



RESUMOS SO

INTRODUÇÃO

Interrupções vs Polling

Tempo Virtual

Multiprogramação

Multiprocessamento

Modos de Execução

GESTÃO DE PROCESSOS

Conceitos de Processo

O processo

Estado de um processo

Process Control Block (PCB)

Escalonamento de Processos

Filas de escalonamento

Critérios de escalonamento

Mudança de contexto

Algoritmos de Escalonamento de Processos

First Come, First Served Scheduling

Shortest Job First Scheduling

Priority Scheduling

Round-Robin Scheduling

Multilevel Feedback Queue Scheduling

NOÇÕES DE PROGRAMAÇÃO CONCORRENTE

Mecanismos de Sincronização

Semáforos

GESTÃO DE MEMÓRIA

Espaços de Endereçamento Lógicos e Físicos

Swapping

Alocação de Memória

Paginação

Segmentação

Memória Virtual

Problemas

Solução dos problemas

Demanding Page

Copy-on-Write

Rejeição de páginas

Alocação global vs Alocação local

Trashing

Tamanho das páginas

INTRODUÇÃO

O sistema operativo providencia um meio/ambiente no qual outros programas podem realizar trabalho útil. Para além disso funciona como um intermediário entre o utilizador e o *hardware* do computador. Pode se ver o sistema operativo como uma extensão da máquina, i.e., simula uma máquina virtual por acima da máquina real escondendo os detalhes do *hardware* através de API's mais fáceis de usar.



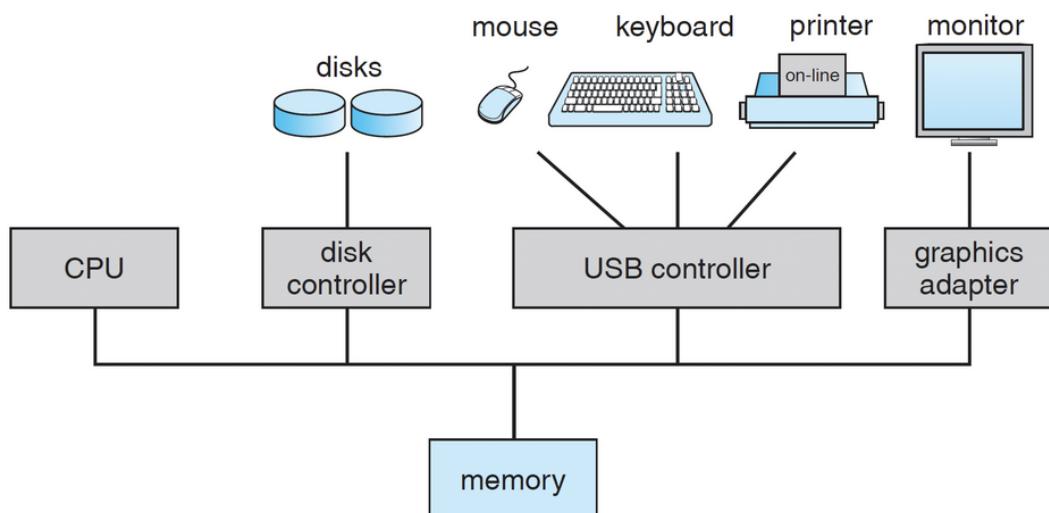
Analogia: O sistema operativo é como um governo, em si não tem nenhuma função útil. Simplesmente fornece um ambiente em que os outros programas possam fazer trabalho útil.

Um sistema operativo é composto por:

- **Hardware:** composto por uma unidade central de processamento (CPU), memória e dispositivos I/O (recursos básicos para o sistema). O sistema operativo deve colocar o *hardware* à disposição dos programas e utilizadores de uma forma conveniente, protegida, eficiente e justa.
- **Aplicações:** processadores de texto, compiladores, *browsers*, etc. (definem a maneira em como os recursos são usados para resolver os problemas dos utilizadores)
- **Sistema operativo:** controla o *hardware* e coordena o seu uso entre as diferentes aplicações
- **Utilizadores do sistema**

Pode-se ver o sistema operativo de dois pontos de vista:

- **Utilizador:** o objectivo é maximizar o trabalho que o utilizador está a executar. Nesta perspectiva, o sistema operativo, é concebido para **facilitar trabalho**.
- **Sistema:** do ponto de vista do computador, o sistema operativo é o programa mais envolvido com o *hardware*, é um programa de **controlo**. Nesta perspectiva, é visto como um alocador de recursos. Esses recursos podem ser tempo de *CPU*, espaço de memória, espaço de armazenamento de ficheiros, dispositivos *I/O*, e por aí fora.



Analogia: é o sistema operativo que define a 'personalidade' de um computador.

Um computador consiste num ou mais *CPU's* e um conjunto de controladores de dispositivos conectados através de um *bus* que fornece acesso a memória partilhada. Para garantir acesso ordenado à memória, é fornecido um **controlador de memória** cuja função é sincronizar acessos à memória.

Quando um computador arranca ou é reiniciado necessita de um programa inicial para correr. Este programa tende a ser simples e designa-se por **bootstrap program**. Tipicamente, está guardado em memória apenas de leitura (**ROM - Read-Only Memory**). O programa inicializa todos os aspectos do sistema, desde os registos de **CPU**, controladores de dispositivos, conteúdo de memória, etc. Este programa tem que saber como carregar o sistema operativo e como começar a executá-lo. Para isto, o programa **bootstrap** localiza e carrega para memória o **kernel** do sistema operativo. O sistema operativo começa então a executar o primeiro processo e espera que um evento ocorra.

Os controladores têm alguma capacidade de processamento?

Têm capacidade de processamento. Tem registos. É o sistema operativo que trata da gestão.

Interrupções vs Polling

A ocorrência de um evento é normalmente sinalizado por uma **interrupção** que pode ser proveniente do **hardware** ou do **software**. O **hardware** pode desencadear uma interrupção a qualquer momento enviando um **sinal** ao **CPU**, normalmente pelo **bus** do sistema. O **software** pode desencadear uma interrupção executando uma operação especial chamada **system call**. Quando o **CPU** é interrompido, para o que está a fazer e transfere a sua execução imediatamente para a localização em específico. A interrupção deve transferir o controlo para o serviço de rotina da **interrupção**. O método directo para tratar este caso é invocar uma rotina genérica que examine a informação da interrupção; a rotina, por sua vez, chama o **handler** específico para tratar a interrupção. Uma vez que as interrupções devem ser tratados o mais rápido possível, existe uma tabela de apontadores para as rotinas de interrupção, fornecendo a velocidade de processamento desejada.

Os tipos de interrupções são os seguintes:

- **Hardware:** Geradas pelos dispositivos de **hardware** para sinalizar o facto de precisarem de 'atenção' do sistema operativo. Podem ter recebido dados (teclas do teclado, dados do cartão de **ethernet**, etc); ou apenas completaram uma tarefa que o sistema operativo tinha pedido anteriormente, como por exemplo, transferir dados entre o disco rígido e a memória.

- **Software:** Geradas pelos programas quando pretendem pedir uma chamada ao sistema para ser executada pelo sistema operativo. As interrupções de software também podem ser causadas por erros do programa em execução, como por exemplo, divisão por zero. Este tipo de interrupções são chamadas de *traps* ou *excepciones*.

O fluxo da invocação de uma interrupção é o seguinte:

1. O estado do programa que está atualmente a correr é guardado para posterior retoma de execução
2. O CPU muda para modo *kernel*/supervisor.
3. É localizado o código do *kernel* para tratar a interrupção através da tabela de *handlers* e do vetor de interrupção.
4. O código é executado.
5. O CPU volta ao modo utilizar e carrega o estado do programa que estava a executar.

Este sistema de interrupções é usado em alternativa ao sistema de *Polling*. Esta técnica é menos eficiente pois, ao contrário de receber uma interrupção, o CPU está sempre a perguntar aos dispositivos se precisam de alguma coisa. O problema disto é, se um dispositivo não precisa de nada, foi gasto tempo de CPU para nada.



Analogia: A técnica de *polling* pode ser equiparada ao facto de, a cada segundo, estarmos sempre a pegar no telemóvel para ver se estamos a receber uma chamada.

Tempo Virtual

Só conta enquanto executa o processo. Dependendo do sistema operativo pode contar em modo *kernel* ou não. Se houver *threads* é o somatório do tempo de cada uma.

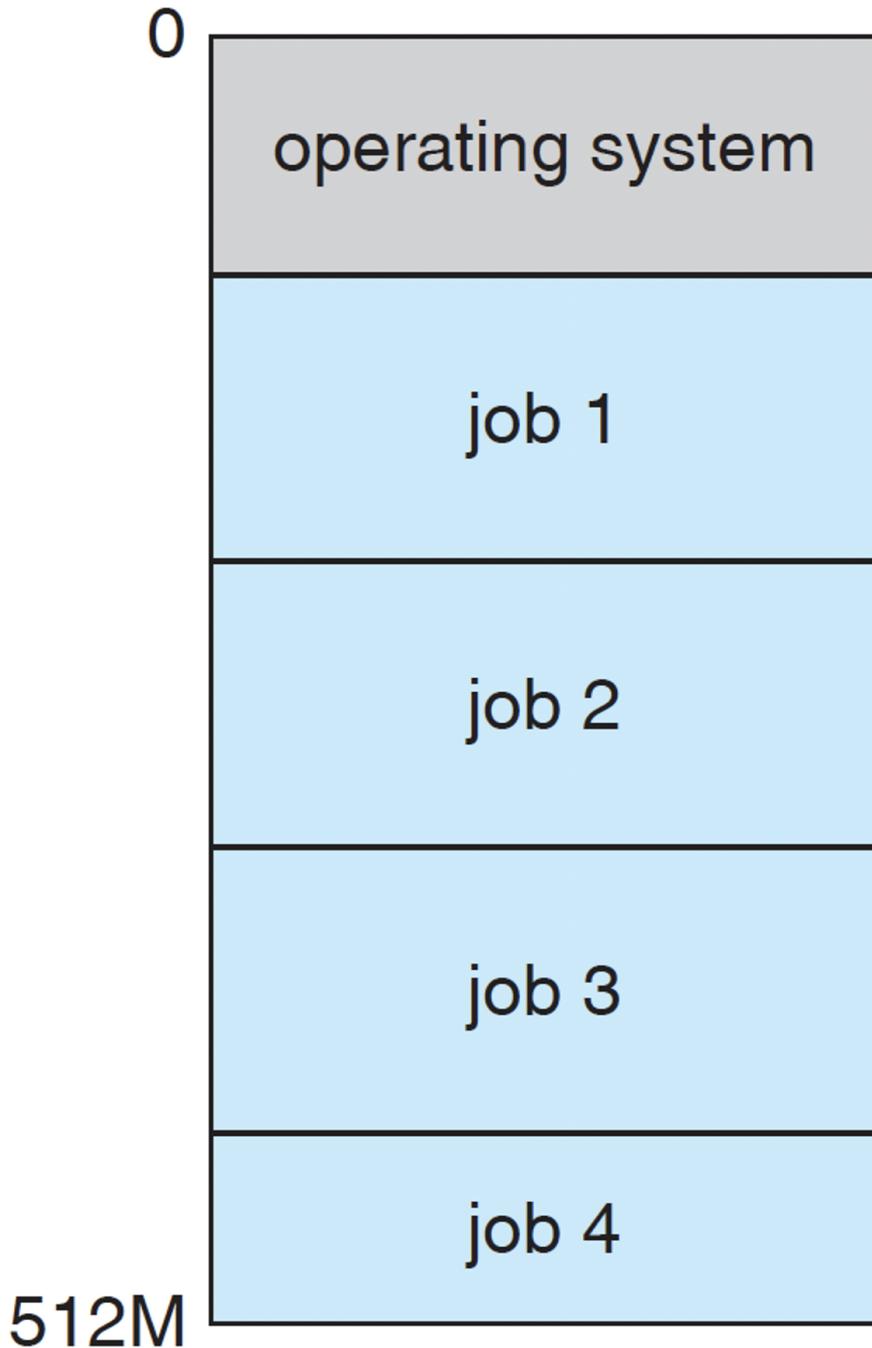
Multiprogramação

Um dos mais importantes aspectos dos sistemas operativos é a habilidade para suportar vários programas em execução ao mesmo tempo. Uma vez que, de um modo geral, apenas um programa não consegue ocupar o tempo inteiro o CPU ou os dispositivos I/O, a multiprogramação permite aumentar a utilização destes recursos, organizando os *jobs* (código + dados).

