Aula 2: processos

Prof. DSc. Newton Spolaôr
Apoio: Prof. Sérgio Campos (UFMG), Marcelo Johann (UFRGS),
Roberta Gomes (UFES) e Daniel Abdala (UFU)

Disciplina Sistemas Operacionais
Bacharelado em Ciência da Computação
Universidade Estadual do Oeste do Paraná (UNIOESTE)
Brasil

24/08/2020

unioeste
Universidade Estadual do Oeste do Paraná

Aula anterior em um olhar

- Na aula anterior foram tratados conceitos básicos de Sistemas Operacionais (SO), incluindo definições, histórico e estrutura

Aula anterior em um olhar

- Na aula anterior foram tratados conceitos básicos de Sistemas Operacionais (SO), incluindo definições, histórico e estrutura
- Na aula de hoje, será apresentado o conceito de processos, uma abstração importante oferecida por esses sistemas

O conceito de processo [1]

- Um programa é...
  - Uma seqüência finita de instruções
  - Uma entidade estática (seu estado não se altera com o passar do tempo)
  - Armazenado em disco

O conceito de processo [1]

- Um programa é...
  - Uma seqüência finita de instruções
  - Uma entidade estática (seu estado não se altera com o passar do tempo)
  - Armazenado em disco
- Por sua vez, um processo é...
  - Uma abstração que representa um programa em execução
  - Uma entidade dinâmica (seu estado se altera conforme for executando)
  - Armazenado na memória

O conceito de processo [1]

- É possível encontrar mais de um processo instanciando um programa único (exemplo: Google Chrome)
- O SO coordena a execução de processos concorrentes, *i.e.*, processos que disputam um recurso, como o processador

Processos e concorrência [2]

- O SO também coordena a execução de programas concorrentes, os quais são abstraídos como processos
- Que vantagens podem ser obtidas com a execução concorrente de programas diferentes?

Processos e concorrência [2]

- O SO também coordena a execução de programas concorrentes, os quais são abstraídos como processos
- Que vantagens podem ser obtidas com a execução concorrente de programas diferentes?
  - Permitir que vários usuários usem juntos uma mesma máquina
  - Quando um multiprocessador é usado, é possível completar uma tarefa mais rapidamente

Processos e concorrência [2]

- O SO também coordena a execução de programas concorrentes, os quais são abstraídos como processos
- Que vantagens podem ser obtidas com a execução concorrente de programas diferentes?
  - Permitir que vários usuários usem juntos uma mesma máquina
  - Quando um multiprocessador é usado, é possível completar uma tarefa mais rapidamente
- Outra abstração comumente associada à programação concorrente (aplicada inclusive para desenvolvimento web) consiste nas *threads* – segmentos de código

O processo do ponto de vista do SO [1]

- Imagem de um programa
  - Segmento de código (o que processo fará)
  - Espaço de endereçamento (trecho de memória em que processo fará algo)

O processo do ponto de vista do SO [1]

- Imagem de um programa
  - Segmento de código (o que processo fará)
  - Espaço de endereçamento (trecho de memória em que processo fará algo)
- Conjunto de recursos de hardware alocados pelo SO que compõe o contexto do processo (estado do processador)
  - Registradores (PC, ponteiro de pilha...)
  - Memória
  - Espaço no disco (arquivos de E/S)

Processo e recursos [3]

- Um processo tem a ilusão de que todos os recursos do sistema estão disponíveis para ele
  - Na realidade, em um sistema convencional, o único processador está disponível apenas uma parcela (quantum) de tempo;
  - Além disso, apenas uma parcela da memória está disponível
  - Finalmente, somente os dispositivos requeridos pelo processo estarão disponíveis

Processo e recursos [3]

- Um processo tem a ilusão de que todos os recursos do sistema estão disponíveis para ele
  - Na realidade, em um sistema convencional, o único processador está disponível apenas uma parcela (quantum) de tempo;
  - Além disso, apenas uma parcela da memória está disponível
  - Finalmente, somente os dispositivos requeridos pelo processo estarão disponíveis
- Quem gera e gerencia essas “ilusões” é o SO

Estruturas de dados para processos [1]

- Para manter as informações relativas aos processos, o *kernel*/núcleo deve manter
  - Uma estrutura de dados ("`struct`") relativa a um dado processo, como o descritor de processo, ou *Process Control Block* (PCB)
  - Uma estrutura de dados que gerencia o conjunto de processos, como a tabela de processos – vide exemplo Windows

Estruturas de dados para processos [1]

- Para manter as informações relativas aos processos, o *kernel*/núcleo deve manter
  - Uma estrutura de dados (“`struct`”) relativa a um dado processo, como o descritor de processo, ou *Process Control Block* (PCB)
  - Uma estrutura de dados que gerencia o conjunto de processos, como a tabela de processos – vide exemplo Windows
- As chamadas de sistema (funções do *kernel*) que gerenciam os processos irão interagir com essas estruturas de dados

Estruturas de dados para processos [3]

- PCB
  - Contém toda a informação necessária para
    - Agendar a execução do processo
    - Colocar um processo em espera por um recurso
    - Retomar a execução de um processo
  - É uma das estruturas de dados mais complexas em um SO, pois referencia outras estruturas

Estruturas de dados para processos [3]

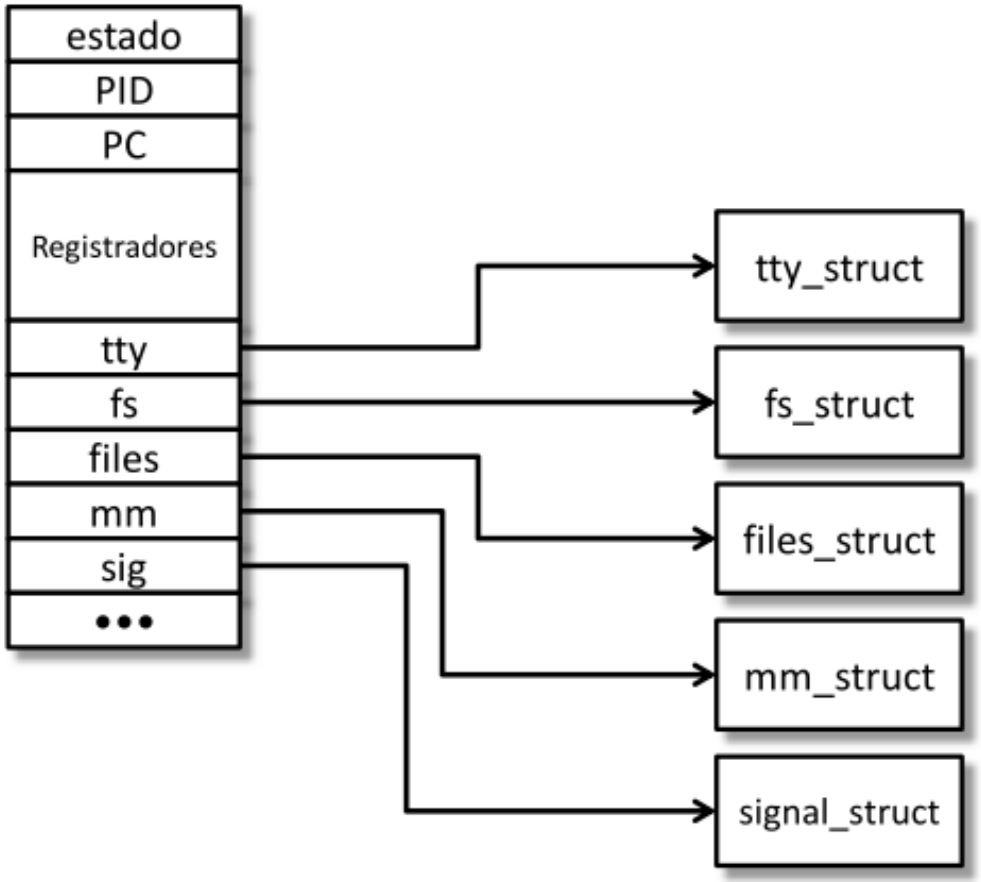


Figura: Exemplo de PCB

Estruturas de dados para processos [4]

- PCB
  - Questão de projeto: a quantidade de registros PCB é fixa ou variável?
  - Considerando quantidade variável, pode ter ponteiro para o próximo PCB em uma lista encadeada
  - PCB pode ser relacionado a filas específicas, como ilustrado a seguir

Estruturas de dados para processos [4]

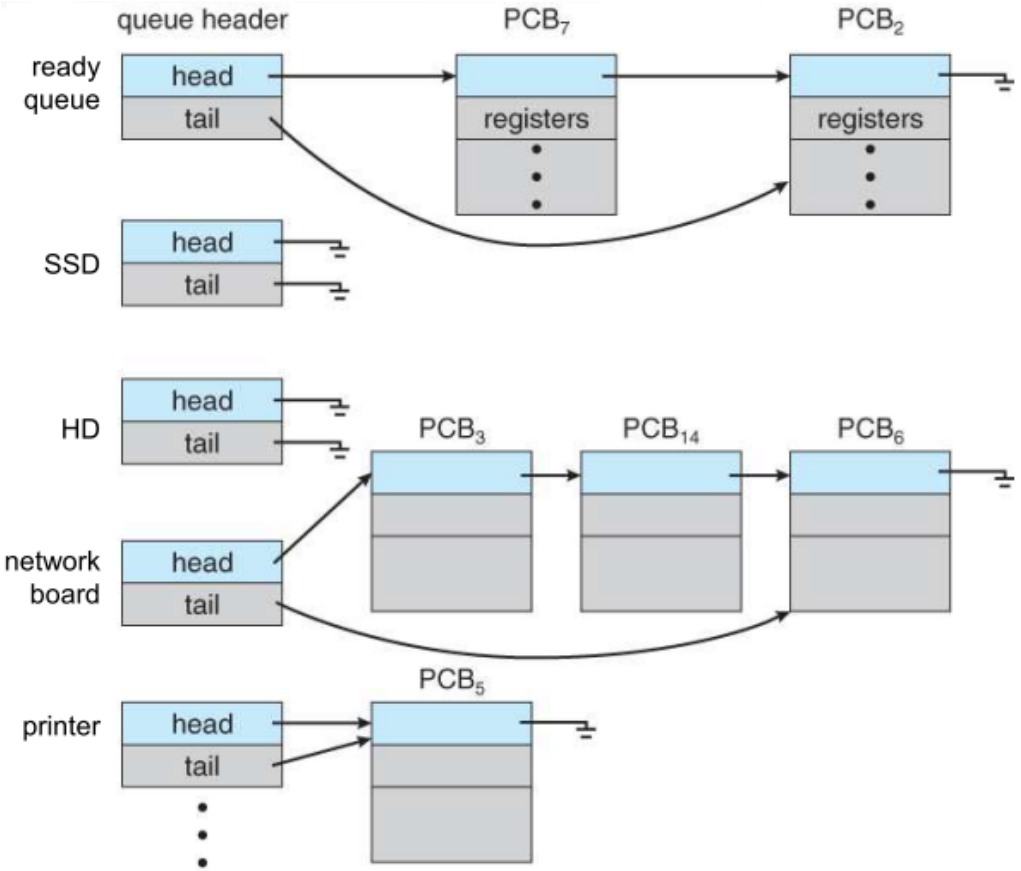


Figura: PCB e as filas do sistema

Tipos de processo [1]

- Enquanto executam, os processos apresentam 2 tipos de comportamento
  - Ou usam muito o processador (CPU), por exemplo, para realizar cálculos ou atuar sobre memória e registradores
  - Ou fazem muitas ações de E/S, como escrita na tela ou recebimento de dados da rede, liberando a CPU

Tipos de processo [1]

- Enquanto executam, os processos apresentam 2 tipos de comportamento
  - Ou usam muito o processador (CPU), por exemplo, para realizar cálculos ou atuar sobre memória e registradores
  - Ou fazem muitas ações de E/S, como escrita na tela ou recebimento de dados da rede, liberando a CPU
- Esses tipos possibilitariam categorizar processos como *CPU-bound* ou *IO-bound*
- Na prática, contudo, é difícil dizer quando um processo é limitado por processador ou E/S

Ciclo de vida de processo [1]

- Processos nascem no momento de sua criação (via chamada de sistema)
- Processos vivem
  - Executam na CPU e liberam a CPU para realizar E/S
  - Executam programas dos usuários e do sistema (como *daemons* no *background*)
- Processos morrem porque terminaram sua execução ou algum processo os encerrou

Ciclo de vida de processo [1]

- Processos evoluem ao longo da vida
- Nesse período, trocam de estado, ora atuando na CPU, ora realizando E/S
- Essa troca é realizada por meio de chamada de sistema, interrupção, ou por causa de um evento

Ciclo de vida de processo [1]

- Ao ser criado, um processo fica pronto para execução na CPU
  - O que acontece se a CPU não está disponível?
  - O que acontece se vários processos estão sendo criados ao mesmo tempo?

Ciclo de vida de processo [1]

- Ao ser criado, um processo fica pronto para execução na CPU
  - O que acontece se a CPU não está disponível?
  - O que acontece se vários processos estão sendo criados ao mesmo tempo?
  - Precisa-se manter uma lista de processos prontos!

Ciclo de vida de processo [1]

- Ao executar, o processo pode querer realizar E/S
  - O que acontece se o recurso de E/S está sendo ocupado?

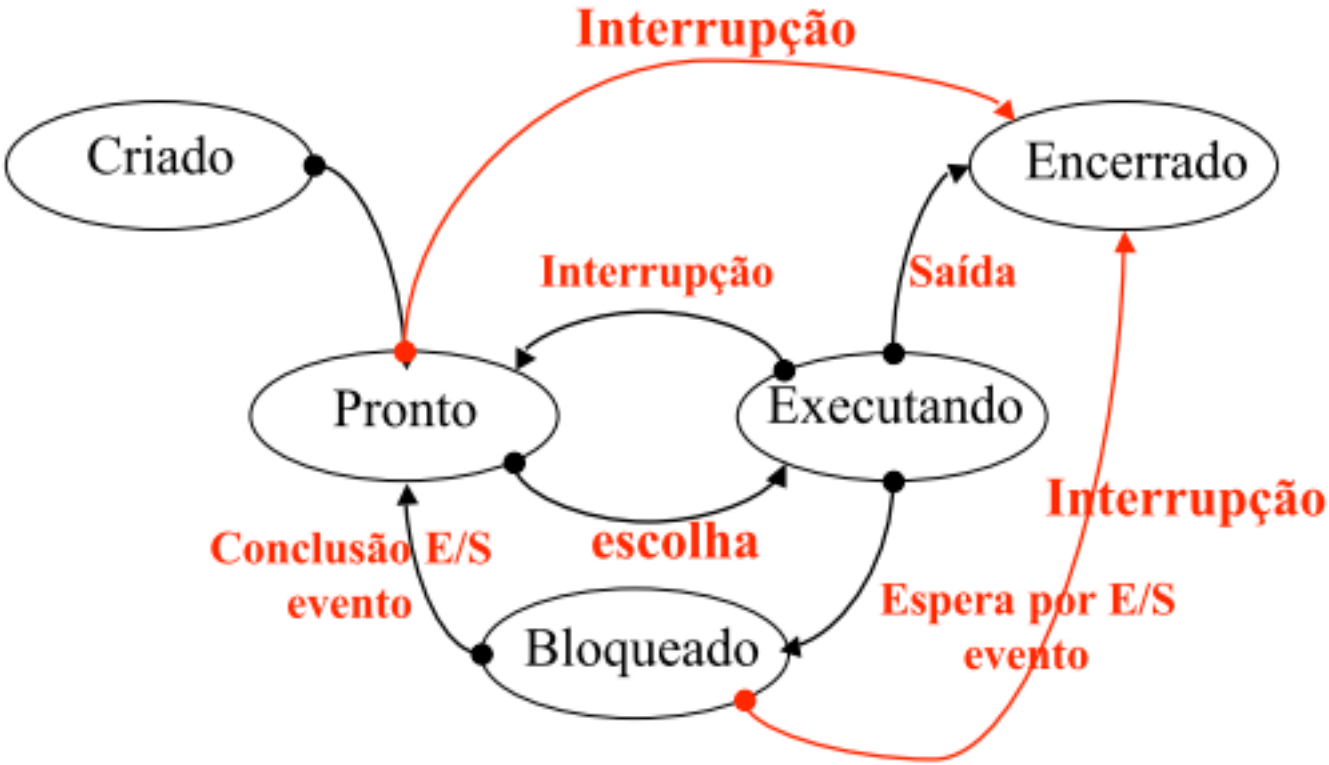
Ciclo de vida de processo [1]

- Ao executar, o processo pode querer realizar E/S
  - O que acontece se o recurso de E/S está sendo ocupado?
  - É preciso de uma fila de processos bloqueados

Estados de um processo [3, 1]

- Em resumo, processos podem estar em nos seguintes estados
  - Criado: processo novo
  - Pronto: apto a ser executado, podendo estar em uma fila de agendamento de processos para execução
  - Executando (rodando): em execução
  - Bloqueado: usualmente, esperando E/S completar
  - Encerrado: usualmente, é o estado final do processo
- Transições entre estados são possíveis

Estados de um processo [1, 5]



O que causa uma transição? [1]

- Pronto $\rightarrow$ executando: algoritmo de escalonamento (agendamento da execução de processo)

O que causa uma transição? [1]

- Pronto $\rightarrow$ executando: algoritmo de escalonamento (agendamento da execução de processo)
- Executando $\rightarrow$ pronto: interrupção do algoritmo de escalonamento ou parada espontânea (chamada *yield*)

O que causa uma transição? [1]

- Pronto $\rightarrow$ executando: algoritmo de escalonamento (agendamento da execução de processo)
- Executando $\rightarrow$ pronto: interrupção do algoritmo de escalonamento ou parada espontânea (chamada *yield*)
- Executando $\rightarrow$ bloqueado: E/S

O que causa uma transição? [1]

