BARMAN

```
/* aguarda por vaga no balcão*/
P(espaço)
pousar_copo_no_balcao();
/* avisa que há +1 copo cheio) */
V(copo)
….
```

CLIENTE

```
/* aguarda por copo cheio*/
 P(copo)
tirar_copo_do_balcao();
/* há espaço livre para +1*/
V(espaço)
…
```

Falta inicializar os semáforos espaço e copo a ZERO

Exclusão mútua com semáforos

- Para cada região crítica, é criado um semáforo com **valor inicial 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 */

BA 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 e 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?)

RTInt teclado

```
Disable Interrupts;
If livres == 0 then "PIP" Else
   livres--
    Buf[p++ % N] = px;
Endif
Enable Interrupts;
V(não_vazio);
```

Consumidor(es)

```
P(não_vazio)
Disable_interrupts
cx = Buf[c++ % N];
livres ++
Enable_interrupts
```

Falta inicializar o semáforo não_vazio a ZERO, as variáveis p e c a zero, e livres = capacidade do buffer.

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 interacção 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

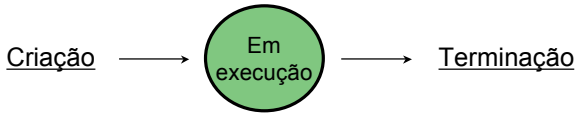
Objectivos

- 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)



Criação de 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

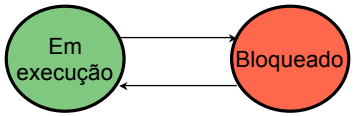
- Possibilidades na execução dos filhos:
  - Pai e filho executam concorrentemente
  - Pai aguarda pelo fim da execução do filho para continuar
- Possibilidades no espaço de endereçamento:
  - O do filho é uma duplicação do do pai
  - O do filho é um programa diferente desde a criação

