----------Capitulo 1-INTRODUÇÃO----------

1.1- Objectivos e funções do SO:

Um PC moderno consiste num ou mais processadores, memória principal, discos, impressoras, teclado,rato,monitor e outros aplicativos.
Escrever programas que controlem estes aplicativos e os usem correctamente é um trabalho extremamente difícil.
Por essa razão os PC's estão equipados com uma camada de software chamada de Sistema operativo cuja função é administrar todos estes aplicativos e providenciar ao utilizador programas com um interface simples com o hardware.
Um sistema operativo não e nada mais nada menos do que um programa (ou um conjunto de programas) cuja função é gerir os recursos do sistema (defenir qual programa recebe a atenção do processador, gerir memoria, criar sistema de arquivos, etc), alem de fornecer interface entre o computador e o utilizador, o primeiro programa que a maquina executa quando o computador é ligado e só deixa de funcionar quando este é desligado.
O sistema operativo varia a sua execução com a de outros programas, como se estivesse vigiando, controlando e orquestrando todo o processo computacional.
Pode ser visto de 2 formas:

1- Na perspectiva do utilizador (visão top-down): é uma abstracção do hardware, fazendo o papel do intermediário entre o programa e os componentes físicos do computador(hardware).

2- Numa visão bottom-up: gerir recursos, isto é controlar quais processos podem ser executados,quando,que recursos(memoria,disco,etc) podem ser utilizados.

1.2-Evolução e estrutura dos sistemas operativos modernos

Evolução:

The First Generation (1945-55) Vacuum Tubes and Plugboards
The Second Generation (1955-65) Transistors and Batch Systems
The Third Generation (1965-1980) ICs and Multiprogramming
The Fourth Generation (1980-Present) Personal Computers

Estrutura dos SO's modernos:

1-sistemas monolíticos:

1.1-um programa principal que invoca o procedimento do serviço requisitado.
1.2-Uma serie de procedimentos de serviço que contem as chamadas ao sistema.
1.3-uma serie de procedimentos de utilidade que ajudam os procedimentos de serviço.

2-Layered systems:

Layer	Function
5	Operador
4	Programas do utilizador
3	Gestão de Input/output
2	Processos de comunicação com o operador
1	Gestão de memoria
0	Alocação do processador e multiprogramação

3-Máquinas virtuais.

----------Capitulo 2 – Gestão de Processos----------

Quando um computador tem multiprogramação frequentemente tem múltiplos processos a competir pelo CPU ao mesmo tempo. Esta situação ocorre quando dois ou mais processos estão simultaneamente no estado de "pronto".Uma escolha tem de ser feita sobre qual processo pode executar de seguida. Aqui entra uma parte do sistema operativo chamada de escalonador de processos que tem um algoritmo próprio chamado de algoritmo de escalonamento.
Quando fazer o escalonamento?

1- quando um processo é criado uma decisão tem de ser tomada se executa o processo pai ou o filho. Como os dois estão prontos a ser executados o escalonador pode decidir executar um ou outro.

2- a decisão de escalonamento deve ser feita quando um processo existe. Se um processo já não puder ser executado(por já não existir) outro processo deve ser escolhido do conjunto dos processos prontos a executar.

3- Quando um processo bloqueia no I/O, num semáforo ou por outra razão outro processo deve ser seleccionado para executar

4- quando uma interrupção de I/O ocorre uma decisão de escalonamento pode ser feita. Se a interrupção veio do I/O que já completou o seu trabalho outro processo que estava bloqueado á espera do I/O pode agora estar pronto a ser executado. O escalonador deve decidir se deixa esse processo executar, se o processo que estava a executar quando se deu a interrupção deve ser escalonado ou se um outro processo.
Os ciclos de interrupção podem ser dados por clock(dá por exemplo interrupções periódicas de 50 em 50 hertz).Podem ser dadas por um tempo de execução(um processo executa durante esse tempo e depois um sinal de interrupção é dado).
Apesar de todas as circunstancias é necessário haver justiça, isto é processos comparáveis devem ter serviços comparáveis. Dar mais tempo a um processo do que a outro processo equivalente não é justo. Mas diferentes categorias de processos podem ser criadas e podem ser tratadas de forma diferente.

Throughput: numero de serviços que o sistema termina por hora.

Turnaround time:estatisticamente é a percentagem de tempo desde o momento em que um serviço batch é submetido até ao momento em que é terminado. Mede quanto tempo o utilizador do serviço tem de esperar pelo output.

FIFO:
Talvez o algoritmo de escalonamento mais simples de todos. Neste processo os processos "inscrevem-se" no CPU na ordem em que chegam. Basicamente há uma única fila de processos prontos a executar. Quando o primeiro processo começa a executar é autorizado a executar o tempo que quiser. Os processos á medida que vão chegando vão para o fim da fila. Quando o processo em execução bloqueia o primeiro processo da fila executa de seguida. Quando um processo bloqueado (depois de ter executado) fica de novo pronto a executar passa para o fim da fila. Este algoritmo é de fácil compreensão e programação.

Serviço mais pequeno primeiro:

Quando vários e igualmente importantes processos estão na fila de entrada á espera de processar o escalonador escolhe aquele que tem a tarefa mais pequena

Tempo em falta mais pequeno de seguida:

Com este processo o escalonador escolhe sempre o processo com o mais curto tempo de execução em falta.O tempo de execução tem de ser conhecido.Quando um novo processo chega o seu tempo em falta é comparado com o que falta do processo em execução, caso seja mais curto o processo em execução é suspenso e o novo processo inicia o seu trabalho.

Escalonamento de 3 níveis:

Quando um processo chega ao sistema é inicialmente posto numa fila de entrada que está no disco.O escalonador de admissão decide que processo a admitir no sistema.Os outros ficam nessa fila até serem admitidos.O escalonador de admissão é livre de manter alguns processos na fila de entrada e admitir outros processos que chegaram mais tarde desde que assim escolha.

Round-Robin (RR):

Um dos mais antigos, mais simples e mais usados algoritmo.A cada processo é atribuído um intervalo de tempo no qual ele é autorizado a executar.Se ele ao fim desse intervalo de tempo ainda estiver a executar o CPU é-lhe retirado e dado a outro processo. Se o processo for bloqueado ou tiver terminado antes desse intervalo de tempo o CPU e dado a outro processo quando este ultima bloqueia. Quando um processo já usou o seu intervalo de tempo passa para o fim da lista.

Escalonamento com prioridade:

A necessidade de ter em conta factores externos leva a um escalonamento com prioridade. A ideia básica é que a cada processo é atribuída uma prioridade e o processo pronto a executar com a prioridade mais alta é autorizado a executar.

----------Capitulo 3-Noções de programação concorrente-------

Processos:

Todos os computadores modernos podem fazer varias coisas ao mesmo tempo.
Enquanto esta a executar um programa do utilizador o computador pode estar ao mesmo tempo a ler de um disco e enviar texto para o monitor ou uma impressora. Num sistema de multi programação o CPU alterna de programa em programa executando-os durante alguns milissegundos.

O modelo do processo:

Todo o software executável num computador, as vezes inclusive o sistema operativo esta organizado num numero de processos sequenciais. Um processo é só um programa em execução, incluindo os valores do "program counter", registos e variáveis. Conceptualmente um processo tem o seu próprio CPU virtual. Na realidade CPU vai alternando de processo em processo. Esta rápida alternação é chamada de multiprogramação.
A diferença entre um processo e um programa é mínima mas crucial.
A principal ideia é que um processo é uma actividade de algum tipo. Tem um programa,um input, um output e um estado.Um processador único pode ser partilhado por vários processos com algum algoritmo de escalonamento.

Criação de um processo:

Um sistema operativo tem de alguma forma garantir que todos os processos necessários existem.
Há 4 eventos principais que podem causar a criação de um processo:

1- Inicialização do sistema
2- Execução de uma criação de processo por um processo em execução.
3- Utilizador pede para criar um processo.
4- Inicialização de uma tarefa de batch.

Quando se faz boot ao sistema operativo tipicamente vários processos são criados, alguns destes são processos que interagem com os utilizadores (humanos) e executam tarefas para eles. Outros processos correm em background e não estão associados a nenhum utilizador em particular mas tem uma função específica.
Depois dos processos criados com o boot outros novos processos podem ser criados.
A criação de novos processos é particularmente útil quando a tarefa a executar pode ser facilmente dividida em varias partes que são independentes.
Nos sistemas iterativos os utilizadores podem iniciar um programa executando um comando ou clicando no icon do programa, estas acções iniciam um novo processo e executa o programa seleccionado nele.
Em todos os casos um novo processo é criado por um processo existente que executa uma chamada de criação de processos ao sistema .Esse processo pode ser um processo do utilizador em execução, um sistema de processos invocado do teclado ou rato ou um processo de gestão da batch.A chamada ao sistema diz ao sistema operativo para criar um novo processo e indica de uma forma directa ou indirecta que programa deve ser executado nele.
No Unix só existe uma chamada ao sistema para criar processos: **fork()**.

Esta chamada ao sistema cria um clone do processo que o criou. Depois do fork ambos os processos, tanto o pai como o filho têm a mesma imagem de memória, as mesmas variáveis de ambiente, etc.

Terminar um processo:

Depois de um processo ser criado ele começa a executar e faz o seu trabalho. Contudo nada dura para sempre, nem os processos e mais tarde ou mais cedo o processo vai terminar normalmente devido a:

1- Saída normal (voluntaria).
2- Saída por erro (voluntaria).
3- Erro fatal (involuntária).
4- Morto por outro processo (involuntária).

1-Muitos processos terminam porque já fizeram o seu trabalho. Quando um processo termina o seu trabalho este executa uma chamada ao sistema para dizer ao sistema operativo que já terminou (faz um **exit**).

2-Este segundo caso ocorre quando por exemplo fazemos: cc foo.c
Ao fazer a compilação de foo.c ele verifica que este não existe e dá uma mensagem de erro.

3-Esta razão ocorre quando ocorre um erro causado por outro processo normalmente devido a um bug do programa (por exemplo fazer uma divisão por zero).

4-Esta razão ocorre quando um processo executa uma chamada ao sistema dizendo ao sistema operativo para terminar (kill) outro processo. O processo que faz o kill tem de ter autorização para tal.

Hierarquia de processos:

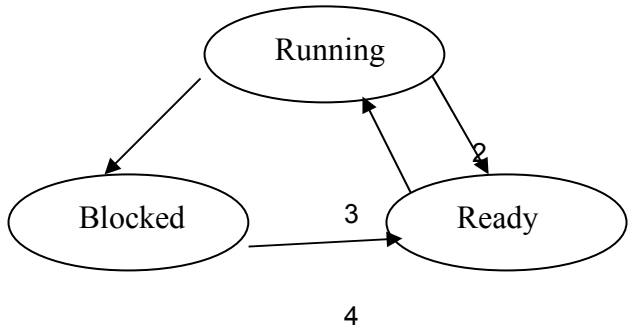
Nalguns sistemas quando um processo cria outro o processo pai e o processo filho continuam de alguma forma associados. O processo filho pode criar outros processos criando assim uma hierarquia de processos. Ao contrario do mundo humano que são necessários 2 humanos para criar um filho, um processo apenas tem um pai mas pode ter zero ou mais filhos.Em Unix um processo, os seus filhos e os outros descendentes formam um grupo de processos.Quando um sinal é enviado esse é entregue a todos os membros do grupo.De forma individual cada processo pode apanhar o sinal, ignorar o sinal ou executar a acção desse sinal que é de ser terminado (kill) pelo sinal. Quando o Unix é inicializado um processo especial é criado: **Init.**
Quando este começa a executar ele lê um ficheiro que diz quantos terminais tem depois faz um fork para um novo processo por terminal.

Estados de um processo:

Embora cada processo seja uma entidade independente, os processos muitas vezes precisam de interagir com outros processos. Um processo pode gerar output que outro processo necessita como input.
Quando isto acontece o processo que necessita do output de outro processo, caso este não esteja disponível deve ser bloqueado até ao output do outro processo estar disponível.

Quando um processo bloqueia é porque não pode continuar.Também é possível que um processo pronto a executar ser parado porque o sistema operativo decidiu alocar o CPU para outro processo por algum tempo.
Estas duas condições são completamente diferentes, o primeiro caso a paragem deve-se á impossibilidade de continuar o segundo caso é uma técnica do sistema.
Um processo pode estar em 3 estados diferentes:

1- em execução(usando o CPU nesse instante).
2- Pronto(temporariamente parado para deixar outro processo correr).
3- Bloqueado(Não pode executar ate que um determinado evento externo ocorra).



