Sistemas Operativos

Índice

O sistema operativo

Os primeiros sistemas informáticos não tinham sistema operativo, nem nada semelhante, apenas quando o hardware começou a melhorar se começou a pensar em otimizações a nível do software. Foram criados os primeiros programas que simplificaram as operações do computador.

Podemos pensar no sistema operativo como uma máquina virtual que abstrai os detalhes da máquina física, disponibiliza também um conjunto de recursos lógicos e interfaces para os gerir e programar.

Objetivos

- Conveniência:
  - SO esconde os detalhes do hardware
  - Simula máquina virtual com valor acrescentado (cada processo executa numa "máquina" protegida)
  - Fornece API mais fácil de usar do que o hardware (ficheiros vs. blocos em disco)
- Eficiência:
  - SO controla a alocação de recursos:
    - Se 3 programas usarem a impressora ao mesmo tempo à sai lixo?
    - Programa em ciclo infinito -> computador bloqueia?
    - Processo corrompe a memória dos outros ao programas morrerem?
  - Multiplexação:
    - Tempo: cada processo usa o recurso à vez (impressora, CPU)
    - Espaço: recurso é partilhado (memória central, disco)

Papel do Sistema Operativo

Gestor de recursos

O sistema operativo gere recursos lógicos, mapeando-os para os seus correspondentes físicos;

Ficheiros <-> Espaço em disco;

Processos <-> Processador;

Periféricos virtuais <-> Periféricos físicos.

Desta forma o sistema operativo torna os recursos lógicos genéricos e reutilizáveis;

Lista de recursos lógicos geridos pelo sistema operativo:

- Processos (estudados detalhadamente nos próximos capitulos);

- Memória virtual (estudada também mais a frente);
    - Gestão de espaços de endereçamento.
- Sistema de ficheiros;
- Periféricos;
    - Gere qualquer periférico através de periféricos digitais, obtendo um grau de abstracção.
- Utilizadores.
    - Identidade;
    - Privilégios.

2. Interface

- Interface operacional;
    - Shell (bash, zsh, etc);
    - GUI (Linux, Apple, Windows).
- System calls
    - Permitem executar operações associadas aos objectos do sistema;
    - Os objectos do sistema representa os recrusos lógicos e os seus metodos associados.
    - Fazem parte do modelo computacional dos sistemas opertativos.

3. Máquina virtual

- O sistema operativo é uma maquina virtual sobre a máquina fisica.
- Facilidade na sua utilização.

O consumo de recursos no acesso livre

O utilizador é um faz-tudo:

- Carrega manualmente o programa e dados;
- Executa o programa
- Se não está correto, tem de descobrir sozinho os erros
- Eficiência é muito baixa devido ao desperdício de tempo

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 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)

Mas havia o risco de:

- 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

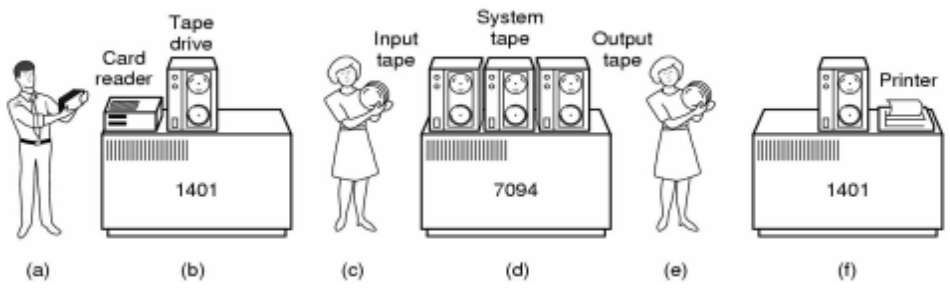
Cada utilizador tem um determinado tempo atribuído só para si, podem utilizar o computador exclusivamente.

Durante maior parte do tempo o computador está parado a espera do input do utilizador ou de operações Input/Output.

Seguiram-se então os sistemas de **tratamento em batch**. Sendo que **maior parte do tempo de processador era gasto em operações I/O (espera activa)**, a solução foi separar os periféricos de I/O. Começaram também a surgir os primeiros dispositivos de memória secundária.

