Aula 6 - Problemas Clássicos de IPC e Escalonamento de Processos

Sistemas Operacionais
Ciência da Computação
IFB - Campus Taguatinga

Professor João Victor de A. Oliveira

Hoje

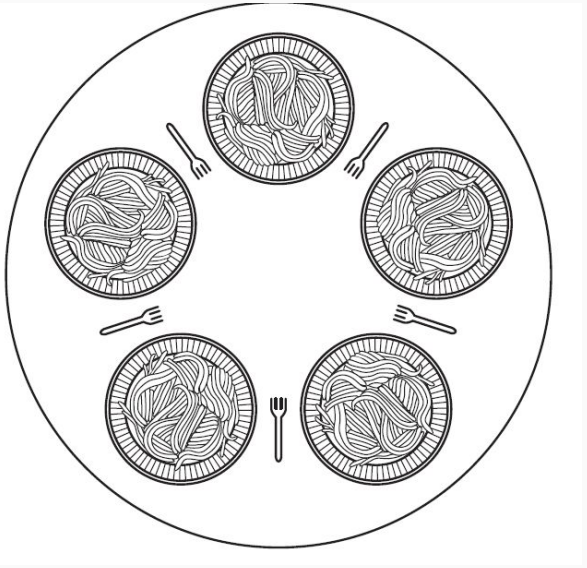
- **Problemas clássicos de IPC**
    - Problema do Jantar dos Filósofos
- **Escalonamento**
    1. Introdução
    2. Escalonamento em Sistemas em Lote
    3. Escalonamento em Sistemas Interativos

Problemas clássicos de IPC

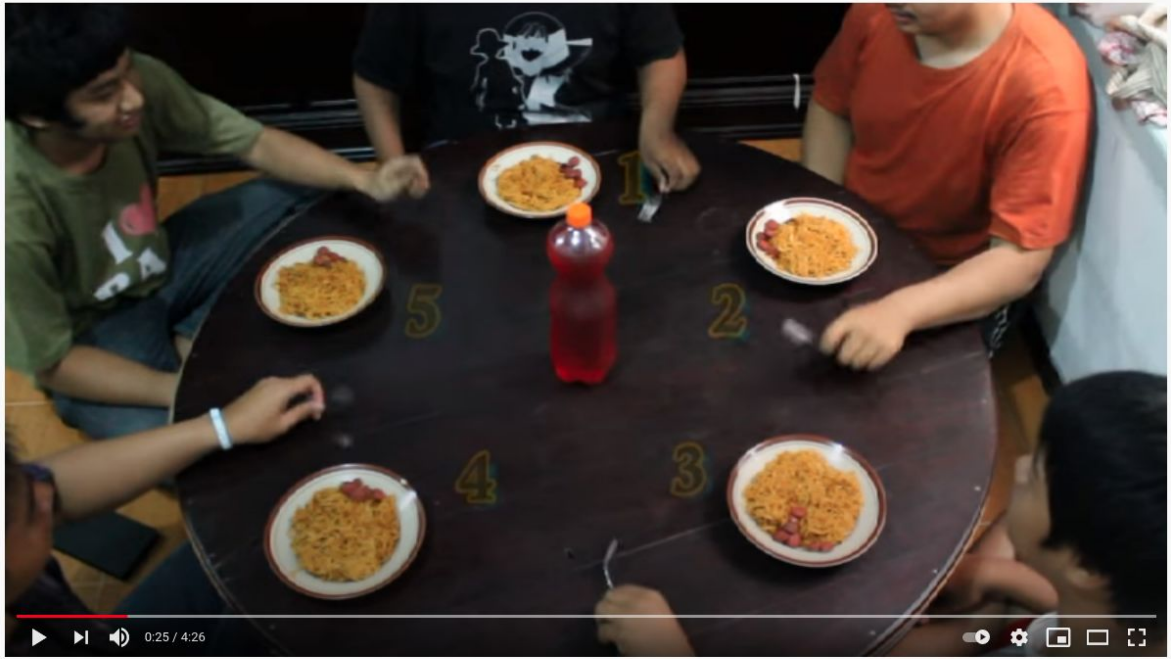
- Examinaremos dois problemas clássicos de comunicação entre processos
  - Problema do jantar dos filósofos
  - Problema dos leitores e escritores

Problema do Jantar dos Filósofos

- **Jantar dos Filósofos** [Dijkstra, 1965]
  - Cinco filósofos sentados em torno de uma mesa circular, cada um com um prato de espaguete
  - Entre cada par de pratos há um grafo
  - O espaguete é tão escorregadio que um filósofo precisa de **dois garfos para comê-lo**
  - A vida de um filósofo (nesse problema) consiste em alternar períodos de alimentação e pensamento
  - **Filósofo faminto:** Tenta pegar seus garfos à esquerda e à direita, um de cada vez, não importando a ordem
  - **Se bem-sucedido**: come por um tempo e, então, larga os garfos e volta a pensar
  - **Qual o problema deste jantar ilustre?**



