Fundamentos de Sistemas Operacionais I

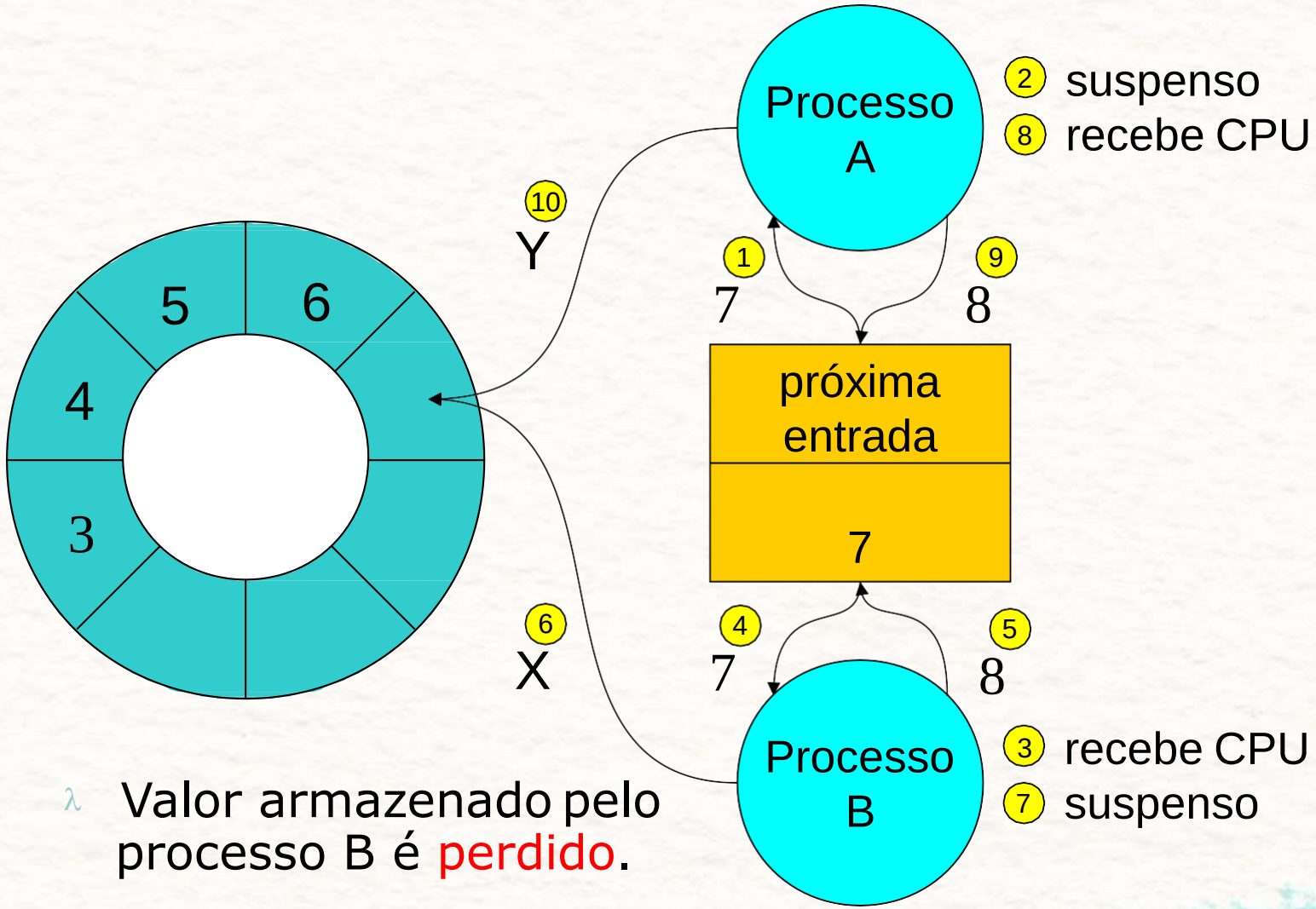
Prof. Me. Paulo Sérgio Germano

Condições de Corrida (Disputa)

- Condições de Corrida:
  - Situações onde dois ou mais processos estão acessando dados compartilhados.
  - O resultado final pode variar de acordo com a ordem de execução.

- Mecanismo de Sincronização.
  - Garante o compartilhamento de recursos e a comunicação entre os processos.
  - Garante a integridade e a confiabilidade dos dados compartilhados.

