Estrutura

Divide-se em duas compontentes essenciais: User mode e Kernel mode.

Kernel mode

É responsável por estabelecer uma ligação mais simples e abstrata entre o User e o hardware, detendo as funções básicas e essenciais do SO e não podendo ser modificado. Este oferece alguns serviços como: I/O, execução de programas, agendamento de processos, alocação de recursos, etc.

User mode

Corresponde à "camada" mais superficial e tem apenas acesso a um conjunto limitado de instruções(system calls) disponibilizado pelo kernel. UIs e programas utilizados habitualmente fazem parte do User mode.

As system calls são a forma de comunicação entre kernel mode e user mode. Além de permitir um maior conforto, sendo o próprio SO responsável por gerir os recursos, mantém o sistema e o hardware seguros a programas intrusivos (uma vez que apenas o sistema operativo tem comunicação direta com os componentes físicos, é impossível um software malicioso alterar, por exemplo, o comportamento do relógio de CPU).

Processos

Um processo contém a informação sobre um programa em execução (registos, variáveis, etc) para que este possa ser parado e recuperado sempre que necessário, essencial para o escalonamento de processos.

Criação

- Inicialização do sistema

Quando o sistema operativo é inicializado, são criados alguns processos, que são, maioritariamente, daemons. Estes processos são, geralmente, utilizados para gerir pedidos(de rede, por exemplo) ou configurar hardware, e são executados em background sem estarem associados específicamente a nenhum utilizador.

- System calls

Novos processos podem também ser criados por outros processos através de system calls. Isto torna-se principalmente útil quando existe trabalho que pode ser facilmente executado por processos independentes, tal como processamento de dados.

- Pedido do utilizador

O utilizador pode também criar novos processos utilizando uma linha de comandos ou, com uma GUI, com duplo clique, normalmente.

- Batch

Nos sistemas clássicos, batch consistia num computador central, podendo ter ou não ligados a ele vários terminais, onde os processos ou jobs eram escalonados por uma simples queue.

No que toca à criação de processos, um é criado por cada tarefa.

Para criar um novo processo em sistemas UNIX, existe apenas uma system call: fork. Quando invocada, um processo idêntico(incluindo memória e descritores) ao que invocou a função é criado, no entanto, cada um dos processos tem o seu próprio espaço de endereçamento, não partilhando assim as modificações efetuadas após o fork.

Terminação

- Saída normal e de erro (voluntárias)

Um processo termina com a invocação da system call exit. Esta system call recebe um inteiro como argumento e, por convenção, quando um processo termina normalmente, deve ser passado 0 nesse parâmetro. De outro modo, na ocorrência algum erro durante a execução, deve ser devolvido um status diferente no exit que poderá ser recebido pelo processo pai.

- Erro fatal e “kill” (involuntárias)

Estas terminações envolvem o uso de sinais. No primeiro caso, os sinais são enviados pelo sistema ao processo, normalmente devido a bugs no programa(segfaults - SIGSEGV -, operações de aritmética inválidas - SIGFPE -, etc). No segundo caso, um sinal pode ser enviado a um processo invocando explicitamente a system call kill. Kill recebe dois parâmetros, o pid do processo a matar e o sinal a enviar.

Apesar de os sinais serem essencialmente utilizados para matar processos, a maioria deles podem ser intercetados e utilizados também como meio de comunicação entre processos.

Hierarquia

Em sistemas UNIX existe uma hierarquia de processos, isto é, todos os processos pertencem a uma mesma árvore. Cada processo forma um grupo com os processos filho criados e estes continuam sempre associados. Quando um processo pai morre antes dos filhos, estes são herdados pelo processo com pid 1, init.

Estados

- Running

Quando um processo se encontra neste estado, este está a utilizar o CPU.

- Ready

O processo está à espera que outros processos acabem de correr para fazer uso do CPU.

- Blocked

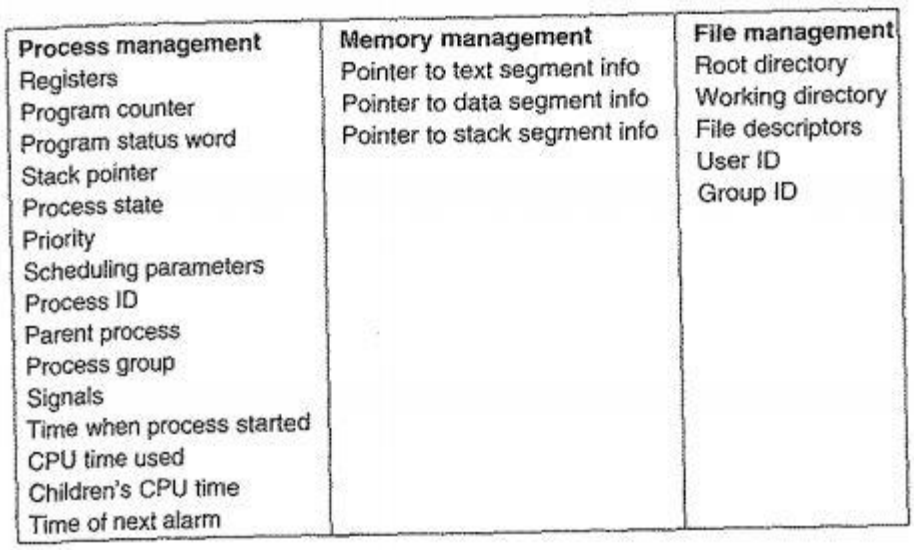
O processo encontra-se à espera que algo aconteça para poder avançar. Um exemplo pode ser um processo à espera de input do utilizador.