Exemplo



Dining Philosophers

33.597 visualizações • 26 de dez. de 2014

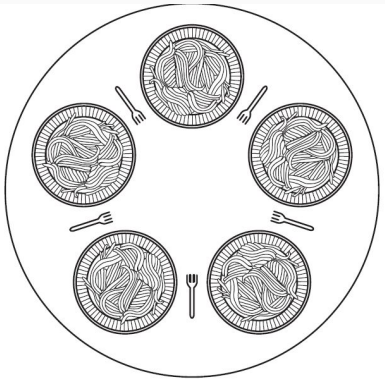
242 67 COMPARTILHAR SALVAR ...

https://www.youtube.com/watch?v=1G7HTwKvYHc

Problema do Jantar dos Filósofos

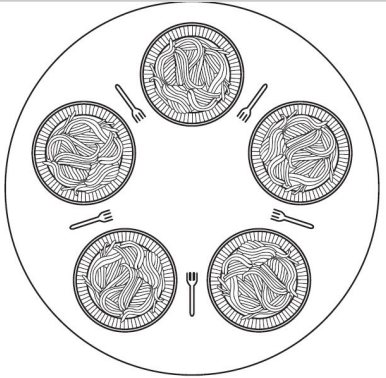
- **Solução óbvia**
  - ***Take_fork:*** espera até o garfo específico estiver disponível e então o pega
  - **Infelizmente essa solução está errada :(**
  - Suponha que todos os cinco filósofos peguem seus garfos esquerdos simultaneamente
    - **Nenhum será capaz de pegar seus garfos direitos**
  - Podemos modificar a solução de maneira que, após pegar o garfo esquerdo, o programa confere para ver se o garfo direito está disponível
    - Se não estiver, o filósofo coloca de volta o esquerdo sobre a mesa, espera por um tempo e repete todo o processo
  - **Essa proposta também fracassa!**
- Situação onde programas executam indefinidamente, mas fracassam em realizar qualquer processo é chamada de ***inanicação (starvation)***

```
#define N 5

void philosopher(int i)
{
        while (TRUE) {
                think( );
                take_fork(i);
                take_fork((i+1) % N);
                eat( );
                put_fork(i);
                put_fork((i+1) % N);
        }
}
```



Problema do Jantar dos Filósofos

- **Outra solução**
  - Fazer os filósofos esperarem um tempo aleatório em vez de ao mesmo tempo fracassarem em conseguir o garfo direito
    - Há uma chance de tudo continuar em um impasse
- **Solução que não apresenta impasse:**
  - Proteger os cinco comandos após a chamada *think* com um semáforo binário
  - Antes de pegar garfos os filósofos realizam um down em mutex
  - Após devolver os garfos ele realizaria um up em mutex
  - **Do ponto de vista teórico:** solução adequada
  - **Do ponto de vista prático:** Erro de desempenho: apenas um filósofo pode estar comendo a qualquer instante



```
#define N 5

void philosopher(int i)
{
        while (TRUE) {
                think( );
                take_fork(i);
                take_fork((i+1) % N);
                eat( );
                put_fork(i);
                put_fork((i+1) % N);
        }
}
```

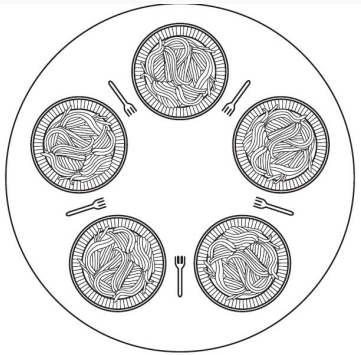
Problema do Jantar dos Filósofos

- **Solução mais buscada: permitir o máximo de paralelismo**
  - Usar um arranjo **estado** para controlar se um filósofo está comendo, pensando ou com fome
  - Um filósofo pode passar para o estado comendo apenas se nenhum de seus vizinhos estiver comendo

```
#define N           5
#define LEFT        (i+N–1)%N
#define RIGHT       (i+1)%N
#define THINKING    0
#define HUNGRY      1
#define EATING      2

typedef int semaphore;
int state[N];
semaphore mutex = 1;
semaphore s[N];

void philosopher(int i)
{
        while (TRUE) {
                think( );
                take_forks(i);
                eat( );
                put_forks(i);
        }
}
```

```
void take_forks(int i)
{
        down(&mutex);
        state[i] = HUNGRY;
        test(i);
        up(&mutex);
        down(&s[i]);
}

void put_forks(i)
{
        down(&mutex);
        state[i] = THINKING;
        test(LEFT);
        test(RIGHT);
        up(&mutex);
}
```