1- Processo bloqueia por input
2- Escalonador escolhe outro processo
3- Escalonador escolhe este processo
4- Input fica disponível

Implementação de processos:

Para implementar o modelo de processos o sistema operativo mantém uma tabela (um array de estruturas) chamada de tabela de processos com uma entrada por processo.Esta entrada contem informação do estado do processo, o seu program counter, o stack pointer, a alocação de memoria o status dos seus ficheiros abertos e tudo o resto acerca do processo que necessite de ser guardado quando um processo passar de em execução para pronto a executar ou bloqueado.

Threads:

Nos sistemas operativos tradicionais cada processo tem um espaço de endereçamento e uma única thread de controlo.De facto isto é quase a definição de um processo.Mas frequentemente há situações em que é desejável ter múltiplas threads de controlo no mesmo espaço de endereçamento.

O modelo de uma thread:

O modelo de um processo é baseado em dois conceitos independentes: agrupamento de recursos e execução.As vezes pode ser útil separar estas partes, é aqui que as threads entram.
Uma forma de olhar para um processo é que estes agrupam recursos relacionados.Um processo tem um espaço de endereçamento que contem texto e

dados do programa bem como outros resursos.Estes recursos podem incluir ficheiros abertos,processos filho,alarmes pendentes,etc.Juntando-os em forma de processo podemos geri-los de uma forma mais fácil.
Uma thread tem um "program counter" que mantém a informação sobre que instrução a executar de seguida. Tem registos que guardam as variáveis correntes. Tem uma "stack" que contem a história da execução.
Embora uma thread tenha de executar num processo,a thread e o seu processo são conceitos diferentes e podem ser tratados em separado.Processos são usados para agrupar recursos.Threads são as entidades escalonadas para executar no CPU.
O que as threads adicionam ao modelo de um processo é de permitir múltiplas execuções que podem decorrer no mesmo ambiente de um processo.Tendo múltiplas threads a correr em paralelo num processo é análogo a ter múltiplos processos a correr em paralelo num computador. Inicialmente as threads partilham um espaço de endereçamento , ficheiros abertos , e outros recursos.
Agora processos partilham memoria física,discos,impressoras e outros recursos.Devido as threads terem algumas propriedades dos processos as vezes são chamadas de processos de peso leve.O termo "multithreading" é usado para descrever a situação de permitir múltiplas threads no mesmo processo.
Diferentes threads num processo não são bem independentes como o são processos diferentes.Todas as threads tem exactamente o mesmo espaço de endereçamento o que significa que também partilham as variáveis globais.Como cada thread pode aceder a todo o espaço de memoria que esta no espaço de endereçamento de um processo,uma thread pode ler , escrever e até limpar a stack de outra thread.
Não há nenhuma protecção entre threads porque não é possível haver e não deveria ser necessário haver.Ao contrário de processos diferentes que posem ser de utilizadores diferentes e que podem ser hostis uns para os outros,um processo é sempre de um único utilizador que presumivelmente criou múltiplas threads de forma a que possam cooperar entre si e não lutar entre si.Além de partilharem um espaço de endereçamento todas as threads partilham o mesmo lote de ficheiros abertos, processos filho , alarmes,etc.
Tal como um processo tradicional uma thread pode estar a executar,bloqueada, pronta a executar ou terminada.Uma thread em execução tem o CPU e está activa.Uma thread bloqueada está á espera que algum evento a desbloqueie.Uma thread pode bloquear á espera que algum evento externo ocorra ou espera que alguma outra thread a desbloqueie.Uma thread pronta a executar é escalonada para executar assim que a sua vez chegar.A transição entre os estados de uma thread é a mesma que a transição entre estados de um processo.
Quando o multithreading está presente, os processos normalmente começam com uma única thread presente.A thread tem a habilidade de criar novas threads através de um **thread_create.**
Quando uma thread termina o seu trabalho ela pode sair através de um procedimento:**thread_exit.**Aqui a thread é eliminada e não será mais escalonada.O procedimento de por uma thread á espera é:**thread_wait.**
Este procedimento bloqueia a thread até uma thread específica fazer exit.
O procedimento **thread_yield** permite á thread abdicar voluntariamente do CPU para deixar outra thread executar. Este procedimento é importante porque aqui não existe uma interrupção por "clock".

Uso de threads:

A principal razão para o uso de threads é que em muitas aplicações, múltiplas actividades estar a decorrer ao mesmo tempo.Algumas destas podem bloquear de tempos em tempos. Decompondo a aplicação em múltiplas threads sequenciais que correm de uma forma quase paralela o modelo do programar fica simplificado.Com threads temos um novo elemento: a habilidade de as entidades paralelas de partilharem entre elas um espaço de endereçamento e todos os dados.
O segundo argumento para termos threads é que como elas não tem nenhum recurso estas são mais fáceis de criar e destruir que um processo.Em muitos sistemas criar uma thread é 100 vezes mais rápido que criar um processo.
O terceiro argumento para usar threads é um argumento de performance.

Implementar threads no espaço do utilizador:

Há 2 formas principais para implementar um pacote de threads: no espaço do utilizador e no espaço do "kernel".
O primeiro método consiste em por os pacotes de threads inteiramente no espaço do utilizador. O "kernel" não tem conhecimento nenhum sobre os pacotes.
A primeira vantagem é que as threads do nível do utilizador podem ser implementadas num sistema operativo que não suporta threads.
Quando as threads são geridas no espaço do utilizador cada processo necessita da sua própria tabela de thread.
Quando uma thread faz algo que pode causar o seu bloqueio local faz uma chamada de um procedimento do sistema.este procedimento verifica se a thread deve ser posta num estado de bloqueio, caso seja põe os registos da thread na tabela da thread, bloqueia na tabela para um estado pronto a executar e recarrega os registos da maquina com os novos valores da thread.Quando o program counter e o stack pointer forem mudados a nova thread volta "á vida" automaticamente.
Quando uma thread termina a sua execução por um momento através de um procedimento,este pode guardar a informação da thread na sua tabela.Depois pode chamar o escalonador de threads para escolher outra thread para executar.O procedimento que guarda o estado da thread e o escalonador é um procedimento local ,sendo assim invoca-lo é muito mais eficiente do que invocar pelo kernel.
Entre outras coisas não é necessário seguir o seu rato, a memoria cache não precisa de um "flush".Isto torna o escalonamento de uma thread muito rápido.
Estas threads ainda tem outras vantagens, elas permitem cada processo tenha o seu próprio algoritmo de escalonamento.
Mas apesar da sua melhor performance estes pacotes de threads têm outros problemas mais graves. Em primeiro esta o problema de como as chamadas ao sistema de bloqueio são implementadas.Imagine-se a situação em que a thread le de teclado antes de alguma tecla ter sido premida.Deixar a thread fazer a chamada ao sistema é inaceitável já que isso ira parar todas as threads.Um dos principais objectivos do uso de threads é de permitir cada uma de usar chamadas de bloqueio,mas prevenir que uma thread bloqueada de afectar as outras.Com chamadas de bloqueio ao sistema o objectivo não é atingido.

Implementar threads no kernel:

Nesta implementação nenhum sistema de "run-time" é necessário, bem como não existe nenhuma tabela de thread em cada processo.O kernel é que tem uma tabela da thread que regista todas as threads do sistema.Quando uma thread que criar ou destruir outra faz uma chamada ao kernel, que depois faz a criação ou destruição da thread ao actualizar o tabela de thread do kernel.
Todas as chamadas que podem bloquear uma thread estão implementadas como uma chamadas ao sistema. Quando uma thread bloqueia o kernel pode executar outra thread do mesmo processo ou uma thread de outro processo.
Devido ao custo elevado da criação de threads pelo kernel alguns sistemas fazem reciclagem das suas threads.Quando uma thread é destruída é marcada como não executável, mas no kernel as estruturas de dados não são afectadas.Mais tarde quando uma thread tem de ser criada uma thread antiga é reactivada ganhando assim algum "overhead".

3.2.1-Comunicação entre processos:

Frequentemente processos necessitam de comunicar entre si.de uma forma muito breve há 3 assuntos aqui:
O primeiro é a forma como um processo pode passar informação para outro.
O segundo é de garantir que dois ou mais processos não se metem no caminho uns dos outros quando estão a fazer actividades criticas.
O terceiro tem a ver com uma sequência certa quando estão presentes dependências.

3.2.1.1- Race conditions:

Em alguns sistemas operativos processos que estão a trabalhar em conjunto podem partilhar alguns dados armazenados que qualquer um pode ler ou escrever.Esses dados podem estar na memória principal ou num ficheiro partilhado.
Para perceber como a comunicação entre processos funciona imaginemos uma simples comunicação com a impressora.Quando um processo quer fazer uma impressão ele insere o nome do ficheiro numa directoria própria .Outro processo de forma periódica verifica se nessa directoria esta algum ficheiro para imprimir e caso exista este imprime-o e retira o nome do ficheiro da directoria.
Situações do tipo de dois ou mais processos estão a ler ou escrever dados partilhados e o resultado final depende de quem executa num exacto determinado momento são chamadas de "race conditions".

3.2.1.2- Regiões criticas:

A chave para prevenir o problema das "race conditions" é de arranjar uma forma de proibir mais do que um processo de ler e escrever os dados partilhados ao mesmo tempo. Por outras palavras o que precisamos é de **exclusão mútua** isto é alguma forma de garantir que se um processo esta a utilizar num dado momento dados partilhados os outros processos serão proibidos de fazer a mesma coisa.
A parte do programa em que existe acesso á memoria partilhada é chamada de região critica ou secção critica. Se conseguirmos arranjar formas de garantir que dois processos nunca estão numa região critica ao mesmo tempo conseguimos evitar "race conditions".

Precisamos de 4 condições para termos uma boa solução:

1- Dois processos não podem estar em simultâneo na sua região critica.
2- Não podemos fazer pressupostos acerca de velocidades ou nº de CPU's.
3- Nenhum processo a correr fora da região critica poderá bloquear outros processos.
4- Nenhum processo terá de esperar para sempre para entrar na sua zona critica.

3.2.2- Exclusão mutua

3.2.2.1-Desabilitar interrupções:

A solução mais simples é de cada processo desabilitar as suas interrupções logo após entrar na região critica e habilitar as mesmas no instante antes de sair da região critica.Desta forma não podem ocorrer interrupções por clock.Como o CPU só alterna entre processo com interrupções por clock ou outras quando estas estão desabilitadas o CPU não será alternado.
Este processo não é atractivo porque um processo não deverá ter o poder de desabilitar as suas interrupções.Imagine-se que 1 processo desabilita as interrupções e nunca mais as habilita.Da mesma forma com dois ou mais CPU's desabilitar as interrupções afecta apenas o CPU que executou essa instrução.Os outros continuaram em execução podendo aceder á memoria partilhada.

3.2.2.2- Bloquear variáveis:

Esta é uma solução de software.Imaginemos a existência de uma única e partilhada variável (**lock**) inicialmente com o valor zero.quando um processo quer entrar na região critica primeiro testa essa variável, se esta for zero o processo passa o valor da variável a um e entra na região critica.Se a variável já for um o processo espera que esta tenha o valor zero.Desta forma a variável com valor zero significa que não esta nenhum processo na região critica e se for um significa que esta.
Mas esta ideia contem uma falha:suponhamos que um processo le o valor da variável e vê que esta a zero.Antes de esse processo alterar o valor da variável outro processo é escalonado,executa e altera o valor da variável para um.Quando o primeiro executar passa o valor para um também e dois processos estarão na região critica ao mesmo tempo.

3.2.2.3- Alternância Estrita:

```
while (TRUE) {                          while (TRUE) {
   while (turn != 0)   /* loop */ ;        while (turn != 1);  /* loop */ ;
   critical_region();                      critical_region();
   turn = 1;                               turn = 0;
   noncritical_region();                   noncritical_region();
}                                       }
(a)                                     (b)
```

Quando o processo 0 deixa a região critica passa a sua variável a 1 para autorizar o processo 1 a entrar na região critica.
Mas imaginemos que o processo 1 termina a sua actividade na região critica de uma forma tão rápida que ambos os processos estão na sua região não critica com a variável com o valor zero.Nesse momento o processo zero executa a sua actividade na região critica de uma forma tão rápida e passa a variável a 1.Nesse momento a variável esta a 1 e os 2 processos estão a executar na sua região não critica.De repente o processo 0 quer entrar na região critica mas a variável esta a 1 e ele não pode e também o processo 1 esta a executar na sua região não critica,Vai ter de esperar ate o processo 1 executar na região critica e passar a variável a zero o que pode fazer deste processo um processo muito lento.
Esta situação viola a condição de que nenhum processo deve ser bloqueado por outro na sua região não critica.

3.2.2.4- Solução de Peterson:

Combinando a ideia anterior com a ideia de bloquear variáveis e de variáveis de aviso (warnings) uma ideia de ter uma solução de software com exclusão mutua que não necessite de alternância estrita.
Depois peterson descobriu uma solução muito mais simples de forma a garantir exclusão mútua:

```
#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 */
}
```

Antes de usar as variáveis partilhadas (antes de entrar na região critica) cada processo chama a função enter_region com a sua própria variável (zero ou um) passada como parâmetro.Esta chamada vai fazer com que espere se necessário ate que seja seguro entrar.Depois chama a função leave_reagion para indicar que o seu trabalho esta feito e para permitir que outro processo entre se assim necessitar.