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
        */
}
```


Estados de um processo (ii)

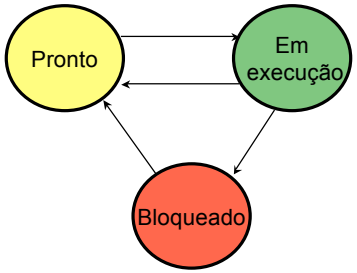
Podemos para já admitir que durante a sua "vida" os processos passam por 2 estados:



O estado Bloqueado corresponde a uma espera passiva

Estados de um processo (iii)

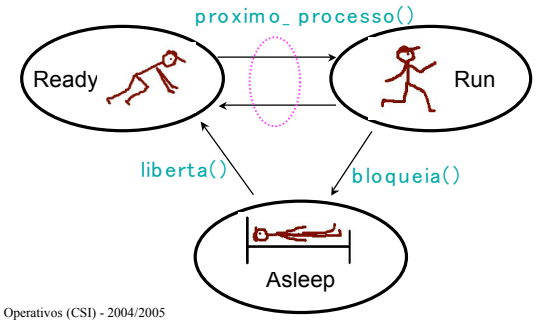
- Na prática, há mais processos não bloqueados do que CPUs
- Surge uma fila de espera com processos **Prontos a executar**
- Processos em execução podem ser retirados do CPU



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?

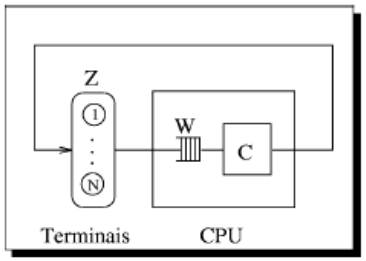
Escalonamento de processos

- 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 ao fim do quantum ou porque surgiu outro de maior prioridade diz-se que o escalonamento é com **desafectação forçada** (preemptive)

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



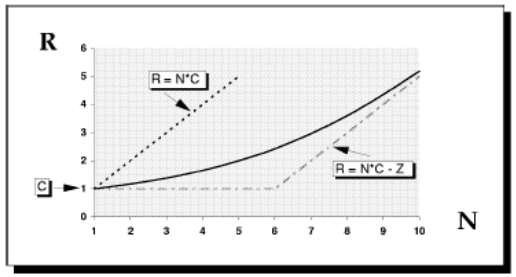
Z = Think time

C = Service time

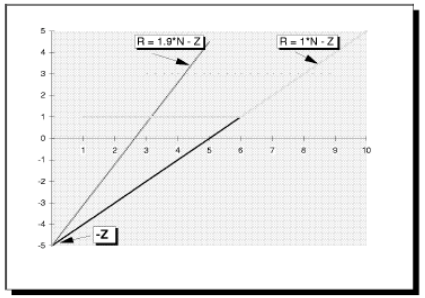
W = Wait time

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)

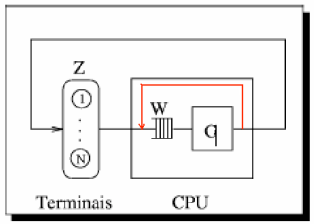
- Para evitar que as interações longas monopolizem o CPU e aumentem o tempo de resposta das restantes deve usar-se desafectação forçada.
- Neste caso deve atribuir-se um quantum (ou time slice) para permitir a troca rápida de processos:
  - Interacções curtas terminam dentro dessa fatia de tempo, logo não são afectadas pela política de desafectação.
  - Interacções longas executam durante 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!

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



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:
  - 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

- Diferentes algoritmos de escalonamento favorecem optimizações diferentes:
  - Tempo de resposta
  - Máxima utilização do CPU

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 selecionado o da frente
- Método cooperativo
- Nada apropriado para ambientes interactivos

FCFS

- Tempo de espera com grandes flutuações dependendo da ordem de chegada e das características dos processos
- Sujeito ao "efeito de comboio"
- Uma vantagem óbvia do FCFS é sua simplicidade de implementação
- Parece haver vantagens em escalonar os processos mais curtos à frente…

SJF (Shortest Job First)

- A ideia é escalonar sempre o processo mais curto primeiro
- Possibilidades:
  - Desafectação forçada (SRTF) - interrompe o processo em execução se aparecer um mais curto
  - Cooperativo – aguardar pela terminação do processo em execução mesmo na presença de um processo recente mais curto

SJF

- Não se consegue adivinhar o tempo de processamento dos processos
- Apenas se podem fazer estimativas
- Usa uma combinação de tempos reais e suas estimativas para fazer futuras previsões.

Preemptive Priority

- Associa uma *prioridade* (geralmente um inteiro) a cada processo.
- A *ready queue* é uma fila seriada por prioridades.
- Escalona sempre o processo na frente da fila.
- Se aparece 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)

- Dá a cada processo um intervalo de tempo fixo de CPU de cada vez
- Quando um processo esgota o seu quanto retira-o do CPU e volta a colocá-lo no fim da fila.
- 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 o 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)

Multi-Level Queues (MLQ)

- RR tem apenas uma fila, mas podia ser optimizado
  - e.g. se só usou metade da fatia disponível, é inserido no meio da fila
- MLQ divide os processos em várias filas, consoante o tipo de processo e o tempo de CPU consumido
- Pode haver feedback
  - Se esgotou fatia de tempo, vai para fila de prioridade mais baixa;
