Sistemas Operacionais

Aula 03 - Processos

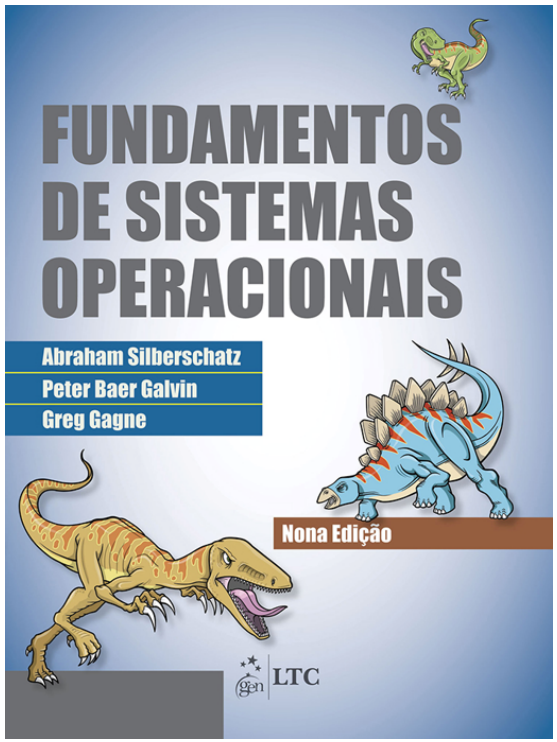
Prof. Samuel Souza Brito

Universidade Federal de Ouro Preto – UFOP
Instituto de Ciências Exatas e Aplicadas – ICEA
Departamento de Computação e Sistemas – DECSI

João Monlevade-MG
2024/2

Referencial Teórico

- O material aqui apresentado é baseado no Capítulo 3 do livro:
  - Silberschatz, A.; Galvin, P. B.; Gagne, G., Fundamentos de Sistemas Operacionais, Editora LTC, 9ª edição, 2015.



Introdução

- Os primeiros computadores permitiam que apenas um programa fosse executado por vez.
  - Total controle do sistema e acesso a todos os recursos do sistema.
- Os computadores atuais permitem que vários programas sejam carregados na memória e executados concorrentemente.
  - Qual(is) técnica(s) permite(m) a execução concorrente dos programas?
- Processos de usuário e processos do SO podem ser executados de forma concorrente.
  - CPU alterna entre processos, tornando o computador mais produtivo.

Conceito de Processo

Conceito de Processo

- O que é um processo?
  - Um programa em execução.
  - Processo *vs* Programa.
    - **Programa = seção de texto** (geralmente, um arquivo executável).
    - Quando um programa vira um processo?

Conceito de Processo

- Processo é uma entidade ativa.
- Alguns componentes:

Conceito de Processo

- Processo é uma entidade ativa.
- Alguns componentes:
  - **Contador de Programa** (Program Counter) e outros registradores.

Conceito de Processo

- Processo é uma entidade ativa.
- Alguns componentes:
  - **Contador de Programa** (Program Counter) e outros registradores.
  - **Pilha** com dados temporários:
    - Parâmetros de método, endereço de retorno, variáveis locais, etc.

Conceito de Processo

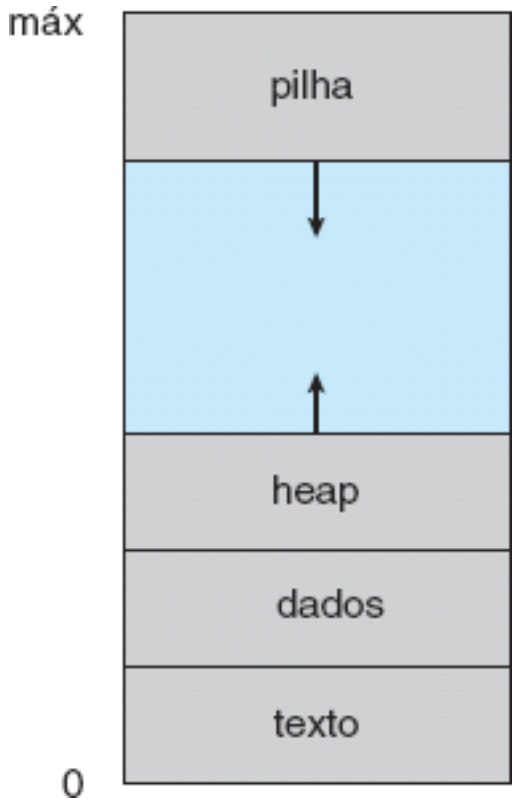
- Processo é uma entidade ativa.
- Alguns componentes:
  - **Contador de Programa** (Program Counter) e outros registradores.
  - **Pilha** com dados temporários:
    - Parâmetros de método, endereço de retorno, variáveis locais, etc.
  - **Seção de dados:**
    - Variáveis globais.

Conceito de Processo

- Processo é uma entidade ativa.
- Alguns componentes:
  - **Contador de Programa** (Program Counter) e outros registradores.
  - **Pilha** com dados temporários:
    - Parâmetros de método, endereço de retorno, variáveis locais, etc.
  - **Seção de dados:**
    - Variáveis globais.
  - **Heap:**
    - Memória alocada dinamicamente.

Conceito de Processo

Processo na memória:



Conceito de Processo

- Dois processos podem estar associados ao mesmo programa.
  - Duas sequências de execução separadas.
  - Exemplos:
    - Execução de várias janelas de um navegador Web.
    - Vários documentos do Word abertos simultaneamente.
    - Dois ou mais terminais (shell, linha de comando) abertos.
  - Seções de texto são equivalentes, mas as seções de dados, heap e pilha variam.
- Também é comum haver um processo que gera muitos outros processos ao ser executado.

Estado do Processo

- O estado de um processo é definido, em parte, pela atividade corrente do processo.
- Um processo pode estar em um dos seguintes estados:

Estado do Processo

- O estado de um processo é definido, em parte, pela atividade corrente do processo.
- Um processo pode estar em um dos seguintes estados:
  - **Novo**: processo está sendo criado.

Estado do Processo

- O estado de um processo é definido, em parte, pela atividade corrente do processo.
- Um processo pode estar em um dos seguintes estados:
  - **Novo**: processo está sendo criado.
  - **Em execução**: instruções estão sendo executadas.

Estado do Processo

- O estado de um processo é definido, em parte, pela atividade corrente do processo.
- Um processo pode estar em um dos seguintes estados:
  - **Novo**: processo está sendo criado.
  - **Em execução**: instruções estão sendo executadas.
  - **Em espera**: processo está esperando que algum evento ocorra (como a conclusão de E/S ou o recebimento de um sinal).

Estado do Processo

- O estado de um processo é definido, em parte, pela atividade corrente do processo.
- Um processo pode estar em um dos seguintes estados:
  - **Novo**: processo está sendo criado.
  - **Em execução**: instruções estão sendo executadas.
  - **Em espera**: processo está esperando que algum evento ocorra (como a conclusão de E/S ou o recebimento de um sinal).
  - **Pronto**: processo está esperando que seja atribuído a um processador.

Estado do Processo

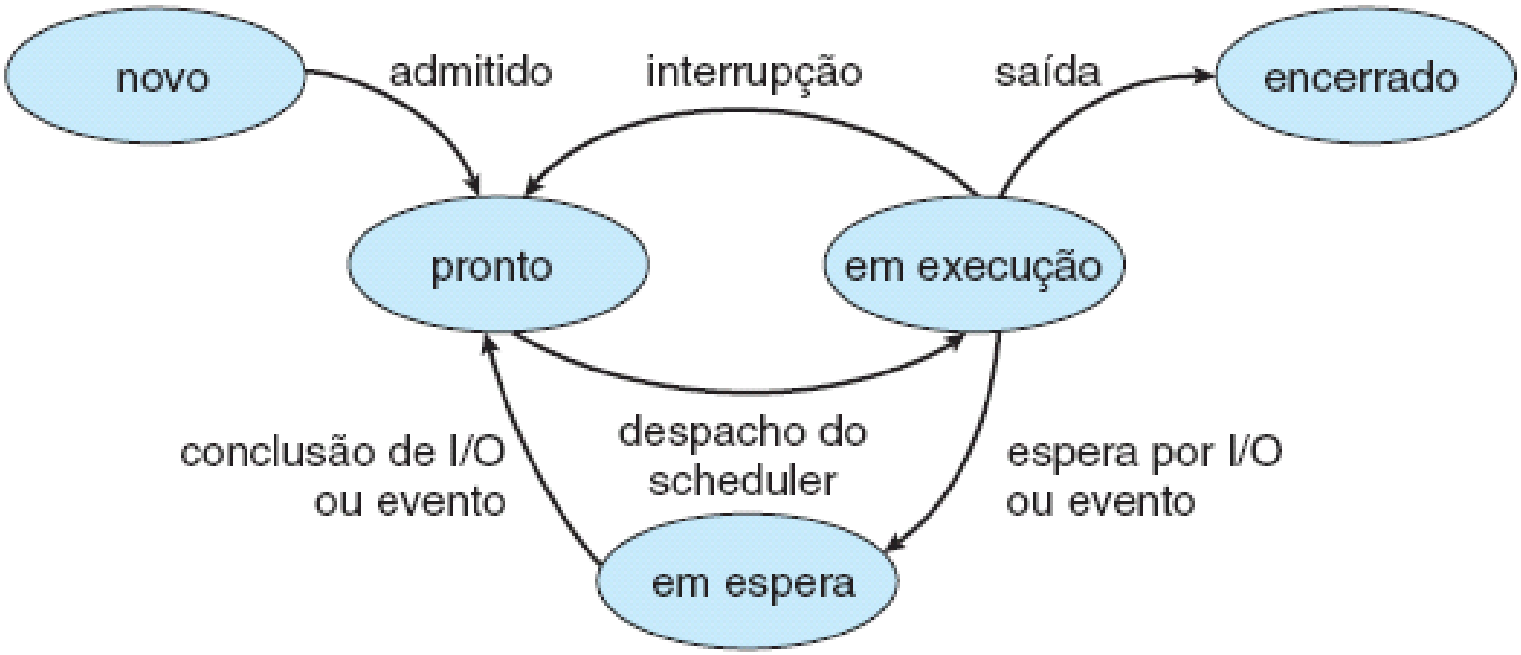
- O estado de um processo é definido, em parte, pela atividade corrente do processo.
- Um processo pode estar em um dos seguintes estados:
  - **Novo**: processo está sendo criado.
  - **Em execução**: instruções estão sendo executadas.
  - **Em espera**: processo está esperando que algum evento ocorra (como a conclusão de E/S ou o recebimento de um sinal).
  - **Pronto**: processo está esperando que seja atribuído a um processador.
  - **Concluído**: processo terminou sua execução.

