INFRAESTRUTURA DE SOFTWARE (by @lvrma)

Slides de Aula - IF677 2017.1

Prof. Carlos André Guimarães Ferraz

Introdução

Um *sistema operacional* é um modelo mais simples de abstração de um computador para facilitar o gerenciamento dos componentes, já *sistemas distribuídos* são máquinas distintas que se comunicam.

Existem dois modos de acesso em um OS como o UNIX: *User mode* (que não pode executar instruções que afetam o controle da máquina ou fazer I/O, tendo que executar *Syscalls* para tal) e *Kernel mode* (que pode fazer tudo que a máquina for capaz de executar).

Programas em C são geralmente pré-processados, no UNIX ganhando a extensão "*.i*". São então traduzidos pelo compilador para Assembly ganhando a extensão "*.s*". Assim, programas em C, Java, Python, etc acabam na mesma linguagem final. O *assembler* então transforma o programa em binário, criando o programa-objeto "*.o*". Endereços de acesso e posição do programa na memória permanecem indefinidos. O *linker* então une os *.o* necessários para gerar o executável.

Multiprogramação: um sistema que roda vários processos em *pseudo-paralelismo*.

Sistema Monolítico: OS é um único processo *raíz* com *N children*.

Sistema por Camadas: Organiza camadas como Hardware, Kernel, Libraries, etc.

Sistema Cliente-Servidor: OS distribuído com compartilhamento de recursos entre máquinas.

Sistema de Virtualização: Virtual Machine Monitor administra VMs, utilizado bastante em *cloud* (com VM pools sob demanda). Este VM Monitor pode ser um *Bare-Metal Hypervisor*, rodando em um *hypervisor* diretamente ligado ao hardware, ou um *Hosted Hypervisor*, que roda por cima de algum OS (e.g.: VirtualBox).

Processos: Interrupção e Syscall

Um *processo* é um programa sendo executado, quase para um verbo como o programa está para o substantivo. Informações importantes para seu gerenciamento: registradores em uso (CPU), posições em uso (Memória), estado das requisições (I/O), estado do processo (Ready, Blocked, Running), entre outras menos comuns.

Especificamente, os seguintes campos são associados: *lifetime* (tempo de vida do processo em execução), *PID* (ID do processo), *UID* (ID do usuário que o criou), *parent process*, *parent PID* (ID do seu processo *parent*), *environment* (lista de variáveis associada), *current working directory* (diretório associado).

Um processo *ready* só pode mudar seu estado para *running*. Quando ele estiver *running*, ele pode ir para *blocked* para aguardar alguma input, ou voltar para *ready* quando o *scheduler* julgar que ele já teve tempo de CPU suficiente. Um processo *blocked* só pode mudar seu estado de volta para *ready* quando alguma input ou informação que lhe permita continuar sua lógica estiver disponível.

Principais eventos que levam a criação de processos: Início do sistema, execução de Syscall para criar processos por um processo já existente, solicitação do usuário, e início de um *batch job*.

Principais eventos que levam ao término de processos: Saída normal (voluntário), saída por erro (voluntário), erro fatal (involuntário), e *kill* por outro processo (involuntário).

Na hierarquia de processos, um *parent* cria uma *child*, que pode ter seus próprios *child processes*, criando uma hierarquia, como o processo *init* do UNIX que é a raíz de tudo. No UNIX essa árvore se chama um *group*. No Windows não existe conceito de *parent* e *child*, apenas *handles* transferíveis para se referir a um processo existente.

Multiprogramação é um conceito chave que trata vários programas rodando independentemente como se fossem paralelos, mas na verdade eles estão sendo *escalonados* e só um está ativo a cada momento. Qual dos processos *ready* será executado a cada dado momento quando cada *running* for interrompido fica a cargo do *scheduler*, ou *escalonador*, que toma estas decisões baseado no seu *algoritmo de escalonamento*.

Interrupções são o elo entre hardware e software. Quando são provenientes de algo externo à CPU, como um teclado, elas são interrupções de hardware, ou *interrupções assíncronas*, e ocorrem independentemente do que a CPU está executando. Quando estas acontecem, a CPU salva o contexto atual (incluindo o *PC* ou *program counter,* entre outros registradores de *status*) e chama um programa chamado *tratador de interrupção*. Ao acabar de rodá-lo, ela resume suas operações. É responsabilidade do *tratador* salvar os registradores de *dados* do programa que ele interrompeu, por isso ele tem uma *stack* separada própria.

Além disso, o OS atribui quotas de tempo de execução, chamadas de *quantum*, a cada um dos processos do sistema com *multiprogramação*. O sistema faz então *interrupções de relógio*, e a cada uma o *tratador* verifica se o *quantum* do processo em execução já acabou. Caso sim, o processo volta a ser *ready* e o *scheduler* decide o próximo a executar.

Tirando as interrupções de hardware, ainda existem as de software, ou *interrupções síncronas*, também conhecidas como *traps*. Estas podem até ser chamadas pelo hardware em caso de *overflow* ou acesso de memória não permitido, por exemplo, que vá interromper a lógica do programa e o impedir de prosseguir. Mas geralmente *traps* ocorrem por ordem do próprio programa, como por exemplo em *Syscalls* que são passadas de parâmetro para o *tratador*, que vai encaminhá-las para o local correto.

Os estados dos processos em um OS multitarefa são salvos numa tabela chamada *Process Control Block*, com uma entrada por processo em execução. Estas entradas são atualizadas e recarregadas durante e após interrupções, respectivamente.