Existe uma queue para cada estado (note-se que uma queue não é, necessariamente, uma priority queue) onde os processos são colocados à medida que o seu estado vai alterando. A sua seleção é depois feita através do CPU scheduler. Esta parte enquadra-se no escalonamento de processos que é explicado mais detalhadamente noutra secção.

Implementação

Para armazenar a informação sobre processos, o sistema operativo utiliza uma tabela de processos, com uma entrada por processo. Cada uma destas entradas tem o nome de “Bloco de controlo de processo”. Esta estrutura é necessária para o escalonamento de

processos, uma vez que guarda toda a informação essencial para a recuperar a execução do programa, tal como valores dos registos, stack pointer, pid, etc.



Process management	Memory management	File management
Registers	Pointer to text segment info	Root directory
Program counter	Pointer to data segment info	Working directory
Program status word	Pointer to stack segment info	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		

Figure 2-4. Some of the fields of a typical process table entry.

.

Escalonamento de Processos

Conceito

O conceito é simples. Um processo é executado até ter de esperar (normalmente por um request de I/O). Num sistema simples, o CPU fica numa espera ativa, i.e, sem fazer nada. Está, portanto, a desperdiçar tempo.

É o que acontece num sistema de monoprogramação: um processo tem direito ao CPU enquanto os seguintes ficam em espera ativa, testando se o CPU encontra-se livre (chama-se *polling* e é o caso do MS-DOS).

A ideia é introduzir multiprogramação: utilizar este tempo de forma produtiva. Vários processos são mantidos em memória ao mesmo tempo. Quando um tem de esperar, o SO tira o CPU desse processo e dá a outro. E isto é contínuo: de cada vez que um processo tem de esperar, o CPU vai ser atribuído a outro.

Isto é decidido através do *CPU scheduler*.

É importante recordar a queue de processos e os estados que podem tomar (wait, ready, run).

Burst-Cycle

Se pensarmos que os processos estão constantemente à espera de I/O, concluímos que nesse período não estão a ser feitos cálculos importantes. Por outro lado, quando o processo trata de fazer cálculos, o I/O é mantido a niveis muito reduzidos.

É então fácil de concluir que o tempo é dividido em bursts de I/O ou de CPU. O processo faz cálculos (%CPU elevada), precisa de IO, (%CPU baixa e a de I/O aumenta), volta aos cálculos. Isto é essencial para perceber as estratégias de escalonamento.

Os processos com vários burst cycles de CPU (e, tipicamente, mais longos) chamam-se processos CPU bound, enquanto que os que possuem mais burst cycles (e tipicamente mais longos) de I/O denominam-se I/O bound.

Escalonamento

O escalonamento é necessário em 4 casos:

1. Quando um processo muda de running para waiting (p.e, request de I/O);
2. Quando um processo muda de running para ready (p.e, interrupção);
3. Quando um processo muda de waiting para ready (p.e, completou I/O);
4. Quando um processo termina.

No caso de 1 e 4, o escalonamento necessário é cooperativo, não são precisas técnicas adicionais, além do algoritmo de escolha de próximo processo, para atribuir o CPU a outro processo. Já para 2 e 3, estes só acontecem se o escalonamento for preemptive.

Escalonamento cooperativo vs preemptive

As decisões relativas a escalonamento têm a ver com: **Qual o próximo processo a executar, quando começa a executar e durante quanto tempo executa?**

A primeira destas questões é inerente ao algoritmo usado pelo scheduler, enquanto que as outras duas dependem se o escalonamento é cooperativo ou preemptive.

Escalonamento preemptive ocorre quando um processo possui o CPU mas este é-lhe retirado, pelo SO, para dar a outro processo, sem que o primeiro o tenha explicitamente libertado (exemplos de libertação explícita são passar para a fila de waiting por I/O ou terminar).

A isto chama-se desafetação forçada.

Escalonamento cooperativo permite que um processo execute até, deliberadamente, libertar o CPU - isto ocorre nas situações 1 e 4 acima descritas. O processo termina ou passa para waiting.

Existem algumas plataformas de hardware que apenas funcionam com este tipo de escalonamento, uma vez que não requerem equipamento especial (como um CPU timer) necessário para preemptive.

Por outro lado, também traz menos custos de manutenção (overhead) uma vez que não é necessário tratar da troca de processos nem de guardar o estado de um processo. Simplifica também a complexidade de um scheduler.

Dito isto, percebe-se facilmente as vantagens e desvantagens de escalonamento preemptive. A maior vantagem é que podemos manter a correr vários processos e interagir com eles todos.

Por outro lado, há um overhead adicional dado que para trocar de processo é preciso preservar o estado (registos, stacks, etc) - imaginem estar numa aula e um responsável entra e manda trocar de sala. Temos que arrumar as coisas e copiar o que está no quadro antes de sair.

Também aumenta a complexidade do hardware (p.e, é preciso um timer) e do scheduler.

No entanto, se o escalonamento não for preemptive, é impossível ter mais que um processo a correr simultaneamente. Por exemplo, se pensarmos num sistema de janelas como o do Windows, a cada janela está associado um processo. Portanto, se o escalonamento fosse sempre cooperativo, seria impossível um utilizador manter o MSN aberto e o Internet Explorer (LOL).

MS-DOS usava escalonamento cooperativo e assim foi até Windows 95, onde o escalonamento preemptive foi introduzido. Quantos aos Macintosh, a era pré-OS X contava com escalonamento cooperativo e de OS X em diante é preemptive.