3.2.2.5- Sleep e wake up:

Forma de primitivas de comunicação entre processos que bloqueiam em vez de desperdiçar tempo de CPU quando não estão autorizados a entrar nas regiões criticas.Um dos mais simples é o par sleep e wakeup.Sleep é uma chamada ao sistema que cause o que faz a chamada bloqueie, isto é seja suspenso ate que outro processo o acorde.A chamada wakeup tem um parâmetro que é o processo a ser acordado.Alternativamente ambas as chamadas sleep e wakeup tem um parâmetro, um endereço de memoria usado para ligar os sleeps com os wakeups.

Algoritmo do produtor consumidor:

```
#define N 100       /* number of slots in the buffer */
int count = 0;      /* number of items in the buffer */

void producer (void)
{
    int item;

    while (TRUE) {                  /* repeat forever */
        item = produce_item();      /* generate next item */
        if (count == N) sleep();    /* if buffer is full, go to sleep
*/
        insert_item(item);          /* put item in buffer */
        count = count + 1;          /* increment count of items in
buffer */
        if (count == 1) wakeup(consumer);  /* was buffer empty? */
    }
}

void consumer(void)
{
    int item;

    while (TRUE) {                  /* repeat forever */
        if (count == 0) sleep();    /* if buffer is empty, got to
sleep */
        item = remove_item();       /* take item out of buffer */
        count = count - 1;          /* decrement count of items in
buffer */
        if (count == N - 1) wakeup(producer); /* was buffer full? */
        consume_item(item);         /* print item */
    }
}
```

3.2.3- Sincronização:

Em programas concorrentes não é possível determinar a ordem em que os eventos vão ocorrer. Alguma forma de sincronização entre processos é necessária. Sincronização entre processos é uma forma de coordenar as tarefas paralelas no tempo. Isso é feito especificando um ponto no programa onde duas ou mais tarefas devem estar para que se possa continuar a execução.

3.2.3.1- Sincronização com memoria partilhada:

O mecanismo mais usado é o de **semáforos.**:
Um semáforo S tem associado:

1- Um valor inteiro
2- Uma fila de processos bloqueados.
3- E duas operações: wait e signal
4-

Wait:

Se o valor de S > 0 decrementa o valor.
Caso contrario o processo é bloqueado na fila.

Signal:

Se há um processo na fila desbloqueia o primeiro processo.
Caso contrario incrementa o valor de S.

O semáforo é uma variável que guarda o número de wakeup's feitos. Caso seja zero nenhum foi feito, caso seja um número positivo temos o número de wakeup's feitos. Todo o semáforo tem de ser atómico de forma a garantir que quando começa uma operação num semáforo nenhum outro processo acede ao semáforo ate este concluir a sua operação. Se estivermos a trabalhar em múltiplos CPU's cada semáforo deve estar protegido por uma variável lock.

Problemas com o uso de semáforos:

1- Uso de primitivas não estruturadas o que origina uma difícil depuração do programa.
2- A distribuição correcta de waits e signals é do programador.
3- O uso de semáforos pode facilmente levar a uma situação de impasse o que leva a que se caia muito facilmente em deadlocks.

Exemplo de um deadlock:
Um conjunto de processos esta num deadlock quando cada processo no conjunto está á espera de um evento que só pode ser causado por outro processo nesse conjunto. Se todos os processos estão á espera então nenhum deles causara o evento e o conjunto fica bloqueado para sempre.

3.2.3.2- Sincronização sem memoria partilhada:

Na ausência de memória partilhada, o problema de resultados inconsistentes torna-se muito menor.
Valores compartilhados devem ser passados através de mensagens.
Como cada mensagem é totalmente processada antes de se tratar a próxima, os problemas de inconsistência não aparecem.
Inconsistências possíveis: processos diferentes podem receber mensagens em ordens diferentes...

----------Capitulo 4 – Gestão de memoria----------

Sistemas de gestão de memória podem ser divididos em duas classes:
As que movem processos para trás e para a frente entre a memoria principal e o disco durante a execução(swapping e paging) e as que não o fazem(monoprogramação).Estas ultimas são as mais simples.

4.2.1 – Monoprogramação:

A possibilidade mais simples na gestão de memória é de executar apenas um programa de cada vez partilhando a memória entre esse programa e o sistema operativo.

4.2.1.1: com partições fixas:

Á excepção dos sistemas embebidos a monoprogramação já dificilmente é usada.
Neste caso a memória é dividida até n partições (possivelmente de dimensões diferentes) podendo ser feita manualmente.
Quando uma tarefa chega pode ser posta na fila para a partição mais pequena mas com tamanho suficiente para essa tarefa.Como as partições são fixas o espaço que nessa partição não seja usado por essa tarefa é perdido.

4.2.1.2- Modelo de multiprogramação:

Quando a multiprogramação é usada a utilização do CPU pode ser melhorada.
Em media num processo em computação 20% do seu tempo este esta na memoria, com 5 processos na memoria em cada instante o CPU deve estar ocupado todo tempo.

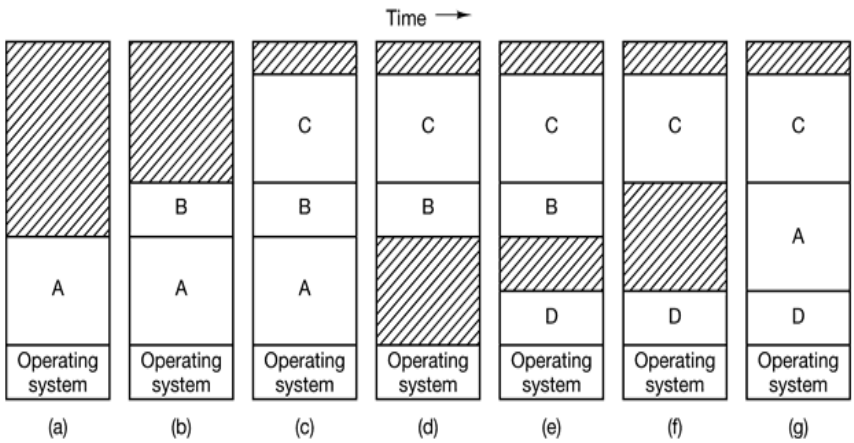
4.2.1.3-Recolocação e protecção:

Multiprogramação introduz dois problemas essenciais que devem ser resolvidos:
A recolocação e a protecção. Quando um programa é ligado(linked) o linker tem de saber em que endereço de memoria o programa vai iniciar.
Se por exemplo supormos que a primeira instrução é uma chamada a um procedimento no endereço 100.Se o programa for carregado na partição 1(no endereço 100K) essa instrução vai saltar para o endereço 100 que esta dentro do sistema operativo.O que é necessário é uma chamada para o endereço 100K + 100.
Se o programa for carregado na partição 2 então será 200K + 100 e por ai fora.Este problema é conhecido como um problema de recolocação.
Uma possível solução é de alterar as instruções quando o programa é carregado para memoria.Programas carregados para a partição 1 terão para cada endereço uma soma de 100K e por ai fora.
Mas a recolocação durante o carregamento do programa não resolve o problema de protecção.Um programa malicioso pode sempre construir uma nova instrução e saltar para ela.
A solução que a IBM escolheu para a protecção foi de dividir a memoria em blocos de 2KB com uma protecção de 4 bits para cada bloco.
Uma alternativa para os problemas de protecção e recolocação é de equipar a maquina com os registos especiais de hardware chamados de base e limite.

Quando um processo é escalonado o registo base é carregado com o endereço do inicio da sua partição e o registo limite é carregado com o comprimento da partição.Todos os endereços de memoria gerados depois tem o conteúdo do registo base somado a esse endereço antes de ser enviado para a memoria.
A desvantagem disto é a necessidade de se fazer uma adição e uma comparação em cada referencia á memoria.As comparações são rápidas mas as adições são lentas .

4.2.2-Swapping:

Com um sistema batch organizando a memoria em partições fixas é simples e eficaz.Cada tarefa é carregada numa partição quando chega ao inicio da fila e fica em memoria ate ter terminado.Enquanto podermos manter em memoria tarefas suficientes de forma a manter o CPU ocupado todo tempo não há razão para usar algo mais complicado.
Com sistemas de timesharing ou orientadas graficamente a situação é diferente. Por vezes não existe memoria principal suficiente para manter todos os processos activos no momento, então os processos em excesso devem ser mantidos no disco e trazidos para execução de uma forma dinâmica.
Duas abordagens á gestão de memoria podem ser usadas dependentes(em parte) do hardware disponível.
A estratégia mais simples é chamada de swapping consiste em trazer em cada processo na sua totalidade, executando-o por algum tempo e depois voltar a po-lo no disco.
A outra estratégia é chamada de memoria virtual que permite os programas executarem mesmo que só estejam em parte na memoria principal.
Agora vamos explicar a operação de swapping:
Inicialmente só temos um processo (chamemos de A) em memoria.Depois os processos B e C são criados ou swapped do disco.
A é swapped para fora do disco, depois D entra e B sai.Depois A volta ao disco.
Como A agora já tem uma localização diferente , o seu conteúdo de endereço deve ser realocado ou por software quando é swapped para o disco de novo ou muito provavelmente por hardware durante a execução do programa.



A principal diferença entre as partições fixas e as variáveis é o que o número, localização e tamanho da partição varia dinamicamente com a entrada e saída de processos no ultimo caso. A flexibilidade de não estar agarrado a um numero de partições que pode ser muito grande ou muito pequeno melhora a utilização da memoria mas complica a alocação e deslocação de memoria.

Quando o swapping cria muitos buracos na memoria é possível combina-los todos num único, movendo todos os processos de uma forma descendente ate onde for posssivel.Esta técnica e conhecida como compactação de memoria. Normalmente não é feita porque requer muito tempo de CPU.
Se os segmentos de dados de um processo poderem crescer ocorre um problema sempre que esse processo quer crescer.Se adjacentemente ao processo houver um buraco este poderá crescer para la.Por outro lado se tiver outro processo adjacente o processo em crescimento terá de ser mudado para um buraco que seja suficientemente grande para ele.Um ou mais processos podem ser "swapped" para fora de forma a arranjar espaço suficiente para este processo.Se ele não conseguir crescer ele terá de esperar ou ser terminado (killed).
Se é esperado que um processo cresça talvez seja boa ideia alocar um espaço extra.

4.2.2.1-Gestão de memoria com bitmaps:

Uma memoria atribuída dinamicamente o sistema operativo deve geri-la.
De uma forma geral há duas formas de gerir o uso de memória: bitmaps e free lists.
Com um bitmap a memoria é dividida em unidades de alocação, talvez tão pequenas como algumas palavras ou talvez tão grande como alguns kilobytes.
Correspondente a cada unidade de alocação esta um bit num bitmap que é zero se a unidade esta livre ou um se esta ocupada(ou vice versa).
O tamanho da unidade alocada é uma questão importante ao nível do design.
Quanto mais pequena for a unidade alocada maior é o bitmap.
Uma memoria de 32n bits ira usar n bits para o bitmap desta forma o bitmap ira usar 1/33 de memoria.Se a unidade de memoria alocada for maior o bitmap será mais pequeno mas uma quantidade apreciável de memoria será desperdiçada na ultima unidade do processo se o tamanho do processo não for múltiplo da unidade alocada.
Um bitmap providencia uma forma simples de controlar as palavras da memoria numa quantidade fixa de memoria o tamanho do bitmap apenas depende do tamanho da memoria e do tamanho da unidade alocada.O principal problema é que quando decide trazer um processo com tamanho de K unidades para memoria o gestor de memoria tem de procurar o bitmap para encontrar K bits consecutivos no map.Esta operação é lenta o que torna este argumento contra os bitmaps.

4.2.2.2-Gestão de memoria com Linked lists:

Esta forma mantém uma lista ligada com memoria alocada e segmentos de memoria livres onde um segmento pode ser um processo ou um buraco entre processos.
Quando os processos e os buracos são mantidos numa lista classificada por endereço, vários algoritmos podem ser usados para alocar memoria para um processo criado recentemente ou que foi "swapped" do disco.O primeiro é o algoritmo de "first fit", o gestor de memoria verifica a lista dos segmentos ate encontrar um buraco com tamanho suficiente.Depois disso o buraco é partido em dois pedaços, um para o processo e outro para a memoria que não será usada(com excepção de encontrar logo um com tamanho exactamente igual ao que queria).Este é um algoritmo rápido já que procura o menos possível.
Uma variante do anterior é o "next fit": funciona da mesma forma do anterior com excepção de que mantém a sua localização sempre que encontra um buraco que agrade.O anterior tem melhor performance.