Exemplos de *Syscall* no UNIX incluem a *fork()*, que cria uma cópia do processo *parent* (sem compartilhar memória de escrita jamais, de leitura talvez) e a chamada *execve()*, que troca o espaço de endereçamento do *child*, o diferenciando do *parent*.

Processos: Threads e Concorrência

Uma *thread* é a menor sequência de código de um programa que pode ser processada pelo *scheduler* independentemente. Threads de vários programas ou de um único programa podem rodar paralelamente, uma vez que são independentes. Dependendo do caso, quebrar um programa em várias threads pode ser bastante proveitoso.

Geralmente, as *threads* possuem contadores de programa, registradores, *stack* própria, e estado privados. Entretanto, podem compartilhar coisas como *environment*, espaço de endereçamento, tratadores de sinais de interrupção, e processos filhos.

Existem *user threads*, *kernel threads*, e *hybrid threads*. Os nomes são autoexplicativos. No caso das *user threads*, o OS só gerencia a tabela de processos, mas nas que têm acesso *kernel level* ele também administra a tabela de threads.

Uma vez que programas que requerem alto poder computacional e paralelismo estão surgindo num ritmo muito maior do que a criação de CPUs mais rápidas consegue acompanhar, concorrência de threads (10~100 vezes mais leves que processos) e *pseudo-paralelismo* aparecem como ótimas soluções.

Contudo, com elas também surgem problemas como: não determinismo na execução, co-dependência das execuções em paralelo (e.g.: programa 1 em certo momento precisa do resultado de programa 2, mas ambos estão rodando ao mesmo tempo), *starvation*, que é quando um processo é feito esperar muito tempo por recursos alocados para processos de maior prioridade, e *deadlock,* que ocorre quando dois processos bloqueiam um ao outro, criando uma espera infinita.

Outros exemplos de ganho de desempenho nas threads além do paralelismo direto, é a facilidade de multiplicá-las (por serem fáceis de criar) e a distribuição delas para administrar os recursos requeridos pelo programa (e.g.: sobrepor atividades de CPU e I/O, assumindo que nem todas as threads são *CPU-Bound*, ou seja, gastam a maior parte do seu tempo sendo computadas na CPU). Geralmente, processos *IO-Bound* (análogo a CPU bound) devem ter prioridade sobre processo *CPU-Bound*.

Uma das soluções para os problemas listado acima é a *exclusão mútua*, isto é, fazer com que os processos não se "atropelem" e atrapalhem o funcionamento um do outro. Para isto se cria *regiões críticas*, onde só um processo pode acessar por vez (bloqueando os outros processos somente enquanto estiver dentro dela) independente de velocidade ou número de CPU. Ressaltando que a espera deve ser justa.

Para controlar este acesso a regiões críticas, é comum o uso de *semáforos* (mais uma vez, nome autoexplicativo). Entretanto, semáforos podem permitir *N* processos numa mesma região, então para se ter exclusão mútua deve se explicitar que *N = 1* na sua inicialização, criando um *semáforo binário*, similar a um *mutex* (abreviação de *mutual exclusion*, programa que permite o compartilhamento de recursos para múltiplos programas, um por vez). O funcionamento de semáforos geralmente segue a linha *wait* e *signal*, pausando e ativando os processos a ele ligados, respectivamente.

Além disso, existem *monitores*, que são conjuntos de instruções para realizar *exclusão mútua*. Pode se pensar em monitores como semáforos mais complexos, uma vez que um semáforo é somente uma espécie de "cadeado" enquanto monitores implementam rotinas.

Algumas vezes a exclusão mútua não é suficiente, tornando necessário que o programador implemente outras formas de controle como variáveis condicionais próprias para uso com threads.

Escalonamento

É o processo de decisão por parte do OS, mais especificamente do *escalonador* ou *scheduler*, de qual processo dentre os que estão *ready* será executado em determinado momento. Processos *I/O-Bound* passam menos tempo no estado *running* na CPU e mais tempo *blocked* esperando por I/O, e têm prioridade sobre os processos *CPU-Bound*. Ressaltando que o escalonamento *preemptivo* é justamente o escalonamento que permite a interrupção de um processo e sua retirada da CPU antes do fim de sua execução (o conceito de *quantum* já foi definido acima), mas também existe escalonamento *não-preemptivo*.

Geralmente, existem quatro tipos de *scheduling*:

- L*ong-term scheduling*, ou escalonamento de longo prazo, admite programas, cria processos, e aloca memória. O programa responsável por este escalonamento é o *admission scheduler* do OS. A fila ligada a este escalonamento é a *long-term queue*, que recebe *requests* de criação de processos e tem saída ligada à *short-term queue*, não à CPU.
- S*hort-term scheduling*, ou escalonamento de curto prazo, administrado pelo *CPU scheduler* do OS. Este escalonamento cuida dos processos *ready* que serão imediatamente passados para a CPU.
- *I/O scheduling* é um escalonamento implementado em sistemas operacionais que tem fila(s), chamada(s) *I/O queue(s),* para lidar com processos com I/O pendente. Os processos entram na *I/O queue* pela CPU, saem para o I/O desejado, e seguem direto para a *short-term queue*. Isto é apenas uma outra forma de explicar o que foi dito acima sobre o ciclo de estado *ready -> running -> blocked -> ready*. Vale ressaltar que o *tratador de interrupções* é bastante presente em todo o procedimento.
- *Medium-term scheduling* é uma forma de intercambiar processos da fila com processos na memória virtual. Será visto em mais detalhe no tópico de memória a seguir.