Critérios de Escalonamento

Os critérios utilizados para escalonamento são os seguintes:

- Utilização do CPU (ideal entre 40-90%)
- Throughput - número de processos por unidade de tempo, mede portanto quantos processos consegue concluir
- Turnaround time - tempo desde que um processo entra na queue até que sai
- Waiting time - tempo que um processo está na queue de ready
- Response time - Melhor que o turnaround time num sistema interactivo. Tempo desde que o processo é submetido até obtermos a primeira resposta

Algoritmos de Escalonamento

First Come, First Served

- O mais simples
- FIFO queue
- Fácil de codificar
- Tempo de espera muito elevado
- Favorece os processos CPU Bound:

Imaginem que temos um processo CPU Bound e 2 I/O bound, se os processos I/O bound chegarem primeiro, quando forem para a queue de waiting, o processo CPU bound vai tomar conta do CPU durante um período de tempo longo. Quando este tiver de passar para fila de I/O, o CPU vai ser atribuído um dos processos I/O bound, mas como estes têm muita interação, vão voltar muito rapidamente para a fila de waiting e vão lá ficar até o outro processo libertar o CPU.

- Escalonamento cooperativo

Shortest-Job-First (SJF)

- Associam a cada processo uma estimativa do próximo CPU burst.
- Atribuem o CPU ao processo com menor CPU burst.
- Se o valor for igual, usa-se FCFS.
- É praticamente ótimo, mas é difícil saber a duração do próximo CPU burst.
- Para longa duração, pode-se usar o process limit que o user submete (os users são convidados a estimar o limite de tempo - se o processo demorar mais que isso tem de ser resubmetido).
- Não pode ser implementado em short-term CPU scheduling (quando cada burst dura milissegundos) porque não há maneira de saber a duração de bursts muito pequenos. Pode-se estimar através dos bursts anteriores, no entanto o cálculo consome tempo.
- Pode ser cooperativo ou preemptive (neste último caso, interrompe se chegar um processo com mais prioridade)
- Leva à starvation (ver Preemptive Priority)

Preemptive Priority

- Caso mais geral do SJF.
- Inteiro atribuído ao processo. Pode representar os requerimentos de memória, número de ficheiros abertos, ratio de I/O bursts, etc.
- Podem ser preemptive ou cooperativos.

- Tal como o SJF levam à starvation. Um processo com uma prioridade muito reduzida pode demorar demasiado tempo a ser corrido. Por exemplo, quando desmontaram o IBM 7094 no MIT em 1973 descobriram um processo de baixa prioridade que ainda não tinha sido corrido e tinha sido submetido em 1967.
- A solução é atribuir mais prioridade aos processos mais antigos à medida que o tempo passa.

Round Robin

- Feito para sistemas de partilha de tempo.
- Semelhante ao FCFS mas com preemption.
- É gerado um time quantum, geralmente de 10 a 100 milisegundos.
- A ready queue torna-se circular. O CPU roda por todos os processos, dando a cada um o tempo correspondente a um quantum.
- 2 casos: ou o processo acaba antes do fim do quantum (liberta ele próprio o CPU e o scheduler segue para o próximo processo) ou o quantum chega ao fim (o temporizador acaba, vai causar uma interrupção, o estado do processo é guardado e é colocado no final da waiting queue).
- Se o quantum for muito elevado, comporta-se como FCFS.
- Se o quantum for muito reduzido, o overhead de troca de processo toma conta do CPU e cada processo passa muito pouco tempo no CPU.
- Como o SJF requer vários cálculos, o tempo de resposta do RR é melhor

Multilevel Feedback-Queue

- Há várias queues de ready com diferentes níveis de prioridade.
- Se um processo usa demasiado CPU é movido para uma queue com menos prioridade.
- Só quando todos os processos da queue 0 terminam é que avança para a queue 1.
- Processos I/O bound e interativos são movidos para a queue 0.
- Para evitar starvation, os processos que estejam numa queue com menos prioridade há mais tempo, passam para queues com mais prioridade.
- Se estiver a executar um processo da queue 1 e chegar um da queue 0, há desafetação forçada.
- Na prática: um processo que fique ready é colocado na queue 0 e dado um certo quantum. Se não terminar dentro desse quantum, é movido para a queue 1 com um quantum maior. Se acontecer o mesmo é colocado na queue 2 (tipicamente a partir daqui é FCFS).
- Na queue 0 e 1 é um misto de Round Robin com SJF (RR dentro da própria queue, SJF quando está na queue 1 e chega um processo para a queue 0).
- Como consequência, os processos com CPU bursts reduzidos são servidos rapidamente passando para o I/O. Os processos com CPU burst intermédios (queue 1) também são servidos rapidamente, mas com menos prioridade.

Níveis de Escalonamento

Dado que há muitos critérios de escalonamento, normalmente são divididos em níveis, para evitar a sobrelotação de projetos.

Nível 0: Só despacha o que está na RAM.

Nível 1: Decide que processos são multiprogramados (alguns são swapped out)

Nível 2: Não deixa criar processos.

À medida que o CPU vai ficando sobrecarregado, o nível de escalonamento aumenta.

Sincronização de Processos

RACE CONDITION

Race Condition é a situação em que vários processos alteram os mesmos dados concorrentemente, e o resultado das suas execuções depende da ordem pela qual os processos manipulam os dados. Por exemplo, imaginemos dois processos: o primeiro altera o valor da variavel x para 1, e o segundo altera o valor da variavel x para 0. Ora, se executarmos esses dois processos concorrentemente, o valor final de x depende da ordem pela qual executamos os processos.