Uma vez que o tamanho da memória principal é demasiado pequeno para alocar espaço para todos os processos, o sistema operativo mantém em disco uma pool de processos (*job pool*), que contém os processos que aguardam alocação na memória principal.

O conjunto de *jobs* na memória principal pode ser um sub-conjunto daqueles que residem na *job pool*. Quando um programa necessita, por exemplo, de uma operação de I/O, ao contrário do que aconteceria num sistema de monoprogramação onde o CPU não fazia trabalho útil enquanto não fosse feita essa operação, num sistema de multiprogramação, este troca para outro processo que está à espera de ser alocado. Isto permite que, enquanto haja processos para executar, o CPU nunca está parado.

Time-sharing é um conceito lógico muito importante na multiprogramação pois os recursos são partilhados por vários processos. A troca de contexto entre os diferentes processos é tão rápida que os utilizadores não se apercebem que estão a partilhar recursos com outros processos.



Analogia: Um advogado nunca trabalha apenas para um cliente. Por exemplo, se estiver à espera de papelada ou à espera de ir a julgamento, ao contrário de estar parado, ele tem outros clientes para tratar.

A multiprogramação, para além de trazer vantagens e eficiência, requer mais cuidados a nível de gestão uma vez que agora existem vários programas a correr em paralelo. É necessário agora ter o seguinte em atenção:

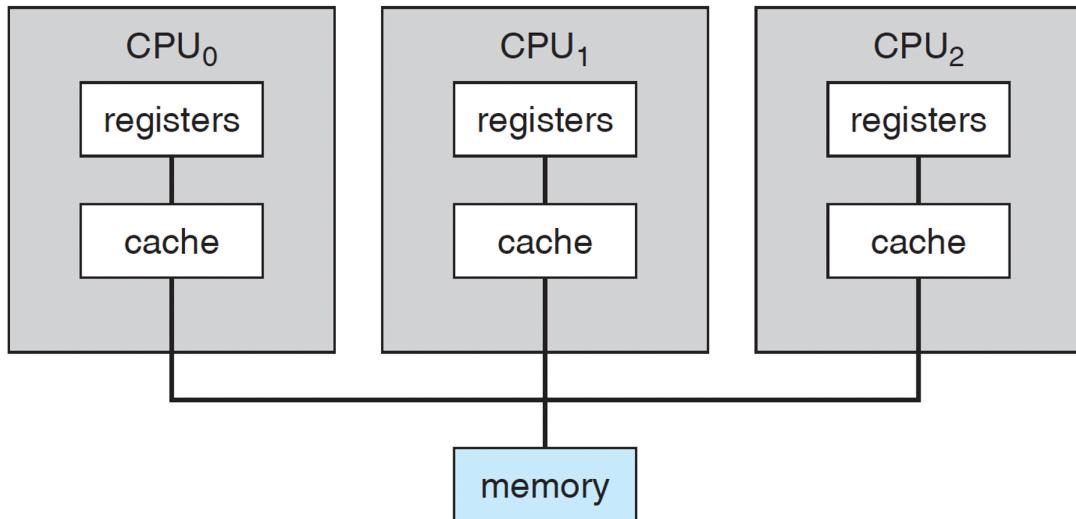
- **Job Scheduling:** a memória principal, dado o seu espaço limitado, pode estar cheia. É necessário que o sistema escolha um dos processos que estejam na job pool para o colocar em execução na memória principal.
- **CPU Scheduling:** Escolha de um processo para executar de entre os que estão na memória principal à espera de executar.
- **Gestão de memória:** limitar o espaço de cada processo e não deixar que um possa aceder/afectar o espaço do outro.

Multiprocessamento

Os sistemas com multiprocessadores são sistemas que têm dois ou mais processadores e que partilham o bus do computador, assim como memória, periféricos e o relógio.

Estes sistemas têm as seguintes vantagens:

- **Maior taxa de processamento:** aumentando o número de processadores é expectável que haja mais trabalho feito em menos tempo. No entanto, este aumento não é proporcional ao número de processadores pois existe overhead ao fazer com que as diferentes partes trabalhem correctamente. Este overhead juntamente com a partilha de recursos diminui o ganho proporcional que se esperava ter.
- **Economia:** os sistemas com multiprocessadores têm um custo mais baixo do que o equivalente a ter múltiplos processadores singulares pois é possível partilhar armazenamento, periféricos, etc.
- **Maior confiabilidade:** se as funcionalidades são distribuídas pelos diferentes processadores, a falha de um não afecta o funcionamento dos outros. Se tivermos 10 processadores e um deles falhar, o resto dos processadores conseguem ter acesso ao trabalho que já foi feito pelo que falhou e então o sistema corre apenas 10% mais lento. A propriedade de continuar a operar proporcionalmente ao nível do hardware que está funcional designa-se por *graceful degradation*.



Os sistemas multiprocessamento podem ser divididos em dois tipos:

- Assimétricos: um processador principal controla o resto dos processadores. O resto do processadores ou têm uma tarefa em específico ou lhes é atribuída uma tarefa pelo processador principal. Este tipo de multiprocessamento tem uma arquitectura *master-slave* onde o processador principal aloca e faz o escalonamento do trabalho para os outros processadores.
- Simétricos: cada processador pode executar qualquer tarefa.

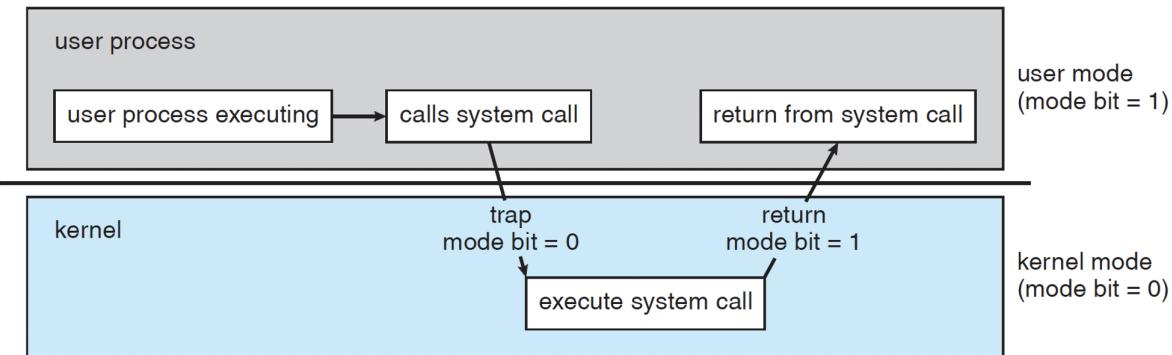
Modos de Execução

Para assegurar a correcta execução do sistema operativo, é necessário distinguir entre a execução de código do sistema operativo do código definido pelo utilizador. A forma como isto é feito é através do suporte de hardware. Um bit, designado por *mode bit*, foi adicionado ao hardware para distinguir estes dois modos de execução, *kernel* (0) ou *user* (1). Com este bit pode-se distinguir uma tarefa que é executada por parte do sistema operativo de uma executada por parte do utilizador.

Os modos de execução trazem segurança para o sistema pois é garantido que apenas uma parte do sistema, o *kernel*, aceda diretamente ao hardware. Doutra maneira não seria possível ter esta garantia. Desta maneira, o código do lado do utilizador está proibido de aceder diretamente ao hardware (ficheiros, memória, etc).

Em resumo:

- **Kernel mode:** instruções para gerir a memória e como pode ser acedida. Para além disso, também tem privilégios para aceder a periféricos como os discos e cartões de rede. As interrupções também são executadas neste modo.
- **User mode:** modo mais restrito onde apenas é possível aceder a certas zonas da memória e o acesso a periféricos não é possível.



O fluxo de uma chamada ao sistema, como se pode ver na imagem, é o seguinte:

1. O programa em *user mode*, quando invoca a chamada ao sistema, coloca os argumentos da mesma em registos ou numa *stack frame*, indicando que serviço pretende que seja executado pelo sistema operativo.
2. O programa em *user mode* executa a interrupção 'trap'
3. Imediatamente, o *CPU* guarda o estado do programa actual, muda para o modo *kernel* e salta para um sítio fixo da memória que contém as instruções da interrupção.
4. A rotina genérica de interrupção lê o serviço pretendido e os argumentos e executa a rotina de interrupção correspondente.
5. Quando a rotina acaba de ser executada, o *CPU* volta a colocar o *bit* a 1, indicando que voltou ao *user mode* e carrega o contexto do processo que estava a executar.

GESTÃO DE PROCESSOS

Programa: entidade estática que corresponde ao código executável de um programa. Um programa em si é apenas código em texto que está armazenado em disco.

Processo: entidade dinâmica que corresponde ao programa em execução. Um processo não é apenas o código do programa mas também todo o ambiente necessário para a sua execução. Um processo contém o *program counter*, conteúdo dos registos do *CPU*, *stack*, *heap*, etc.



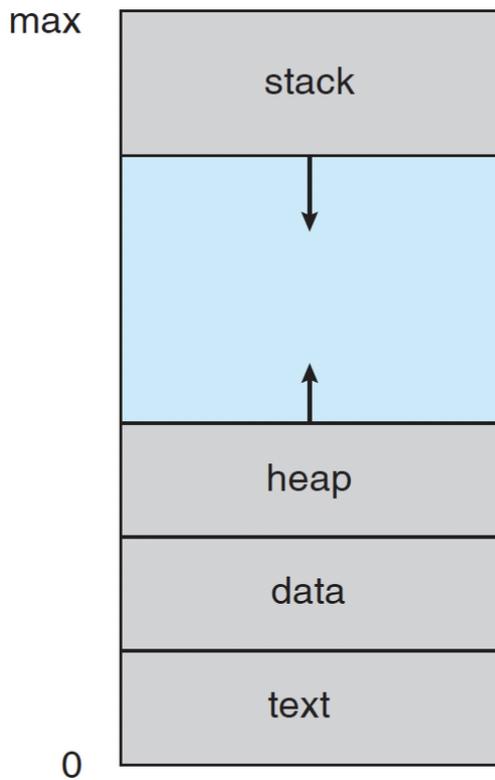
Nota: duas invocações do mesmo programa resultam em dois processos distintos

À medida que um programa executa vai variando o seu estado, sendo este definido pela sua atividade atual.

Conceitos de Processo

O processo

Como já foi referido, um programa não é um processo. Um processo é mais do que o código de um programa, pois necessita de um ambiente onde possa ser executado. Um programa torna-se um processo quando é carregado para memória principal onde o sistema operativo prepara esse ambiente para um processo ser executado.



