos de gestão e recuperação rápidos

Objectivos (2)

- Acesso
  - Escalável
  - Conveniente
    - estrutura interna visivel (pelo kernel) ou só pelas aplicações?
      - Sequencia de bytes vs. Ficheiros indexados
  - Seguro
    - controlo de acessos
    - auditoria
    - privacidade...

Objectivos (3)

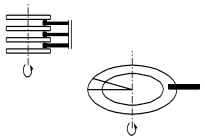
- Acesso
  - Rápido (alguns exemplos de "bom-senso")
    - Evitar dispersão de blocos pelo disco => cuidado na alocação, usando por exemplo
      - os "cilinder groups" do BSD, "file extents" do JFS e XFS
      - "hot file clustering" e desfragmentação "on-the-fly" do Mac OS X
    - Uso de caches (em disco e RAM) e delayed write => CUIDADO!
    - Directorias
      - Podem ter milhares de entradas (eg. e-mail!)
      - Procura sequencial? Binária? B-trees?

Objectivos (4)

- Acesso rápido
  - Escalonamento de pedidos de transferência do disco para minimizar movimentos do braço
    - A ideia é reduzir o tempo médio de acesso a disco
    - Como de costume, ao alterar a ordem de serviço, atrasa alguns pedidos em benefício de outros...
  - Várias estratégias:
    - FIFO
    - SSTF
    - SCAN
    - SCAN circular

Discos

- O tempo necessário para aceder a um bloco é determinado por três factores:
  - Tempo de procura (posicionamento na pista)
  - Tempo de rotação do disco (posicionamento no sector)
  - Tempo de transferência
- O tempo de procura (seek) é dominante



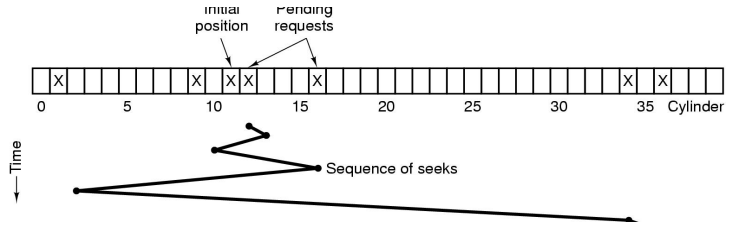
Escalonamento de pedidos de transferência

- FIFO
- SSTF
- Elevator
- Scan circular

Consegue imaginar os algoritmos?

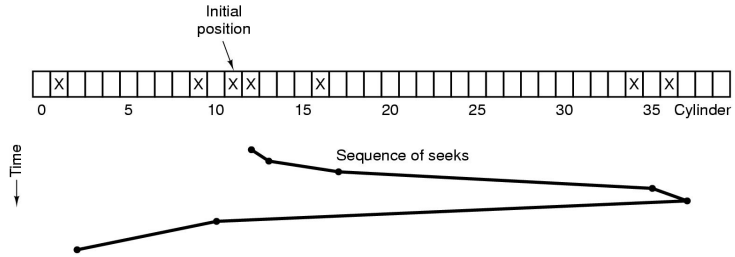
Como bloquear um processo até que chegue a vez do seu pedido?

Escalonamento de pedidos a disco



Shortest Seek First (SSF)

Escalonamento de pedidos a disco



Elevador

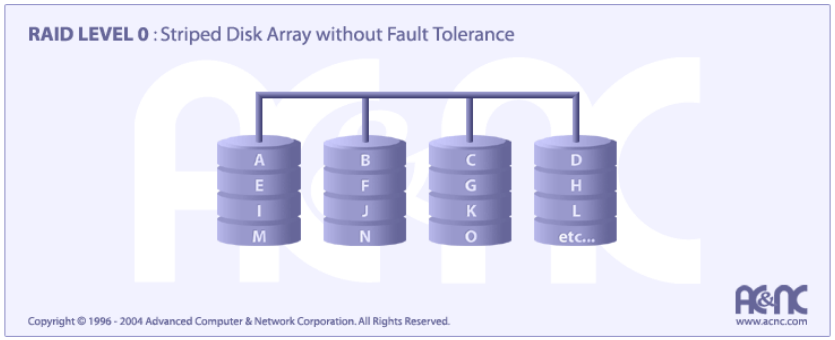
E se um disco tem uma avaria?