Para resolvermos esse problema, precisamos de sincronizar os processos de alguma forma. Vamos apresentar algumas soluções:

PROBLEMA DA REGIÃO CRÍTICA

A possibilidade de execução “simultânea” leva ao acesso em concorrência a recursos partilhados, feito a zonas de endereçamento partilhadas ou, na maior parte dos casos, a ficheiros. Para garantir a coerência dos dados é necessário que os processos cooperem e acedam ordenadamente aos recursos partilhados.

Para um dado recurso partilhado X, cada processo “declara” as regiões do seu código que acedem ao recurso como REGIÕES CRÍTICAS. A execução de uma região crítica por parte de um processo está dependente deste receber garantias de que nenhum outro processo executará a sua região crítica.

Algumas soluções para esse problema:

Solução de Peterson

- Só funciona com 2 processos
- Variavel “turn” que indica qual dos processos pode entrar na região critica
- Array flag indexado pelo número do processo (0 se for o primeiro a entrar, 1 se for o segundo) com o booleano a TRUE

- Quando entra na região crítica, enquanto a vez é do processo atual e o outro está interessado em entrar, fica num ciclo de espera até o outro processo chamar a função de saída
- Essa função, define que o array de interessados do processo que sai fica a falso
- Esta solução não é aceitável nos sistemas modernos devido às intruções de load/store não garantirem que as variáveis vão ser colocadas a tempo.

```
#define FALSE  0
#define TRUE   1
#define N      2                        /* number of processes */

int turn;                               /* whose turn is it? */
int interested[N];                      /* all values initially 0 (FALSE) */

void enter_region(int process);         /* process is 0 or 1 */
{
     int other;                         /* number of the other process */

     other = 1 - process;               /* the opposite of process */
     interested[process] = TRUE;        /* show that you are interested */
     turn = process;                    /* set flag */
     while (turn == process && interested[other] == TRUE) /* null statement */ ;
}

void leave_region(int process)          /* process: who is leaving */
{
     interested[process] = FALSE;       /* indicate departure from critical region */
}
```

Figure 2-24. Peterson's solution for achieving mutual exclusion.

Alternativamente, em vez de ficar em espera ativa num ciclo, pode fazer sleep.

SEMÁFOROS

Outra solução para o problema da região crítica é semáforos. Um semáforo S é um inteiro que varia com o uso de funções wait() ou P() e signal() ou V(). wait() decrementa S, signal() incrementa S. Estas duas funções devem ser executadas sem interrupção.

Existem 2 tipos de semáforos. Counting semaphores e binary semaphores.Binary semaphores variam entre 0 e 1. Counting semaphores don't have a limit. Binary semaphores também se podem chamar mutex (mutual exclusion).

Para usar semáforos "mutex" basicamente usamos wait() quando um processo entra na região crítica, proíbindo os outros processos de entrarem (porque S está a 0) e usamos signal() ao sair da região crítica. Nesse momento, outro processo pode entrar.

Counting semaphores são mais utilizados para controlar o acesso a um recurso limitado (tipo, o António só pode executar a função "comerLaranja()" se tiver comida. Se todos os amigos dele chamaram a função wait() (comer, decrementando o número de laranjas), o António fica sem laranjas, e nao pode comer). Inicializamos o semaforo com o número de

recursos disponiveis. O António, ao chamar wait(), repara que não há laranjas (semáforo está a 0), portanto vai esperar que existam laranjas para comer. O semáforo vai ser incrementado quando um dos amigos acabar de comer a sua laranja, e chamar a função signal() (não me perguntem o que é o signal() na analogia das laranjas).

O grande problema desta implementação é que isto vai usar “busy waiting” (também conhecida por polling), ou espera activa, porque um processo que não possa entrar na região crítica vai esperar que o semáforo seja incrementado. Este tipo de semáforos são também chamados “spinlock” porque vão estar a fazer “spin” enquanto esperam para entrar.

Para resolver isso, foi alterada a definição de semáforo. Agora não é só um inteiro, mas sim uma estrutura contendo um inteiro e uma lista de processos em espera para entrar. Ou seja, quando um processo faz wait(), se não puder entrar, adiciona-se à lista de processos em espera. Quando é feito signal(), é retirado o processo da lista. Nas definições de wait() e signal() são chamadas as funções block() e wakeup(), a primeira insere na queue de processos em espera, a segunda retira-os.

Com esta lista de processos em espera, o valor do semáforo pode ser negativo e indicar o número de processos em espera, se for útil nesse algoritmo. Nesse caso também teria de se alterar as definições das funções wait() e signal().

A lista de processos em espera pode ser implementada com apontadores para o Process Control Block de cada processo em espera. Esta lista pode ser uma Queue (FIFO) de forma a remover processos em espera por ordem de entrada.

Em máquinas single-processor, interrupções devem ser desativadas nas funções wait() e signal(), garantindo que estas não são interrompidas pelo sistema (não garante, no entanto que resista a falhas de sistema, bugs, etc.).

Em maquinas multi-processor, é mais complicado, visto que teríamos de desativar interrupções em todos os processadores, o que seria uma tarefa demasiado dificil e afetaria demasiado a performance. Para, de certa forma, resolver o problema, são usados spinlocks dentro das funções wait() e signal(), o que cria busy waiting, mas visto que essas funções são executadas em pequenos intervalos de tempo e raramente, a perda em performance é reduzida. Excepto no caso de programas nos quais a região crítica é longa. Aí a tecnica de busy waiting é extremamente ineficiente.

Na realidade, o busy waiting ainda não foi completamente removido.

DEADLOCKS

Fenómeno que acontece quando um processo nunca consegue sair de um estado de espera, uma vez que está à espera de recursos utilizados por outros processos.

