Processos

Ricardo de la Rocha Ladeira

Introdução

A aula passada trouxe uma visão geral sobre os principais conceitos da Programação Concorrente. Entre esses conceitos estavam processos e *threads*, que serão abordados no decorrer da disciplina. Nesta aula, a ideia é abordar o tema *Processos*. O objetivo principal desta aula é compreender conceitos, criação, escalonamento e gerenciamento de processos. Ao final, exercícios serão disponibilizados para possibilitar a aplicação dos conhecimentos adquiridos e o aprofundamento da compreensão sobre processos.

Grande parte do texto é baseada no capítulo 3 de [1], cuja leitura foi solicitada nos exercícios da aula anterior.

Processos [1]

Um processo é um programa em execução. Um processo precisará de certos recursos — como tempo de CPU, memória, arquivos e dispositivos de E/S — para cumprir sua tarefa. Esses recursos são geralmente alocados para o processo enquanto ele está sendo executado.

Os primeiros computadores permitiam que apenas um programa fosse executado por vez. Este programa tinha controle total do sistema e acesso a todos os recursos do sistema. Em contraste, os sistemas operacionais contemporâneos permitem que vários programas sejam carregados na memória e executados simultaneamente. Essa evolução exigiu um controle mais firme e mais compartimentalização dos vários programas; e dessas necessidades resultou a noção de um processo, que é um programa em execução. Um processo é a unidade de trabalho em um sistema operacional moderno.

Os sistemas operacionais modernos suportam processos com múltiplas *threads* de controle. Em sistemas com vários núcleos de processamento de hardware, essas *threads* podem ser executadas em paralelo.

Quanto mais complexo for o sistema operacional, mais se espera que ele faça pelos usuários. Embora sua principal ação seja a execução de programas de usuário, ele também precisa cuidar de várias tarefas do sistema que são melhor executadas no espaço do usuário, em vez de dentro do *kernel*. Um sistema, portanto, consiste em uma coleção de processos, alguns executando o código do usuário, outros executando o código do sistema operacional. Potencialmente, todos esses processos podem ser executados simultaneamente, com a CPU (ou CPUs) multiplexada entre eles.

O estado da atividade atual de um processo é representado pelo valor do contador do programa (*program counter* — PC) e pelo conteúdo dos registradores do processador. O *layout* da memória de um processo normalmente é dividido em várias seções, como mostrado na Figura 1.

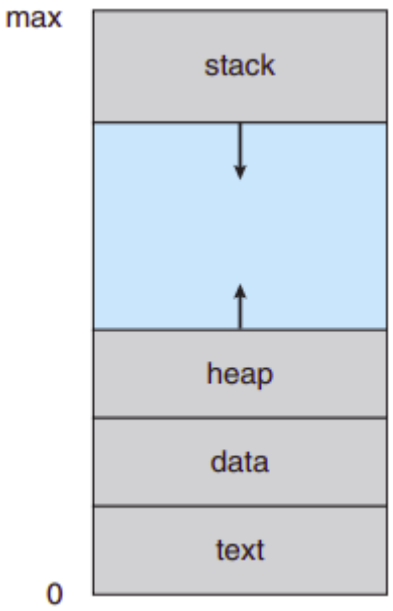

Figura 1. Processo na memória [1]. Disponível em https://drive.google.com/file/d/1vgHXE4OxH3gNQN2_hPOSlPkqXn0jxLnj/view?usp=sharing.

Essas seções incluem:

• Seção de **texto (*text*)**: o código executável (tamanho fixo)

• Seção de **dados (*data*)**: variáveis globais (tamanho fixo)

• Seção ***Heap***: memória que é alocada dinamicamente durante a execução do programa, como objetos criados, dados de tamanho variável e estruturas compartilhadas entre *threads*.

• Seção **Pilha (*Stack*)**: armazenamento temporário de dados ao invocar funções (como parâmetros de função, endereços de retorno e variáveis locais)

As seções *heap* e pilha possuem tamanho dinâmico, sendo alteradas durante a execução do programa.

Estados de um processo [1]

Quando um processo é executado, seu estado é alterado. O estado de um processo é definido pela atividade que realiza, e pode ser um dos seguintes:

- **Novo**: estado em que o processo está sendo criado.
- **Executando**: as instruções do processo estão sendo executadas.
- **Esperando**: o processo está esperando que algum evento ocorra (recepção de algum sinal ou aguardando alguma operação de E/S). Há autores que chamam esse estado de "bloqueado".
- **Pronto**: o processo está esperando para ganhar o processador.
- **Concluído**: o processo concluiu todas as suas instruções.

Um diagrama de estados para apresentar visualmente os possíveis estados de um processo está disponível na Figura 2.