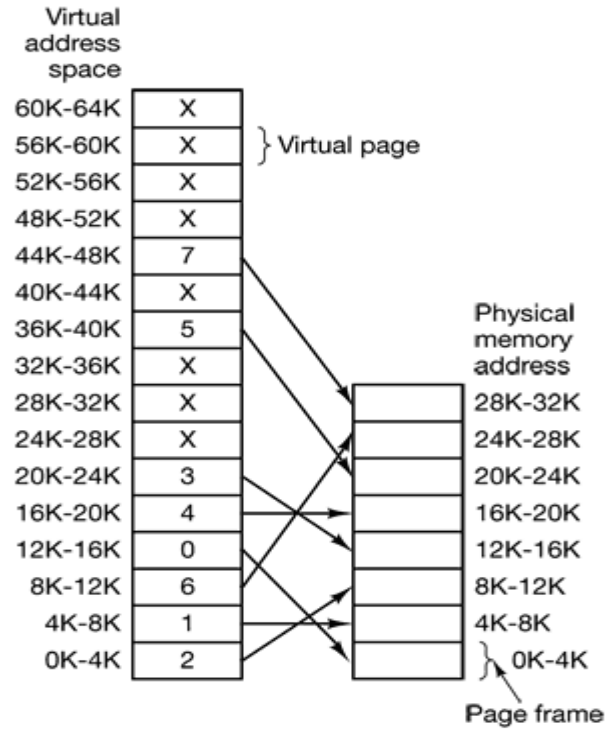
Outro algoritmo bem conhecido é o de “best fit”.Este procura toda a lista e escolhe o buraco com o menor tamanho (mas suficiente). Em vez de partir um buraco grande que poderá fazer falta mais tarde tenta encontrar um buraco que tenho um tamanho pêro do tamanho necessário.Este algoritmo é mais lento que o algoritmo “first fit” já que tem de procurar toda e lista sempre que é invocado.Surpreendentemente também resulta em mais memoria desperdiçada que o “first fit” ou o “next fit” já que tende a encher a memoria com buracos pequeninos e inutilizáveis.
Para contornar o problema de arranjar buracos com o tamanho de quase o necessário podem ser pensados os “worst fit” que consiste em escolher sempre o maior buraco disponível já que quando este se partir o que sobra tenha um tamanho suficientemente grande para que possa ser usado.Este algoritmo foi provado por simulação não ser uma boa ideia.
Outro algoritmo de alocação é o “quick fit” que mantém listas separadas para alguns dos tamanhos requeridos mais comuns.
Com este algoritmo encontrar um buraco com o tamanho desejado é extremamente rápido mas tem a desvantagem de todos os esquemas do tipo de seleccionar buraco pelo seu tamanho,principalmente quando um processo termina ou é “swapped” encontrar os seus vizinhos para ver se é possível uma fusão é caro. Se a fusão não é feita, a memória será fragmentada rapidamente num grande número de pequenos buracos nos quais não cabem processos.

4.3-Memória virtual:

Há muitos anos as pessoas viram-se confrontadas com programas que eram demasiado grandes para caber na memoria disponível.A solução na altura foi dividir os programas em pedaços chamados de sobreposições(overlays). A sobreposição zero executava primeiro, quando esta terminava chamava outra mas estes sistemas eram muito complexos, este serviço tinha de ser feito pelo programador alem de consumir muito tempo.
Depois alguém pensou em passar este trabalho para o computador, este método passou a ser conhecido como memoria virual.A ideia básica da memoria virtual é que o tamanho do+ programa, os dados, e a stack podem exceder a quantidade de memoria física disponível.O sistema operativo mantém as partes do programa que estão a ser usadas no momento na memoria principal e o resto no disco.

4.3.1- Paging:

Muitos dos sistemas de memoria virtual usam uma técnica chamada “Paging”.
Em qualquer computador, existe um conjunto de endereços de memória que os programas podem produzir. Quando um programa utiliza uma instrução como:
MOV REG 1000
O que esta instrução faz é copiar o conteúdo de memoria da posição 1000 para o registo.Endereços podem ser gerados usando indexação, registos base, registos de segmento ou de outras formas.Estes endereços gerados pelo programa são chamados de endereços virtuais e formam o espaço de endereçamento virtual.Em computadores sem memoria virtual, o endereço virtual é posto directamente no barramento de memoria.Quando a memoria virtual é usada os endereços virtuais não vão directamente para o barramento de3 memoria.Em vez disso para uma unidade de gestão de memoria MMU (Memory Managment Unit) que mapeia os endereços virtuais para endereços físicos da memória.

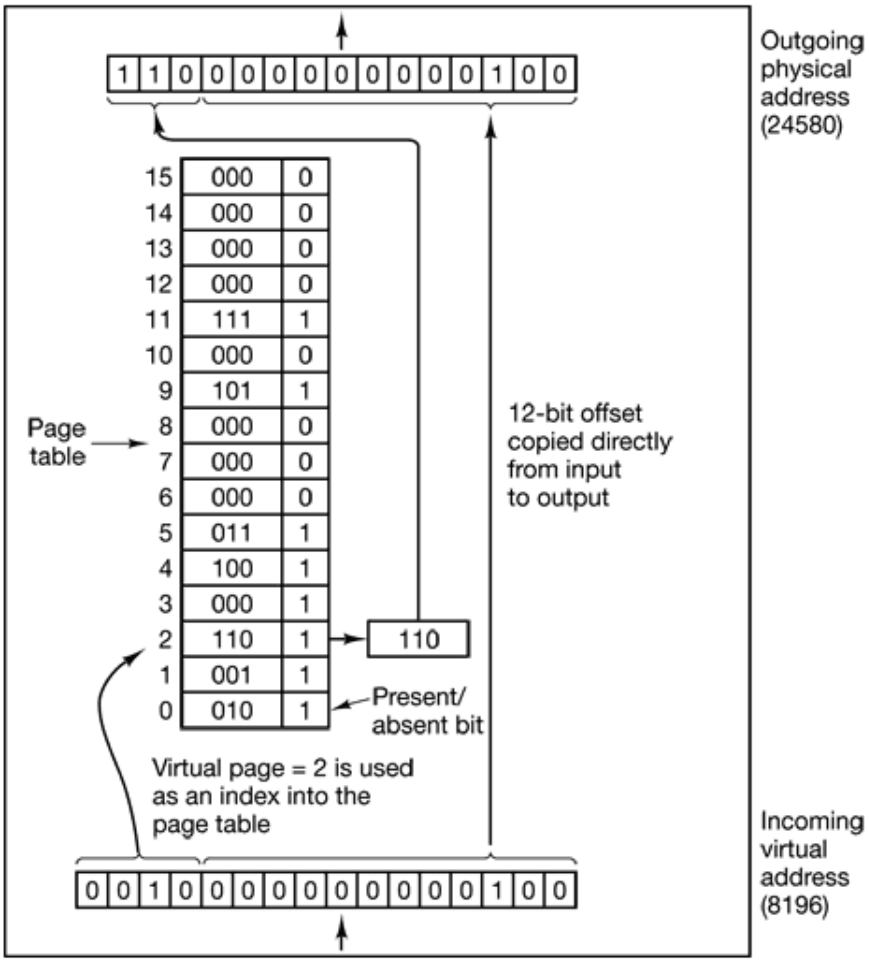


Neste exemplo acima quando um programa tenta aceder ao endereço 0 usando por exemplo a instrução: MOV REG,0 o endereço virtual é enviado para a MMU.a MMU verifica que este endereço cai no endereço virtual na frame 0(de 0 a 4095) o que na parte da memoria física corresponde á frame 2(de 8192 a 12287).Então

transforma o endereço para 8192 e põe este endereço no barramento.A memoria não sabe nada sobre a MMU e só vê um pedido de leitura ou escrita no endereço 8192 o que faz.Desta forma a MMU mapeou os endereços virtuais de 0 a 4095 para endereços físicos de memoria de 8192 a 12287.

4.3.2-Page tables:

O caso mais simples se mapear endereços virtuais para endereços físicos é o descrito anteriormente.
No caso de "page tables" o endereço virtual é dividido para um numero de pagina virtual(bits mais significativos) e um offset(bits menos significativos).



O numero da pagina virtual é usado como 1 índex para a pagina da tabela de forma a encontrar a entrada para essa pagina virtual. A partir da entrada de tabela de página, o número do quadro página (qualquer) é encontrado. O número do quadro página é anexado no fim do bit mais significativo do offset, substituindo o número da página virtual, para formar um endereço físico que pode ser enviado para a memória.
O objectivo da tabela de pagina é mapear paginas virtuais em tabelas de paginas.Matematicamente falando a tabela de pagina é uma função que tem o

numero de pagina virtual como argumento e o numero da tabela física como resultado.Usando o resultado desta função o campo da pagina virtual num endereço virtual pode ser substituído por o campo da tabela da pagina, formando assim um endereço de memoria físico.
Apesar desta descrição simples duas grandes questões devem ser enfrentadas:

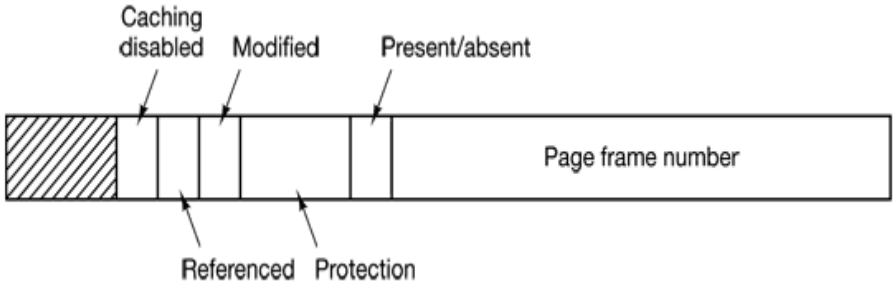
1- a tabela da pagina pode ser muito grande.
2- O mapeamento tem de ser rápido.

4.3.2.1-Multilevel Page tables:

De forma a contornar o problema do tamanho das paginas muitos computadores usam multilevel page table.
O segredo deste método é evitar manter sempre todas em paginas em memoria .Em particular as que não são necessárias não devem ser mantidas por perto.Imaginando que 1 processo necessita de 12Mbytes, os 4 Mbytes mais baixos de memoria para texto do programa,os 4M seguintes para dados e os 4M de cima para a stack.Entre o top, os dados e o fundo da stack esta um buraco gigante que não é usado .

4.3.2.2-estrutura de uma entrada na tabela da pagina:

O exacto layout de uma entrada é altamente dependente da maquina, mas a informação presente é aproximadamente a mesma de maquina para maquina.O tamanho varia de computador para computador mas um tamanho comum é 32 bits.O campo mais importante é o "Page frame number" já que o grande objectivo no mapeamento de pagina é este valor.a seguir a este valor temos o "Present/absent" bit.Se este tem valor 1 a entrada é valida e pode ser usada,se é 0 a pagina virtual á qual a entrada pertence não esta actualmente em memoria.aceder a uma entrada na tabela com este bit a 0 causa uma falha da pagina.



Os bits de protecção (protection) dizem que tipos de acesso são permitidos.Na sua forma mais simples este campo tem 1 bit com o valor 0 para acessos de leitura e escrita e 1 para acessos só de leitura.Num caso mais sofisticado o campo tem 3 bits para leitura,outro para escrita e outro para execução.

Os bits "modified" e "reference" mantém o registo sobre o uso da pagina.Quando uma pagina é escrita o hardware automaticamente define o "modified" bit.O valor deste bit é utilizado se o sistema operativo decide reclamar a tabela da pagina.Se a pagina foi modificada deve ser escrita de volta no disco, se não foi pode ser

abandonada já que a copia no disco é valida.este bit é muitas vezes chamado de "dirty bit" já que ele reflecte o estado da pagina.
O bit "reference" é definido sempre que a pagina é referenciada ou para leitura ou para escrita.O seu valor ajuda o sistema operativo a escolher uma pagina a evitar quando ocorre uma falha.As paginas que não estão a ser usadas são melhores candidatas que as que estão.

O bit "cashing desable" permite que a cache seja desabilitada para essa pagina.É importante para paginas que mapeiam registos dos dispositivos em vez de memoria.