- Pronto $\rightarrow$ executando: algoritmo de escalonamento (agendamento da execução de processo)
- Executando $\rightarrow$ pronto: interrupção do algoritmo de escalonamento ou parada espontânea (chamada *yield*)
- Executando $\rightarrow$ bloqueado: E/S
- Bloqueado $\rightarrow$ pronto: interrupção

O que causa uma transição? [1]

- Pronto $\rightarrow$ executando: algoritmo de escalonamento (agendamento da execução de processo)
- Executando $\rightarrow$ pronto: interrupção do algoritmo de escalonamento ou parada espontânea (chamada *yield*)
- Executando $\rightarrow$ bloqueado: E/S
- Bloqueado $\rightarrow$ pronto: interrupção
- Executando $\rightarrow$ encerrado: interrupção ou término normal

O que causa uma transição? [1]

- Pronto $\rightarrow$ executando: algoritmo de escalonamento (agendamento da execução de processo)
- Executando $\rightarrow$ pronto: interrupção do algoritmo de escalonamento ou parada espontânea (chamada *yield*)
- Executando $\rightarrow$ bloqueado: E/S
- Bloqueado $\rightarrow$ pronto: interrupção
- Executando $\rightarrow$ encerrado: interrupção ou término normal
- Bloqueado/pronto $\rightarrow$ encerrado: interrupção

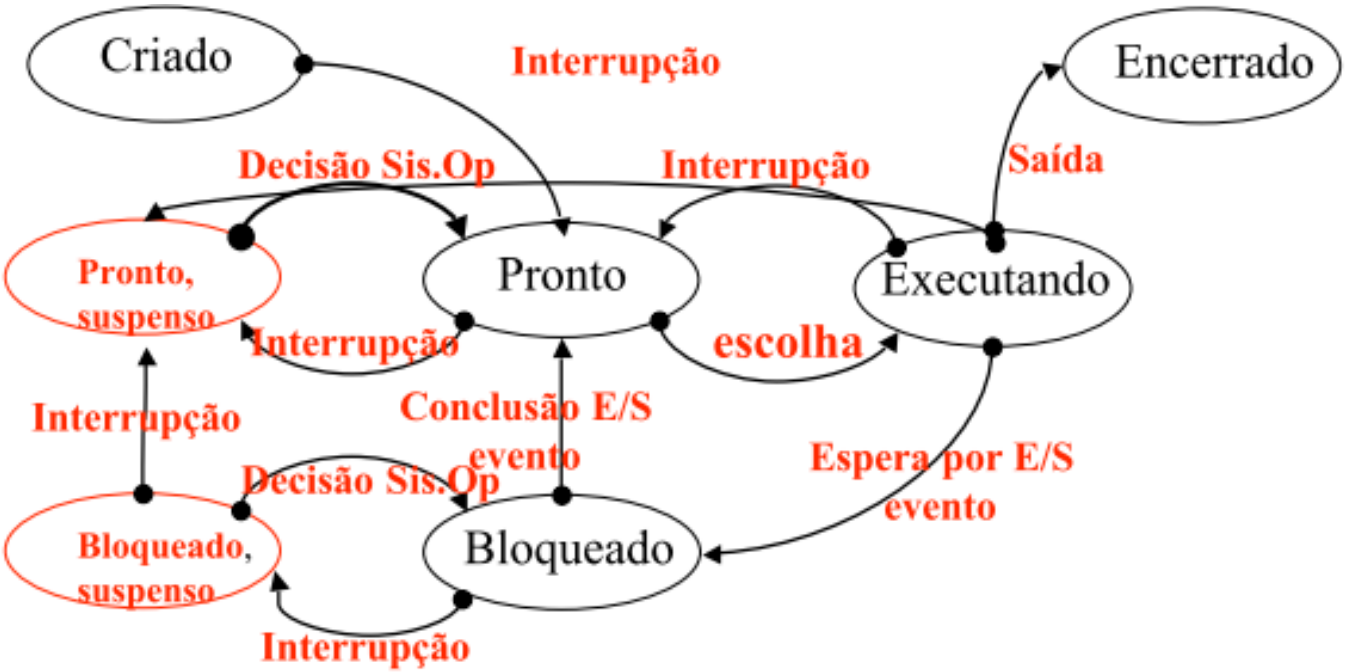
Estado suspenso [1]

- Dois problemas que ocorrem na prática
  - A CPU é muito mais rápida do que a memória
  - A memória é de tamanho finito

Estado suspenso [1]

- Nesse cenário, processos bloqueados que estão na memória podem ser transferidos para o disco (*swap*) até sua E/S ser encerrada
- Processos prontos podem também ser descarregados para o disco
- Assim, surgem mais dois estados referentes a processos armazenados em disco
  1. Suspenso e bloqueado
  2. Suspenso e pronto

Estados de um processo [1, 5]



Relacionamento entre processos [1]

- Caso mais simples: os processos são independentes
- Grupo de processos
  - Compartilhamento de recursos
  - Baseados em hierarquia de processos
    - Um processo pai cria processos filhos
    - Os filhos podem executar o mesmo código, ou trocá-lo
  - Questão de projeto: término de processo encerra ele apenas ou encerra também toda sua descendência?

Sinalização de processos [1]

- Uma das formas de interagir entre processos é através de sinais
  - A recepção é assíncrona
  - Ao receber um sinal, o processo para sua atividade
  - Ele executa um tratamento de sinal adaptado (*signal handler*)
  - Ao se encerrar o tratamento, o processo pode voltar ao estado onde estava antes
- Um sinal é uma versão em nível de software das interrupções de hardware

Suporte de hardware: interrupções [1]

- Erros e eventos são detectados por hardware
  - Exemplos de evento: inserir um pendrive na porta USB, escrever um bloco em disco, receber um pacote pela rede, escrever numa área proíbida...
  - O hardware emite uma interrupção

Suporte de hardware: interrupções [1]

- São tratados pelo SO
  - Identifica a interrupção (número)
  - Verifica sua prioridade
  - Acha no vetor de interrupções qual procedimento é apropriado (*handler*)

Suporte de hardware: modos de execução [1]

- O hardware provê no mínimo dois modos de execução diferentes para um processo
  - Modo privilegiado, protegido ou de sistema
    - Todo o conjunto de operações é disponível
    - É o modo de execução do SO
  - Modo usuário
    - Uso limitado
    - Os processos executados diretamente pelos usuários operam neste modo

Suporte de hardware: modos de execução [1]

- Chaveamento de modos
  - É o fato de passar de um modo para o outro
    - Usuário $\rightarrow$ protegido: por interrupção
    - Protegido $\rightarrow$ usuário: por instrução clássica (exemplos: *yield* e *return*)

Exemplo de uso dos modos de execução [1]

- Para proteger os periféricos, as instruções de E/S são privilegiadas
- Logo, um processo usuário não pode acessá-las
- Como exemplo de instruções, estão as que realizam escrita em disco e leitura de um CD
- O usuário deve então passar pelo SO através de uma chamada de sistema, a qual gera uma interrupção (*trap*)

Chamada de sistema com interrupção [1]

- A chamada de sistema oferece um serviço ao processo usuário com “segurança”
  - Ela gera uma interrupção a partir de informações como identificação do processo e prioridade
  - Ela implica em uma troca de contexto – troca do processo em execução por outro processo
    - O processo chamador deve deixar o lugar para o código do núcleo!
    - Assim, a troca de modo de execução implica em uma troca de contexto

Chamada de sistema com interrupção [1]

- Conforme for a prioridade e o tipo de escalonador (mecanismo que gerencia a troca de processos), a troca de contexto pode ser imediata ou atrasada
- O que acontece se houver uma interrupção durante o tratamento de uma interrupção?

Chamada de sistema com interrupção [1]

- Conforme for a prioridade e o tipo de escalonador (mecanismo que gerencia a troca de processos), a troca de contexto pode ser imediata ou atrasada
- O que acontece se houver uma interrupção durante o tratamento de uma interrupção?
  - Comparam-se as prioridades
  - Possibilidade de desabilitar as interrupções em casos críticos

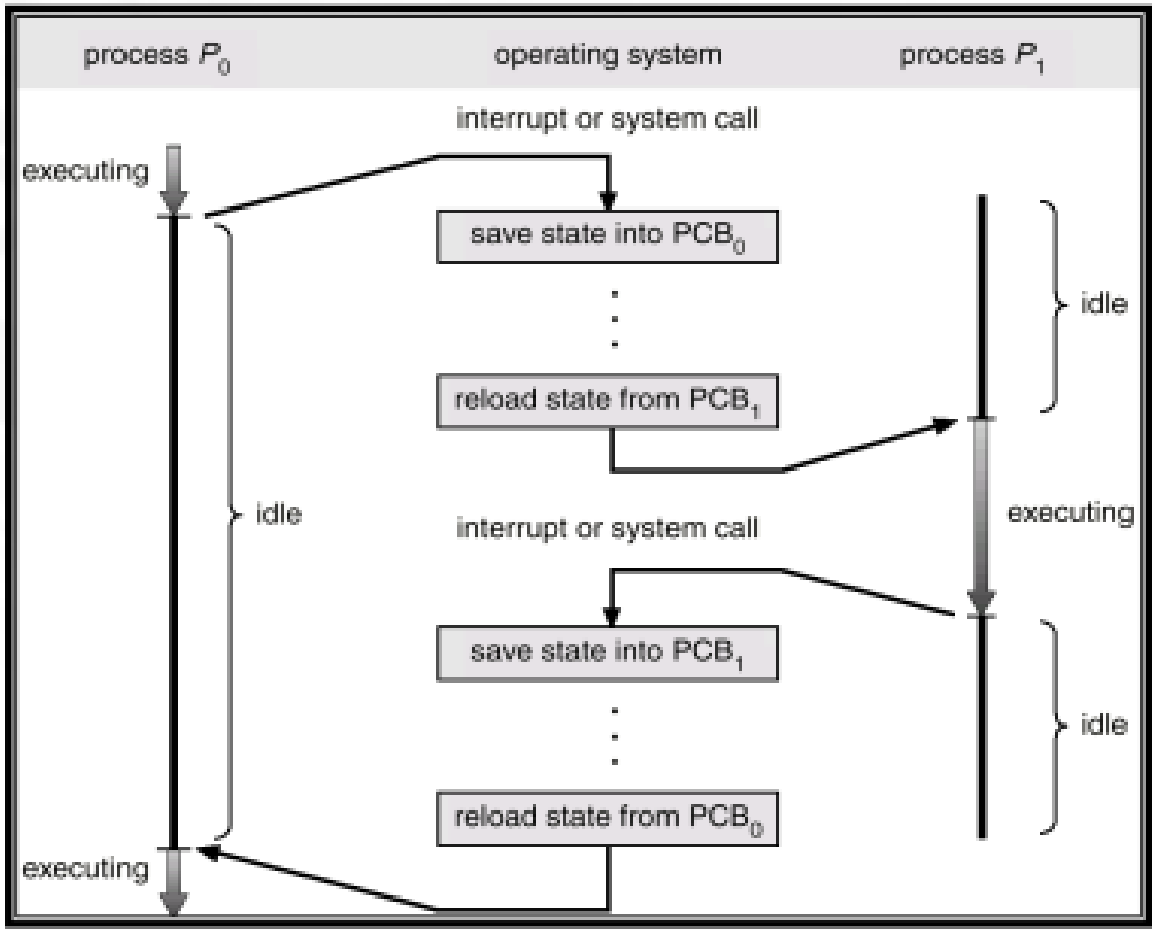
Troca de contexto [4]

- Ações na troca de contexto
  - Salvar o contexto do processador, incluindo o PC e outros registradores
  - Alterar o PCB do processo que está no estado “executando”
  - Mover o PCB para a fila apropriada

Troca de contexto [4]

- Ações na troca de contexto
  - Selecionar outro processo para execução
  - Alterar o PCB do processo selecionado
  - Alterar as tabelas de gerência de memória
  - Restaurar o contexto do processo selecionado

Troca de contexto [4]



Troca de contexto

- Em algum momento, será realizada uma troca de contexto entre processos
- Assim, diferentes processos podem executar no processador
- O agendamento de processos para execução é denominado escalonamento

Processos e disco [1]

- Os processos devem interagir com o disco para armazenar e recuperar dados não voláteis
- O disco físico é abstraído pelo Sistemas de Arquivos, de acordo com uma hierarquia
  - Diretórios
  - Arquivos

Processos e disco [1]

- Os diretórios estão freqüentemente organizados de acordo com uma hierarquia em árvore
  - Raiz (’/’)
  - Diretório de trabalho de um processo (’.’)
  - Caminho relativo / absoluto
- Devem existir chamadas de sistema para acessar o Sistema de Arquivos!

Exemplos de chamadas de sistema [1]

- Minix 2 provê 53 chamadas de sistema no total
  - Gerenciamento de processos: *fork()*, *exit()*, *getpid()* e *execve()*
  - Sinais: *kill()*
  - Gerenciamento de arquivos: *open()*, *close()*, *read()*, *mkdir()* e *mount()*
  - Gerenciamento de tempo: *time()*

A especificação POSIX [3]

- A especificação *Portable Opeating System Interface* (POSIX) é uma norma da IEEE para padronizar principalmente nome e função de chamadas de sistema
- A maioria dos sistemas atuais baseados em UNIX, incluindo MINIX, são compatíveis com POSIX
- O objetivo é facilitar a portabilidade de aplicações entre sistemas baseados no UNIX

Criação de processos com POSIX [3]

- Em geral, todo processo é iniciado por outro processo
- Processos só podem ser iniciados pelo SO, consequentemente, para criar um novo processo uma chamada de sistema é requerida
- As chamadas previstas no POSIX relacionadas a criação de processos são *system*, *exec* (e variantes) e *fork*

Criação de processos com POSIX [3]

- Quando um processo é criado, apenas o seu descritor de processo (PCB) é criado
- O processo criado é uma cópia exata do processo que o criou, exceto pelo identificador de processo (PID)
- Seu segmento de dados, texto, pilha, PC, demais registradores, descritores de arquivo e espaço de endereçamento são exatamente o mesmo
- Tal fato torna a criação de um novo processo eficiente e rápida

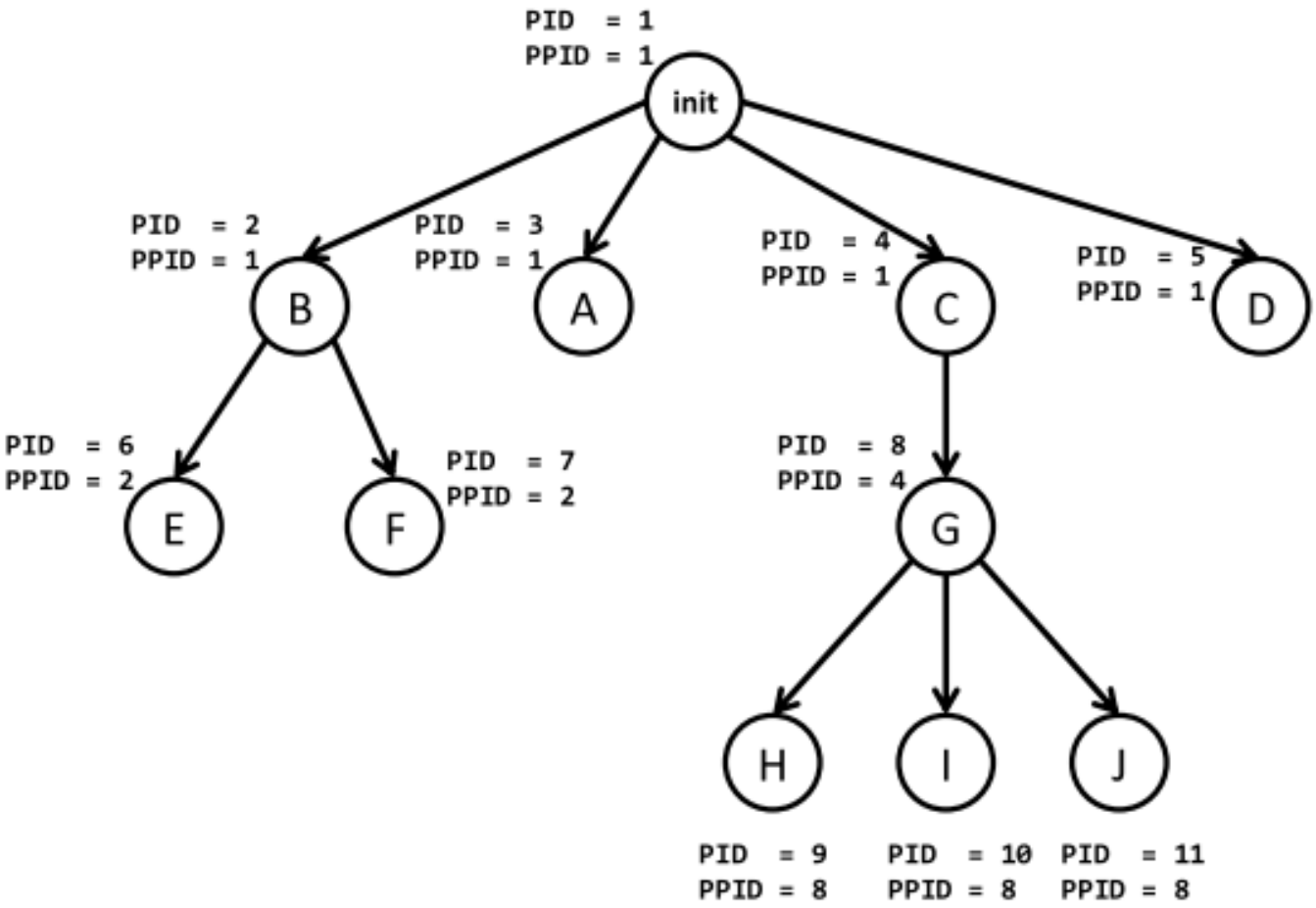
Criação de processos com POSIX [3]

- Quando o novo processo executa, itens no contexto dele podem começar a variar
- Por exemplo, novos arquivos podem ser abertos ou o endereço no PC variará em relação ao processo pai

Hierarquia de processos [3]

- O processo A que cria um novo processo B é chamado de pai de B
- O processo B é chamado de filho de A
- No Linux apenas um processo não possui pai (*init*), pois ele é o processo inicial do SO