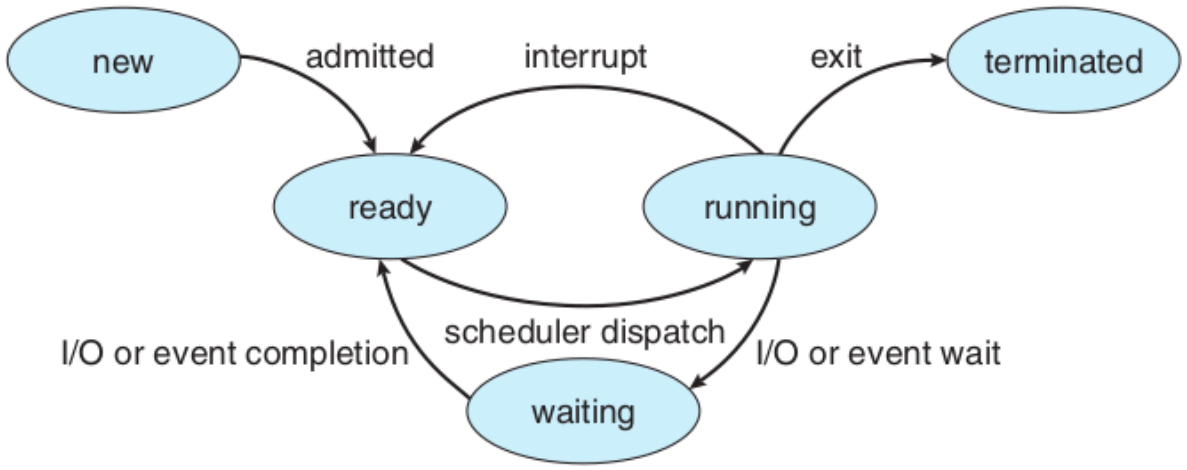

Figura 2. Diagrama de estados de processos [1]. Disponível em https://drive.google.com/file/d/13GOHiJFsHzIiuJL8t7qvkOtuh6k7D43c/view?usp=drive_link.

Observe que é possível que um processo em estado *executando* possa passar para o estado *pronto*. Na sequência deste material, exploramos o componente responsável por promover essa troca de estado (Escalonador de Processos).

Bloco de Controle de Processo [1]

Cada processo é representado por um bloco de controle de processo (*Process Control Block* — PCB), também chamado de *bloco de controle de tarefas*. Ele contém informações associadas a processos específicos, tais como o estado do processo, o contador de programa (PC), registradores, informações de escalonamento, informações de gerenciamento de memória, de contabilidade de tempo e de status de E/S. Em resumo, o PCB serve simplesmente como repositório de todos os dados necessários para iniciar ou reiniciar um processo, juntamente com alguns dados contábeis. A Figura 3 mostra o PCB.

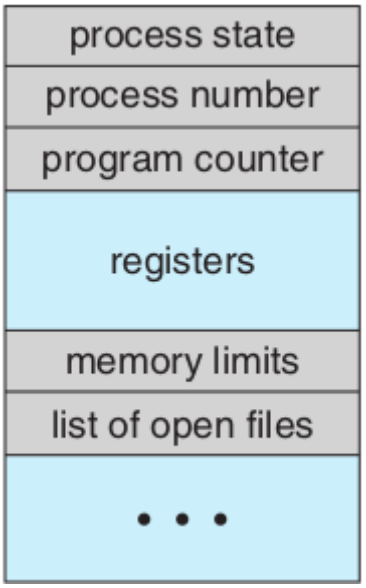

Figura 3. *Process control block* – PCB [1]. Disponível em https://drive.google.com/file/d/1U7BkYxgkDlDIu1SANnmmlCcUIIdkzfA0/view?usp=drive_link.

Processos *CPU Bound* e *I/O Bound*

Processos podem ser classificados em *CPU Bound* e *I/O Bound*. Esses termos são usados para descrever o desempenho e o comportamento de um processo em relação ao uso de recursos do sistema, como a CPU e operações de E/S. Essa classificação é relevante para fins de otimização e gerenciamento do sistema.

Exemplo: em sistemas multitarefa, o escalonador pode priorizar processos *I/O Bound* para evitar que fiquem esperando muito tempo e, assim, melhorar a eficiência geral do sistema. Da mesma forma, em um ambiente onde recursos de CPU são escassos, a priorização de processos *CPU Bound* pode ser importante para garantir um bom desempenho em tarefas críticas que exigem muitos recursos de processamento.

Um processo é *CPU Bound* se ele depende principalmente do processamento da CPU para a conclusão de suas tarefas. Esse tipo de processo geralmente envolve muitos cálculos intensivos e requer uma quantidade significativa de tempo da CPU para executar suas operações. Enquanto o processo está em execução, ele mantém a CPU ocupada a maior parte do tempo e pode deixar outros processos esperando na fila até que a CPU esteja disponível novamente.

Exemplo: Um programa de renderização gráfica que processa complexas animações 3D pode ser considerado *CPU Bound*, pois a maioria do tempo é gasto realizando cálculos intensivos relacionados à renderização.

Um processo é *I/O Bound* se ele passa a maior parte do tempo aguardando operações de E/S serem concluídas, em vez de depender principalmente do processamento da CPU. Esse tipo de processo normalmente envolve operações de leitura ou gravação em dispositivos de armazenamento (como disco rígido, SSD, rede, etc.) ou E/S com dispositivos periféricos.

Exemplo: Um programa que lê dados de arquivos armazenados em um disco rígido e realiza operações com esses dados (como um aplicativo de edição de vídeo) pode ser considerado *I/O Bound*, pois grande parte do tempo é gasto aguardando a conclusão das operações de leitura do disco.

Diz-se ainda que o *surto de CPU* é o período que um processo quer executar processamento, assim como o *surto de I/O* é o período em que o processo fica bloqueado em operações de I/O.