Em relação a algoritmos de escalonamento, existem três tipos que serão descritos abaixo, cada um com um foco. Entretanto, todos compartilham o foco em: *justiça* ou *fairness* (distribuir o uso de CPU de forma justa), ter uma política de escalonamento rígida, e

manter todas as partes do sistema equilibradas. Especificamente, algoritmos de escalonamento para *batch systems* ou *sistemas em lote* (que executam vários *jobs* sem intervenção humana), focam fortemente em maximizar *throughput* ou *vazão* (tarefas/hora), minimizar *turnaround time* ou *tempo de retorno* (tempo entre início e término do processo), e manter a CPU sempre ocupada, maximizando seu uso. Enquanto isso, algoritmos para *sistemas interativos* (com muita interação entre seres humanos e a máquina) focam fortemente no tempo de resposta a requisições e na proporcionalidade distribuição de recursos (diferente de *fairness*, esta foca em satisfazer às expectativas dos usuários). Já *sistemas em tempo real* (que requerem um tempo específico para a execução dos processos) focam em cumprimento das *deadlines*, evitar perda de dados, e evitar a degradação da qualidade.

Ao trocar de processos num escalonamento *preemptivo*, existe uma *troca de contexto* para atualizar tabelas, listas, memórias, etc. O tempo levado por esta troca se chama *overhead administrativo* (e.g.: um sistema com 20ms de *quantum* e 5ms de *overhead administrativo* possui 20% do seu *tempo de CPU* somente sendo gasto com *overhead*). Uma solução para casos como o do exemplo é aumentar o *quantum*, tornando a porcentagem de *tempo de CPU* gasta em *overhead administrativo* negligível. Entretanto, aumentar o *quantum* diminui a eficiência da CPU, enquanto diminuí-lo aumenta o *overhead*. A verdadeira dificuldade é encontrar um valor equilibrado, uma vez que este depende do sistema e seus usos.

Os sistemas de tempo real geralmente utilizam algoritmos de escalonamento *EDF* (earliest deadline first), fazendo uma *priorização dinâmica* e *preemptiva* dos processos na fila conforme eles se aproximam de suas *deadlines* individuais.

Já sistemas em lote *(batch systems)* utilizam geralmente ou algoritmos *FIFO* (*não-preemptivo,* primeiro a entrar é o primeiro a sair, *justo* e simples), ou *SJF* (*não-preemptivo,* shortest job first), ou *SRJN* (versão *preemptiva* do SJF. Seu nome significa *shortest remaining job next)*. Tanto SJF quanto SRJN *não são justos*, podendo causar *starvation* caso a prioridade não seja alterada dinamicamente.

No caso de sistemas interativos existem vários e vários algoritmos de escalonamento. Alguns deles sendo:

- *Round-robin* ou chaveamento circular: Utiliza uma fila e quando o programa consome todo o seu *quantum*, caso não esteja terminado, volta para o fim da fila.
- Escalonamento por prioridade: Quanto mais alta a prioridade, primeiro ele executa.
- *Multilevel Queue:* Possui *N* filas de prioridades diferentes, cada uma podendo ter seu próprio algoritmo de escalonamento. Enquanto existir algo em uma fila de prioridade maior as de prioridades menor vão aguardar, podendo causar *starvation.*
- *Multiple Feedback Queue*: Adapta as prioridades (uma fila por nível de prioridade) dinamicamente dependendo do comportamento dos processos (se o comportamento tende para algo mais I/O ou CPU *bound*). Processos novos (desconhecidos) entram na primeira fila (de prioridade mais alta). Acabar o *quantum* faz descer um nível, requisitar I/O faz subir um nível. Estas variâncias de nível podem ser diferentes dependendo da vontade do programador.
- *Fair Share*: Distribui a CPU igualmente entre os *usuários* (não processos). Cada usuário divide sua *share* entre seus processos *igualmente*, sem afetar a *share* de outros usuários.
- Garantido: Conhecendo o processo sendo executado e os próximos na fila, estima e cumpre um prazo de tempo de execução.
- Loteria: Distribui "bilhetes" de acesso à CPU de forma não uniforme e o *scheduler* sorteia um aleatoriamente. Pode ser preemptivo ou não, mas garante que não haverá *starvation*.
- Muitos outros, incluindo o SJF já listado acima.

Memória

No OS existe uma função chamada *gerenciamento de memória*, que aloca e realoca memória seguindo sua política especificada (incluindo para processos novos), gerencia a hierarquia de memória, garante isolamento mútuo de processos, faz *swapping* entre memória principal e disco, mapeia a relação memória virtual/física.

A memória ideal é: grande, rápida, e não volátil. Entretanto, quanto mais rápida a memória mais cara ela é, logo, menos estará disponível e ela será menor. A hierarquia de um computador comum segue: cache -> memória principal (RAM) -> disco.

Com a *multiprogramação*, memórias limitadas começaram a precisar ser realocadas dinamicamente enquanto evitando que acessos incorretos sejam feitos. Isso foi solucionado pelo uso de *bases* (início das partições) e *limites* (tamanhos das partições), controlando o acesso e limitando-o somente a este *range*.

Entretanto, conforme a memória vai sendo alocada e realocada dinamicamente (*swapping*), surge a possibilidade de haverem brechas entre as partições. Também muitas vezes não existem programas que caibam nestes espaços, criando então o que se chama de *fragmentação externa*. O processo de *desfragmentação*, que deixa a memória compacta novamente, é extremamente custoso.

