Objectivos Principais de um Sistema Operativo

O sistema operativo tem como objectivos gerir todos os recursos do computador, bem como providenciar aos utilizadores uma interface simples que permita a interacção dos utilizadores com o hardware do computador, com o auxílio de programas existentes no computador.

Sistema Operativo como máquina Virtual

O sistema operativo tem a função de servir de intermediário entre o hardware e o utilizador, de modo a simplificar a tarefa de interacção entre ambas as partes. Como tal, o sistema operativo funciona como máquina virtual que é equivalente ao hardware, porém muito mais acessível. Deste modo o utilizador interage com o hardware sabendo o mínimo possível sobre tarefas de gestão do hardware em si.

Sistema Operativo como Gestor de Recursos

O sistema operativo ao funcionar como gestor de recursos segue uma perspectiva **bottom-up**, ao tratar da gestão dos recursos de hardware disponíveis na máquina. Nesta perspectiva cabe ao sistema operativo fornecer um esquema de alocação dos processadores, das memórias e dos dispositivos de entrada/saída entre os múltiplos processos que competem pela utilização dos respectivos recursos. A tarefa principal da perspectiva bottom-up consiste essencialmente na gestão da utilização de cada um dos recursos da máquina, controlando o tempo de uso de cada um e assegurando o acesso ordenado aos recursos regulando as requisições dos diversos processos dos utilizadores do sistema.

Recursos de um Sistema Operativo

Os tipos de recursos que podem existir são os **recursos lógicos** que são por exemplo, os dados que poderão ser partilhados por entre utilizadores, e os **recursos físicos** que se tratam do hardware usado pelos utilizadores, como por exemplo, impressoras ,etc..

Entidades responsáveis pela gestão de Recursos

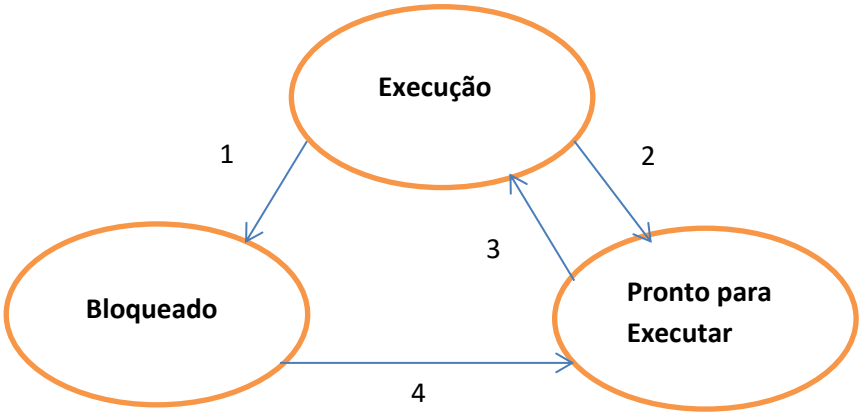
As entidades responsáveis pela gestão de recursos são os diversos processos que se vão originando à medida que os utilizadores requerem que certa operação seja executada, sendo estes depois geridos pelo **escalonador** do Sistema Operativo.

Processo

Um processo é o conjunto das instruções de uma certa aplicação a serem executadas num dado momento. Um processo tem sempre associado um endereço de memória, que contém o código executável, os dados do programa e a sua stack, que irá armazenar as chamadas de sistema feitas a este processo. Poderão igualmente estar associadas outras coisas a cada processo tais como registos de hardware, o contador do programa, o ponteiro da stack, entre outras.