Processos ancestrais e filhos

Um processo pode ainda criar subprocessos. Neste caso, o processo que cria subprocessos é dito ancestral (ou pai), e todos os subprocessos por ele criados são considerados processos filhos. Quando um programa em C utiliza a função `fork()`, a própria execução do programa será o processo ancestral e a execução do novo PID (identificador de processo) gerado pela função `fork()` será o processo filho.

Quando a função `fork()` é chamada, o sistema operacional cria uma cópia do processo ancestral, resultando em dois processos distintos: o processo ancestral e o processo filho.

Ambos os processos têm o mesmo código do programa até o ponto onde a função `fork()` foi chamada.

Os dois processos continuam a executar a partir do ponto subsequente à chamada `fork()`. No entanto, eles retornam valores diferentes para essa função. O valor de retorno é `0` para o processo filho e um valor positivo (o ID do processo filho) para o processo ancestral. Isso permite que o programa saiba qual parte do código está sendo executada em cada processo.

É importante observar que os processos ancestral e filho têm sua própria cópia das variáveis e do estado do programa, ou seja, as alterações feitas em uma cópia não afetam a outra.

No Linux, e em sistemas Unix-like em geral, todos os processos têm um ancestral comum conhecido como processo `init`. O processo `init` é o primeiro processo a ser executado pelo *kernel* durante o processo de inicialização do sistema, e ele é responsável por iniciar todos os outros processos. O nome do processo é `systemd`.

O processo `init` tem o PID (identificador de processo) 1 e é criado pelo *kernel* assim que o sistema operacional é iniciado. Isso pode ser confirmado digitando

```
> ps -p 1
```

Para ver os processos rodando em sistemas Unix-like, pode-se utilizar o comando `ps`. Outro comando útil é o comando `pstree`, que permite visualizar os processos em forma de árvore, mostrando assim a hierarquia entre ancestrais e filhos, bem como mostra também quando um processo tem *threads* (assunto futuro da disciplina). O comando "`ps f`" também permite exibir os processos hierarquicamente.

Quando um processo tem *threads*, o `pstree` mostra essas *threads* como subprocessos do processo ancestral. Elas são identificadas pelo nome do processo ancestral, que é seguido de um conjunto de chaves contendo o nome da *thread* entre elas. As *threads* são representadas como filhos do processo ancestral na árvore de processos.

Exemplo com uma possível saída do `pstree` que contém *threads*:

```
icecast2---13*[{icecast}]
```

Nesse exemplo, o processo chamado `icecast` possui uma *thread*. O `pstree` representa essa *thread* como um subprocesso do processo `icecast2`. O número 13 após o hífen indica o PID da *thread*, e o nome da *thread* (`icecast`) está entre chaves, indicando que é uma *thread* do processo `icecast2`. Essa notação ajuda a identificar quais processos têm *threads* associadas e permite visualizar a hierarquia de processos e *threads* na árvore de processos.

Criando processos

Criar processos em C é feito utilizando a função `fork()`[1]. Esta função está disponível em `unistd.h`. Além disso, ela retornará sempre um identificador de processo (*Process ID*, ou PID), que é de tipo `pid_t`. O tipo `pid_t` é definido em `sys/types.h`.

Exemplos criados:

- exemplo-1-criar-processo.c
- exemplo-2-criar-dois-processos-filhos.c
- exemplo-3-processos-mais-uteis.c
- exemplo-4-processos-pai-filho.c

Há pontos interessantes a serem discutidos.

- O primeiro código mostra como criar um processo filho. Mas ele é filho de quem?
- O segundo código contém um mecanismo de sincronização. Porém, se ele não estiver presente, como o Sistema Operacional decide que processo executar primeiro?
- O terceiro código executa ações mais interessantes.
- O quarto código utiliza duas chamadas subsequentes da função `fork()`, o que acaba gerando quatro processos. Mas por quê? O primeiro `fork()` bifurca o processo original em dois. Esses dois continuam a execução a partir da próxima instrução, que é uma nova bifurcação. Assim, os dois processos criados após o primeiro `fork()` bifurcam novamente, gerando ao todo quatro processos. Por esse motivo a mensagem é exibida quatro vezes.

---

[1] A função `fork()` cria processos em sistemas Unix-like. A chamada de sistema que cria um processo no Windows, por exemplo, não é feita com `fork()`, mas com `CreateProcess()`. A Figura 4 mostra alguns exemplos de chamadas de sistema para Windows e Unix.

EXAMPLES OF WINDOWS AND UNIX SYSTEM CALLS

The following illustrates various equivalent system calls for Windows and UNIX operating systems.

	Windows	Unix
Process control	`CreateProcess()`	`fork()`
	`ExitProcess()`	`exit()`
	`WaitForSingleObject()`	`wait()`
File management	`CreateFile()`	`open()`
	`ReadFile()`	`read()`
	`WriteFile()`	`write()`
	`CloseHandle()`	`close()`
Device management	`SetConsoleMode()`	`ioctl()`
	`ReadConsole()`	`read()`
	`WriteConsole()`	`write()`
Information maintenance	`GetCurrentProcessID()`	`getpid()`
	`SetTimer()`	`alarm()`
	`Sleep()`	`sleep()`
Communications	`CreatePipe()`	`pipe()`
	`CreateFileMapping()`	`shm_open()`
	`MapViewOfFile()`	`mmap()`
Protection	`SetFileSecurity()`	`chmod()`
	`InitlializeSecurityDescriptor()`	`umask()`
	`SetSecurityDescriptorGroup()`	`chown()`

Figura 4. Chamadas de Sistema no Windows e no Unix [1]. Disponível em https://drive.google.com/file/d/1bN8Sq2BRAZZw7ex9HmLYj-qQW8Fy37d4/view?usp=drive_link.

