Cruzeiro do Sul
Educacional

Gerenciamento de Processos

Conteudista
Prof. Me. Claudney Sanches Júnior

Revisão Textual
Aline de Fátima Camargo da Silva

Sumário

Objetivos da Unidade

- Entender e caracterizar processos em SOs;
- Aprender sobre gerenciamento de processos;
- Conhecer o escalonador e seus principais algoritmos.

Atenção, estudante! Aqui, reforçamos o acesso ao conteúdo *on-line* para que você assista à videoaula. Será muito importante para o entendimento do conteúdo.

Este arquivo PDF contém o mesmo conteúdo visto *on-line*. Sua disponibilização é para consulta *off-line* e possibilidade de impressão. No entanto, recomendamos que acesse o conteúdo *on-line* para melhor aproveitamento.

VOCÊ SABE RESPONDER?

Nem sempre um programa irá equivaler a apenas um processo, como gerenciar esses processos?

Contextualização

Figura 1 – Programador
Fonte: Vida de Programador

#ParaTodosVerem: a imagem mostra uma história em quadrinho em quatro partes. No primeiro quadrinho, há o título "Vida de Programador", com fundo preto. No segundo, há um homem, um programador, em frente a um monitor, ao telefone, com o texto "Estou tentando entrar no sistema aqui, mas está me dando a mensagem 'usuário não cadastrado no sistema'", dito pela pessoa do outro lado da linha. No terceiro, há um programador dizendo "Humm... você é uma funcionária nova?". No quarto quadrinho, a pessoa do outro lado da linha responde "sim, 23 anos..."; então, o programador cai para trás. Fim da descrição.

Gerenciador de Processos

Para resolver as dificuldades de multiprogramação e tempo compartilhado ou *multiprogramming* e *time-sharing,* há cinco grandes assuntos que todo SO moderno deve considerar. Vamos abordar o primeiro que é o gerenciamento de processos.

Os SO mutiprogramados devem proporcionar o melhor aproveitamento da CPU, quer ela tenha um processador ou vários processadores dividindo o tempo dela, para que vários programas rodem em fila, com intervalos de tempo tão pequenos e com trocas tão rápidas que acabam dando a ilusão ao usuário que todos estão sendo executados simultaneamente, independentemente se trabalham fisicamente em paralelo ou logicamente em paralelo por meio do SO.

Um programa de computador é o algoritmo escrito em uma linguagem de programação, como a linguagem C ou linguagem Java, com o objetivo de resolver um determinado problema. Quando um programa está em execução em uma CPU receberá o nome de processo. Processo é um termo mais genérico do que *job* ou *task*, introduzido para obter uma maneira sistemática de monitorar e controlar a execução de um programa. O conceito de processo é dinâmico em contraposição ao conceito de programa que é estático. O programa reside no disco, não faz nenhuma ação sem entrar em execução, enquanto que o processo reside na memória principal da máquina e está no processo de execução. O processo consiste em um programa executável associado aos seus dados e ao seu contexto de execução. Nem sempre um programa irá equivaler a apenas um processo, pois existe o SO que permite a reentrância, possibilitando a um programa gerar diversos processos. Uma característica marcante do processo é que o programa ou código que o gera não pode apresentar nenhuma característica de suposição de temporização. Em geral, o SO determina a fatia de tempo entre os processos e essa fatia de tempo é imprevisível. Um algoritmo de escalonamento determina quando um processo irá parar e outro irá entrar em execução.

O contexto de execução de um processo é o conjunto de dados necessários a sua execução, como a identificação do processo chamado de *pid*, todos os conteúdos dos registradores dos processadores, tais como o contador de programa ou simplesmente *PC - Program Counter*, os ponteiros SP - *Stack Pointer*, as variáveis e dados armazenados na memória, a lista de arquivos que são utilizados, tempo de CPU disponível, prioridade de execução, eventos que o processo pode esperar entre outros.

Essas informações são fundamentais para que um processo interrompido pelo escalonador possa voltar a executar exatamente a partir do ponto de parada sem perda de dados ou inconsistências. Elas são armazenadas em estruturas de dados conhecidas como tabela de processos ou descritor de processos, ou bloco descritor de programa. O bloco descritor de programa (BCP) consiste em uma estrutura de dados contendo informações importantes sobre o processo.

A Figura 2 apresenta o BCP contendo os dados do contexto de execução.

