**Gestão de Processos**

**Hierarquia:**

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

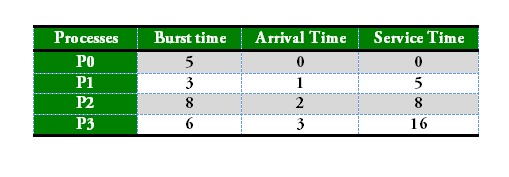
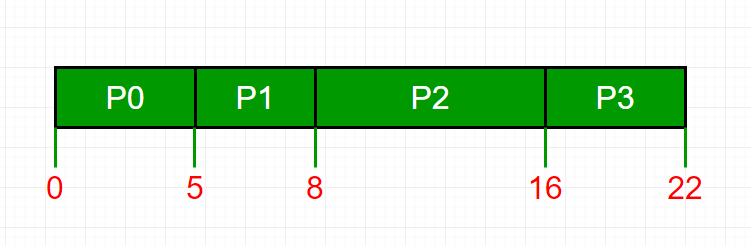
**Estados:**

* **Running**
  + Quando um processo se encontra neste estado, este está a utilizar o CPU.
* **Ready**
  + O processo está à espera que outros processos acabem de correr para uso do CPU.
* **Blocked/Waiting**
  + O processo encontra-se à espera de que algo aconteça para poder avançar. Um exemplo pode ser um processo à espera do input do utilizador.