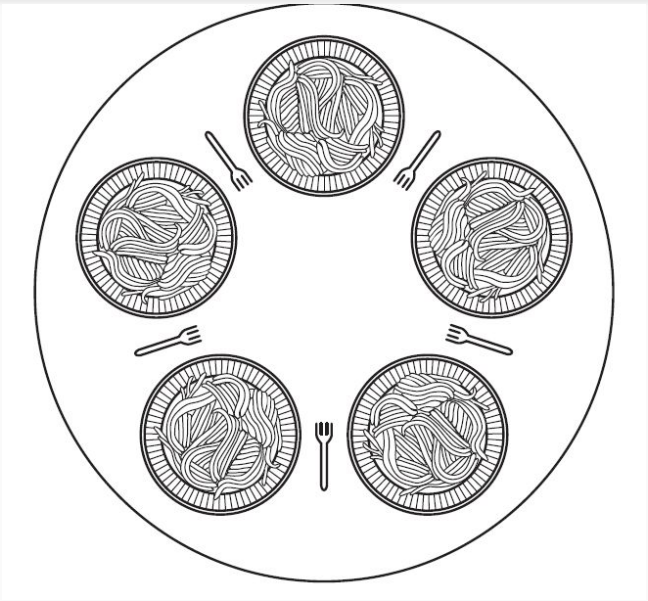
```
void test(i) /* i: philosopher number, from 0 to N–1 */
{
        if (state[i] == HUNGRY && state[LEFT] != EATING && state[RIGHT] != EATING) {
                state[i] = EATING;
                up(&s[i]);
        }
}
```



Problema do Jantar dos Filósofos

```
#define N           5
#define LEFT        (i+N–1)%N
#define RIGHT       (i+1)%N
#define THINKING    0
#define HUNGRY      1
#define EATING      2

typedef int semaphore;
int state[N];
semaphore mutex = 1;
semaphore s[N];


void philosopher(int i)
{
        while (TRUE) {
                think( );
                take_forks(i);
                eat( );
                put_forks(i);
        }
}
```

```
void take_forks(int i)
{
        down(&mutex);
        state[i] = HUNGRY;
        test(i);
        up(&mutex);
        down(&s[i]);
}

void put_forks(i)
{
        down(&mutex);
        state[i] = THINKING;
        test(LEFT);
        test(RIGHT);
        up(&mutex);
}
```



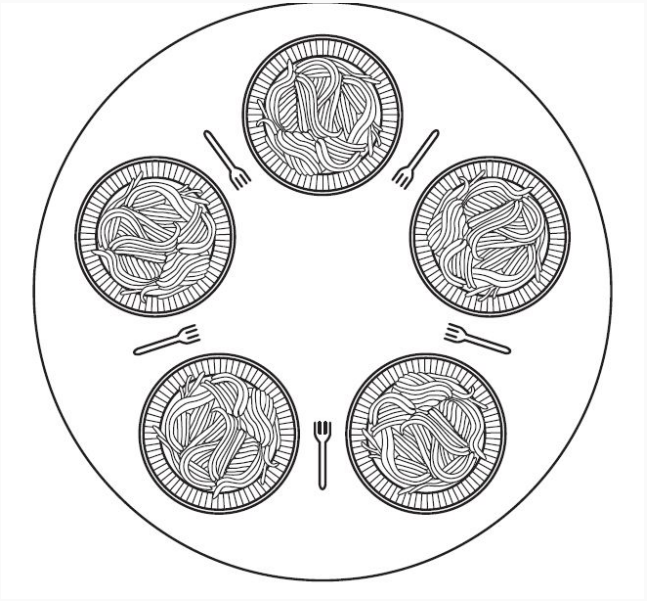
```
void test(i) /* i: philosopher number, from 0 to N–1 */
{
        if (state[i] == HUNGRY && state[LEFT] != EATING && state[RIGHT] != EATING) {
                state[i] = EATING;
                up(&s[i]);
        }
}
```

Exercício 1

No problema do jantar dos filósofos, deixe o protocolo a seguir ser usado:

- Um filósofo de número par sempre pega o seu garfo esquerdo antes de pegar o direito
- Um filósofo de número ímpar sempre pega o garfo direito antes de pegar o esquerdo.

Esse protocolo vai garantir uma operação sem impasse?