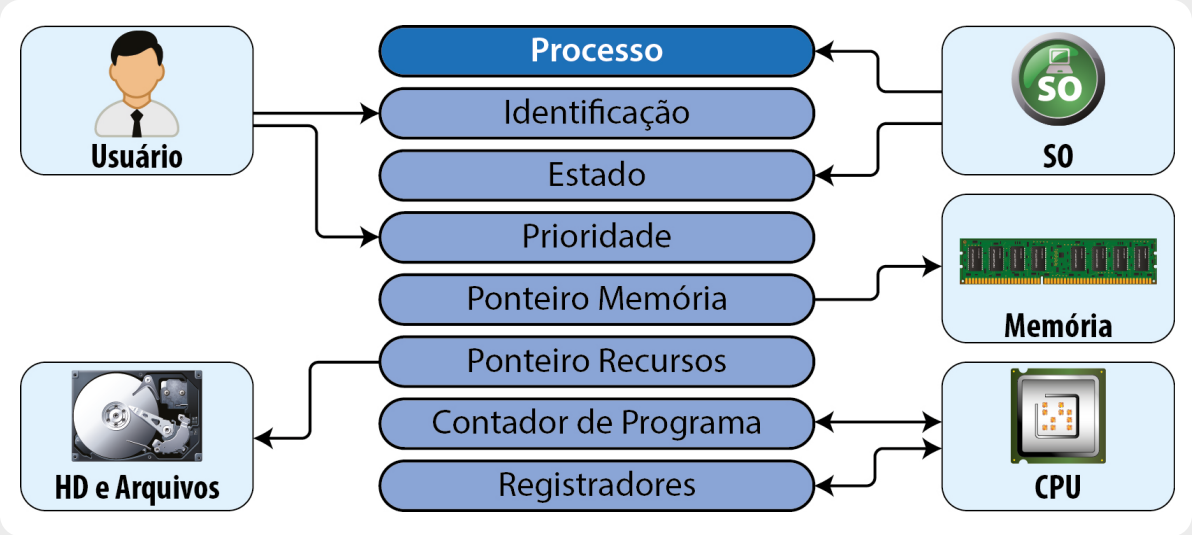


Figura 2 – BCP contendo dados do contexto de execução
Fonte: Elaborada pelo conteudista

#ParaTodosVerem: a imagem mostra um diagrama formado por 8 linhas azuis ao centro, nelas há os termos "Processo", "Identificação", "Estado", "Prioridade", "Ponteiro Memória", "Ponteiro Recursos", "Contador de Programa" e "Registradores". Ao lado esquerdo, há duas formas retangulares, a primeira, "Usuário", liga-se, por meio de flechas pretas, a "Identificação" e "Prioridade"; a segunda, "HD e Arquivos", está ligada a "Ponteiro Recursos". Do lado direito há três caixas retangulares: "SO", que se liga a "Processo" e "Estado"; "Memória", que se liga a "Ponteiro Memória"; e CPU, que se liga a "Contador de Programa" e "Registradores". Fim da descrição.

A troca entre processos concorrentes pela disputa da CPU se chama *mudança de contexto*. Quando o processo é interrompido pelo sistema operacional, seu contexto é salvo no seu BCP. Ao retornar a execução, o sistema operacional restaura o contexto do processo, o qual continua a executar como se nada tivesse ocorrido.

A Figura 3 demonstra os sistemas logicamente paralelos concorrentes em uma única CPU.

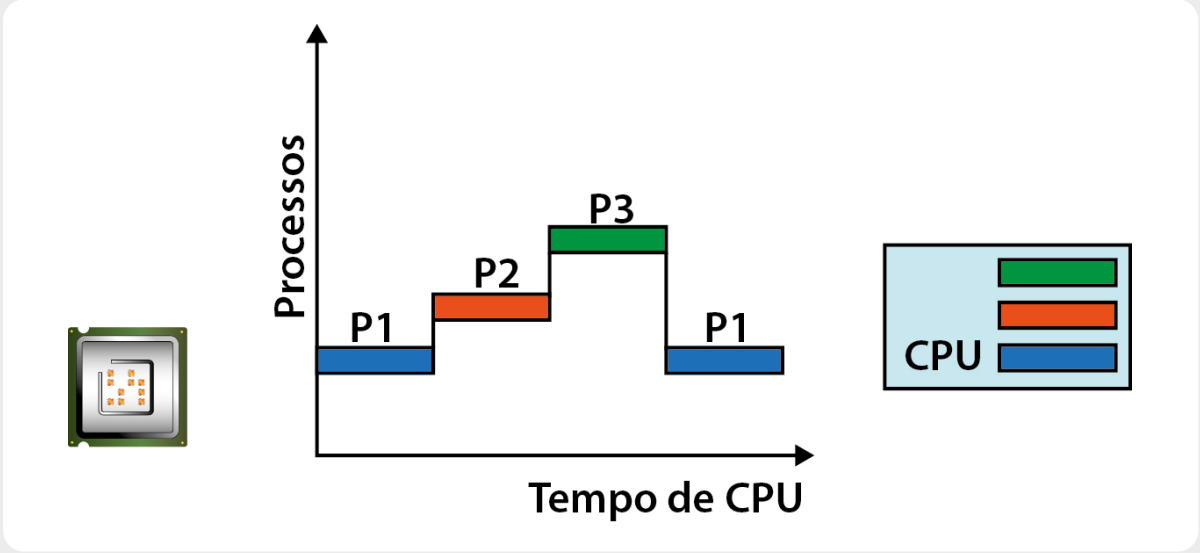


Figura 3 – Sistemas Logicamente Paralelos
Fonte: Elaborada pelo conteudista

#ParaTodosVerem: a imagem mostra um gráfico formado pelo eixo "Processo" (vertical) e "Tempo CPU" (horizontal). Ao centro, há "P1", com uma linha azul, mais acima e ao lado, "P2", com uma linha vermelha, mais acima, "P3", com uma linha verde; por fim, mais embaixo, "P1" novamente, seguindo a mesma linha do primeiro. Fim da descrição.