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

**Escalonamento de Processos**

* **Conceito**
  + Um processo é executado até ter de esperar (normalmente por um request de I/O). Num sistema simples, o CPU fica numa espera ativa, i.e., sem fazer nada. Está, portanto, a desperdiçar tempo.
  + É o que acontece num sistema de monoprogramação: um processo tem direito ao CPU enquanto os seguintes ficam em espera ativa, testando se o CPU se encontra livre (chama-se *polling*), é o caso do MS-DOS.
  + A ideia é introduzir multiprogramação: utilizar este tempo de forma produtiva. Vários processos são mantidos em memória ao mesmo tempo. Quando um tem de esperar, o SO tira o CPU desse processo e dá a outro. E isto é contínuo: de cada vez que um processo tem de esperar, o CPU vai ser atribuído a outro.
  + Isto é decidido através do CPU *scheduler*.
  + É importante recordar que a quele de processos e os estados que podem tomar (wait,ready, run).
* **Burst-Cycle**
  + Se pensarmos que os processos estão constantemente à espera de I/O, concluímos que nesse período não estão a ser feitos cálculos importantes. Por outro lado, quando o processo trata de fazer cálculos, o I/O é mantido a níveis muito reduzidos.
  + É então fácil de concluir que o tempo é dividido em bursts de I/O ou de CPU. O processo faz cálculos (%CPU elevada) - precisa de IO, (%CPU baixa e a de I/O aumenta) - volta aos cálculos. Isto é essencial para perceber as estratégias de escalonamento.
  + Os processos com vários burst cycles de CPU (e, tipicamente, mais longos) chamam-se de CPU Bound, enquanto os que possuem mais burst cycles de I/O (e tipicamente mais longos) denominam-se de I/O bounds.
* **Escalonamento**
  + **O escalonamento é necessário em 4 casos:**
    1. Quando um processo muda de *running* para *waiting* (p.e. request de I/O)
    2. Quando um processo muda de *running* para *ready* (p.e. interrupção)
    3. Quando um processo muda de *waiting* para *ready* (p.e. completou I/O)
    4. Quando um processo termina
  + **O escalonamento é non-preemptive (cooperativo)**
* No caso 1 e 4, o escalonamento é non-preemptive, i.e., não são precisas técnicas adicionais, além dos algoritmos de escolha do próximo processo, para atribuir o CPU a outro processo.
* Quando, uma vez atribuído a um processo, o CPU nunca lhe é retirado.
  + **O escalonamento é preemptive (desafectação forçada)**
* No caso 2 e 3, estes só acontecem quando o escalonamento for preemptive, i.e., quando o CPU pode ser retirado a um processo ao fim do quantum ou porque surgiu outro de maior prioridade.
  + **Escalonamento non-preemptive vs preemptive:**
* As decisões relativas a escalonamento têm a ver com: **Qual o próximo processo a executar, quando começa a executar e durante quanto tempo executa?**
* A primeira destas questões é inerente ao algoritmo usado pelo *scheduler*, enquanto que as outras duas dependem se o escalonamento é non-preemptive ou preemptive.
* Escalonamento preemptive ocorre quando um processo possui o CPU, mas este é-lhe retirado, pelo SO, para dar a outro processo, sem que o primeiro, sem que o primeiro o tenha explicitamente libertado (ex. de libertação explícita são passar para a fila de waiting por I/O ou terminar).
* Escalonamento non-preemptive permite que um processo execute até, deliberadamente, libertar o CPU. O processo termina ou passa para waiting.
* Existem plataformas de hardware que apenas funcionam com este tipo de escalonamento, uma vez qua não requerem equipamento especial (como um CPU timer) necessário para preemptive.
* Por outro lado, também traz menos custos de manutenção (overhead) uma vez que não é necessário tratar da troca de processos nem de guardar o estado de um processo. Simplifica também a complexidade de um scheduler.
* Dito isto, percebe-se as vantagens e desvantagens de escalonamento preemptive. A maior vantagem é que podemos manter a correr vários processos e interagir com eles todos.
* Por outro lado, há um overhead adicional dado que para trocar de processo é preciso preservar o estado (registos, stacks,..) – imaginem estar numa aula e um responsável entra e manda trocar de sala. Temos de arrumar as coisas e copiar o que está no quadro antes de sair.
* Também aumenta a complexidade do hardware (p.e. é preciso um timer) e do *scheduler*.
* No entanto, se o escalonamento for non-preemptive, é impossível ter mais que um processo a correr simultaneamente. Por exemplo, se pensarmos num sistema de janelas como o do Windows, a cada janela está associado um processo. Portanto, se o escalonamento fosse sempre non-preemptive, seria impossível manter o browser e o spotify.
* **Critérios de escalonamento**
  + Utilização do CPU (ideal entre 40-90%)
  + **Throughput:** número de processos por unidade de tempo, mede portante quantos processos consegue concluir
  + **Turnaround time:** tempo desde que um processo entra na queue até que sai
  + **Waiting Time:** tempo que um processo está na queue de ready
  + **Response Time:** melhor que o turnaround time num sistema interativo. Tempo desde que o processo é submetido até obtermos a primeira resposta.
* **Algoritmos de Escalonamento**
  + **FCFS (First Come, First Served)**
* O mais simples de todos
* FiFO QUEUE
* Fácil de codificar
* Tempo de espera muito elevado, com grandes flutuações dependendo da ordem de chegada e das características dos processos
* Favorece os processos CPU BOUND
* Nada apropriado para ambientes interativos
* Nestea algoritmo 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 à medida que vão chegando, vão para o fim da fila. Quando um 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.
* **Escalonamento non-preemptive**