Escalonamento

Escalonamento

- Múltiplos processos ou threads competem a todo momento pela CPU em computadores multiprogramado
  - Uma **escolha** precisa ser feita sobre qual processo em estado **pronto** será executado pela CPU
  - A parte do SO que faz a escolha se chama **escalonador** e o algoritmo que ele usa para isso é chamado de **Algoritmo de Escalonamento**

Introdução

- **Escalonamento na história dos SOs**
  - **Sistemas em lote antigos**
    - Execute o processo que chegou primeiro na fita
  - **Multiprogramação**
    - Múltiplos usuários esperando pelo serviço
    - Alguns computadores de grande porte combinavam processos em lote e processos executados interativamente por usuários
      - Necessidade do escalonador decidir quem irá usar a CPU primeiro
      - CPU ainda era um recurso escasso nessas máquinas
    - **Um bom escalonador pode fazer uma grande diferença no desempenho e na satisfação do usuário**

Introdução

- **Duas mudanças com os Computadores Pessoais**
  1. Maior parte do tempo **há apenas um processo ativo (na percepção do usuário)**
  2. **Computadores tornaram-se tão rápidos que a CPU dificilmente ainda é um recurso escasso**
     i. Maioria dos programas é limitada por E/S e não pela taxa na qual o CPU pode processar
     ii. Ao executar mais de um processo, provavelmente pouco importa qual processo irá ser processado primeiro, pois o usuário está esperando que ambos terminem

Introdução

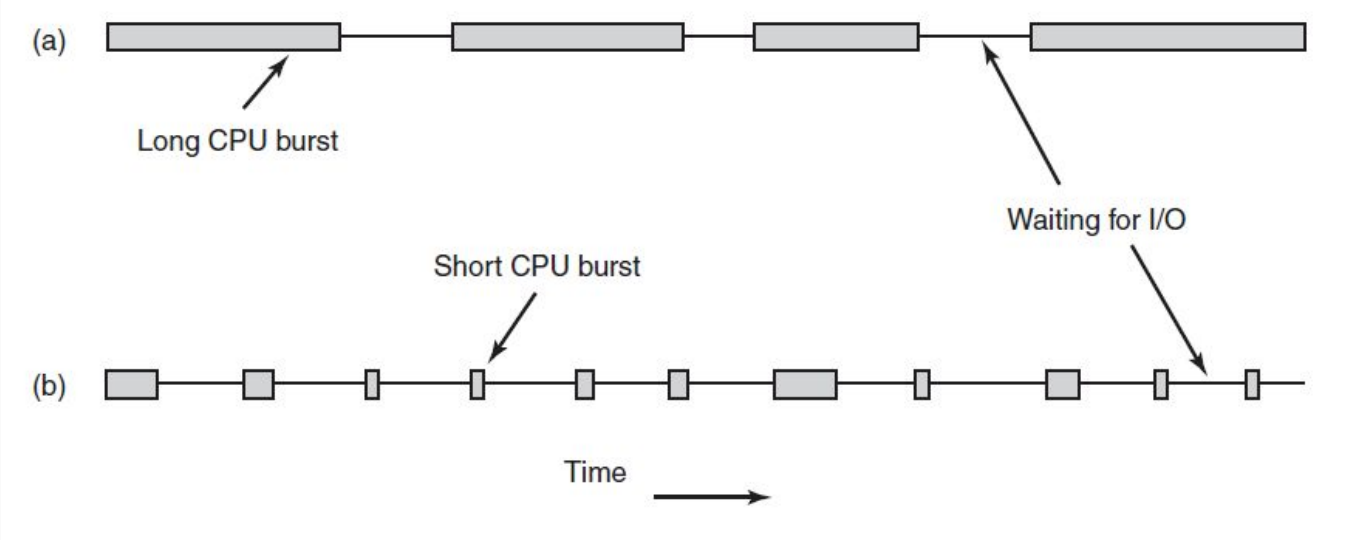
- **Servidores de Rede**
  - Múltiplos processos competindo pela CPU (**Escalonamento importante)**
  - **Ex.:** Quando a CPU tem de escolher entre executar um processo que reúne estatísticas diárias e um que serve solicitações de usuários, Estes ficarão mais contentes se o segundo receber a primeira chance de acessar a CPU
- **Dispositivos móveis e Redes de sensores**
  - “Abundância de recursos” não pode ser usada aqui (pelo menos para smartphones mais antigos)
  - Em alguns casos escalonadores podem ser otimizados para economizar bateria

Introdução

- **Preocupação com o uso eficiente da CPU**
  - Chaveamento de processos é algo caro!
  - Realizar muitas trocas de processos por segundo pode consumir um montante substancial do tempo da CPU
  - Algoritmos de Escalonamento devem lidar com esse problema!