A multiprogramação surgiu com o objetivo de melhorar o uso da CPU com processamento útil. Com isso, espera-se que o sistema possa executar

I. programas mais rapidamente; e

II. mais programas por unidade de tempo.

Como os processos são instâncias de programas, faz-se necessário usar um componente denominado escalonador de processos, peça vital do Sistema Operacional, para ser possível executar I e II.

Escalonador de processos

O componente do sistema operacional responsável por decidir a ordem de execução dos processos é o *Escalonador de Processos*, também chamado de *scheduler*. O escalonador é parte do *kernel* do sistema operacional e sua principal função é decidir qual processo será executado na CPU em um determinado momento. Ele também gerencia a alocação de tempo da CPU entre os processos concorrentes e define a ordem de execução deles.

O escalonador baseia suas decisões em algoritmos de escalonamento, que podem variar de acordo com o sistema operacional e suas configurações. Os algoritmos de escalonamento determinam a política de prioridade e a forma como os processos são escolhidos para serem executados na CPU.

Sem nenhum mecanismo específico de sincronização, o escalonador pode decidir a ordem de execução dos processos com base em vários fatores, incluindo prioridade dos processos, quantidades de CPU que cada processo já utilizou (escalonamento por fatia de tempo), entre outros critérios

O escalonador executa suas decisões de forma dinâmica e pode alterar a ordem de execução dos processos ao longo do tempo, dependendo das condições do sistema. É por isso que, em cada execução do programa, a ordem de execução do processo ancestral e do processo filho pode variar, já que ela é determinada pelo escalonador do sistema operacional.

Essa troca de processos no processador é feita por um componente chamado *despachador* (*dispatcher*).

O *dispatcher* é o componente que realiza a troca de contexto entre os processos. Ele é responsável por salvar o estado do processo que está em execução atualmente (contexto) e carregar o estado do próximo processo selecionado pelo escalonador, permitindo que ele seja executado. A Figura 5 apresenta esse processo.

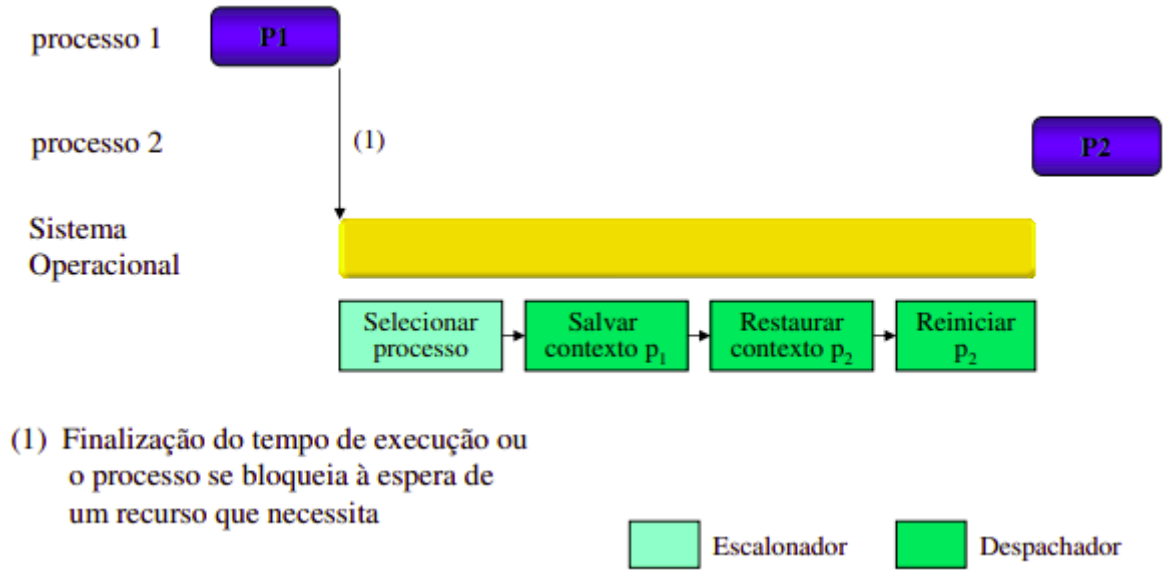

Figura 5. Troca de processos [2]. Disponível em https://drive.google.com/file/d/1e1ThN-vCHhSJENh8upahVylYjR4aKreG/view?usp=drive_link.

Tipicamente, costuma-se classificar os escalonadores de processos em ***não preemptivos*** e ***preemptivos***.

Nos escalonadores *não preemptivos*, o processo que possui a CPU somente a libera quando quer, ou seja, quando acaba sua execução. Esses escalonadores não necessitam suporte de hardware adicional e não são convenientes para ambientes de tempo compartilhado [2]. Neles, portanto, um processo pode monopolizar a CPU [2]. O MS-DOS e o Windows 3.1 possuíam escalonador não preemptivo.