Um computador auxiliar lia para a banda magnética os programas a executar, quando o trabalho em curso terminasse o sistema operativo ia a lista de trabalho e selecionava o próximo a executar. Alem disso, em vez de imprimir o resultado dos programas diretamente, os ficheiros são enviados para uma impressora quando acabam de executar.

Os periféricos avisam o processador no fim da sua execução, através de interrupções.



Teve de esperar pelos sistemas de Time-Sharing para haver mais conveniência:

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

Soluções (hardware)

- Interrupções
- Relógio de Tempo Virtual
- Instruções privilegiadas, 2 ou mais modos de execução
- Proteção de memória
- Exemplos: Polling IO e Interrupt-driven IO

Polling IO

Carrega o controlador de disco com parâmetros adequados (pista, sector, endereço de memória, direcção...)

- While (NOT IO_done) /* do nothing*/

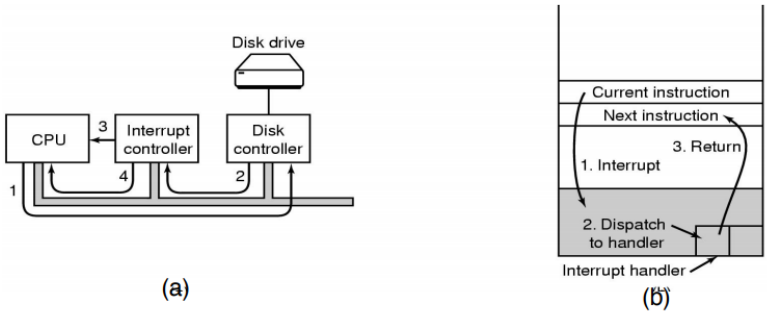
  (Equivalente a Já acabaste? Já acabaste? Já acabaste? Já acabaste? Já acabaste? Já acabaste? Já acabaste? Já acabaste? Já acabaste? Já acabaste? Já acabaste? Já acabaste?...)

- OK, regressa de disk_io() Resulta em desperdício de tempo de CPU

Interrupt-driven IO

(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



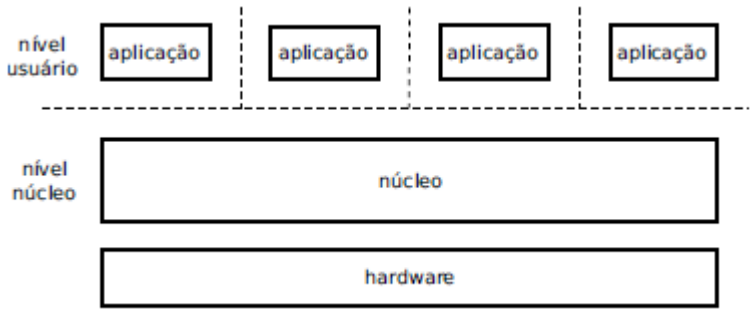
Níveis de execução

Kernel mode/space

Para um código executando nesse nível, todo o processador está acessível: todos os recursos internos do processador (registradores e portas de entrada/saída) e áreas de memória podem ser acessados. Além disso, todas as instruções do processador podem ser executadas. Ao ser ligado, o processador entra em operação neste nível.

User mode/space

Neste nível, somente um sub-conjunto das instruções do processador, registradores e portas de entrada/saída estão disponíveis.
Instruções "perigosas" como HALT (parar o processador) e RESET (reiniciar o processador) são proibidas para todo código executando neste nível. Além disso, o hardware restringe o uso da memória, permitindo o acesso somente a áreas previamente definidas. Caso o código em execução tente executar uma instrução proibida ou acessar uma área de memória inacessível, o hardware irá gerar uma exceção, desviando a execução para uma rotina de tratamento dentro do núcleo, que provavelmente irá abortar o programa em execução.



Interrupções

Interrupções/excepções podem ser geradas tanto a partir do software como do hardware.

Alguns exemplos:

1. Periféricos I/O
   - Quando acabam as suas operações "avisam" o sistema operativo através de uma chamada ao sistema (que gera uma interrupção).
2. Excepções
   - Quando uma instrução executa uma operação inválida, e.g: dividir por zero, aceder a uma zona de memória não permitida.
3. Timers do CPU.