Condições necessárias para que surja um Deadlock:
→ Existir um recurso que não pode ser partilhado, e este está a ser usado por um processo, enquanto que há outros processos que precisam desse mesmo recurso
→ Existir um processo está a usar um recurso, mas, ao mesmo tempo, à espera de recursos adicionais, a serem usados por outros processos.
→ Serem os próprios processos responsáveis pela libertação do recurso que usam
→ Espera circular: um processo está à espera de um recurso a ser usado por outro processo, que por sua vez está à espera de um recurso……..

Formas de lidar com Deadlocks:
→ Assegurar que os deadlocks nunca acontecem (basta fazer com que uma das condições nunca se verifique)
→ Permitir que os deadlocks ocorram, sendo estes detetados e tratados
→ Ignorar o problema (forma mais usada, UNIX, Windows… é o programador que tem de tratar)

Memória Central

Vários processos devem poder ser executados ao mesmo tempo, e para isso é preciso dividir a memória entre eles. É através da gestão da memória que o SO determina quanta memória é alocada para cada processo, e quanto tempo lhe é dado para a utilizar.

À medida que os processos entram no sistema, são inseridos numa “input queue” e ficam à espera que o SO lhes atribua um bocado de memória, de acordo com uma das seguintes estratégias.

Swapping

Tipicamente, a quantidade de RAM necessária por todos os processos activos costuma ser superior à disponível.

Para lidar com isto pode-se recorrer a "swapping", que consiste em mover um processo inteiro para a memória quando é executado, e depois para o disco, onde é guardado até poder

continuar a sua execução. Maior parte dos processos que se encontram num estado "idle" estão guardados em disco para não ocuparem memória. (Existe outra estratégia para lidar com isto: memória virtual).

Exemplo: Se um processo de prioridade elevada surge e precisa de ser executado, um ou mais processos de menos prioridade são "swapped" para o disco, libertando memória que vai ser utilizada pelo novo processo. Quando este termina, os processos que foram guardados em disco voltam ao mesmo espaço de RAM que estavam a utilizar antes.

Alocação Contígua

Estratégia mais antiga e simples de gestão de memória. SO compara tamanho do proximo processo a ser executado com a quantidade de RAM disponível, e se tiver suficiente para esse processo, aloca-a de maneira a que fique contígua à que já estava a ser utilizada por outros processos. Se não existir memória livre suficiente para o processo, ele é adicionado à “input queue”.

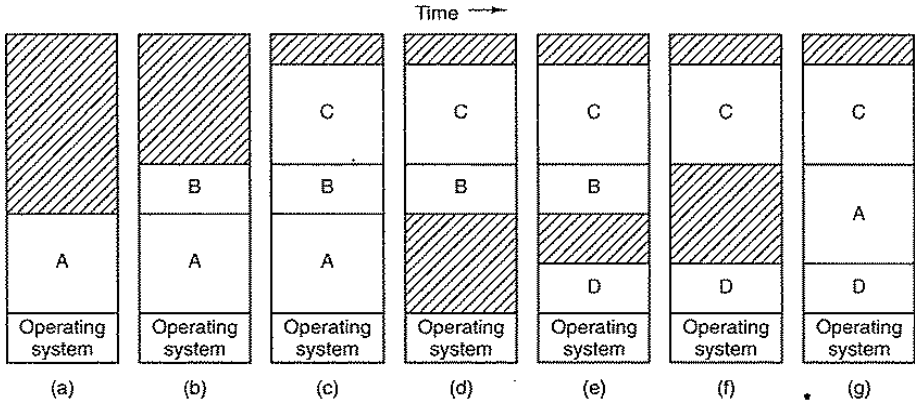


Figure 3-4. Memory allocation changes as processes come into memory and leave it. The shaded regions are unused memory.

Surge outro problema quando o processo aloca memória dinamicamente. Se existir um “buraco” (memória livre) adjacente a este processo, ele pode ser alocado para o processo. Se não, ele terá de se mover para um buraco maior na memória ou terá de ser “swapped” para disco até haver memória o suficiente para o processo terminar a sua execução.

À medida que os processos terminam, vão libertando a memória para outros. Como é alocada memória para cada processo, muitas vezes deixam “buracos” na memória que podem ser pequenos demais para o proximo processo da queue. Este terá de esperar até que espaço suficiente fique disponivel. Os “buracos” vão ficando cada vez mais pequenos e dificeis de preencher. Isto pode provocar uma situação de fragmentação externa, que ocorre quando existe espaço livre em memória dividido em diferentes “buracos” mais pequenos, tornando impossível a alocação contígua.

Segmentação

Consiste na divisão da memória em segmentos/secções. Os seus endereços lógicos incluem um valor que identifica o segmento e o seu offset.

<segment-number, offset>

Este tuplo é traduzido para um endereço de memória. Cada segmento tem um comprimento e um conjunto de permissões. Um processo só pode fazer referência a um segmento se tipo desta for permitido de acordo com as permissões.

Segmentos também podem ser usados para implementar memória virtual, em que ficam associados a uma flag que indica se este está presente na memória física ou não.

A segmentação pode ser implementada de várias formas:

- Sem paging - associado a cada segmento existe informação que indica onde o segmento está na memória ("segment base").
- Com paging - em vez de ter associado uma posição na memória, a informação do segmento inclui um endereço de uma tabela de páginas.

Memória Virtual

A necessidade de memória virtual surgiu quando os programas a serem desenvolvidos eram demasiado grandes para a memória disponível. A solução encontrada consistia em dividir os programas em pedaços chamados *overlays*.