Os principais algoritmos de escalonamento não preemptivos são

- *First Come, First Served* (FCFS)
- *Shortest Job First* (SJF)

Os escalonadores *preemptivos* têm a capacidade de interromper um processo em execução e substituí-lo por outro processo com maior prioridade. Nesse caso, o escalonador deve ser capaz de tomar decisões de preempção com base nas prioridades dos processos e nas regras de escalonamento. Esses escalonadores são de maior custo, mas evitam que um processo tenha 100% da CPU [2]. Os sistemas *Unix-like* e os Windows mais recentes (Windows 10, por exemplo) possuem escalonador preemptivo.

Os principais algoritmos de escalonamento preemptivos são

- *Round-Robin*

- Fila de prioridades
- Filas multinível

Algoritmos de escalonamento não preemptivos

First Come, First Served (FCFS)

- Ao usar este algoritmo, o escalonador está utilizando a política de ordem de chegada, ou seja, o primeiro processo a chegar na fila de processos prontos é o primeiro que utilizará a CPU.
- O processo que tem a CPU não a libera até que acabe sua execução ou até que fique bloqueado por uma operação de E/S [2].
- A implementação é simples, pois basta ter uma fila. Por isso, muitas vezes o algoritmo FCFS também é chamado de FIFO (*first in, first out*).
- Desvantagem: enquanto um processo ganha a CPU, todos os outros esperam.

Shortest Job First (SJF)

- Este algoritmo ordena os processos na fila de processos prontos pelo tempo de uso da CPU, colocando sempre à frente os processos que a utilizarão menos. Isso reduz o tempo de espera dos outros processos.
- Pressupõe que surtos de CPU mais curtos são de processos que farão I/O.
- Caso dois processos precisem da CPU pelo mesmo tempo, aplica-se algum critério de desempate (por exemplo, a ordem de chegada na fila — FCFS).
- O processo que possui a CPU somente a libera quando termina sua execução ou quando se bloqueia [2].
- Existe uma variação preemptiva desse algoritmo chamada SRTF (*Shortest Remaining Time First*), que considera a existência de processos que precisem da CPU por menos tempo do que o processo atual na CPU ainda precisa para concluir sua execução (baseado em [2]).

Algoritmos de escalonamento preemptivos (baseado em [2])

Round-Robin (RR)

- Atribui-se a cada processo durante um intervalo de tempo um valor fixo de forma rotativa, denominado *quantum*.
- Se assemelha ao algoritmo FCFS, pois usa uma fila FIFO de processos.
- O escalonador percorre a fila e atribui até 1 quantum para cada processo.

- É ideal para sistemas de tempo compartilhado.
- Se um processo não deixar a CPU dentro do quantum, ele é preemptado.
- Se houver *n* processos e o *quantum* for *q*, cada processo possui *1/n* tempo de CPU, executado em porções de tempo de tamanho até *q*.
- Nenhum processo espera mais do que *(n-1)q* para utilizar CPU.
- Não ocorre *starvation*.
- O *quantum* não pode ser muito grande, pois assim o algoritmo acaba sendo praticamente como o FCFS.
- O *quantum* também não pode ser muito pequeno, pois assim ocorrem muitas trocas de contexto e isso gera um custo muito alto.
- O *quantum* deve ser pequeno o suficiente para garantir o tempo compartilhado.
- O *quantum* deve ser grande bastante para compensar trocas de contexto

Fila de prioridades

- Cada processo tem sua prioridade de execução, normalmente representada por um valor inteiro.
- O processo que ganha a CPU é aquele que tem maior prioridade entre os que estão na fila de processos prontos.
- A fila é ordenada pela prioridade.
- Caso um processo passe para o estado *Esperando* e tenha maior prioridade em relação ao processo de estado *Executando*, ocorre a preempção.
- Uma desvantagem deste algoritmo é a possibilidade de ***inanição***, também chamada de ***starvation***. Isso pode acontecer se um processo tiver prioridade muito baixa e, ao longo do tempo, processos de prioridade mais alta entrarem na fila, ganhando sempre o processador em vez do processo de prioridade mais baixa.
- Algumas estratégias adotadas para evitar *starvation* ao utilizar escalonadores por prioridade são:
    - **Prioridades Dinâmicas**: adotar um sistema onde a prioridade de um processo é ajustada com o tempo. Por exemplo, a cada vez que um processo é executado, sua prioridade pode ser reduzida gradualmente. Isso garante que processos de baixa prioridade, mesmo que inicialmente, tenham chances de serem executados.
    - ***Aging* (Envelhecimento)**: consiste em aumentar gradualmente a prioridade dos processos de baixa prioridade à medida que eles permanecem no sistema esperando por execução. Dessa forma, quanto mais tempo um processo passa na fila de espera, maior é sua prioridade, o que aumenta

suas chances de ser selecionado para a execução. É uma variação de prioridades dinâmicas.

- **Filas Multinível**: O uso de filas multinível no escalonamento pode ser uma solução para evitar *starvation*. Nesse esquema, existem várias filas de prioridades, e os processos são promovidos ou rebaixados entre essas filas com o tempo. Isso garante que todos os processos, independentemente de sua prioridade inicial, tenham oportunidades de serem executados em algum momento.