Um dos principais requisitos de um SO é intercalar ou fazer a mudança de contexto entre processos visando maximizar a utilização da CPU fornecendo um razoável tempo de resposta. Atualmente, tem se popularizado os computadores com múltiplas CPUs que compartilham os demais recursos da máquina, tais como memória principal e dispositivos de entrada e saída.

A Figura 4 exibe os processos em um sistema fisicamente paralelo.

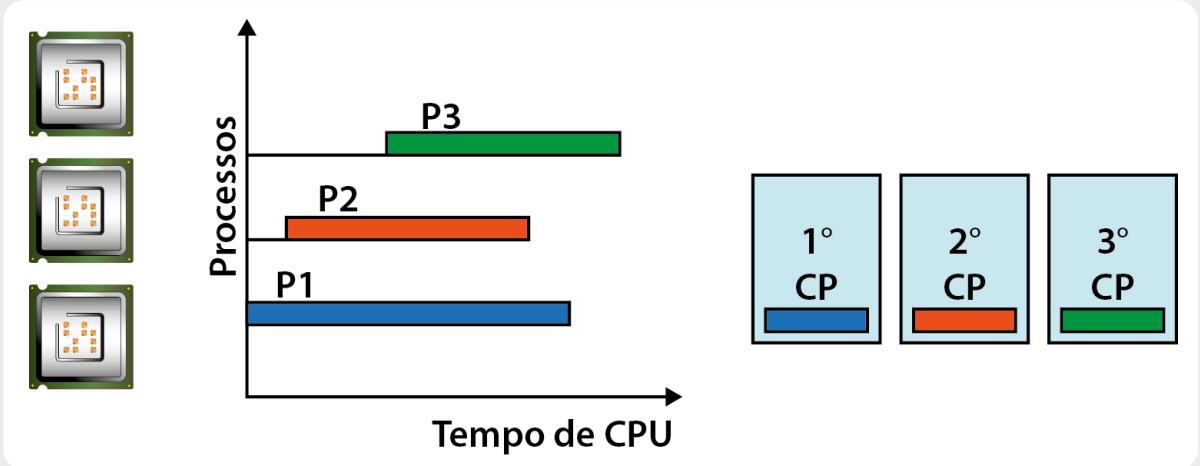


Figura 4 – Sistemas Fisicamente Paralelos
Fonte: Elaborada pelo conteudista

#ParaTodosVerem: a imagem mostra um gráfico formado pelo eixo "Processo" (vertical) e "Tempo CPU" (horizontal). No centro, há três linhas, uma acima da outra, na horizontal: P1, azul; P2, vermelha; e P3, verde. Elas não são de mesma extensão, P3 é a mais longa, seguida de P1, um pouco menor, e P2, mais curta que as duas primeiras. Fim da descrição.

O SO deve evitar que os processos entre em *deadloock* e permitir a criação e comunicação entre processos, os quais, após sua criação, podem assumir alguns estados. Os primeiros SO multiprogramados tinham a previsão de poucos estados, como o criação, o em execução, o estado pronto, o estado bloqueado e o estado encerrado, conforme ilustrado na Figura 5.