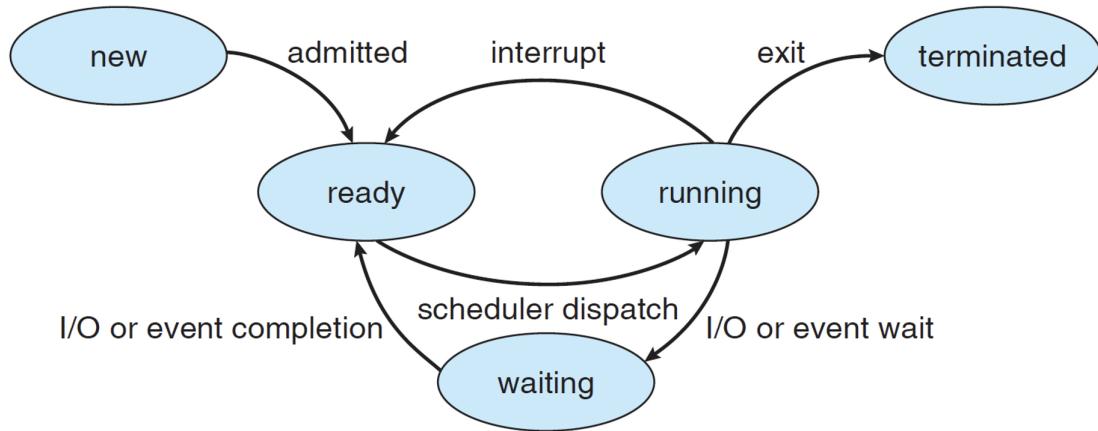
A imagem acima demonstra o conteúdo de um processo na memória principal:

- **Stack:** contém os parâmetros das funções, endereços de retorno, variáveis locais, etc.
- **Heap:** memória alocada dinamicamente.
- **Data:** contém variáveis globais, variáveis estáticas, etc.
- **Text:** código fonte do programa em execução

Estado de um processo

À medida que um processo executa vai variando de estado sendo os estados possíveis os seguintes:

- **New:** o processo está a ser criado
- **Running:** o processo está em execução
- **Waiting:** o processo está à espera que um evento ocorra, como por exemplo, uma operação de I/O.
- **Ready:** o processo está à espera de ser atribuído ao CPU.
- **Terminated:** o processo acabou a sua execução



Process Control Block (PCB)

Todos os processos são representados no sistema operativo por um **PCB**. Esta estrutura contém peças de informação associada a um específico processo servindo como um **repositório** que varia de processo para processo.

O **PCB** contém informação relativa a:

- **Estado do processo**: pode ser um dos estados referidos anteriormente.
- **Program Counter**: endereço da próxima instrução a ser executada para um determinado processo.
- **Registros de CPU**: variam conforme a arquitectura do computador contém acumuladores, índices dos registos, *stack pointers*, etc. Em conjunto com o **program counter**, esta é a informação que é guardada quando uma **interrupção ocorre**, para permitir que, quando o processo volta a ter tempo de **CPU**, volte ao estado em que estava.
- **Informação sobre o escalonamento de CPU**: contém informação sobre a **prioridade** do processo, apontadores para as **queries de escalonamento**, e qualquer outro tipo de **informação sobre escalonamento**.
- **Informação sobre a gestão de memória**: pode conter informação sobre os **valores da base** e **limite dos registos**, assim como informação sobre a **paginação**.
- **Accounting information**: **tempo de CPU usado**, **tempo real usado**, **limites de tempo**, **números do processo**, etc.
- **Informação sobre o estado I/O**: número de **dispositivos I/O alocados** para o processo, **lista de ficheiros abertos**, etc.

Escalonamento de Processos

- Escalonamento cooperativo(*non-preemptive*): uma vez atribuído o *CPU* ao processo, só lhe é retirado se entrar no estado de *wait* ou *terminated*.
- Escalonamento por desafetação forçada(*preemptive*): o *CPU* é retirado ao processo ao fim de um *time-slice* ou porque surgiu um de maior prioridade.

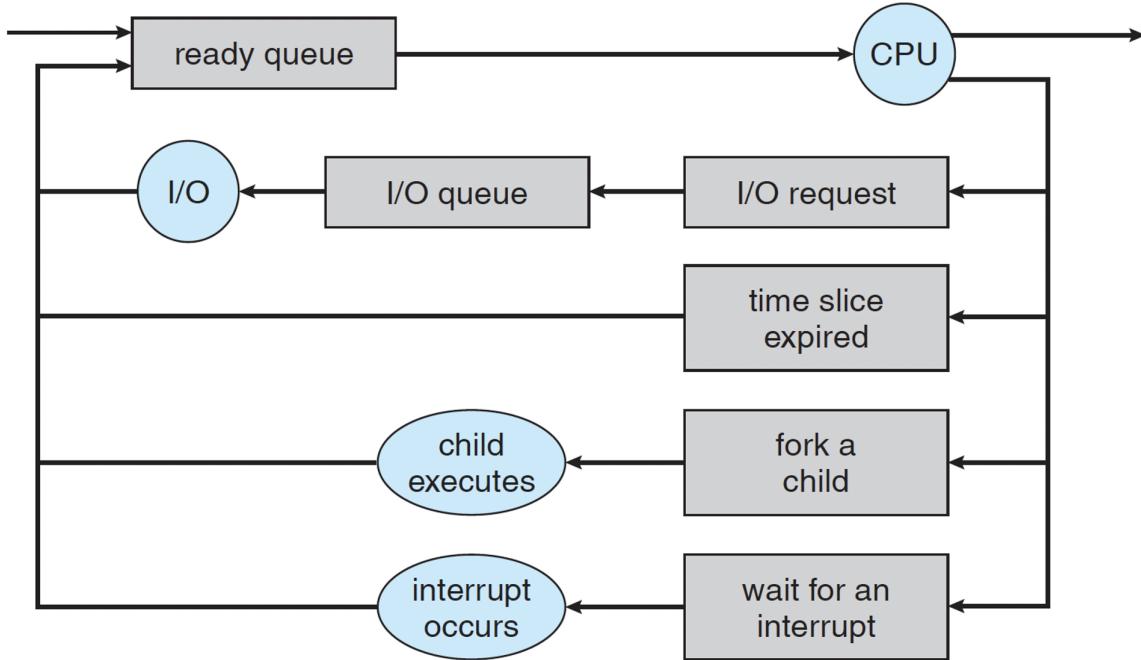
Deve-se usar desafetação forçada para evitar que interações longas monopolizem o *CPU*. Desta maneira, interações curtas terminam dentro de uma *time-slice* e as interações longas executam durante uma *time-slice* e, após este tempo, passam a estar no estado de *ready* dando lugar a outro processo. Mais tarde será-lhe atribuído novamente uma fatia de tempo, e assim sucessivamente.

- *Process scheduler*: escolhe um processo, de um conjunto de processos, para ser executado pelo *CPU*. Num sistema *single-processor*, nunca vai existir mais do que um processo a ser executado ao mesmo tempo. Se existirem mais processos, estes vão ter que esperar até que o *CPU* esteja livre.

Filas de escalonamento

À medida que os processos entram no sistema, são colocados numa *job queue*, que consiste em todos os processos do sistema. Os processos que residem na memória principal e que estão prontos ou à espera para serem executados são guardados numa *ready queue*, geralmente guardada como uma lista ligada onde a cabeça da lista aponta para o primeiro e último *PCB*.

Quando um processo está à espera de uma operação de *I/O*, e uma vez que existem vários processos que possam estar à espera do mesmo, existe também uma fila para os processos que esperam por um determinado dispositivo *I/O*, designada por *device queue*. Cada dispositivo tem a sua própria fila.



Um processo quando chega à memória principal é colocado na **ready queue** e fica à espera até ser seleccionado para execução. Quando um processo é alocado para o **CPU** podem acontecer os seguintes eventos:

- O processo pode pedir uma operação de **I/O** e é colocado na **I/O queue**.
- O processo pode **criar um sub-processo** e espera que este acabe.
- O processo pode ser **removido forçosamente** do **CPU** através de uma **interrupção** e ser colocado novamente na **ready queue**.

Um processo mantém-se neste ciclo **até acabar a sua execução**. Uma vez acabada, é **removido de todas as filas** e o seu **PCB** e recursos **desalocados**.

Em resumo, as filas e os respetivos escalonadores são:

- **Job queue**: processos à espera de serem escolhidos para irem para memória principal para serem executados.
 - **job scheduler (long-term scheduler)** : executado com menos frequência, controla o nível de multiprogramação
- **Ready queue**: processos em memória principal que estão no estado de **ready ou waiting**, à espera que sejam escolhidos para serem executados pelo **CPU**.
 - **cpu scheduler (short-term scheduler)** : executado com muita mais frequência, tipicamente uma vez a cada 100ms
- **Device queue**: processos à espera de serem atendidos por um dispositivo **I/O**.

Existem sistemas, como os *Unix* e *Microsoft Windows*, que não têm o *job-scheduler*, mas têm o designado *medium-term scheduler* que usa uma técnica designado por *swapping* onde os programas vão entrando e saindo da memória principal.

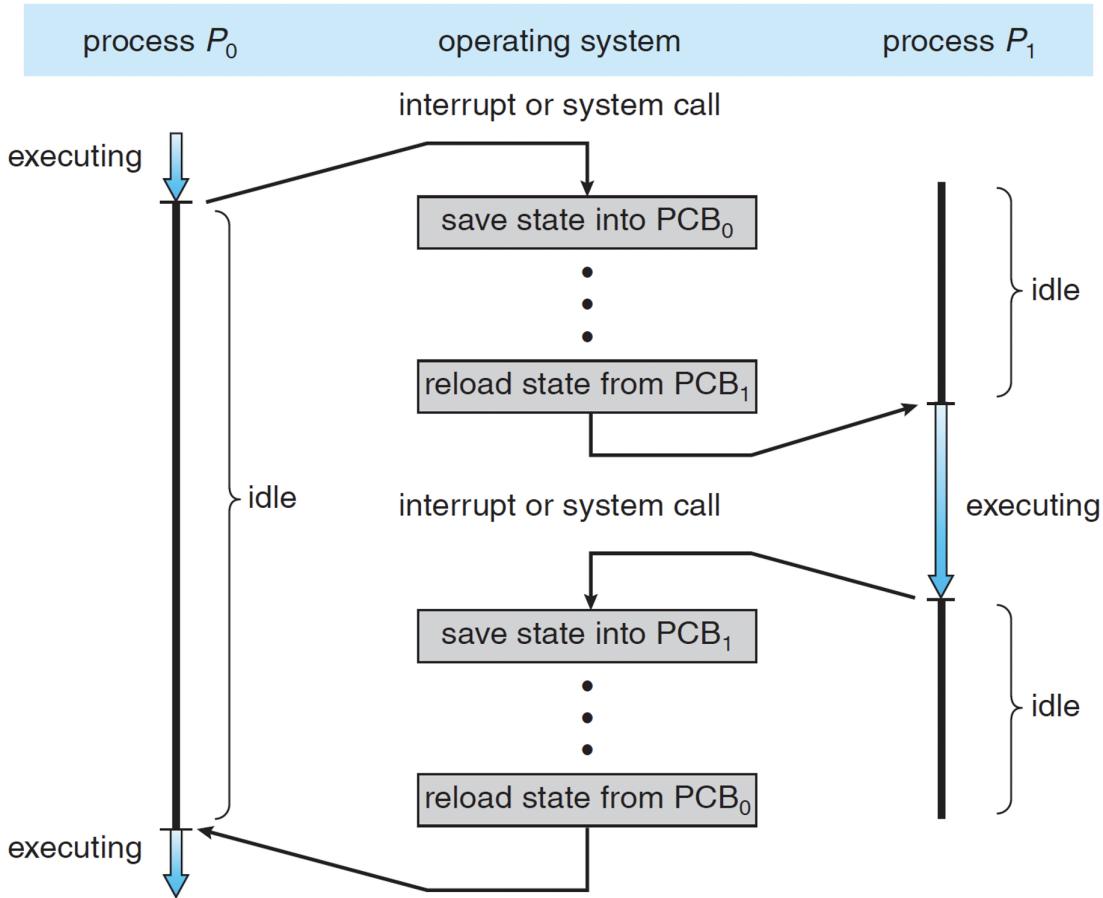
Critérios de escalonamento

Existem vários critérios de escalonamento:

- *IO-Bound / CPU-Bound*: dizem respeito a processos que, respetivamente, precisam de mais tempo de leituras e escritas e processos que precisam mais de poder computacional. O sistema é balanceado se forem escalonados uma mistura dos dois.
- Interativo ou não.
- Urgência de resposta.
- Comportamento recente (utilização de memória, *CPU*)
- Necessidade de periféricos especiais.
- 'Pagou' para ir à frente dos outros.