?

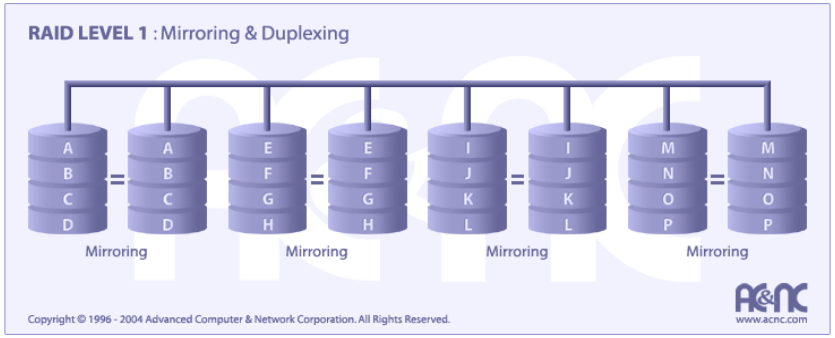
RAID

- Redundant Arrays of Inexpensive Disks
  - Pesquise no google por "Raid-1 Raid-5 primer"
- Objectivos:
  - Desempenho
  - Disponibilidade
    - Tolerância a faltas nos discos (depende do tipo de RAID)
    - Não resolve ficheiros apagados, virus, bugs, etc
      - Continua a precisar de BACKUPs!!

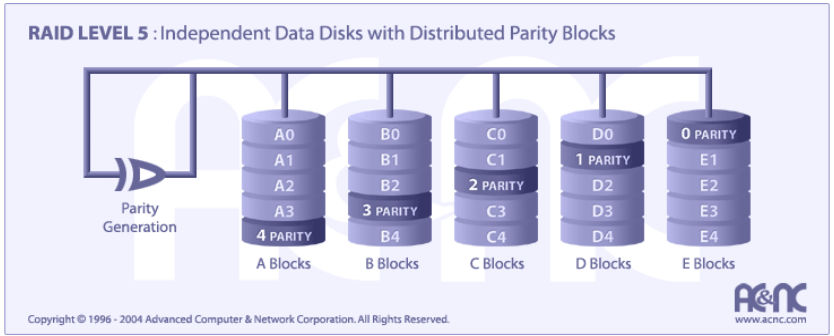
Sistemas RAID



Sistemas RAID



Sistemas RAID



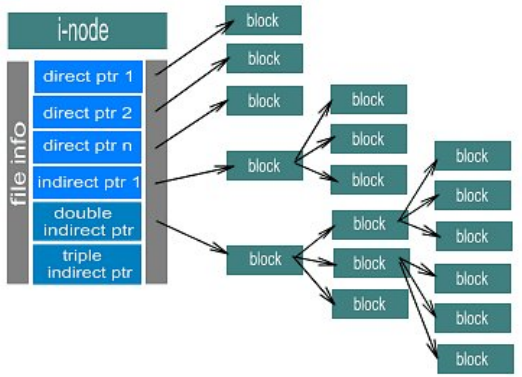
Estudo de casos

- MS-DOS
  - Baseado em FATs
    - File Allocation Tables indicam blocos ocupados por cada ficheiro e ainda os blocos livres na partição
    - Entrada na directoria indica o primeiro bloco do ficheiro. Para localizar o seguinte é preciso seguir a FAT
    - Dimensão da FAT ? Pode obrigar a overlays de partes da FAT
    - Duplicação de FATs para tolerar corrupção

Estudo de casos

- Unix
  - Directorias + I-nodes + data blocks
  - Directorias
    - São ficheiros especiais que fazem a associação nome / i-node
  - I-nodes contêm restantes atributos dos ficheiros, incluindo permissões (ugo), datas e localização dos blocos (até 3 níveis de indirecção)

Estudo de casos



Log-structured File Systems

- Devido à existência de caches em memória, e a necessidade de várias escritas em disco, há hipótese da informação ficar incoerente após crash => corrupção do SF
- FSCK pode demorar muito tempo pois tem de testar todos os meta-dados
  - inaceitável em certos cenários
- É preciso que o sistema de ficheiros recupere depressa

Solução?