Comportamento de Processos

- **Surtos de uso da CPU alternam-se em surtos de computação com períodos de espera por E/S**
  a. Alguns processos passam a maior parte do tempo computando (chamados de **limitados pela computação ou limitados pela CPU)**
  b. Outros passam a maior parte esperando pela E/S (**Limitados pela E/S)**



Quando escalonar

- **Quando vale a pena tomar decisões de escalonamento?**
  1. Quando um novo processo é criado
     - Qual processo, pai ou filho, deve ser executado primeiro?
  2. Quando um processo é terminado
  3. Quando um processo bloqueia para E/S, em um semáforo ou por outra razão
  4. Quando ocorre uma interrupção de E/S
     - Cabe ao escalonador decidir se deve executar o processo que ficou pronto ou continuar com o processo que estava executando antes da interrupção, ou processar algum outro

Quando escalonar

- Se tivermos um hardware de relógio fornecendo interrupções periódicas a 50 ou 60 hz
  - Uma decisão de escalonamento pode ser feita a cada interrupção ou a cada k-ésima interrupção de relógio
  - Algoritmos escalonados são divididos em duas categorias em relação a **como lidar com interrupções de relógio**
    - **Escalonamento não preemptivo**
    - **Escalonamento preemptivo**

Quando escalonar

- **Escalonamento não preemptivo**
  - Escolhe um processo para ser executado e então o deixa ser executado até que ele seja bloqueado ou que libere voluntariamente a CPU
- **Escalonamento preemptivo**
  - Escolhe um processo e o deixa executar no máximo um certo tempo fixado
  - Se ainda estiver executando ao fim do intervalo de tempo, ele é suspenso e o escalonador escolhe outro processo para executar

Categorias de algoritmos de escalonamento

- **Diferentes áreas de aplicação têm diferentes metas**
- Três ambientes de destaque
  - **Sistemas em lote**
    - Usados em tarefas rotineiras no mundo dos negócios (ex.: folha de pagamentos)
    - Não há usuários esperando impacientemente em seus terminais por uma resposta rápida a uma solicitação menor
    - Normalmente utilizam-se **algoritmos não preemptivos** (reduzir chaveamento de processos)
  - **Sistemas Interativos**
    - **Preempção é essencial** para evitar que um processo tome conta da CPU e negue serviço para os outros (Servidores web também ficam nessa categoria)
  - **Sistemas em Tempo Real**

Categorias de algoritmos de escalonamento

- Diferentes áreas de aplicação têm diferentes metas
- Três ambientes de destaque
  - **Sistemas em lote**
  - **Sistemas Interativos**
  - **Sistemas em Tempo Real**
    - Preempção as vezes não se torna necessária
    - Processos sabem que não podem executar por longos períodos e em geral, realizam sua tarefa e bloqueiam rapidamente
    - Programas em tempo real visam o progresso da aplicação como um todo (colaborativa)

Objetivos do Algoritmo de Escalonamento

- **Objetivo comum a todos os sistemas**
  - **Justiça -** dar a cada processo uma porção justa da CPU
  - **Aplicação da política -** Verificar se a política estabelecida é cumprida
  - **Equilíbrio -** Manter ocupadas todas as partes do sistema, quando possível
- **Objetivos em um sistemas em lote**
  - **Vazão (*throughput*) -** Maximizar o número de tarefas por hora
  - **Tempo de Retorno -** Minimizar o tempo médio entre a submissão e o término de processos
  - **Utilização de CPU -** Manter a CPU ocupada o tempo todo
    - Quando a utilização de CPU não é uma boa métrica nos sistemas em lote? Quando ela é útil?

Objetivos do Algoritmo de Escalonamento

- **Objetivos em Sistemas interativos**
  - **Tempo de resposta -** Responder rapidamente às requisições do usuário
    - Tempo entre emitir um comando e receber o resultado
  - **Proporcionalidade -** Satisfazer as expectativas do usuário
    - Ex.: Enviar um vídeo de 8GB para um servidor na nuvem demora mais do que executar a ação de desconectar do mesmo servidor
- **Objetivos em Sistemas de tempo real**
  - **Cumprimento dos prazos -** Evitar a perda de dados
  - **Previsibilidade -** Evitar a degradação da qualidade em sistemas multimídia

Escalonamento em sistemas em lote

- Estudaremos os seguintes algoritmos
  - Primeiro a chegar, primeiro a ser servido - ***First come, first served (FCFS)***
  - Tarefa mais curta primeiro - ***shortest job first (SJF)***
  - Tempo restante mais curto em seguida - ***shortest remaining time next***