Mudança de contexto

As interrupções causam o sistema operativo a mudar a tarefa que o *CPU* estava a correr e correr uma rotina do *kernel*. Quando ocorre uma interrupção, o sistema precisa de guardar o contexto atual do processo em execução no *CPU* para depois ser restaurado uma vez que a rotina do *kernel* acabe, ficando o processo que estava a ser executado suspenso. O contexto é guardado no *PCB* do processo que estava a correr para depois ser restaurado. A mudança de contexto é puro overhead uma vez que não está a ser feito trabalho útil e a sua velocidade varia de sistema para sistema.



Algoritmos de Escalonamento de Processos

Os algoritmos de escalonamento de processos têm objectos diferentes, uns pretendem diminuir o tempo de resposta (diminuindo o tempo de espera para determinados processos) e outros tendem maximizar a utilização de CPU.

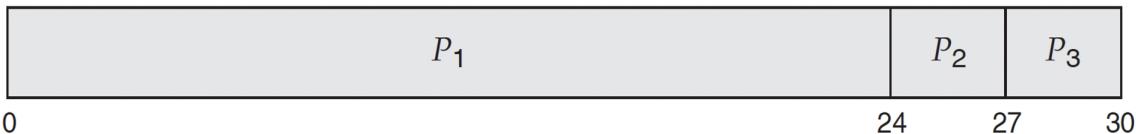
Os algoritmos a estudar são os seguintes:

- FCFS - *First Come, First Served*
- SJF - *Shortest Job First*
- SRTF - *Shortest Remaining Time First*
- PPS - *Preemptive Priority Scheduling*
- RR - *Round Robin*
- MFQ - *Multilevel Feedback Queue Scheduling*

Um conceito importante é o de **CPU-Burst** que representa o tempo consecutivo que um processo aguenta só a computar dados até necessitar de auxilio de I/O.

First Come, First Served Scheduling

- **Descrição :** O primeiro processo a pedir CPU é o primeiro processo a adquirir o mesmo. Quando um processo entra na ready queue, o seu PCB é colocado na cauda da fila, e então, a ready queue funciona como uma FIFO queue. Quando o CPU está livre, o processo que está à cabeça da fila é alocado.
- **Tipo :** cooperativo
- **Pontos positivos :** código fácil de escrever e de fácil leitura
- **Pontos negativos :** o tempo médio de espera com esta política de escalonamento é demorado. Problemático para sistemas de time-sharing e interativos. Tempo de espera com grandes flutuações dependendo da ordem de chegada.
- **Nota :** teria melhores resultados se houverem muitos mais processos I/O-bounded do que CPU-bounded.



Se a ordem de chegada for P1, P2 e P3, e uma vez que neste algoritmo são executados por ordem, o processo P1 tem tempo de espera 0, o P2 tempo de espera 24ms e o P3 27ms. O tempo médio de espera é de 17ms. É obvio que se a ordem não for esta e se for, por exemplo, P2, P3 e P1 o tempo médio de espera é de 3ms. No entanto, o tempo médio de espera não tende para valores mínimos.

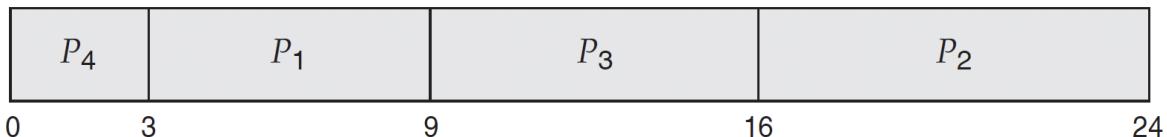
Shortest Job First Scheduling

- **Descrição :** A cada processo é associado o tamanho do próximo CPU-Burst. Quando o CPU está livre é associado ao processo mais curto. Caso haja empate, é usado o FCFS para decidir.
- **Tipo :** pode ser cooperativo ou desafectação forçada. A escolha é feita no momento em que o processo chega à ready queue. Quando é usado em modo de desafectação forçada é designado por Shortest-Remaining-Time-First Scheduling.
- **Pontos positivos :** é provado que seja ótimo, na medida em que fornece o tempo médio de espera mínimo para um dado conjunto de processos.
- **Pontos negativos :** não se consegue adivinhar o tamanho do próximo pedido do CPU, apenas se podem fazer estimativas.

- **Nota :** Não é usado no *CPU-Scheduler* pois não há maneira ao certo de saber quanto é o comprimento do próximo *CPU-Burst* mas é possível saber uma aproximação baseada nos valores anteriores. É usado no *Job-scheduler*.

Process Burst Time

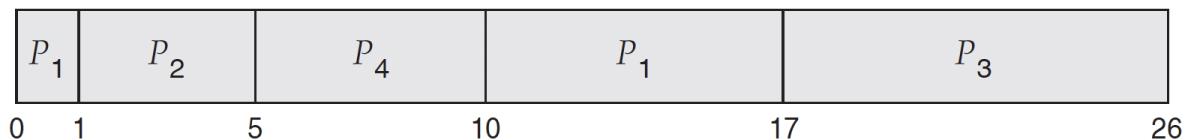
P_1	6
P_2	8
P_3	7
P_4	3



Em modo cooperativo, a fila estaria organizada como na imagem acima, por ordem de tempo, e a média de tempo de espera é 3ms enquanto se fosse usado o FCFS o tempo médio seria de 10.25ms. Ao mover os processos curtos para a frente dos processos mais longos, o tempo médio de espera baixa.

Process Arrival Time Burst Time

P_1	0	8
P_2	1	4
P_3	2	9
P_4	3	5



Em modo de desafectação forçada, se aparecer um processo com o tempo mais curto, é retirado o CPU ao processo que está a ser executado e atribuído ao de tempo com menor duração. No exemplo acima, o processo P1 é o primeiro a chegar e como não existe mais nenhum começa a ser executado. No entanto chega um processo P2 com o tempo mais curto do que o tempo atual do processo (que passou para 7 por exemplo), e por isso será atribuído o CPU a esse processo. De seguida, são executados por ordem crescente de tempo. O tempo médio de espera, neste caso, é de 6.5ms enquanto que se fosse cooperativo demoraria 7.75ms.

Priority Scheduling

- **Descrição :** A cada processo é associado uma prioridade e o *CPU* é alocado ao processo com maior prioridade. O algoritmo anterior é um caso especial deste algoritmo. Processos com a mesma prioridade são escalonados com o algoritmo *FCFS*.
- **Tipo :** pode ser cooperativo ou desafectação forçada.
- **Pontos negativos :** pode haver *starvation* se um processo com pouca prioridade for sempre ultrapassado por processos com maior prioridade. A técnica para resolver isto designa-se por *aging*, e o objectivo é aumentar a prioridade à medida que o tempo passa.
- **Nota :** As prioridades podem ser definidas interna ou externamente. As internas usam quantidades mensuráveis, como por exemplo, limites de tempo, requisitos de memória, número de ficheiros abertos ou a relação entre *CPU-Burst* e *I/O-Burst*. As externas não são definidas pelo sistema operativo e baseiam-se na importância do processo, ou a quantidade de dinheiro que se está a pagar para o uso de computador.

Process	Burst Time	Priority
P_1	10	3
P_2	1	1
P_3	2	4
P_4	1	5
P_5	5	2

P_2	P_5	P_1	P_3	P_4	
0	1	6	16	18	19

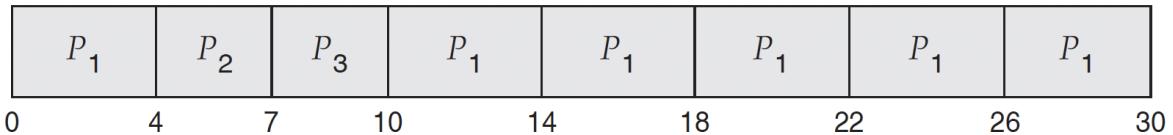
Se os processo P_1, \dots, P_5 chegarem todos no instante zero ao sistema, os processos serão escalonados como na imagem acima. Se o algoritmo for cooperativo, acaba o processo que está a executar e, de seguida, passa para o próximo com mais prioridade e, se entretanto chegar um ao sistema com maior prioridade é posto à cabeça da lista. Caso seja de desafetação forçada, o *CPU* abandona o processo que estava a executar e é alocado de imediato ao de maior prioridade.

Round-Robin Scheduling

- **Descrição :** É semelhante ao *FCFS* mas é *preemptive* para permitir que o sistema troque entre processos. Uma unidade pequena de tempo, designada por *time quantum* ou *time slice*, é definida, normalmente entre os 10-100ms. A *ready queue* é tratada como um fila circular *FIFO*. O *CPU Scheduler* anda à volta da fila alocando o *CPU* para cada processo durante um *time slice*. Novos processos são adicionados à cauda da fila e o escalonador escolhe o processo que está à cabeça da mesma. Pode acontecer uma de duas coisas, ou o processo acaba antes de 1 *time slice* e o processo liberta o *CPU* e é retirado o próximo processo da cabeça da lista, ou o processo não acaba dentro deste intervalo de tempo e uma interrupção é lançada ao sistema operativo. É executada uma troca de contexto, o processo é colocado na cauda da lista e o *CPU* é alocado para o processo que está na cabeça da lista.
- **Tipo :** desafectação forçada
- **Pontos positivos :** Fácil de implementar
- **Pontos negativos :** O tempo de espera é normalmente longo
- **Nota :** Se o *time slice* for muito grande então o algoritmo tem um comportamento semelhante ao *FCFS*. Se for muito pequeno tem-se *overhead* de mudanças de contexto, degradando os níveis de utilização do *CPU*. Cada um dos n processos *CPU-Bound* terá $1/n$ do tempo disponível no *CPU*.

Process Burst Time

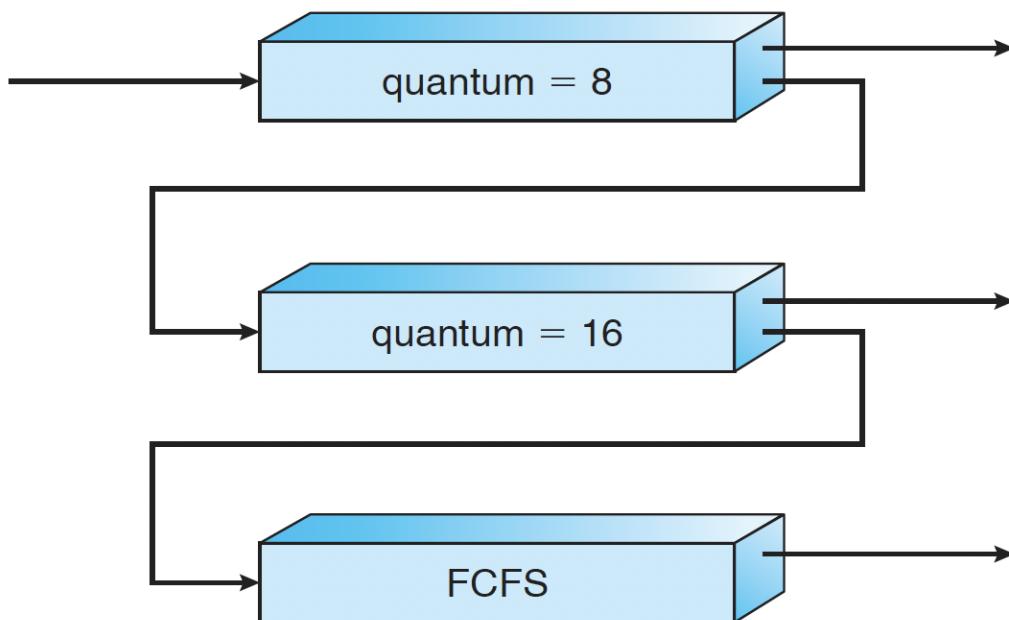
P_1	24
P_2	3
P_3	3



Definindo o *time slice* em 4ms, o processo P_1 corre durante 4s, esgotando o tempo sobrando ainda 20ms de execução. O processo P_1 é colocado na cauda da fila e, de seguida, à mudança de contexto para o processo P_2 . Tanto o processo P_2 e P_3 , acabam dentro das *time slices* correspondentes. Por último, o processo P_1 é finalizado em 5 *time slices*, mas, como é o único processo que sobra, não há troca de contexto. O tempo médio de espera é de 5.66ms.