4.3.2.3TLBs— Translation Lookaside Buffers:

Em muitos esquemas de paginação as tabelas da pagina são mantidas em memoria devido ao seu grande tamanho. Este design tem potencialmente um enorme impacto na performance.Considerando por exemplo uma instrução que copie um registo para outro, na ausência de paginação esta instrução faz apenas uma referencia á memoria.Com paginação referencias adicionais á memoria irão ser necessárias para aceder á tabela da pagina.Como a velocidade de execução é normalmente limitada pela taxa o CPU pode receber instruções e dados fora da memoria tendo que fazer duas referencias por quadro de pagina de referencia de memoria reduzindo a performance em dois terços. A solução para isto é baseada na observação de que todos os programas tendem a fazer um grande numero de referencias para um numero pequeno de paginas.a solução passa por equipar os computadores com um pequeno dispositivo de hardware para mapear endereços virtuais em endereços físicos sem passa pela tabela de pagina.O dispositivo é chamado de **Translation Lookaside Buffer(TLB)** .Normalmente esta dentro do MMU e consiste num pequeno numero de entradas raramente não mais de 64.Cada entrada contem a informação de uma pagina, incluindo o numero virtual da pagina, um "modified" bit, o "protection" bit e a tabela física da pagina onde a pagina esta alocada.

Funcionamento do TTL:

Quando um endereço virtual é presente ao MMU o hardware primeiro verifica se o numero virtual da pagina esta presente no TLB, comparando-o de uma forma simultânea com todas as entradas.Se o encontra e não viola os bits de protecção a tabela da pagina é "pegada" directamente do TLB sem ir á tabela da pagina.Se ele estiver no TLB mas a instrução for de escrita numa pagina só de leitura um erro de protecção é gerado.

O caso mais interessante acontece quando o número de pagina virtual não esta no TLB.O MMU detecta essa falha e faz uma normal procura na pagina.

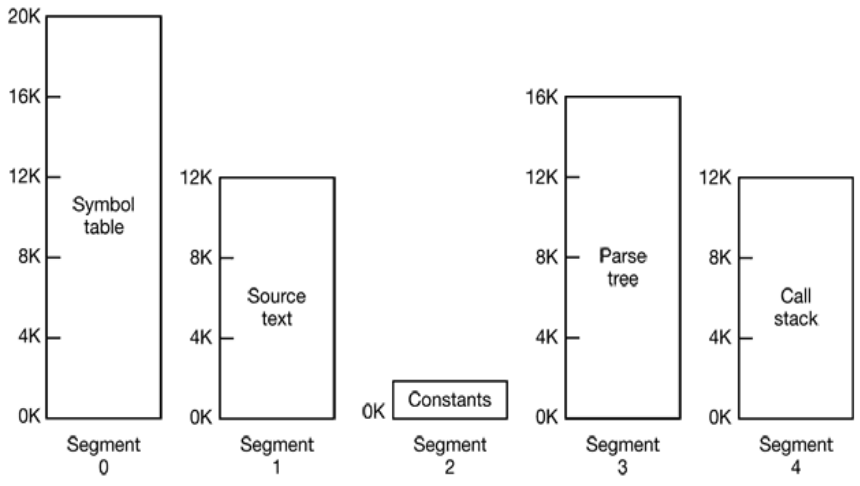
4.3.2.4 Inverted Page Tables:

Nas paginas não invertidas se o espaço de endereçamento consiste em 2^32 bytes,com 4096 bytes por pagina então mais de um milhão de entradas de

tabela de pagina serão necessárias.Uma solução para isto é uma tabela de pagina invertida que ganha muito espaço.

4.4- Segmentação:

A memória virtual discutida ate agora é uni dimensional já que os endereços virtuais vão de 0 ate um numero n de uma forma sequencial.Para muitas situações ter dois ou mais endereços virtuais separados pode ser muito melhor que ter só um.Imaginemos um programa que tem um numero excepcionalmente grande de variáveis mas um numero normal de tudo o resto, o bloco de espaço de endereçamento alocado para a tabela de símbolos pode encher mas poderá existir muito espaço nas outras tabelas.nesta situação(ou noutras) o que é mesmo necessário é uma forma de livrar o programador de ter de gerir a expansão e as contracções das tabelas.Uma solução simples e geral é providenciar a maquina com muitos espaços de endereçamento completamente independentes chamados de **segmentos.** Cada segmento consiste numa sequencia linear de endereços de 0 ate um numero máximo.O comprimento de cada segmento pode ser de 0 ate esse máximo permitido.Segmentos podem, e normalmente tem tamanhos diferentes, e acima de tudo o tamanho dos segmentos pode variar durante a execução.O tamanho do segmento pode ser aumentado quando ocorre um push e decrementado quando ocorre um pop.Como cada segmento constitui um espaço de endereçamento diferente, cada um pode crescer ou decrescer de uma forma independente sem se afectarem entre si.Se a stack de um certo segmento precisa de mais espaço de endereçamento, esta pode tê-lo já que nesse espaço de endereçamento não há mais nada.Claro que um segmento pode atingir o máximo mas normalmente os segmentos são muito grandes e assim esta ocorrência é muito rara.

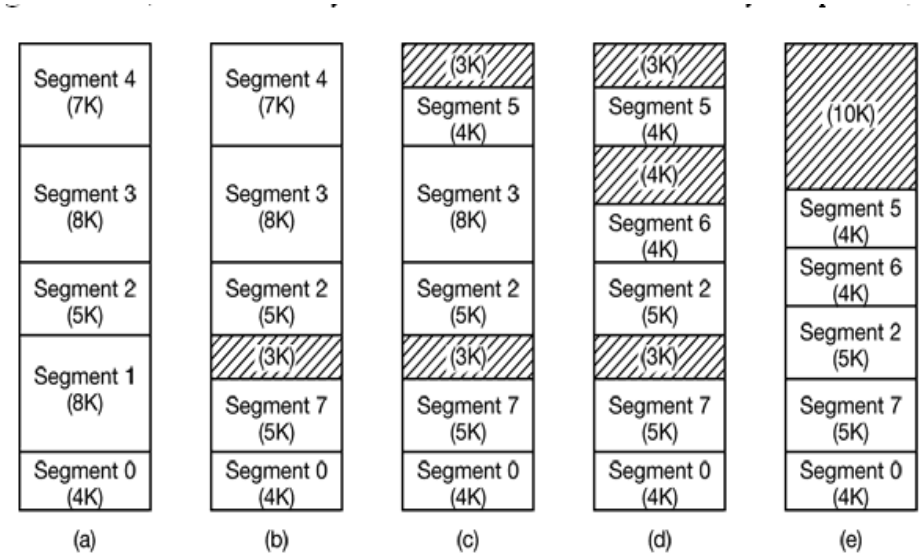


Um segmento é uma entidade lógica, e o programador sabe disso e usa-o dessa forma.um segmento pode conter um procedimento ou um array, ou uma stack ou uma colecção de variáveis escalares mas normalmente não contem uma mistura de tipos diferentes.Alem da vantagem de simplificar a utilização de estruturas de dados que crescem e diminuem ainda tem a seguinte vantagem:
Se cada procedimento ocupa um segmento separado como endereço de inicio o zero a linkagem dos procedimentos compilados em separado é altamente simplificada já

que todos os procedimentos que constituem um programa foram compilados e linkados.
Se o procedimento no segmento n é muitas vezes modificado e recompilado nenhum outro procedimento precisa de ser alterado mesmo que a nova versão seja maior que a actual.Numa memorial unidimensional os procedimentos são agrupados uns ao pé dos outros sem nenhum espaço de endereçamento entre eles, por consequência alterar o tamanho de um procedimento pode afectar o endereço de inicio de outro, isto requer modificar alterar os procedimentos todos.Se um programa contem muitos procedimentos isto pode ter um custo muito elevado.
A segmentação também facilita a partilha de procedimentos os dados entre vários processos.Um exemplo comum é a livraria partilhada .Como cada segmento forma uma entidade lógica e tendo tanto o programador como um procedimento ou um array ou uma stack tem conhecimento disso , diferentes segmentos podem ter diferentes tipos de protecções.devemos tentar perceber que a protecção faz sentido numa memoria segmentada mas numa pagina unidimensional de memoria não o faz.Numa memoria segmentada o utilizador tem conhecimento do que é cada segmento.Normalmente um segmento não contem por exemplo um procedimento e uma stack mas sim um ou outro.Como um segmento só tem objectos de um tipo ele pode ter protecção apropriada para esse tipo.

4.4.1- Implementação de segmentação pura:

Aqui a implementação difere da paginação numa forma essencial: as paginas são de tamanho fixo e os segmentos não.Imaginemos a memoria física inicialmente com 5 segmentos.Imaginemos que o segmento 1 é expulso e o segmento 7 que é mais pequeno é posto no seu lugar.Entre o segmento 7 e o segmento 2 esta uma área que não é usada(um buraco).Depois segmento 4 é substituído pelo 5 e o 3 é substituído pelo 6.Depois do sistema executar por um determinado tempo a memoria estará dividida num numero de pedaços alguns com segmentos outros com buracos.Este fenómeno chamado de "checkerboarding" ou "external fragmentation" desperdiça memoria em buracos.



4.4.1- Segmentação com paginação:

Se um segmento for grande pode ser inconveniente ou mesmo impossível mantê-lo por completo na memoria principal.Aqui vem a ideia de os paginar, de forma a que apenas as paginas necessárias no momento estejam por perto.A isto chama-se de **"Multics".**

Cada programa "Multics" tem uma tabela de segmentos com um descritor por segmento.Como potencialmente existe mais de um quarto de milhão de entradas na tabela a tabela é por si só um segmento e esta paginada.Um descritor de segmento contem uma indicação se o segmento esta na memoria principal ou não.Se alguma parte do segmento estiver na memoria principal considera-se que este esta em memoria principal e a sua tabela de paginação estará em memoria e o seu descritor contem um apontador de 18 bits para a sua tabela de paginação.O descritor contem também o tamanho do segmento,os bits de protecção e mais alguns itens.O endereço do segmento em memoria secundaria não esta no seu descritor mas noutra tabela usada pelo tratador de falhas.Um endereço em "Multics" consiste em duas partes:o segmento e o endereço dentro do segmento.

Quando ocorre uma referencia á memoria o seguinte algoritmo ocorre:

O numero do segmento é usado para encontrar o seu descritor.

È feita uma verificação para verificar se a sua tabela de paginação esta em memoria.Se esta é localizada senão ocorre uma falha de segmento.Se há uma violação da protecção ocorre um erro.

A entrada na tabela referente á pagina virtual requerida é examinada.Se não esta em memoria ocorre um erro caso contrario o endereço da memoria principal do inicio dessa pagina é retirado da entrada na tabela da pagina.

Adiciona-se um offset á origem da pagina para dar o endereço da memoria principal onde a "palavra" está.

A operação ler ou guardar é executada.

----------Capitulo 5 – Gestão de periféricos----------

Uma das principais funções de um sistema operativo é controlar todos os dispositivos de input e output do computador.Este deve emitir comandos para os dispositivos, apanhar interrupções e tratar os erros.Deve também providenciar uma interface entre os dispositivos o resto do sistema simples e fácil de usar.

5.1- Gestão por hardware:

As pessoas podem olhar para o hardware de uma forma diferente umas das outras.Os engenheiros electrónicos olham para os chips,fios,alimentações,motores e outros componentes físicos do hardware.Os programadores olham para a interface apresentada ao software - os comandos que o hardware aceita,as funções que exerce e os erros que podem ser reportados. Aqui só falamos do hardware aos olhos do programador.

5.1.1- Dispositivos de I/O:

Dispositivos de I/O podem ser mais ou menos divididos em 2 categorias: dispositivos de bloco e dispositivos de caracteres.Um dispositivo de blocos é um que guarda informação em blocos de tamanho fixo cada um com o seu endereço.A propriedade essencial de um dispositivo de blocos é que é possível ler ou escrever em cada bloco de forma independente.Os discos são os mais comuns. A fronteira entre dispositivos que são endereçados por blocos e os que não são não esta bem definida. Todos concordam que 1 disco é um bloco dispositivo endereçável. Considerando uma "tape drive" usada para fazer backups ao disco. As "tapes" contem uma sequencia de blocos.Se á "tape" é dado um comando para ler o bloco n, pode sempre rebobinar a cassete ou ir para a frente ate chegar ao bloco n.Esta operação é análogo á operação de procura num disco com a excepção de levar muito mais tempo.Tambem pode ou não ser possível reescrever um bloco no meio da "tape".

O outro tipo de dispositivo de I/O é o dispositivo de caracteres.Este dispositivo entrega ou aceita um fluxo (stream) de caracteres sem olhar para nenhum bloco de estruturas.Nao é endereçável e não tem nenhuma operação de procura. Impressoras, interfaces de network e muitos outros dispositivos podem ser vistos como dispositivos de caracteres.
No entanto existem dispositivos que não cabem em nenhum destes: por exemplo os "clocks" não são endereçáveis por blocos mas também não geram ou aceitam fluxos de caracteres.Mesmo assim estes dois tipos de dispositivos geralmente são suficientes

5.1.2-Controladores de Dispositivos:

As unidades de I/O são tipicamente um componente mecânico e um componente electrónico.Normalmente é possível separar as duas parcelas para providenciar um design mais modular e geral.O componente electrónico é chamado de controlador de dispositivos ou adaptador.Num computador pessoal normalmente é um simples cartão inserido num slot.O componente mecânico é o dispositivo em si.O cartão normalmente tem um conector que inserindo um cabo neste podemos liga-lo ao dispositivo.Muitos cartões podem controlar ate 8 dispositivos idênticos.A interface entre o cartão e o interface é standard e oficial (ANSI, IEEE ou ISSO). Esta interface é muitas vezes de baixo nível.
A função do cartão é converter um fluxo de serie de bits num bloco de bytes e proceder a uma correcção de erro se necessário.O bloco de bytes é primeiramente montado bit a bit num buffer dentro do cartão.Depois verifica o seu "checksum" e se declara que o bloco esta livre de erros este pode ser copiado para a memoria principal.