Condições de Corrida (Exemplo)



- Valor armazenado pelo processo B é perdido.

Condições de Corrida (Região Crítica)

Como evitamos condições de corrida? A chave para evitar problemas aqui e em muitas outras situações que envolvem memória compartilhada, arquivos compartilhados e tudo o mais compartilhado é encontrar alguma maneira de proibir mais de um processo de ler e escrever os dados compartilhados ao mesmo tempo.

Em outras palavras, o que precisamos é de **exclusão mútua** de execução, isto é, alguma maneira de garantir que, se um processo estiver usando uma variável, ou um arquivo compartilhado, os demais serão impedidos de fazer a mesma coisa.

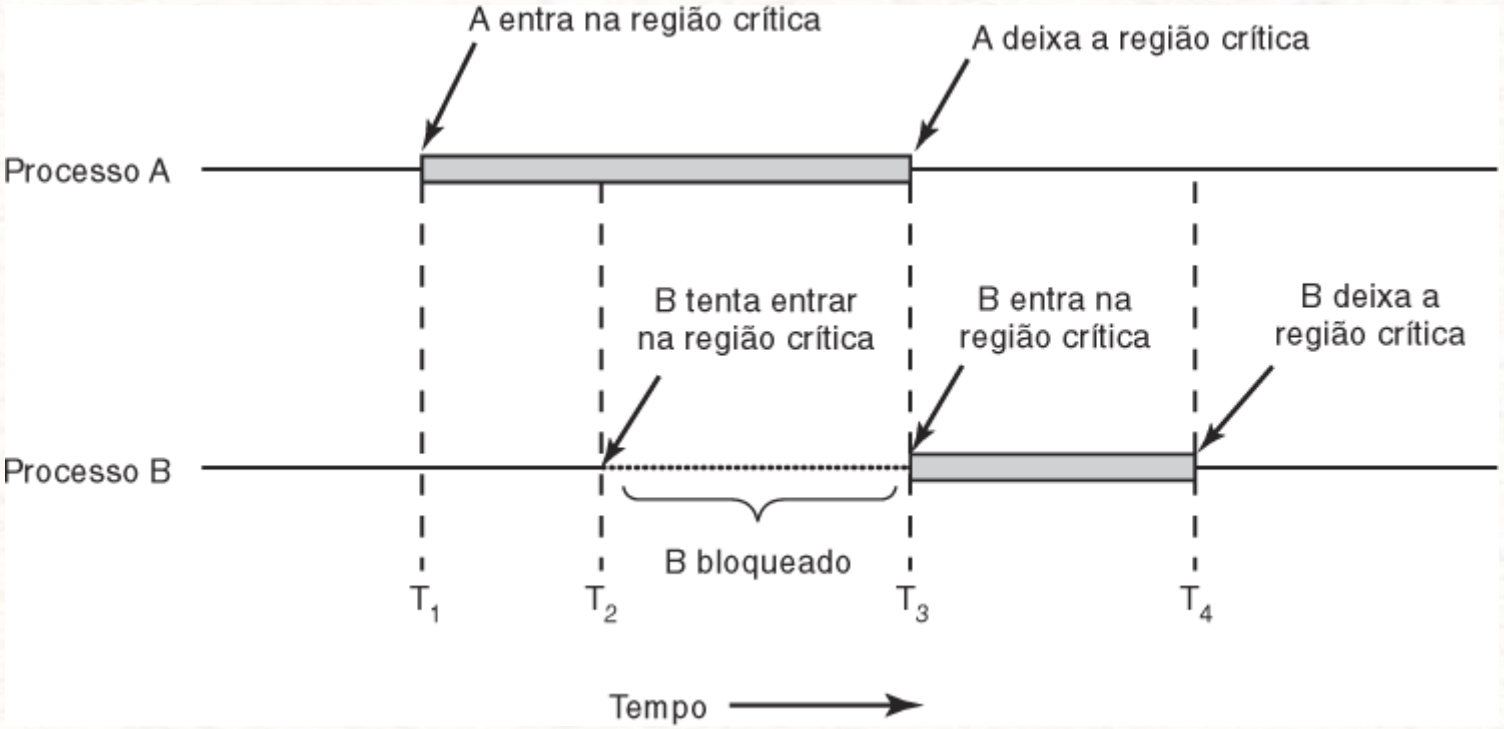
A parte do programa em que a memória compartilhada é acessada é chamada de região crítica ou seção crítica.