A interrupção transfere o contexto de execução para a rotina apropriada que vai lidar com o "problema". Visto que apenas algumas interrupções são permitidas existe uma tabela de apontadores, chamada de interrupt vector para as rotinas de tratamento de excepções. Cada rotina é chamada indiretamente através da tabela, sem nenhuma rotina/programa intermédia/o. Lidar com interrupções fica menos custoso, em termos de tempo, para o sistema operativo.
O controlo de interrupções é o seguinte:

1. Um processo do utilizador gera uma interrupção e o Program Counter e o estado do processador são guardados num stack especial do kernel, ficando o processo bloqueado.
2. O controlador de interrupções decide que tipo de interrupção foi gerada e chama a rotina apropriada do kernel para lidar com ela.
3. O estado geral do processador é guardado (visto que mudamos de contexto).
4. O kernel executa a rotina apropriada para lidar com a interrupção.
5. Se esta rotina foi rapida o kernel volta para o proesso que estava a executar, se não o processo entra para a lista de processos candidatos a CPU.
6. O Estado do processo é carregado para os registos e o controlo é novamente transferido para o processo.

De notar que o CPU muda o contexto entre user space e kernel space, onde todas as operações são permitidas. Esta mudança é feita internamente pelo hardware.

Dado as exceções serem assíncronas tem que haver forma de lidar com aparecimento de várias exceções em simultaneo. Este controlo é feito com dois mecanismos. O primeiro inibe interrupções depois de uma ter sido aceite. Assim se uma interrupção tiver a ser tratada, não é possível mudança de contexto ao aparecer outra. O segundo inibe interrupções dado a sua gravidade, podendo interromper rotinas de tratamento de exceções com menor prioridade.

O returno a modo de user é feito pela instrução de terno de interrupção **RIT**

Soluções (software)

- Chamadas ao Sistema
- Virtualização de periféricos
- Multiprogramação

Multiprogramação

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

O mecanismo de interrupções permite multiplexar o processador. A capacidade de alternar a execução **não** é limitada a um programa e periféricos I/O mas pode ser estendida a **vários programas em memória**.
Um programa que queira aceder a um ficheiro em disco fica bloqueada enquanto o controlador de disco atua. Durante este tempo outro programa pode ser executado pelo processador. Desta forma conseguimos otimizar a utilização do processador, tendo sempre algo para fazer.

Chamadas ao Sistema

O confinamento de cada aplicação na sua área de memória, imposto pelos mapeamentos de memória realizados nos acessos em nível usuário, provê robustez e confiabilidade ao sistema, pois garante que uma aplicação não poderá interferir nas áreas de memória de outras aplicações ou do núcleo. Entretanto, essa proteção introduz um novo problema: como chamar, a partir de uma aplicação, as rotinas oferecidas pelo núcleo para o acesso ao hardware e suas abstrações? Em outras palavras, como uma aplicação pode acessar a placa de rede para enviar/receber dados, se não tem privilégio para acessar as portas de entrada/saída correspondentes nem pode invocar o código do núcleo que implementa esse acesso (pois esse código reside em outra área de memória)?

A resposta a esse problema está no mecanismo de interrupção que ao ser executada, comuta o processador para o nível privilegiado e procede de forma similar ao tratamento de uma interrupção. Por essa razão, esse mecanismo é denominado interrupção de software, ou **trap.** A ativação de procedimentos do núcleo usando interrupções de software (ou outros mecanismos correlatos) é denominada chamada de sistema (system call ou syscall).

Evolução dos sistemas operativos

Depois do time-sharing apostou-se no computador pessoal, sendo que com isto voltava-se à monoprogramação, e como consequência, baixa eficiência, no entanto era mais conveniente e mais barato para o utilizador.Para colmatar as diferenças que havia em relação ao time-sharing desenvolveu-se:

- Multiprocessamento
- Sistemas Distribuídos

Multiprocessamento

Com o exemplo de 2 CPUs:

- A ideia é suportar 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

Pode ser:

- 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 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

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.