* Process Wait Time : Service Time - Arrival Time
* P0 0 - 0 = 0
* P1 5 - 1 = 4
* P2 8 - 2 = 6
* P3 16 - 3 = 13
* Average Wait Time: (0 + 4 + 6 + 13) / 4 = 5.75
* **SJF (Shortest Job First)**
  + Associam a cada processo uma estimativa do próximo CPU burst
  + Atribuem o CPU ao processo com menor CPU burst
  + Se o valor for igual, utiliza-se FCFS
  + É praticamente óptimo, mas é difícil saber a duração do próximo burst
  + Para longa duração, pode-se usar o process limit que o user submete (os users são convidados a estimar o limite de tempo - se o processo demorar mais que isso tem de ser tem de voltar a ser submetido)
  + Não pode ser implementado em short-term CPU scheduling (quando cada burst dura milissegundos) porque não há maneira de saber a duração de bursts mais pequenos. Pode-se estimar através dos bursts anteriores, no entanto o cálculo consome tempo.
  + Pode ser **non-preemptive ou preemptive** (neste último caso, interrompe se chegar um processo com mais prioridade)
  + Leva a **starvation (**ver Preemptive Priority**)**
* **Preemptive Priority**
  + Caso mais geral do SJF
  + Inteiro atribuído ao processo. Pode representar os requerimentos de memória, número de ficheiros abertos, ratio de I/O bursts, etc
  + Pode ser **preemptive ou non-preemptive**
  + Tal como o SJF levam à **starvation:** Um processo com uma prioridade muito reduzida pode demorar demasiado temo a ser corrido. Por exemplo, quando desmontaram o IBM 7094 no MIT em 1973, descobriram um processo de baixa prioridade que ainda não tinha sido corrido e tinha sido submetido em 1967.
  + **Solução:** atribuir mais prioridade aos processos mais antigos à medida que o tempo passa
* **RR (Round Robin)**
  + Foram feitos para sistemas de partilha de tempos
  + Semelhante ao **FCFS** mas com preemption.
  + É gerado um time quantum, geralmente de 10 a 100 milissegundos
  + A ready quele torna-se circular. O CPU roda por todos os processos, dando a cada um o tempo correspondente a um quantum.
  + **2 casos:** ou o processo caba antes do fim do quantum (liberta ele próprio o CPU e o scheduler segue para o próximo passo) ou o quantum chega ao fim (o temporizador acaba, vai causar uma interruºção, o estado do processo é guardado e é colocado no final da waiting queue)
  + Se o quantum for muito elevado, comporta-se como **FCFS**
  + Se o quantum for muito reduzido, o overhead de troca de processo toma conta do CPU e cada processo passa muito pouco tempo no CPU
  + **Preemptive e non-preemptive**
  + **Como o SJF requer vários cálculos, o tempo de resposta do RR é melhor**
* **Multilevel Feedback Queue**
  + Há várias queues de ready com diferentes níveis de prioridade
  + Se um processo usa demasiado CPU é movido para uma queue com menos prioridade
  + Só quando todos os processos da queue 0 terminam é que avança para a queue 1
  + Processos I/O bound e interativos são movidos para a queue 0
  + Para evitar starvation, os processos que estejam numa queue com menos prioridade há mais tempo, passam para queles com mais prioridade.
  + Se estiver a executar um processo de queue 1 e chegar um da queue 0, há desafetação forçada
  + Na prática: um processo que fique ready é colocado na queue 0 e dado um certo quantum. Se não terminar dentro desse quantum, é movido para a queue 1 com um quantum maior. Se acontecer o mesmo é colocado na queue 2 (e a partir daqui é **FCFS**)
  + Na queue 0 e 1 é um misto de **RR** com **SJF (RR** dentro da própria queue, **SJF** quando está na queue 1 e chega um processo para a queue 0**)**
  + **Como consequência,** os processos com CPU bursts reduzidos s\ao servidos rapidamente passando para o I/O. Os processos com CPU burst intermédios (queue 1) também são servidos rapidamente, mas com menos prioridade.
* **Nível de Escalonamento**
  + Dado que há muitos critérios de escalonamento, normalmente são divididos em níveis, para evitar sobrelotação de projetos.
  + **Nível 0:** só despacha o que está na RAM
  + **Nível 1:** decide que processos são multiprogramados (alguns são swapped out)
  + **Nível 2:** Não deixa criar processos
  + **À medida que o CPU vai ficando sobrecarregado, o nível de escalonamento aumenta.**
* **Sincronização de Processos**
  + **Race Condition**