Condições de Corrida (Região Crítica)

Precisamos de quatro condições para ter uma boa solução das corridas em regiões críticas:

1. Nenhum processo pode estar simultaneamente dentro de suas regiões críticas.
2. Nenhuma suposição pode ser feita sobre velocidades ou o número de CPUs.
3. Nenhum processo em execução fora de sua região crítica pode bloquear qualquer processo.
4. Nenhum processo deveria ter que esperar para sempre entrar em sua região crítica.

Região Crítica – Exemplo de exclusão mútua



Aqui o processo A entra em sua região crítica no tempo T1. Um pouco mais tarde, no tempo T2, o processo B tenta entrar em sua região crítica, mas falha porque outro processo já está em sua região crítica e permitimos apenas um de cada vez. Conseqüentemente, B está temporariamente suspenso até o tempo T3 quando A deixa sua região crítica, permitindo que B entre imediatamente. Eventualmente B sai (em T4) e estamos de volta à situação original sem nenhum processo em suas regiões críticas.

Soluções de Hardware

Desabilitação das interrupções:

- Solução mais simples para a exclusão mútua;
- Falha de Proteção:
  - O processo precisa voltar a habilitar as interrupções.
- Não aplicável para múltiplas CPUs.
- Este recurso só deve ser permitido ao SO.

Instrução Test-and-Set:

- Utilização de uma variável para testar a possibilidade de executar a região crítica.
- O teste e o bloqueio é realizado através de uma única instrução (TSL).
- O processo impedido de executar sua região crítica executa um loop de espera.
  - Gasto de CPU.

Soluções de Software

Em geral, as soluções por software resolvem a exclusão mútua, mas geram a espera ocupada (Busy Wait):

- Teste contínuo de uma variável até que ocorra uma mudança no seu valor;
- O processo impedido de executar sua região crítica executa um loop de espera.
  - Gasto de CPU.

Soluções de Software (Desativando Interrupções)

Em um sistema de processador único, a solução mais simples é fazer com que cada processo desative todas as interrupções logo após entrar em sua região crítica e reativá-las antes de sair dela. Com interrupções desativadas, nenhuma interrupção de clock pode ocorrer.
Com interrupções desligadas a CPU não será comutada para outro processo. Assim, uma vez que um processo tenha desativado interrupções, ele pode examinar e atualizar a memória compartilhada sem medo de que qualquer outro processo intervenha.

Falhas desta abordagem:

- É arriscado dar aos processos do usuário o poder de desligar as interrupções. E se um deles fizesse isso e nunca mais os ligasse? Esse poderia ser o fim do sistema;

- Em multiprocessadores (com duas ou mais CPUs), desabilitar as interrupções afetará apenas a CPU que executou a instrução de desabilitação.

- É conveniente que o próprio kernel desabilite interrupções para algumas instruções enquanto atualiza variáveis ou especialmente listas.