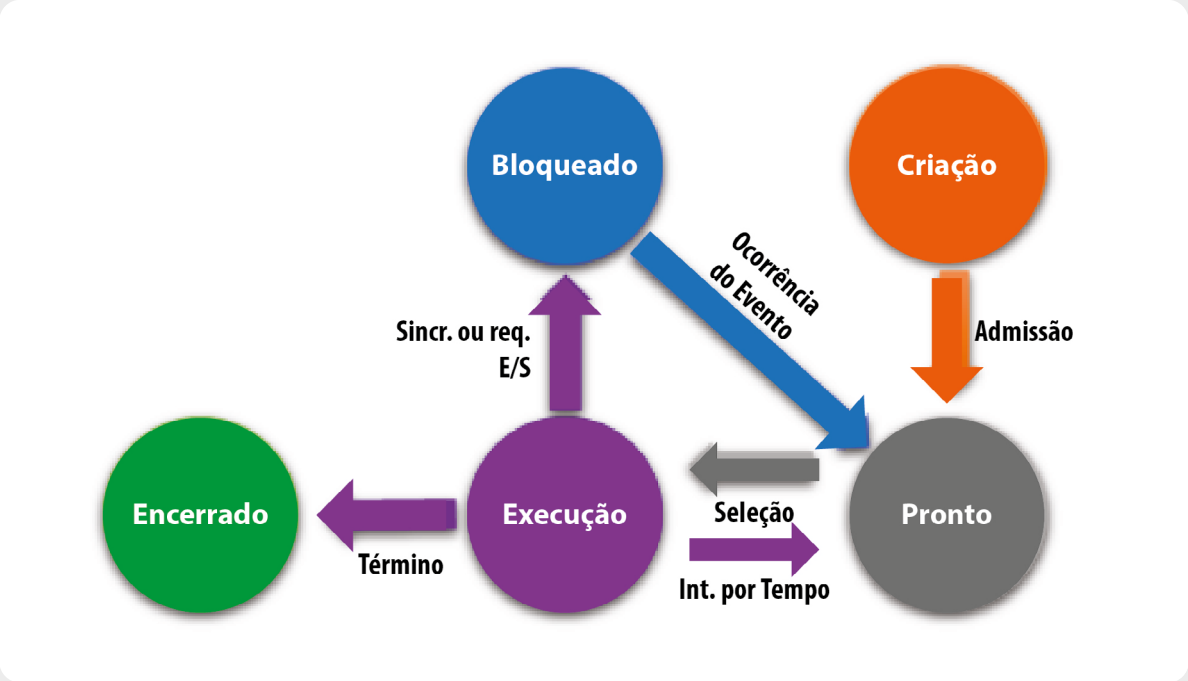


Figura 5 – Estados dos processos de um SO
Fonte: Fonte: Elaborada pelo conteudista
#ParaTodosVerem: a imagem mostra um diagrama formado por 4 círculos dispostos de modo que formam um quadrado, em sentido horário, as cores são Laranja, com o termo "Criação", cinza, com o termo "Pronto", roxo, com o termo "Execução" e azul, com o termo "Bloqueado"; do lado esquerdo, há um círculo na cor verde com o termo "Encerrado". Há flechas nas mesmas cores ligando os círculos. Fim da descrição.

Quando um usuário solicita a execução de um programa, o SO gerará um processo com a identificação do programa e deverá criar o contexto de execução a fim de poder escalonar, isto é, colocá-lo em execução. Quanto o SO conclui todos os preparativos para rodar o processo, ele muda o estado do processo de criação ou indefinido para pronto que este fica à disposição do escalonador do *kernel* do SO.

O *software* escalonador cuidará dos processos definindo quem deverá permanecer em execução. Uma das políticas para o escalonador é dar fatia de tempo iguais a todos os processos mudando o estado do processo de pronto para em execução.

Quando o processo está em execução, ele tem todos os recursos da CPU para utilizar, ou seja, está em seu estado progressivo tendo um andamento normal. Enquanto não ocorrer uma chamada de entrada e saída ou enquanto não findar sua fatia de tempo o processo deverá permanecer nesse estado.

Ao terminar sua quota de tempo para execução, o escalonador o interrompe, colocando-o no estado pronto, isto é, o processo só não está em execução pelo fato da CPU ser usada por outro processo. Ao chegar novamente sua vez de executar, o escalonador invoca ou acorda o processo permitindo sua continuidade. Um processo pode ser bloqueado se os dados de entrada ou saída, ainda, não estiverem disponíveis ou se ele estiver parado à espera da ocorrência de um evento, ou ele não poder continuar sua execução ou andamento progressivo.

O processo pode ir para o estado bloqueado informando ao escalonador que entrou em espera, simplesmente, enviando um sinal solicitando que outro processo entre em execução ou por decisão do escalonador. O processo é desbloqueado por um evento externo como a chegada dos dados ou sinalização de outro processo e ele, então, passa para o estado pronto.

Para efetuar o compartilhamento da CPU entre processos, o SO possui duas filas de controle: a de processos prontos ou *ready-list* e a de processos bloqueados ou *blocked-list*. A manipulação dessas filas depende da política de escalonamento adotada pelo sistema. Um escalonador poderia funcionar da seguinte forma: quando a CPU se torna disponível, o primeiro elemento da fila de processos prontos é retirado e inicia sua execução. Caso seja bloqueado, este irá para o final da fila de processos bloqueados. Se a sua quota de execução se esgotar, este será retirado da CPU e colocado no final da lista de processos prontos.

Saiba Mais

O escalonador ou *scheduler* do SO é o elemento responsável pela alocação de processo(s) no(s) processador(es), definindo sua ordem de execução. A política de escalonamento ou *scheduling* em inglês é um problema complexo e depende do tipo de sistema suportado e da natureza das aplicações. Em sistemas do tipo *batch*, o escalonamento era feito simplesmente selecionando o próximo processo na fila de espera. Em sistemas multiusuário de tempo repartido, geralmente combinados a sistemas em *batch*, o algoritmo de escalonamento deve ser mais complexo em virtude da existência de diversos usuários solicitando serviços e da execução de tarefas em segundo plano ou *background*.