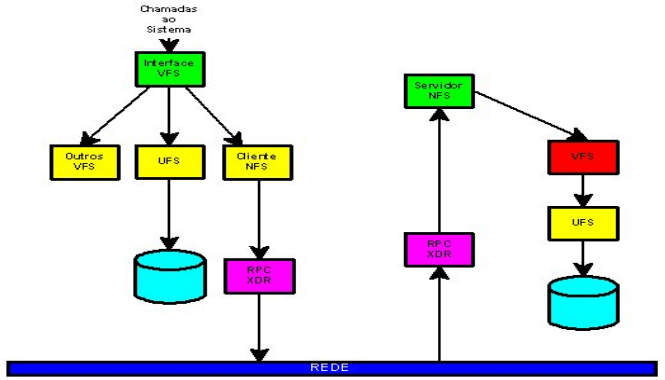
Log-structured File Systems

- A solução passa por utilizar as "boas práticas" dos sistemas de gestão de Bases de Dados...
- SGBDs há muito utilizam Logs para garantir as propriedades ACID (aqui interessa em particular a Atomicidade)
  - SGBD escrevem no Log operações e dados
  - FS tendem a escrever apenas meta-dados (i-nodes, free block allocation maps, i-nodes maps, etc.)

Log-structured File Systems

- Os sistemas de ficheiros baseados em "diário" (Log) mantêm um registo (log) das operações de actualização do SF.
  - As transacções são registadas no Log
  - Em background, as operações indicadas no Log são executadas sobre o sistema de ficheiros e a transacção marcada como committed. Em caso de crash, reexecuta-se apenas o log não completado
  - Checkpointing pode atrasar aplicações
  - Numa leitura, se o bloco pretendido não tiver sido alvo de "checkpoint" há que consultar o log => atraso.

Network File System (NFS)



E ainda...

- Extent-based file systems
- Parallel File Systems
- Distributed File Systems
- Storage Area Networks

...

O "estado-da-arte"

- Perquise no Google por Ext3, XFS, JFS, NTFS, Coda...
- Ou passe algum tempo em http://www.aspsys.com/software/links.aspx/14.aspx

- Se o tempo é limitado, recomenda-se a leitura de
  - Reiser FS (http://www.namesys.com/)

Backups

- Assegure-se que percebe a diferença entre
  - Backup
  - Redundância nos discos, por exemplo Raid-1(mirroring) ou Raid-5

- Backups
  - Para onde? Quando? Que garantias de integridade?

Introdução à Programação Concorrente

Paralelismo versus concorrência

- Execução paralela => hardware
  - Vários computadores, eg. cluster
  - Multiprocessamento, eg. um processo em cada CPU
  - Hyper-threading / Dual core
  - CPU a executar instruções em paralelo com a operação de disco (que se manifesta através de uma interrupção, com prioridade superior à actividade no CPU)

Concorrência

- Criada pelo SO ao repartir tempo de CPU por várias actividades, em resultado de esperas passivas ou desafectação forçada
- Também conhecida por pseudo-paralelismo

Em geral...

- Seja num ambiente de paralelismo real ou simulado pelo SO
- Existem várias "actividades" em execução "paralela"
- Normalmente essas actividades não são independentes, há interacção entre elas
- Este facto que levanta algumas questões...