Estado do Processo

- Nomes dos estados podem variar entre os SOs.
  - Porém, os estados que eles representam são encontrados em todos os sistemas.
- Apenas um processo pode estar *em execução* em algum processador a cada instante.
  - Muitos processos podem estar *prontos* ou *em espera*.

Estado do Processo

Diagrama de estado do processo:



Bloco de Controle de Processo

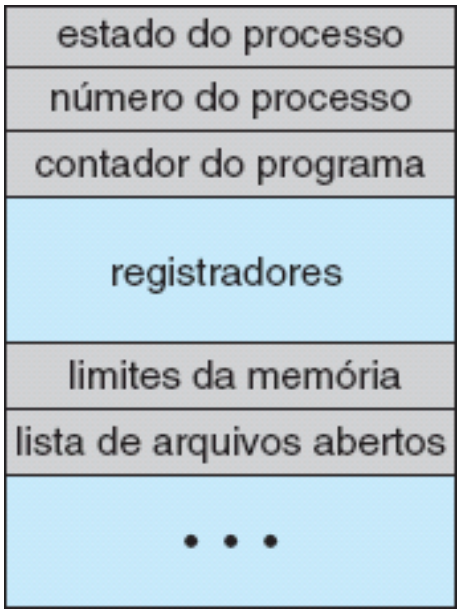
- Cada processo é representado no SO por um **Bloco de Controle de Processo** (*Process Control Block* - PCB).
  - Informações associadas a um processo específico.
- Informações contidas no PCB:
  - **Estado do processo**:
    - Estados vistos no slide anterior.
  - **Contador de programa**:
    - Indica o endereço da próxima instrução a ser executada para esse processo.
  - **Registradores da CPU**:
    - Depende da arquitetura do computador: acumuladores, índices, ponteiros de pilha, etc.
    - Essas informações precisam ser salvas quando uma interrupção ocorre.

Bloco de Controle de Processo

- Informações contidas no PCB (continuação):
  - **Informações de escalonamento de CPU**:
    - Prioridade do processo, ponteiros para filas de escalonamento e outros parâmetros relacionados ao escalonamento.
  - **Informações de gerenciamento de memória**:
    - Informações como o valor dos registradores base e limite e as tabelas de página ou segmento, etc.
  - **Informações de contabilização**:
    - Informações sobre o montante de tempo real e de CPU usados, limites de tempo, números de conta, números de jobs ou processos, e assim por diante.
  - **Informações de status de E/S**:
    - Lista de dispositivos de E/S alocados ao processo, lista de arquivos abertos, etc.

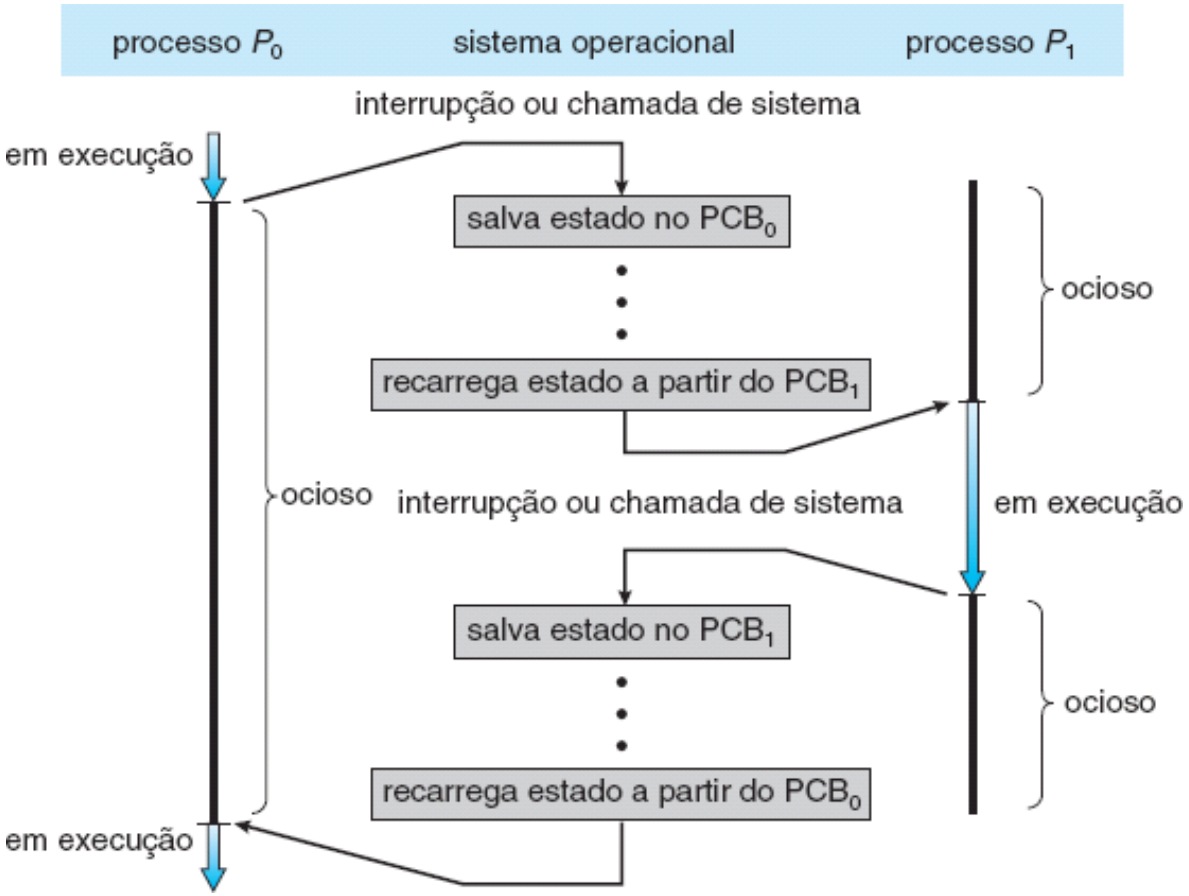
Bloco de Controle de Processo

- PCB serve como um repositório para quaisquer informações que possam variar de um processo para outro.



Bloco de Controle de Processo

Diagrama mostrando a alternância da CPU de um processo para outro:



PCB no Linux

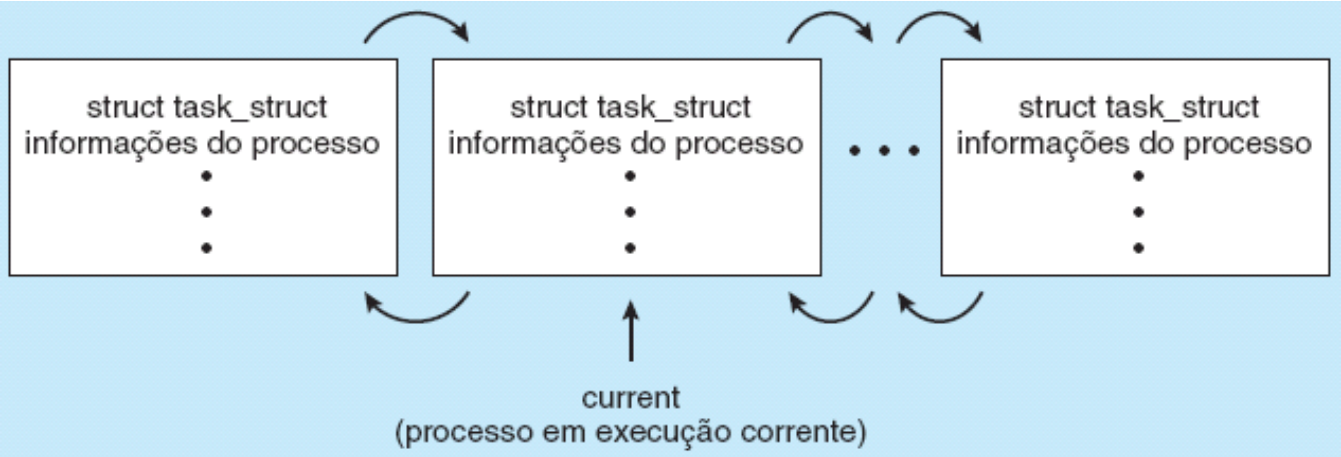
- PCB é representado no Linux pela estrutura em C `task_struct`, que é encontrada no arquivo de inclusão `<linux/sched.h>` no diretório de código-fonte do kernel.
  - Exemplo: `/usr/src/linux-headers-4.15.0-58/include/linux/sched.h`
- Contém todas as informações necessárias à representação de um processo.
  - Estado do processo, informações de escalonamento e de gerenciamento da memória, a lista de arquivos abertos, ponteiros para o pai do processo e uma lista de seus filhos e irmãos.