Hierarquia de processos [3]



Chamadas de sistema Linux para processos [3, 1]

- Criação de processo com *fork()*
  - Provavelmente a chamada do sistema mais utilizada para criação de novos processos
  - Cria um novo processo igual ao pai, *i.e.*, um clone
  - Bifurca a execução do código, prosseguindo a partir do mesmo ponto em ambos os processos (pai e filho)
  - No processo pai, *fork()* retorna o pid do filho

Chamadas de sistema Linux para processos [3, 1]

- Mudar o segmento de código com *exec()*
  - Executa o binário apontado em argumento
  - Em geral, chamado logo após o *fork()* (*"fork-exec"*), de modo que o processo filho recém criado execute um binário específico

Chamadas de sistema Linux para processos [1]

- Recuperar o identificador com *getpid()*

Chamadas de sistema Linux para processos [1]

- Recuperar o identificador com *getpid()*
- Terminar o processo com *kill()*
  - Manda um sinal (e.g. TERM) para o processo cujo pid é dado em argumento
  - Comando comum em terminais de comando

Chamadas de sistema Linux para processos [1]

- Recuperar o identificador com *getpid()*
- Terminar o processo com *kill()*
  - Manda um sinal (e.g. TERM) para o processo cujo pid é dado em argumento
  - Comando comum em terminais de comando
- Terminar o processo com *exit()*

Chamadas de sistema Windows para processos [3]

- Processos no Windows não seguem as mesmas convenções previstas no padrão POSIX
- Exemplos

UNIX	Win32	Descrição
fork	CreateProcess	Cria um novo processo
waitpid	WaitForSingleObject	Espera que um processo termine
execve	-	Substitui a imagem de um processo
exit	ExitProcess	Conclui a execução

Processos e *threads* [1, 2]

- Como mencionado, um processo é um programa em execução
  - Segmento de código (ou fluxo de controle)
  - Espaço de endereçamento (“dados do programa”)

Processos e *threads* [1, 6, 2]

- Por sua vez, uma *thread* consiste somente em um segmento de código
  - Espaço de endereçamento custa caro

Processos e *threads* [1, 6, 2]

- Por sua vez, uma *thread* consiste somente em um segmento de código
  - Espaço de endereçamento custa caro
  - A troca de contexto é mais complexa do que a troca de *threads*, pois envolve o espaço de endereçamento

Processos e *threads* [1, 6, 2]

- Por sua vez, uma *thread* consiste somente em um segmento de código
  - Espaço de endereçamento custa caro
  - A troca de contexto é mais complexa do que a troca de *threads*, pois envolve o espaço de endereçamento
  - Muitas vezes, processos compartilham dados; uma alternativa mais eficiente para troca de dados entre segmentos de código envolve usar *threads* no mesmo espaço de endereçamento, *i.e.*, atribuir múltiplas *threads* em um processo

Processos e *threads* [1, 6, 2]

- Por sua vez, uma *thread* consiste somente em um segmento de código
  - Espaço de endereçamento custa caro
  - A troca de contexto é mais complexa do que a troca de *threads*, pois envolve o espaço de endereçamento
  - Muitas vezes, processos compartilham dados; uma alternativa mais eficiente para troca de dados entre segmentos de código envolve usar *threads* no mesmo espaço de endereçamento, *i.e.*, atribuir múltiplas *threads* em um processo
  - A criação (e destruição) de uma *thread* é mais fácil e rápida do que de um processo

Processos e *threads* [1, 6, 2]

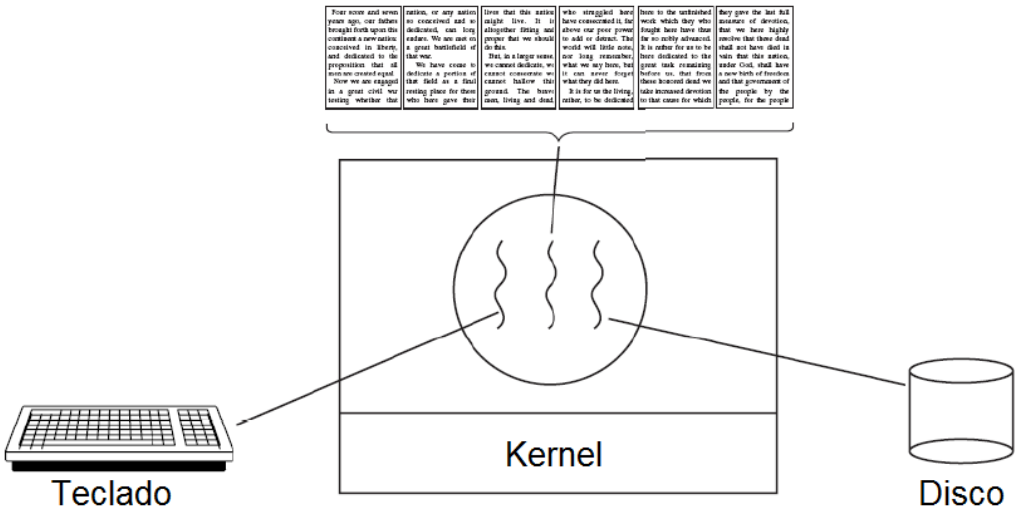
- Por sua vez, uma *thread* consiste somente em um segmento de código
  - Espaço de endereçamento custa caro
  - A troca de contexto é mais complexa do que a troca de *threads*, pois envolve o espaço de endereçamento
  - Muitas vezes, processos compartilham dados; uma alternativa mais eficiente para troca de dados entre segmentos de código envolve usar *threads* no mesmo espaço de endereçamento, *i.e.*, atribuir múltiplas *threads* em um processo
  - A criação (e destruição) de uma *thread* é mais fácil e rápida do que de um processo
  - Algoritmos de escalonamento de processos podem ser estendidos para escalonamento de *threads*

A noção de *thread* [7, 1, 6, 2]

- Em vários SO, um processo pode ter múltiplas *threads*, cada um resolvendo um determinado problema
  - Para tanto, há compartilhamento de recursos do PCB do processo entre as múltiplas *threads*
  - Nesse cenário, o PCB deve incluir uma lista de *threads*!

A noção de *thread* [7, 1, 6, 2]

- Editor de texto com múltiplas *threads*
  - Uma *thread* interage com o usuário, recebendo comandos do teclado e exibindo o texto
  - Uma *thread* atua em *background*, concretizando os comandos solicitados pelo usuário e alterando o texto
  - Uma *thread* atua em *background*, interagindo com o disco para realizar o salvamento automático do texto (*auto-save*)



A noção de *thread* [7, 1, 6, 2]

- O que ocorreria se o editor fosse abstraído como um processo de única *thread*?
- Há alguma desvantagem em usar múltiplas *threads* ao invés de múltiplos processos para resolver o problema de edição de texto?

Implementação de *threads* [1]

- Cada *thread* é associada com um descritor, o qual inclui
  - Cópia privada de registradores, como PC e *Stack Pointer* (SP)

Implementação de *threads* [1]

- Cada *thread* é associada com um descritor, o qual inclui
  - Cópia privada de registradores, como PC e *Stack Pointer* (SP)
  - Uma pilha, a partir da qual se acessam chamadas a subrotinas que não completaram ainda e suas variáveis locais, bem como o endereço de retorno acessado após completar cada chamada na pilha

Implementação de *threads* [1]

- Cada *thread* é associada com um descritor, o qual inclui
  - Cópia privada de registradores, como PC e *Stack Pointer* (SP)
  - Uma pilha, a partir da qual se acessam chamadas a subrotinas que não completaram ainda e suas variáveis locais, bem como o endereço de retorno acessado após completar cada chamada na pilha
  - Identificação – *thread* ID

Implementação de *threads* [1]

- Cada *thread* é associada com um descritor, o qual inclui
  - Cópia privada de registradores, como PC e *Stack Pointer* (SP)
  - Uma pilha, a partir da qual se acessam chamadas a subrotinas que não completaram ainda e suas variáveis locais, bem como o endereço de retorno acessado após completar cada chamada na pilha
  - Identificação – *thread* ID
  - Ponteiros para outras *threads*

Implementação de *threads* [1]

- Cada *thread* é associada com um descritor, o qual inclui
  - Cópia privada de registradores, como PC e *Stack Pointer* (SP)
  - Uma pilha, a partir da qual se acessam chamadas a subrotinas que não completaram ainda e suas variáveis locais, bem como o endereço de retorno acessado após completar cada chamada na pilha
  - Identificação – *thread* ID
  - Ponteiros para outras *threads*
  - Ponteiro para o PCB em que se encontra, permitindo acessar o espaço de endereçamento do processo pai

Implementação de *threads* [1]

- Cada *thread* é associada com um descritor, o qual inclui
  - Cópia privada de registradores, como PC e *Stack Pointer* (SP)
  - Uma pilha, a partir da qual se acessam chamadas a subrotinas que não completaram ainda e suas variáveis locais, bem como o endereço de retorno acessado após completar cada chamada na pilha
  - Identificação – *thread* ID
  - Ponteiros para outras *threads*
  - Ponteiro para o PCB em que se encontra, permitindo acessar o espaço de endereçamento do processo pai
  - Informação sobre escalonamento, incluindo estado da *thread*

Tipos de *threads* [1, 2]

- *Thread* “usuário”
  - Uma biblioteca dá suporte à criação e escalonamento, entre outras tarefas
  - *Thread* executa em modo usuário, sem interferência do núcleo
  - Uma vantagem é que são muito rápidas de gerenciar pois não necessitam do núcleo (chamada de sistema...)
  - Exemplo: bibliotecas Pthreads (POSIX) e *threads* de SOLARIS

Tipos de *threads* [1, 2]

- *Thread* “núcleo”
  - O núcleo oferece suporte às *threads*
  - Uma vantagem é que o SO pode escalonar mais eficientemente as *threads*, inclusive em máquinas multi-processadas
  - Exemplo: Windows, Solaris e Linux

Tipos de *threads* [1, 2]

- Os dois níveis podem ser conciliados por meio de uma das seguintes soluções
  - N *threads* usuário por *thread* de núcleo (N:1)
  - Uma *thread* usuário por *thread* de núcleo (1:1)
  - Meio termo entre os dois (N:M)

O modelo N:1 [1, 2]

- N *threads* usuário estão fisicamente implementadas em uma *thread* de núcleo
- Todo o gerenciamento das *threads* se faz em nível de usuário, atingindo grande velocidade

O modelo N:1 [1, 2]

- N *threads* usuário estão fisicamente implementadas em uma *thread* de núcleo
- Todo o gerenciamento das *threads* se faz em nível de usuário, atingindo grande velocidade
- O escalonamento das *threads* é uma grande limitação
  - O núcleo escalona as *threads* na CPU
  - Ele só enxerga a *thread* de núcleo sem distinguir as *threads* de usuário nela implementadas
  - Se uma *thread* de usuário for bloqueada, toda a *thread* de núcleo estará bloqueada!
- Exemplo: POSIX *Pthreads*

Outro exemplo N:1 – Java *threads* [1, 2]

- Java disponibiliza uma interface de programação com *threads*
- Um programa Java executa dentro de uma máquina virtual
  - A MVJ é executada, em geral, como um processo único
  - A MVJ usa várias *threads* para seu gerenciamento, resolvendo tarefas como coleta de lixo

Outro exemplo N:1 – Java *threads* [1, 2]

- Java disponibiliza uma interface de programação com *threads*
- Um programa Java executa dentro de uma máquina virtual
  - A MVJ é executada, em geral, como um processo único
  - A MVJ usa várias *threads* para seu gerenciamento, resolvendo tarefas como coleta de lixo
  - O mapeamento das *threads* Java da MVJ para as *threads* do SO depende da implementação da MVJ!
    - Em alguns casos, o mapeamento pode ser de *threads* Java para *threads* de usuário
    - No Windows, mapeamento 1:1 de *threads* Java para *threads* de núcleo

O modelo 1:1 [1, 2]

- Cada *thread* de usuário é mapeada em uma *thread* de núcleo
- Assim, o SO escalona as *threads* na CPU, bloqueando somente uma *thread* de usuário por vez
- Adequado para arquiteturas multi-processadas

O modelo 1:1 [1, 2]

- Cada *thread* de usuário é mapeada em uma *thread* de núcleo
- Assim, o SO escalona as *threads* na CPU, bloqueando somente uma *thread* de usuário por vez
- Adequado para arquiteturas multi-processadas
- Uma desvantagem é o maior custo de criação/manutenção das *threads*, pois uma *thread* de núcleo é mais lenta que uma *thread* de usuário
- Sistemas que optam por essa alternativa limitam o número de threads que podem ser criadas

O modelo N:M [1, 2]

- Várias *threads* de usuário são implementadas em múltiplas *threads* de núcleo
- O número exato pode variar conforme for a arquitetura e ou a aplicação
- Junta as vantagens dos dois outros modelos
- É uma solução herdada diretamente do sistema Solaris

Processo e *thread* no Windows 2000+ [1, 2]

- O processo é descrito por um PCB, com parte posicionado no núcleo e parte no espaço de usuário
- A parte do PCB que atua no espaço de usuário recebe o nome de *Process Environment Block* (PEB)
- Um processo inclui no mínimo uma *thread*
  - Essa *thread* é de núcleo

Processo e *thread* no Windows 2000+ [1, 2]

- O processo é descrito por um PCB, com parte posicionado no núcleo e parte no espaço de usuário
- A parte do PCB que atua no espaço de usuário recebe o nome de *Process Environment Block* (PEB)
- Um processo inclui no mínimo uma *thread*
  - Essa *thread* é de núcleo
  - Pode haver mais de uma para máquinas multi-processadas

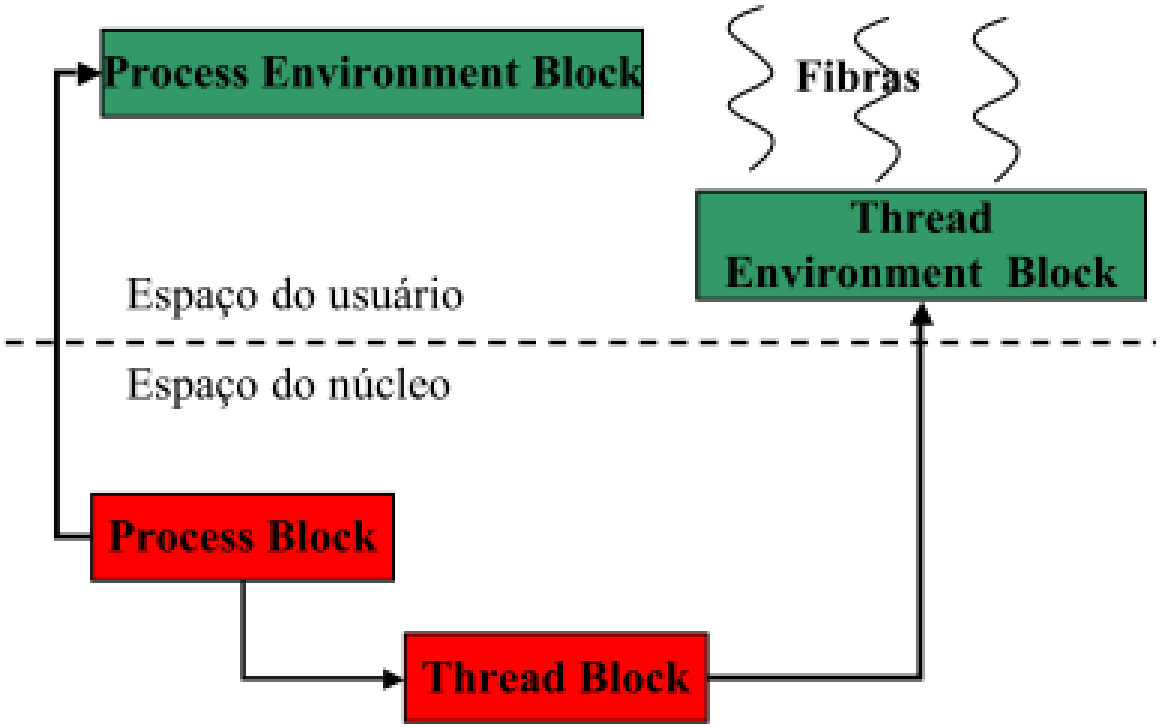
Processo e *thread* no Windows 2000+ [1, 2]

- O processo é descrito por um PCB, com parte posicionado no núcleo e parte no espaço de usuário
- A parte do PCB que atua no espaço de usuário recebe o nome de *Process Environment Block* (PEB)
- Um processo inclui no mínimo uma *thread*
  - Essa *thread* é de núcleo
  - Pode haver mais de uma para máquinas multi-processadas
  - Uma *thread* possui uma parte no espaço de núcleo, uma outra no espaço do usuário (2 pilhas!)