  - Se mostra ser interactivo, sobe

Estudo de casos

- Unix, Solaris, Windows XP, ...
- Suportam simultaneamente
  - Processos do sistema: prioridade elevada, por vezes fixa
  - Processos interactivos: fatias de tempo, prioridade dinâmica
  - Processos longos, em background

Programa

- Introdução, conceitos de programação de sistemas e noções de gestão de periféricos
- Gestão de processos e noções de programação concorrente
- Gestão de memória
- Gestão de ficheiros e noções de sistemas distribuídos

Gestão de Memória

- Idealmente a memória seria:
  - Inesgotável, rápida e não volátil
- Mas na realidade temos

  Cache: Rápida, pequena, cara (e quase "invisível")

  RAM: Velocidade média, custo aceitável

  Disco: Gigabytes de memória lenta e barata

Como gerir esta hierarquia?

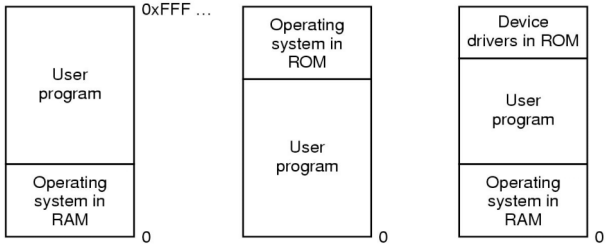
Gestão de Memória

- Para complicar
  - Cache e RAM são endereçáveis directamente, mas são *voláteis*
  - Disco são *persistentes*, mas obrigam a **operações de IO**
- E ainda...
  - Há mais tipos de memória secundária, e.g.
    - Tape, CD-ROM, DVD
  - Recursos podem ser locais ou remotos,
    - Por exemplo, numa rede rápida pode ser preferível ir buscar um bloco à RAM de outro computador do que ao meu próprio disco (faça as contas!)

Gestão de Memória

- Memória real
  - Monoprogramação
  - Multiprogramação
    - Partições de dimensão fixa
    - Partições de dimensão variável
- Memória virtual
  - Segmentação
  - Paginação
  - Segmentação + paginação

Monoprogramação



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

Monoprogramação

- É inconveniente
  - Memória de dimensão insuficiente (=> overlays)
  - Complicada de alocar
    - Onde começa a memória do programa?
    - Onde começa a zona de dados?
- É (muito) pouco eficiente
  - Memória parcialmente ocupada
  - CPU parado durante operações de IO

Swap

- Para aumentar a eficiência
  - Se um programa se bloquear durante algum tempo
    - Pode ser copiado para a Área de Swap em disco (Swap out), libertando a memória central
    - Outro programa que lá esteja, e já possa executar, é "Swapped in" e é-lhe atribuído o CPU
- Primeiros sistemas de time-sharing funcionavam assim (por exemplo CMS), se bem que a eficiência ainda fosse baixa. Porquê?

Multiprogramação

- Pensada para aumentar a eficiência
- Tendo mais do que um programa em memória central
  - Se o programa em execução se bloquear à espera de algum evento, escolhe-se imediatamente outro que esteja pronto a executar (Ready)
  - Não se perde tempo com SwapIn e SwapOut

Multiprogramação

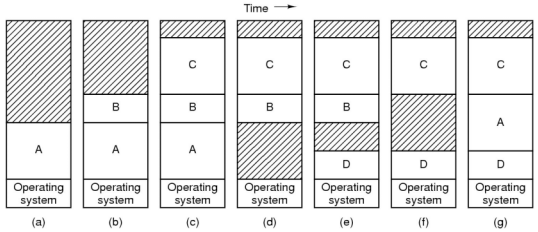
- Mas...
  - Ao multiprogramar estou a dividir a memória disponível, dando menos a cada programa (incluindo espaço para ele crescer).
  - Complico a alocação
    - Em que partição vai correr o programa? Onde começa? São todas do mesmo tamanho? Será que vou ter de fazer overlays outra vez?
  - E tenho de isolar as partições => protecção

Analogia com um restaurante

- Só mesas de 4? Algumas de 2, outras de 4 e, à cautela, uma de 8?
- Tenho uma mesa de 4 livre e chega uma pessoa. Deixo-a sentar-se na mesa de 4 ou espero? Porquê?
- Posso juntar uma de 2 a uma de 4? Que custo tem isso? E se a de 2 está no outro lado da sala?
- Posso pedir às pessoas de uma mesa que passem para a do lado? Qual o custo?

E se as mesas tivessem capacidade (dimensão) variável?

Partições de dimensão variável



- Neste exemplo o Processo A sai da memória (Swapped Out) e regressa para outro local => recolocação dinâmica
- Na transição f-g, se não existisse espaço para o programa A, podia-se COMPACTAR a memória

Recolocação e Protecção

Recolocação:

- Compilador gera endereços a partir de Zero
- Registo base aponta para o início da partição
- Se o programa mudar de local, basta mudar a base
- Mas os endereços têm de ser sempre somados à base...

Protecção:

- Endereços legais variam entre Zero e o valor indicado no **Registo Limite**
- Todos têm de ser comparados com o limite...

Gestão de Memória: estratégias

- Pense nos custos associados a cada uma:
  - First-fit
  - Best-fit
  - Worst-fit
  - Pré-selecção de tamanhos "populares"
  - Buddy system
  - …

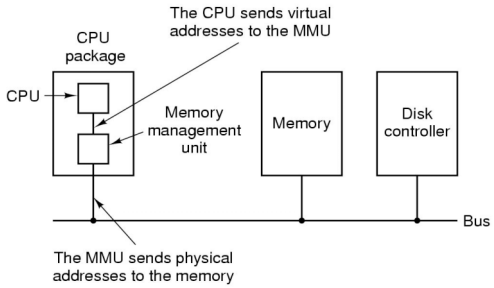
Você disse "eficiência"?

- Começamos a ver que afinal a memória está pouco ocupada:
  - Fragmentação
    - interna (c.f. mesas parcialmente ocupadas)
    - externa (c.f. mesas livres mas separadas)
  - Dispersão de referências
    - estática (c.f. reserva mesa de 4 mas afinal só aparecem 2)
    - dinâmica (c.f. durante a refeição, cliente passa o tempo a ausentar-se)
  - Não permite partilha
- E é complicada de gerir

Memória real baseada em Partições

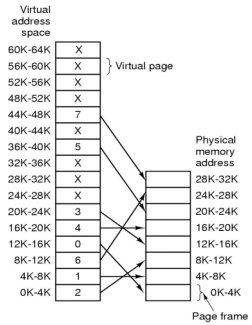
- É *Inconveniente*:
  - Restringe a dimensão máxima dos programas
  - Alocação contígua não facilita atribuição de endereços
  - Não permite protecção selectiva (read/write/execute)
- É *Ineficiente*:
  - Causa fragmentação
  - Não tira partido da dispersão de referências
  - Não permite partilha

Solução? Memória Virtual!

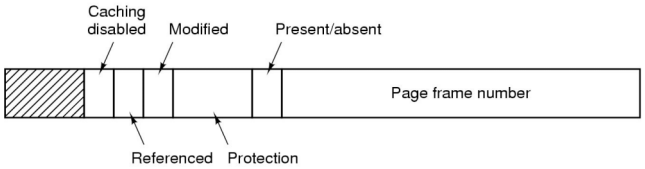


Memória Virtual

- A relação entre endereços virtuais e reais é dada por uma tabela
- Esta tabela mantém a ilusão de contiguidade de endereços
  - Por exemplo, o bloco 12k-16k está logicamente contíguo ao 8k-12k, embora na realidade estejam em locais distintos da memória central



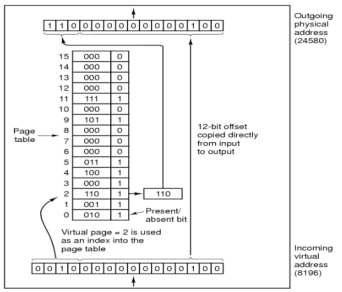
Memória Virtual



Entrada típica da tabela de páginas

Memória Virtual

Operação da MMU com 16 páginas de 4 KB



- Note que as páginas
  - 6-8
  - 10
  - 12-15
- não estão acessíveis ao processo em execução

Tradução de Endereços

- Actualmente a RAM é grande e as páginas continuam a ser pequenas (porquê?)
  - Torna-se impraticável usar indexação
  - Por exemplo, com RAM de 1GB e páginas de 4KB
    - A MMU teria de ter (potencialmente) 256K entradas...
- Que fazer?

Translation Lookaside Buffers (TLBs)

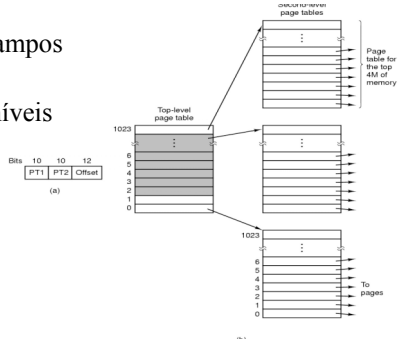
- Não precisamos de traduzir todos os endereços, só os que estão a ser precisos numa determinada altura
- Basta ter uma...

Cache de endereços!

Valid	Virtual page	Modified	Protection	Page frame
1	140	1	RW	31
1	20	0	R X	38
1	130	1	RW	29
1	129	1	RW	62
1	19	0	R X	50
1	21	0	R X	45
1	860	1	RW	14
1	861	1	RW	75

Memória Virtual

- Endereço de 32 bit c/ 2 campos para tabelas de páginas
- Tabelas de páginas de 2 níveis



Paginação

- Procura resolver os problemas da organização física da memória central, por exemplo
  - Dar a ilusão que a memória é muito grande
  - Facilitar a alocação de espaço: blocos pequenos e de tamanho fixo são fáceis de colocar em qualquer local que esteja livre
  - Tirar partido da dispersão de referências: só carregar para memória o conjunto de blocos que está a ser referenciado
- É invisível aos programas (se bem que os programas possam "ver" uma memória muito maior do que a real)

Segmentação

- Procura resolver os problemas da organização lógica do espaço de endereçamento de *cada* processo, por exemplo,
  - Multiplas zonas de dimensão variável para código, libs, dados, stack, mapped files, ...
  - Partilha de bibliotecas
  - Protecção selectiva: código r-x, dados rw- ou r—
- É visível aos programas (=> syscalls que conhecem "zonas" de memória)

Segmentação + Paginação

- Combinam os benefícios das duas técnicas