PCB no Linux

```
long state; /* estado do processo */
struct sched_entity se; /* informações de scheduling */
struct task_struct *parent; /* pai desse processo */
struct list_head children; /* filhos desse processo */
struct files_struct *files; /* lista de arquivos abertos */
struct mm_struct *mm; /* espaço de endereçamento desse processo */
```

PCB no Linux

- Dentro do kernel do Linux, todos os processos ativos são representados com o uso de uma lista duplamente encadeada de `task_struct`.
- O kernel mantém um ponteiro (`current`) para o processo em execução corrente no sistema.



Threads

- O modelo de processo discutido até agora sugere que um processo é um programa que executa apenas um *thread*. Exemplo:
  - Processo executando um programa de processamento de texto.
  - Um único *thread* executando.
  - Uma tarefa de cada vez.
    - O usuário não pode digitar caracteres e executar, simultaneamente, o corretor ortográfico dentro do mesmo processo, por exemplo.

Threads

- SOs modernos estenderam o conceito de processo para permitir que um processo tenha múltiplos *threads* de execução.
  - Desempenhando mais de uma tarefa de cada vez.
  - Benéfico em sistemas *multicore*.
- Em um sistema que suporte *threads*, o PCB é expandido de modo a incluir informações para cada *thread*.
  - Também são necessárias outras alterações no sistema.

Scheduling de Processos

Scheduling de Processos

- Qual é o objetivo da **multiprogramação**?
- Qual é o objetivo do **compartilhamento de tempo**?

Scheduling de Processos

- Qual é o objetivo da **multiprogramação**?
  - Fazer com que a CPU esteja executando algum processo o tempo todo.
- Qual é o objetivo do **compartilhamento de tempo**?

Scheduling de Processos

- Qual é o objetivo da **multiprogramação**?
  - Fazer com que a CPU esteja executando algum processo o tempo todo.
- Qual é o objetivo do **compartilhamento de tempo**?
  - Alternar a CPU entre processos com tanta frequência que os usuários possam interagir com cada programa enquanto ele está sendo executado.

Scheduling de Processos

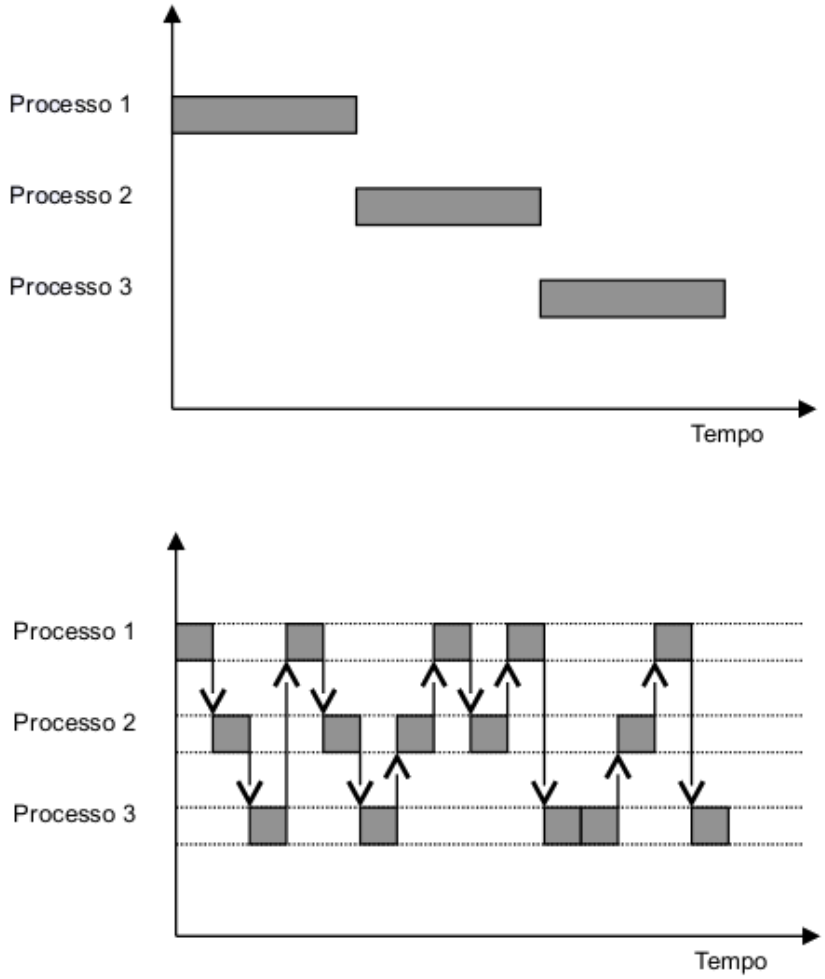
- Qual é o objetivo da **multiprogramação**?
  - Fazer com que a CPU esteja executando algum processo o tempo todo.
- Qual é o objetivo do **compartilhamento de tempo**?
  - Alternar a CPU entre processos com tanta frequência que os usuários possam interagir com cada programa enquanto ele está sendo executado.
- Como atender os dois objetivos?

Scheduling de Processos

- Qual é o objetivo da **multiprogramação**?
  - Fazer com que a CPU esteja executando algum processo o tempo todo.
- Qual é o objetivo do **compartilhamento de tempo**?
  - Alternar a CPU entre processos com tanta frequência que os usuários possam interagir com cada programa enquanto ele está sendo executado.
- Como atender os dois objetivos?
  - **Escalonador de processos** (*process scheduler*).
    - Seleciona um processo disponível (pronto) para a execução na CPU.
    - A partir de um conjunto de processos (fila).

Scheduling de Processos

Sistemas com compartilhamento de tempo desativado e ativado:

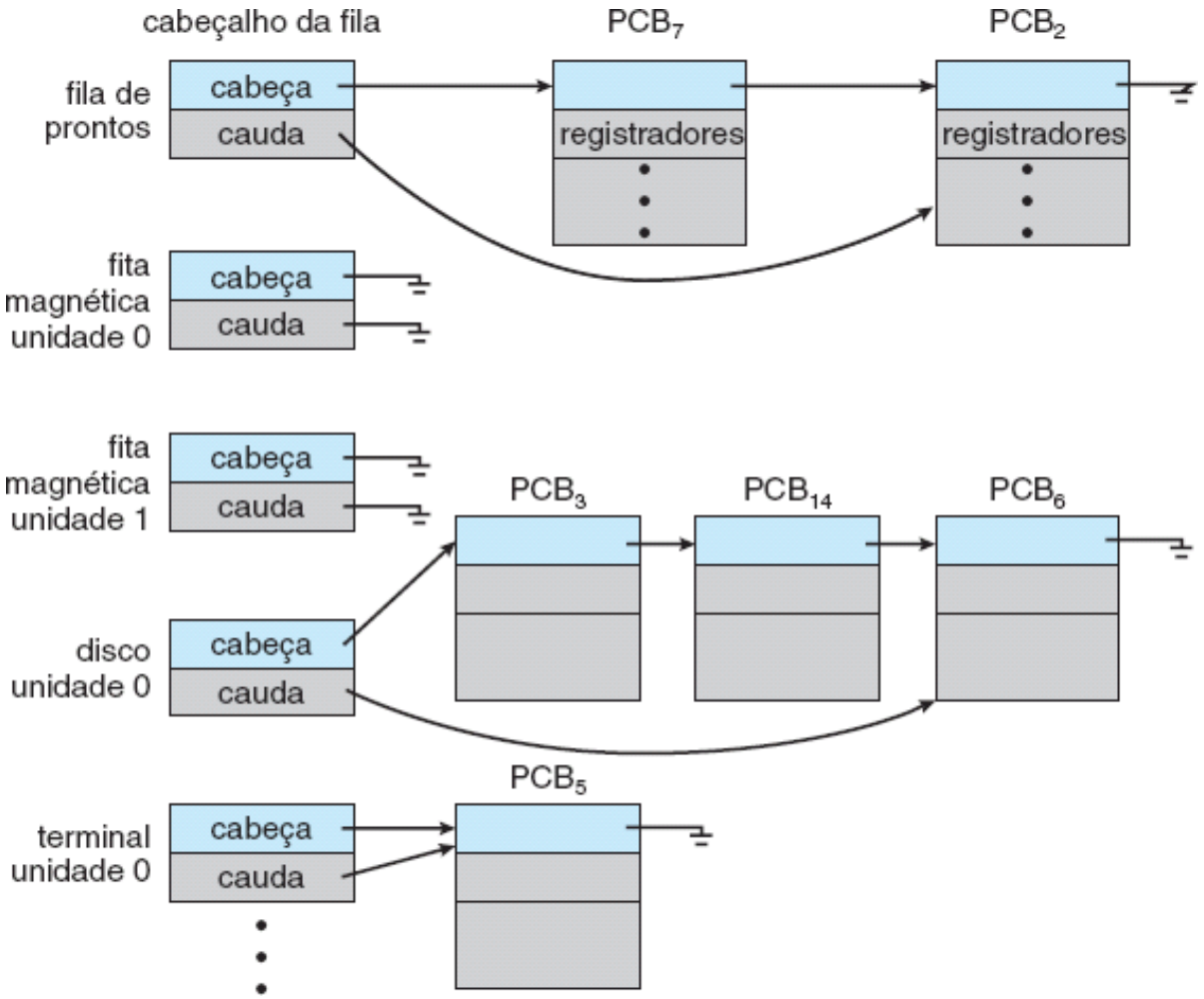


Filas de Scheduling

- **Fila de jobs**:
  - Conjunto de todos os processos do sistema.
- **Fila de prontos**:
  - Todos os processos em memória principal, aguardando apenas pelo escalonamento da CPU.
- **Fila de dispositivo**:
  - Conjunto de processos esperando por um dispositivo de E/S.