Outra forma de administração de memória é criando partições fixas e alocando um *job* da fila por partição. Isto pode causar o problema de *fragmentação interna*, pois ocorre dentro da partição e se refere ao espaço não utilizado pelo *job* alocado no momento.

Uma estratégia para administrar espaços livres na memória é o *Free Space Bitmap*, que consiste em dividir a memória em setores, e em um *array de bits* definir um bit correspondendo a cada setor. Um bit com valor 1 equivaleria a um setor ocupado por um processo, enquanto um bit com valor 0 equivaleria a um setor livre ou *hole*. Apesar desta implementação mais simples ser bastante ineficiente, existem inúmeras heurísticas, como algumas que incluem o uso de listas encadeadas.

Mas caso o programa precise de mais memória do que disponível, se criam os conceitos de *memória virtual* (memória que aparenta estar na memória primária mas na verdade está na

secundária)*, page frame* (um bloco na *memória principal*, normalmente de 4 KB. Um endereço *físico)*, e *página* ou *page* (um bloco de tamanho fixo da *memória virtual*, normalmente do mesmo tamanho das *page frames*. Um endereço *lógico*).

Para administrar tudo isso, se utiliza uma unidade (normalmente implementada na CPU) chamada *MMU* ou *Memory Management Unit* ou ainda *Unidade de Gerenciamento de Memória*.

É utilizada uma tabela de página por processo, por isso a tabela não precisa saber o ID do processo em questão. Nas tabelas se utiliza os dados: endereço *lógico*, *deslocamento*, endereço *físico*, bit de *presença* (indicando se a página está na memória física ou não).

Para calcular o *endereço físico* os seguintes passos são seguidos:

1. Dividindo (divisão inteira) o endereço *lógico* pelo tamanho da *frame*, obtemos o índice da página na tabela (e.g.: endereço 46100, frame size 4096 bits 46100/4096 = 11. Ou seja, existem 11 blocos de 4096 bits (ou 11 páginas) até aquela, fazendo aquela página ser a página de índice 11).
2. O resto da divisão inteira feita acima nos dará a posição exata do endereço *lógico* que desejamos dentro do *bloco físico* (no caso do exemplo, 46100 mod 4096 = 1044, logo, o que desejamos se encontra 1044 bits após a base do bloco físico).
3. Com o índice da página em mãos, podemos checar na *page table* qual o índice *físico* daquela página. Após isso, multiplicamos o índice *físico* pelo *frame size* para encontrar o endereço físico da *page frame* e somamos com o deslocamento encontrado acima para chegar ao resultado exato (digamos que na *page table* a página 11 estivesse ligada à *page frame* de índice 7, calcularíamos o endereço físico da seguinte forma: (7 * 4096) + 1044 = 29716).

De maneira similar, para calcular o tamanho da tabela de páginas basta multiplicar o tamanho da entrada pela quantidade máxima de entradas. Para calcular a quantidade máxima de entradas, pegamos a quantidade de bits do nosso endereçamento lógico (e.g.: um endereço lógico de 32 bits pode representar 2^{32} endereços diferentes) e dividimos pelo tamanho da página (4 KB = 2^{12}, dividindo $2^{32}/2^{12}$, descobrimos que podemos ter até 2^{20} páginas representadas com endereço único. Com isso sabemos que 20 bits dos 32 bits do endereçamento serão necessários para indicar a página, e os 12 bits que sobraram

indicarão o deslocamento dentro dela). Com isso em mãos, basta fazer os cálculos: quantidade de páginas * tamanho do endereço (2^{20} bits * 32 bits = 4MB). Ressaltando que existe uma tabela *por processo*.

Mais um problema surge, entretanto, caso a *memória virtual* seja muito grande, pois a tabela de páginas será igualmente grande gerando atrasos inviáveis uma vez que a velocidade de acesso à *memória física* é crucial. Para solucionar isso utilizamos o *TLB* ou *Translation Lookaside Buffer* ou ainda *Memória Associativa*. Eles funcionam quase como um *cache* para tabelas de página, armazenando os endereços mais recentemente acessados e evitando os cálculos.

Quando uma página não está disponível no *cache*, ocorre um *cache miss* e a página é procurada na TLB. Caso ela não seja encontrada, ou seja, ocorre um *TLB miss*, só então esta página é procurada na *page table*. Caso, após os cálculos feitos na *page table*, o endereço *físico* resultante não seja encontrado na memória principal, ocorre uma *page fault*.

Mas em alguns casos, uma TLB ainda não é suficiente, uma vez que armazenar muitas *page tables* grandes na memória ainda é inviável. Com isso se criam *tabelas de página multinível*, onde o nível acima aponta para tabelas do nível mais baixo até chegar ao último nível, o endereço físico. Com isso, se evita varredura de tabelas, apenas percorrendo um caminho único até o endereço de destino.

Outra alternativa para reduzir a quantidade de memória física utilizada é utilizar uma *tabela de páginas invertida*, isso é, uma tabela de páginas única para todos os processos, *global* para todo o sistema. Ela contém informações adicionais como o *PID*.

Para calcular o tamanho de uma tabela de páginas *invertida* o procedimento é igual, mas deve-se considerar também a quantidade de bits ocupada pelo *PID* (no caso do exemplo anterior, com um PID de, digamos, 8 bits ou 2^8 processos possíveis simultaneamente, só sobrariam 4 bits para o deslocamento ou *access information*. A partir daí, podemos dividir o tamanho total da memória, por exemplo, 2 GB ou 2^{31} pelo *page size*, resultando em 2^{19} páginas possíveis. Multiplicando esta quantidade por 32 bits temos 2 MB de *table size*).