Filas Multinível

- Nesta abordagem, não é utilizada uma fila para os processos prontos, mas muitas filas (2 ou mais).
- Cada fila pode utilizar um algoritmo de escalonamento diferente.
- Por exemplo: podem ser utilizadas 2 filas, uma fila `f1` para processos em *foreground* e outra, `f2`, para processos em *background*. A fila `f1` pode utilizar RR e a fila `f2` pode utilizar o FCFS.
- É necessário haver escalonamento entre as filas, para escolher o processo de qual fila será executado.
- Se uma fila tiver prioridade sobre a outra, pode ocorrer *starvation*. Neste caso, alguma abordagem precisa ser utilizada para resolver isso (exemplo: *aging*).
- Outra opção: dividir o tempo de execução entre as filas. Exemplo: `f1` fica com 80% e `f2` com 20% do tempo de CPU.
- A preempção nas filas multiníveis pode acontecer em várias situações, tais como:
  - **Prioridade mais alta**: Um processo que possui uma prioridade mais alta, como um processo de tempo real ou uma tarefa crítica, pode ser interrompido e movido para a fila de prioridade mais alta para garantir que seja atendido o mais rapidamente possível, mesmo que existam outros processos em execução na fila de prioridade inferior.
  - **Quantum de tempo**: Em sistemas de escalonamento por fatia de tempo (*Round-Robin*), cada processo recebe uma fatia de tempo para execução. Se o processo não conseguir concluir sua tarefa dentro do quantum de tempo, ele será interrompido e colocado novamente na fila de prioridade mais baixa para aguardar sua próxima fatia de tempo.
  - **Chegada de tarefas prioritárias**: Se uma tarefa de alta prioridade chegar durante a execução de um processo em uma fila de prioridade mais baixa, o processo em execução pode ser interrompido para dar prioridade à nova

tarefa. O processo interrompido será movido para a fila de prioridade mais alta e aguardará sua vez para ser executado novamente.

Escalonamento de Processos no Linux

Sistemas operacionais que utilizam o *kernel* do Linux possuem um escalonador de processos conhecido como *Completely Fair Scheduler* (CFS), um algoritmo de escalonamento de processos preemptivo.

O CFS foi introduzido no *kernel* 2.6.23 e foi projetado para fornecer um escalonamento justo entre os processos em execução no sistema. Ele trata cada processo como se estivesse recebendo uma parte justa do tempo de CPU disponível, independentemente do número de núcleos da CPU ou do número de processos em execução. Isso é especialmente importante em sistemas multiusuários e multiprocessados, onde a equidade é um fator crítico.

Funcionamento do CFS:

Utilização do tempo: O CFS acompanha o tempo de execução real de cada processo. Os processos que utilizaram menos tempo de CPU em relação ao seu tempo alocado têm uma prioridade maior.

Prioridades dinâmicas: O CFS atribui prioridades dinâmicas aos processos. Processos que estão esperando há mais tempo (ou seja, têm maior duração de espera) recebem uma prioridade mais alta para serem executados.

Medição do *vruntime*: O CFS mede o *vruntime* (tempo virtual) de cada processo. O *vruntime* é usado para determinar a quantidade de tempo de CPU que cada processo deve receber. Processos com *vruntime* menor têm prioridade para a execução.

Quantum: O CFS usa um conceito de *quantum infinito*. Na prática, isso significa que o CFS divide o tempo de CPU disponível em fatias muito pequenas e aloca essas fatias aos processos com base em suas prioridades dinâmicas.

Retirar a CPU: Quando um processo bloqueia ou é preemptado, ele é retirado da CPU e colocado em uma fila de processos prontos. O escalonador seleciona o próximo processo com a menor *vruntime* para executar.

O CFS visa otimizar a utilização do tempo de CPU, minimizar o tempo médio de espera dos processos e garantir que cada processo receba sua "fatia justa" de tempo de CPU, criando assim uma sensação de responsividade para o usuário.

É importante notar que o CFS é apenas um componente do subsistema de escalonamento do *kernel* Linux. Existem outros algoritmos de escalonamento que podem ser escolhidos, dependendo das necessidades específicas do sistema ou do cenário de uso, como o *Deadline Scheduler* para sistemas em tempo real ou o *Completely Fair Queuing* (CFQ) para dispositivos de armazenamento. O *kernel* Linux é altamente configurável e permite que diferentes algoritmos de escalonamento sejam selecionados com base nas necessidades do sistema.

Escalonamento de Processos no Windows

O escalonador de processos do Windows é conhecido como *Windows Scheduler* (Escalonador do Windows) ou *Multilevel Feedback Queue Scheduler* (Escalonador de Filas de Feedback Multinível). O *Windows Scheduler* é um escalonador de tempo compartilhado e preemptivo. Ele utiliza múltiplas filas de prioridade, conhecidas como filas de feedback, para organizar os processos.

Funcionamento do Windows Scheduler:

Filas de Feedback Multinível: O *Windows Scheduler* divide os processos em várias filas de feedback, cada uma com uma prioridade diferente. Cada fila tem um tempo de quantum associado, que é o tempo de CPU que um processo pode executar antes de ser preemptado e movido para uma fila de prioridade mais baixa.

Prioridades Dinâmicas: À medida que os processos são executados, o escalonador monitora o comportamento dos processos e ajusta suas prioridades dinamicamente com base em seu histórico de uso da CPU. Processos que usam muita CPU podem ser rebaixados para filas de prioridade mais baixa, enquanto processos inativos ou que usam pouca CPU podem ser promovidos para filas de prioridade mais alta.

Preempção: O *Windows Scheduler* permite a preempção de processos. Se um processo com uma prioridade mais alta estiver pronto para ser executado, ele pode interromper um processo de menor prioridade e assumir o controle da CPU.