Multilevel Feedback Queue Scheduling

- **Descrição :** Classe de algoritmos de escalonamento em que os processos podem ser classificados em diferentes grupos, por exemplo, processos a correr em *background(Batch)* ou processos a correr em *foreground(interactive)*. Parte a *ready queue* em várias filas separadas. A ideia da separação dos processos pelas filas tem como base o *CPU-Burst*. Se um processo necessita de muito tempo de *CPU* é adicionado a uma fila com baixa prioridade, deixando processos interativos e *I/O-Bounded* em filas com maior prioridade. Um processo que esteja muito tempo numa fila com baixa prioridade é movido para uma fila com maior prioridade usando a técnica anteriormente mencionada, *aging*.
- **Tipo :** desafectação forçada ou cooperativo
- **Pontos positivos :** prioridade de processos
- **Pontos negativos :** complexa implementação
- **Nota :** Dá mais prioridade aos processos com *CPU-Burst* $\leq 8\text{ms}$



Imaginemos um *multilevel feedback queue scheduler* com três filas numeradas de 0 a 2. O escalonador primeiro executa todos os processos contidas na fila 0. Só quando a primeira fila se encontra vazia é que vai para a segunda e, só quando as duas primeiras se encontram vazias é que vai para a terceira. Um processo que chegue à fila 0 vai antecipar um processo que chegue à fila 1, e um processo que chegue à fila 1 vai antecipar um processo que chegue à fila 2.

Um processo quando chega ao sistema é colocado na fila 0 e é-lhe dado um *time-slice* de 8ms. Se o processo não terminou dentro desse intervalo de tempo é colocado na cauda da fila 1. Se a fila 0 estiver vazia, o *CPU* é alocado ao processo que está à cabeça da fila 1 e é-lhe dado um *time-slice* de 16ms. Se não terminar, é colocado na cauda da fila 2. Os processos na fila 2 são executados com base no *FCFS*.

Tipicamente, este tipo de algoritmos é definido com os seguintes parâmetros:

- Número de filas
 - Algoritmo de escalonamento para cada fila
 - Método usado para quando um processo sobe onde desce de nível
 - Método usado para determinar a fila a que um processo pertence
-

NOÇÕES DE PROGRAMAÇÃO CONCORRENTE

O sistema operativo tem a responsabilidade de fornecer mecanismos de interação entre processos. Há normalmente dois padrões de interação entre processos:

- Cooperação : cooperam entre si para atingir um resultado em comum.
- Competição: competem por recurso (*CPU*, memória, impressora, etc).

Os casos anteriores são exemplos de sincronização, isto é, atraso deliberado de um processo até que determinado evento surja. Convém que este atraso seja passivo e não activo para não desperdiçar tempo de *CPU*.

Para haver interação tem que haver comunicação, que pode acontecer através de ficheiros, *pipes*, *sockets*, memória partilhada, etc.

- Comunicação : escrita/leitura de dados passados através de vários mecanismos (ficheiros, memória partilhada, ...)
- Sincronização : coordenação de eventos entre processos (usando semáforos, variáveis de condição, *locks*, etc) para garantir ordem de eventos - "só leio depois de tu escreveres"

Mecanismos de Sincronização

Exemplos de mecanismos de sincronização:

- Semáforos
- Exclusão mútua (*mutexes*, métodos *synchronized*)
- *Wait/signal/notify*

Semáforos

Servem para resolver problemas de sincronização, exclusão mútua e controlo de capacidade e são definidos por três operações:

- Inicialização : `s = cria_semaforo(valor_inicial)`
- `P(s)` ou `Down(s)`
- `V(s)` ou `Up(s)`

 **Analogia:** Imaginemos uma caixa cheia de bolas. Operação P : se há bolas na caixa, retiro uma e continuo, senão aguardo passivamente que alguém deposite uma; Operação V : devolvo a bola à caixa, se há alguém bloqueado à espera, acordo-o.

O pseudo-código de cada uma das operações é o seguinte:

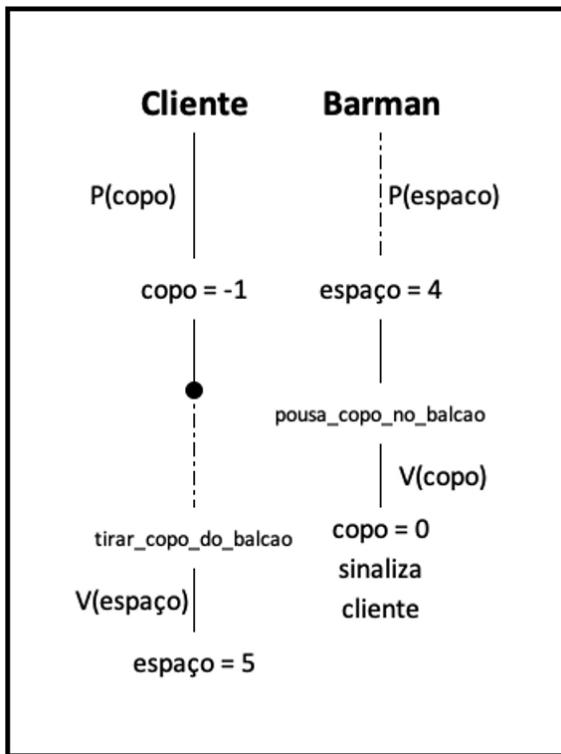
```
P(s){ s = s - 1 if( s < 0 ) bloqueia(s) } V(s){ s = s + 1 if( s <= 0 ) liberta(s) }
```

Para exemplo, vai ser usado o problema de consumidor e cliente. Estão definidos dois semáforos para o exemplo, copo e espaço. O valor inicial do semáforo copo é de 0 e do espaço é a capacidade do balcão.

```
Cliente{ P(copo) tira_copo_do_balcao() V(espaço) } Barman{ P(espaço) pousar_copo_no_balcao() V(copo) }
```

Assumindo que o espaço do balcão é 5, que corresponde ao valor inicial do semáforo espaço, e que no início não há copos, que corresponde ao valor inicial do semáforo copo, o comportamento pode ser o seguinte:

- Chega um cliente e pede um copo, baixando o valor do semáforo em 1, ficando a -1.
- Como é negativo, o cliente fica à espera.
- O barman retira um valor ao espaço do balcão ficando com o valor de 4.
- O barman coloca um copo no balcão.
- O barman indica que existe um copo, por isso o copo fica com o valor de 0 e liberta o cliente que estava à espera.
- O cliente prossegue e tira o copo do balcão.
- O cliente aumenta em 1 o espaço no balcão.



O algoritmo acima garante sincronização mas não exclusão mútua. Para isso é necessário criar um semáforo para o balcão, designado por *mutex*, com o valor inicial de 1, e variáveis *c* e *p* inicializadas a 0. Com isto, o código fica da seguinte maneira:

```

Cliente{ P(copo) P(mutex) cx = buf[c++ % N] V(mutex) V(espaço) }
Barman{ P(espaço) P(mutex) buf[p++ % N] = px V(mutex) V(copo) }

```

Ao começar o semáforo *mutex* com o valor de 1, é garantido que pelo menos um cliente ou barman acedem ao balcão. Se houver um segundo a querer aceder fica preso no *mutex*. Numa situação em que existem três copos no balcão, então o semáforo copo é igual a 3 e o semáforo espaço igual a 2. Quando um cliente pretende adquirir um copo, não fica preso no *P(copo)* pois existem copos disponíveis. Também não fica preso no semáforo *mutex* porque assumimos que neste momento é o único cliente. No momento que o cliente está a pegar no copo, entre *P(mutex)* ... *V(mutex)* aparece outro cliente. Esse cliente não fica preso no semáforo copo pois ainda existem 2 copos. No entanto, vai ficar preso no *mutex* pois já existe alguém no balcão a pegar no copo. Quando o primeiro cliente liberta o balcão, o segundo é libertado e entra no balcão. Desta maneira garante-se que o balcão é mutuamente exclusivo.

GESTÃO DE MEMÓRIA

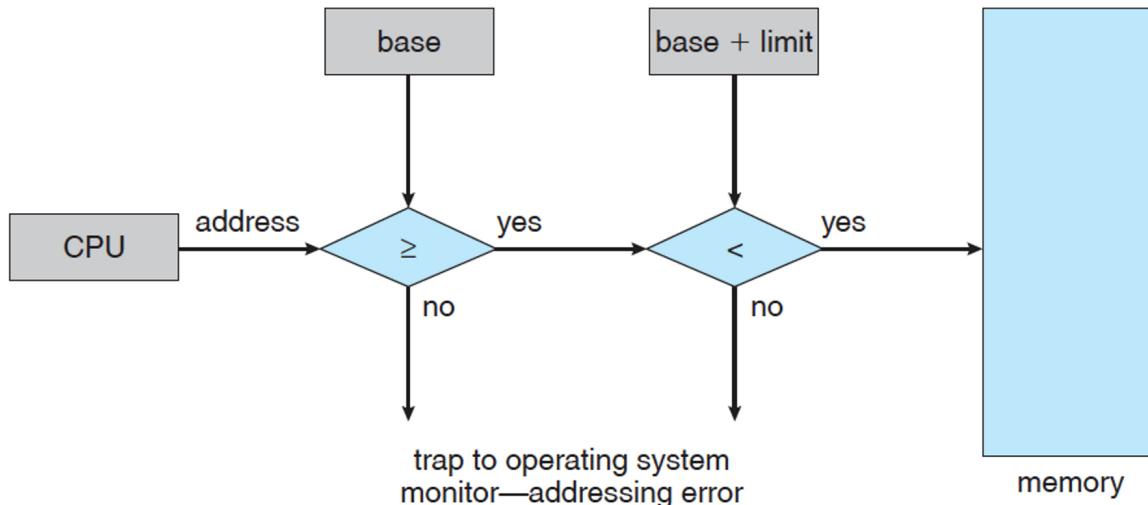
O ponto essencial da gestão de memória é o conceito de memória virtual. Para isso, numa primeira fase irão ser explicadas algumas estratégias de gestão de memória que, posteriormente, serão usadas para implementação da memória virtual. É de notar que o conceito de memória virtual não quer dizer que temos mais memória do que aquela que a máquina fornece, é apenas uma estratégia que permite, com a memória existente, simular uma memória maior do que aquela que é fornecida. Ou seja, a memória virtual, não é uma memória física, mas sim uma memória conceptual.

É muito importante, por razões de segurança, proteger o acesso ao sistema operativo por parte de processos do utilizador, bem como proteger os processos de utilizadores um dos outros. Esta proteção é fornecida pelo hardware.

É necessário garantir que cada utilizador tenha o seu espaço de endereçamento e, para isso, é necessário definir quais os endereços válidos que um processo pode aceder e garantir que um processo apenas tem acesso a esses endereços. Esta garantia pode ser assegurada através de dois registos, um base e um limite. O endereço base é o endereço válido mais baixo do programa, onde o mesmo começa. O limite é a gama de endereços válidos desse programa.