Parâmetro de Escalonamento

Em um sistema multitarefa, vários programas compartilham a mesma CPU e, portanto, em um dado instante de tempo, somente um processo estará executando e vários processos podem estar a espera de sua fatia de tempo. Isto introduz filas de processos no estado de pronto. Quando estoura a fatia de tempo do processo em execução, o SO deve decidir qual processo da fila de prontos deverá receber a CPU.

A parte do SO responsável por decidir qual processo, dentre os prontos, deve ganhar o direito de uso da CPU é denominada escalonador de processos ou *scheduler*. O escalonador ordena a fila de prontos de acordo com sua política de escalonamento, a qual pode levar em consideração a prioridade do processo, a sua ordem de chegada no sistema, seu prazo para ser atendido etc.

O escalonamento de um SO pode ser classificado como preemptivo e não-preemptivo. Ele é dito preemptivo se o processo em execução na CPU puder ser interrompido para a execução de outro processo. A preempção é utilizada em sistemas multitarefa para garantir que todos os processos possam progredir e a fim de evitar que um processo monopolize a CPU. E o escalonamento é dito não-preemptivo se durante a execução de um processo a CPU não puder ser liberada para outro processo.

Vale dizer que inúmeros critérios devem ser considerados para a implementação de um bom algoritmo de escalonamento. Alguns deles são:

Justiça

Garantir que cada processo tenha direito de acesso a CPU;

Eficiência

Procurar maximizar a utilização da CPU;

Tempo de Resposta

Procurar minimizar o tempo de resposta para aplicações interativas. O Tempo de resposta é o tempo decorrido entre o momento no qual um usuário submete uma tarefa ao sistema e instante em que ele recebe de volta os resultados;

Throughput ou vazão

Maximizar o número de tarefas processadas por unidade de tempo;

Turnaround ou tempo de utilização da CPU

A gestão estratégica do tempo de utilização da CPU por job procura minimizar o tempo de execução das tarefas do tipo lote.

Um dos algoritmos de escalonamento mais conhecidos é o FIFO, do inglês *First In First Out*, ou primeiro a entrar na fila é o primeiro a sair, em português. Em um SO do tipo FIFO os processos vão sendo colocados na fila e retirados por ordem de chegada. A ideia fundamental é a de uma fila, em que só se pode inserir um novo elemento no final desta, bem como apenas se pode retirar o elemento do início.

Nesse algoritmo, os processos são selecionados a partir da sua ordem de chegada, ou seja, o primeiro a chegar é o primeiro a ser servido, daí o motivo de o mesmo algoritmo receber o nome de *First Come First Served* (FCFS). O mecanismo adotado pelo escalonador é não-preemptivo, isto é, as tarefas, ao conseguirem a CPU, executam até o final.

Os processos maiores fazem com que processos menores esperem em demasia e devido à igualdade total entre tarefas, aqueles mais importantes não têm acesso privilegiado à CPU. A falta de garantia quanto ao tempo de resposta o torna inadequado para sistemas interativos, porém pode ser utilizado em sistemas *Batch*.

Outro algoritmo muito conhecido para o escalonamento é *Round Robin* ou circular. É um algoritmo antigo, porém justo e simples. O processo é colocado a fim de executar na ordem em que fica pronto para execução, contudo só pode executar por uma fatia de tempo, após a qual o processo é interrompido e, caso ainda não tenha terminado sua execução, volta para o final da fila de prontos.

Um dos problemas do algoritmo *Round Robin* é determinar a fatia de tempo ou como alguns autores preferem chamar a fatia de tempo de *quantum*, o qual, por sua vez, é muito pequeno e pode levar a sucessivas trocas de contexto reduzindo a eficiência do processador. Um *quantum* muito grande pode se tornar inviável para sistemas interativos elevando o tempo de resposta.

Como nem sempre todos os processos têm a mesma prioridade, o algoritmo de escalonamento com prioridade passa a ser interessante. A sua ideia é simples, a cada processo é associada uma prioridade e o processo pronto com maior prioridade é executado primeiro.

Para evitar o monopólio do processador pelo processo com maior prioridade a cada *quantum*, o sistema deverá decrementar a prioridade do processo em execução. O escalonamento por prioridade pode ser por prioridade estáticas ou por prioridade dinâmicas. A maioria dos SO de tempo compartilhado implementa um esquema de prioridades. Pode ser conveniente, algumas vezes, agrupar os processos em classes de prioridades e usar o escalonamento entre as classes e o *round robin* dentro das classes.