Gerenciamento de E/S: O *Windows Scheduler* também leva em consideração o gerenciamento de E/S. Se um processo estiver bloqueado esperando por uma operação de E/S, ele será colocado em uma fila separada (fila de espera de E/S) e não será considerado para a execução até que a operação de E/S seja concluída.

***Thread* Scheduling**: O *Windows Scheduler* gerencia *threads* em vez de processos diretamente. Cada processo pode ter vários *threads*, e o escalonador gerencia a execução dos *threads* em vez dos processos em si.

Avaliação dos Algoritmos de Escalonamento (baseado em [2])

Avaliação determinística: Nesse caso, os algoritmos de escalonamento são analisados de forma teórica e, muitas vezes, matemática, para determinar seu comportamento em cenários específicos. Isso envolve a utilização de métricas teóricas, como *throughput*, tempo de retorno, tempo de espera, etc., e permite uma compreensão das características do algoritmo em diferentes condições.

Throughput: também chamado de *vazão* ou *produtividade*, é o número de processos terminados por unidade de tempo.

Tempo de retorno: também chamado de *turnaround time*, é o tempo transcorrido desde que se lança um processo (entra na fila de processos prontos) até que finalize sua execução. É a soma do tempo de espera para ir para a memória, tempo de espera na fila de processos prontos, tempo em execução na CPU e o tempo de espera por recursos.

Tempo de retorno médio: tempo médio transcorrido desde que um processo entra na fila de processos prontos até que seja finalizada sua execução.

Tempo de espera: tempo que o processo permanece na fila de processos prontos. É a soma dos períodos utilizados pelo processo no estado de Pronto.

Tempo de espera médio: tempo médio que todos os processos esperam.

Testes pré-determinados: Os algoritmos de escalonamento podem ser testados em situações pré-determinadas com base em um conjunto de casos de teste específicos. Esses casos de teste são projetados para cobrir diversas situações de carga de trabalho e tipos de processos, permitindo uma análise mais detalhada das características de cada algoritmo em cenários simulados.

Implementação: Os algoritmos de escalonamento também podem ser avaliados por meio de sua implementação prática no *kernel* de em um sistema operacional. Nessa abordagem, são utilizadas métricas de desempenho colhidas durante a execução do sistema em condições reais de uso. É a abordagem mais realística, mas é também a mais trabalhosa.

Simulação: A simulação computacional é uma ferramenta precisa para avaliar o desempenho de algoritmos de escalonamento em ambientes controlados e reproduzíveis. Com simulações, é possível testar diferentes algoritmos sob uma variedade de cenários e condições de carga (processos, picos de CPU etc), que podem ser gerados aleatoriamente, sem afetar um ambiente de produção real.

Resumo

Nesta aula, foram abordados os conceitos de processos em sistemas operacionais. Um processo é um programa em execução que requer recursos como CPU, memória e dispositivos de E/S. Sistemas operacionais contemporâneos permitem que vários programas sejam carregados na memória e executados simultaneamente. O conceito de processo foi desenvolvido para lidar com a execução simultânea de vários programas. Os processos podem criar subprocessos, onde o criador é chamado de processo ancestral e os criados são os filhos. O escalonador de processos, parte do *kernel*, é o componente do sistema operacional que decide a ordem de execução dos processos na CPU. Ele pode usar algoritmos de escalonamento não preemptivos ou preemptivos, que determinam a política de prioridade e como os processos são escolhidos para serem executados. O *dispatcher* é responsável por realizar a troca de contexto entre os processos, salvando o estado do processo em execução e carregando o próximo processo escolhido pelo escalonador para execução.

Os escalonadores não preemptivos permitem que um processo utilize a CPU até que ele voluntariamente libere o controle. Exemplos de algoritmos não preemptivos são o *Shortest*

Job First (SJF) e o *First Come, First Served* (FCFS). Já os escalonadores preemptivos têm a capacidade de interromper um processo em execução e substituí-lo por outro com maior prioridade. Exemplos de algoritmos preemptivos são Fila de Prioridades, *Round-Robin* e Filas Multinível. Os sistemas operacionais variam quanto ao uso de escalonadores preemptivos e não preemptivos, dependendo dos requisitos e do ambiente de uso. Alguns sistemas que usam escalonadores preemptivos incluem sistemas Unix-like modernos e Windows 10. Por outro lado, sistemas mais antigos como MS-DOS e Windows 3.1 usavam escalonadores não preemptivos. O escalonador de processos é fundamental para a eficiência e justiça na alocação de recursos do sistema operacional, garantindo que vários processos possam ser executados simultaneamente e de forma adequada.

Exercícios

1) (POSCOMP 2011, Questão 41) O gerenciamento de processos em sistemas modernos é feito, quase sempre, com o uso de preempção de processos através de técnicas de compartilhamento de tempo. O que a introdução de processadores com vários núcleos altera nesse gerenciamento?

a) Torna-se possível a paralelização efetiva de processos concorrentes.
b) Torna-se possível eliminar a condição de corrida em processos concorrentes executados em paralelo.
c) Torna-se possível o uso de *threads* para a execução de processos concorrentes.
d) Torna-se possível separar os demais mecanismos de gerenciamento do sistema operacional do gerenciamento de processos.
e) Torna-se possível o uso de sistemas operacionais multitarefas.