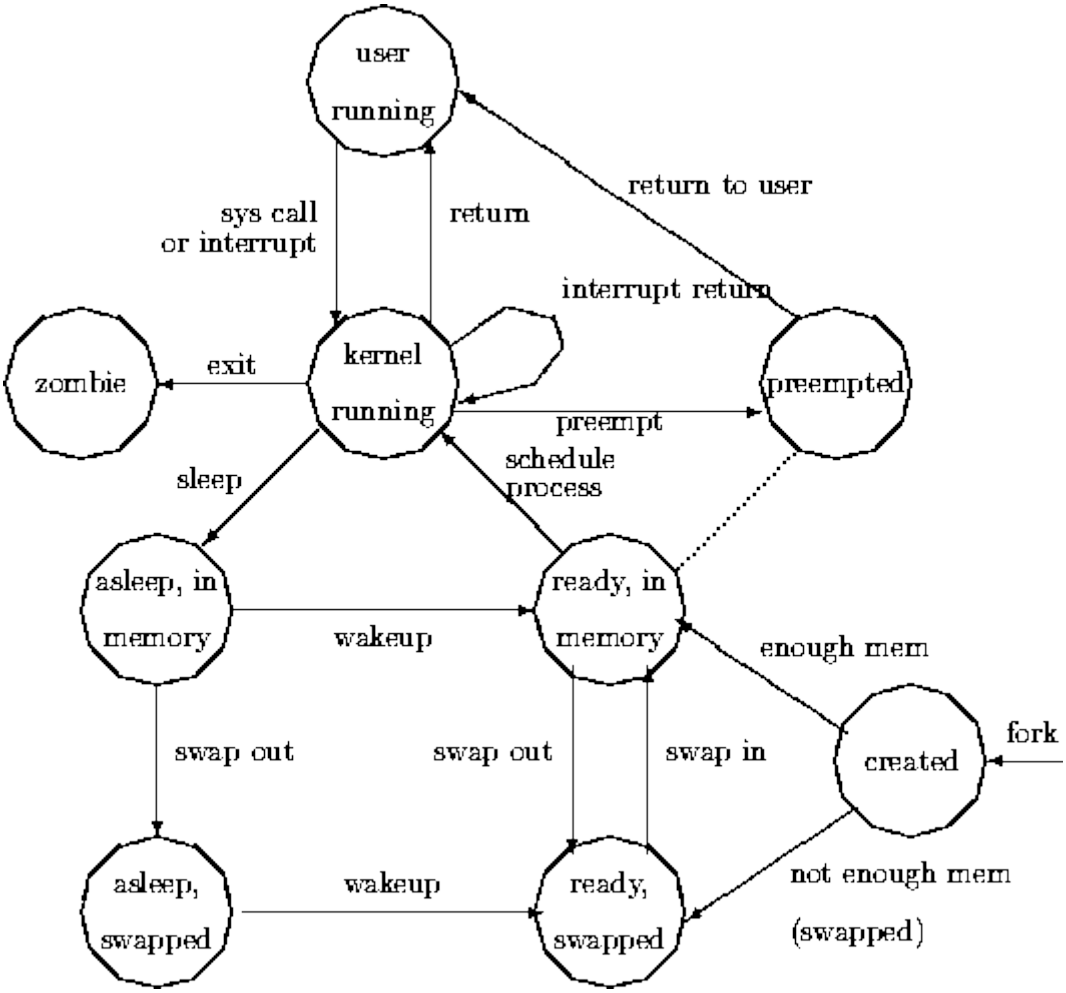
Estados de um processo

Um processo tem na essência três estados principais:



1- **Processo bloqueia para input**
2- **Escalonador selecciona outro processo**
3- **Escalonador escolheu o processo**
4- **Input torna-se possível**

Ciclo de Vida de um processo



Processos Concorrentes

Não existe processamento em paralelo. Existe **Time Sharing,** que funciona de forma a que o SO disponibilize determinadas fatias de tempo para a execução de cada processo.

Thread

Uma thread consiste numa entidade programada para execução na CPU. Trata-se da unidade de processamento mais pequena que um sistema operativo poderá ter, e é contida dentro de um processo. Poderão existir múltiplas threads dentro de um processo, sendo isto denominado de **Multithreading.** As threads podem partilhar recursos entre si, tais como a memória, ao contrário dos processos, onde isto não se verifica. Cada thread possui uma stack que irá ser uma propriedade única da mesma.