Filas de Scheduling

Fila de prontos e várias filas de dispositivo de E/S:



Filas de Scheduling

- Inicialmente, um novo processo é inserido na fila de prontos.
- Quando a CPU é alocada ao processo (estado: em execução), um dos seguintes eventos pode ocorrer:

Filas de Scheduling

- Inicialmente, um novo processo é inserido na fila de prontos.
- Quando a CPU é alocada ao processo (estado: em execução), um dos seguintes eventos pode ocorrer:
  - O processo pode emitir uma solicitação de E/S, sendo inserido em uma fila de dispositivos.

Filas de Scheduling

- Inicialmente, um novo processo é inserido na fila de prontos.
- Quando a CPU é alocada ao processo (estado: em execução), um dos seguintes eventos pode ocorrer:
  - O processo pode emitir uma solicitação de E/S, sendo inserido em uma fila de dispositivos.
  - O processo pode criar um novo processo-filho e esperá-lo terminar.

Filas de Scheduling

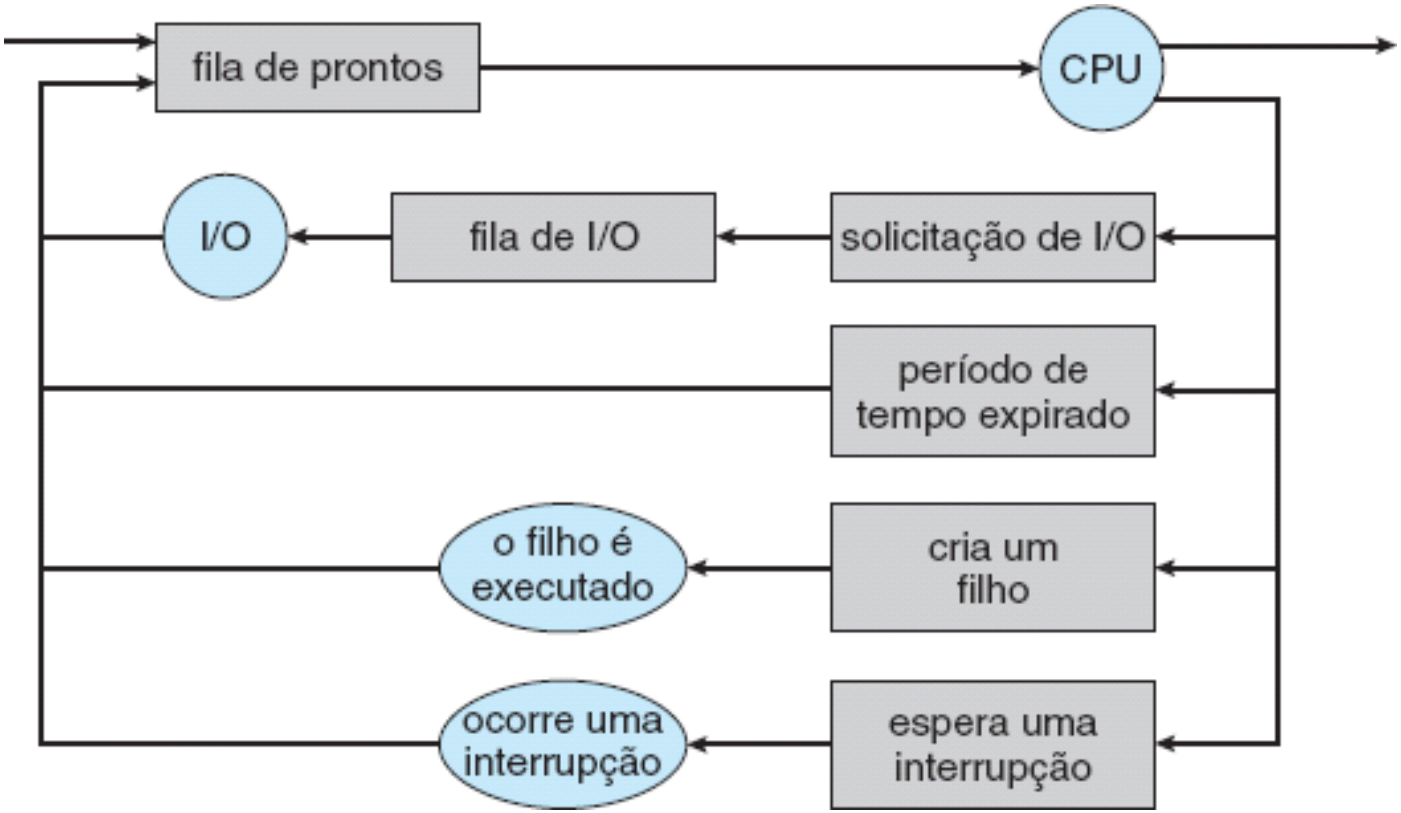
- Inicialmente, um novo processo é inserido na fila de prontos.
- Quando a CPU é alocada ao processo (estado: em execução), um dos seguintes eventos pode ocorrer:
  - O processo pode emitir uma solicitação de E/S, sendo inserido em uma fila de dispositivos.
  - O processo pode criar um novo processo-filho e esperá-lo terminar.
  - O processo pode ser removido à força da CPU, como resultado de uma interrupção, e ser devolvido à fila de prontos.

Filas de Scheduling

- Inicialmente, um novo processo é inserido na fila de prontos.
- Quando a CPU é alocada ao processo (estado: em execução), um dos seguintes eventos pode ocorrer:
  - O processo pode emitir uma solicitação de E/S, sendo inserido em uma fila de dispositivos.
  - O processo pode criar um novo processo-filho e esperá-lo terminar.
  - O processo pode ser removido à força da CPU, como resultado de uma interrupção, e ser devolvido à fila de prontos.
- Quais as mudanças de estado ocorridas nos três eventos anteriores?

Filas de Scheduling

Representação do scheduling de processos em diagrama de enfileiramento:



Escalonadores (*Schedulers*)

- Um processo migra entre as diferentes filas de escalonamento no decorrer do seu tempo de vida.
  - SO precisa selecionar os processos dessas filas.
  - O processo de seleção é executado pelo **escalonador** apropriado.

Tipos de Escalonadores

- **Escalonador de longo prazo**:
  - Escalonador de tarefas ou *scheduler de jobs*.
  - Em um sistema batch, são submetidos mais processos do que é possível executar imediatamente.
    - Esses processos são reservados em um dispositivo de armazenamento de massa, no qual são mantidos para execução posterior.
    - Seleciona processos nesse *pool* e os carrega na memória para execução (enviando-os para a fila de prontos).
  - Define o grau de multiprogramação.
  - Executado com menos frequência.
    - Pode ser lento.

Tipos de Escalonadores

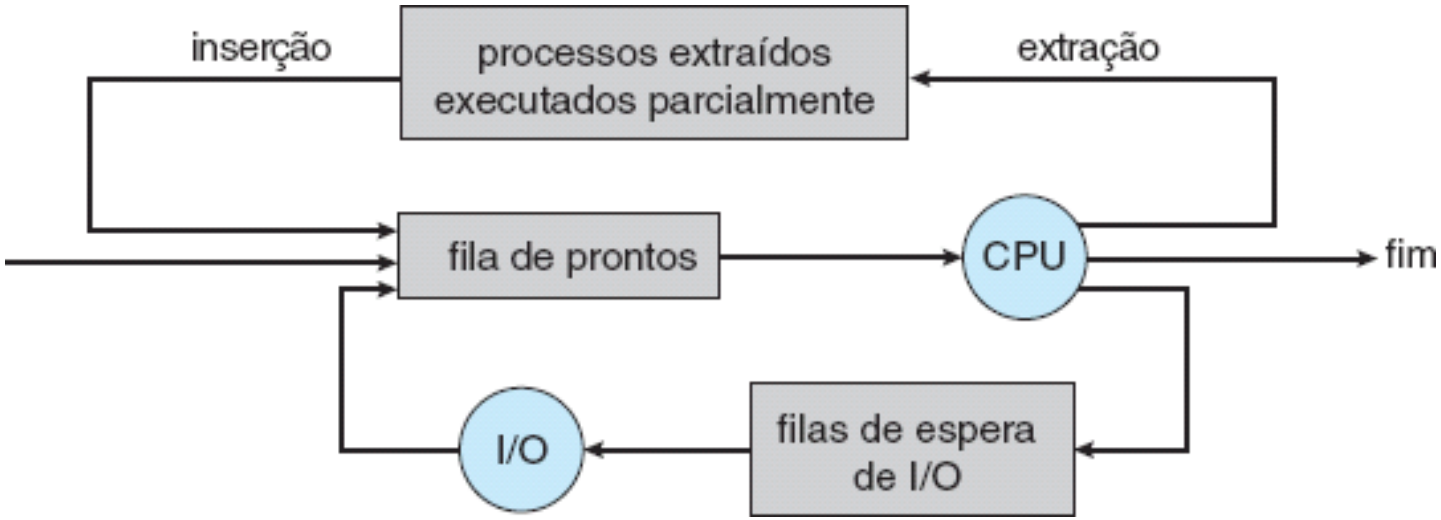
- **Escalonador de curto prazo**:
  - Escalonador de CPU (*scheduler* da CPU).
  - Seleciona entre os processos que estão prontos para execução e aloca a CPU a um deles.
  - Executado com alta frequência.
    - Milissegundos
    - Precisa ser rápido.