Os *overlays* usados no momento da execução são mantidos na memória principal, os restantes no disco.

Paginação

A alocação de memória não precisa de ser contígua.

A paginação evita a ocorrência de fragmentação externa. Envolve dividir a memória física em blocos de tamanho fixo chamados frames, e dividir a memória lógica em blocos de tamanhos iguais chamados pages. O tamanho das frames e das pages é definido pelo hardware.

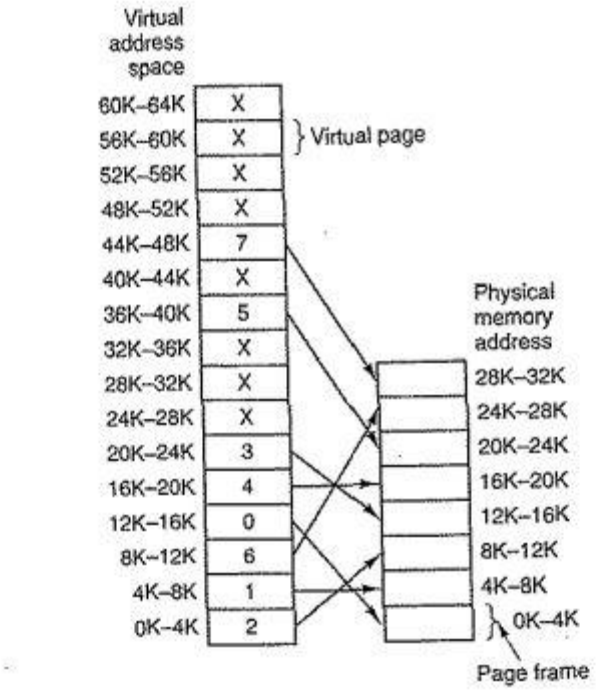
Cada "page" precisa de uma frame, por isso se um processo tiver n pages, precisa de n frames para poder ser executado. Se isto se verificar, essas n "frames" serão alocadas. A primeira page vai para uma das frames alocadas e o seu número de frame é guardado na tabela de páginas (isto repete-se para todas as pages do processo).

Sempre que uma page necessária à execução de um processo não se encontra numa frame, ocorre uma "page fault". Embora page faults sejam normais, quando estes se dão constantemente dá-se o nome de thrashing, fenómeno que causa um impacto notável no desempenho da máquina uma vez é gasto tempo de CPU para efetuar a reposição de páginas.

Tabela de Páginas

A tabela de páginas inclui o endereço base de cada “page” da memória física.

O endereço de base em conjunto com o offset da “page” definem o endereço da memória física.



Algoritmos de Substituição de Páginas

Not Recently Used Page

- Cada página de memória é mantida com 2 bits extra que indicam o estado
- Esses bits são atualizados a cada instrução, de acordo com o grau de utilização
- Quando ocorre um page fault, o SO inspecciona estes bits e remove um ao acaso dos que tenham bits correspondentes ao código “não usado recentemente”.

FIFO Page Replacement Algorithm

- Remove-se a page mais antiga
- Este algoritmo é pouco usado porque ser mais antiga não significa que seja menos importante ou que não tenha sido usada recentemente.

Least Recently Used

- Algoritmo ótimo de substituição de páginas;

- Tem custos muito elevados de manutenção: é necessário manter uma lista ligada de todas as páginas em memória, que se seja atualizada a cada alteração das referências na memória.
- Alternativamente, pode ser feito através de hardware que forneça contadores (um por frame) e estes sejam atualizados a cada instrução.
- Como nenhuma máquina tem tal hardware equipado, o algoritmo mais aproximado e usado é o NRU.

RAID

Basicamente organiza varios discos de diferentes formas (dependendo do nivel) e o sistema vê o RAID como uma única unidade logica.

-Focado no desempenho e disponibilidade de dados

Mirroring:

Cada disco lógico na estrutura RAID contém dois discos físicos, iguais. No caso de falha de um dos discos, os dados podem ser recuperados.

Esta é a técnica mais lenta (mas mais simples) de redundância (dados redundantes em termos de informação, mas uteis para recuperar de falhas). É a mais lenta por razões óbvias, tem de copiar tudo.

Mas as falhas, normalmente nao sao independentes, e sao causadas por falhas de energia. Para isso é usada a NVRAM (nao volatil), uma write-back cache que resiste a falhas de energia, para a escrita poder ser resumida apos uma falha de energia.

Com mirroring, visto que há dois discos iguais, a taxa de leitura é o dobro, visto que podem ser lidos em paralelo.

Data Striping

Divisão dos dados (por bit ou bloco) por todos os discos da RAID.

BIT-LEVEL STRIPING

Divide os vários bits de um byte por todos os discos. Quando é usada esta técnica, usam-se N discos na RAID, com N sendo múltiplo ou divisor de 8, para distribuir os bits igualmente por cada disco.

BLOCK-LEVEL STRIPING

Basicamente o mesmo que por bit, mas com blocos de ficheiro. Não está bem detalhada por isso nao sei bem qual é a definiçao de bloco neste contexto.

Há mais tipos de striping, nao falados.

Com Data striping, todos os discos participam em cada acesso, logo a taxa de acessos ao RAID é semelhante ao que seria com um so disco, mas cada acesso consegue ler muito mais do que seria num so disco (se a RAID tivesse 8 discos, cada acesso poderia ler 8 vezes mais do que num disco unico)

Com este tipo de paralelismo, os acessos pequenos têm mais throughput (o sistema consegue fazer mais coisas num menor período de tempo)
Em acessos de maior escala, demora menos tempo.