Distinção entre Processo e Thread

Processo	Thread
Não é possível a partilha de recursos entre processos	É possível a partilha de recursos entre threads, como por exemplo a memória e as variáveis globais.
Um processo é composto por uma thread pelo menos, sendo várias threads num processo denominado de Multithreading.	Executa um determinado conjunto de instruções para as quais foi originada.
Cada processo requer um espaço de memória.	As threads são sujeitas a escalonamento internamente ao nível do processo.
Processos são utilizados para agrupar recursos	Cada thread possui uma stack que a torna única mediante todas as outras.

Tabela de Processos

A tabela de processos é o lugar onde são armazenadas todas as informações relativas aos processos que não façam parte dos endereços de memória, sendo esta um array de estruturas, um para cada processo existente. A tabela de processos está centrada em 3 categorias:

- **Process Management** (Gestão de Processos);
- **Memory Management** (Gestão de Memória);
- **File Management** (Gestão de Ficheiros);

O armazenamento da informação dos vários ponteiros para os segmentos de dados, texto e stack na tabela de processos é importante, pois estes ponteiros irão permitir o acesso aos processos lá existentes e determinar qual a acção a tomar.

Multiprogramação

A multiprogramação baseia-se no rápido encadeamento do processador ao executar múltiplos processos, realizando estas operações em tempos de execução atribuídos ao CPU, para cada execução de um processo.

Pseudoparalelismo de Processos

Consiste na execução dos processos no menor espaço de tempo, criando uma ilusão de simultaneidade de execução das aplicações, quando no entanto estas estão a ser executadas em sequência. Este tipo de situação é mais visível em casos em que somente existe um único processador no computador e é mais notável em escalas de tempo mais reduzidas, como por exemplo, 1 segundo.

Multiprocessing

Consiste em ter dois ou mais processadores a partilhar a mesma memória física, permitindo a existência do paralelismo de processos, ao contrário do que aconteceria em computadores com um único processador.

Escalonador do Sistema Operativo

É a parte do Sistema Operativo responsável pela regulação dos processos, decidindo a ordem da sua execução. O algoritmo utilizado para esta tarefa é chamado de **algoritmo de escalonamento**, sendo que este obedece a um conjunto de características, com o propósito de regular bem os processos. As principais características são:

- **Justiça**, de modo a garantir que todos os processos terão iguais chances de uso do processador.
- **Eficiência**, de modo a manter o processador ocupado 100% do tempo.
- Tempo de Resposta, de modo a minimizar o tempo de resposta aos utilizadores.
- **Turnaround,** de modo a minimizar o tempo que os utilizadores aguardam pelo fim da aplicação.
- **Throughput**, de modo a maximizar o número de processos executados na unidade de tempo, geralmente uma hora.

O escalonamento pode seguir duas abordagens:

- **Preemptive Scheduling,** que consiste em seleccionar um processo e deixa-lo correr por um período máximo de tempo definido.
- **Nonpreemptive Scheduling,** que consiste em seleccionar um processo e deixa-lo correr indefinitivamente até bloqueie ou voluntariamente liberta a CPU.

Round Robin Scheduling

Lista de processos prontos para serem executados. Esta lista irá fazendo o round(passar para o fim da fila) dos processos executados.

Shortest Process Next

Algoritmo que coloca em execução o processo de mais curta execução. Uma metodologia inútil em sistemas interactivos.

Chamadas de Sistema

Métodos especiais invocados pelo Sistema operativo. As chamadas de sistema estão também guardadas na zona de memória reservada ao sistema operativo.

Fork()

Chamada de sistema para criação de processos em Linux. É feita em modo kernel, pois sendo uma chamada de sistema é algo programado pelo programador do sistema e não por uma aplicação.

Driver (Controlador)

Fragmento de código, software que é construído para permitir que existir comunicação entre o computador e um determinado dispositivo de hardware.

DMA (Direct Memory Access)

Chip específico utilizado para efectuar especificamente transferências de ficheiros do disco para a memória, poupando assim ciclos de execução ao CPU, que poderão ficar disponíveis para outros processos que pretendam executar algo.

IRQ (Interrupt Request)

Avisa a CPU, de quando existe alguma espécie de alteração/mudança num determinado dispositivo de hardware. Exemplo: Carregar na tecla de um PC/Como se geram alguns eventos.