Papel do SO

- O sistema operativo tem a responsabilidade de
  - Fornecer mecanismos que permitam a criação e interacção entre processos
  - Gerir a execução concorrente (ou em paralelo), de acordo com as políticas definidas pelo administrador de sistemas

Cooperação e Competição

- Em geral, um conjunto de actividades tem 2 tipos de interacção
  - Cooperam entre si para atingir um resultado comum
    - Processo inicia transferência do disco e aguarda passivamente que esta termine
    - O disco interrompe e a rotina de tratamento avisa o processo que pode prosseguir
  - Competem por recursos partilhados (CPU, espaço livre, etc.)
    - Há necessidade de forçar os processos a esperar até que o recurso fique disponível

Sincronização

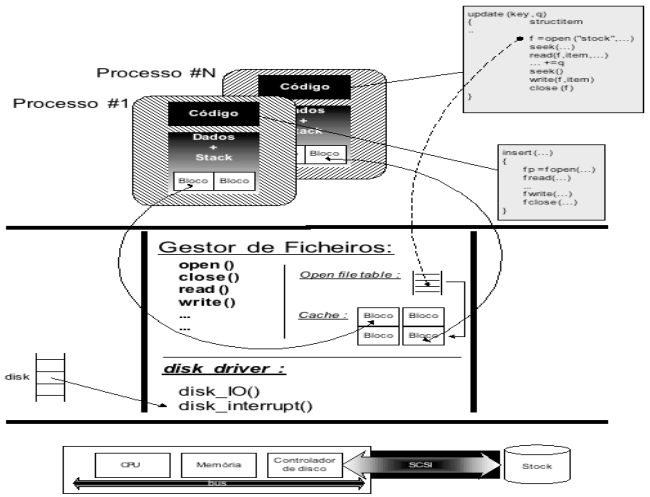
- Em ambos os casos, estamos perante uma questão de sincronização:
  - Cooperação (espera até que evento seja assinalado)
  - Competição (espera até que recurso esteja disponível)
- Sincronizar é atrasar deliberadamente um processo até que determinado evento surja
  - Convém que a espera seja passiva.

Comunicação

- Para haver interacção tem de haver de comunicação:
  - Processos podem requisitar ao SO um segmento partilhado, podem escrever/ler ficheiros comuns, enviar/receber *mensagens* através de "canais", pipelines, sockets, etc.
  - Threads do mesmo processo podem comunicar através de variáveis globais
- A comunicação pode ser tão simples como o assinalar a ocorrência de um evento (de sincronização), ou pode transportar dados

Exemplos

- Acesso a ficheiros partilhados
- Disk IO
- Mirroring de discos
- Impressão



Mais exemplos de concorrência

- Inspirados na vida real
  - Diálogo cliente/bar(wo)men
  - Acesso ao WC
  - Acesso a um parque de estacionamento ou sala de cinema sem marcação de lugar
- São exemplos, respectivamente, de
  - Sincronização
  - Exclusão mútua (caso particular em que capacidade = 1)
  - Controlo de capacidade

Sincronização

BARMAN	CLIENTE
/* aguarda vaga no balcão*/ while (n_copos == MAX) ;	/* aguarda por copo cheio*/ while (n_copos == 0) ;

n_copos = n_copos + 1;
pousar_copo_no_balcao();
...
	n_copos = n_copos – 1;
	tirar_copo_do_balcao();
	…

Acha que o algoritmo anterior está correcto?

- NÃO !!
  - Existem esperas activas => desperdício de CPU
  - Não garante que não haja copos partidos
  - Nem clientes a pegarem no mesmo copo…

Mecanismos de sincronização

- Existem vários
  - Semáforos, mensagens, …
- Nestas aulas usamos apenas semáforos

Semáforos

- Imagine uma caixa com bolas, rebuçados, pedras...
- E as operações seguintes:

P:

Se há bola(s) na caixa, retiro uma e continuo, senão aguardo (passivamente) que alguém deposite uma.

V:

Devolvo a bola à caixa; se há alguém bloqueado à espera, acordo-o

Semáforos

- Servem para resolver problemas de sincronização e de exclusão mútua
- Apenas com 3 operações*:
  - Inicialização :

    s = cria_semáforo (valor_inicial)
  - P (s) ou Down (s)
  - V (s) ou Up (s)

* Na realidade há mais operações (e.g. Trylock)

Semáforos

```
P(s)
{
   s = s - 1
   if  (s <  0)
     then bloqueia("S")
}
```

```
V(s)
{
   s = s + 1
   if  (s ≤ 0)
     then liberta("S")
}
```

Quando s < 0, o seu valor absoluto |s| conta o número de processos bloqueados no P()

Semáforos

- Bloquear significa retirar o processo corrente do estado RUN e inseri-lo na fila "S"
- "S" contém os processos BLOQUEADOS no semáforo S
- Libertar significa escolher um processo da fila "S" e inseri-lo na fila READY
- Normalmente essa escolha é FIFO
- Mas pode não ser...