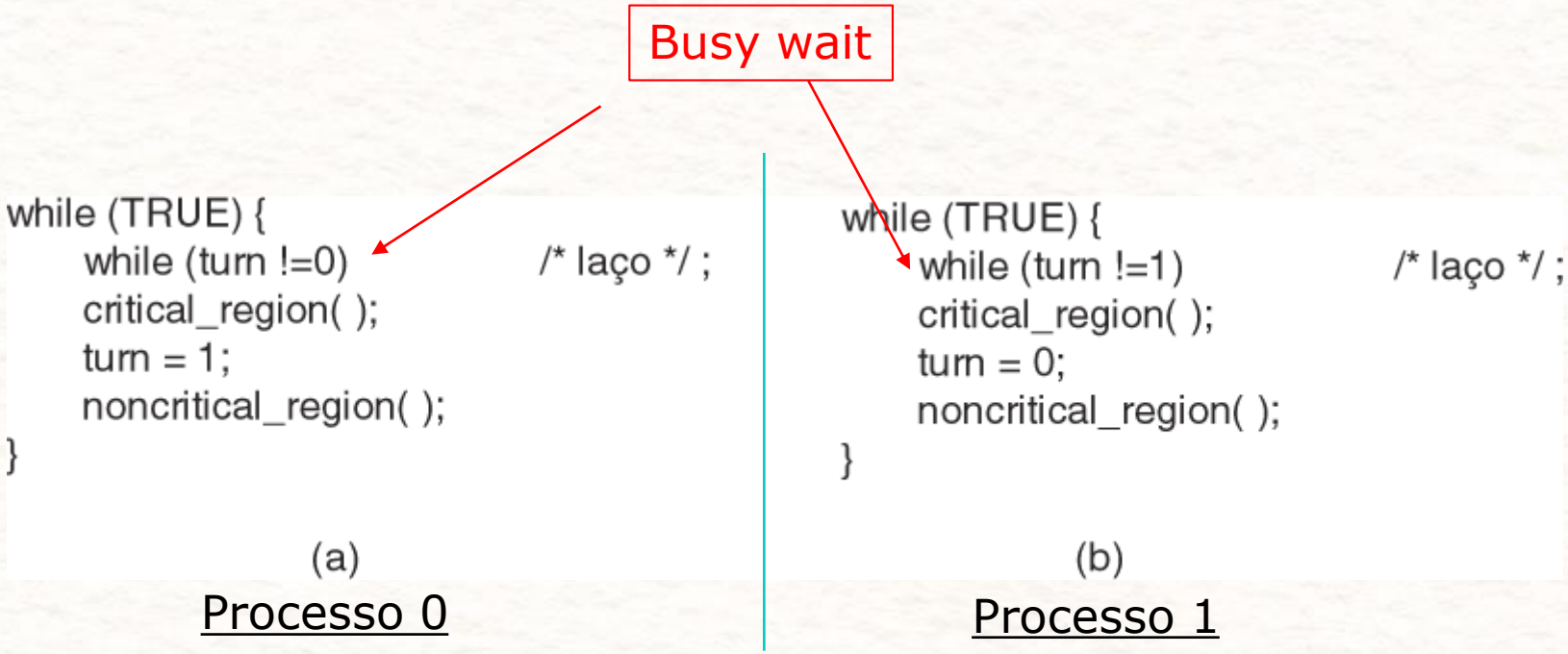
Soluções de Software (Variáveis de Bloqueio)

Considere ter apenas uma variável compartilhada (lock), inicialmente 0. Quando um processo deseja inserir sua região crítica, ele primeiro testa o bloqueio. Se o bloqueio for 0, o processo o define como 1 e entra na região crítica. Se o bloqueio já é 1, o processo apenas espera até que ele se torne 0. Assim, um 0 significa que nenhum processo está em sua região crítica e um 1 significa que algum processo está em sua região crítica.

Essa abordagem contém exatamente a mesma falha fatal que vimos no diretório do spooler.

Suponha que um processo leia o bloqueio e veja que ele é 0. Antes de definir o bloqueio como 1, outro processo é planejado, executado e define o bloqueio para 1. Quando o primeiro processo for executado novamente, ele também definirá o bloqueio como 1 e dois processos estarão em suas regiões críticas ao mesmo tempo.

Soluções de Software (Alternância Estrita)



- Embora esse algoritmo evite todas os problemas, ele não é a melhor solução porque viola a condição 3 - Nenhum processo em execução fora de sua região crítica pode bloquear qualquer processo.

Soluções de Peterson

Em 1981, G. L. Peterson descobriu uma maneira muito mais simples de alcançar a exclusão mútua. Consiste de dois procedimentos básicos:

- Chamada das rotinas antes de entrar e após sair da região crítica;
- A rotina de entrada só retorna quando não houver outro processo executando na sua região crítica.

Soluções de Peterson

```
#define FALSE  0
#define TRUE   1
#define N      2                        /* número de processos */

int turn;                               /* de quem é a vez? */
int interested[N];                      /* todos os valores inicialmente em 0 (FALSE) */

void enter_region(int process);         /* processo é 0 ou 1 */
{
     int other;                        /* número de outro processo */

     other = 1 – process;               /* o oposto do processo */
     interested[process] = TRUE;        /* mostra que você está interessado */
     turn = process;                    /* altera o valor de turn */
     while (turn == process && interested[other] == TRUE) //* comando nulo */;
}

void leave_region(int process)          /* processo: quem está saindo */
{
     interested[process] = FALSE;       /* indica a saída da região crítica */
}
```

Soluções de Peterson

A solução da Peterson têm o defeito de exigir um “Busy Waiting”. Em essência o que essa solução faz é: quando um processo deseja entrar em sua região crítica, ele verifica se a entrada é permitida. Se não for, o processo apenas fica em um laço, esperando até seja permitida.