Entradas/Saídas

Efectuam a ligação entre o PC e tudo o resto. Inputs com Outputs.

Objectivos da comunicação entre processos

A comunicação entre processos é essencial, pelo que existe uma necessidade de haver comunicação entre certos processos, sendo esta comunicação preferencialmente de forma estruturada, que não seja baseada em interrupções. O grande objectivo da comunicação entre processos é a possibilidade de todos os processos possam aceder a determinados recursos, ou à **região crítica**, sem que haja problemas no decorrer dessa operação.

Existem múltiplos tipos de comunicação entre processos entre os quais:

- **Pipes**
- **Semáforos**
- **Signals**
- **Memória Partilhada**

Pipes

O conceito de pipe baseia-se num pseudo-ficheiro que poderá ser utilizado para estabelecer uma **comunicação entre dois processos**. Quando um dos processos desejar comunicar com o outro, este escreve no pipe os respectivos dados, tratando o pipe como um **ficheiro de output**.

O outro processo por sua vez irá ler os dados provenientes do pipe como se este fosse um **ficheiro de input**, sendo que este tipo de comunicação é semelhante a uma leitura-escrita de ficheiros. Pode-se dizer que os pipes são canais de informação entre processos. A comunicação por pipes é **síncrona, num único sentido**. Somente um processo escreve e somente um processo lê.

Semáforos

O conceito de semáforo consiste numa variável do tipo abstracto que tem como objectivo controlar o fluxo de acesso dos processos à **região crítica** (conjunto de instruções executadas por um processo com vista a aceder a um recurso partilhado por vários processos de forma exclusiva). Para que tal possa ocorrer, o semáforo possui duas operações, **DOWN e UP.**

- ***UP*, ou P()** irá bloquear o acesso dos outros processos à região critica.
- ***DOWN*, ou V()** irá libertar o bloqueio imposto pelo método, libertando assim as restrições à região critica.

Semáforo Mutex

O mutex é um tipo de **semáforo binário** que apenas poderá assumir os valores 0 e 1, e tem como objectivo, regular o acesso dos processos à região critica, com auxilio dos métodos P() (UP) e V() (DOWN).

Algoritmos de Semáforos

Existem diversas soluções no que toca à comunicação de processos com semáforos, entre os quais:

- **Produtor-Consumidor**

```
#define N 100                                   //número de slots no buffer
Typedef int semaphore;                          //semáforo
Semaphore mutex = 1;                            //controla o acesso à região critica
Semaphore empty = N;                    //conta o número de slots vazios no buffer
Semaphore full = 0;                     //conta o número de slots ocupados no buffer
Void producer(void)
{
        Int item;
While(TRUE){                                    //TRUE é a constante 1
        Item = produce_item();                  //gera algo para colocar no buffer
        Down(&empty);                           //decrementa o número de slots vazios
```

```
        Down(&mutex);                   //acede à região critica
        Insert_item(item);              //coloca o novo item no buffer
        Up(&mutex);                     //sai da região critica
        Up(&full);              //incrementa o número de slots ocupados
        }
}
Void consumer(void)
{
        Int item;
While(TRUE){                    //Loop infinito
        Down(&full);            //decrementa o número de slots ocupados
        Down(&mutex);           //acede à região critica
        Item = remove_item();    //remove o item do buffer
        Up(&mutex);             //sai da região critica
        Up(&empty);             //incrementa o número de slots vazios
        Item = consume_item();  //efectua alguma acção com o item
        }
}
```