Processo e *thread* no Windows 2000+ [1, 2]

- O processo é descrito por um PCB, com parte posicionado no núcleo e parte no espaço de usuário
- A parte do PCB que atua no espaço de usuário recebe o nome de *Process Environment Block* (PEB)
- Um processo inclui no mínimo uma *thread*
  - Essa *thread* é de núcleo
  - Pode haver mais de uma para máquinas multi-processadas
  - Uma *thread* possui uma parte no espaço de núcleo, uma outra no espaço do usuário (2 pilhas!)
  - As *threads* (de núcleo) podem hospedar várias “fibras” (*threads* de usuário, também denominadas como “*lightweight thread*”)

Processo e *thread* no Windows 2000+ [1, 2]



Bibliotecas de *threads* em UNIX [1]

- Exemplos de bibliotecas de *threads* para UNIX
  - POSIX threads (Pthreads)
    - Biblioteca definida pelo padrão POSIX para gerenciamento de *threads* a nível de usuário e de núcleo
    - Possibilita a criação, escalonamento, sincronização e encerramento de *threads*
  - Florida State University Pthreads: compatível com POSIX e suporte a gerenciamento de *threads* de usuário
  - LinuxThreads: similar a anterior, mas também suporta gerenciamento de *threads* de núcleo

Motivação para escalonamento de processos [3]

- O escalonamento (agendamento de processos para execução) é uma necessidade em um cenário com multiprogramação, pois dois ou mais processos podem estar simultaneamente no estado “pronto”
- Um sistema com múltiplos usuários (humanos ou não) também demanda um agendamento eficiente dos processos para execução

Motivação para escalonamento de processos [3]

- O escalonamento (agendamento de processos para execução) é uma necessidade em um cenário com multiprogramação, pois dois ou mais processos podem estar simultaneamente no estado “pronto”
- Um sistema com múltiplos usuários (humanos ou não) também demanda um agendamento eficiente dos processos para execução
- O algoritmo de escalonamento é um módulo importante em um SO
- Nessa aula, alguns dos algoritmos mais populares serão tratados

Escalonamento em sistemas computacionais distintos [3]

- Um bom escalonador pode impactar consideravelmente o desempenho de um sistema computacional
- Computadores pessoais
  - Apenas um programa “em primeiro plano”
  - Sistemas interativos
  - Escalonador deve levar tais fatos em consideração

Escalonamento em sistemas computacionais distintos [3]

- Um bom escalonador pode impactar consideravelmente o desempenho de um sistema computacional
- Computadores pessoais
  - Apenas um programa “em primeiro plano”
  - Sistemas interativos
  - Escalonador deve levar tais fatos em consideração
- Servidores: escalonador fundamental para maximizar desempenho de um ou mais programas com alta prioridade

Comportamento de um processo [3]

- Uma tendência geral que se observa é que, quanto mais rápida for a CPU, maior será a tendência dos processos serem *IO-bound*
- Algumas questões de projeto
  1. Que processo executar após a criação de um novo processo: pai ou filho?

Comportamento de um processo [3]

- Uma tendência geral que se observa é que, quanto mais rápida for a CPU, maior será a tendência dos processos serem *IO-bound*
- Algumas questões de projeto
  1. Que processo executar após a criação de um novo processo: pai ou filho?
  2. Ao término de um processo, o escalonador deve escolher outro processo para ser executado; o que ocorre se não há processos prontos para serem executados?

Tipos de algoritmos de escalonamento [3]

- Podem ser organizados como
  - Não-preemptivos – mantém processo executando até que uma determinada condição ocorra
  - Preemptivos – interrompe e troca processo, mesmo se ele não terminou de executar ainda

Tipos de algoritmos de escalonamento [3]

- Podem ser organizados como
  - Não-preemptivos – mantém processo executando até que uma determinada condição ocorra
  - Preemptivos – interrompe e troca processo, mesmo se ele não terminou de executar ainda
- Podem ser categorizados como
  - Algoritmos para sistemas em lote – sistemas que executam uma sequência de tarefas com pouca interação com usuários durante a execução
  - Algoritmos para sistemas interativos
  - Algoritmos para sistemas de tempo real
  - Algoritmos híbridos

Escalonamento não-preemptivo [3]

- Escolhe um processo e o deixa executar até que uma das seguintes três condições ocorram
  - Demanda por E/S, levando ao bloqueio de processo

Escalonamento não-preemptivo [3]

- Escolhe um processo e o deixa executar até que uma das seguintes três condições ocorram
  - Demanda por E/S, levando ao bloqueio de processo
  - Espera por outro processo

Escalonamento não-preemptivo [3]

- Escolhe um processo e o deixa executar até que uma das seguintes três condições ocorram
  - Demanda por E/S, levando ao bloqueio de processo
  - Espera por outro processo
  - Liberação voluntária da CPU para o escalonador alocar outro processo – “o comando `nice` do Linux permite ao usuário voluntariamente reduzir a prioridade de seu processo, sendo simpático com outros usuários. Ninguém nunca o usa.” [6]

Escalonamento não-preemptivo [3]

- Implementação relativamente simples
- Uma boa opção para sistemas de grande porte que devem implementar simultaneamente duas modalidades
  1. Sistema em lotes: pouca interação com usuário dispensa escalonamento complexo
  2. Sistema multitarefa/multiusuário: interromper uma tarefa/usuário para atender outra/outro de prioridade similar pode deteriorar experiência, por exemplo

Escalonamento preemptivo [3]

- Escolhe um processo e o deixa em execução por um tempo máximo especificado (quantum)
- Ao final do quantum, se o processo ainda estiver em execução, ele será substituído por um processo “pronto” que está esperando para executar
- Requer que uma interrupção de relógio (preempção) seja gerada em intervalos regulares

Escalonamento preemptivo [3]

- Definir o quantum ótimo para um sistema é uma tarefa complexa
  - Quantum curto: muita troca de contexto, uma tarefa pesada
  - Quantum longo: diminui a capacidade de resposta (interatividade) do sistema
- O escalonamento, preemptivo ou não, também é aplicável a *threads*; nesta aula, focaremos em processos

Objetivos dos algoritmos de escalonamento [3]

- Todos os sistemas
  - Justiça: cada processo deveria receber uma porção justa da CPU
  - Política: verificar se o esquema de rodízio dos processos é cumprido
  - Equilíbrio: manter ocupada, na medida do possível, todas as partes do sistema (CPU, E/S...)

Objetivos dos algoritmos de escalonamento [3]

- Sistemas em lote
  - Vazão (*throughput*): medida usada também em áreas como arquitetura de computadores que reflete o número de tarefas realizadas por unidade de tempo; escalonadores buscam maximizar a vazão
  - Tempo de retorno: minimizar o tempo entre submissão e término de processos
  - Uso da CPU: manter a CPU ocupada o tempo todo

Objetivos dos algoritmos de escalonamento [3]

- Sistemas interativos
  - Tempo de resposta: responder rapidamente às requisições
  - Proporcionalidade: satisfazer as expectativas dos usuários

Objetivos dos algoritmos de escalonamento [3]

- Sistemas interativos
  - Tempo de resposta: responder rapidamente às requisições
  - Proporcionalidade: satisfazer as expectativas dos usuários
- Sistemas de tempo real
  - Cumprimento de prazos: evitar a perda de dados que devem ser processados em tempo crítico
  - Previsibilidade: evitar a degradação da qualidade em sistemas multimídia

Algoritmos de escalonamento [8, 9, 3]

- A seguir, conceitos de alguns algoritmos de escalonamento são considerados
- Animações e exemplos de uso de algoritmos de escalonamento estão disponíveis em
  - Tutorialspoint: https://goo.gl/c97JK5
  - Youtube (diferentes alternativas)

First Come First Served (FCFS) [3]

- Algoritmo não-preemptivo útil para sistemas em lote
- Como nome diz, o primeiro (processo) a chegar é o primeiro a ser servido

First Come First Served (FCFS) [3]

- Algoritmo não-preemptivo útil para sistemas em lote
- Como nome diz, o primeiro (processo) a chegar é o primeiro a ser servido
- Em outras palavras, CPU é atribuída aos processos na ordem em que eles a requisitam
- Provavelmente foi o primeiro e mais simples algoritmo de escalonamento

First Come First Served (FCFS) [3]

- Processo monopoliza a CPU pelo tempo que ele requerer ou até que ele requisite uma operação de E/S
- Basicamente existe uma fila única de processos prontos

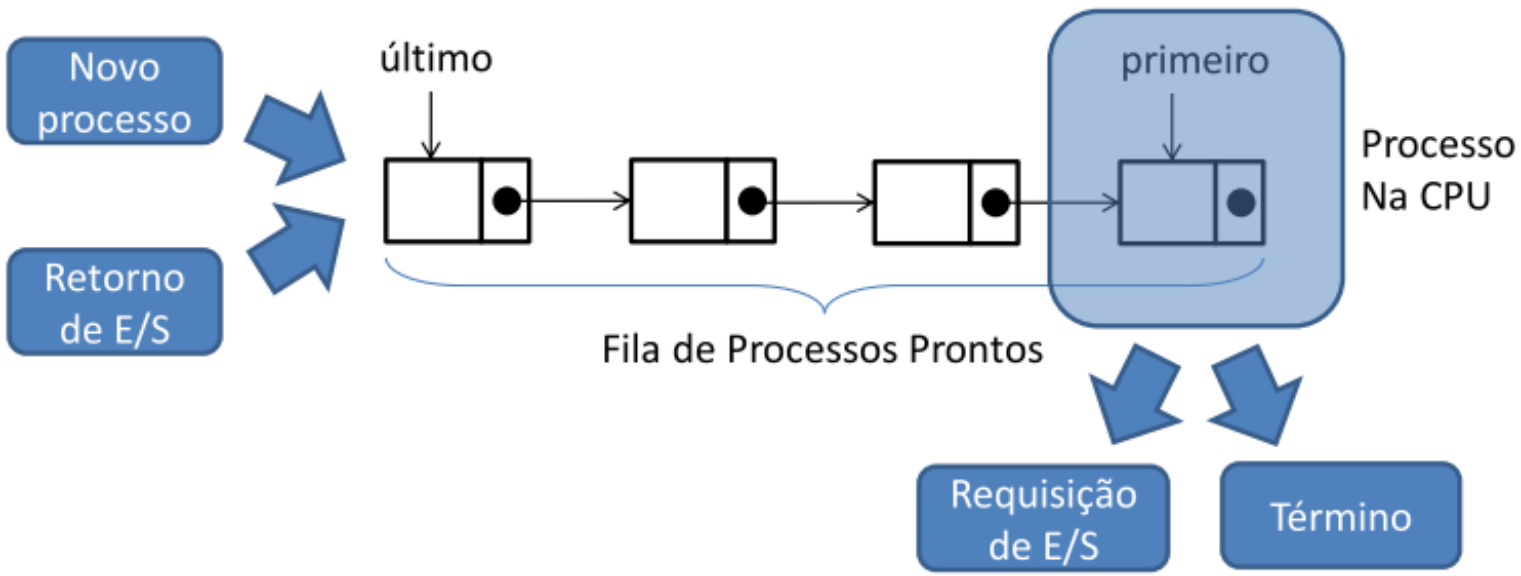
First Come First Served (FCFS) [3]

- Processo monopoliza a CPU pelo tempo que ele requerer ou até que ele requisite uma operação de E/S
- Basicamente existe uma fila única de processos prontos
- Requisições de E/S retiram o processo da fila e o colocam em estado bloqueado
- Quando um processo é desbloqueado, ele volta para o final da fila de processos prontos

First Come First Served (FCFS) [3]

- O algoritmo FCFS é fácil de entender e programar
- Além disso, é considerado um algoritmo justo, tratando igualmente os processos que deve escalonar
- Desvantagens
  - Não é adequado para processos *IO-bound* (por quê?)
  - Não é adequado para sistemas interativos (por quê?)

First Come First Served (FCFS) [3]



Shortest Job First (SJF) [3]

- Algoritmo não-preemptivo útil para sistemas em lote
- Como o nome diz, aloca ao processador a tarefa (processo) mais curta primeiro
- Utiliza a mesma lista encadeada de processos prontos do FCFS

Shortest Job First (SJF) [3]

- De modo geral, requer uma rotina que percorra toda a lista de processos prontos à procura daquele com o menor tempo estimado de processamento
- Contudo, em alguns casos, o tempo que um processo demorará é conhecido ou pode ser estimado, facilitando a atuação do SJF
- O algoritmo só é aplicável quando todos os processos estão disponíveis simultaneamente na lista de processos prontos

Shortest Job First (SJF) [3]

Figura: Estimativas de Tempo de Resposta (TR) e Tempo Médio de Resposta (TMR)

Shortest Job First (SJF) [3]

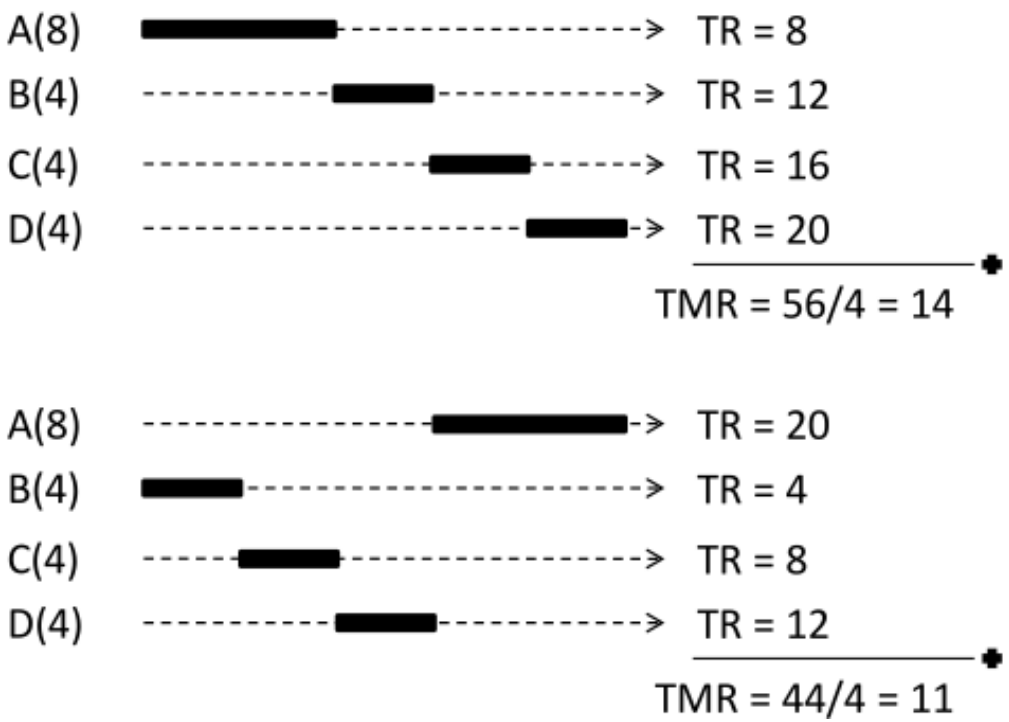
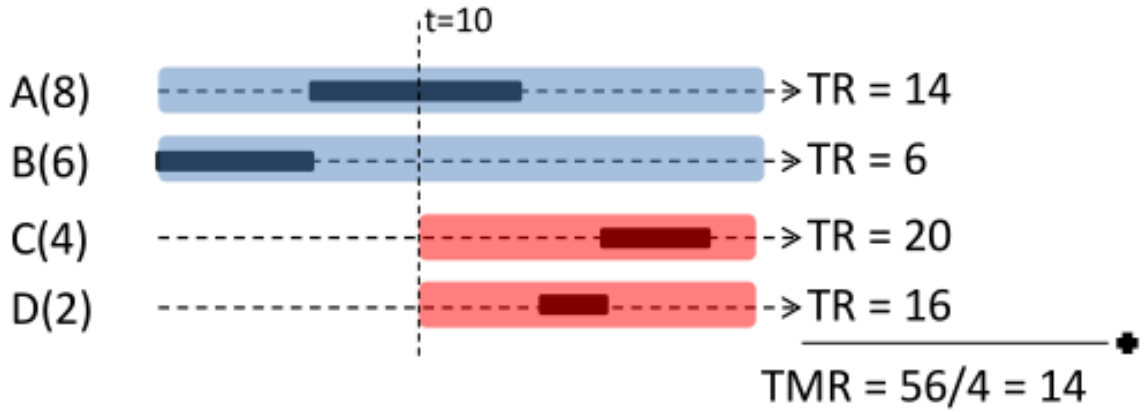
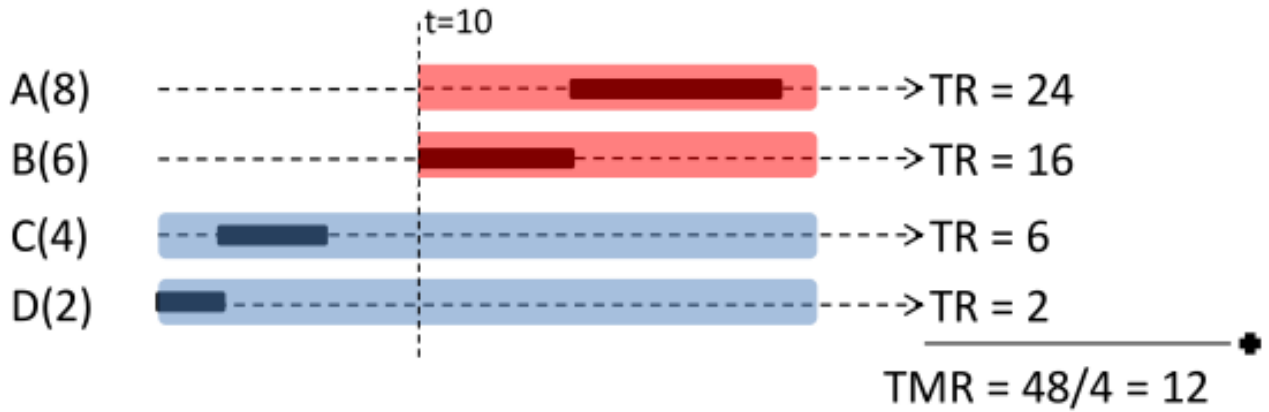


Figura: Estimativas de Tempo de Resposta (TR) e Tempo Médio de Resposta (TMR)

Shortest Job First (SJF) [3]