Base = 300040 Limit = 120900 Range = [300400, 420939] = (endereços válidos de um processo)

A proteção do espaço de endereçamento é garantida fazendo com que o CPU verifique todos os endereços gerados pelos programas em user mode. Apenas o sistema operativo consegue aceder aos registos de base e limite pois tem acesso ao kernel mode e, com isso, consegue também definir esses mesmos registos. Sempre que um programa em user mode tenta aceder a endereços fora do seu espaço de endereçamento, é lançada uma trap ao sistema operativo.

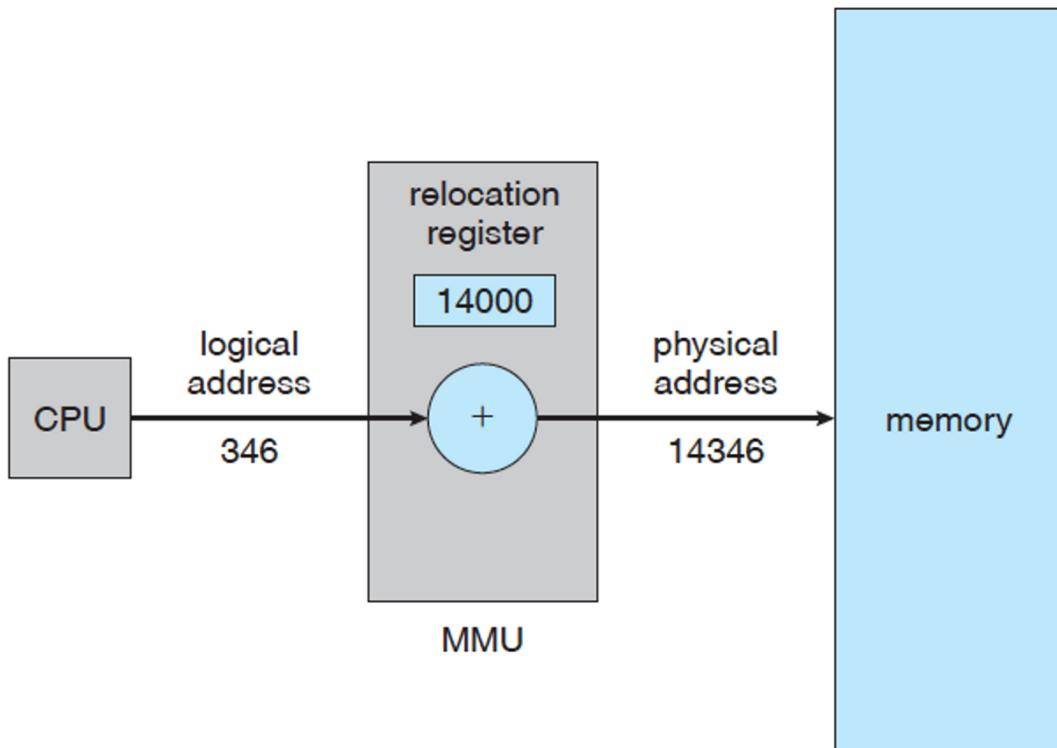


Espaços de Endereçamento Lógicos e Físicos

Um conceito importante neste contexto é o de *compile-time binding-address*. Um programa compilado usando o método de *binding compile-time*, gera endereços lógicos, não físicos, isto é, os endereços gerados no processo de compilação não são os endereços relativos à memória principal (endereços físicos) mas sim relativos ao inicio do programa (endereços lógicos). Ou seja, o endereço gerado é relativo à distância ao seu registo base.

Os endereços gerados pelo programa definem o espaço de endereçamento lógico. O conjunto dos endereços físicos correspondentes aos endereços lógicos definem o espaço de endereçamento físico. Os espaços de endereçamento físicos e lógicos diferem em tempo de execução.

O mapeamento de endereços lógicos para físicos é feito através de uma componente hardware designada por **Memory-Management Unit (MMU)**.



Se o programa tentou endereçar algo para o endereço 0, no caso da imagem acima, é imediatamente realocado para o endereço 14000. Um acesso ao endereço 346 é imediatamente realocado para o endereço 14346. O programa do utilizador nunca usa os endereços físicos efetivos.

Isto tem uma vantagem enorme, vários programas podem gerar exatamente os mesmo endereços, por exemplo, quando se faz um *fork*, mas, através deste nível de indireção dos endereços não haverá colisões sendo que cada processo irá ter o seu próprio espaço de endereçamento físico.

Com esta técnica, é também possível que o sistema operativo varie, dinamicamente, de tamanho, não afetando os outros programas pois, esses mesmos, não dependem de um endereçamento directo para a memória, isto é, se os endereços gerados por um programa fossem estáticos, se mapeassem sempre para a mesma zona da memória central, caso o sistema operativo aumentasse de tamanho, os programas tinham que ser recompilados para obterem um novo mapeamento direto para a memória. Com o conceito de realocação, isso não é necessário pois, os programas geram endereços lógicos e, caso o sistema operativo cresça, é só preciso mudar o registo base do programa.

Em resumo, existem dois tipos de endereçamento, lógico ([0,MAX]) e físico ([R, R + MAX], onde R é um endereço base). O utilizador apenas gera endereços lógicos e pensa que o processo corre na gama de valores de 0 a MAX. Estes endereços têm que ser realocados antes de serem usados.

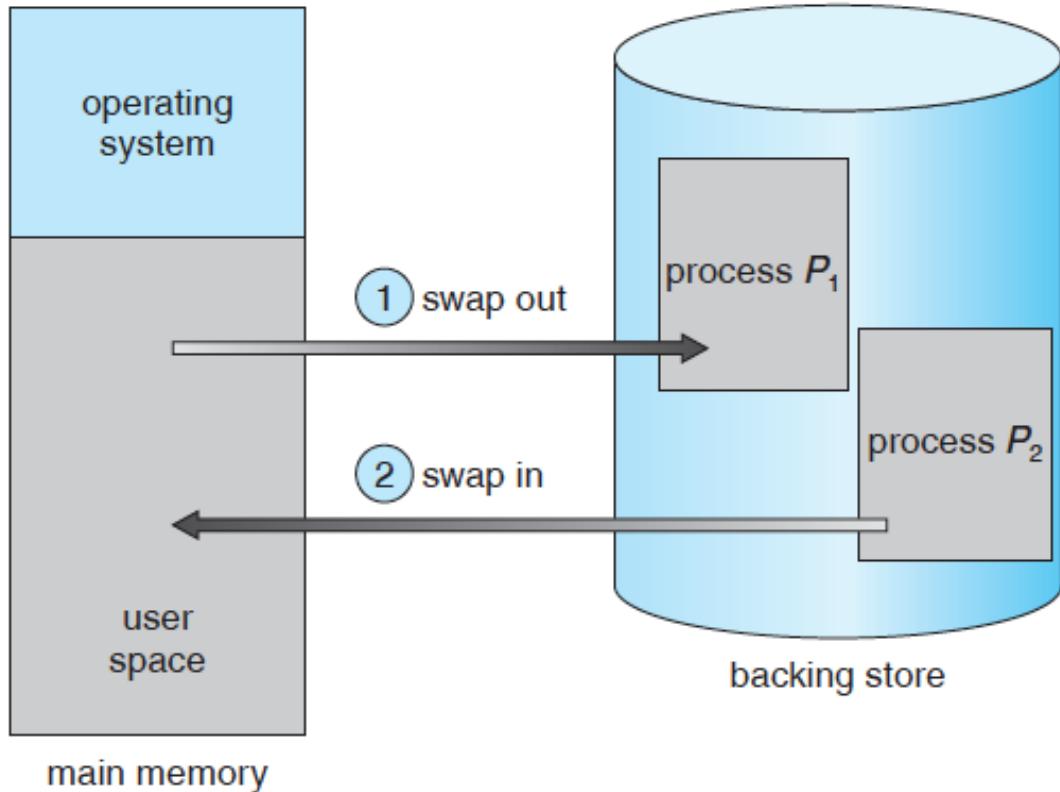
Swapping

Um processo pode ser retirado da memória principal e, posteriormente, voltar a ser colocado na mesma para acabar a sua execução. Isto pode acontecer quando a memória principal não tiver mais espaço para processos. Por exemplo, imaginando o algoritmo de escalonamento baseado em prioridades, se a memória estiver cheia e aparecer um processo com maior prioridade, um processo com prioridade mais baixa poderá ser removido da memória para dar lugar a esse processo. Os processos ao saírem da memória, são guardados numa *backing store*. Um processo quando volta à memória principal, pode não ser atribuído ao mesmo espaço de endereçamento físico anterior, mas isto não será um problema porque, como já foi visto, pode-se tirar partido dos endereços lógicos para a realocação. Quando o escalonador de CPU decide executar um processo é chamado o *dispatcher*. O *dispatcher* verifica se o processo pretendido está na memória, e caso não esteja e esta estiver cheia, é retirado um processo da memória para dar lugar a este novo processo.

Nem todos os processos podem fazer swap. Imaginemos que existe um processo que está à espera de I/O mas a fila de espera de um determinado dispositivo de I/O está muito ocupada. Se removermos o processo da memória e colocarmos lá outro, a operação de I/O, quando concluída, irá tentar usar a memória desse novo processo. Um processo quando é removido da memória principal tem que estar *idle*.

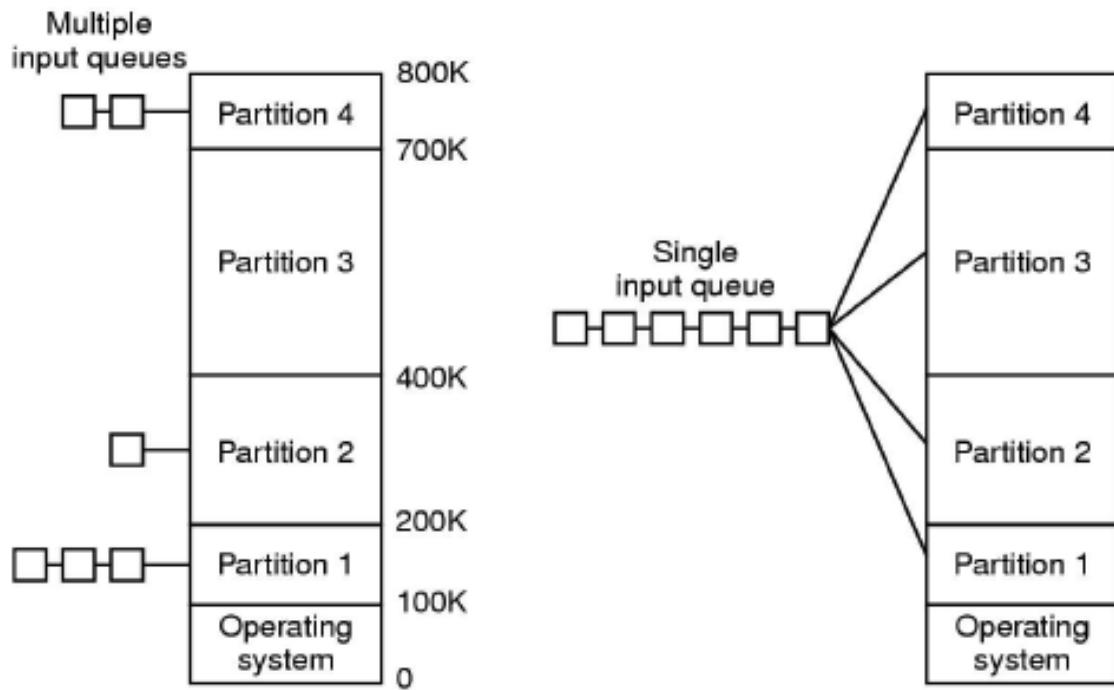
É de notar que o tempo de troca de contexto é muito elevado, sendo a maior parte do tempo perdido em transferência e, por isso, não é desejado fazer muito swapping. O swapping está normalmente desativado nos sistemas mas será ativado se muitos processos estiverem a usar muita memória principal. Será desativado outra vez quando a carga do sistema baixa.

- Swap out : remove um processo da memória principal.
- Swap in : adiciona um processo à memória principal.