- **Jantar de Filósofos**

```
#define N 5                     //número de filósofos
#define LEFT (i+N-1)%N           //número do vizinho esquerdo
#define RIGHT (i+1)%N           //número do vizinho direito
#define THINKING 0              //filósofo está a pensar
#define HUNGRY 1                //o filósofo está a tentar arranjar garfos
#define EATING 2                 //o filósofo está a comer
Typedef int semaphore;          //semáforo
Int state[N];                   //array de estados
```

```
Semaphore mutex = 1;            //exclusão mútua para regiões criticas
Semaphore s[N];                 //um semáforo por filósofo
Void philosopher(int i)         // i : número do filósofo, de 0 até N-1
{
    While(TRUE){                //repete eternamente
        Think();                //filósofo está a pensar
        Take_forks(i);          //adquire dois garfos ou bloqueia
        Eat();                  //yum-yum, esparguete…
        Put_forks(i);           //coloca os garfos de volta na mesa
        }
}
Void take_forks(int i)          // i : número do filósofo, de 0 até N-1
{
    Down(&mutex);               //acede à região critica
    State[i] = HUNGRY;          // indica que o filosofo i está com fome
    Test(i);                    //tenta arranjar dois garfos
    Up(&mutex);                 // sai da região crítica
    down(&s[i]);                //bloqueia se não arranjar garfos
}
Void put_forks(i)               // i : número do filósofo, de 0 até N-1
{
    Down(&mutex);               //acede à região crítica
    State[i] = THINKING;        //filósofo terminou de comer
    Test(LEFT);                 //verifica se o filósofo da esquerda pode comer
    Test(RIGHT);                //verifica se o filósofo da direita pode comer
    Up(&mutex);                 //sai da região crítica
}
```

```
Void test(i)                        // i : número do filósofo, de 0 até N-1
{
    If(state[i] == HUNGRY && state[LEFT] != EATING && state[RIGHT] != EATING) {
State[i] = EATING;
Up(&s[i]);
    }
}
```

- **Leitor-Escritor**

```
Typedef int semaphore;              //semáforo
Semaphore mutex = 1;                //controla o acesso a 'rc'
Semaphore db = 1;                   //controla o acesso à base de dados
Int rc = 0;                         //# de processos a ler ou que pretendam ler
Void reader(void)
{
    While(TRUE) {                   //repete eternamente
        Down(&mutex);               //recebe exclusividade de acesso a 'rc'
        Rc = rc+1;                  // mais um leitor
        If(rc == 1) down(&db);      //se for o 1ºleitor
        Up(&mutex);                 //liberta a exclusividade de acesso a 'rc'
        Read_data_base();           //acede aos dados
        Down(&mutex);               //recebe exclusividade de acesso a 'rc'
        Rc = rc-1;                  //menos um leitor
        If(rc == 0) up(&db);        //se for o ultimo leitor
        Up(&mutex);                 //liberta a exclusividade de acesso a 'rc'
        Use_data_read();            //região não critica
        }
    }
```

```
Void writer(void)
{
        While(TRUE){                    //repete eternamente
                Think_up_data();        //região não critica
                Down(&db);              //recebe exclusividade de acesso
                Write_data_base();      //actualiza os dados
                Up(&db);                //liberta a exclusividade de acesso
        }
}
```

- **Barbeiro Adormecido**

```
#define CHAIRS 5                    //# de cadeiras para os clientes
Typedef int semaphore;              //semáforo
Semaphore customers = 0;            //# de clientes à espera de serem atendidos
Semaphore barbers = 0;              //# de barbeiros à espera de clientes
Semaphore mutex = 1;                //para exclusão mútua
Int waiting = 0;                    //clientes estão à espera (não atendidos)
Void barber(void)
{
    While(TRUE){
        Down(&customers);   //adormece se o # de clientes for 0
        Down(&mutex);       //adquire acesso a 'waiting'
        Waiting = waiting-1;    //decremente o número de clientes à espera
        Up(&barbers);       //um barbeiro está pronto para cortar cabelo
        Up(&mutex);         // liberta o acesso a 'waiting'
        Cut_hair();         // corta o cabelo (fora da região critica)
    }
}
```

```
Void customer(void)
{
    Down(&mutex);                   //acede à região critica
    If(waiting < CHAIRS){           //se não houverem cadeiras disponíveis, sai
        Waiting = waiting +1;       //incrementa o número de clientes à espera
        Up(&customers);             //acorda o barbeiro se necessário
        Up(&mutex);                 //liberta o acesso a 'waiting'
        Down(&barbers);             //vai dormir se o # de barbeiros for 0
        Get_haircut();              //senta-se e é atendido
    }else{
        Up(&mutex);                  //loja está cheia, não espera
    }
}
```

Signals

Os signals são o equivalente em software às interrupções de hardware. Têm a particularidade de poderem ser utilizados na comunicação entre processos. Os processos podem informar o sistema do que pretendem que aconteça quando o signal ocorrer. Este poderá:

- Ignorar o signal;
- Capturar o signal;
- Deixar que o signal mate o processo (opção por defeito, na maior parte dos signals).