Primeiro a chegar, primeiro a ser servido

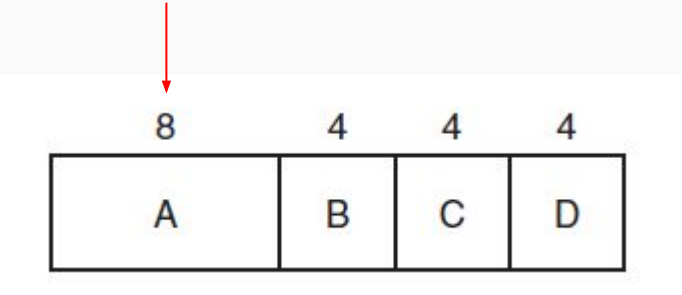
- ***First Come, First Served (FCFS)***
- **Não preemptivo**
  - O processo executa até que termine ou seja bloqueado
- **A CPU é atribuída aos processos na ordem em que requisitam**
  - Há uma fila única de processos prontos
  - A medida que outros processos chegam, eles são postos no final da fila
  - Quando um processo é bloqueado, o próximo processo é colocado em execução
    - Quando o processo bloqueado fica pronto ele é colocado no final da fila

Primeiro a chegar, primeiro a ser servido

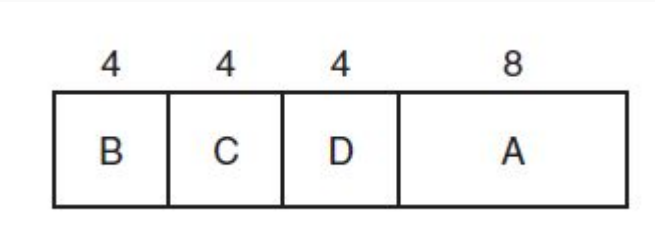
- Vantagem
  - Fácil de compreender e de implementar
    - Uma lista encadeada controla todos os processos
- Desvantagem
  - **Algoritmo não é justo!**
    - Por que?

Tarefa mais curta primeiro (SJF)

- **Não preemptivo**
- São conhecidos antecipadamente os tempos de execução dos processos
- Quando há vários trabalhos igualmente importantes esperando na fila de entrada para serem iniciados, o escalonador escolhe a **tarefa mais curta primeiro (*shortest job first*)**



Tempo de retorno = (8 + 12 + 16 + 20)/4 = 14 minutos



Tempo de retorno = (4 + 8 + 12 + 20)/4 = 11 minutos

Exercício 2

- Suponha agora que as tarefas A, B, C, D, E possuem tempo de execução de 3, 4, 2, 1 e 1, respectivamente
- Seus tempos de chegada são 0, 0, 3, 3 e 3.
- Qual será o tempo de retorno usando o algoritmo *shortest job first*?

Tempo restante mais curto em seguida

- **Versão preemptiva da tarefa mais curta primeiro**
  - **O escalonador escolhe o processo cujo tempo de execução restante é o mais curto**
  - Quando uma nova tarefa chega, seu tempo total é comparado com o tempo restante do processo atual
    - Se a nova tarefa precisa de menos tempo para terminar do que o processo atual, este é suspenso e a nova tarefa é iniciada
    - Permite que tarefas curtas novas tenham um bom desempenho

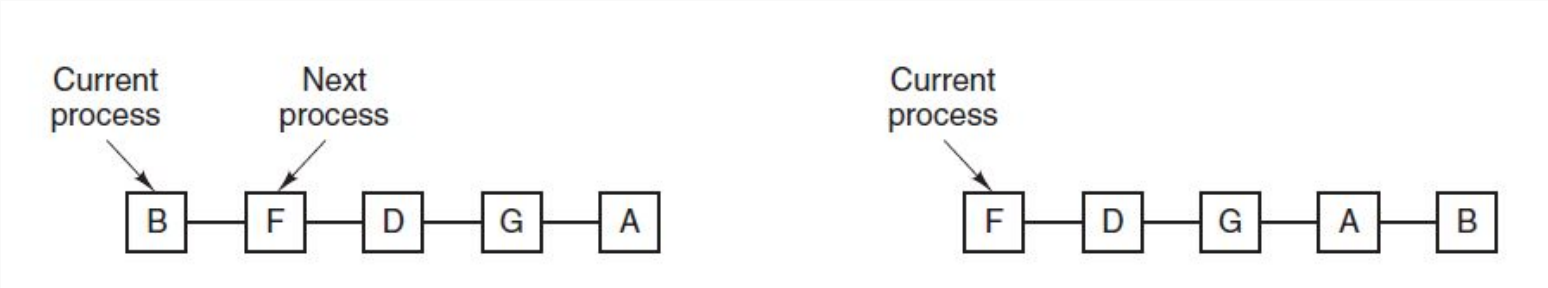
Escalonamento em Sistemas Interativos

- Comuns em computadores pessoais, servidores e outros tipos de sistemas
- **Objetivos em Sistemas interativos**
  - **Tempo de resposta -** Tempo entre emitir um comando e receber o resultado
  - **Proporcionalidade -** Satisfazer as expectativas do usuário