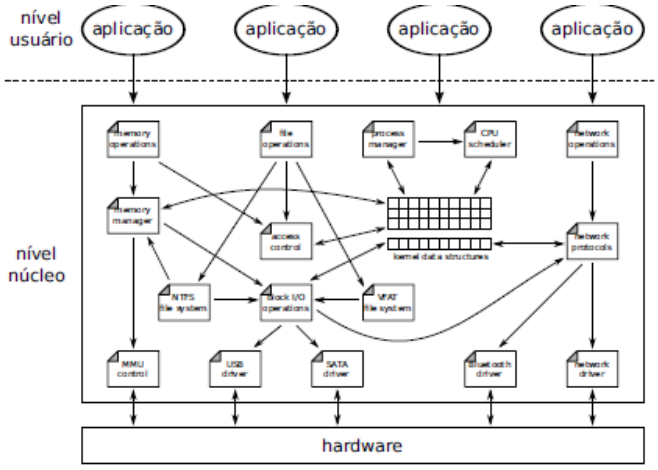
Outros tipos de sistemas

- SO de tempo Real, há 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, Raspberry PI, Arduino.
- Renovou-se o interesse pela virtualização: Xen, Vmware, Hyper V…

Arquiteturas de Sistemas Operativos

Sistemas monolíticos

Em um sistema monolítico, todos os componentes do núcleo operam em modo núcleo e se inter-relacionam conforme suas necessidades, sem restrições de acesso entre si, pois o código no nível núcleo tem acesso pleno a todos os recursos e áreas de memória.



A grande vantagem dessa arquitetura é seu desempenho: qualquer componente do núcleo pode acessar os demais componentes, toda a memória ou mesmo dispositivos periféricos diretamente, pois não há barreiras impedindo esse acesso.

Sistemas em camadas

Uma forma mais elegante de estruturar um sistema operacional faz uso da noção de camadas: a camada mais baixa realiza a interface com o hardware, enquanto as camadas intermediárias provêem níveis de abstração e gerência cada vez mais sofisticados. Por fim, a camada superior define a interface do núcleo para as aplicações (as chamadas de sistema). Essa abordagem de estruturação de software fez muito sucesso no domínio das redes de computadores, através do modelo de referência OSI.

Sistemas micro-núcleo

Uma outra possibilidade de estruturação consiste em retirar do núcleo todo o código de alto nível (normalmente associado às políticas de gerência de recursos), deixando no núcleo somente o código de baixo nível necessário para interagir com o hardware e criar as abstrações fundamentais (como a noção de atividade).Usando essa abordagem o código de acesso aos blocos de um disco rígido seria mantido no núcleo, enquanto as abstrações de arquivo e diretório seriam criadas e mantidas por um código fora do núcleo, executando da mesma forma que uma aplicação do usuário. Por fazer os núcleos de sistema ficarem menores, essa abordagem foi denominada micro-núcleo.

Máquinas virtuais

É possível contornar os problemas de compatibilidade entre os componentes de um sistema através de técnicas de virtualização. Usando os serviços oferecidos por um determinado componente do sistema, é possível construir uma camada de software que ofereça aos demais componentes serviços com outra interface. Essa camada permitirá assim o acoplamento entre interfaces distintas, de forma que um programa desenvolvido para uma plataforma A possa executar sobre uma plataforma distinta B. O sistema computacional visto através dessa camada é denominado máquina virtual.

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)

Processo: um programa em execução, tem atividade 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.)

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

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

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

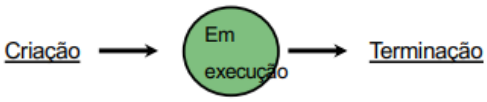
- Para executar outro programa:

  – exec: substitui o programa do processo corrente por um novo programa

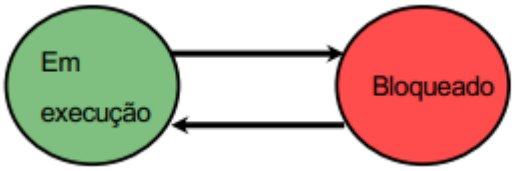
- Para terminar a execução:

  – exit

Estados de um processo



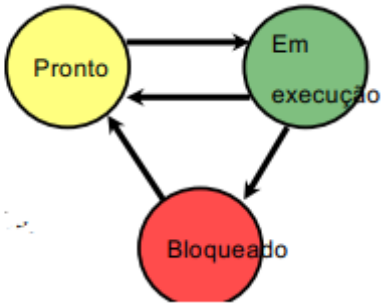
Na prática, raros serão os processos que conseguem executar continuamente do princípio ao fim. Muito provavelmente terão de aguardar por eventos que ainda não ocorreram (e.g. bloco lido do disco, caracter do teclado...). Assim, durante a sua “vida” os processos passam por 2 estados:



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.