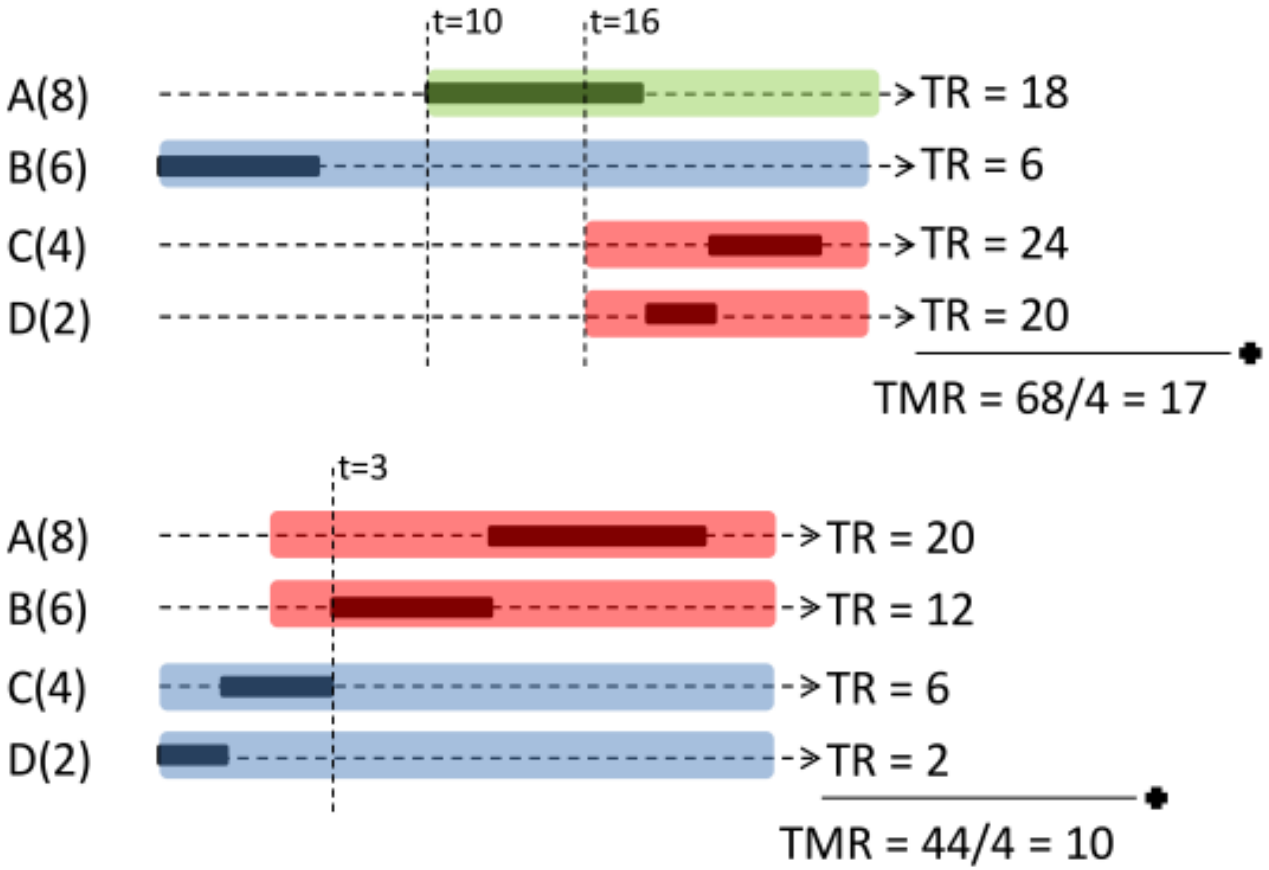
- O que acontece quando processos são inseridos na fila de prontos em tempos distintos?





Shortest Job First (SJF) [3]

- O que acontece quando processos são inseridos na fila de prontos em tempos distintos?



Shortest Remaining Time Next (SRTN) [3]

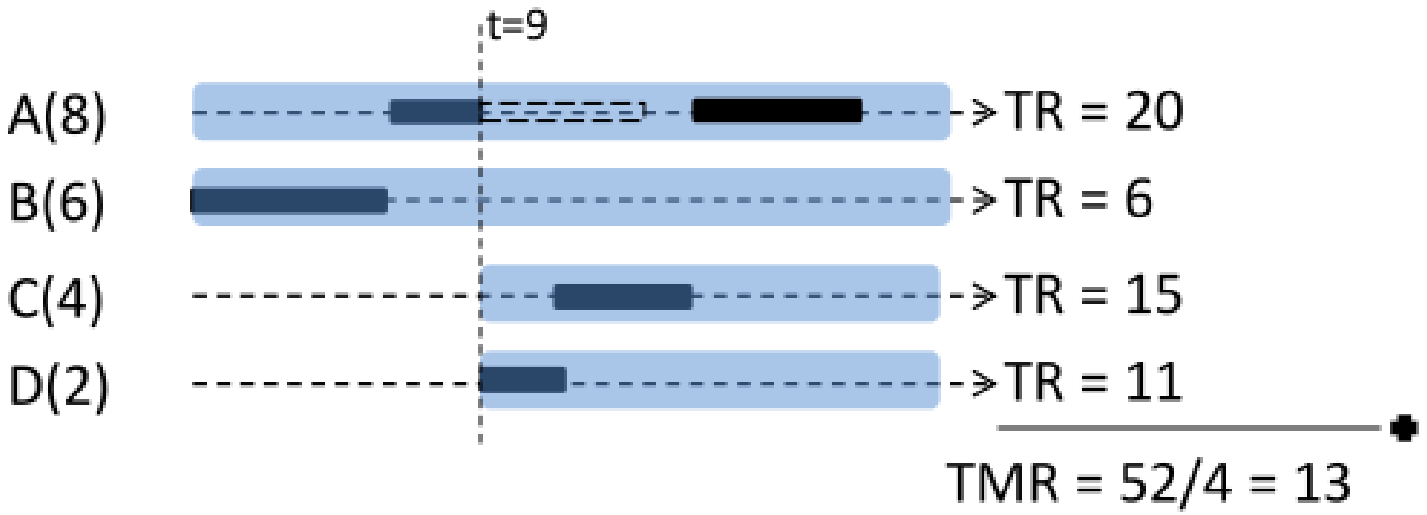
- Como nome diz, elege como próximo processo o que tem menor tempo restante para encerrar sua execução
- Requer que o tempo de execução seja conhecido *a priori*

Shortest Remaining Time Next (SRTN) [3]

- Como nome diz, elege como próximo processo o que tem menor tempo restante para encerrar sua execução
- Requer que o tempo de execução seja conhecido *a priori*
- Diferentemente dos anteriores, é preemptivo
- A preempção acontece apenas quando um processo pronto tem tempo restante menor que o processo atual

Shortest Remaining Time Next (SRTN) [3]

- O que acontece quando processos são inseridos na fila de prontos em tempos distintos?



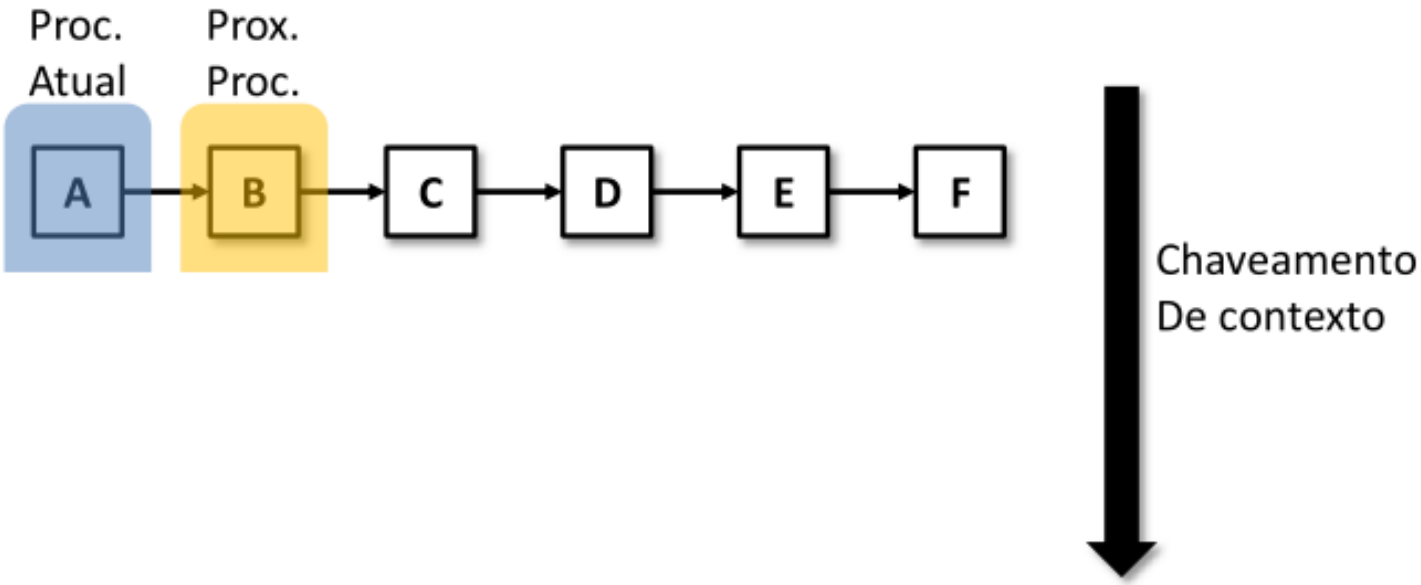
Round Robin (RR) [3]

- Escalonamento preemptivo
- Um dos algoritmos mais antigos, simples e justos
- Amplamente utilizado; virtualmente, todos SO usam *Round robin*

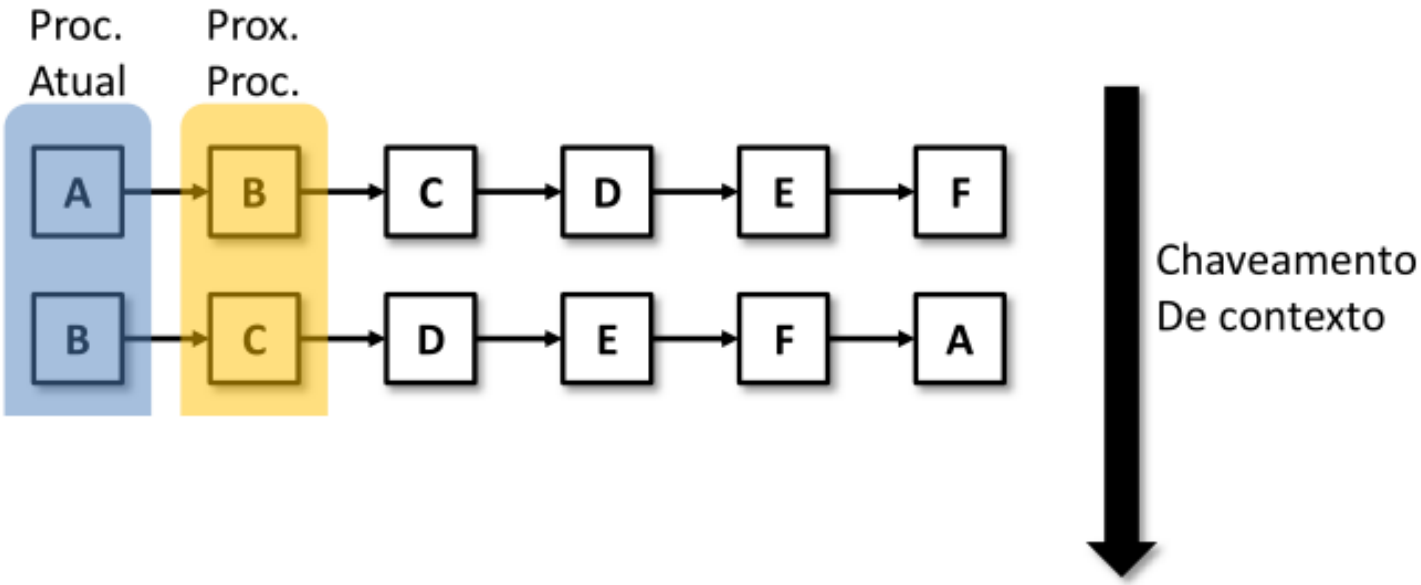
Round Robin (RR) [3]

- Implementação simples, baseada em lista circular
- A cada processo é atribuído uma “fatia de tempo” (quantum)
- Um processo executa até que
  - Tenha executado por um tempo igual ao quantum
  - Requeira E/S ou outro tipo de interrupção do processador

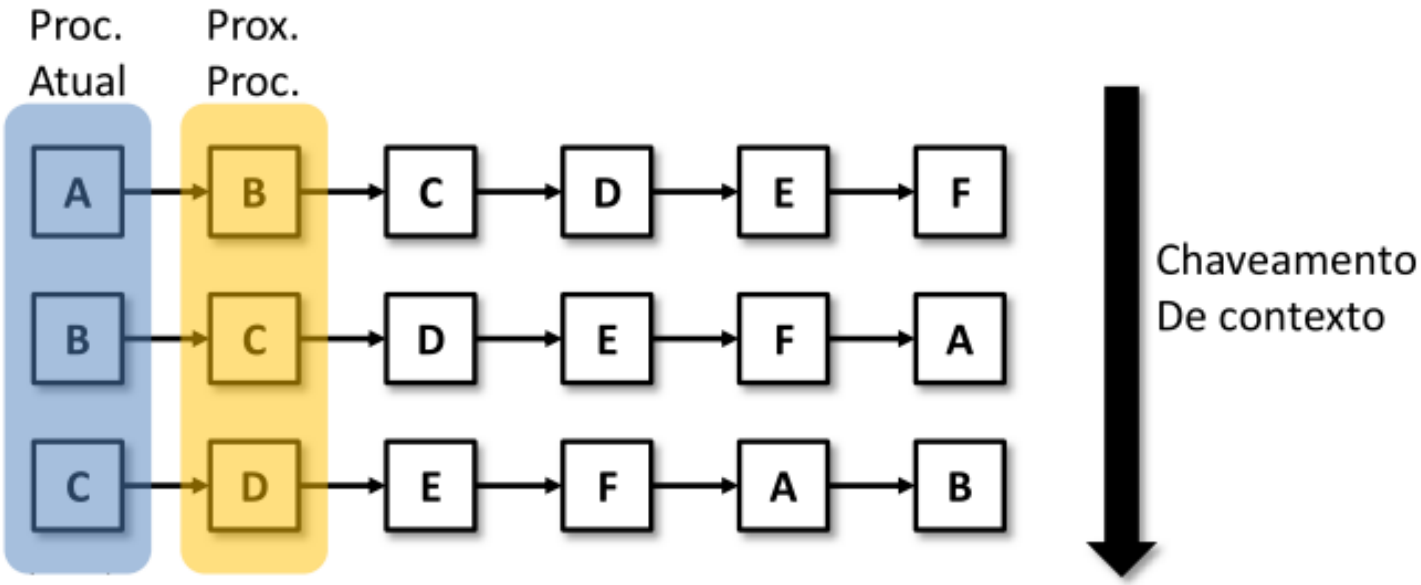
Round Robin (RR) [3]



Round Robin (RR) [3]



Round Robin (RR) [3]



Round Robin (RR) [3, 2]

- Em sua realização mais simples, todos os processos recebem o mesmo quantum
- Definir a duração do quantum pode ser problemático, como já mencionado nessa aula (escalonamento preemptivo)

Round Robin (RR) [3, 2]

- Em sua realização mais simples, todos os processos recebem o mesmo quantum
- Definir a duração do quantum pode ser problemático, como já mencionado nessa aula (escalonamento preemptivo)
- Na prática, valores de quantum entre 20 e 50 ms são utilizados
- Desvantagem?

Round Robin (RR) [3, 2]

- Em sua realização mais simples, todos os processos recebem o mesmo quantum
- Definir a duração do quantum pode ser problemático, como já mencionado nessa aula (escalonamento preemptivo)
- Na prática, valores de quantum entre 20 e 50 ms são utilizados
- Desvantagem? Nenhum processo termina antes de todos executarem um pouco

Escalonamento por Prioridades [3, 6]

- Escalonamento preemptivo
- Desdobramento do escalonamento circular

Escalonamento por Prioridades [3, 6]

- Escalonamento preemptivo
- Desdobramento do escalonamento circular
- Baseado na ideia de que processos não são igualmente importantes para o sistema e/ou usuário
- Exemplo: um processo que envia um email em *background* deveria ter prioridade menor do que um processo que está exibindo um filme na tela do computador em tempo real

Escalonamento por Prioridades [3, 2]

- O próximo processo a ser escolhido respeita a ordem de prioridades
- Pode ser implementado via fila de prioridades
- Atribui um número (prioridade) a cada processo

Escalonamento por Prioridades [3, 2]

- O próximo processo a ser escolhido respeita a ordem de prioridades
- Pode ser implementado via fila de prioridades
- Atribui um número (prioridade) a cada processo
- A prioridade pode ser alterada estática ou dinamicamente
  - Exemplo de prioridade estática: processos que monitoram a temperatura do urânio rodam primeiro
  - Exemplo de prioridade dinâmica: processos que estão mais perto do seu prazo rodam primeiro

Escalonamento por Prioridades [3, 6]

- Várias formas de ajustar dinamicamente a prioridade
  - Uso de quantum – especialização do *Round-Robin*

Escalonamento por Prioridades [3, 6]

- Várias formas de ajustar dinamicamente a prioridade
  - Uso de quantum – especialização do *Round-Robin*
  - A cada interrupção do relógio, a prioridade é decrementada para evitar que processos de alta prioridade ocupem sozinhos a CPU; verifica-se então se há algum processo na lista de prontos com prioridade maior que o processo atual para realizar a troca

Escalonamento por Prioridades [3, 6]

- Várias formas de ajustar dinamicamente a prioridade
  - Uso de quantum – especialização do *Round-Robin*
  - A cada interrupção do relógio, a prioridade é decrementada para evitar que processos de alta prioridade ocupem sozinhos a CPU; verifica-se então se há algum processo na lista de prontos com prioridade maior que o processo atual para realizar a troca
  - Decrementa-se a prioridade de um processo quando ele executa durante todo o quantum

Escalonamento por Prioridades [3, 6]

- Várias formas de ajustar dinamicamente a prioridade
  - Uso de quantum – especialização do *Round-Robin*
  - A cada interrupção do relógio, a prioridade é decrementada para evitar que processos de alta prioridade ocupem sozinhos a CPU; verifica-se então se há algum processo na lista de prontos com prioridade maior que o processo atual para realizar a troca
  - Decrementa-se a prioridade de um processo quando ele executa durante todo o quantum
  - Incrementa-se a prioridade de processos que bloqueiam antes de terminar o quantum (*IO-bound*), já que com pouco uso da CPU eles já podem bloquear novamente para realizar E/S

Filas multinível [3, 2]