NIVEL 0:
Não usa redundancia, so tem Block-level striping
NIVEL 1:
Usa mirroring. se tem 4 discos de dados, tem 4 discos de overhead
NIVEL 2:
Usa memory-style error-correcting-code. Para cada byte do sistema de memória existe um bit de paridade, que indica se o número de bits existentes é par (0) ou impar(1). Ora, se houver um erro de um só bit, é detetado pelo ECC, porque ve que a paridade é diferente.
Para guardar estas paridades, se tem 4 discos de dados, tem 3 discos de overhead.
Pode-se usar data striping para guardar paridades.

DESVANTAGENS

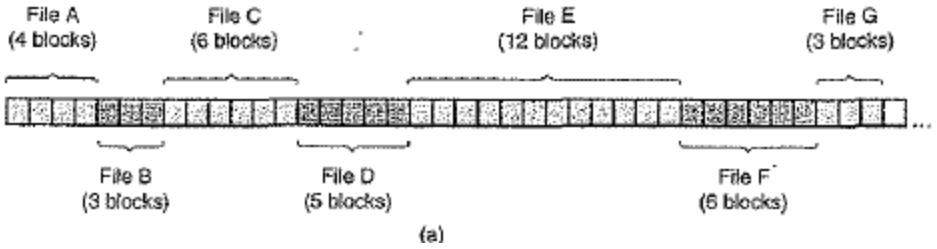
A disponibilidade de dados nao é 100% garantida. Podem haver apontadores errados, writes incompletos que não são recuperados, etc. O sistema RAID protege contra problemas relacionados com armazenamento de dados fisicos, mas não contra bugs de hardware ou software.

FILE SYSTEM

Métodos de estruturação do file system:

1. Alocação Contígua

Os ficheiros, ou melhor, os seus blocos sao todos alocados contíguamente:

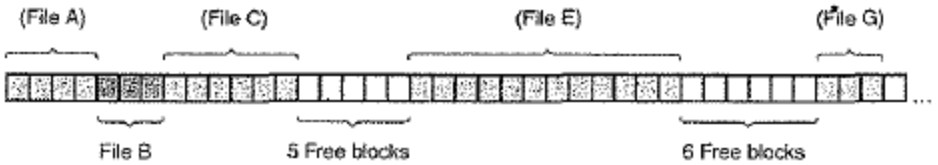


(a)

Vantagens:

- Implementação simples, basta saber o endereço de inicio e o numero de blocos de um ficheiro.
- Excelente performance de leitura por ser tudo contíguo.

Desvantagens:



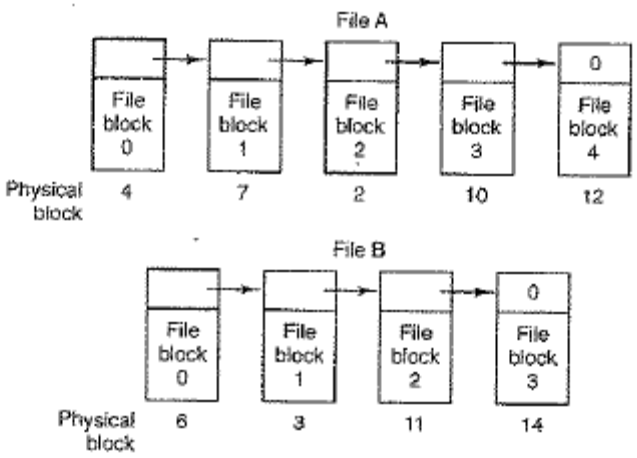
(b)

- Fragmentação (em cima). Quando se remove ficheiros ficam buracos, e ao escrever novas coisas temos de escrever no fim, não nos buracos, porque não sabemos onde estão e se cabem la. Desfragmentaçao demora muito porque...bem estás a escrever um monte de cenas.

No entanto, alocação contigua é usada para CDs, etc. porque nunca vais remover nada.

2. Alocação por listas ligadas

Um ficheiro é uma lista ligada de blocos. A primeira “word” de cada elemento da lista ligada é um apontador para o proximo bloco.



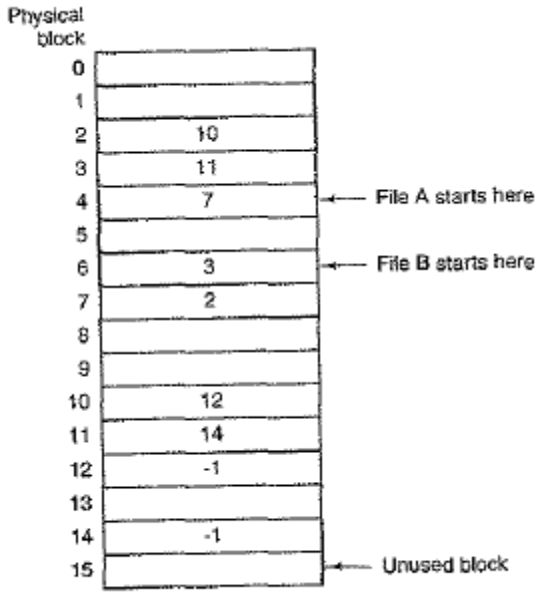
Vantagens:

- Não há problemas de fragmentação, não há buracos.

Desvantagens:

- Leitura extremamente lenta. Para se chegar a um bloco N, temos de passar por N-1 blocos anteriores.
- Por causa do espaço necessário para o apontador, o tamanho de cada elemento da lista ligada não é potência de 2 e isso torna a leitura e escrita menos eficiente. Não percebi bem porquê.