Essa abordagem não apenas desperdiça tempo de CPU, mas também pode ter efeitos inesperados. Considere um computador com dois processos, H, com alta prioridade, e L, com baixa prioridade. As regras de agendamento são tais que H é executado sempre que estiver no estado pronto. Em um determinado momento, com L em sua região crítica, H fica pronto para ser executado (por exemplo, uma operação de E / S é concluída). H agora começa a esperar, mas como L nunca está programado enquanto H está rodando, L nunca tem a chance de deixar sua região crítica, então H faz um loop para sempre. Esta situação é por vezes referida como o problema de inversão de prioridade.

Soluções Sleep e Wakeup

Agora, vamos examinar algumas primitivas de comunicação entre processos que bloqueiam em vez de desperdiçar tempo de CPU quando elas não podem entrar em suas regiões críticas. Duas das mais simples são “sleep” e “wakeup”. “Sleep” é uma chamada do sistema que faz com que processo bloqueie, isto é, seja suspenso até que outro processo o acorde. A chamada de “wakeup” tem como parâmetro, o processo a ser despertado. Como alternativa, tanto o modo de espera quanto o de ativação têm como parâmetro, um endereço de memória usado para combinar “sleeps” com “wakeups”.

Sleep:

- Coloca o processo chamador no estado de espera.

Wakeup:

- Coloca um outro processo que está em espera no estado de pronto.

Semáfaros

Ferramenta de sincronização criada por Dijkstra (1965) sugeriu o uso de uma variável inteira, não negativa, para contar o número de "wakeups" salvos para uso futuro. Em sua proposta, um novo tipo de variável, que ele chamou de semáforo, foi introduzido. Um semáforo poderia ter o valor 0, indicando que nenhuma ativação foi salva ou algum valor positivo se um ou mais wakeups estiverem pendentes.

Manipulados exclusivamente por duas operações atômicas:

- DOWN (generalização do sleep)
- UP (generalização do wakeup)
- Implementados como chamada ao sistema.

Mutexes

Um mutex é uma variável compartilhada que pode estar em um dos dois estados: desbloqueado ou bloqueado.

- Simplificação do Semáforo;
- Mutexes são bons apenas para gerenciar a exclusão mútua de algum recurso compartilhado ou pedaço de código.
- Eles são fáceis e eficientes de implementar, o que os torna especialmente úteis em pacotes de encadeamento implementados inteiramente no espaço do usuário.

Escalonamento de Processos (SCHEDULING)

Quando um computador é multiprogramado, ele freqüentemente tem vários processos ou threads competindo pela CPU ao mesmo tempo. Essa situação ocorre sempre que dois ou mais deles estão simultaneamente no estado pronto. Se apenas uma CPU estiver disponível, é preciso escolher qual processo executar em seguida. A parte do sistema operacional que faz a escolha é chamada de escalonador (Scheduler) e o algoritimo utilizado é chamado de Algoritmo de Escalonamento.

Categorias de algoritmos de escalonamento

Escalonamento **não preemptivo**:
Um algoritmo de agendamento sem preempção escolhe um processo para ser executado e, em seguida, permite que ele seja executado até que ele seja bloqueado (seja na E / S ou à espera de outro processo) ou libere voluntariamente a CPU. Depois que o processamento de interrupção do relógio tiver sido finalizado, o processo que estava em execução antes da interrupção é retomado, a menos que um processo de prioridade mais alta esteja aguardando.
Escalonamento **preenptivo**:
Escolhe um processo e permite que ele seja executado por um período máximo de tempo fixo. Se ainda estiver em execução no final do intervalo de tempo, ele será suspenso e o "scheduler" selecionará outro processo para ser executado.

Metas de algoritmos de escalonamento

- Todos os sistemas
  - Equidade - dar a cada processo um espaço justo.
  - Política da CPU - permitir que a política declarada seja executada.
  - Equilíbrio - manter todas as partes do sistema ocupadas.

- Sistemas de lote (batchs)
  - Taxa de transferência – maximizar o número de jobs por hora.
  - Tempo de resposta - minimizar o tempo entre o envio e a finalização da utilização.
  - CPU - manter a CPU ocupada o tempo todo.

- Sistemas interativos
  - Tempo de resposta - responder a solicitações rapidamente.
  - Proporcionalidade - atender às expectativas dos usuários.

- Sistemas em tempo real
  - Cumprindo prazos - evitar perder dados.
  - Previsibilidade - evitar a degradação da qualidade em sistemas multimídia.