Uma tabela *invertida* é muito mais leve em questão de memória ocupada, no caso do exemplo anterior comportando 256 processos no espaço de meia tabela normal. Isto se deve ao fato de nas *page tables* normais, existirem muitos mapeamentos, tanto físicos

quanto lógicos, repetidos para cada processo. Ainda assim, a tabela invertida não é uma boa saída para compartilhamento de memória, pois a tradução de endereços fica muito mais lenta, requerendo uma varredura na tabela do sistema inteiro e criando uma dependência muito forte nos *hits* da TLB. Também se usa funções de *hash* para indexar as entradas em uma tabela invertida.

Para medir o desempenho da paginação implementada, se utiliza duas variáveis: *page fault rate* (porcentagem de *miss*, entre 0 e 1, sendo 0 nenhum *miss* e 1 sempre *miss*) e *page fault overhead* (tempo de remover uma página ou *swap out* + tempo de carregar outra página no seu lugar ou *swap in* + tempo de retomar o processo ou *restart overhead*). Com essas duas variáveis, podemos calcular o *Effective Access Time* ou *EAT.*
EAT = ((1 - *page fault rate*) * *memory access time*) + (*page fault rate* * *page fault overhead*)

Este cálculo torna bem visível os dois grandes problemas da *paginação*: tabelas muito grandes e paginação demorada. No caso de *page fault* como decorrência de um *miss*, a decisão de qual página remover e onde alocar o espaço também pode não ser fácil, uma vez que páginas modificadas não podem ser sobrepostas e páginas muito utilizadas não devem ser removidas. Por isso temos *algoritmos de substituição de páginas* como:

- *Algoritmo Ótimo*: Consiste em um algoritmo que só substitui a página que vai ser utilizada o mais distante no futuro. Entretanto, como não podemos prever o futuro, este algoritmo é somente utilizado para comparações de eficiência com outros algoritmos reais em testes.
- *Second Chance*: Uma melhora do algoritmo *FIFO*. Toda página possui um *referenced bit* que inicia em 0 e muda para 1 quando página é referenciada. Quando este bit está como 1 a página é dada uma "segunda chance" ao ser encontrada na cabeça da fila e o bit é zerado, a próxima página da fila é verificada e por aí vai. Se seu *referenced bit* estiver zerado, ela será *swapped out*.
- *Not Recently Used* ou *NRU*: Este algoritmo favorece manter páginas modificadas e recentemente acessadas acima de tudo. Cada página possui um *referenced bit* e um *modified bit*. Quando a página é referenciada, seu *referenced bit* é ativado, análogo para o *modified bit* e modificações. Existe ainda um *timer* fixo que de tempos em tempos reseta todos os *referenced bits*, para evitar que páginas visitadas muito tempo atrás ainda estejam priorizadas. Na hora de remover uma página, o OS as

divide em 4 classes: [0] ambos bits zerados (menor prioridade), [1] não referenciada recentemente porém modificada, [2] referenciada recentemente porém não modificada, [3] referenciada recentemente e modificada (maior prioridade). A página é removida aleatoriamente da menor classe de prioridade não vazia.

- *Least Recently Used* ou *LRU*: Retira da memória a página usada há mais tempo, que ele ordena em uma lista encadeada por uso recente e atualiza a cada referência. Outra alternativa é manter um contador em cada entrada da tabela para remover mais facilmente, zerando os contadores periodicamente. Enquanto *NRU* olha para um período de tempo mais longo, *LRU* foca nos *ciclos de clock* mais recentes.
- *Working Set* ou *WS*: Este algoritmo tenta manter uma janela das próximas instruções que acessam a memória, movendo esta janela sempre que a memória é acessada (custo elevado). Cada entrada tem um *referenced bit* (atualizado todo *clock)* e uma *idade*, definida pelo *tempo virtual atual - instante da última referência* a aquela página. Caso o *referenced bit* esteja ativado, o *instante da última referência* daquela entrada é definido para ser igual ao *tempo virtual atual*. Mas caso a entrada tenha o seu bit de referência zerado, a sua idade é checada em relação a um parâmetro *T* (definido pelo tamanho da janela medido em tempo virtual, para definir se uma página pertence ou não à janela). Se a idade for maior do que este parâmetro, ela é *swapped out*.
- *Clock* ou Relógio: Basicamente um *Second Chance* que ao invés de adicionar e remover páginas do início para o fim, mantém uma lista circular e guarda a posição da "mão do relógio", sendo mais eficiente. O sistema do *referenced bit* funciona da mesma forma que o do *Second Chance*.
- *WSClock*: Um dos algoritmos mais importantes na prática hoje em dia. Combina o algoritmo *WS* com o *Clock*, basicamente utilizando a filtragem do *WS* mas com uma lista encadeada circular e o conceito de "mão de relógio" ao invés de uma janela sempre seguindo em frente. O tratamento de ambos para o *referenced bit* se encaixa perfeitamente quando combinado.

Na maioria dos testes comparando com o *algoritmo ótimo*, se viu que quanto mais *page frames* utilizadas, menos *misses*. Entretanto, isto somente não acontece com o *FIFO*, que fica pior. Esta anomalia é conhecida como a *Anomalia de Bélády*.