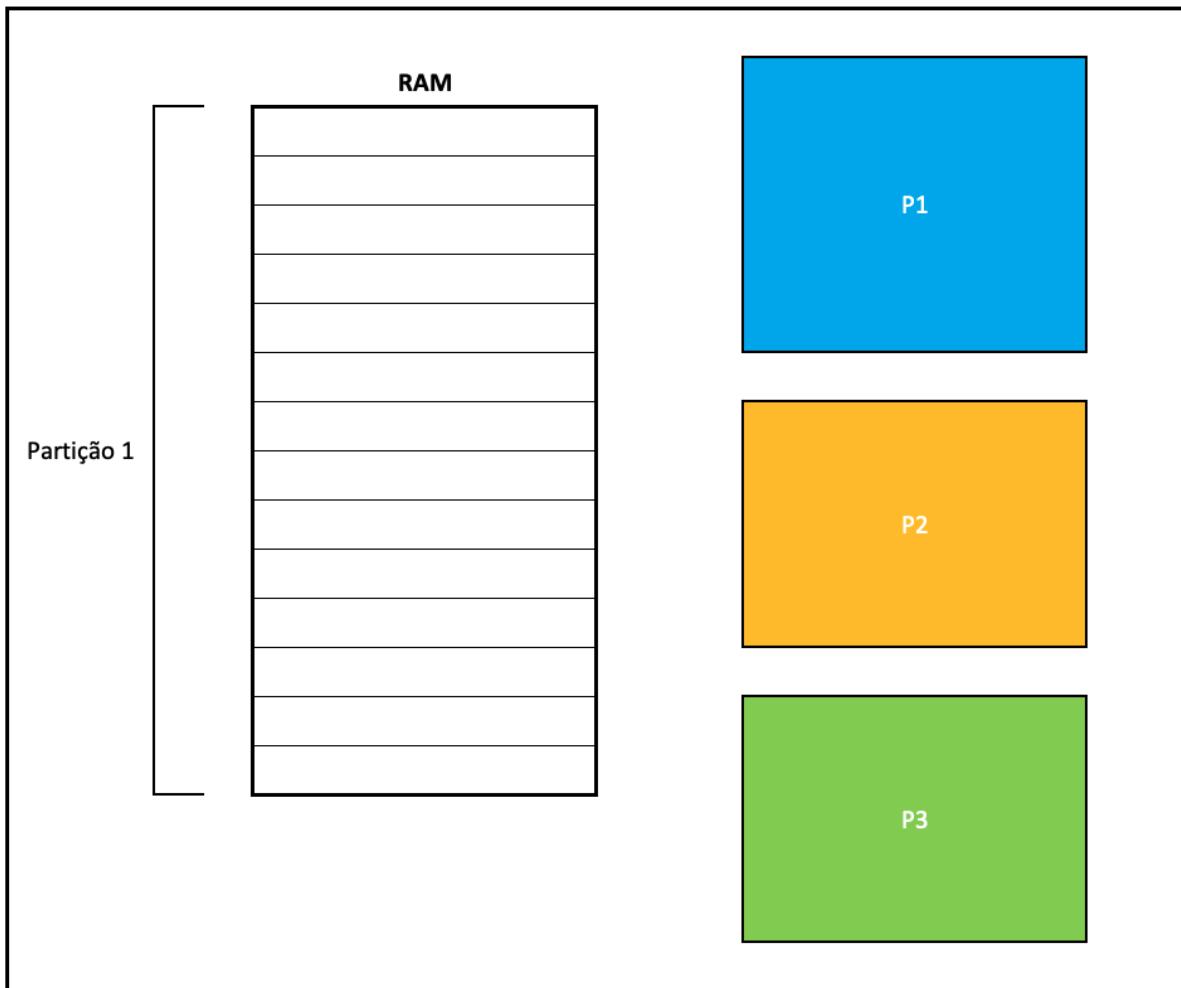


Alocação de Memória

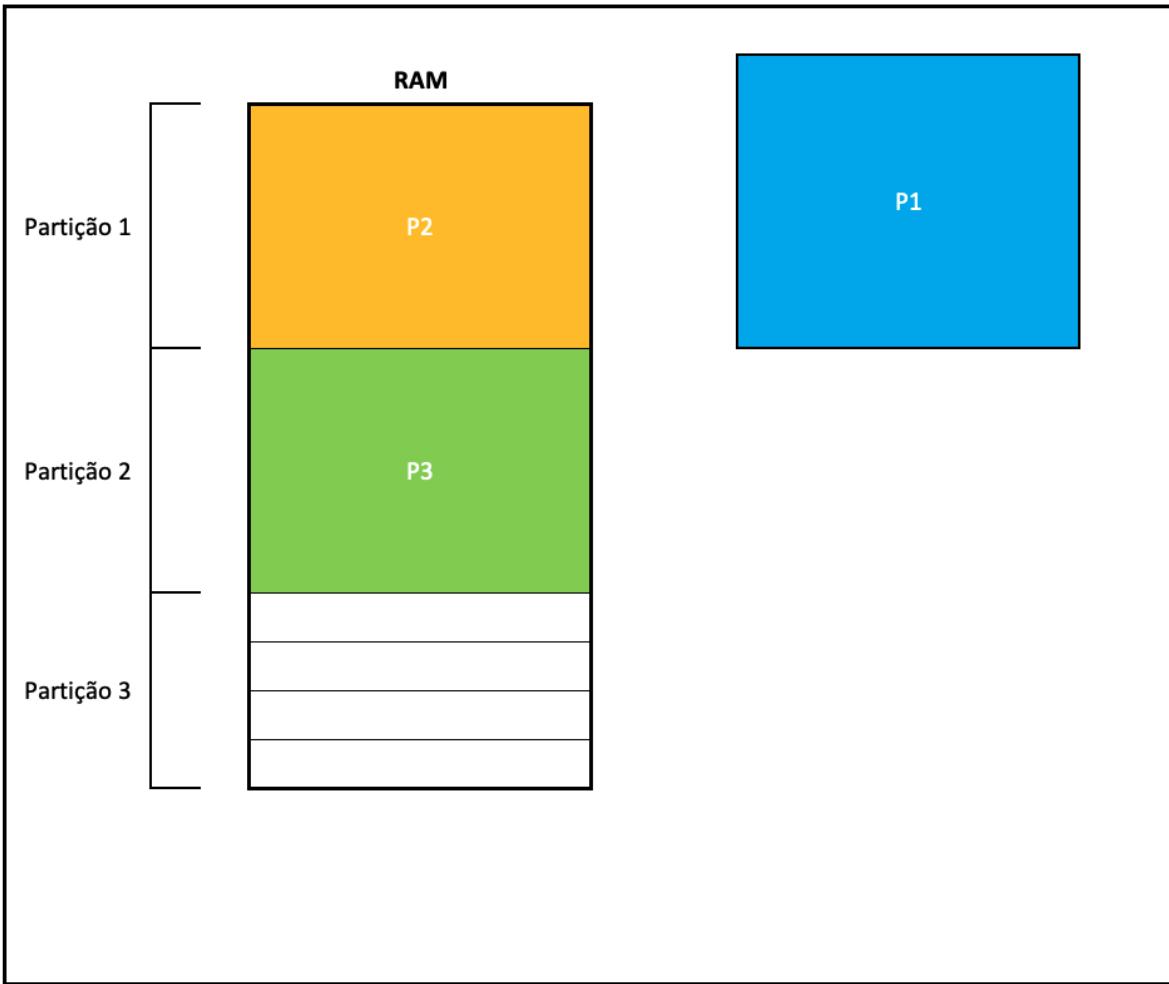
Um dos métodos simples para alocar memória é dividir a memória em partições fixas em que cada partição contém apenas um processo. Dado isto, o nível de multiprogramação, isto é, o número de processos que podem estar na memória, é limitado pelo número destas partições. Neste método, quando uma partição está livre, um processo é selecionado e colocado numa partição livre. Quando um processo termina, a partição correspondente fica livre. Este método já não é usado porque não é eficiente.



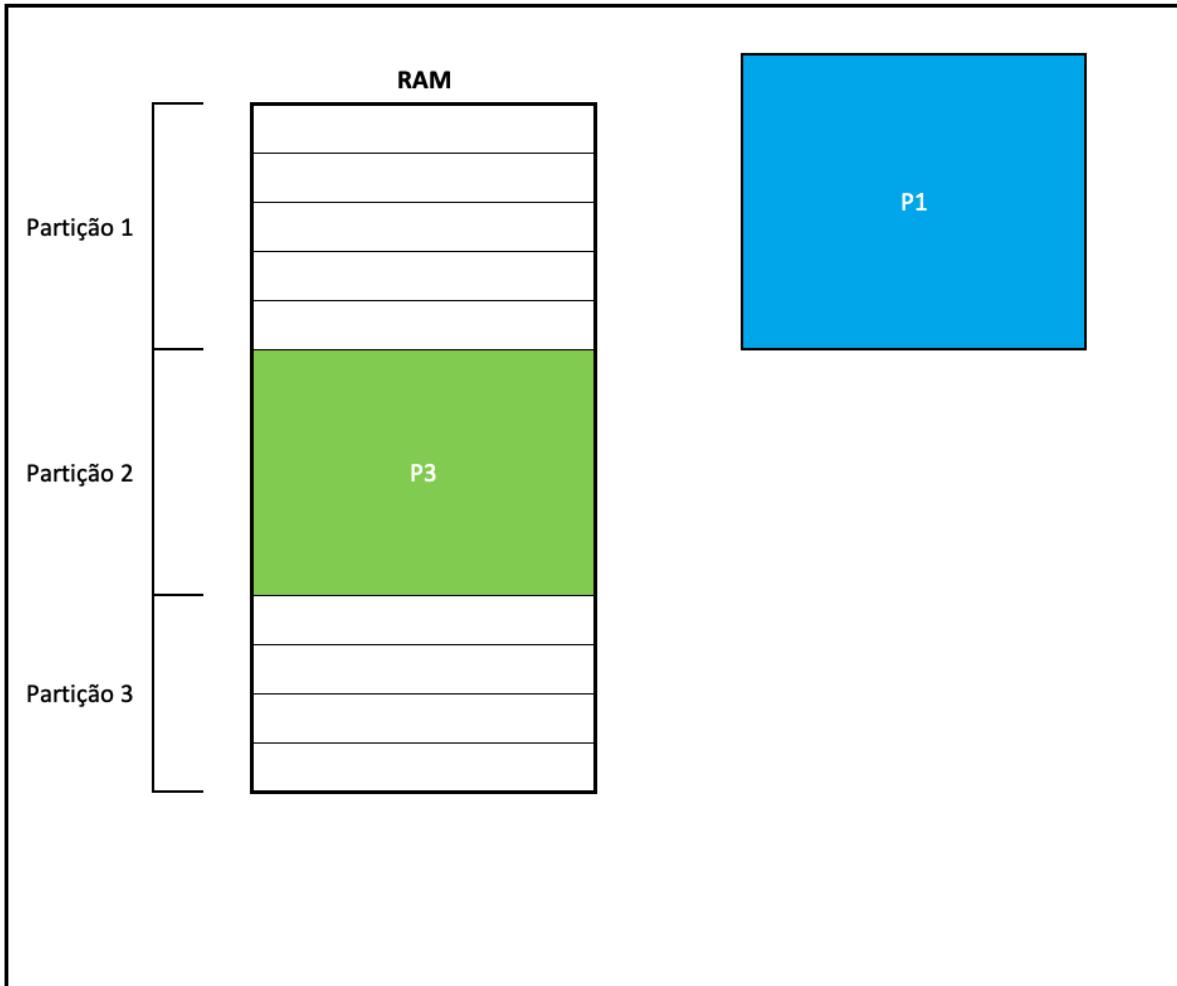
No esquema de partição variável, o sistema operativo mantém uma tabela que indica quais as partições que estão livres e as que estão ocupadas. Num estado inicial, a memória é vista como uma partição apenas, um bloco gigante de memória disponível, como se vê na imagem abaixo. O processo P1 ocupa 6 blocos, P2 5 blocos e P3 5 blocos.