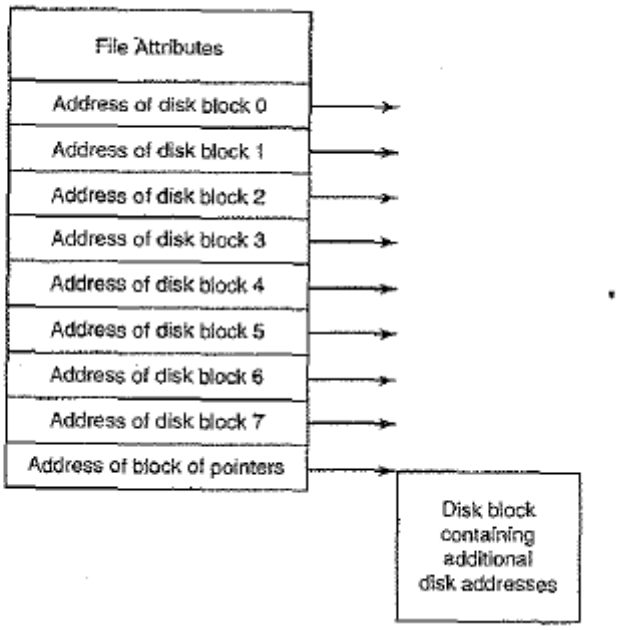
Estas desvantagens podem ser resolvidas criando uma tabela de apontadores para cada bloco de todos os ficheiros, em memória, sempre aberta. Para acedermos a um ficheiro temos de saber o integer do indice do inicio do ficheiro, só, porque a partir daí percorremos a lista ligada. Demonstrado em baixo:



Há problemas com isto porque a tabela aumenta linearmente com o tamanho do disco, e fica muito grande, chegando aos 600-800 MB num disco de 200 GB e blocos de 1 KB..

3. i-nodes

i-node é uma data structure que é associada a um ficheiro. É uma tabela guardada em memória contendo os endereços de cada bloco do ficheiro, esse blocos estão em disco. Esta tabela tem um número fixo de endereços, o que acontece quando fica cheia? Para isso é guardado um apontador, no final da tabela, para um bloco com os endereços que não couberam na tabela. A imagem em baixo deve tornar isso mais claro:



Se cada i-node ocupar N bytes, e estiverem K ficheiros abertos, precisamos apenas de NK bytes para ter os ficheiros aberto. Isto é claramente melhor que a opção das listas ligadas, visto que o espaço ocupado em memória não aumenta linearmente com o tamanho do disco.

Log-structured file system

Tudo tem melhorado, RAM, Cache, etc. Mas há um bottleneck criado pelo Disk Seek Time. Para além disso a criação de ficheiros é muito lenta, há que criar um i-node para a diretoria e ficheiro, o bloco da diretoria e o ficheiro em si.

Os gajos que criaram o LFS acharam que isso era chatinho, por tanto quiseram redesenhar UNIX e estruturar o file system como um log. Tudo. Para sempre.Sem fim.

Assim, sempre que é pedida uma escrita, esta é buffered e colocada num segmento de escritas, em memória. Quando é necessário escrever esse segmento, todo ele é escrito, contiguamente, no fim do log. No inicio de cada segmento, ha informações acerca desse segmento. Tá tudo la metido no segmento, i-nodes, blocos, blocos de diretoria, tudo.

Os i-nodes têm a mesma estrutura que em UNIX, mas estão todos espalhados pelo log. Ora, como se encontra um i-node para ler? Não se pode calcular o endereço a partir do i-number (que identifica o i-node) . Portanto criou-se um map de i-nodes, indexado pelo i-number. Este map é guardado tanto em disco como em cache, para ser rapido.

Uma vantagem deste sistema é que, no caso de um crash durante a escrita, as escritas mais recentes estão no segmento mais recente do log. A partir desse log pode-se fazer as escritas novamente. Este sistema é então um dos melhores no caso de crash.

Há um problema, quando se faz overwrite de ficheiros, os seus i-nodes apontam para novos blocos, mas os antigos ficam, e o sistema não pode escrever neles. Para resolver isso, o LFS tem um **cleaner**, que está sempre a percorrer o log. Começa pelo primeiro segmento do log. Compara o sumário do segmento com o map de i-nodes atual. Vê o que está atualizado e a ser usado. O que estiver igual ao atual é marcado para escrita no proximo segmento, o resto vai para o lixo e o primeiro segmento é declarado como livre e o sistema pode escrever nele outra vez se precisar. Continua a fazer isso para todo o log.

Mas sinceramente, não compreendi muito bem este parágrafo. Portanto, se não perceberam nada do que eu disse acerca do cleaner, aqui está:

> To deal with this problem, LFS has a **cleaner** thread that spends its time scanning the log circularly to compact it. It starts out by reading the summary of the first segment in the log to see which i-nodes and files are there. It then checks the current i-node map to see if the i-nodes are still current and file blocks are still in use. If not, that information is discarded. The i-nodes and blocks that are still in use go into memory to be written out in the next segment. The original segment is then marked as free, so that the log can use it for new data. In this manner, the cleaner moves along the log, removing old segments from the back and putting any live data into memory for rewriting in the next segment. Consequently, the disk is a big circular buffer, with the writer thread adding new segments to the front and the cleaner thread removing old ones from the back.

Tudo isto é bastante complexo, mas através de medições comparativas com UNIX, confirmou-se que escritas pequenas foram muito mais rápidas. Leituras e escritas de grande dimensão tiveram uma performance igual ou melhor que UNIX.