- **Iremos estudar os seguintes escalonamentos**
  - Escalonamento por chaveamento circular (*round robin*)
  - Escalonamento por prioridades
  - Processo mais curto em seguida

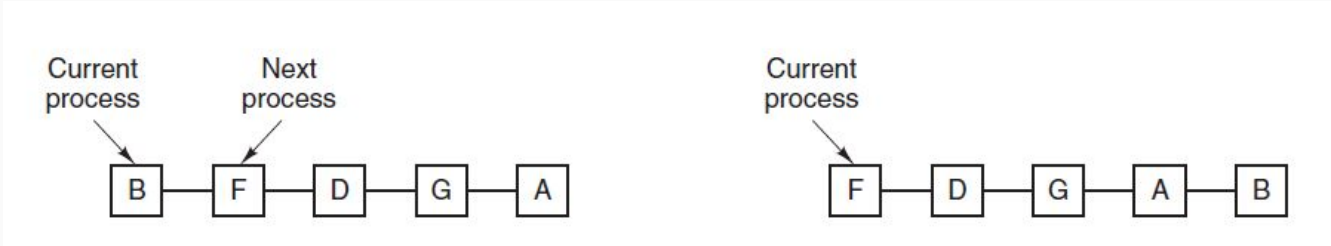
Escalonamento por Chaveamento Circular

- **Também chamado de *Round Robin***
  - A cada processo é designado um intervalo, chamado **quantum**, durante o qual ele é deixado executar
  - Se o processo ainda está executando ao fim do **quantum**, a CPU sofrerá uma **preempção** e receberá outro processo
  - Se o processo foi bloqueado ou terminado antes de o quantum ter decorrido, o chaveamento será feito quanto bloquear/terminar
  - Simples de implementar (lista de processos)



Escalonamento por Chaveamento Circular

- Questão interessante: qual o comprimento do quantum?
  - Chavear um processo para outro exige um montante de tempo para fazer toda a administração
    - Salvando e carregando registradores, mapas de memória, atualizando tabelas e listas, carregando e descarregando memória cache, dentre outros
  - Suponha que um **chaveamento entre processos leva 1 ms**
  - Suponha também que o **quantum é estabelecido em 4 ms**
    - Com isso, após realizar 4 ms de trabalho útil, a CPU deverá gastar 1ms no chaveamento de processo
    - 20% do tempo de CPU será jogado fora em overhead administrativo!



Escalonamento por Chaveamento Circular

- Questão interessante: qual o comprimento do quantum?
  - Melhoramos a eficiência agora, definindo um quantum de 100 ms
    - Tempo desperdiçado caiu para 1%
  - **Imagine agora um sistema servidor com 50 solicitações em um intervalo curto de tempo**
    - 50 processos na lista de processos executáveis
    - O último da fila (azarado) pode ter que esperar 5 segundos antes de ter uma chance, caso todos usem todo o seus quantuns
      - Com um quantum curto, o usuário dessa solicitação poderia ter recebido uma resposta em um tempo menor
  - **Normalmente um quantum de 20 a 50 ms é uma escolha bastante razoável**

Escalonamento por Prioridades

- **Escalonamento circular: todos os processos são de igual importância**
  - O que nem sempre é verdade!
- **Escalonamento por prioridades**
  - A cada processo é designada uma prioridade
    - o processo executável com a prioridade mais alta é autorizado a executar
  - Para evitar que processos de prioridade mais alta executem indefinidamente, o escalonador talvez diminua a prioridade do processo que está sendo executado em cada tique do relógio (interrupção de relógio)
  - **Alternativa:** a cada processo é designado um quantum de tempo máximo no qual ele é autorizado a executar
    - Quando esse quantum for esgotado, o processo seguinte na escala de prioridade recebe uma chance de ser executado