- Chaveamento (troca) de processos é caro!
- Uma forma de aliviar o custo associado a preempção é executar diferentes processos por diferentes quantidades de tempo
- No escalonamento por filas multinível, um processo pode ser agendado para execução em um dos diferentes níveis (filas de execução)

Filas multinível [3, 2]

- Chaveamento (troca) de processos é caro!
- Uma forma de aliviar o custo associado a preempção é executar diferentes processos por diferentes quantidades de tempo
- No escalonamento por filas multinível, um processo pode ser agendado para execução em um dos diferentes níveis (filas de execução)
- Cada fila pode ter a sua política de escalonamento particular

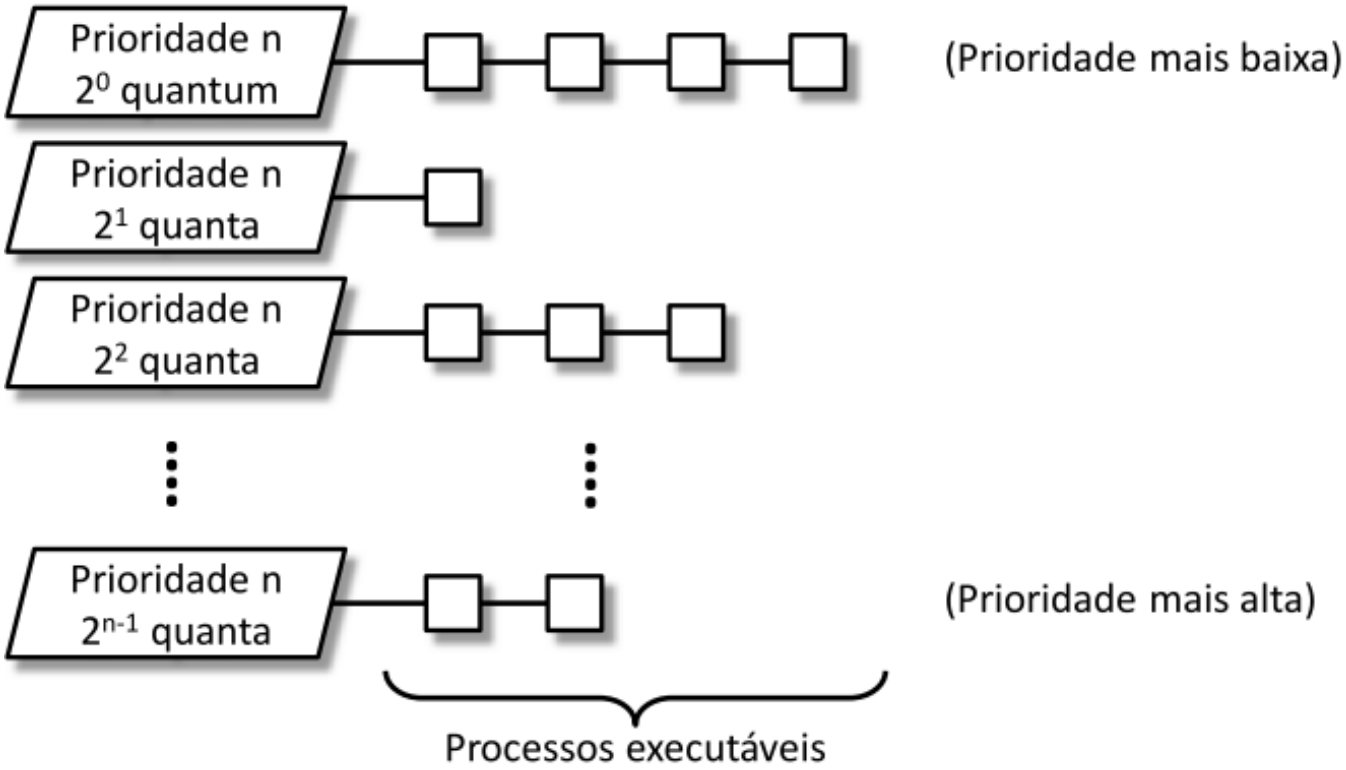
Filas multinível [3]

- No algoritmo de filas multinível, um processo é escolhido para execução sempre do nível mais alto
- Caso não hajam processos no nível mais alto, o nível subsequente é utilizado, e assim sucessivamente

Filas multinível [3]

- No algoritmo de filas multinível, um processo é escolhido para execução sempre do nível mais alto
- Caso não hajam processos no nível mais alto, o nível subsequente é utilizado, e assim sucessivamente
- Quando um processo executa durante todos os quanta a ele destinados (por sua altura de prioridade nas filas), ele é movido para o nível imediatamente menos prioritário
- Se um processo bloqueia antes de terminar o seu quanta, ele é movido para o nível imediatamente mais prioritário

Filas multinível [3]



Filas multinível no BSD UNIX [2]

- Filas múltiplas com realimentação – processos podem mudar de fila
  - Entre filas: escalonamento por prioridade
  - Dentro da fila: *Round robin*

Filas multinível no BSD UNIX [2]

- Filas múltiplas com realimentação – processos podem mudar de fila
  - Entre filas: escalonamento por prioridade
  - Dentro da fila: *Round robin*
- O processo muda de fila baseado na resposta a questões
  - Processo usou todo o quantum? Reduzir prioridade
  - Usou CPU por muito tempo? Reduzir prioridade
  - Está esperando por muito tempo (envelhecendo)? Aumentar prioridade

Filas multinível no BSD UNIX [2]

- Efeito
  - Processos interativos rodam mais rápido, pois estão em filas de maior prioridade
  - Processos que usam muito a CPU rodam depois
  - Prioridades mudam dinamicamente

Escalonamento garantido [3]

- Geralmente utilizado para dividir a atenção da CPU em sistemas multiusuário
- Uma garantia simples de cumprir seria a seguinte
  - Caso hajam N usuários conectados no sistema, a CPU será dividida entre os N usuários igualitáriamente resultado em $\frac{1}{N} - TP$ do tempo de CPU para os processos de cada usuário
  - TP se refere ao custo de chaveamento, *i.e.*, o tempo transcorrido executando o escalonador de processos pelo SO

Escalonamento garantido [3]

- Para fazer valer esta garantia, o escalonador tem que manter um controle de quanto tempo já foi gasto pela CPU nos processos de um dado usuário
- Deve manter um controle detalhado não apenas do número de usuários e número de processos por usuário, como também do tempo decorrido em cada processo
- A criação e/ou destruição de um processo, assim como a autenticação/saída de um usuário, demanda que toda a estrutura seja reformulada
- Logo, o algoritmo é difícil de implementar

Escalonamento por loteria [3]

- “Bilhetes” são atribuídos a cada processo
- Cada bilhete dá direito ao acesso a um tipo de recurso do computador
- Randomicamente um bilhete é sorteado
- O processo detentor do bilhete recebe acesso ao recurso em questão

Escalonamento por loteria [3]

- Em um cenário com 50 sorteios por segundo (1000 ms), há uma preempção a cada $\frac{1000}{50} = 20$ ms
- Caso o mesmo processo seja sorteado duas vezes não há necessidade de troca de contexto
- Prioridade pode ser implementada conferindo mais bilhetes ao mesmo processo

Escalonamento por loteria [3]

- Se considerarmos apenas os processos e não os usuários, uma situação um “injusta” pode ocorrer
  - Imagine que o usuário A possui 9 processos, enquanto o usuário B contém 1 processo
  - O usuário A consumirá então 90% do tempo de CPU, enquanto o somente usuário B somente 10%

Avaliação de algoritmos de escalonamento [2]

- Alguns métodos podem ser aplicados para avaliar e comparar o desempenho de algoritmos de escalonamento
  - Avaliação analítica
  - Simulação
- Esses métodos podem ser úteis para decidir qual algoritmo usar em um SO em desenvolvimento

Avaliação analítica [2]

- Assume um conjunto fixo de processos
- Calcula-se como cada algoritmo escalona este conjunto
- Determinam-se medidas como o tempo de resposta nesse cenário
  - Vantagem: fácil de calcular
  - Desvantagem: conjunto fixo de processos

Simulação [2]

- Implementa-se e aplica-se um algoritmo para um número grande de processos
- Calculam-se medidas a partir dos resultados da execução
  - Vantagem: qualquer algoritmo pode ser considerado
  - Desvantagem: resultados não são garantidos em uma situação prática

Objetivo geral desta aula

Introduzir conceitos de processos em sistemas operacionais

Sumário

Motivação para comunicação entre processos [2]

- A comunicação de processos é importante para aspectos variados
  - Compartilhamento de recursos: um computador é usado por muitos usuários (humanos ou não)
  - Multiprocessadores: um problema é resolvido mais rápido se for dividido entre múltiplos processadores
  - Modularidade: dividir o problema em problemas menores (ex: escalonador, comunicação entre processos, ...)
  - Sistemas distribuídos: Internet e outros sistemas que contém componentes interligados em rede que se comunicam

Condições de corrida [2]

- Processos podem se comunicar através do compartilhamento de variáveis
- Quais problemas podem ocorrer quando dois ou mais processos compartilham variáveis?

Condições de corrida [2]

Considere 2 processos que compartilham as variáveis A e B:

P_1	P_2	Qual será o resultado final? A ordem de execução dos processos tem importância?
A = 1	B = 2	

Condições de corrida [2]

Considere 2 processos que compartilham as variáveis A e B:

P_1	P_2	Qual será o resultado final? A ordem de execução dos processos tem importância?
A = 1	B = 2	

Agora considere:

P_1	P_2	Qual é o resultado final?
A = B + 1	B = 2 * B	

Condições de corrida [2]

Considere 2 processos que compartilham as variáveis A e B:

P_1	P_2
$A = 1$	$B = 2$

Qual será o resultado final? A ordem de execução dos processos tem importância?

Agora considere:

P_1	P_2
$A = B + 1$	$B = 2 * B$

Qual é o resultado final?

E no seguinte caso?

P_1	P_2
$A = 1$	$A = 2$

Qual é o resultado final?

E em um computador com múltiplos processadores?

Condições de corrida em situação real [3]

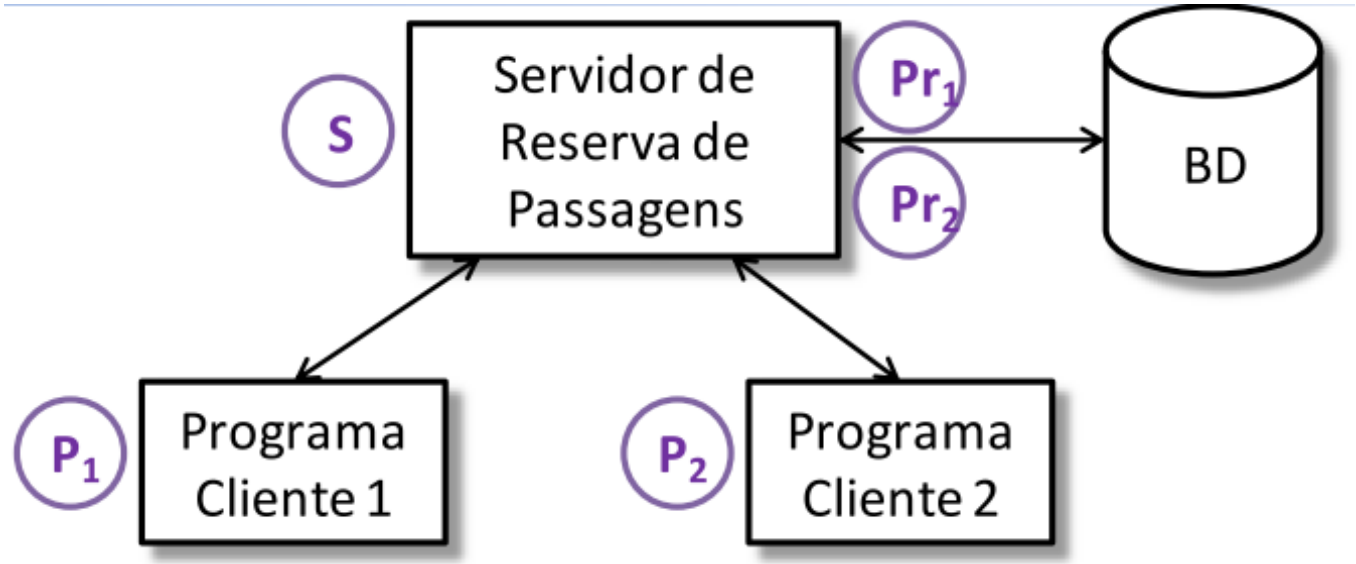


Figura: Exemplo de condição de corrida com processos que executam em computadores diferentes e que desejam reservar o mesmo assento

Operações atômicas [2]

- A fim de evitar condições de corrida, o conceito de operações atômicas é introduzido
- Operações atômicas são operações que não podem ser interrompidas
  - Não é possível ver as “partes” de uma operação atômica, mas apenas seu efeito final
  - Ou seja, não é possível ver uma “operação em progresso”

Operações atômicas [2]

- Exemplos de operações atômicas
  - Tocar a campainha
  - Desligar a luz
- Exemplos de operações não-atômicas
  - Encher um copo de água
  - Caminhar até a porta

Operações atômicas [2]

- Operações atômicas são relevantes em outras áreas além de SO
  - Elas são a base para transações atômicas que, por sua vez, formam uma base para uma área denominada Processamento de Transações
  - Esta área trata de problemas de coordenação de acessos múltiplos e concorrentes a bancos de dados; bancos eletrônicos são uma das aplicações importantes desta área (por quê?)

Operações atômicas [2]

- Em geral, o hardware provê algumas operações atômicas
  - Se o hardware não fornecer uma determinada operação atômica, como é possível implementá-la em um uniprocessador?
  - E em multiprocessador?

Operações atômicas [2]

- Em geral, o hardware provê algumas operações atômicas
  - Se o hardware não fornecer uma determinada operação atômica, como é possível implementá-la em um uniprocessador?
  - E em multiprocessador?
- A resposta pode ser obtida a partir do conceito de sincronização, um mecanismo baseado em operações atômicas simples que garante o funcionamento correto de processos que cooperam

Sincronização [2]

- O problema do espaço na geladeira

Hora	Pessoa A	Pessoa B
6:00	Olha a gel.: sem leite	...
6:05	Sai para a padaria	...
6:10	Chega na padaria	Olha a gel.: sem leite
6:15	Sai da padaria	Sai para a padaria
6:20	Em casa: guarda leite	Chega na padaria
6:25	...	Sai da padaria
6:30	...	Chega em casa: Ops!

- O que houve de errado?

Sincronização [2]

- O problema do espaço na geladeira

Hora	Pessoa A	Pessoa B
6:00	Olha a gel.: sem leite	...
6:05	Sai para a padaria	...
6:10	Chega na padaria	Olha a gel.: sem leite
6:15	Sai da padaria	Sai para a padaria
6:20	Em casa: guarda leite	Chega na padaria
6:25	...	Sai da padaria
6:30	...	Chega em casa: Ops!

- O que houve de errado? Falta de comunicação

Sincronização [2]

- O problema anterior era causado porque uma pessoa não sabia o que a outra estava fazendo
- Uma solução para o problema envolve dois novos conceitos
  - Exclusão mútua
    - Apenas um processo pode fazer alguma coisa em determinado momento
    - Exemplo: apenas uma pessoa pode sair para comprar leite em qualquer momento

Sincronização [2]

- O problema anterior era causado porque uma pessoa não sabia o que a outra estava fazendo
- Uma solução para o problema envolve dois novos conceitos
  - Seção crítica
    - Uma seção de código na qual apenas um processo pode executar de cada vez
    - O objetivo é tornar atômico o conjunto de operações que estão na seção crítica
    - Exemplo: comprar leite

Exclusão mútua [2]

- Existem várias maneiras de se obter exclusão mútua
- A maioria envolve trancamento (*locking*)
  - Evitar que alguém faça alguma coisa em determinado momento
  - Exemplo: deixar um aviso na porta da geladeira

Exclusão mútua [2]

- Três regras devem ser satisfeitas para o trancamento funcionar
  1. Trancar antes de utilizar – no caso da geladeira, corresponde a deixar aviso
  2. Destrancar quando terminar – retirar aviso
  3. Esperar se estiver trancado – não sai para comprar se houver aviso

Exclusão mútua [2]

- Primeira tentativa de resolver o Problema do Espaço na Geladeira

Processos A e B

```
if (SemLeite) {
    if (SemAviso) {
        Deixa Aviso;
        Compra Leite;
        Remove Aviso;
    }
}
```

- Esta “solução” funciona?

Exclusão mútua [2]

- Nem sempre, por causa da troca de contexto

Processos A e B

```
if (SemLeite) {
   if (SemAviso) {
      Deixa Aviso;
      Compra Leite;
      Remove Aviso;
   }
}
```

Exclusão mútua [10, 2]

- A “solução” piorou o problema!
- Agora, falha só de vez em quando, ou seja, a depuração fica muito mais difícil

Exclusão mútua [10, 2]

- A “solução” piorou o problema!
- Agora, falha só de vez em quando, ou seja, a depuração fica muito mais difícil
- Um exemplo de problema é se cada processo executasse uma linha por vez antes da troca de contexto
- *Heisenbug*: não importa se (o *bug*) é raro, na prática vai acontecer nos primeiros 5 minutos. A não ser quando você estiver querendo que aconteça!”

Exclusão mútua [2]

- Segunda tentativa de resolver o Problema do Espaço na Geladeira – mudar o significado de aviso

Processo A

```
if (SemAviso) {
   if (SemLeite) {
      Compra Leite;
   }
   Deixa Aviso;
}
```

Processo B

```
if (Aviso) {
   if (SemLeite) {
      Compra Leite;
   }
   Remove Aviso;
}
```

- Esta “solução” funciona? Por quê?

Exclusão mútua [2]

- Que tal o seguinte argumento?
  - Somente A deixa um aviso, e somente se já não existe um aviso
  - Somente B remove um aviso, e somente se houver um aviso
  - Portanto, ou existe um aviso, ou nenhum

Exclusão mútua [2]

- Segunda tentativa de resolver o Problema do Espaço na Geladeira – mudar o significado de aviso

Processo A

```
if (SemAviso) {
   if (SemLeite) {
      Compra Leite;
   }
   Deixa Aviso;
}
```

Processo B

```
if (Aviso) {
   if (SemLeite) {
      Compra Leite;
   }
   Remove Aviso;
}
```

- Esta “solução” funciona? Por quê?