Um processo para capturar signals, necessita de um procedimento que faça a gestão dos mesmos, sendo que o signal ao ser capturado é reencaminhado para procedimento que o irá gerir.

Um processo apenas poderá por signals com os membros do seu grupo, que consistem no **processo pai** (e respectivos antepassados), **irmãos** e respectivos **processos filho** (e respectivos descendentes). Um processo poderá enviar um signal global a todos os seus parentes com uma única chamada de sistema.

Memória Partilhada

A memória partilhada é um tipo de memória que pode ser **acedida por múltiplos processos**, permitindo a comunicação entre si, e evitando cópias redundantes. Este mecanismo dá-se quando um processo pretende efectuar troca de dados com outro processo, sendo que o outro processo irá reservar um espaço na RAM onde todos os processos que assim o pretendam possam aceder à informação do processo. É um método de comunicação

relativamente rápido, no entanto deve-se ter cuidado e coordenar este mecanismo através do uso de semáforos, por exemplo, de modo a **evitar interferências e bloqueios indesejáveis**.

Paginação

A paginação é uma técnica utilizada pelos **sistemas de memória virtual**. Consiste na divisão do espaço de onde estão armazenados os endereços virtuais. Estes endereços virtuais são mapeados em endereços físicos da memória, através do **MMU (Memory Management Unit).**

Os Endereços virtuais são divididos em unidades denominadas **páginas**, sendo o correspondente em memória física denominado de **page frames**. Estas são sempre do mesmo tamanho. No hardware propriamente dito, existe um **Present/Absent bit** que verifica se as páginas estão fisicamente presentes na memória.

Page Fault

É uma situação que se dá quando uma aplicação acede a uma página não mapeada. Pelo que o MMU ao detectar esta anomalia vai indicar ao sistema operativo que seleccione a page frame **menos usada** e reescreva o seu conteúdo no disco, sendo que de seguida irá buscar a página não mapeada e referenciar esta à page frame libertada.

Page Table

É o lugar onde são **indexadas** as páginas mapeadas, indicando o número da page frame correspondente a cada página.

Devem igualmente salientar dois aspectos:

- A Page Table pode vir a ser **extremamente grande**;
- O processo de mapeamento necessita de ser **bastante rápido**;

TLB

Translator Lookaside Buffer, ou memória associativa contida dentro do MMU, que contém um número limitado de entradas e é responsável pelo mapeamento dos endereços virtuais sem ter de aceder à memória principal. Cada entrada contém os seguintes dados:

- Número Página Virtual
- Modified Bit
- Permissões (R/W/X)
- Número Página Física
- Valid Bit

Algoritmos de Substituição

- **Algoritmo Óptimo**

No momento em que ocorre um page fault, existe um conjunto de páginas armazenadas em memória. Uma dessa páginas será referenciada na próxima

instrução, e vai “empurrando” o problema do page fault o máximo possível. Sendo este algoritmo inatingível, pois o sistema nunca saberia qual a próxima pagina a referenciar.

- **NRU (Not Recently Used Page Algorithm)**

Remove aleatoriamente uma página da classe (0 – Não Referenciadas, Não modificadas; 1 – Não Referenciadas, Modificadas; 2 – Referenciadas, Não Modificadas; 3 – Referenciadas, Modificadas) com o menor número de identificação, e que não esteja vazia. Tem uma performance relativamente aceitável, mas não é a melhor solução.