2) (adaptado de POSCOMP 2008, Questão 53) O algoritmo FIFO (*First In, First Out*) de escalonamento de processos é inerentemente preemptivo. Verdadeiro ou falso?

3) (AOCP 2012 — BRDE — Analista de Sistemas — DBA) Em sistemas operacionais, encontramos uma série de algoritmos de escalonamento para facilitar o gerenciamento de processador. Analise as assertivas e assinale a alternativa que aponta a(s) correta(s) sobre o escalonamento *Shortest-Job- First* e o escalonamento Preemptivo.

   I. O escalonamento *Shortest-Job-First* associa cada processo (ou *job*) ao seu tempo de execução. Dessa forma, quando o processador está livre, o processo em estado

de pronto que precisar de menos tempo de UCP para terminar seu processamento é selecionado para execução.

II. O escalonamento *Shortest-Job-First* favorece os processos que executam programas menores, além de reduzir o tempo médio de espera em relação ao FIFO.

III. O escalonamento preemptivo permite que o sistema dê atenção imediata a processos mais prioritários, como no caso de sistemas de tempo real, além de proporcionar melhores tempos de respostas em sistemas de tempo compartilhado.

IV. Um algoritmo de escalonamento é dito preemptivo quando o sistema pode interromper um processo em execução para que outro processo utilize o processador.

a) Apenas I.
b) Apenas I, II e III.
c) Apenas I, III e IV.
d) Apenas II, III e IV.
e) I, II, III e IV.

4) (UFSC 2022 — Técnico de TI) Analise as afirmativas abaixo sobre escalonamento de processos em sistemas operacionais e assinale a alternativa correta.

I. No algoritmo de escalonamento por prioridades, um processo que sofre uma preempção passa imediatamente para o estado “bloqueado”.
II. O algoritmo *First Come First Served* (FCFS) é muito indicado para sistemas operacionais interativos.
III. O algoritmo *Round-Robin* reparte uniformemente o tempo da CPU entre todos os processos prontos para a execução.
IV. O Algoritmo de Envelhecimento (*Aging*) pode ser utilizado para evitar o problema da postergação indefinida existente no algoritmo de escalonamento por prioridades.

a) Somente as afirmativas III e IV estão corretas.
b) Somente as afirmativas I e III estão corretas.
c) Somente as afirmativas II e IV estão corretas.
d) Somente as afirmativas I, II e III estão corretas.
e) Somente as afirmativas I, II e IV estão corretas.

5) Considere a existência de três processos cujos dados estão abaixo:

```
Processo de ID = 1010, Chegada = 0, Execução=4
```

```
Processo de ID = 2020, Chegada = 1, Execução=3
Processo de ID = 3123, Chegada = 2, Execução=2
```

Significado:

```
ID = PID
Chegada = momento (em unidade de tempo) em que o processo está pronto.
Execução = tempo necessário para executar o processo.
```

Calcule o tempo médio de espera dos processos utilizando os algoritmos SJF e FCFS.

6) Implemente um simulador de escalonador de processos preemptivo utilizando preferencialmente a linguagem C. O simulador deve permitir a criação de processos com suas respectivas informações, como ID, tempo de chegada, tempo de execução e prioridade.

   O escalonador deve utilizar o algoritmo de escalonamento de prioridades.

   Para executar seu código, crie processos informando o ID, tempo de chegada, tempo de execução e prioridade de cada processo. O exercício pede apenas uma simulação, então criar processos é uma tarefa hipotética que consiste apenas em declarar e utilizar a estrutura criada (ou seja, não é necessário usar `fork()`).

   O simulador deve continuar a execução dos processos até que todos tenham sido concluídos.

   Responda no fim código quantas unidades de tempo foram necessárias até a conclusão do último processo e quantas unidades de tempo o processador ficou ocioso.

   Dica: Utilize a função `usleep()`, declarada em `unistd.h`, para simular o tempo de execução de cada processo.

Gabarito

1) a

2) Falso
3) e
4) a
5)

Algoritmo SJF:

Processo 1: Chegada = 0, Início da Execução = 0, Tempo de Espera = 0

Processo 3: Chegada = 2, Início da Execução = 4, Tempo de Espera = 4 - 2 = 2

Processo 2: Chegada = 1, Início da Execução = 6, Tempo de Espera = 6 - 1 = 5

Tempo médio de espera = (0 + 2 + 5) / 3 = 7 / 3 = 2,33... unidades de tempo

Algoritmo FCFS:

Processo 1: Chegada = 0, Início da Execução = 0, Tempo de Espera = 0

Processo 2: Chegada = 1, Início da Execução = 4, Tempo de Espera = 4 - 1 = 3

Processo 3: Chegada = 2, Início da Execução = 7, Tempo de Espera = 7 - 2 = 5

Tempo médio de espera = (0 + 3 + 5) / 3 = 8 / 3 = 2.66... unidades de tempo

6) Exemplo de resposta válida no arquivo `exercicio-6-aula-2.c`.

Referências

[1] SILBERSCHATZ, A., GALVIN, P. B., & GAGNE, G. (2018). Operating System Concepts (10ª ed.). Wiley.

[2] LEITE, A. Sistemas Operacionais: Escalonamento de Processos. Disponível em: http://www.univasf.edu.br/~andreza.leite/aulas/SO/ProcessosEscalonamento.pdf. Acesso em 22 jul. 2023.