Sincronização com semáforos

- Para cada evento de sincronização, é criado um semáforo com **valor inicial zero**
- Um processo espera *passivamente* pelo evento, e só avança depois do evento acontecer

```
P(s)        /* se caixa vazia, espera; senão evento já ocorreu */
```

- Outro processo assinala ocorrência do evento

```
V(s)        /* se ninguém à espera, deixa bola na caixa para indicar
               que o evento já ocorreu*/
```


BARMAN

```
/* aguarda por vaga no balcão */
/* e só depois vai... */

pousar_copo_no_balcao();

/* avisa que há +1 copo cheio) */
V(copo)
....
```

CLIENTE

```
/* aguarda por copo cheio */
P(copo)

tirar_copo_do_balcao();

/* avisa que há vaga no balcão
   */
```

Falta inicializar o semáforo copo a Zero

Capacidade

- Para aguardar por espaço no balcão, usa-se um semáforo inicializado à capacidade do recurso partilhado (e não a Zero)
- É um caso particular de sincronização:
  - Só bloqueia se o recurso estiver esgotado naquele instante
  - Equivale a inicializar o semáforo a 0 e de imediato executar tantos V() quantas as posições livres no balcão

BARMAN	CLIENTE
/* aguarda por vaga no balcão */ P(espaço) pousar_copo_no_balcao(); /* avisa que há +1 copo cheio */ V(copo)	/* aguarda por copo cheio */ P(copo) tirar_copo_do_balcao(); /* avisa que há vaga */ V(espaço)

....

Falta inicializar o semáforo copo a Zero e espaço à capacidade do balcão

Exclusão mútua

- Exclusão mútua é também uma forma de "sincronização"
- Talvez aqui a palavra seja (des)sincronização, pois queremos garantir que 2 ou mais processos não estão simultaneamente dentro da região crítica...
- Esqueleto da solução
  - Entrada: /* espera por região livre, e ocupa-a */
  - Código correspondente à região crítica
  - Saída: /* liberta região*/

Exclusão mútua com semáforos

- Para cada região crítica, é criado um semáforo s com valor inicial igual a **UM**
- No início da região crítica

  P(s) /* só avança se região está livre */
- No fim da região crítica

  V(s) /* assinala que a região está livre */

"Receitas" com semáforos

- Sabendo que valor inicial + #V() ≥ #P() concluídos
  - Sincronização:

    valor inicial = 0
  - Capacidade:

    valor inicial = N = capacidade do recurso
  - Exclusão mútua:

    valor inicial = 1

Produtor/consumidor com semáforos

- Agora que resolvemos os aspectos de sincronização
- E já sabemos a "receita" da exclusão mútua
- É altura de reparar que o balcão é uma variável partilhada pelos vários processos (M barmen + N clientes)

  =>Falta garantir exclusão mútua no acesso ao balcão!

Produtor(es)

```
P(espaço_livre);
   P(mutex);
      Buf[p++ % N] = px;
   V(mutex);
V(não_vazio);
```

Consumidor(es)

```
P(não_vazio)
   P(mutex);
      cx = Buf[c++ % N];
   V(mutex);
V(espaço_livre)
```

Falta inicializar os semáforos espaço_livre a N, não_vazio a ZERO, mutex a UM, e as variáveis p e c a ZERO.

Produtor Consumidor

- O exemplo anterior surge frequentemente na comunicação entre processos
  - Utilizam-se buffers múltiplos (porquê?)
  - Tem de ser modificado no caso do "produtor" ser uma rotina de tratamento de interrupções (porquê?)