- **FIFO (First in, First Out)**

Consiste em remover a página do início da fila, adicionando a nova página ao fim da fila. O FIFO, na sua essência pura nunca é usado.

- **Algoritmo da Segunda Chance**

Seguindo a lógica da FIFO, no caso de ocorrer uma situação de page fault, ao invés da página do início ser removida, esta é colocada no fim da fila, sendo-lhe concedida uma segunda oportunidade.

- **Algoritmo do Relógio**

Semelhante ao algoritmo da Segunda Chance, este segue a analogia de um relógio, onde o ponteiro aponta para a página mais velha. Tem uma implementação mais eficiente que o anterior.

- **LRU (Least Recently Used Page Algorithm)**

Consiste em remover a página que não tenha sido referenciada à mais tempo da memória, na ocorrência de um page fault, seguindo a analogia de as páginas que não são usadas, também não irão ser usadas muito em breve. Requer a manutenção de uma lista ligada entre todas as páginas da memória, sendo a sua implementação dispendiosa.

Segmentação

A segmentação é uma técnica que incide na **criação de múltiplos espaços de endereçamento** denominados de **segmentos**. Cada segmento é uma sequência linear de endereços de 0 até um máximo, sendo o tamanho de cada segmento poderá variar entre 0 e o máximo permitido. Como tal, os segmentos poderão ter variados tamanhos. Ao serem independentes entre si podem aumentar ou diminuir o seu tamanho, sem afectar outros segmentos. O segmento é considerado uma **entidade lógica**, podendo conter um procedimento, um array, uma stack, um conjunto de variáveis escalares, mas **nunca uma mistura de tipos**.

Bibliotecas Partilhadas

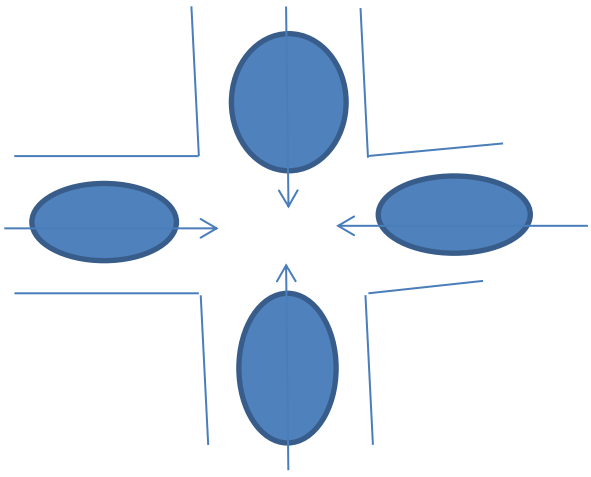
Ao colocar determinada biblioteca num sistema segmentado, esta pode ser partilhada entre múltiplos processos, eliminando a necessidade dessa biblioteca ter de ser colocada no espaço de endereçamento de cada processo.

Distinção entre Paginação e Segmentação

	Paginação	**Segmentação**
O Programador necessita de conhecer as técnicas que estão a ser usadas?	Não	Sim
Quantos espaços de endereços virtuais existem?	1	Muitas
Pode o espaço reconhecido ser maior que o espaço físico real?	Sim	Sim
Existem espaços separados para procedimentos e dados?	Não	Sim
Podem dados e procedimentos ser protegidos separadamente?	Não	Sim
Podem as tabelas cujos tamanhos variem ser acomodadas com facilidade?	Não	Sim
Partilha de procedimentos entre utilizadores facilitada?	Não	Sim
Porque foi esta técnica inventada?	Para obter um grande conjunto de espaço de endereçamento sem ter de comprar mais memória	Para permitir a programas e dados serem separados em espaços diferentes e ajudar na partilha e protecção

Deadlock

Um **deadlock** verifica-se quando num conjunto de processos, cada processo bloqueia aguardando um evento que somente outro processo poderá despoletar.



Estratégias para resolver o problema:

- **Ignorar o problema;**
  - **Algoritmo da Avestruz**

    Consiste em ignorar o problema por completo, sendo que este tipo de abordagem é adoptada em situações em que a solução seria muito restritiva.