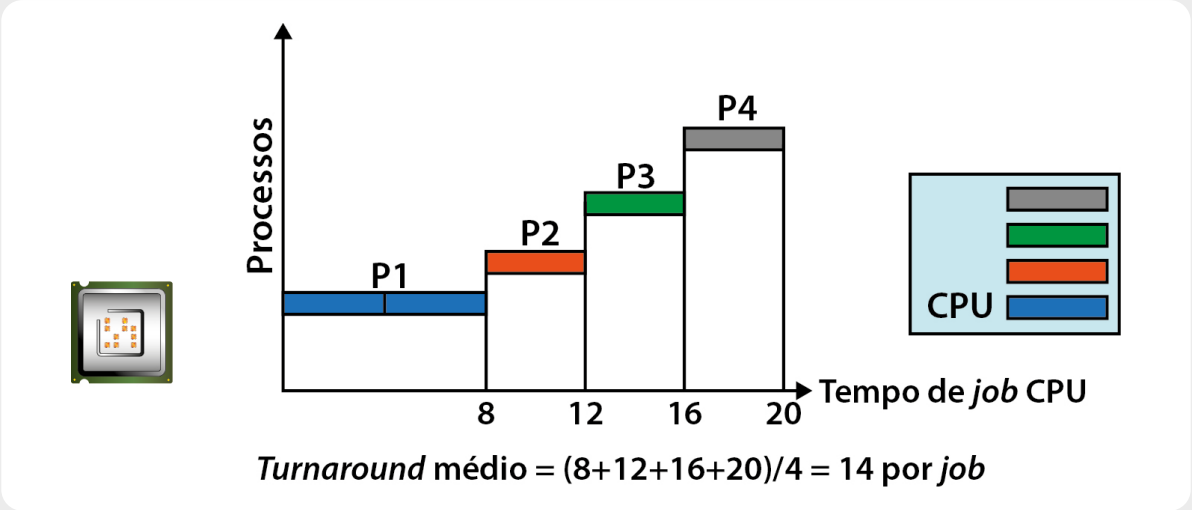
Saiba Mais

Um dos mais antigos escalonadores com prioridade foi projetado para o SO CTSS. O CTSS só mantinha na memória principal um processo pequeno, logo foi necessário atribuir um *quantum* longo para processos que utilizavam muito a CPU como uma forma de reduzir a *swap*. Para evitar o tempo de resposta ruim, o SO dividiu os processos em classes de prioridades. As classes com maior prioridade rodavam 1 *quantum*, a próxima 2 *quantum*, a outra 4 *quantum* e assim sucessivamente. Outro SO que atribuiu classes de prioridade foi o XDS 940, no qual as classes terminal e de entrada e saída tinham maior prioridade que as demais.

O algoritmo mais apropriado para sistemas que executam *jobs* em *batch* é o de menor tarefa primeiro ou *Shortest Job First* (SJF). Nesse algoritmo, o processo com menor tempo de execução previsto é o próximo escolhido para executar. Ele

é não-preemptivo e visa reduzir o tempo médio de espera das tarefas. O seu maior problema reside na estimativa prévia do tempo de execução do processo.

Alguns processos em *batch* que executam regularmente como folha de pagamento e contabilidade, tem geralmente tempos de execução conhecidos, porém processos interativos, normalmente, não usufruem dessa propriedade. Considere o caso ilustrado na Figura 6 que apresenta 4 *jobs* com seus tempos de execução e compare com o a Figura 7 que apresenta os mesmos jobs mas agora com o algoritmo da menor tarefa primeiro e observe o tempo médio menor de CPU por *job*.