A nossa RAM é composta por uma partição no estado inicial, com 14 blocos. Tem apenas uma partição porque ainda não foram alocados processos. Agora imaginemos que os processos P2 e P3 são alocados na memória. O estado desta mesma é o seguinte.



Uma tabela no sistema operativo irá conter a informação de que a partição 1 está ocupada bem como a partição 2, e que a partição 3 se encontra livre. No entanto, o tamanho dessa partição não permite que o P1 seja alocado pois não existe espaço suficiente para o alocar pois tem-se 4 blocos livres na *RAM* e o processo P1 ocupa 6 blocos. Quando o processo P2 termina a sua execução, é removido da *RAM* e o espaço de memória livre aumenta, deixando a partição 1 livre.



Embora haja agora espaço livre suficiente na memória, o processo P1 continua a não puder ser alocado pois não existe espaço contíguo na memória para o alocar. A partição 1 tem 5 blocos livres e a partição 3 tem 4 blocos livres, no entanto o processo P1 tem 6 blocos. Este problema é designado por **fragmentação de memória**.

Em síntese, a alocação de memória tem o seguinte fluxo:

- Os processos são colocados numa **fila de espera para obterem CPU**.
- O sistema operativo tem em conta os **requisitos de memória** de um processo e o **espaço disponível** na memória principal.
- Os processos vão sendo alocados na memória, nos espaços livres, formando uma partição.
- À medida que os processos **terminam**, **deixam a partição livre** para outro processo.
- Se existirem **duas partições consecutivas livres**, é feito **merge** às partições.
- O sistema operativo vai escolhendo os processos que cabem nas partições, sendo que os **processos que não cabem**, têm de **esperar**, procurando na fila processos que cabem na partição.

Estratégias para atribuir uma partição a um processo:

- *First-fit* : Alocar o primeiro buraco que é largo suficiente para um processo.
- *Best-Fit* : Alocar o buraco mais pequeno que é grande suficiente para um determinado processo. É necessário percorrer a lista toda dos buracos disponíveis.
- *Worst-Fit* : Alocar o maior buraco ao processo. Mais uma vez, é necessário percorrer a lista toda de buracos disponíveis.

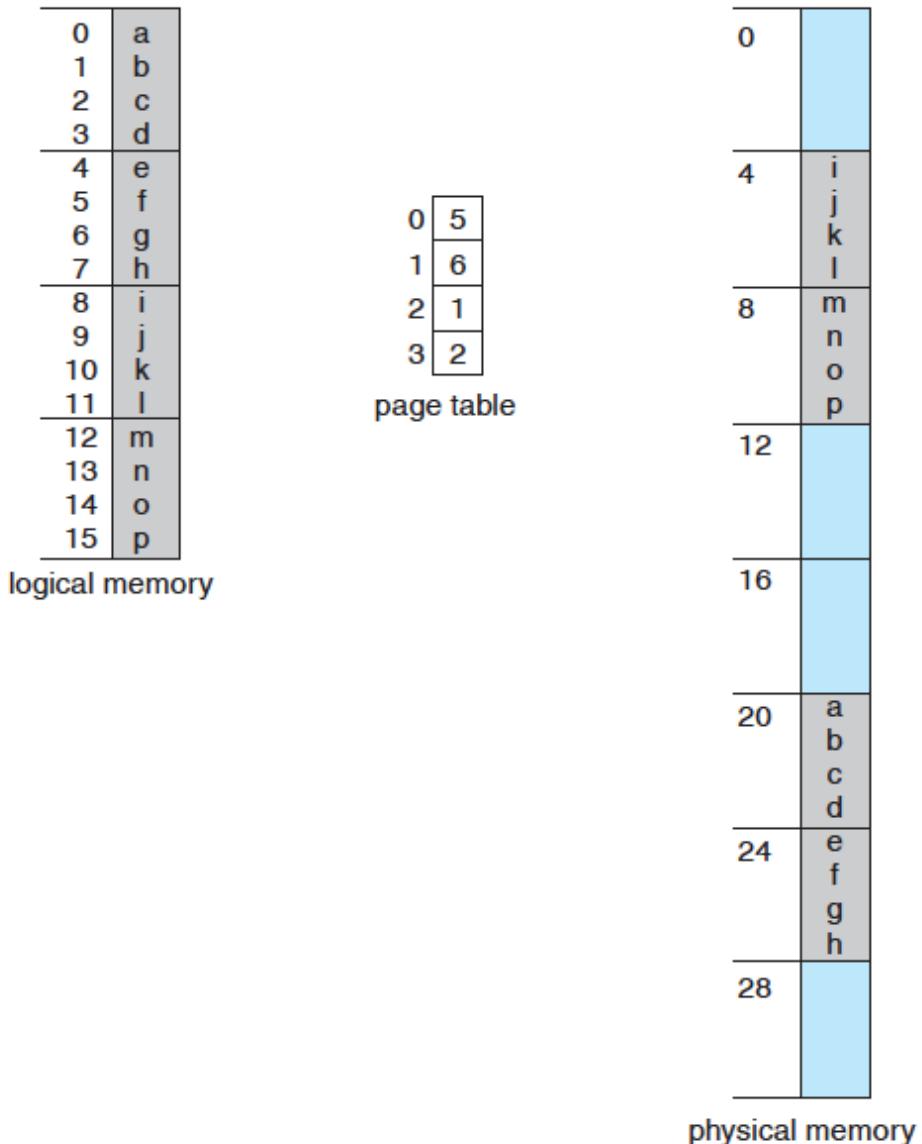
O ponto negativo deste tipo de alocação é o facto de os processos terem que estar contíguos na memória, o que leva à fragmentação. Uma solução seria ir juntando os blocos que estão a ser usados tornando-os contíguos, este técnica designa-se por compactação.

Paginação

A paginação vem resolver o problema da fragmentação de memória. A ideia da paginação é permitir que o espaço de endereçamento físico não seja contíguo, evitando a compactação e a fragmentação.

O método usado para implementar páginas é dividir a memória física em blocos de tamanho fixo, designados por *frames*, e dividir a memória lógica em blocos do mesmo tamanho, designados por páginas. A paginação tem suporte de *hardware*. Cada endereço gerado pelo *CPU* é composto por duas partes: o número da página e o seu *offset*. O número da página é usado como índice numa tabela de paginação.

O tamanho de um bloco é normalmente uma potência de 2, pois torna a tradução de endereços mais fácil.



Assumindo que existem 4 páginas, cada uma com 4 blocos, então os endereços são mapeados da seguinte maneira:

- Endereço lógico 3 -> está contido na página 0. Vai-se buscar o *frame* à tabela de paginação, fazendo com que o endereço lógico seja o índice da tabela, ou seja, o endereço lógico 0 é mapeado para a *Frame* 5 da memória física. Fazendo $5 * 4$ (*frame* x tamanho do *frame*), somos colocados no endereço físico 20, correspondente ao *frame* 5. Somando 3 (*offset*) obtém-se o endereço físico 23. Então, o endereço físico pode ser obtido fazendo: $5 * 4 + 3 = 23$.

```
pageTable[page(logicalAdress)] * pageSize + logicalAdress =  
physicalAdress
```

A vista da memória por parte do utilizador continua a mesma, ele pensa que tem a memória para si e que contém uma zona única de memória apenas para ele. No entanto, o que acontece é que os endereços que são gerados pelo *CPU* não correspondem aos endereços reais de memória, havendo este processo de mapeamento de endereços.

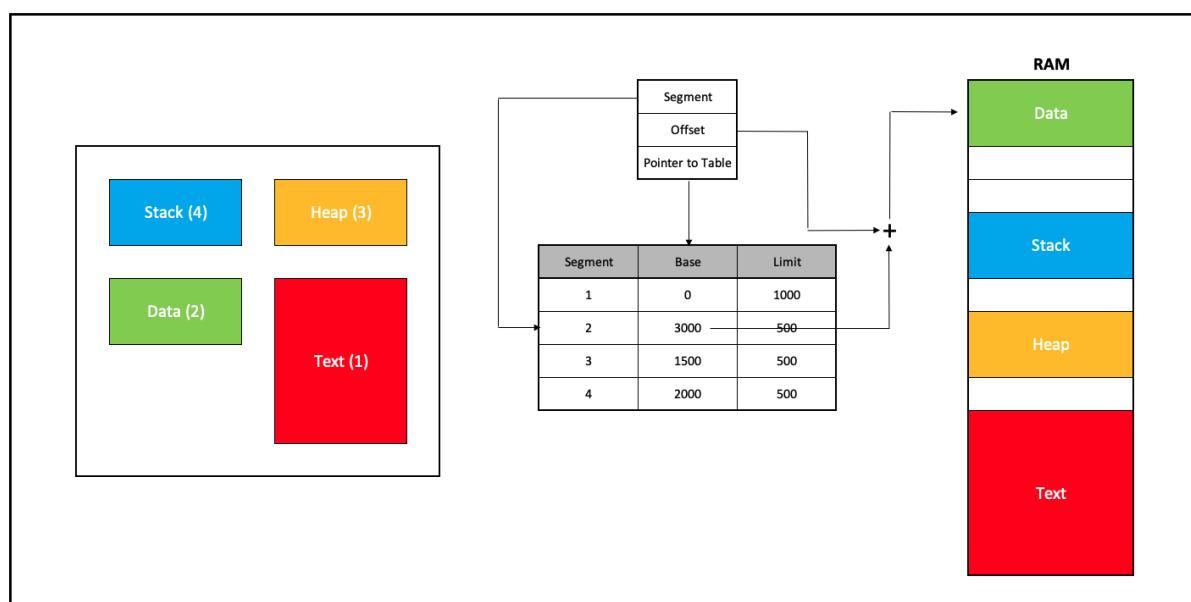
O sistema operativo contém uma *frame table* necessária para manter toda a informação das *frames* disponíveis, ocupadas, número total, etc.

Segmentação

Ao contrário da paginação em que o programa é dividido em páginas, todas com o mesmo tamanho, sem lógica alguma, a segmentação pretende dividir os programas de uma maneira mais lógica, uma divisão em módulos. É o que acontece com o compilador de C, em que divide o programa em:

- Código
- Variáveis globais
- *Heap*
- *Stack*
- Bibliotecas de C

A segmentação permite que os seus módulos sejam guardados de uma maneira não contígua na memória, no entanto, os módulos em si têm que estar contíguos.



Para calcular a localização de um segmento na memória, para o exemplo acima, basta fazer os seguintes cálculos:

- Procura-se o endereço base da memória física através do número de segmento.
- De seguida, basta somar o endereço lógico do segmento e obtém-se o endereço físico.
- O sistema operativo tem que verificar se o endereço é válido, isto é, $base + offset < base + limit \Leftrightarrow offset < limit$

A segmentação apresenta o mesmo problema de alocação de memória que é a fragmentação de memória. Como se pode ver na imagem, existe 4 blocos livres mas não contíguos.

Memória Virtual

O conceito de memória virtual surge para resolver problemas de memória. Os principais problemas relacionados com a memória principal são:

- **RAM insuficiente.** Antigamente era impossível correr um programa cujo espaço de endereçamento fosse maior do que a memória principal. Este foi o principal motivo para a criação da memória virtual, permitir que programas maiores que a memória pudessesem ser executados sem problemas.
- **Buracos no espaço de endereçamento.** Quando existem vários programas em execução pode existir espaço para novos programas na memória principal, mas como não são contíguos, novos processos podem não conseguir ser executados.
- **Programas a escreverem uns em cima dos outros.** Isto apresenta um grau de segurança muito baixo, pois os processos podem corromper-se uns aos outros.

- ▶ **Problemas**
- ▶ **Solução dos problemas**