- **Detecção e recuperação do problema;**
  - **Detecção de deadlocks em sistemas**
    - Com um único recurso de cada tipo
    - Com múltiplos recursos de cada tipo
  - **Recuperação:**
    - **Recovery through Preemption**

      Em casos em que é possível tirar um recurso ao seu dono temporariamente e entregá-lo a outro processo. Requer intervenção manual na maior parte dos casos.

    - **Recovery through Rollback**

      Em situações em que os deadlocks são relativamente frequentes, é possível que os processos sejam alvo de verificações periódicas, armazenando o estado do processo em arquivos, nunca reescrevendo em cima dos antigos. Ao ser detectado o deadlock os processos irão regredir a um estado anterior, deste modo regredindo no tempo mas solucionando o problema.

    - **Recovery through Killing Processes**

      Em certos casos a maneira mais eficiente de acabar com um deadlock consiste em matar um ou mais processos, de modo a acabar com o ciclo.

- **Evitar o problema;**
  - **Determinar trajectórias de recursos**
  - **Estados Seguros e Inseguros**
    - Um estado é dito seguro se não provocar deadlocks e conseguir satisfazer todas as requisições pendentes dos processos em execução.
  - **Algoritmo do Banqueiro**
    - **Para um único tipo de Recurso**

      Considera cada solicitação de um recurso no momento em esta ocorre, e verifica se irá conduzir a um cenário seguro, caso não o seja o atendimento será adiado.

    - **Para diversos tipos de Recursos**

      Irá verificar com base numa matriz, as quantidades de recursos disponíveis no momento, podendo verificar se determinada requisição é segura ou

não. Este algoritmo no entanto na prática é inútil pois os processos nunca irão saber com antecedência a quantidade de recursos que irão precisar.

- **Prevenir (Evitar o problema a 100%);**
  - **Ataque à exclusividade mútua**

  Assegurar que o mínimo possível de processos acede aos recursos.

  - **Ataque ao "Hold and Wait"**

  Exigir que todos os processos requisitem os recursos necessários, antes de iniciar a sua execução.

  - **Ataque à "No Preemption Condition"**

  Retirar à força um recurso em utilização por parte de um processo, sendo esta hipótese inaceitável.

  - **Ataque ao problema da condição de espera circular**

  Consiste em ordenar numericamente todos os recursos, eliminando assim todos os potenciais deadlocks, mas nunca obtendo uma ordenação óptima.

Gestão de memória em listas ligadas

Algoritmos:

- **First-Fit**
  - O gestor de memória irá percorrer a lista de segmentos até detectar um espaço livre suficientemente grande. Esse espaço é repartido em duas partes:
    - Uma para o processo
    - Uma para a memória que restar, excepto em casos que o processo caiba perfeitamente no espaço.
- **Best-Fit**
  - Percorre a lista completa e escolhe o espaço mais pequeno que seja adequado. Em vez de encontrar um espaço grande, irá procurar um espaço recomendado comparativamente ao tamanho do processo, evitando assim repartir espaços grandes.

Multiplexagem

Consiste na partilha do uso do processador entre as várias tarefas, de forma a atendê-las da melhor maneira possível.

Gestão Memória

- **Re-alocação**

A solução passa em permitir que as instruções possam ser alteradas, à medida que a partição do programa seja carregada.

- **Protecção**

A solução passa em dividir a memória em blocos e atribuir um código de protecção de 4 bits a cada um deles. O código é também armazena na PSW, sendo designado como chave de memória.

Solução para ambos:

Equipar máquina com dois “Registers Especiais”, denominados de Base Register e Limit Register. Um processo ao ser alocado, inicia o Base Register com o seu endereço no inicio da partição e o Limit Register é carregado com o tamanho da partição. Desta maneira o hardware protege os Registers e previne eventuais interferências por parte de programas do utilizador.

Swapping

Swapping é uma estratégia que consiste em fazer a troca de um processo entre a memória e o disco.

Memória Virtual

Mecanismos que permite que o programas estejam em execução quando somente uma parte destes esteja presente na memória principal.