Algumas vezes, quando muitos processos requerem muita memória mas nenhum pode abrir mão, ocorre o *thrashing*, que é a paginação excessiva. Uma solução para controlar esta carga é mover alguns processos para a memória *swap* de disco, comprometendo seu desempenho mas liberando memória, e reconsiderar o *grau de multiprogramação*.

O tamanho da página é outra fonte de problemas, pois uma página pequena causa menos fragmentação e armazena menos "programa parado", mas requer uma tabela de páginas bem maior já que os programas vão requerer mais páginas.

Uma alternativa para o problema acima é separar o que antes era espaço de endereçamento *único* em páginas separadas somente para *instruções* e somente para *dados*. Com isso, a processos que compartilhem o mesmo código (como instâncias do mesmo programa) podem compartilhar a tabela de *instruções* enquanto mantendo *dados* separados.

É responsabilidade do OS na *paginação*: determinar o tamanho do programa, criar a *page table*, iniciar o *MMU* para cada novo processo, determinar o *endereço virtual* que causou uma *page fault*, descartar páginas antigas, fazer o *swapping*, liberar *page tables*, liberar páginas, e liberar espaço de disco.

Em alguns casos, um programa aguardando I/O pode ser marcado como inativo por engano. Por causa disso, uma alternativa é fixar páginas envolvidas com I/O na memória.

No final das contas, *paginação* tem uma falha grave: conforme as tabelas crescem durante a execução, elas podem colidir, uma vez que o espaço da memória é unidimensional. Por causa disso existe a alternativa da *segmentação*, que permite que cada tabela cresça e diminua de maneira independente do resto. Basicamente, enquanto *paginação* foca em oferecer um espaço de endereçamento enorme sem a necessidade de comprar mais memória de alto custo, *segmentação* existe para garantir endereçamentos logicamente independentes com compartilhamento e confiabilidade.

Entretanto, o sistema precisa sempre mover segmentos entre a memória secundária e primária, alocando espaço suficiente para o segmento e gerando muita fragmentação. Além do que alocar muitos segmentos muito grandes se torna inviável. Com isso surgiu a

ideia de *segmentação com paginação*, criando uma memória virtual de duas dimensões, primeiro implementada no OS Multics.

No Multics, os programadores decidiram quebrar os segmentos em tabelas, por isso cada programa possui uma *segment table*, com um *segment descriptor* por segmento deste programa. O *descriptor* leva para a tabela individual de cada segmento. A tabela de cada segmento, como dito acima, armazena as páginas em que o segmento foi repartido. Se alguma destas páginas estiver na memória física, a *page table* deste segmento também estará. Caso esta tabela não possa ser encontrada, ocorre uma *segmentation fault*. Caso a tabela seja encontrada com sucesso, mas a página desejada não estiver na memória principal, ocorre uma *page fault*. Caso contrário, o *offset* é somado para se chegar à instrução desejada dentro do segmento.

Sistema de Arquivos

Para se armazenar informações a longo prazo, deve-se ter grande capacidade de armazenamento, a informação deve sobreviver o término do processo que a utiliza, e múltiplos processos devem ter acesso simultâneo à informação.

Arquivos podem seguir várias nomenclaturas diferentes (como extensões), ser sequências de bytes, registros, ou árvores, ser apenas focados em armazenar dados ou ser executáveis, podem ter atributos e *flags*, e podem ser operados de diversas formas pelo OS e usuário (*create, delete, open, close, read, write*) dependendo das variáveis descritas anteriormente.

Surgem então as perguntas: como encontrar os arquivos eficientemente? Como prevenir a leitura dos dados de um usuário pelo outro? Como saber onde tem espaço livre?

Uma das formas de solucionar a primeira pergunta é com o uso de diretórios, seguindo uma hierarquia desde o diretório raiz. Estes podem ser operados de maneira análoga aos arquivos (e.g.: *create, delete, open, close*).

Dentre as possíveis formas de alocação de arquivos, existe a *alocação contígua*, que é uma forma elegante de descrever alocação contínua, ou seja, uma alocação de espaços consecutivos. Ela pode ser feita de três maneiras principais: *first-fit* (mais rápida. O primeiro segmento livre com tamanho suficiente para alocar o arquivo encontrado na busca sequencial é o escolhido), *best-fit* (menos desperdício. O menor segmento livre disponível com tamanho suficiente para armazenar o arquivo é escolhido, entretanto é necessário ou uma varredura total ou uma ordenação por tamanho dos segmentos presentes no disco), e finalmente *worst-fit* (mais expansível. O maior segmento é alocado para o arquivo, também sendo necessária uma varredura completa caso a lista não esteja ordenada por tamanho).

Independente da estratégia, haverá *fragmentação*, o que pode ser muito grave caso o espaço necessário esteja livre mas sem segmentos contínuos onde o arquivo possa ser armazenado. Logo, *desfragmentações* são algo necessário periodicamente, entretanto a reorganização dos arquivos demora muito e é uma solução somente temporária.

Assim como em memória, em sistemas de arquivos também existe *fragmentação externa* (espaços vazios entre blocos de arquivos do disco) e *fragmentação interna* (espaço dentro de um bloco alocado mais do que seria necessário, blocos não podem ser alocados parcialmente). A *fragmentação externa* cresce conforme arquivos são criados e removidos, criando mais espaços vazios menores e menores, mais difíceis de preencher, aumentando o desperdício. Para solucionar por *desfragmentação*, existem vários algoritmos de reagrupamento de fragmentos em espaços maiores, focados em minimizar a movimentação para aumentar a eficiência.