5.1.3-Memory-Mapped I/O:

Cada controlador (cartão) tem alguns registos usados para comunicar com o CPU.Escrevendo nestes registos o sistema operativo pode comandar o dispositivo para enviar ou receber dados, para se ligar ou desligar ou fazer uma tarefa qualquer.Lendo destes registos o sistema operativo pode saber em que estado o dispositivo esta, se esta preparado para aceitar outro comando e por ai fora. Alguns dispositivos tem também um buffer de dados onde o sistema operativo pode ler e escrever.
Uma segunda aproximação foi introduzida que consiste em mapear todos os registos de controlo para o espaço de memoria.Cada registo tem um único endereço de memoria.este modelo é chamado de "memory-mapped I/O".Normalmente são atribuídos endereços que estão no topo do espaço de endereçamento. Como funciona este novo modelo? Quando o CPU quer ler uma palavra de memoria ou da porta do I/O ele põe o endereço que precisa no barramento de endereços e depois põe um sinal de "read" no barramento de controlo.Um segundo sinal é usado se precisa do I/o ou da memoria.Se for memoria esta responde ao pedido senão o dispositivo de I/O responde.Os dois modelos tem distintos pontos fortes e fracos.

5.1.4 Direct Memory Access (DMA)

Não interessa se o CPU tem ou não “memory-mapped I/O”, ele necessita de saber o endereço dos controladores de dispositivos para trocar dados com eles.CPU pode requerer um byte de dados de cada vez de um controlador de I/O mas desperdiça tempo de CPU.

Este novo esquema (DMA) é muitas vezes utilizado.O sistema operativo só pode utilizar DMA se o hardware tiver um controlador de DMA. Independentemente da sua localização física o controlador DMA tem acesso ao barramento do sistema de forma independente do CPU.Ele contem vários registos que podem ser lidos e escritos pelo CPU.Estes incluem um registo de acesso á memoria, um registo contador de bytes, e um ou mais registos de controlo. Para explicar como funciona o DMA vamos primeiro ver como uma leitura do disco ocorre sem DMA.Primeiro o controlador le o bloco da drive bit a bit ate todo o bloco estar no buffer interno do controlador.A seguir executa o “checksum” para verificar se não ocorreram erros.Depois o controlador causa uma interrupção.Quando o sistema operativo começa a executar ele pode ler um byte ou uma palavra de cada vez do bloco do buffer do controlador (executa um ciclo para ler tudo). Em cada iteração le do controlador e põe na memoria principal.

Quando usamos DMA o procedimento é diferente.Primeiro o CPU programa o controlador de DMA definindo os seus registos de forma a saber o que transferir e para onde.Tambem executa um comando para o controlador do disco dizendo-lhe para ler dados do disco para o seu buffer interno e verifica também o “checksum”.Quando dados validos estiverem no disco do buffer do controlador DMA pode começar.

O controlador DMA inicia a transferência através de um pedido de leitura do barramento para o controlador do disco (passo 2). Este pedido de leitura parece outro qualquer e o controlador do disco não sabe nem quer saber se ele veio do CPU ou do controlador DMA.Normalmente o endereço de memoria a escrever esta no barramento de endereços assim quando o controlador do disco faz um “fetch” á próxima palavra do seu buffer interno ele sabe onde escrever.A escrita em memoria é mais um ciclo standard do barramento (step 3).Quando a escrita esta completa o controlador do disco envia um sinal “acknowledgment” para o controlador do disco bem como para o barramento (passo 4). O controlador DMA agora incrementa o endereço de memória a usar e decrementa o contador de bytes.Se o contador de bytes ainda for maior que zero passos de 2 a 4 voltam a ser repetidos ate este ser zero. Quando chegar a zero o controlador DMA interrompe o CPU e fá-lo saber que a transferência esta completa.Quando o sistema operativo começa não tem de copiar o bloco do disco para a memoria, já lá está.

Os controladores DMA variam bastante consoante a sua sofisticação.Os mais simples tratam uma transferência de cada vez tal como descrito acima.Alguns mais complexos podem ser programados para tratar múltiplas transferências ao mesmo tempo.Estes controladores tem vários conjuntos de registos internos,um por cada canal.O CPU começa por carregar cada conjunto de registos com os parâmetros relevantes para a sua transferência.Cada transferência deve diferentes dispositivos controlo.Depois de transferir cada palavra(executando as funções descritas anteriormente nos step’s 2 ,3 e 4) o controlador DMA decide que dispositivo vai servir de seguida.

Muitos barramentos podem trabalhar em 2 modos: o modo de uma palavra de cada vez ou no modo de bloco.Alguns controladores DMA também podem trabalhar desta forma.No primeiro modo a forma de operar esta descrita

acima: o controlador pede para transferir uma palavra e transfere, mas aqui se o CPU também necessita do barramento tem de esperar.Este mecanismo chama-se roubo de ciclo(“cycle stealing”) porque o controlador de dispositivos infiltra-se e rouba ocasionalmente um ciclo de barramento ao CPU atrasando-o. No modo de bloco o controlador de DMA diz ao dispositivo para adquirir o barramento, executa uma serie de transferências e depois liberta o barramento. Este modo de operação é chamado de “burst mode”.É mais eficiente do que o “cycle stealing” já que adquirir o barramento custa tempo mas múltiplas palavras podem ser transferidas pelo preço de uma aquisição de barramento.O aspecto negativo deste modo é que pode bloquear o CPU e outros dispositivos por um período substancial de tempo se um grande “burst” estiver a ser transferido.Neste modo o controlador DMA diz ao controlador de dispositivos para transferir os dados directamente para a memoria principal.
Um modo alternativo usado pelo controlador DMA é por o controlador de dispositivos a enviar a palavra para o controlador DMA, o que executa um segundo pedido de aquisição do barramento para escrever a palavra onde ela tenha de ser escrita, este esquema requer mais um ciclo de barramento por palavra transferida mas é mais flexível já que pode executar copias de dispositivo para dispositivo e mesmo de memoria para memoria.
A maior parte dos controladores DMA usam endereços físicos de memoria para as suas transferências o que obriga o sistema operativo a converter memoria virtual num endereço físico e a escrever este endereço físico no registos de endereços do controlador DMA.Um esquema alternativo usado em poucos controladores DMA é escrever o endereço de memoria virtual no controlador. Depois o DMA precisa de usar o MMU para passar o endereço virtual para endereço físico. Num todos os computadores usam DMA. O argumento é que o CPU principal é a maior parte das vezes mais rápido que o controlador DMA e pode executar a função mais rápido. Se não houver outra tarefa a fazer então o CPU vai ter de esperar pelo controlador DMA.Outra razão é monetária:sem DMA poupa-se.

5.2- Gestão por software:

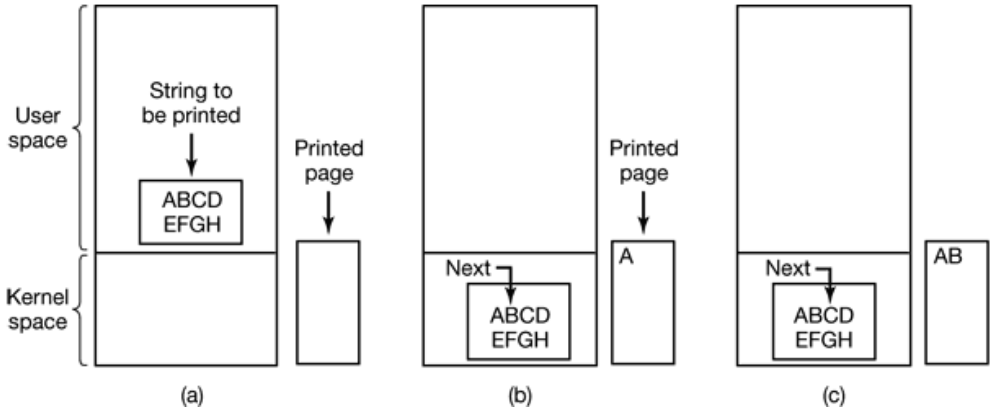
5.2.1-Metas do software de I/O:

Um conceito chave na concepção de software de I/O é a independência do dispositivo. Isto significa que deve ser possível escrever programas que possam aceder a qualquer dispositivo de I/O sem tem de especificar o dispositivo antes. Quase relacionado está o objectivo de uma nomenclatura uniforme. O nome de um ficheiro ou de um dispositivo deve ser uma string ou um inteiro e não depender de nenhuma forma do dispositivo.
Outra questão importante no software de I/O é o tratamento de erros. Geralmente os erros devem ser tratados quanto mais perto do hardware possível. Se o controlador descobre um erro ele deve tentar corrigi-lo. Se não conseguir então o driver do dispositivo deve tratar o erro.
Ainda outra questão é transferências síncronas(bloqueadoras) versus assíncronas(interrupt-driven). A maior parte do I/O físico é assíncrono – o CPU inicia a transferência e vai fazer outra coisa qualquer ate chegar a interrupção. Os programas para o utilizador são muito mais fáceis de escrever se as operações de I/O forem bloqueadoras – após uma chamada de leitura ao

sistema o programa é automaticamente suspenso até os dados estarem disponíveis no buffer. È tarefa do sistema operativo fazer com que as operações do tipo "interrupt-driven" pareçam do tipo bloqueadoras aos programas do utilizador. Outra questão para o software de I/O é o buffering. Muitas vezes os dados que vem de um dispositivo não podem ser armazenados directamente no seu destino. Dessa forma os dados devem ser postos num buffer de saída. Buffereing envolve um numero considerável de copias e muitas vezes tem um grande impacto na performance do I/O.
O ultimo conceito é o de serviços partilhados versus serviços dedicados. Muitos dispositivos de I/O como por exemplo os discos podem ser usados por muitos utilizadores ao mesmo tempo. Nenhum problema é causado por múltiplos utilizadores terem ficheiros abertos no mesmo disco ao mesmo tempo. Mas dispositivos como as "tape drives" tem de ser dedicadas a um único utilizador ate este terminar e só depois outro é que a pode ter. O uso destes sistemas dedicados introduz um numero variado de problemas como deadlocks. Mais uma vez os sistemas operativos tem de ser capazes de utilizar os dois sistemas(partilhados e dedicados) de uma forma que evite problemas.

5.2.1.1-I/O Programado:

Há 3 maneiras diferentes de o I/O ser executado. Nesta secção olhamos para uma delas: **Programmed I/O**. Nas duas próximas secções falamos das outras duas. A forma mais simples de I/O é ter o CPU a fazer o trabalho todo. Este método é chamado de I/O programado.



O utilizador adquire a impressora para escrita fazendo uma chamada ao sistema de abertura da mesma. Se a impressora estiver ocupada este processo ou falha

enviando um erro ou bloqueia ate que a impressora fique livre – depende do sistema operativo ou dos parâmetros com que a chamada ao sistema foi feita. Depois de ter a impressora o processo faz uma chamada ao sistema dizendo ao sistema operativo para imprimir a string na impressora. Depois o sistema operativo normalmente copia o buffer que contem a string para um array no espaço do kernel que vamos chamar p onde é mais facilmente acedido. Depois verifica se a impressora esta disponível, se não estiver espera. Logo que esta esteja disponível o sistema operativo copia o primeiro caracter para o registo de dados da impressora. Esta acção activa a impressora. Depois de copiar o primeiro caracter o sistema operativo verifica se a impressora esta pronta para receber outro. Geralmente a impressora tem outro registo que indica o seu estado, sendo que o acto de escrever um caracter põe esse registo de estado como "not availble" , isto é a impressora esta ocupada sendo que o sistema operativo tem de esperar que este registo de estado altere para "ready" . Faz sempre isto ate que a string seja imprimida e depois o controlo passa para outro processo. Isto é a base do I/O programado. O I/O programado é bastante simples mas tem a desvantagem de ter o CPU todo tempo até que o I/O esteja feito. Em sistemas em que o CPU tem muitas coisas para fazer é necessário outro método de I/O .