Tipos de Escalonadores

- **Escalonador de médio prazo**:
  - Usado em sistemas de tempo compartilhado.
  - Às vezes, pode ser vantajoso remover um processo da memória (e da disputa ativa pela CPU).
    - Reduzir o grau de multiprogramação.
  - Posteriormente, o processo pode ser reintroduzido na memória e sua execução pode ser retomada de onde parou.
  - **Swapping**.
  - Pode ser empregado para melhorar o mix de processos ou quando a memória principal está cheia.

Tipos de Escalonadores

Inclusão do scheduling de médio prazo no diagrama de enfileiramento:



Mudança de Contexto

- Interrupções fazem com que o SO pare a execução de uma tarefa na CPU e execute uma rotina do kernel.
  - Interrupções ocorrem com frequência.
- Quando ocorre uma interrupção, o SO precisa salvar o contexto atual do processo que está em execução na CPU.
  - Mais tarde será necessário restaurar esse contexto.
  - Contexto é representado no PCB do processo.
    - Valor de registradores da CPU.
    - Estado do processo.
    - Informações de gerência de memória.
- Executamos um **salvamento do estado** corrente da CPU e então uma **restauração do estado** para retomar as operações.

Mudança de Contexto

- A alocação da CPU a outro processo requer a execução do salvamento do estado do processo corrente e a restauração do estado de um processo diferente.
  - **Mudança de contexto** ou **troca de contexto**.
  - Kernel salva o contexto do processo antigo em seu PCB e carrega o contexto salvo do novo processo indicado no schedule para execução.

Mudança de Contexto

- Qual a desvantagem da troca de contexto?

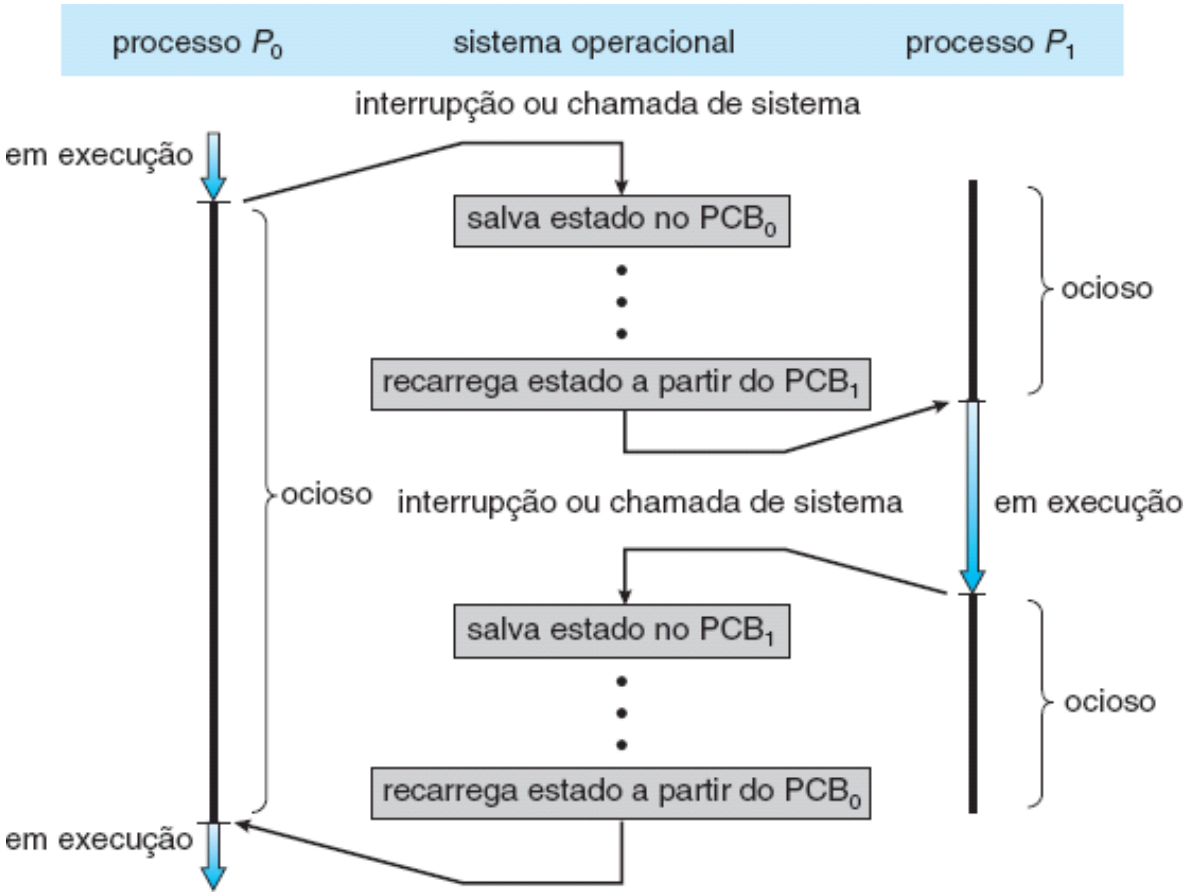
Mudança de Contexto

- Qual a desvantagem da troca de contexto?
  - Tempo de troca de contexto é puramente custo adicional (*overhead*).
  - SO não realiza nenhum trabalho útil durante a troca.

Mudança de Contexto

- Qual a desvantagem da troca de contexto?
  - Tempo de troca de contexto é puramente custo adicional (*overhead*).
  - SO não realiza nenhum trabalho útil durante a troca.
  - Tempo gasto varia de um computador para outro.
    - Velocidade de memória, número de registradores que precisam ser copiados, etc.
  - Tempo também depende do suporte de hardware.
  - Em geral: alguns milissegundos.

Mudança de Contexto



Operações sobre Processos

Operações sobre Processos

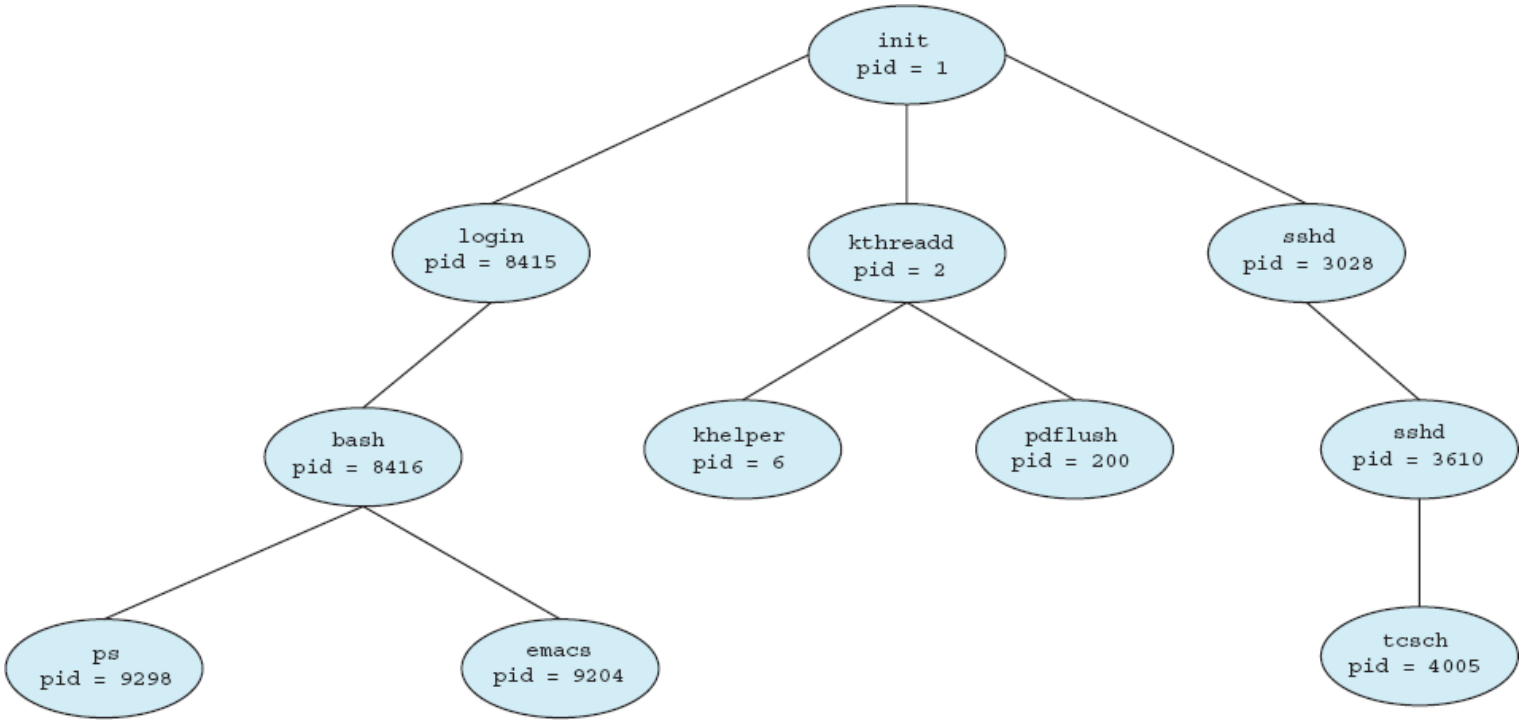
- Processos podem ser executados de forma concorrente.
  - Podem ser criados e removidos dinamicamente.
- É responsabilidade do SO prover um mecanismo para criação e término de processo.