Além da *alocação contígua,* existe também a *alocação por lista encadeada*, que armazena um arquivo como uma lista encadeada de blocos de disco, um bloco por nó da lista apontando para o próximo bloco, podendo ser indexados com o auxílio de uma tabela de alocação na memória principal para orientar qual bloco vem após qual em toda a extensão do arquivo, melhorando performance.

Por fim, também temos a *alocação indexada*, onde cada arquivo recebe um bloco de indexação (como a estrutura de dados do UNIX chamada *inode*, que armazena os atributos e índices de blocos de um arquivo). Tem tempos de acesso melhores do que os da *alocação por lista encadeada*, mas caso o arquivo seja pequeno ocorre desperdício do bloco de índices quase inteiro.

Assim como na memória, a escolha do tamanho dos blocos no disco também é importante. Quanto menores, menos *fragmentação interna*, mas maior custo para gerenciar os múltiplos blocos por arquivo. O oposto vale para quanto maiores sejam.

Blocos livres podem ser monitorados de diversas formas, como listas encadeadas ou *bitmaps*, de maneira similar à memória. Além disso, usuários podem ter *cotas* de uso de disco.

Alguns detalhes de implementação de diretórios valem ser mencionados. Um diretório pode ter atributos fixos ou *inodes*, e pode armazenar nomes grandes de arquivos linearmente ou por referência a uma *heap* temporária. Diretórios também podem compartilhar arquivos mesmo em diferentes níveis de sua hierarquia.

Entrada e Saída

Existem diversos tipos de dispositivos de I/O, desde teclados e mouses até portas Ethernet, USB, e discos rígidos.

A forma como a CPU acessa um dispositivo de I/O depende do tipo de dispositivo. Para dispositivos de I/O *isolados*, o acesso é feito através de instruções especiais especificando a leitura/escrita na porta de I/O. Já para dispositivos de I/O *mapeados na memória*, a leitura é feita através de instruções de leitura e escrita na memória (podem ser de *barramento único*, com todos os endereços de memória e I/O passando pelo mesmo barramento, ou de *barramento duplo*, onde a CPU lê e escreve na memória através de um barramento de alta velocidade exclusivo, e o I/O tem uma porta na memória para acessá-la). Por fim, dispositivos também podem ser *híbridos*, compartilhando de ambas características.

Diferente de quando a CPU lê da memória, onde a leitura e escrita é direta, para ela ler e escrever em dispositivos de I/O existem *interfaces* (circuitos intermediários).

Uma alternativa para o I/O ler e escrever na memória diretamente, sem ocupar a CPU, é através do *Direct Memory Access Controller* ou *Controlador de DMA*. Sem o DMA, a CPU ficaria sempre ocupada aguardando a operação, mas com ele, ela apenas precisa iniciar a operação e depois receber um *interrupt* do DMA quando a operação for concluída. É extremamente útil quando a CPU precisa realizar outros trabalhos enquanto aguarda os dispositivos de I/O.

A CPU pode executar I/O de três formas: *I/O programado* (lê constantemente do *controlador de I/O* verificando se já acabou, fazendo um *busy-waiting* e aguardando até o fim da operação sem fazer mais nada), *I/O por interrupção* (interrompida pelo módulo de I/O quando acaba a transferência. Continua ativa trabalhando em outras coisas mas cada palavra transferida pelo periférico passa pela CPU), *I/O por DMA* (o *controlador de I/O* solicita ao *controlador de DMA* o início da transferência, a CPU então opera sem interferência e é interrompida no final sendo informada do resultado.

A gerência de I/O é importante por ser necessário ter uma eficiência uniforme diante da diversidade de dispositivos possível, sem muitos detalhes de hardware ou baixo nível, focando mais em padronização, abstração, e simplificação como: *read, write, open,* e *close*.

Para obter este resultado, o software de I/O é organizado em camadas. Diretamente acima do hardware está o *tratador de interrupção* (que indica o final das transferências e aciona o *driver*), acima dele está o *driver do dispositivo* (que recebe as *requisições* e aciona o *controlador*, que é parte do hardware), acima dele está o *I/O independente de dispositivo do OS* (que armazena os nomes, protege os dados, e faz a bufferização), e por último no topo está a camada de *I/O em nível de usuário* (que faz *Syscalls* de I/O).

O *tratador de interrupção* precisa fazer interrupções da forma mais transparente possível, para isso, ao receber uma solicitação, ele: bloqueia o *driver* que iniciou a operação de I/O, cumpre sua rotina de interrupção, notifica o fim do I/O, e finalmente desbloqueia o *driver*.

Os *drivers de dispositivos* se comunicam com os controladores de cada dispositivo por meio de barramento, sendo uma ponte entre hardware e software.

Funções do OS para *I/O independente do dispositivo* operam em um nível mais alto, formando *interfaces uniformes para drivers*, armazenando dados em *buffers*, gerando relatórios de erros, alocando e liberando dispositivos dedicados, e fornecendo o tamanho de bloco *independente* do dispositivo.

As funções de I/O de *espaço de usuário*, somente executam as *Syscalls* que são enviadas do topo das camadas (nível mais alto de abstração), até o hardware e de volta para o usuário com o resultado da operação.