5.2.1.2-Interrupt-Driven I/O:

Neste caso a impressão não faz um buffer de caracteres mas sim faz a impressão de cada um á medida que chega. Imaginemos uma impressora que imprime 100 caracteres por segundo então cada caracter leva 10mseg a ser imprimido. Isto significa que depois de cada caracter ser escrito no registo de dados da impressora vai entrar num ciclo de 10msec á espera de ser autorizado a por lá outro caracter. Este tempo é mais do que suficiente para fazer correr outro processo mas vai ser desperdiçado.
A maneira de autorizar o CPU a fazer outra coisa enquanto espera é utilizando interrupções. Quando há uma chamada ao sistema para imprimir a string é construída, o buffer é copiado para o kernel como na situação anterior e o primeiro caracter é copiado para a impressora logo que esta o aceite. Nesse momento o CPU chama o escalonador e põe outro processo a executar. O processo que pediu a impressão da string é bloqueado ate que toda a string seja imprimida. Quando a impressora imprimiu o caracter e esta pronta para imprimir outro gera uma interrupção. Esta interrupção para o processo em execução guardando o seu estado. Então o procedimento de interrupção de serviço da impressora executa. Se não houver mais caracteres para imprimir o tratador de interrupções desbloqueia o utilizador. Se não põe outro caracter no output e retorna ao processo que estava a executar antes da interrupção que continua a executar no estado onde foi interrompido.

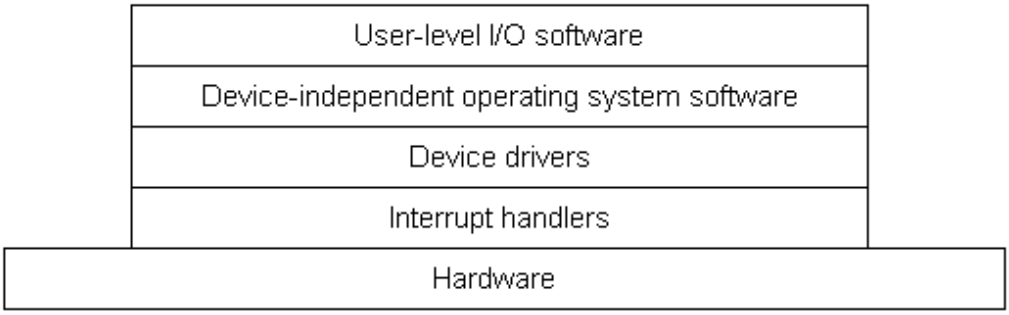
5.2.1.3-I/O usando DMA:

Uma obvia desvantagem do modelo anterior é que é gerada uma interrupção a cada caracter. Como cada interrupção consome tempo esse esquema desperdiça tempo de CPU. Uma solução é usar DMA. A ideia é deixar o controlador de DMA enviar os caracteres para a impressora, um de cada vez sem importunar o CPU. Na sua essência DMA é I/O programado, mas com o controlador DMA a fazer o trabalho todo e não o CPU. O ganho de DMA é reduzir o

numero de interrupções. Se há muitos caracteres então tínhamos muitas interrupções e aqui temos um ganho muito grande. Por outro lado normalmente o controlador DMA é mais lento que o CPU o que leva a que se o DMA não for capaz de alimentar o dispositivo á sua velocidade máxima, ou o CPU não tiver mais nada para fazer os outros dois modelos podem ser melhores.

5.2.2-Camadas de I/O software:

Tipicamente o I/O software esta organizado em 4 camadas, cada camada tem uma função bem definida a exercer e um interface bem definido com as outras camadas. As funcionalidades e as interfaces diferem de sistema para sistema.



5.2.2.1-Interrupt handlers:

Enquanto o I/O programado as vezes é útil, interrupções são desagradáveis mas não podem ser evitadas. Devem ser bem escondidas no sistema operativo de forma a que só uma pequena parte do sistema operativo tenha conhecimento sobre as mesmas . A melhor maneira de as esconder é ter o driver a iniciar uma operação de bloqueio de I/O até o I/O estar completo e a interrupção ocorra. Quando uma interrupção ocorre o procedimento de interrupções faz o que tem a fazer para tratar a interrupção. Depois pode desbloquear o driver que a iniciou.

Os passos que possam ocorrer podem ser em ordens diferentes de maquina para maquina.

Guardar qualquer registo que ainda não foi guardado pelo hardware de interrupções.
Estabelecer um contexto para o serviço do procedimento de interrupção.
Estabelecer a stack para o serviço do procedimento de interrupção.
Dar conhecimento ao controlador de interrupções.
Copiar os registos do sitio onde estavam guardados para a tabela do processo.
Executar o serviço do procedimento de interrupção.
Escolher o processo que executa de seguida.
Estabelecer o contexto do MMU para o processo a executar de seguida.
Carregar os registos do novo processo.
Executar o novo processo.

Como pode ser visto o processo de uma interrupção esta longe de ser trivial. Executa um numero considerável de instruções MMU.

5.2.2.2-Device drivers:

Cada dispositivo de I/O necessita de um código especifico de dispositivo para que o possa controlar. Este código é o chamado "device driver" , é normalmente dado pelo produtor do dispositivo e entregue com o dispositivo. Normalmente os produtores fornecem drivers para todos os sistemas operativos. Cada "device driver" normalmente trata um tipo de dispositivo. De forma a aceder ao hardware do dispositivo isto é os registos do controlador o "device driver" normalmente tem de fazer parte do kernel do sistema operativo.
Um "device driver" tem varias funções. Uma é aceitar pedidos abstractos de escrita e leitura do software independente do dispositivo. O driver tem também de inicializar o dispositivo se necessário. Um driver típico começa por verificar se os parâmetros de entrada são validos, caso não sejam retorna um erro. Se forem passar do termo abstracto para o concreto pode ser necessário. De seguida o driver pode verificar se o dispositivo esta em uso. Se sim o pedido passa para uma fila para ser processada mais tarde. Se o dispositivo esta livre verifica o status do hardware para ver se pode satisfazer já o pedido. Pode ter de por exemplo ligar o dispositivo. Depois disso o controlo do dispositivo pode começar. Controlar o dispositivo significa enviar uma serie de comandos para ele. Logo que o driver saiba que comando enviar começa a escreve-lo no registo de controlo do dispositivo..Depois verifica se o controlador aceitou o comando e esta pronto a aceitar o próximo. Faz isto ate enviar os comandos todos. Depois de enviar os comandos em muitos casos o device driver espera ate o controlador fazer alguma coisa com eles, bloqueando-se á espera de uma interrupção para o desbloquear. Noutros casos a operação termina sem atrasos e ele não tem de se bloquear. De qualquer das formas o driver bloqueado é sempre acordado pela interrupção. Depois de concluída o driver procura por possíveis erros Se não ocorreram erros ele faz outra função se o tiver de fazer. Se ocorreram manda informação de erros.

5.2.2.3-Device – Independent I/O software:

Algumas partes do software são independentes do dispositivo. O limite exacto entre os controladores e o software independente do dispositivo é o sistema dependente porque algumas funções que podiam ser feitas num software independente do dispositivo podem agora ser feitas em drivers por razoes acima de tudo de eficiência. Funções tipicamente feitas em software independente do dispositivo:

Uma interface uniforme para os device drivers.
Buffering.
Reportação de erros.
Alocar e libertar dispositivos dedicados
Providenciar um bloco para o software independente do dispositivo.

5.2.2.4-User-Space I/O software:

Embora a maior parte do software de I/O esteja no sistema operativo uma pequena porção deste consiste em livrarias linkadas conjuntamente com programas do utilizador .Chamadas ao sistema incluindo as de I/O são normalmente feitas em procedimentos das livrarias. Mas nem todas o são, uma categoria

importante é o sistema de “Spooling” que é uma forma de lidar com dispositivos dedicados de I/O num sistema de multiprogramação. Se considerarmos por exemplo uma impressora se pensarmos que um processo “abriu” a impressora e não faz nada lá durante horas, nenhum processo podia utilizar essa impressora durante esse tempo. Em vez disso o que é feito é criar um procedimento especial chamado “deamon” e uma directoria especial chamada de “spooling directory” . É tarefa do “deamon” que é o único processo com permissão para usar o ficheiro especial da impressora de imprimir os ficheiros na directoria. Protegendo este ficheiro especial de uso directo por parte dos utilizadores o problema anterior fica resolvido. O “spooling” é usado em varias situações.

5.3- Discos:

5.3.1-Disk Hardware:

Discos podem ser de uma variedade de tipos sendo que os mais comuns são os discos magnéticos (discos duros e disquetes). São caracterizados pelo facto que lêem e escrevem com a mesma velocidade o que os torna ideais para memória secundária. Para a distribuição de programas, dados, musica, etc outros tipos de discos (CD’s, DVD’s, etc) também são importantes. Vamos estudar as características de hardware de alguns deles.

5.3.1.1- Discos magneticos:

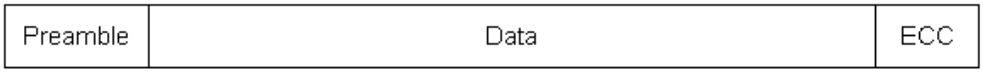
Estes discos estão organizados em cilindros cada um contendo tantas faixas quantas cabeças verticais. As faixas estão divididas em sectores. È possível ao controlador procurar em mais que uma drive ao mesmo tempo. Enquanto o controlador e o software estão á espera do resultado da procura o controlador pode iniciar uma procura numa outra drive. Alguns controladores podem escrever ou ler numa drive enquanto procuram noutra, mas o controlador de uma disquete não pode ler ou escrever em duas drives ao mesmo tempo. Porem a situação é diferente para discos duros com controladores integrados sendo que num sistema com mais que um destes discos eles podem trabalhar em simultâneo mas apenas uma transferência entre o controlador e a memoria principal é possível ao mesmo tempo.

5.3.1.2- CD-ROMs:

Estes discos ópticos foram inicialmente criados para gravar programas de televisão. Um CD usando um laser infravermelho de alta potencia esta preparada para gravar buracos de diâmetro de 0,8microns num disco mestre revestido de vidro. Primeiro é feito um molde, depois com este molde com resina de policarbonato fundido é injectada para formar o CD com o mesmo padrão que o disco mestre revestido de vidro. Depois uma camada fina de alumínio reflexivo é depositada no policarbonato tapado por um verniz de protecção.

5.3.2-Disk Formatting:

Um disco duro consiste numa stack com placas de alumínio, de liga ou de vidro de 5.25, 3.5 ,… polegadas de diâmetro. Em cada placa esta depositado um metal oxido magnetizável. Antes do disco poder ser usado cada placa deve receber uma formatação do software. Formatar consiste numa serie de faixas concêntricas cada uma contendo algum numero de sectores com pequenas lacunas entre os sectores. Estrutura de um sector do disco:



O preâmbulo começa com um certo padrão de bits que permite ao hardware conhecer o início do sector. Contem também o número do cilindro e do sector bem como outra informação. O campo EEC conte informação redundante que pode ser usada para executar um "recover" quando existem erros de leitura. A posição do sector zero em cada faixa esta o offset da faixa anterior. Este offset permite ao disco ler múltiplas faixas numa operação contínua sem perder dados.

5.3.2-Algoritmos de escalonamento:

Vamos ver quanto tempo leva a ler ou escrever um bloco do disco.

Tempo de procura (o tempo de mover a agulha para o cilindro próprio)
Atraso rotacional (tempo do sector rodar na cabeça).
Tempo de transferência de dados.