Estratégias de escalonamento em Sistemas em Lote (batch)

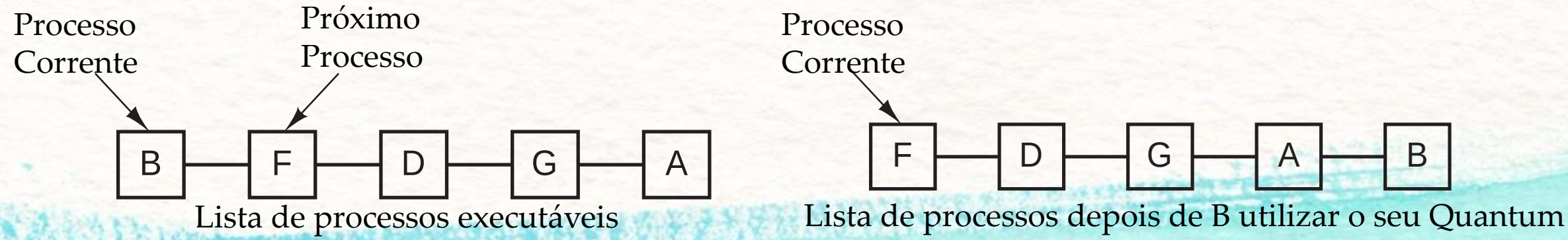
- Primeiro a chegar, Primeiro a ser servido:
  - ✓ Os processos são atribuídos à CPU na ordem em que são solicitados. Basicamente, existe uma única fila de processos prontos.
  - ✓ Fácil de entender e igualmente fácil de programar. Com este algoritmo, uma única lista encadeada controla todos os processos prontos.

- O trabalho mais curto primeiro:
  - ✓ Quando vários trabalhos igualmente importantes estão na fila de entrada aguardando para serem iniciados, o agendador escolhe o trabalho mais curto primeiro.

- O menor tempo restante:
  - ✓ Sempre escolhe o processo cujo tempo de execução restante é o mais curto. Novamente aqui, o tempo de execução deve ser conhecido antecipadamente.

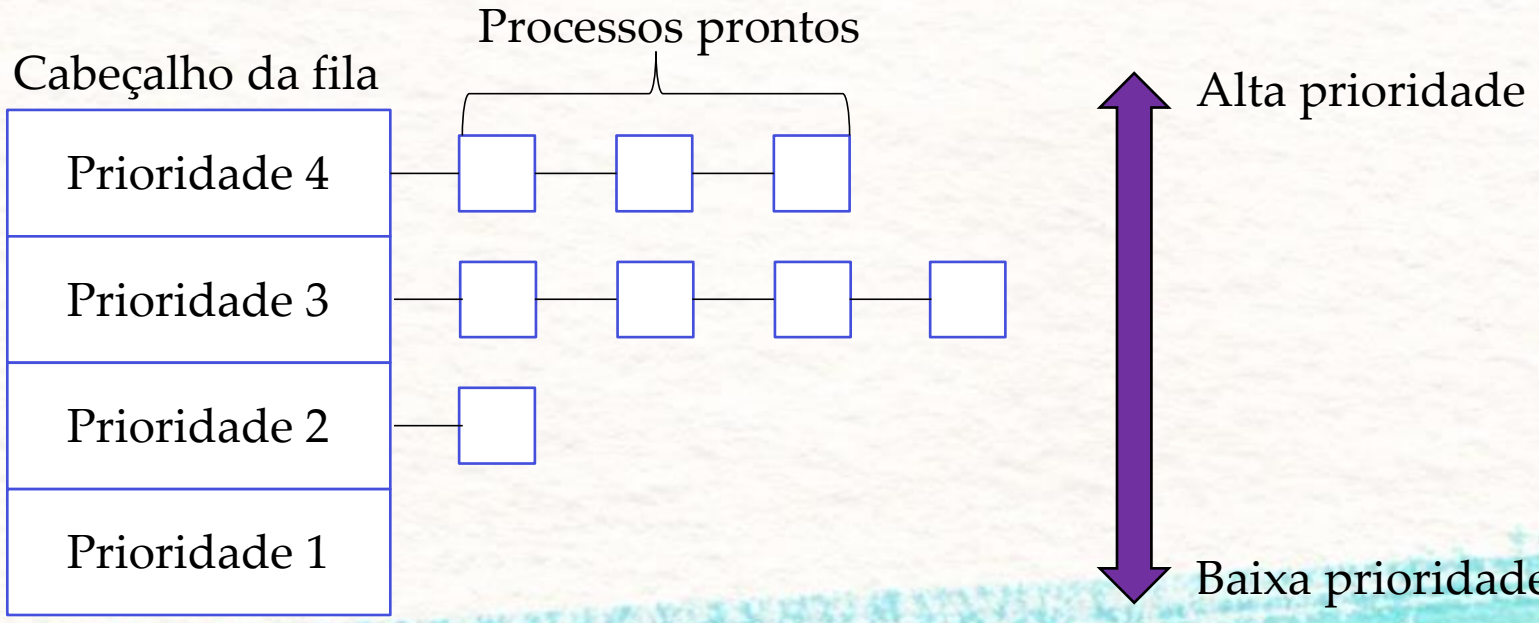
Escalonamento em Sistemas Interativos (Round Robin)