- Em vez de carregar segmentos completos (problema idêntico ao das partições de dimensão variável), de cada segmento apenas estão em memória central as páginas que estão a ser usadas

Carregamento a pedido

- Se um bloco não está em memória e é feita uma referência
  - Gera-se uma interrupção, segment ou page fault
  - Gestor de memória virtual verifica a causa da interrupção
    - Se acesso ilegal, termina programa
    - Se acesso legal,
      - Solicita carregamento do disco para RAM, ou
      - Atribui bloco em RAM (e.g. Malloc)
      - ...

Rejeição de Páginas

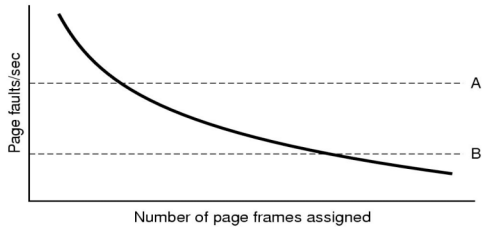
- A memória central vai enchendo à medida que as páginas vão sendo criadas ou carregadas
- Ocasionalmente um processo termina e liberta RAM
- Se a memória encher (ou talvez um pouco antes...)
  - Há necessidade de rejeitar, ou seja, retirar uma ou mais

Rejeição de páginas

- É necessário decidir qual a página a rejeitar
  - Uma página que foi modificada tem que ser escrita de volta no disco => demora tempo
  - Uma página não modificada pode ser usada imediatamente
- Convém não rejeitar uma página frequentemente usada
  - Pois provavelmente terá de ser carregada a seguir
- Como saber que páginas estão a ser usadas, sem causar grande overhead?

Alocação Local e Global

Número de page faults como função do número de páginas atribuídas ao processo



Rejeição de páginas

- Local ou global?
- Estratégias
  - FIFO
  - LRU
  - NRU
  - Second-chance
  - Clock

Working set

- O gestor de memória procura manter em RAM o conjunto de páginas que cada processo necessita (está a usar). É o chamado working set
- Se começar a rejeitar páginas do working set de outro processo, esse processo volta logo a pedi-las...
- E isso demora tempo e causa overhead

Thrashing...

- Podemos chegar a uma situação em que para carregar uma página de um processo tenho de rejeitar uma que pertence ao working set de outro
- Que por sua vez, para a trazer de volta vai rejeitar uma página do working set do processo inicial!
- O disco sofre... (thrash!)
- O CPU está ocupado a decidir qual rejeitar a seguir
- E os processos estão à espera das suas páginas

Thrashing

- Sintomas:
  - Utilização elevada do disco, constantes transferências entre a área de swap e a memória central
  - Elevada utilização do CPU em modo supervisor (para executar algoritmos de rejeição e tratamento de interrupções)
  - Processos com grandes working sets quase "parados"

Soluções?

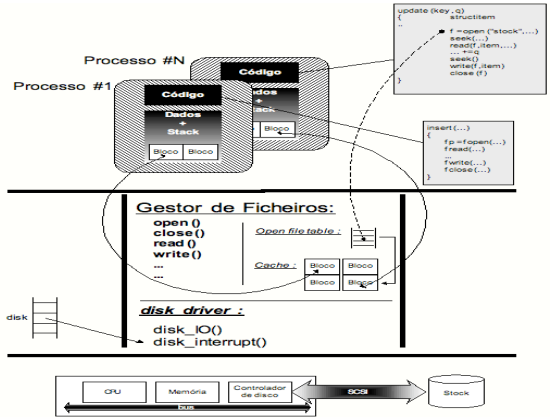
- Aumentar a RAM
- Reorganizar programas de modo a consumirem menos memória
- Reduzir grau de multiprogramação, por exemplo rejeitando processos inteiros => Page daemon + Process swapper
- Escalonamento de 3 níveis
- Aumentar RAM? Aumentar RAM?
- Distribuir a carga por vários computadores?

Programa

- Introdução, conceitos de programação de sistemas e noções de gestão de periféricos
- Gestão de processos e noções de programação concorrente
- Gestão de memória
- Gestão de ficheiros e noções de sistemas distribuídos

Sistemas de Ficheiros

- Objectivo:
  - Armazenamento de informação de forma persistente
- Tipicamente os dados e *meta-dados s*ão guardados em disco
  - Mas existem cópias em RAM para acelerar o acesso
  - Isto pode causar problemas...



Por falar em Distribuir a Carga…

- Nos Sistemas Distribuídos há que estudar questões como
  - Desempenho
  - Dispersão Geográfica
  - Localização dos dados (locais ou remotos)
  - Existência de falhas: comunicação, aplicações ou sistemas
  - Ordem de eventos

    ...

Mas...

- O curso não termina aqui!
- Espero que tenha ficado com uma noção mais clara de como funcionam os sistemas informáticos
- E para que servem os sistemas operativos...