No caso de discos rígidos (que apesar de serem considerados memória secundária, são dispositivos de I/O totalmente separados da memória principal) existem *algoritmos de escalonamento de braço de disco*, que são algoritmos para otimizar o tempo de leitura ou escrita de um bloco no disco com base em 3 fatores: *tempo de posicionamento do braço* (maior de todos), *atraso de rotação do disco*, e *tempo de transferência dos dados*. Erros são checados pelo controlador do dispositivo. Um destes algoritmos é o *Shortest Seek First (SSF)* ou *Posicionamento Mais Curto Primeiro*, que simplesmente direciona o braço para a próxima posição (entre as requisições pendentes) diretamente mais próxima (algoritmo *guloso*), sem considerar o caminho como um todo. Outro algoritmo é o *Elevador* ou *SCAN*, que se baseia no elevador de um prédio, movendo-se na direção que já estava movendo até acabar as requisições naquela direção, e só então movendo-se de volta na direção oposta.

Sistemas Distribuídos

Enquanto um sistema de software simples é apenas uma *sequência* de instruções executadas por um processador, um *sistema de software distribuído* é um conjunto de instruções *concorrentes/paralelas* executadas por *um ou mais* processadores. Esta técnica de dividir para conquistar melhora a escalabilidade e desempenho (através de paralelismo) do sistema como um todo, como um *next step* na linha evolucionária que começou com monoprogramação -> multiprogramação. A principal motivação para a existência desses sistemas é a alta demanda de recursos, e a possibilidade de compartilhamento deles, por exemplo em *cloud computing (PaaS, SaaS, etc)*, computação móvel, multimídia (*Netflix* utiliza sistemas de recomendação, database, busca, autenticação, e streaming distintos que funcionam como um só), entre muitos outros exemplos. Os recursos e a "carga" agora podem ser divididos entre *client* e *server*, quando antigamente o servidor fazia todo o trabalho sozinho, ficando sobrecarregado. Além disso, um *server* pode trabalhar como um mas na verdade consistir de várias máquinas.

O maior problema com esta lógica é que quanto mais descentralizado (e sem um *relógio global*, requerendo sincronização dependendo do uso), maior a probabilidade de falhas individuais (por máquina, também chamadas de *falhas parciais*), perda de mensagens, etc. Algumas das técnicas para solucionar este problema incluem a replicação e repetição das tentativas de comunicação.

Com isso em mente, um sistema distribuído deve ser eficiente (também usado para o mal, como na realização de ataques *DDoS*), resistente a falhas, e consistente com os dados em todas as suas máquinas. Mas para se ter tudo isso se precisa de *transparência* (uma característica de sistemas distribuídos que consiste em esconder seus atributos internos do *usuário final*, criando uma experiência "limpa"). Pode ser de dois tipos: *transparência de localização* (não deixa a localização dos recursos explícita), e *transparência de acesso* (tendo operações iguais para acesso local e remoto, escondendo a migração e realocação de recursos, falhas, concorrência e replicação de recursos). Quando se une essas duas transparências de *localização* e *acesso* temos o que chamamos de *transparência de rede*.

Por último, temos que falar de *middleware*, que faz a integração dos sistemas heterogêneos das máquinas e provê portabilidade para as *aplicações distribuídas*, criando a transparência.

Por definição, é um conjunto reusável e expansível de serviços e funções (*serviços de middleware*) que faz *aplicações distribuídas* funcionarem bem em um *ambiente de rede* através de APIs chamadas *interfaces padrão*.

Existem vários tipos de middleware, como o *MOM* ou *Message Oriented Middleware*, o *TPMON* ou *Transaction Processing Monitors* (usado fortemente em databases, com as propriedades de atomicidade, consistência, isolamento, e durabilidade, conhecidas como *ACID*), *ORB* ou *Object Oriented Middleware*, e entre muitos outros, *RPC* ou *Remote Procedure Calls*, que veremos em mais detalhe.

O *Remote Procedure Call* faz uma *abstração de comunicação*. Uma vez que comunicação é o coração de um sistema distribuído, o *RPC* permite que clientes façam chamadas remotas a servidores, por exemplo.

Um pacote RPC utiliza *ACKs* e *timeout/retry*, entregando no final um serviço que parece um procedimento local. Como o ideal é que um sistema distribuído possa ser programado como se fosse centralizado, o objetivo do RPC é permitir uma chamada remota executar como se fosse local, ocultando os detalhes de entrada e saída.

Para essa integração, por exemplo em casos de cliente e servidor escritos em uma linguagem de programação comum, o RPC faz uso de *stubs*. Eles são pedaços de código que convertem parâmetros entre cliente e servidor (baixados como *libs* em ambos os lados), criando o efeito de *transparência de acesso*. *Stubs* também podem tratar algumas exceções localmente. O processo do *client stub* converter os parâmetros para algo legível para as funções requisitadas ao server se chama *marshalling*. O processo reverso se chama *unmarshalling* e deve ser feito com os resultados recebidos. O *server stub* ou *server skeleton*, é responsável pelo *marshalling* dos resultados enviados e o *unmarshalling* dos parâmetros recebidos, criando o ciclo operacional.

Além de tudo isso, o *RPC* utiliza um *binder* para resolver os nomes que ele conecta. Isto permite *ligação dinâmica* entre máquinas e *transparência de localização*. Entretanto, no final das contas o que realmente nos vai levar à máquina não é um *nome*, e sim um *endereço*. Esta tupla de dois valores (nome e endereço) se chama *serviço*, e é mapeado pelo *resolvedor de nomes*, que conecta os nomes aos seus endereços e se adapta a mudanças, criando a ilusão de *transparência*.