Criação de Processos

- Um processo pode gerar diversos novos processos.
  - Chamada de sistema.
- Cada processo filho pode gerar outros processos.
  - Árvore de processos.
- Como identificar cada processo?
  - Identificador de processo (pid).
    - Identificador exclusivo.
    - Normalmente é um valor inteiro.
    - Pode ser usado como um índice de acesso a vários atributos de um processo dentro do kernel.

Criação de Processos

Uma árvore de processos em um sistema Linux típico:



Criação de Processos

- Um processo precisa de certos recursos para executar sua tarefa.
  - Processo-filho pode obter seus recursos:
    - Diretamente do SO
    - Ou ficar restrito a um subconjunto dos recursos do processo-pai.
- O pai pode ter de dividir seus recursos entre seus filhos ou pode compartilhar alguns recursos entre vários deles.

Criação de Processos

- Processo pai pode passar dados de inicialização (entrada) para processos filhos.
  - Exemplo:
    - Passagem de parâmetros para um programa executado por linha de comando.
- Quando um processo cria um novo processo, existem duas possibilidades de execução:
  1. O pai continua a ser executado concorrentemente com seus filhos.
  2. O pai espera até que alguns de seus filhos ou todos eles sejam encerrados.

Criação de Processos

- Possibilidades de espaço de endereçamento para o novo processo:
  1. O processo-filho é uma duplicata do processo-pai (ele tem o mesmo programa e dados do pai).
  2. O processo-filho tem um novo programa carregado nele.

Criação de Processos no UNIX/Linux

- Cada processo tem seu PID (inteiro exclusivo).
- Um novo processo é criado pela chamada de sistema `fork()`.
  - O novo processo consiste em uma cópia do espaço de endereços do processo original.
    - Esse mecanismo permite que o processo-pai se comunique facilmente com seu processo-filho.
- Os dois processos continuam a execução após o `fork()`, com uma diferença:
  - O código de retorno é zero para o novo processo (filho).
  - O identificador de processo (diferente de zero) do filho é retornado ao pai.
  - **exemplo1.cpp** e **exemplo2.cpp** (disponível no Moodle).

Criação de Processos no UNIX/Linux

- Após uma chamada de sistema `fork()`, um dos dois processos usa, normalmente, a chamada de sistema `exec()` para realocar o espaço de memória do processo para um novo programa.
  - A chamada de sistema `exec()` carrega um arquivo binário na memória e inicia sua execução.
    - Destruindo a imagem de memória do programa que contém a chamada de sistema `exec()`.
  - **exemplo3.cpp** (disponível no Moodle).
- O pai pode criar mais filhos ou, se não tem mais nada a fazer enquanto o filho é executado, pode emitir uma chamada de sistema `wait()` para ser removido da fila de prontos até o encerramento do filho.

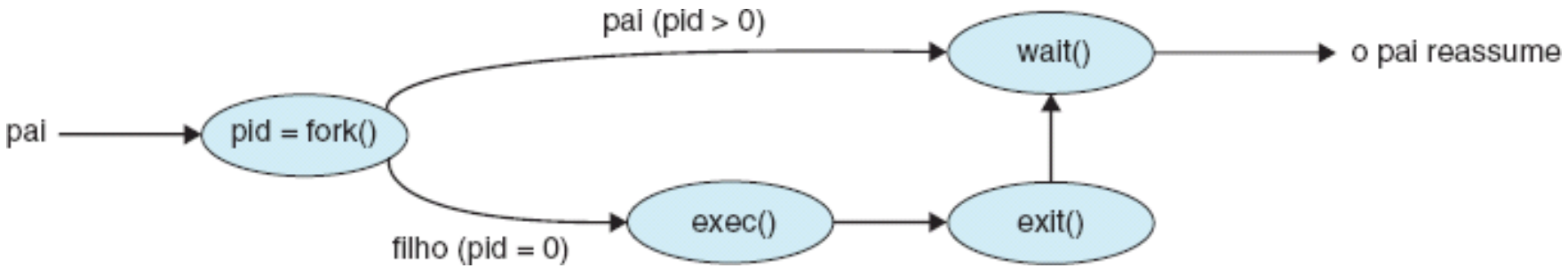
Criação de Processos no UNIX/Linux

Exemplo:

```c
#include <sys/types.h>
#include <stdio.h>
#include <unistd.h>

int main() {
    pid_t pid;

    /* cria um processo-filho */
    pid = fork();

    if (pid < 0) { /* um erro ocorreu */
        fprintf(stderr, "Fork falhou\n");
        return 1;
    } else if (pid == 0) { /* processo-filho */
        execlp("/bin/ls","ls",NULL);
    } else { /* processo-pai */
        /* o pai esperará que o filho seja concluído */
        wait(NULL);
        printf("Processo-filho finalizou\n");
    }

    return 0;
}
```

Criação de Processos no UNIX/Linux

Criação de processo com o uso da chamada de sistema `fork()`:



Criação de Processos no UNIX/Linux

- Para refletir:
  - `fork()` e `exec()` possuem funções semelhantes?

Encerramento de Processos

- Um processo é encerrado quando termina a execução de seu último comando e solicita ao SO que o exclua.
  - Chamada de sistema `exit()`.
  - O processo pode retornar um valor de status (número inteiro) ao seu processo pai.
- Todos os recursos do processo são desalocados pelo SO.
  - Memória física e virtual, arquivos abertos, buffers de E/S, etc.

Encerramento de Processos

- O encerramento também pode ocorrer em outras circunstâncias.
  - Um processo pode causar o encerramento de outro processo por meio de uma chamada de sistema apropriada.
    - Por exemplo, `TerminateProcess()` no Windows.
  - Usualmente, tal chamada de sistema pode ser invocada apenas pelo pai do processo que está para ser encerrado. Por quê?

Encerramento de Processos

- O encerramento também pode ocorrer em outras circunstâncias.
  - Um processo pode causar o encerramento de outro processo por meio de uma chamada de sistema apropriada.
    - Por exemplo, `TerminateProcess()` no Windows.
  - Usualmente, tal chamada de sistema pode ser invocada apenas pelo pai do processo que está para ser encerrado. Por quê?
    - Caso contrário, usuários poderiam encerrar arbitrariamente os jobs uns dos outros.

Encerramento de Processos

- O encerramento também pode ocorrer em outras circunstâncias.
  - Um processo pode causar o encerramento de outro processo por meio de uma chamada de sistema apropriada.
    - Por exemplo, `TerminateProcess()` no Windows.
  - Usualmente, tal chamada de sistema pode ser invocada apenas pelo pai do processo que está para ser encerrado. Por quê?
    - Caso contrário, usuários poderiam encerrar arbitrariamente os jobs uns dos outros.
  - O pai precisa saber as identidades de seus filhos para encerrá-los.
    - Quando um processo cria um novo processo, a identidade do processo recém-criado é passada ao pai.

Encerramento de Processos

- Um pai pode encerrar a execução de um de seus filhos por várias razões, por exemplo:
  - O filho excedeu o uso de alguns dos recursos que recebeu.
  - A tarefa atribuída ao filho não é mais requerida.
  - O pai está sendo encerrado e o SO não permite que um filho continue se seu pai for encerrado.
    - Alguns SOs não permitem que um filho exista se seu pai tiver sido encerrado.
    - **Encerramento em cascata**.

Encerramento de Processos no UNIX/Linux

- Em sistemas Linux e UNIX podemos encerrar um processo usando a chamada de sistema `exit()`.
  - Fornecendo um status de saída como parâmetro.
- No encerramento normal, `exit()` pode ser chamada:
  - Diretamente, conforme apresentado acima.
  - Indiretamente, por um comando `return` na função `main()`.

Encerramento de Processos no UNIX/Linux

- Um processo-pai pode esperar o encerramento de um processo-filho usando a chamada de sistema `wait()`.
  - Recebe um parâmetro que permite que o pai obtenha o status de saída do filho.
  - Retorna o identificador de processo do filho encerrado.
  - **exemplo4.cpp** (disponível no Moodle).

Encerramento de Processos no UNIX/Linux

- **Processo zumbi**:

Encerramento de Processos no UNIX/Linux

- **Processo zumbi**:
  - Quando um processo termina, seus recursos são desalocados pelo SO.
  - Sua entrada na tabela de processos deve permanecer até que o pai chame `wait()`.
    - Porque a tabela de processos contém o status de saída do processo.
  - Um processo que foi encerrado, mas cujo pai ainda não chamou `wait()`.
  - Todos os processos passam para esse estado quando terminam, mas geralmente permanecem como zumbis por pouco tempo.