Escalonamento por Prioridades

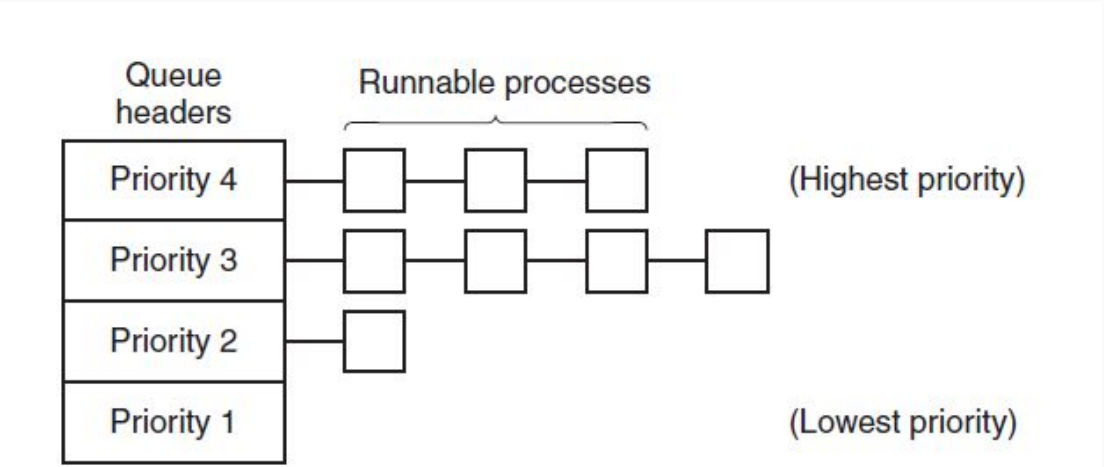
- Prioridades podem ser designadas a processos por dois tipos
  - **Prioridade estática**
    - Ex.: Em um computador militar, processos iniciados por generais recebem com prioridade 100, processos iniciados por coronéis com 90, por majores 80, capitães por 70 e assim por diante
    - Unix usa o comando **nice**, que permite que um usuário reduza/aumente voluntariamente a prioridade de seu processo
  - **Prioridade dinâmica**
    - Ex.: Alguns processos são altamente limitados pela E/S e passam a maior parte do tempo esperando a E/S ser concluída
      - Sempre que um processo assim quer a CPU, deve recebê-la imediatamente

Escalonamento por Prioridades

- **Prioridade dinâmica**
  - **Algoritmo simples para proporcionar um bom serviço para processos limitados pela E/S:**
    - Configurar a prioridade para **1/*f*,** onde *f* é a fração do último quantum que o processo usou
    - Ex.:
      - Um processo que usou apenas 1ms do seu quantum de 50 ms receberia prioridade 50, enquanto um processo que usasse 25ms receberia prioridade 2 e um que usasse o quantum inteiro receberia a prioridade 1

Escalonamento por Prioridades

- **Agrupando processos em classes de prioridades**
  - Desde que existam processos executáveis na classe de prioridade 4, apenas execute cada um por um quantum, estilo circular e jamais se importe com classes de prioridades mais baixas
  - Se a classe 4 estiver vazia, então execute os processos de classe 3 de maneira circular (e assim por diante)
  - **Se as classes de prioridades não forem ajustadas, as classes de prioridade mais baixas podem todas morrer famintas!!!!**



Processo mais curto em seguida

- **Tarefa mais curta primeiro *(shortest job first)*** sempre produz o tempo de resposta médio mínimo para sistemas em lote
  - Seria bom usarmos ela em processos interativos (Executa um comando e espera pelo comando)
  - Problema, qual é a tarefa mais curta em um sistema interativo?
    - **Uma abordagem é fazer estimativas baseadas no comportamento passado**
    - Após isso, executar o processo com tempo de execução estimado mais curto

Processo mais curto em seguida

- **Fazer estimativas baseadas no comportamento passado**
  - Suponha que o tempo estimado por comando para alguns processos é T_0
  - Agora que a operação seguinte é mensurada como sendo T_1
  - Podemos, então, atualizar nossa estimativa tomando a soma ponderada desses dois números, isto é, $aT_0 + (1-a)T_1$, *onde* **a** é usado para decidir o quão rápido o algoritmo esqueça de execuções anteriores
  - Com a=½ temos estimativas sucessivas de:
    - $T_0 \Rightarrow T_0/2 + T_1/2 \Rightarrow T_0/4 + T_1/4 + T_2/2 \Rightarrow T_0/8 + T_1/8 + T_2/4 + T_3/2$
  - Após 3 novas execuções o peso de T_0 caiu para ⅛
  - A técnica de estimar o valor seguinte em uma série, tomando a média ponderada do valor mensurado atual e a estimativa anterior é chamada de **envelhecimento (aging)**

Na próxima aula

- Aula de Exercícios/Revisão para a prova

Referências

Tanenbaum, A. S. e Bos, H.. Sistemas Operacionais Modernos. 4.ed. Pearson/Prentice-Hall. 2016.