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

**Resolução do problema:** precisamos de sincronizar os processos de alguma forma. Vamos apresentar algumas soluções:

* + **Problema da região crítica**

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

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

**Soluções:**

* + 1. **Solução de Peterson**
       - Só funciona com 2 processos
       - Variável “turn” que indica qual dos processos pode entrar na região crítica
       - Array flag indexado pelo número do processo (0 se for o primeiro a entrar, 1 se for o segundo) com o booleano a TRUE
       - Quando entra na região crítica, enquanto a vez é do processo atual e o outro está interessado em entrar, fica num ciclo de espera até o outro processo chamar a função de saída
       - Essa função, define que o array de interessados do processo que sai fica a falso
       - Esta solução não é aceitável nos sistemas modernos devido às instruções de load/store não garantirem que as variáveis vão ser colocadas a tempo
    2. **Semáforos**
       - Um semáforo S é um inteiro que varia com o uso de funções **wait()** ou **P()** e **signal()** ou **V().** **Wait()** decrementa S, **signal()** incrementa S. Estas duas funções devem ser executadas em interrupção.
       - Existem **2 tops** de semáforos. **Counting semaphores (**não tem limite**)** e **binary semaphores (**também se pode chamar mutex (mutual exclusion)).
       - Para usar semáforos **“mutex”** basicamente usamos **wait()** quando um processo entra na região crítica, proibindo os outros processos de entrarem (porque S está a 0) e usamos **signal()** para sair da região crítica. Nesse momento, outro processo pode entrar.
       - **Counting semaphores** são mais utilizados para controlar o acesso a um recurso limitado (**Exemplo:** o António só pode executar a função “comerLaranja()” se tiver comida. Se todos os amigos dele chamarem a função **wait()** (comer, decrementando o número de laranjas), o António fica sem laranjas e, não pode comer. O semáforo vai ser incrementado quando um dos amigos acabar de comer a sua laraja, e chamar a função signal()).
       - **Grande problema:** vai usar **busy waiting** (também conhecida por **polling**), ou espera ativa, porque um processo que não possa entrar na região crítica vai esperar que o semáforo seja incrementado. Este tipo de semáforos são também chamados **spinlock** porque vão estar a fazer **spin** enquanto esperam para entrar
       - **Resolução:** alterar a definição de semáforo. Agora não é só um inteiro, mas sim uma estrutura contendo um inteiro e uma lista de processos em espera para entrar. Ou seja, quando um processo faz **wait(),** se não puder entrar, adiciona-se à lista de processos em espera. Quanfo é feito **signal(),** é retirado o processo da lista. Nas definições de **wait()** e **signal**() são chamadas as funções **block() e wakeup()**, a primeira insere na queue de processos em espera, a segunda retira-os.
       - Com esta lista de processos em espera, o valor do semáforo pode ser negativo e indicar o número de processos em esperar, se for útil ao algoritmo. Nesse caso também teria de se alterar as definições de **wait() e signal().**
       - A lista de processos em espera pode ser implementada com apontadores Process Control Block de cada processo em espera. Esta lista pode ser uma Queue (FIFO) de forma a remover processos em espera por ordem de entrada.
       - Em máquinas single-processor, interrupç~es devem ser desativadas nas funções **wait() e signal(),** garantindo que estas não são interrompidas pelo sistema(não garante, no entanto que resista a falhas de sistema, bugs..)
       - Em máquinas multi-processor, é mais complicado, visto que teríamos de desativar interrupções em todos os processadores, o que seria uma tarefa demasiado difícil e afetaria demasiado a performance. Para, de certa forma, resolver o problema, são usados spinlocks dentro das funções wait() e signal(), que cria busy waiting, mas visto que essas funções são executadas em pequenos intervalos de tempo e raramente, a perda em performance é duzida. Excepto no caso de programas nos quais a região crítica é longa. Aí a técnica de busy waiting é extremamente ineficiente.
       - Na realidade, o busy waiting ainda não foi completamente removido.
* **DEADLOCKS**
  + Fenómeno que acontece quando um processo nunca consegue sair de um estado de espera, uma vez que está à espera de recursos utilizados por outros processos.
  + **Condições necessárias para que surja um DeadLock:**
    1. Existir um recurso que não pode ser partilhado, e este está a ser usado por um processo, enquanto que há outros processos que precisam desse mesmo rescurso
    2. Existir um processo que está a usar um recurso, mas, ao mesmo tempo, à espera de recursos adicionais, a serem usados por outros processos
    3. Serem os próprios processos responsáveis pela libertação do recurso que usam
    4. Espera circular: um processo está à espera de um recurso a ser usado por outro processo, que por sua vez está à espera de um recurso..
  + **Formas de lidar com deadlocks:**
    1. Assegurar que os deadlocks nunca acontecem (basta fazer com que nenhu,a das condições se verifique)
    2. Permitir que os deadlocks ocorram, sendo estes detetados e tratados
    3. Ignorar o problema (forma mais usada**,** UNIX, Windows.. é o programador que tem de tratar)