- Cada processo recebe um intervalo de tempo chamado quantum, durante o qual ele pode ser executado. Se o processo ainda estiver em execução no final do quantum, a CPU será preterida e entregue a outro processo. Se o processo foi bloqueado ou finalizado antes que o quantum tenha passado, a comutação da CPU é feita quando o processo é bloqueado.
- O único problema com o Round Robin é determinar o comprimento do quantum:
  - ✓ se muito curto causa sucessuvas trocas de contexto;
  - ✓ se muito longo pode levar a um tempo de resposta inaceitável, por parte dos usuários.



Escalonamento Sistemas Interativos (Escalonamento com Prioridade)

- Cada processo recebe uma prioridade, e segue a sequencia de execução dos de prioridade mais elevada.
- Para evitar que processos de alta prioridade sejam executados indefinidamente, o agendador pode diminuir a prioridade do processo atualmente em execução em cada interrupção do relógio de tempo de execução.
- As prioridades também podem ser atribuídas dinamicamente pelo sistema para atingir determinadas metas do sistema.



Escalonamento Sistemas Interativos (Filas Múltiplas)

Os processos na classe mais prioritária são executados com quantum 1x, os processos da classe seguinte rodam com quantum 2x e assim por diante. Sempre que um processo esgota o quantum de tempo ele é suspenso e colocado em uma fila de prioridade inferior.

Escalonamento Sistemas Interativos (Menor Job Primeiro - SJF)

Este algoritmo foi inicialmente concebidos para sistemas em batch. Em sistema interativos o problema é descobrir qual dos processos em execução é o mais curto. Uma abordagem é fazer estimativas com base no comportamento passado e executar o processo com o menor tempo de execução estimado. Suponha que o tempo estimado por comando para algum processo seja T_0. Agora suponha que sua próxima execução seja medida como T_1. Poderíamos atualizar nossa estimativa tomando uma soma ponderada desses dois números, ou seja, $aT_0 + (1 - a)T_1$. Através da escolha de a, podemos definir que o processo de estimativa esqueça das execuções antigas rapidamente, ou lembe delas por um longo tempo. Com $a = 1/2$, obtemos estimativas sucessivas de:

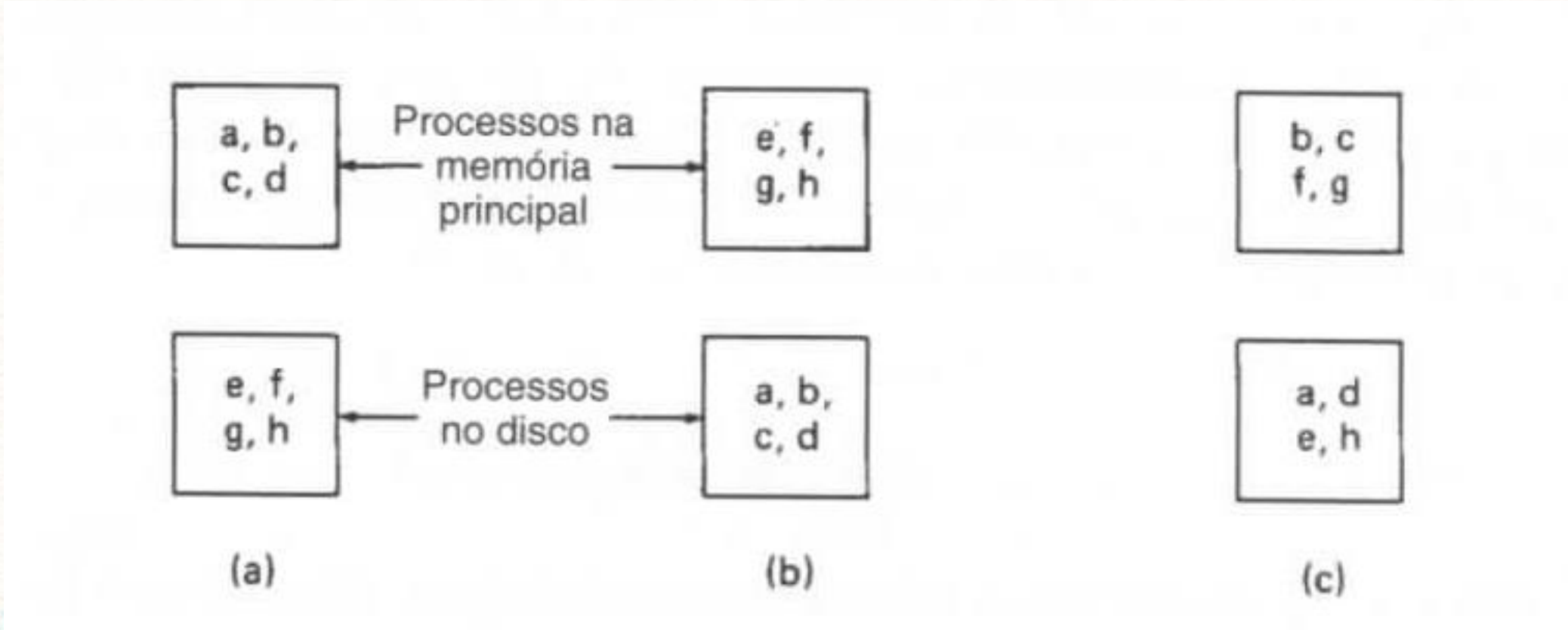
$$T_0,\ T_0/2 + T_1/2,\ T_0/4 + T_1/4 + T_2/2,\ T_0/8 + T_1/8 + T_2/4 + T_3/2$$

Após 3 rodadas o peso de T_0 , em uma nova estimativa, é 1/8 menor.
A técnica de estimar o próximo valor em uma série tomando a média ponderada do valor medido atual, e a estimativa anterior, é chamada de "**aging**".

- Como em sistemas interativos, só é possível obter a situação ótima quando todos os jobs estiverem disponíveis ao mesmo tempo.

Escalonamento Sistemas Interativos (em 2 Níveis)

Até agora temos assumido que todos os processos prontos estão na memória principal. Se não houver memória disponível para todos, alguns processos devem ser mantidos em disco. Um escalonador de dois níveis precisa mover processos entre o disco e a memória principal e ainda escolher o processo que vai rodar dentre aqueles presentes na memória principal. Representamos três instantes de tempo diferentes em (a), (b), e (c).



Escalonamento Sistemas Interativos (em 2 Níveis)

O escalonador limita-se a escolher processos deste subconjunto. Periodicamente, um escalonador de mais alto nível entra em cena, para remover processos que já tenham ficado tempo suficiente na memória principal, e carregar nela processos que estão há muito no disco. Uma vez completada a troca, conforme mostra anterior, o escalonador de baixo nível entra novamente em cena, limitando suas escolhas aos processos armazenados na memória principal. Desta forma, cabe ao escalonador de baixo nível colocar para rodar um dos processos que estiverem na memória principal no momento da ativação do escalonador, enquanto que cabe ao de alto nível movimentar periodicamente processos entre a memória principal e o disco.

- Poderemos lançar mão das técnicas Round Robin, prioridade, ou qualquer outro método de escalonamento, para programar o escalonador de alto nível.

Referências Bibliográficas

- TANENBAUM, Andrew S., BOSS, Herbert. **Sistemas Operacionais Modernos,** Pearson - 4ª ed., 2016.
- SILBERSCHATZ, A., GALVIN, P.B., GAGNE, G. **Fundamentos de Sistemas Operacionais**, Ed. LTC, 8ª ed., 2011
- DEITEL, H.M.; DEITEL, P.J.; CHOFFNES, D.R. **– Sistemas Operacionais**. Prentice Hall, Tradução da 3ª ed., 2005
- MIZRAHI, Victorine Viviane. Treinamento em Linguagem C – Curso Completo módulos 1 e 2, Ed. Person Education.