● Em execução – Foi-lhe atribuído o/um CPU/núcleo, corre o programa correspondente

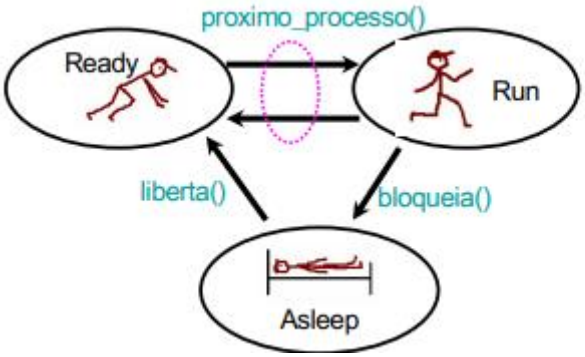
● Bloqueado

– 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



●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()

● 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)

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 prioridade.

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

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.

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.

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)

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

A decisão de escalonar um processo pode ser tomada em diversas alturas:

– Quando um processo passa de a-executar a bloqueado

– Quando um processo passa de a-executar a pronto

– Quando se completa uma operação de I/O

– Quando um processo termina

Tempo de Resposta

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.

Alguns algoritmos de escalonamento

FCFS (First Come, First Served)

- A ready list é uma fila FIFO
- Os processos são colocados no fim da fila e é selecionado o que se encontra na cabeça 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

Preemptive Priority Scheduling

● 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

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
- 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

Gestão de Memória

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

O CPU carrega instruções apenas a partir da memória, o que implica que para que os programas corram tenham que estar em memória. A grande maioria dos programas correm os programas numa memória volátil, a RAM, que vai ser chamada de memória principal.

Esta, e outras formas de memória, disponibilizam um array de bytes. Cada byte tem o seu endereço. Os bytes interagem entre si atráves de instruções de load ou store referidas a endereços de memória específicos. A load move um byte, ou uma word, da memória principal para um registo do CPU e a store faz a operação inversa. De notar que a unidade de memória apenas 'vê' uma série de endereços de memória, é totalmente diagnostica à forma como foram criados ou o seu conteúdo.

Idealmente os processos estariam todos presentes na memória virtual mas isso não é possível.

Surge então a memória virtual que simula um espaço de memória completo para cada processo sendo que na realidade apenas parte do processo se encontra na memória principal..

Paginação

Existem situações onde não é possível manter todos os processos na memória.

A paginação permite que o programa possa ser espalhado por área não contíguas de memória

Características

- O espaço de endereçamento lógico de um processo é divido em páginas lógicas de tamanho fixo.
- A memória física é dividia em páginas com tamanho fixo, com tamanho igual ao da página lógica.
- O Programa é carregado página a página, cada página lógica ocupa uma página física
- As páginas físicas não são necessariamente contíguas
- O endereçamento lógico é inicialmente dividido em duas partes: um número de páginas lógicas e um deslocamento dentro da página.
- O número da página lógica é usado como índice no acesso a tabela de páginas, de forma a obter o número da página física correspondente.
- Não existe fragmentação externa.
- Existe fragmentação interna (EX: um programa que ocupe 201kb. O Tamanho da página é de 4kb, serão alocadas 51 páginas resultando em uma fragmentação interna de 3kb.)
- A transferência das páginas de processo podem ser transferidas para a memória sob demanda, levando apenas o que é necessário para a execução de programas ou por paginação antecipada, onde o sistema tenta prever as páginas que serão necessárias para a execução do programa.

Páginas constantemente referenciadas por um processo devem permanecer na memória.