Threads

- Usam-se para evitar o custo da criação e interacção entre processos quando as actividades "confiam" e mantêm uma estreita colaboração (por exemplo, servidor concorrente que lança uma actividade para cada pedido)
- Partilham o espaço de endereçamento do "processo"
  - Comunicam entre si através de variáveis partilhadas
  - Sincronizam através de variáveis de condição
  - Devem usar mutexes para coordenar acesso a regiões críticas

Threads

- pthread_create(3) - create a new thread
- pthread_detach(3) - detach a thread
- pthread_cond_wait(3) - wait on a condition variable
- pthread_cond_signal(3) - unblock a thread waiting for a condition variable
- pthread_cond_broadcast(3) - unblock all threads waiting for a condition variable

Sincronização de Threads

- Atenção às diferenças entre a sincronização à custa de semáforos e variáveis de condição:
  - Semáforos têm "memória", variáveis de condição não têm.
    - Um pthread_cond_signal sem nenhum thread à espera "perde-se"
    - Um V() sem nenhum processo à espera incrementa o valor do semáforo
  - Com semáforos, só se liberta um processo de cada vez
    - Para libertar todos os processos bloqueados, tem de se executar um ciclo de V().
    - Para libertar todos os threads bloqueados numa variável de condição, deve fazer-se um broadcast

Exercícios

- Barbeiro
- Filósofos
- Parque de estacionamento
- Eco-aventura (barco + corda)
- Lockf
- Implementação de monitores e variáveis de condição
- Spooler
- Escalonamento de pedidos de transferência de disco
- Readers & writers (a.k.a "ponte do porto")
- ...

Dining Philosophers

- 5 filósofos, que repartem a sua vida entre 2 estados:
  - Pensar
  - Comer
- Para comer, sentam-se a uma mesa com 5 garfos,
  - Pegam no garfo esquerdo, se possível
  - Pegam no garfo direito, se possível
  - Comem o esparguete
  - Pousam os garfos e vão pensar mais um bocado

Dining Philosophers

- Os garfos são recursos críticos, um filósofo só come se tiver em seu poder os 2 garfos (esquerdo e direito)
- Aplicando a "receita" da exclusão mútua

```
Come (f){
    P (ESQUERDO(f))
    P (DIREITO(f))
        <come_mesmo>
    V (ESQUERDO(f))
    V (DIREITO(f))
}
```

Nota:

ESQUERDO(x) e DIREITO(x) são macros que indicam os garfos correspondentes ao filósofo X

Dining Philosophers

- Deadlock:
  - Se todos pegarem no garfo esquerdo "ao mesmo tempo", todos param porque não conseguem o direito. Como só devolvem os garfos depois de comer... Ninguém come!
- Starvation:
  - Dois filósofos podem impedir outro de comer

Dining Philosophers

- Soluções?
  - Não deixar entrar na sala mais de 4 filósofos.
  - Ordenar os garfos de modo a que um filósofo comece sempre pelo seu garfo mais baixo. Por exemplo,
    - Filósofo 0: pega no garfo 0 e depois no 1
    - Filósofo 4: pega no garfo 0 e depois no 4

Readers & Writers

- Há um recurso partilhado por 2 classes de utilizadores
  - Leitores
  - Escritores
- Não há necessidade de exigir exclusão mútua entre os leitores, visto que estes não modificam a informação
- Há necessidade de garantir exclusão mútua nos escritores

a.k.a. Ponte do Porto

- 2 classes: automóveis e camiões
- Segurança
  - Quando muito 1 camião na ponte (=> 0 automóveis), ou
  - Qualquer número de automóveis na ponte
- Prioridade
  - Enquanto forem chegando automóveis, o que acontece aos camiões?
  - E vice-versa?

Barco + corda

- Barco tem capacidade N e só avança se estiver cheio
- Na corda só passa um elemento de cada vez
- Barco só regressa depois de todos os elementos terem passado na corda
- Só há um barco...
- Há várias equipas mas para simplificar admita-se que podem ir misturadas (i.e. membros de várias equipas no barco)