Encerramento de Processos no UNIX/Linux

- **Processo zumbi**:
  - Quando um processo termina, seus recursos são desalocados pelo SO.
  - Sua entrada na tabela de processos deve permanecer até que o pai chame `wait()`.
    - Porque a tabela de processos contém o status de saída do processo.
  - Um processo que foi encerrado, mas cujo pai ainda não chamou `wait()`.
  - Todos os processos passam para esse estado quando terminam, mas geralmente permanecem como zumbis por pouco tempo.
- **Processo órfão**:
  - Quando o processo-pai é encerrado sem invocar `wait()`.
  - Linux e o UNIX resolvem esse problema designando o processo *init* como o novo pai dos processos órfãos.

Comunicação Interprocessos

Comunicação Interprocessos

- Um processo pode ser:
  - **Independente**:
    - Se não puder afetar ou ser afetado pelos outros processos em execução.
    - Qualquer processo que não compartilhe dados com outros processos.
  - **Cooperativo**:
    - Se puder afetar ou ser afetado por outros processos em execução.
    - Qualquer processo que compartilhe dados com outros processos.

Comunicação Interprocessos

Razões para o fornecimento de um ambiente que permita a cooperação entre processos:

- **Compartilhamento de informações**:
  - Como diversos usuários podem estar interessados na mesma informação (por exemplo, um arquivo compartilhado), temos de prover um ambiente para permitir o acesso concorrente a tais informações.
- **Aumento da velocidade de computação**:
  - Dividir uma tarefa em subtarefas (execução paralela), a fim de que ela seja executada de forma rápida.
  - Quando é possível obter a agilidade na computação de uma tarefa?

Comunicação Interprocessos

Razões para o fornecimento de um ambiente que permita a cooperação entre processos:

- **Compartilhamento de informações**:
  - Como diversos usuários podem estar interessados na mesma informação (por exemplo, um arquivo compartilhado), temos de prover um ambiente para permitir o acesso concorrente a tais informações.
- **Aumento da velocidade de computação**:
  - Dividir uma tarefa em subtarefas (execução paralela), a fim de que ela seja executada de forma rápida.
  - Quando é possível obter a agilidade na computação de uma tarefa?
    - Somente se o computador tiver vários núcleos de processamento.

Comunicação Interprocessos

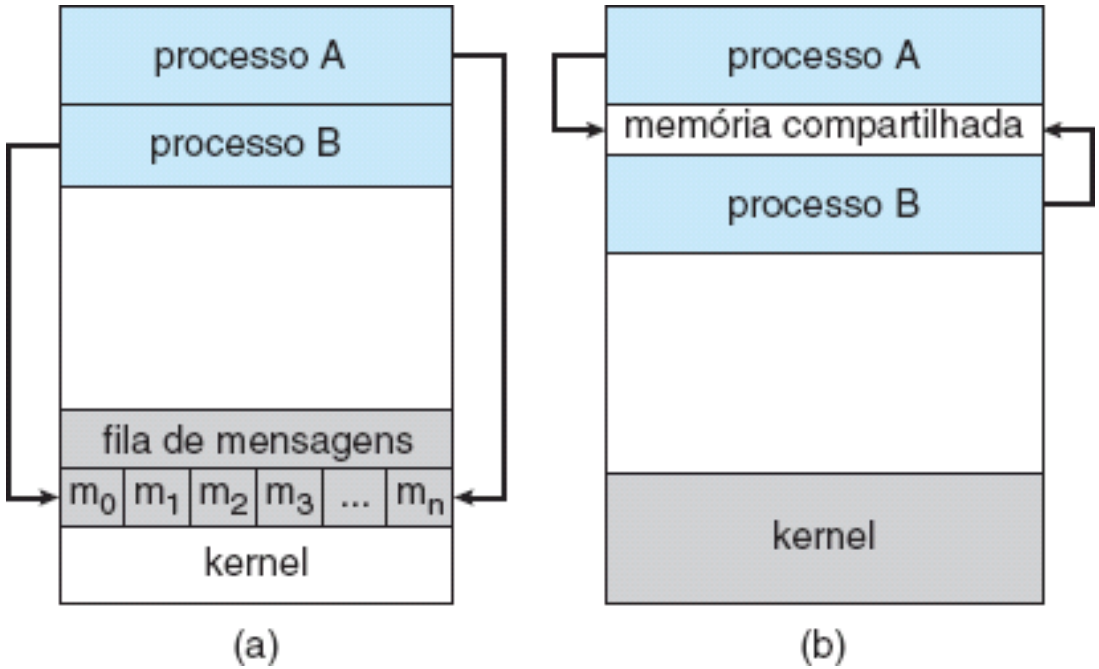
Razões para o fornecimento de um ambiente que permita a cooperação entre processos (continuação):

- **Modularidade**:
  - Construção de um sistema de forma modular, dividindo suas funções em processos ou *threads* separados.
- **Conveniência**:
  - Um usuário individual pode trabalhar em muitas tarefas ao mesmo tempo.

Comunicação Interprocessos

- Processos cooperativos precisam de um mecanismo de **comunicação entre processos** (*Interprocess Communication* - IPC) que lhes permitam trocar dados e informações.
- Dois modelos básicos de comunicação entre processos:
  - **Memória compartilhada**:
    - Estabelece-se uma região da memória que é compartilhada por processos cooperativos.
    - Processos podem trocar informações por meio da leitura e/ou escrita de dados da área compartilhada.
    - Por que é necessário estabelecer essa região? Como isso é feito?
  - **Troca de mensagens**:
    - A comunicação ocorre por meio de mensagens trocadas entre os sistemas cooperativos.

Comunicação Interprocessos



Modelos de comunicação: (a) Troca de mensagens. (b) Memória compartilhada.

Comunicação Interprocessos

- Os dois modelos de comunicação são comuns nos SOs.
  - Muitos SOs implementam ambos.
- **Vantagens da troca de mensagens?**

- **Vantagens da memória compartilhada?**

Comunicação Interprocessos

- Os dois modelos de comunicação são comuns nos SOs.
  - Muitos SOs implementam ambos.
- **Vantagens da troca de mensagens?**
  - Útil para trocar pequenas quantidades de dados.
    - Nenhum conflito precisa ser evitado.
  - Mais fácil de implementar em um sistema distribuído.
- **Vantagens da memória compartilhada?**

Comunicação Interprocessos

- Os dois modelos de comunicação são comuns nos SOs.
  - Muitos SOs implementam ambos.
- **Vantagens da troca de mensagens?**
  - Útil para trocar pequenas quantidades de dados.
    - Nenhum conflito precisa ser evitado.
  - Mais fácil de implementar em um sistema distribuído.
- **Vantagens da memória compartilhada?**
  - Pode ser mais rápida do que a troca de mensagens.
  - Menos chamadas de sistema para efetuar comunicação.
    - Somente para estabelecer as regiões compartilhadas na memória.
  - Todos os acessos são tratados como acessos à memória.

Sistemas de Memória Compartilhada

- A comunicação entre processos usando memória compartilhada requer que os processos estabeleçam uma região de memória compartilhada.
  - Normalmente, a região de memória compartilhada reside no espaço de endereçamento do processo que cria o segmento de memória compartilhada.
  - Outros processos que queiram se comunicar usando esse segmento de memória compartilhada devem anexá-lo ao seu espaço de endereçamento.

Sistemas de Memória Compartilhada

- SO tenta impedir que um processo acesse a memória de outro processo.
  - Memória compartilhada exige que dois ou mais processos concordem em remover essa restrição.
- Formato de dados e local são determinados pelos processos cooperativos.
  - Não estão sob controle do SO.
  - Processos são responsáveis por garantir que não estarão escrevendo no mesmo local simultaneamente.

Sistemas de Memória Compartilhada

- Para ilustrar o conceito de processos cooperativos, considere o **problema do produtor-consumidor**.
  - Paradigma comum dos processos cooperativos.
- Um processo **produtor** produz informações consumidas por um processo **consumidor**.
- Exemplos:
  - Compilador pode produzir código *assembly* que é consumido por um montador (*assembler*). O montador produz módulos objeto, que são consumidos pelo carregador.
  - Arquitetura cliente-servidor. Servidor é um produtor e clientes são consumidores. Servidor Web fornece arquivos HTML e imagens que são lidos pelo navegador Web do cliente.

Memória Compartilhada: Produtor-Consumidor

- Uma solução para o problema do produtor-consumidor usando memória compartilhada:
  - Criação de um **buffer de itens** que possa ser preenchido pelo produtor e esvaziado pelo consumidor.
  - Buffer deve residir em uma região de memória compartilhada por processos produtores e consumidores.
  - Produtor pode produzir um item enquanto o consumidor está consumindo outro item. Problema?

Memória Compartilhada: Produtor-Consumidor

- Uma solução para o problema do produtor-consumidor usando memória compartilhada:
  - Criação de um **buffer de itens** que possa ser preenchido pelo produtor e esvaziado pelo consumidor.
  - Buffer deve residir em uma região de memória compartilhada por processos produtores e consumidores.
  - Produtor pode produzir um item enquanto o consumidor está consumindo outro item. Problema?
    - Necessidade de sincronização.
    - Consumidor não pode tentar consumir um item que ainda não foi produzido.