Exclusão mútua [2]

- Que tal o seguinte argumento?
  - Se houver aviso, B compra leite
  - Se não houver aviso, A compra leite
  - Portanto, apenas uma pessoa (processo) vai comprar leite
- Vocês estão de acordo?

Exclusão mútua [2]

- Que tal o seguinte argumento?
  - Se houver aviso, B compra leite
  - Se não houver aviso, A compra leite
  - Portanto, apenas uma pessoa (processo) vai comprar leite
- Vocês estão de acordo? Se sim, a solução parece mesmo boa

Exclusão mútua [2]

- E o que acontece se B sair de férias, *i.e.*, parar de executar?
  - A vai comprar leite uma vez e não vai comprar mais até que B retorne
  - Portanto, esta solução não é boa; em particular, ela pode levar a uma inanição (*starvation*) do processo A
  - Inanição é um conceito importante em SO, sendo caracterizado por um processo que aguarda por um evento que nunca ocorre

Exclusão mútua [2]

- Terceira tentativa de resolver o Problema do Espaço na Geladeira – usar dois avisos diferentes

Processo A

```
Deixa AvisoA;
while (AvisoB);
if (SemLeite) {
   Compra Leite;
}
Remove AvisoA;
```

Processo B

```
Deixa AvisoB;
if (SemAvisoA) {
   if (SemLeite) {
      Compra Leite;
   }
}
Remove AvisoB;
```

- Esta “solução” funciona?

Exclusão mútua [2]

- Terceira tentativa de resolver o Problema do Espaço na Geladeira – usar dois avisos diferentes

Processo A

```
Deixa AvisoA;
while (AvisoB);
if (SemLeite) {
   Compra Leite;
}
Remove AvisoA;
```

Processo B

```
Deixa AvisoB;
if (SemAvisoA) {
   if (SemLeite) {
      Compra Leite;
   }
}
Remove AvisoB;
```

- Esta “solução” funciona? Sim!

Exclusão mútua [2]

- Argumento
  - Se `SemAvisoA`, B pode comprar porque A ainda não começou
  - Se `AvisoA`, A está comprando ou esperando até que B desista; logo, B pode desistir

Exclusão mútua [2]

- Argumento
  - Se `SemAvisoA`, B pode comprar porque A ainda não começou
  - Se `AvisoA`, A está comprando ou esperando até que B desista; logo, B pode desistir
  - Se `SemAvisoB`, A pode comprar
  - Se `AvisoB`...
    - Se B comprar, B remove `AvisoB` e encerra fim
    - Se B não comprar, B remove `AvisoB` e A pode comprar

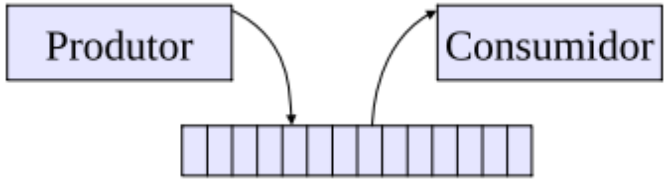
Exclusão mútua [2]

- Esta solução funciona, mas não é boa
  - Muito complicado, pois é difícil de entender e se convencer de que está correto
  - Código de A é diferente de B; e se houver mais de dois processos?
  - Enquanto A está esperando, está consumindo CPU, levando ao fenômeno de *busy waiting* (espera ocupada)

Exclusão mútua [2]

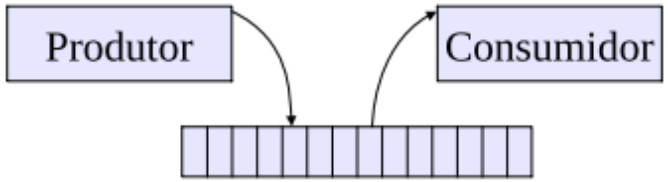
- Pontos importantes
  - Comportamento muito sutil, pois é difícil de programar e entender
  - Como provar que está correto?
  - Quais são os critérios para uma boa solução?

Produtor e consumidor [2]



- Problema tradicional de exclusão mútua em que um produtor gera itens (dados) continuamente e os coloca em um *buffer*
- Consumidor usa itens, lendo-os do *buffer*
- *Buffer* é necessário por causa da velocidade relativa entre produtor e consumidor

Produtor e consumidor [2]

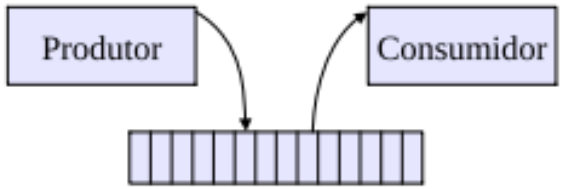


- Sincronização é necessária para acesso ao *buffer*
  - Produtor não pode colocar mais itens em *buffer* cheio
  - Consumidor não pode ler itens de *buffer* vazio

Produtor e consumidor [2]

- Solução “ideal”: usa todas as posições do *buffer*

```
Produtor() {
    while (true) {
        while (counter == n);
        buffer[in]= item produzido;
        in= in+1 mod n;
        counter++;
    }
}
Consumidor() {
    while (true) {
        while (counter == 0);
        item consumido= buffer[out];
        out= out+1 mod n;
        counter--;
    }
}
```



- *Buffer* circular

Sincronização [2]

- Requisitos para uma primitiva de exclusão mútua
  - Deve permitir apenas um processo dentro da região crítica a cada instante
  - Se várias requisições são feitas ao mesmo tempo, deve permitir que um processo prossiga
  - Processos podem “entrar de férias”, *i.e.*, param de executar temporariamente somente fora das regiões críticas

Sincronização [2]

- Propriedades desejáveis para uma primitiva de exclusão mútua
  - Justiça (*fairness*): se vários processos estão esperando, dar acesso a todos, em algum momento
  - Eficiência: um processo não deve utilizar quantidades substanciais de recursos quando estiver esperando; em particular, deve evitar a espera ocupada
  - Simples: deve ser fácil de utilizar

Sincronização [2]

- Propriedades dos processos utilizando os mecanismos necessários para manter coerência
  - Trancar (*to lock*) sempre antes de utilizar dado compartilhado
  - Destrancar sempre que terminar o uso do dado compartilhado

Sincronização [2]

- Propriedades dos processos utilizando os mecanismos necessários para manter coerência
  - Trancar (*to lock*) sempre antes de utilizar dado compartilhado
  - Destrancar sempre que terminar o uso do dado compartilhado
  - Não trancar de novo se já tiver trancado o recurso
  - Não ficar muito tempo dentro das seções críticas

Sincronização básica com *locking* [2]

```
while (!fim) {
    seção_não_crítica;
    lock();
    seção_crítica;
    unlock();
}
```

Implementação de exclusão mútua [2]

- Via software
  - Soluções para dois processos
  - Soluções para múltiplos processos
- Via hardware
  - Desabilitando interrupções
  - *Read-modify-write* (variação *test* & *set*)

Implementação de exclusão mútua [2]

```
while (!fim) {
    seção_não_crítica;
    lock();
    seção_crítica;
    unlock();
}
```

- Uma solução para o problema tem que satisfazer 3 propriedades
  - Exclusão mútua
  - Progresso
  - Espera limitada: sem inanição
- Mas não se sabe nada sobre a velocidade de cada processo

Implementação de exclusão mútua: dois processos [2]

```
lock()
{
  while (vez != i);
};

unlock()
{
  vez = j;
};
```

Figura: Algoritmo 1 com solução baseada em software para exclusão mútua entre dois processos

Implementação de exclusão mútua: dois processos [2]

- Propriedades do Algoritmo 1
  - Exclusão mútua: OK
  - Progresso: problema (se um processo possui seção não crítica muito mais lenta que o outro, pode bloquear o outro por muito tempo) – vide exemplo mais completo a seguir
  - Espera limitada: problema
- Algoritmo deve guardar mais informações para funcionar adequadamente

Implementação de exclusão mútua: dois processos [6]

```
while (TRUE) {
 while (turn != 0) /* espera */;
 critical_region();
 turn = 1;
 noncritical_region();
}
```

(a)

```
while (TRUE) {
 while (turn != 1 ) /* espera */;
 critical_region();
 turn = 0;
 noncritical_region();
}
```

(b)

Figura: Código de dois processos que aplicam ideias do Algoritmo 1

Implementação de exclusão mútua: dois processos [2]

```
lock()
{
  quer_entrar[i] = 1;
  while (quer_entrar[j]);
};

unlock()
{
  quer_entrar[i] = 0;
};
```

Figura: Algoritmo 2 com solução baseada em software para exclusão mútua entre dois processos

Implementação de exclusão mútua: dois processos [2]

- Propriedades do Algoritmo 2
  - Exclusão mútua: OK
  - Progresso: problema – *deadlock*, uma espécie de nó que pode ocorrer entre processos
  - Espera limitada: problema

Implementação de exclusão mútua: dois processos [2]

```
lock() {
  quer_entrar[i] = 1;
  vez = j;
  while
    (quer_entrar[j] &&  vez == j);
};

unlock() {
  quer_entrar[i] = 0;
};
```

Figura: Algoritmo 3 com solução baseada em software para exclusão mútua entre dois processos

Implementação de exclusão mútua: dois processos [2]

- Propriedades do Algoritmo 3
  - Exclusão mútua: OK
  - Progresso: OK
  - Espera limitada: OK

Implementação de exclusão mútua: múltiplos processos [2]

- Algoritmo do padeiro
  - Ao entrar na padaria o cliente recebe um número
  - Quem tiver o número menor é atendido
  - Pode acontecer de dois processos receberem o mesmo número; nesse caso, um pode ser escolhido arbitrariamente (exemplo: o de menor ID)

Implementação de exclusão mútua: múltiplos processos [2]

```
lock() {

  numero[i] = max(numero[0..n-1]) + 1;

  for (j = 0; j < n; j++) {

    while ((numero[j] != 0) &&
           (numero[j],j)<(numero[i],i));
  };
}
unlock() {
  numero[i] = 0;
};
```

(a,b) < (c,d) se
(a < c) ou (a = c e b < d)

Implementação de exclusão mútua: múltiplos processos [2]

```
lock() {
  escolhendo[i] = 1;
  numero[i] = max(numero[0..n-1]) + 1;
  escolhendo[i] = 0;
  for (j = 0; j < n; j++) {
    while (escolhendo[j]);
    while ((numero[j] != 0) &&
           (numero[j],j)<(numero[i],i));
  };
}
unlock() {
  numero[i] = 0;
};
```

(a,b) < (c,d) se
(a < c) ou (a = c e b < d)

Implementação de exclusão mútua: múltiplos processos [2]

- Correto porque
  - Processo sempre escolhe um número maior (ou igual) ao maior existente
  - O processo que aguarda tem sempre um número maior (ou ID de processo maior) do que o processo que está acessando a seção crítica
  - Política de escolha FIFO

Implementação de exclusão mútua: múltiplos processos [3]

- Semáforo
  - Criado por Dijkstra em 1965 para resolver o problema do produtor-consumidor
  - Um semáforo é compartilhado entre múltiplos processos e encapsula uma variável inteira e duas operações
    - *down* (*sleep* ou *wait*): testa se a variável no semaforo é maior que zero, de modo que se for maior, decrementa-a em uma unidade; se for igual a zero, bloqueia o processo que chamou *sleep* antes de decrementar

Implementação de exclusão mútua: múltiplos processos [3]

- Semáforo
  - Criado por Dijkstra em 1965 para resolver o problema do produtor-consumidor
  - Um semáforo é compartilhado entre múltiplos processos e encapsula uma variável inteira e duas operações
    - *down* (*sleep* ou *wait*): testa se a variável no semaforo é maior que zero, de modo que se for maior, decrementa-a em uma unidade; se for igual a zero, bloqueia o processo que chamou *sleep* antes de decrementar
    - *up* (*wakeup* ou *signal*): incrementa a variável no semáforo; se o valor era zero, testa se há processos bloqueados, e em caso afirmativo desbloqueia um deles, o qual conclui *sleep* decrementando a variável no semáforo

Implementação de exclusão mútua: múltiplos processos [3]

- Semáforo
  - Criado por Dijkstra em 1965 para resolver o problema do produtor-consumidor
  - Um semáforo é compartilhado entre múltiplos processos e encapsula uma variável inteira e duas operações
    - *down* (*sleep* ou *wait*): testa se a variável no semaforo é maior que zero, de modo que se for maior, decrementa-a em uma unidade; se for igual a zero, bloqueia o processo que chamou *sleep* antes de decrementar
    - *up* (*wakeup* ou *signal*): incrementa a variável no semáforo; se o valor era zero, testa se há processos bloqueados, e em caso afirmativo desbloqueia um deles, o qual conclui *sleep* decrementando a variável no semáforo
  - Exemplo com processos A, B e C

Implementação de exclusão mútua: múltiplos processos [6]

- Monitor
  - Primitiva que agrupa dados e procedimentos
  - Processos...
    - Podem chamar esses procedimentos
    - Não podem acessar os dados internos

Implementação de exclusão mútua: múltiplos processos [6]

- Monitor
  - Somente um processo pode estar ativo na seção crítica protegida por um monitor
  - A implementação é realizada pelo compilador
  - Comparado com semáforo, o monitor é...
    - Mais amigável ao programador
    - Menos suscetível a erros

Implementação de exclusão mútua: múltiplos processos [6]

- Monitor
  - Dependência de variáveis de condição
    - Quando procedimento P não pode seguir, chama uma operação denominada *wait* na variável
    - O processo P1 que chamou P é então bloqueado
    - Um processo P2 que aguardava sua vez é então autorizado a acessar o monitor

Implementação de exclusão mútua: múltiplos processos [6]

- Monitor
  - Dependência de variáveis de condição
    - Quando procedimento P não pode seguir, chama uma operação denominada *wait* na variável
    - O processo P1 que chamou P é então bloqueado
    - Um processo P2 que aguardava sua vez é então autorizado a acessar o monitor
    - Quando P2 chamar *signal* na variável...
    - ...ele permitirá que um outro processo seja desbloqueado
  - Mais detalhes sobre semáforo e monitor: referências básicas como o livro do Tanenbaum são boas opções

Implementação de exclusão mútua via hardware [2]

- Desabilitando interrupções
  - Em um uniprocessador, operações serão atômicas se não houver troca de contexto
  - Trocas de contexto acontecem quando o escalonador é chamado
  - Vantagem: simples e eficiente
  - Desvantagens
    - Não funciona em multiprocessadores
    - Se o usuário puder desabilitar interrupções, o SO perde controle da CPU

Implementação de exclusão mútua via hardware [2]

```
lock()
{
  disable_interrupts();
};

unlock()
{
  enable_interrupts();
};
```

Implementação de exclusão mútua via hardware [2]

- *Read-modify-write*
  - A maioria dos processadores modernos implementa alguma forma de *read-modify-write*
    - Estas instruções leem um valor da memória, o atualizam e gravam na memória de forma atômica, implementadas por hardware

Implementação de exclusão mútua via hardware [2]

- *Read-modify-write*
  - A maioria dos processadores modernos implementa alguma forma de *read-modify-write*
    - Estas instruções leem um valor da memória, o atualizam e gravam na memória de forma atômica, implementadas por hardware
    - Implementação em multiprocessadores é complicada: necessita modificações no protocolo de coerência de *cache* – protocolo para evitar que processadores tenham cópias desatualizadas de um mesmo dado

Implementação de exclusão mútua via hardware [2]

- *Read-modify-write*
  - A maioria dos processadores modernos implementa alguma forma de *read-modify-write*
    - Estas instruções leem um valor da memória, o atualizam e gravam na memória de forma atômica, implementadas por hardware
    - Implementação em multiprocessadores é complicada: necessita modificações no protocolo de coerência de *cache* – protocolo para evitar que processadores tenham cópias desatualizadas de um mesmo dado
    - Uma das variações mais comuns é o *test* & *set*, uma operação indivisível com um parâmetro que verifica (e retorna) o valor de uma variável e, após, atribui o valor do seu parâmetro à variável

Implementação de exclusão mútua via hardware [2]

```
int M = 0;
lock(M)
{
  while (test&set(M) == 1);
};

unlock(M)
{
  M = 0;
};
```

Figura: Operação mencionada retorna o valor de M (antes de atribuir 1 para M) para testar se M originalmente era igual a 1