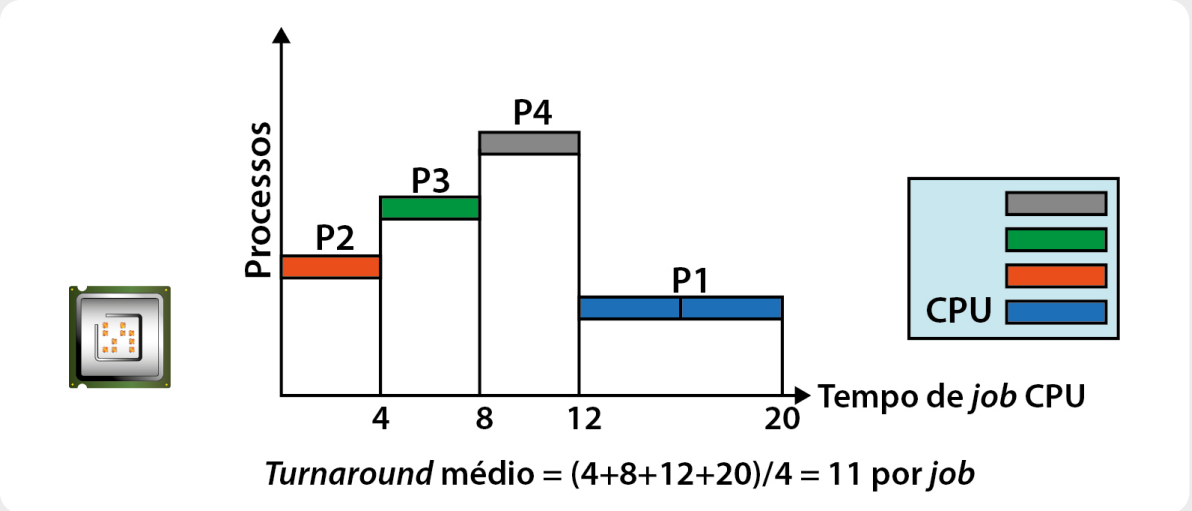


Figura 6 e 7 – Tempo médio por *job*
Fonte: Elaborada pelo conteudista

#ParaTodosVerem: as imagens mostram dois gráficos formados pelo eixo "Processo" (vertical) e "Tempo de *job* CPU" (horizontal), um em cima do outro. O primeiro gráfico tem 4 colunas em ordem crescente: P1, com linha azul, P2, com linha vermelha, P3, com linha verde e P4, com linha cinza. No segundo gráfico, há 3 colunas em ordem crescente: P2, P3 e P4, por fim, a coluna P1, menor em relação às demais. Fim da descrição.

É interessante notar que o algoritmo do menor *job* só conduz resultados ótimos se todos os *jobs* estão disponíveis ao mesmo tempo. Essa abordagem pode ser usada em sistemas interativos, tendo que considerar que cada comando enviado pelo usuário passa a ser um *job* e deverá ter o seu tempo estimado a partir de seu comportamento no passado.

Uma maneira de estimar com mais facilidade é utilizar a técnica chamada de *aging*. Suponha que o tempo para um comando de terminal na primeira vez que foi chamado seja T0. Agora, na rodada seguinte, o tempo medido será T1. Podemos atualizar nossa estimativa considerando aT0+(1-a)T1. A escolha fácil de implementar é fazer a = ½. Assim, já na quarta estimativa teríamos:

1. T0
2. T0/2+T1/2
3. T0/4+T1/4+T2/2
4. T0/8+T1/8+T2/4+T3/2

Note que o peso de T0 ou o tempo inicial com provavelmente o maior erro de estimativa caiu a 1/8 e a estimava tente a ficar mais exata a cada interação.

Para sistemas de tempo real é comum encontrar o escalonamento por prazo ou *deadline*. Os processos são escolhidos para execução em função de suas urgências. Essa política é usada em sistemas *real-time* no qual, a partir do momento em que o processo entra na fila de prontos, inicia seu prazo para receber a CPU. Caso não seja atendido dentro desse prazo pode haver perda de informação.

Os processos com escalonamento garantidos são similares aos escalonadores de tempo real, pois é feita uma promessa ao usuário a respeito do desempenho que tem de ser cumprida. Muitos SO prometem que o escalonador irá atribuir a "n" usuários ativos "1/n" da capacidade do processador.

Importante

Algumas vezes é importante separar o mecanismo de escalonamento da política de escalonamento. Por exemplo, quando um processo-pai possui vários processos-filhos, pode ser importante deixar o processo-pai interferir no escalonamento do processo filho. Para isso, deve se ter um escalonador parametrizado, cujos parâmetros são passados pelos processos do usuário.

Se não existir memória disponível no sistema, alguns processos devem ser mantidos em disco. Essa situação tem grande impacto no escalonamento devido ao atraso provocado pelo *swapping*. Uma forma prática para lidar com o *swapping* é por meio do escalonamento em dois níveis: um para o subconjunto dos processos mantidos na memória principal e outro para os processos mantidos em disco.

O escalonador de mais baixo nível se preocupa em escolher qual processo da memória usará a CPU, enquanto que o escalonador de mais alto nível se preocupa em fazer a troca dos processos da memória com os em disco.

Interrupção

Para que se possa realizar o escalonamento de processos, é necessário um mecanismo para interromper a execução do processo atualmente, usando a CPU e fazer com que esta passe a executar as instruções do SO a fim de que ele possa escalonar um novo processo. Esse mecanismo é denominado interrupção.

Portanto, interrupção é um evento gerado pelo *hardware* ou por *software* e que altera a sequência na qual o processador executa as instruções. Por exemplo, toda máquina possui um *clock*, que é um dispositivo de *hardware* que gera uma interrupção em tempos regulares no PC 18 vezes por segundo, fazendo com que a CPU pare de fazer o que fazia e execute uma rotina que incremente a hora e data do sistema. Outros tipos de interrupção são:

Interrupção de *Software*

Quando um processo solicita um serviço do SO;

Interrupção de Entrada e Saída (E/S)

Geradas por dispositivos de E/S tais como impressora, teclado, *drive* de disco, para indicar ao SO alterações no seu estado final de escrita ou leitura de dados, falha no dispositivo, entre outros;

Traps

Interrupções causadas por erros na execução de programas, tais como tentativa de divisão por zero, *overflow*, entre outras.