Rejeição de páginas FIFO:

●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

● 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”

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

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

Segmentação

- Técnica de gerência de memória onde programas são divido em segmentos de tamanhos variados cada um com seu próprio espaço de endereçamento.
- A principal diferença entre a paginação e a segmentação é a alocação da memória de maneira não fixa, a alocação na segmentação depende da lógica do programa.
- O Mapeamento é feito através das tabelas de mapeamento de segmentos.
- Os endereços são compostos pelo número do segmento e um deslocamento dentro do segmento.
- Cada entrada na tabela mantém o endereço físico do segmento, o tamanho do segmento, se ele está ou não na memória e sua proteção.
- O Sistema operacional mantém uma tabela com as áreas livres e ocupadas da memória.
- Somente segmentos referenciados são transferidos para a memória principal.
- Ocorre fragmentação externa.
- Sistemas que implementam a segmentação com paginação. Cada segmento é dividido fisicamente em páginas.
- O endereço é formado pelo número do segmento número da página dentro desse segmento e o deslocamento dentro dessa página.

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

Objectivos

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

Acesso

– Escalável

– Conveniente

● estrutura interna visivel (pelo kernel) ou só pelas aplicações?

– Sequencia de bytes vs. Ficheiros indexados

– Seguro

● controlo de acessos

● auditoria

● privacidade...

-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?

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

RAID

Apesar dos avanços dos sistemas de armazenamento em estado sólido (como os dispositivos baseados em memórias flash), os discos rígidos continuam a ser o principal meio de armazenamento não-volátil de grandes volumes de dados. Os discos atuais têm capacidades de armazenamento impressionantes: encontram-se facilmente no mercado discos rígidos com capacidade da ordem de terabytes para computadores domésticos.

Um sistema RAID é constituído de dois ou mais discos rígidos que são vistos pelo sistema operacional e pelas aplicações como um único disco lógico, ou seja, um grande espaço contíguo de armazenamento de dados. O objetivo central de um sistema RAID é proporcionar mais desempenho nas operações de transferência de dados, através do paralelismo no acesso aos vários discos, e também mais confiabilidade no armazenamento, usando mecanismos de redundância dos dados armazenados nos discos, como cópias de dados ou códigos corretores de erros.

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!!

RAID 0

Neste nível os discos físicos são divididos em áreas de tamanhos fixo chamadas fatias ou faixas (stripes). Cada fatia de disco físico armazena um ou mais blocos do disco lógico. As fatias são preenchidas com os blocos usando uma estratégia round-robin.

RAID 1

Neste nível, cada disco físico possui um “espelho”, ou seja, outro disco com a cópia de seu conteúdo, sendo por isso comumente chamado de espelhamento de discos.

RAID 5

Esta abordagem também armazena informações de
paridade para tolerar falhas em discos. Todavia, essas informações não ficam concentradas em um único disco físico, mas são distribuídas uniformemente entre todos eles.

Introdução à Programação Concorrente

Paralelismo

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)

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

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

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.

Para haver interação 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

Semáforos

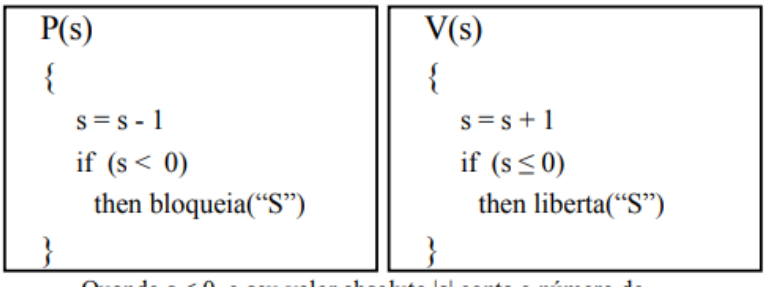
Imagine uma caixa com bolas, rebuçados, pedras...

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

Servem para resolver problemas de sincronização e de exclusão mútua



- Bloquear significa retirar o processo corrente do estado RUN e inseri-lo na fila "S"
- "S" contém os processos BLOQUEADOS no semáforo S
- Bloquear significa retirar o processo corrente do estado RUN e inseri-lo na fila "S" • "S" contém os processos BLOQUEADOS no semáforo S

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	CLIENTE
'* aguarda por vaga no balcão */ '* e só depois vai... */)ousar_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 no balcão */

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)

Exclusão mútua

Os trechos de código de cada tarefa que acede a dados compartilhados são denominados seções críticas (ou regiões críticas).

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 */

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