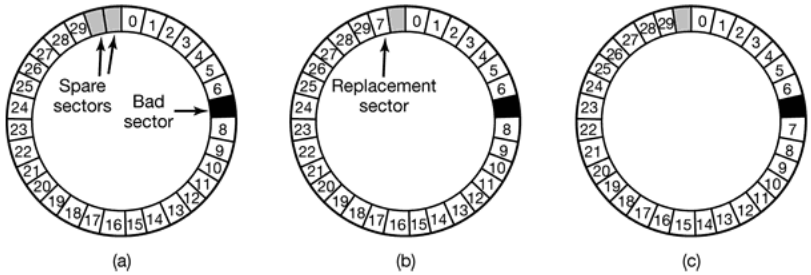
Para a maior parte dos discos o ponto 1 é o que leva a maior parte do tempo o que implica que reduzir este reduz o tempo global. Se a driver do disco requer um de cada vez e executa de forma a "First came, First served" (FCFS) pode ser feito para optimizar o ponto 1. Mas outra estratégia pode ser adoptada: enquanto a agulha esta a procura para satisfazer um pedido outro pedido pode ser feito por outro processo. Muitas drivers do disco mantém uma tabela indexada pelo numero de cilindro com todos os pedidos pendentes para cada cilindro ligados numa lista ligada com as entradas da tabela na cabeça. Desta forma podemos melhorar o ponto 1. Imaginemos um disco com 40 cilindros. Vem 1 pedido para ler um bloco no cilindro numero 11. Enquanto a procura no cilindro numero 11 esta a ser feita chega um novo pedido para os cilindros 1, 36, 16, 34, 9 e 12 (por esta ordem). Estes pedidos são introduzidos na tabela de pedidos pendentes com uma lista ligada separada para cada cilindro. Quando o pedido do cilindro 11 for terminado o driver do disco e pode escolher que pedido tratar de seguida. Se usar o FCFS deveria fazer o pedido do número 1 primeiro e por ai fora. Este algoritmo levaria a movimentos de 10, 35, 20, 18, 25 e 3 num total de 111 cilindros. De uma forma alternativa podia tratar o pedido que tivesse o cilindro mais próximo de forma a minimizar o tempo de procura o que daria uma sequência: 12, 9, 16, 1, 34 e 36. Com esta sequência os movimentos são: 1, 3, 7, 15, 33 e 2 num total de 61 cilindros. Este algoritmo denominado "Shortest Seek First" (SSF) que neste caso tem um custo de quase metade em relação ao FCFS. Mas infelizmente este algoritmo tem um problema: imaginemos mais pedidos a chegar enquanto o primeiro (ou outro) pedido esta a ser processado, se por

exemplo esta a satisfazer o pedido para o cilindro nº 16 e chega um pedido para o cilindro 8, este será o próximo a ser processado e não o nº 1 e se quando esta a processar o nº8 chega um pedido para o nº 13 ele fará este e não o nº 1. O que acontece é que neste pedido a agulha vai passar a maior parte do tempo no meio do disco e os pedidos para posições extremas podem ter de esperar muito. O objectivo de um tempo mínimo de resposta e justiça não será conseguido com este algoritmo. Um outro algoritmo é o algoritmo do elevador, neste algoritmo a agulha continua a mover-se na mesma direcção ate satisfazer os pedidos dessa direcção, quando não houver mais pedidos nessa direcção ela muda de direcção. Este algoritmo obriga o software a manter 1 bit para a direcção (UP ou DOWN). Quando um pedido termina o disco ou a drive do elevador verifica esse bit, se for UP o braço é movido para o pedido mais alto que esta pendente. Se não houver um pedido mais alto a direcção é invertida e o bit é alterado para DOWN e ele move-se para a próxima posição que seja inferior á actual (se existir). Uma propriedade particular deste algoritmo é que dada uma colecção de pedido o limite superior do movimento total é fixo sendo exactamente igual ao dobro do número de cilindros.
Uma pequena alteração a este algoritmo é de procurar sempre na mesma direcção. Quando o cilindro com o nº mais alto (dos pedidos pendentes) for terminado a agulha vai para o cilindro com o nº mais pequeno (dos pedidos pendentes) e depois continua a mover-se para cima. Este algoritmo foi feito já que é pensado que o número de cilindro mais baixo é logo abaixo daquele com o pedido mais alto. Quando varias drives estão presentes no mesmo controlador o operador deve manter uma tabela separada para cada drive.

5.3.3-Tratamento de erros:

Defeitos na produção dos discos introduzem maus sectores, estes não devolvem o valor que foi escrito neles. Se o defeito for pequeno (alguns bits) é possível utilizar este sector deixando o EEC corrigir os erros. Se for grande o erro não pode ser mascarado. Existem duas aproximações gerais para maus blocos:

lidar com eles no controlador ou no sistema operativo. Na primeira abordagem, antes de o disco ser enviado da fábrica é testado e uma lista de maus sectores é escrita no disco. Por cada sector mau uma das peças é substituída.



Erros também podem aparecer durante a operação normal depois do dispositivo ter sido instalado. A primeira linha de defesa para um erro que o EEC não pode tratar é tentar ler outra vez. Caso o controlador avise que esta a ter erros repetidos no mesmo sector ele pode mudar para um "spare sector" antes desse sector morrer. Desta forma nenhum dado é perdido e o tanto o sistema operativo como o utilizador nem notam a ocorrência do problema.
Na segunda abordagem o sistema operativo deve fazer a mesma coisa mas no software. Isto significa que tem primeiro de adquirir uma lista de maus sectores, podem ter essa lista lendo-a do disco ou testando todo o disco. Depois de saber quais os sectores maus, pode construir tabelas remapeadas. Se o sistema operativo estiver a fazer este tratamento tem de ter a certeza que os maus sectores não ocorrem em nenhum ficheiro.
Contudo ainda existe outro problema: Backups. Se o backup é feito ficheiro a ficheiro é importante que não tente copiar os maus blocos do ficheiro. Para prevenir isto o sistema operativo esconde o mau bloco do ficheiro para que a utilidade de Backup não o encontre. Se o Backup for feito bloco a bloco será difícil ou mesmo impossível não ler erros.

5.3.4-Armazenamento estável:

Discos por vezes cometem erros. Bons sectores podem de repente passar a maus.
Para algumas aplicações é essencial que os dados nunca se percam ou fiquem corrompidos. O que é alcançável é um subsistema no disco que: quando é

feita uma escrita ou o disco escreve os dados ou não faz nada deixando os dados existentes intactos. Este sistema é chamado armazenamento estável (Stable Storage) e esta implementado no software. Temos de perceber melhor o modelo de possíveis erros: o modelo assume que quando o disco escreve um bloco ou a escrita é correcta ou incorrecta e neste caso o erro pode ser detectado examinando os valores do EEC. O modelo assume também que uma escrita correcta pode de repente correr mal e ficar ilegível e assume também que o CPU pode falhar. O armazenamento estável usa um par de discos idênticos com os blocos correspondentes a trabalhar em conjunto para formar um bloco livre de erros. Os blocos correspondentes nas duas drives são iguais, qualquer um pode ser lido obtendo-se o mesmo resultado.
Para atingir este objectivo estão definidas 3 operações:

Escritas estáveis: Consiste em primeiro escrever o bloco na drive 1 depois ler o bloco de volta para verificar se foi escrito de forma correcta. Se não o foi então repete n vezes ate estar correcto. Se depois dessas n vezes continuar a falhar este bloco passa para uma spare e a operação é repetida ate ter sucesso. Depois de a escrita no bloco na drive 1 ter sucesso o bloco é escrito na drive 2 e lido as vezes necessárias ate ter sucesso. Na ausência de um crash do CPU o bloco foi escrito e verificado nas duas drives.

Leituras estáveis: Primeiro lê o bloco da drive 1. Se dá um EEC incorrecto a leitura é tentada de novo ate n vezes se necessário. Se nessas vezes deu sempre más EEC's o bloco correspondente é lido da drive 2. Dado o facto de a escrita ter sido estável termos o mesmo bloco nas 2 drives de forma espontânea a ter erros é muito improvável, sendo assim esta operação leva a uma leitura estável.

Crash recovery: Depois de um crash (colisão), um programa recovery faz um scan aos dois discos comparando os blocos correspondentes. Se um par de blocos iguais esta bom nada é feito. Se um deles tem um erro EEC, o bloco mau e reescrito com os dados correspondentes do bloco bom. Se os dois blocos estão bons mas são diferentes o bloco da drive 1 é escrito na drive 2.

----------Capitulo 6 – Gestão de ficheiros----------

6.1- File naming:

Ficheiros são um mecanismo abstracto. Fornecem uma forma de guardar informação no disco e mais tarde aceder á mesma. Deve ser feito de forma a proteger o utilizador de detalhes do tipo onde a informação é guardada. Provavelmente o mecanismo mais importante é a forma como os objectos são geridos e nomeados. Quando um processo cria um ficheiro dá-lhe um nome. Quando o processo termina o ficheiro continua a existir e pode ser acedido através do seu nome por outro processo. As regras para nomear um ficheiro diferem de sistema para sistema sendo que actualmente os sistemas operativos permitem strings de uma a oito letras como nome do ficheiro. Alguns ficheiros distinguem maiúsculas de minúsculas. A maior parte dos sistemas operativos suporta nomes com duas partes separadas por um ponto: program.c. A parte depois do ponto é chamada de extensão.

6.2- File structure:

Ficheiros podem ser estruturados numa de varias formas. Um ficheiro é uma não estruturada sequência de bytes. O sistema operativo não sabe nem quer saber o que esta no ficheiro, tudo o que ele vê são bytes. Os utilizadores podem o que quiserem, dar o nome que quiserem a um ficheiro que o sistema operativo não ajuda nem se mete no caminho.

6.3- File types:

Muitos sistemas operativos suportam vários tipos de ficheiros. “Regular files” são aqueles que contem informação do utilizador. Directorias são ficheiros do sistema para manter a estrutura do sistema de ficheiros. “Charater special files” estão relacionados com os dispositivos de I/O. “Block special files” são usados para modular discos.

“Regular files” são geralmente ou ficheiros ASCII ou ficheiros binários. Os ficheiros ASCII consistem em linhas de texto. A grande vantagem deste tipo de ficheiros é que podem ser vistos e imprimidos da forma que são e qualquer editor de texto os pode abrir. Os ficheiros binários ao ser listados apresentam um conjunto de caracteres incompreensíveis que aparentemente parecem lixo. Todos os sistemas operativos têm de reconhecer pelo menos um tipo de ficheiros: o seu próprio ficheiro executável.

6.4- File access:

Os primeiros sistemas operativos só permitiam um tipo de acesso: sequencial. Nestes sistemas um processo só podia ler os bytes de um ficheiro de forma sequencial começando no seu inicio.

Quando os discos começaram a ser usados para guardar ficheiros passou a ser possível ler os bytes de um ficheiro fora de ordem, ou aceder aos dados por uma chave. Estes ficheiros chamam-se: “random access files”. Estes tipos de ficheiros são essenciais para muitas aplicações como por exemplo bases de dados. Existem dois métodos para se especificar onde queremos começar a ler. No primeiro a operação de leitura dá a posição no ficheiro onde quer começar a ler. Na segunda uma operação especial SEEK (procura) é providenciada para estabelecer a posição. Depois do seek o ficheiro pode ser lido de forma sequencial a partir dessa posição. Nos sistemas modernos todos os ficheiros são automaticamente do tipo “random access”.

6.5- File atributes:

Todos os ficheiros têm um nome e os seus dados. Adicionalmente os sistemas operativos associam outra informação ao ficheiro como por exemplo data e hora de criação, tamanho, etc. Isto são os atributos dos ficheiros. A lista de atributos pode variar consideravelmente de ficheiro para ficheiro.

Attribute	Meaning
Protection	Who can access the file and in what way
Password	Password needed to access the file
Creator	ID of the person who created the file
Owner	Current owner
Read-only flag	0 for read/write; 1 for read only
Hidden flag	0 for normal; 1 for do not display in listings
System flag	0 for normal files; 1 for system file
Archive flag	0 for has been backed up; 1 for needs to be backed up
ASCII/binary flag	0 for ASCII file; 1 for binary file
Random access flag	0 for sequential access only; 1 for random access
Temporary flag	0 for normal; 1 for delete file on process exit
Lock flags	0 for unlocked; nonzero for locked
Record length	Number of bytes in a record
Key position	Offset of the key within each record
Key length	Number of bytes in the key field
Creation time	Date and time the file was created
Time of last access	Date and time the file was last accessed
Time of last change	Date and time the file has last changed
Current size	Number of bytes in the file
Maximum size	Number of bytes the file may grow to

Os primeiros quatro atributos estão relacionados com a protecção do ficheiro e dizem quem lhe pode ou não aceder.

As flags são bits que controlam alguma propriedade específica. O “Record length”, “Key position” e “Key length” só estão presentes em ficheiros cujos registos podem ser vistos usando uma chave. Os “Time…” guardam os dados sobre quando o ficheiro foi criado, a ultima vez acedido e a ultima vez modificado. “Current size” diz-nos o tamanho do ficheiro no momento.

6.5- File operations:

Chamadas ao sistema mais comuns relativas a ficheiros:

Create: criar um ficheiro sem nenhum dado inicialmente.
Delete: Apagar o ficheiro.
Open: antes de usar o ficheiro o processo tem de o abrir. Esta chamada permite ao sistema fazer um "fetch" aos atributos e á lista de endereços do disco e passar para a memória principal para rápido acesso em chamadas posteriores.
Close: O disco esta escrito em blocos e fechar o ficheiro força a escrita no ultimo bloco do ficheiro mesmo que o bloco ainda não esteja completamente cheio.
Read: dados são lidos do ficheiro.
Write: Dados são escritos no ficheiro normalmente a partir do fim do ficheiro. Se forem escritos por exemplo a partir do meio do ficheiro o que esta para baixo é perdido.
Append: è uma forma restrita de write. Só se podem acrescentar dados a partir do fim do ficheiro.
Seek: Para "random access files" é preciso um método para especificar o início de onde queremos ler os dados. Esta chamada ao sistema faz isso.
Get attributes: processos muitas vezes precisam de ler os atributos do ficheiro para executarem a sua função. Como exemplo de um atributo temos o make
Set attributes: Alguns atributos podem ser atribuídos pelo utilizador, e podem ser alterados depois do ficheiro ter sido criado.
Rename: renomear um ficheiro.