O tratamento de interrupções é feito pelo SO. Toda vez que uma interrupção acontece, o SO assume o controle, salva o contexto de execução do processo interrompido no seu BCP, analisa a interrupção e passa o controle para a rotina de tratamento apropriada.

Threads

A ideia da *threads* é levar para dentro das aplicações a multitarefa, permitindo que determinados trechos de código do mesmo programa possam ser executados de forma concorrente ou paralela.

No modelo tradicional, um programa é composto de uma única *thread*, ou seja, há apenas uma linha de execução e as instruções são executadas sequencialmente, do início ao final, uma após a outra. O conceito de *thread* implica a possibilidade de passar a executar vários pedaços de um processo ao mesmo tempo. Tudo se passa como se estivesse invocando a execução de uma função.

No entanto, enquanto no modelo tradicional ao chamar a função se transfere o controle de fluxo de execução para ela, quando se utiliza uma *thread* simplesmente se ativa a execução da função e prossegue a execução da função invocadora. As duas *threads*, a função chamadora e a chamada, vão ser executadas ao mesmo tempo, concorrendo pelo uso dos recursos alocados ao processo.

Material Complementar

Sites

Com as Máquinas Virtuais (VM) que você instalou na Unidade I, inicie no SO *Linux* a gerência de seus processos. O SO *Linux* apresenta um dos melhores conjuntos de algoritmos para gerenciamento de processos. Pesquise os comandos, teste e experimente gerenciar seus processos no *Linux*.

Gerenciamento de Processos no Linux
https://bit.ly/3Q9lwBG

Comandos para criar, gerenciar, monitor e eliminar processos
https://bit.ly/475grRz

Atividades de Fixação

1 – Qual é o principal objetivo do gerenciamento de processos em sistemas operacionais?

a. Alocar recursos de *hardware*, como memória e CPU, para todos os processos simultaneamente.
b. Garantir que todos os processos em um sistema operacional sejam executados em ordem alfabética.
c. Gerenciar a inicialização e desligamento do sistema operacional.
d. Controlar e coordenar a execução de múltiplos processos, permitindo uma multitarefa.
e. Manter um registro de todos os processos em execução em um sistema.

2 – Qual é a função principal de um escalonador (*scheduler*) em sistemas operacionais e qual é um exemplo de algoritmo de escalonamento?

a. A principal função de um escalonador é garantir que todos os programas sejam executados em ordem alfabética. Um exemplo de algoritmo de escalonamento é a "Ordem Alfabética".
b. A principal função de um escalonador é gerenciar a inicialização e o desligamento do sistema operacional. Um exemplo de algoritmo de escalonamento é o "Gerenciamento de Inicialização".
c. A principal função de um escalonador é controlar e coordenar a execução de múltiplos processos, permitindo uma multitarefa. Um exemplo de algoritmo de escalonamento é o "*First-Come, First-Served* (FCFS)".
d. A principal função de um escalonador é armazenar um registro de todos os processos em execução em um sistema operacional. Um exemplo de algoritmo de escalonamento é o "Registro de Processos".
e. A principal função de um escalonador é gerenciar a alocação de recursos de *hardware*, como memória e CPU, para todos os processos simultaneamente. Um exemplo de algoritmo de escalonamento é a "Alocação de Recursos".

Atenção, estudante! Veja o gabarito desta atividade de fixação no fim deste conteúdo.

Referências

DEITEL, H. M. **Sistemas Operacionais**. 3. ed. São Paulo: Pearson Prentice Hall, 2005.

TANENBAUM, A. S. **Sistemas Operacionais Modernos**. 3. ed. São Paulo: Pearson Prentice Hall, 2009.

Gabarito

Questão 1

d) Controlar e coordenar a execução de múltiplos processos, permitindo uma multitarefa.

Justificativa: o principal objetivo do gerenciamento de processos em sistemas operacionais é controlar e coordenar a execução de processos múltiplos, permitindo uma multitarefa. Isso envolve uma alocação de recursos de *hardware*, como CPU e memória, para os processos de forma eficiente, garantindo que eles sejam executados de maneira justa e eficaz.

Questão 2

c) A principal função de um escalonador é controlar e coordenar a execução de múltiplos processos, permitindo uma multitarefa. Um exemplo de algoritmo de escalonamento é o "*First-Come, First-Served* (FCFS)".

Justificativa: a função principal de um escalonador em sistemas operacionais é controlar e coordenar a execução de múltiplos processos, permitindo uma multitarefa. Isso envolve decidir qual processo deve ser executado a seguir e alocar recursos de *hardware* de forma eficiente. Um exemplo de algoritmo de escalonamento é o "*First Come, First-Served* (FCFS)", que executa os processos na ordem em que eles chegam, sem considerar prioridades.