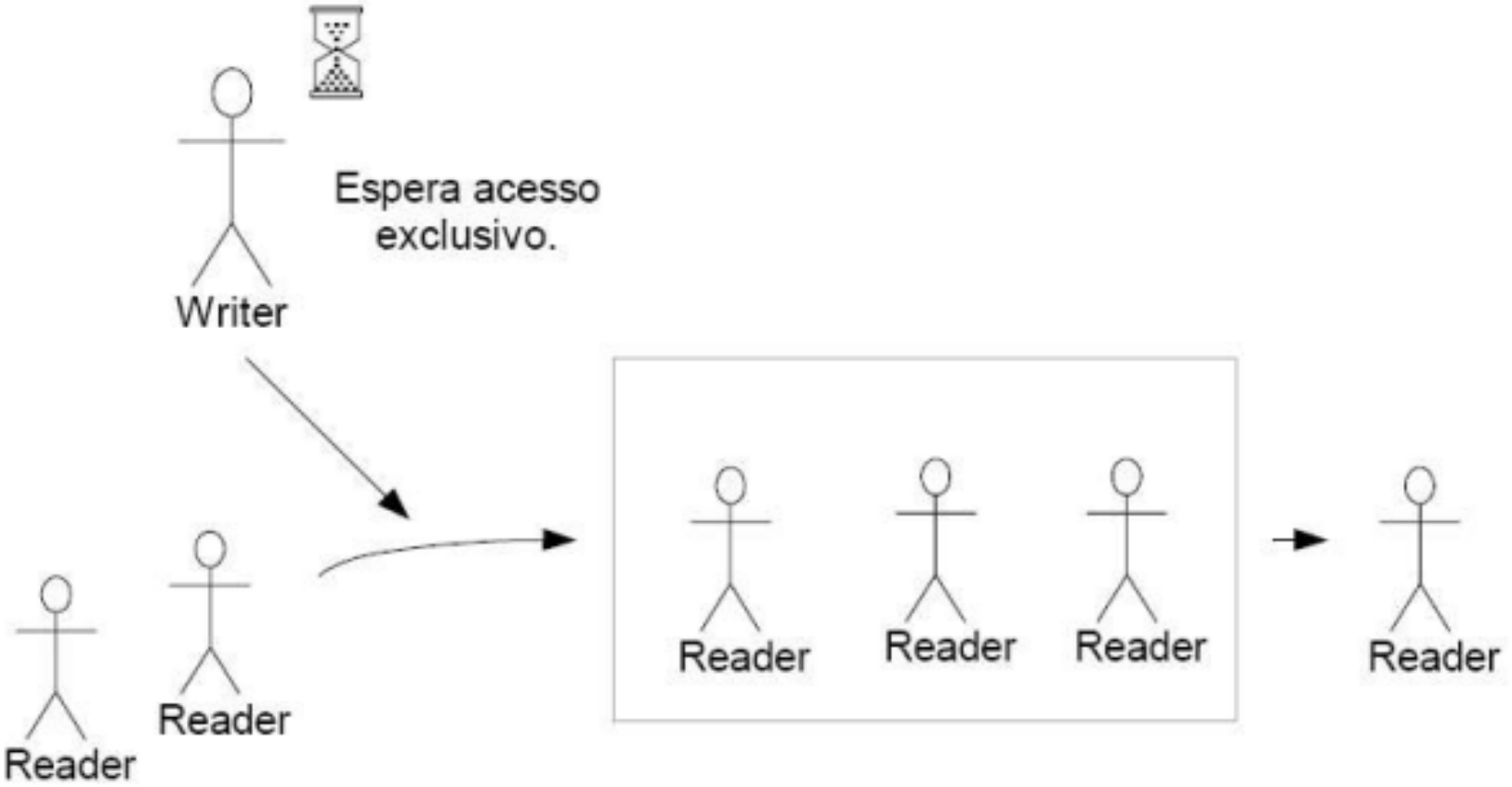
Hardware *vs* software [2]

- Se é tão mais simples acessar exclusão mútua via hardware, porque estudar via software?
  - Porque existem processadores que não possuem primitivas para exclusão mútua via hardware! Exemplo: MIPS (DecStation)
  - Porque exclusão mútua via hardware só fornece primitivas de muito baixo nível
  - Exclusão mútua via software serve de introdução a problemas mais complexos e importantes

Problemas clássicos [6, 2]

- Os problemas clássicos de comunicação representam modelos gerais para problemas reais de sincronização de recursos e processos
- Por meio deles, é possível por exemplo avaliar a qualidade de métodos e primitivas de sincronização
- Três problemas frequentemente estudados
  - Leitores e escritores
  - Barbeiro dorminhoco
  - Jantar dos filósofos

Leitores e escritores [4]



Leitores e escritores [4]

- Problema
  - Suponha que existe um conjunto de processos que compartilham um determinado conjunto de dados (ex: um banco de dados)
  - Existem processos que lêem os dados
  - Existem processos que escrevem (gravam) os dados

Leitores e escritores [4]

- Análise do problema
  - Se dois ou mais leitores acessarem os dados simultaneamente, não há problemas
  - E se um escritor quiser escrever sobre os dados, há alguma restrição?
  - Dois ou mais escritores podem acessar os dados simultaneamente?

Leitores e escritores [4]

- Em uma das possíveis soluções para o problema apresentado, há prioridade para leitores
  - Leitores podem ter acesso simultâneo aos dados compartilhados
  - Os escritores podem apenas ter acesso exclusivo aos dados compartilhados

Leitores e escritores [4]

```
//número de leitores ativos
  int rc

//protege o acesso à variável rc
  Semaphore mutex

//Indica a um escritor se este
//pode ter acesso aos dados
Semaphore db

//Incialização:
    mutex=1,
     db=1,
     rc=0
```

Escritor

```
while (TRUE)
  down(db);
  ...
  //writing is
  //performed
  ...
  up(db);
  ...
```

Leitor

```
while (TRUE)
  down(mutex);
  rc++;
  if (rc == 1)
     down(db);
  up(mutex);
  ...
  //reading is
  //performed
  ...
  down(mutex);
  rc--;
  if (rc == 0)
     up(db);
  up(mutex);
```

Leitores e escritores [4, 6]

- A solução anterior pode levar à inanição dos escritores
- Para evitar esse problema, é possível alterar o programa para realizar as seguintes ações
  - Quando um leitor chega e um escritor está esperando no semáforo `db`, o leitor é bloqueado atrás do escritor ao invés de ser admitido imediatamente

Leitores e escritores [4, 6]

- A solução anterior pode levar à inanição dos escritores
- Para evitar esse problema, é possível alterar o programa para realizar as seguintes ações
  - Quando um leitor chega e um escritor está esperando no semáforo `db`, o leitor é bloqueado atrás do escritor ao invés de ser admitido imediatamente
  - Desse modo, um escritor precisa esperar os leitores que estavam ativos quando ele chegou para terminar, mas não precisa esperar leitores que vieram depois dele

Leitores e escritores [4, 6]

- A solução anterior pode levar à inanição dos escritores
- Para evitar esse problema, é possível alterar o programa para realizar as seguintes ações
  - Quando um leitor chega e um escritor está esperando no semáforo `db`, o leitor é bloqueado atrás do escritor ao invés de ser admitido imediatamente
  - Desse modo, um escritor precisa esperar os leitores que estavam ativos quando ele chegou para terminar, mas não precisa esperar leitores que vieram depois dele
  - Desvantagem: paralelismo (e desempenho) inferior

Leitores e escritores [4, 6]

- A solução anterior pode levar à inanição dos escritores
- Para evitar esse problema, é possível alterar o programa para realizar as seguintes ações
  - Quando um leitor chega e um escritor está esperando no semáforo `db`, o leitor é bloqueado atrás do escritor ao invés de ser admitido imediatamente
  - Desse modo, um escritor precisa esperar os leitores que estavam ativos quando ele chegou para terminar, mas não precisa esperar leitores que vieram depois dele
  - Desvantagem: paralelismo (e desempenho) inferior
  - Existem alternativas mais eficientes [6]

Barbeiro dorminhoco [4]

- A barbearia consiste numa sala de espera com N cadeiras mais a cadeira do barbeiro
- Se não existirem clientes, o barbeiro dorme
- Ao chegar um cliente...
  - se todas as cadeiras estiverem ocupadas, este vai embora

Barbeiro dorminhoco [4]

- A barbearia consiste numa sala de espera com N cadeiras mais a cadeira do barbeiro
- Se não existirem clientes, o barbeiro dorme
- Ao chegar um cliente...
  - se todas as cadeiras estiverem ocupadas, este vai embora
  - se o barbeiro estiver ocupado, mas existirem cadeiras livres, o cliente senta-se e fica esperando sua vez

Barbeiro dorminhoco [4]

- A barbearia consiste numa sala de espera com N cadeiras mais a cadeira do barbeiro
- Se não existirem clientes, o barbeiro dorme
- Ao chegar um cliente...
  - se todas as cadeiras estiverem ocupadas, este vai embora
  - se o barbeiro estiver ocupado, mas existirem cadeiras livres, o cliente senta-se e fica esperando sua vez
  - se o barbeiro estiver dormindo, o cliente o acorda e corta o cabelo

Barbeiro dorminhoco [4]

- A barbearia consiste numa sala de espera com N cadeiras mais a cadeira do barbeiro
- Se não existirem clientes, o barbeiro dorme
- Ao chegar um cliente...
  - se todas as cadeiras estiverem ocupadas, este vai embora
  - se o barbeiro estiver ocupado, mas existirem cadeiras livres, o cliente senta-se e fica esperando sua vez
  - se o barbeiro estiver dormindo, o cliente o acorda e corta o cabelo
- Em uma variação do problema, existem múltiplos barbeiros na barbearia

Barbeiro dorminhoco [4]

Barbeiro dorminhoco [4]

```
#define CHAIRS 5                        /* # chairs for waiting customers */
typedef int semaphore;                  /* use your imagination */
semaphore customers = 0;                /* # of customers waiting for service */
semaphore barbers = 0;                  /* # of barbers waiting for customers */
semaphore mutex = 1;                    /* for mutual exclusion */
int waiting = 0;                        /* customers are waiting (not being cut) */

void barber(void) {
   while (TRUE)
   { down(customers);                   /* go to sleep if # of customers is 0 */
     down(mutex);                       /* acquire access to 'waiting' */
     waiting = waiting - 1;             /* decrement count of waiting customers */
     up(barbers);                       /* one barber is now ready to cut hair */
     up(mutex);                         /* release 'waiting' */
     cut_hair();                        /* cut hair (outside critical region) */
   } }
```

Barbeiro dorminhoco [4]

```
void customer(void) {
   down(mutex);                          /* enter critical region */
   if (waiting < CHAIRS) {               /* if there are no free chairs, leave */
      waiting = waiting + 1;             /* increment count of waiting customers */
      up(customers);                     /* wake up barber if necessary */
      up(mutex);                         /* release access to 'waiting' */
      down(barbers);                     /* go to sleep if # of free barbers is 0 */
      get_haircut();  }                  /* be seated and be serviced */
   else {
     up(mutex); }}                       /* shop is full; do not wait */
```

Jantar dos filósofos [4]

- Considere cinco filósofos $i = 0, 1, \ldots, 4$ que passam a vida a comer e a pensar

Jantar dos filósofos [4]

- Considere cinco filósofos $i = 0, 1, \ldots, 4$ que passam a vida a comer e a pensar
- Eles compartilham uma mesa circular, com um prato de arroz ao centro

Jantar dos filósofos [4]

- Considere cinco filósofos $i = 0, 1, \ldots, 4$ que passam a vida a comer e a pensar
- Eles compartilham uma mesa circular, com um prato de arroz ao centro
- Na mesa existem cinco garfos, colocados um de cada lado do filósofo

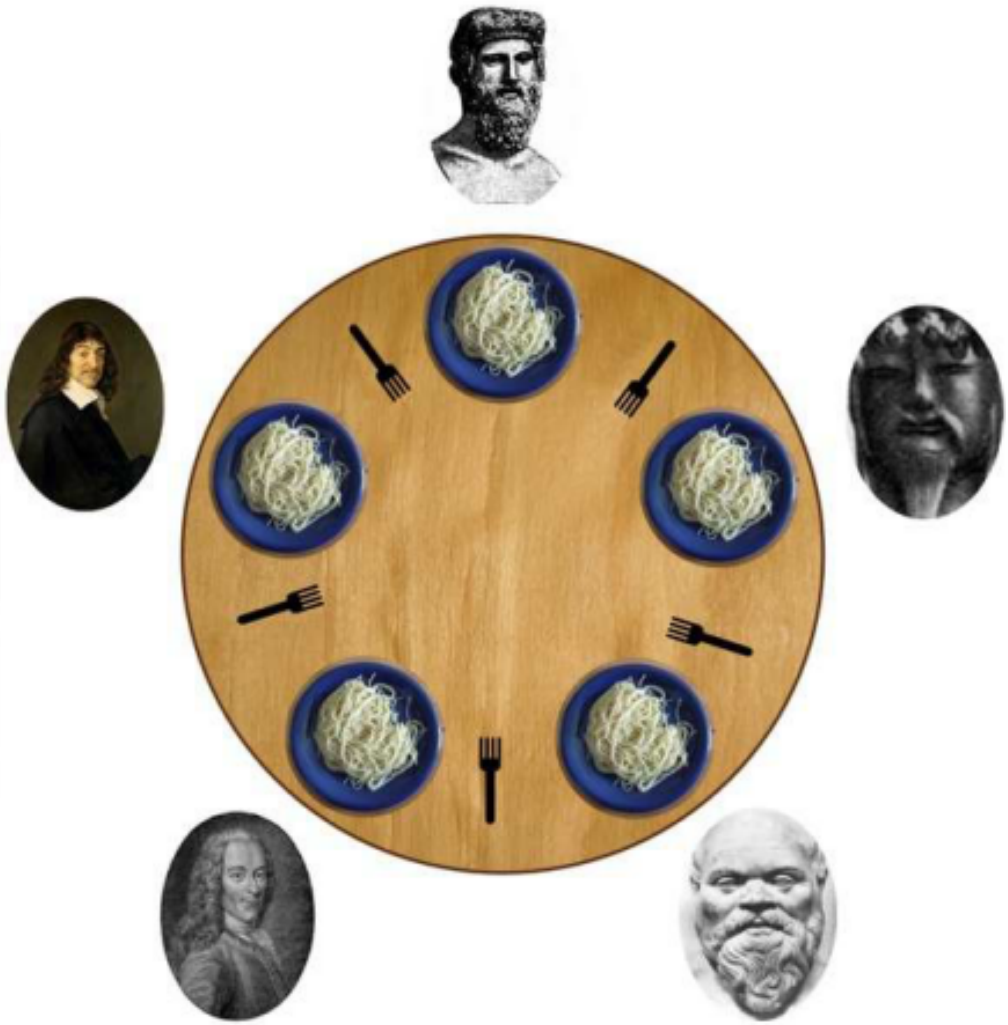
Jantar dos filósofos [4]

- Considere cinco filósofos $i = 0, 1, \ldots, 4$ que passam a vida a comer e a pensar
- Eles compartilham uma mesa circular, com um prato de arroz ao centro
- Na mesa existem cinco garfos, colocados um de cada lado do filósofo
- Quando um filósofo fica com fome, ele pega os dois garfos mais próximos, um de cada vez, e come até ficar saciado

Jantar dos filósofos [4]

- Considere cinco filósofos $i = 0, 1, \ldots, 4$ que passam a vida a comer e a pensar
- Eles compartilham uma mesa circular, com um prato de arroz ao centro
- Na mesa existem cinco garfos, colocados um de cada lado do filósofo
- Quando um filósofo fica com fome, ele pega os dois garfos mais próximos, um de cada vez, e come até ficar saciado
- Quando acaba de comer, ele repousa os garfos e volta a pensar

Jantar dos filósofos [4]



Jantar dos filósofos [4]

- Algumas observações
  - Dado que a mesa é circular, os vizinhos do filósofo i são $(i + N - 1)\%N$ à esquerda e $(i + 1)\%N$ à direita
  - Cada filósofo possuirá um de três estados: pensando (*default*), faminto e comendo
  - Cada filósofo possuirá um semáforo exclusivo para si

Jantar dos filósofos [4]

```
#define N 5                                     /* number of philosophers */
#define LEFT (i+N-1)%N                          /* number of i's left neighbor */
#define RIGHT (i+1)%N                           /* number of i's right neighbor */
#define THINKING 0                              /* philosopher is thinking */
#define HUNGRY 1                                /* philosopher is trying to get forks */
#define EATING 2                                /* philosopher is eating */
typedef int semaphore;                          /* semaphores are a special kind of int */
int state[N];                                   /* array to keep track of everyone's state */
semaphore mutex = 1;                            /* mutual exclusion for critical regions */
semaphore s[N];                                 /* one semaphore per philosopher */

void philosopher(int i)                         /* i: philosopher number, from 0 to N-1 */
   { while (TRUE) {                             /* repeat forever */
      think();                                  /* philosopher is thinking */
      take_forks(i);                            /* acquire two forks or block */
      eat();                                    /* yum-yum, spaghetti */
      put_forks(i); }}                          /* put both forks back on table */
```

Jantar dos filósofos [4]

```
void take_forks(int i)                  /* i: philosopher number, from 0 to N-1 */
     { down(   mutex);                  /* enter critical region */
     state[i] = HUNGRY;                 /* record fact that philosopher i is hungry */
     test(i);                           /* try to acquire 2 forks */
     up(   mutex);                      /* exit critical region */
     down(   s[i]); }                   /* block if forks were not acquired */

void put_forks(i)                       /* i: philosopher number, from 0 to N-1 */
     {down(   mutex);                   /* enter critical region */
      state[i] = THINKING;              /* philosopher has finished eating */
      test(LEFT);                       /* see if left neighbor can now eat */
      test(RIGHT);                      /* see if right neighbor can now eat */
      up(   mutex); }                   /* exit critical region */

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]); }}
```

Considerações finais

Considerações finais

- Nesta aula foram apresentados conceitos de processos em sistemas operacionais

Considerações finais

- Nesta aula foram apresentados conceitos de processos em sistemas operacionais
- Na próxima aula, será tratado o tema de memória

Contato

newtonsp.unioeste@gmail.com

Referências bibliográficas I

[1] Marcelo Johann. Sistemas operacionais, 2009. Notas didáticas.

[2] Sérgio Campos and Marcus Rocha. Sistemas operacionais. http://homepages.dcc.ufmg.br/ scampos/cursos/so/, 2002. Notas didáticas.

[3] Daniel Abdala. Sistemas operacionais, 2016. Notas didáticas.

[4] Roberta Lima Gomes. Sistemas operacionais. http://www.inf.ufes.br/ rgomes/so.htm, 2016. Notas didáticas.

Referências bibliográficas II

[5] A. Silberschatz, P. B. Galvin, and G. Gagne. *Fundamentos de sistemas operacionais*. LTC, 6 edition, 2004.

[6] A. S. Tanenbaum. *Modern Operating Systems*. Pearson education, 3rd edition, 2009.

[7] Tong Lai Yu. Operating systems, 2010. Notas didáticas.

[8] Stephen B. Rainwater. Cosc 3355 animations. http://cs.uttyler.edu/Faculty/Rainwater/COSC3355/Animations/inc 2017.

[9] Tutorialspoint. Operating systems scheduling algorithms. https://www.tutorialspoint.com/operating_system/os_process_scl 2017.

[10] Wikipedia. Heisenbug. https://en.wikipedia.org/wiki/Heisenbug, 2017.