Memória Compartilhada: Produtor-Consumidor

- Dois tipos de buffer podem ser usados:
  - **Buffer ilimitado**:
    - Não coloca um limite prático no tamanho do buffer.
    - Consumidor precisa esperar se o buffer estiver vazio.
    - Produtor sempre pode produzir novos itens.
  - **Buffer limitado**:
    - Tamanho do buffer é fixo.
    - Consumidor precisa esperar se o buffer estiver vazio.
    - Produtor precisa esperar se o buffer estiver cheio.
- Trabalharemos com o buffer limitado para ilustrar a comunicação entre processos usando memória compartilhada.

Memória Compartilhada: Produtor-Consumidor

Interface para implementações do buffer de itens:

```
public interface Buffer {
    // produtores executam este método
    public abstract void insert(Object item);

    // consumidores executam este método
    public abstract Object remove();
}
```

Memória Compartilhada: Produtor-Consumidor

```
public class BoundedBuffer implements Buffer {
    private static final int BUFFER_SIZE = 5;
    private int count; // número de itens contidos no buffer
    private int in;  // aponta para a próxima posição livre
    private int out; // aponta para a primeira posição preenchida
    private Object[] buffer;

    public BoundedBuffer() {
        // criando um buffer vazio
        count = 0;
        in = 0;
        out = 0;
        buffer = new Object[BUFFER_SIZE];
    }

    // produtores executam este método
    public void insert(Object item) {
        ...
    }

    // consumidores executam este método
    public Object remove() {
        ...
    }
}
```

Memória Compartilhada: Produtor-Consumidor

```
// produtores executam este método
public void insert(Object item) {
    while(count == BUFFER_SIZE); // buffer cheio

    // acrescenta um item ao buffer
    count++;
    buffer[in] = item;
    in = (in + 1) % BUFFER_SIZE;
}
```

Memória Compartilhada: Produtor-Consumidor

```
// consumidores executam este método
public Object remove() {
    while(count == 0); // buffer vazio

    // remove um item do buffer
    count--;
    Object item = buffer[out];
    out = (out + 1) % BUFFER_SIZE;

    return item; // retorna o item removido
}
```

Memória Compartilhada: Produtor-Consumidor

- Processo produtor chama o método `insert()` quando deseja inserir um item no buffer.
- Processo consumidor chama o método `remove()` quando quer consumir um item do buffer.
- Produtor e consumidor serão bloqueados quando o buffer não puder ser utilizado por eles.
- A solução apresentada não aborda a situação em que tanto produtor quanto consumidor tentam acessar o buffer compartilhado concorrentemente.
  - Discutiremos soluções para esse problema nas aulas de sincronização de processos.

Sistemas de Troca de Mensagens

- SO fornece meios para que os processos cooperativos se comuniquem uns com os outros por meio de um recurso de transmissão de mensagens.
- A transmissão de mensagens fornece um mecanismo para permitir que os processos se comuniquem e sincronizem suas ações sem compartilhar o mesmo espaço de endereçamento.
  - Particularmente útil em um ambiente distribuído em que os processos em comunicação podem residir em diferentes computadores conectados por uma rede.
- Exemplo: um aplicativo de bate-papo na Internet.

Sistemas de Troca de Mensagens

- Um sistema de troca de mensagens fornece pelo menos duas operações básicas:
  - `send(message)`
    - Enviar uma mensagem.
  - `receive(message)`
    - Receber uma mensagem.
- Mensagens podem ter um tamanho fixo ou variável.
  - Tamanho fixo torna a implementação no nível do sistema mais simples e a programação mais difícil.
  - Mensagens de tamanho variável requerem uma implementação mais complexa no nível do sistema, mas a tarefa de programar torna-se mais simples.

Sistemas de Troca de Mensagens

- Se os processos P e Q querem se comunicar, eles devem enviar e receber mensagens entre si.
  - Um **link de comunicação** deve existir entre eles.
  - Esse link pode ser implementado de várias maneiras.
- Métodos para implementar logicamente um link e as operações `send()` e `receive()`:
  - Comunicação direta ou indireta
  - Comunicação síncrona ou assíncrona
  - Armazenamento em buffer automático ou explícito

Nomeação

- Processos que querem se comunicar precisam contar com uma forma para referenciar um ao outro.
- **Comunicação direta**:
  - Cada processo que quer se comunicar deve nomear explicitamente o receptor ou o emissor da comunicação.
    - `send(P, message)` envia uma mensagem ao processo P.
    - `receive(Q, message)` recebe uma mensagem do processo Q.
  - Um link de comunicação nesse esquema tem as seguintes propriedades:
    - Um link é estabelecido automaticamente entre cada par de processos que querem se comunicar. Os processos precisam saber apenas a identidade um do outro para se comunicar.
    - Um link é associado a exatamente dois processos.
    - Entre cada par de processos, existe exatamente um link.
  - Uma variante desse esquema emprega a assimetria no endereçamento.

Nomeação

- **Comunicação indireta**:
  - Mensagens são enviadas para e recebidas de **caixas postais**, ou **portas**.
  - Uma caixa postal pode ser considerada abstratamente como um objeto no qual mensagens podem ser inseridas por processos e do qual mensagens podem ser removidas.
  - Cada caixa postal tem uma identificação exclusiva.
  - Um processo pode se comunicar com outro processo por meio de várias caixas postais diferentes, mas dois processos só podem se comunicar se tiverem uma caixa postal compartilhada.
  - `send(A, message)`: envia uma mensagem para a caixa postal A.
  - `receive(A, message)`: recebe uma mensagem da caixa postal A.

Nomeação

- **Comunicação indireta** (continuação):
  - Nesse esquema, o link de comunicação tem as seguintes propriedades:
    - Um link é estabelecido entre um par de processos apenas se os dois membros do par possuem uma caixa postal compartilhada.
    - Um link pode estar associado a mais de dois processos.
    - Entre cada par de processos em comunicação, podem existir vários links diferentes, com cada link correspondendo a uma caixa postal.

Sincronização

- A transmissão de mensagens pode ser **com bloqueio** ou **sem bloqueio** (síncrona e assíncrona):

Sincronização

- A transmissão de mensagens pode ser **com bloqueio** ou **sem bloqueio** (síncrona e assíncrona):
  - **Envio com bloqueio**: o processo emissor é bloqueado até que a mensagem seja recebida pelo processo receptor ou pela caixa postal.

Sincronização

- A transmissão de mensagens pode ser **com bloqueio** ou **sem bloqueio** (síncrona e assíncrona):
  - **Envio com bloqueio**: o processo emissor é bloqueado até que a mensagem seja recebida pelo processo receptor ou pela caixa postal.
  - **Envio sem bloqueio**: o processo emissor envia a mensagem e retoma a operação.

Sincronização

- A transmissão de mensagens pode ser **com bloqueio** ou **sem bloqueio** (síncrona e assíncrona):
  - **Envio com bloqueio**: o processo emissor é bloqueado até que a mensagem seja recebida pelo processo receptor ou pela caixa postal.
  - **Envio sem bloqueio**: o processo emissor envia a mensagem e retoma a operação.
  - **Recebimento com bloqueio**: o receptor é bloqueado até que a mensagem fique disponível.

Sincronização

- A transmissão de mensagens pode ser **com bloqueio** ou **sem bloqueio** (síncrona e assíncrona):
  - **Envio com bloqueio**: o processo emissor é bloqueado até que a mensagem seja recebida pelo processo receptor ou pela caixa postal.
  - **Envio sem bloqueio**: o processo emissor envia a mensagem e retoma a operação.
  - **Recebimento com bloqueio**: o receptor é bloqueado até que a mensagem fique disponível.
  - **Recebimento sem bloqueio**: o receptor recupera uma mensagem válida ou uma mensagem nula.

Sincronização

- A solução para o problema do produtor-consumidor torna-se trivial quando usamos comandos `send()` e `receive()` com bloqueio.
  - Produtor invoca a chamada `send()` com bloqueio e espera até que a mensagem seja distribuída para o receptor ou a caixa postal.
  - Quando o consumidor invoca `receive()`, ele é bloqueado até que uma mensagem esteja disponível.
- Qual a desvantagem dessa solução?

Armazenamento em Buffer

- Independentemente de a comunicação ser direta ou indireta, as mensagens trocadas por processos em comunicação residem em uma fila temporária.
- Essas filas podem ser implementadas de três maneiras:
  - **Capacidade zero**:
    - A fila tem tamanho máximo de zero.
    - Link não pode ter quaisquer mensagens em espera.
    - Emissor deve ser bloqueado até que o receptor receba a mensagem.

Armazenamento em Buffer

- Essas filas podem ser implementadas de três maneiras (continuação):
  - **Capacidade limitada**:
    - A fila tem tamanho finito n.
    - No máximo n mensagens podem residir nela.
    - Se a fila não está cheia quando uma nova mensagem é enviada, a mensagem é inserida na fila e o emissor pode continuar a execução sem esperar.
    - Se a fila está cheia, o emissor deve ser bloqueado até haver espaço disponível na fila.
  - **Capacidade ilimitada**:
    - O tamanho da fila é potencialmente infinito.
    - Pode conter qualquer número de mensagens em espera.
    - O emissor nunca é bloqueado.

Troca de Mensagens: Produtor-Consumidor

```java
public interface Channel {
    // Envia uma mensagem ao canal
    public abstract void send(Object item);

    // Recebe uma mensagem do canal
    public abstract Object receive();
}
```

Troca de Mensagens: Produtor-Consumidor

```java
public class MessageQueue implements Channel {
    private Vector<Object> queue;

    public MessageQueue() {
        // Criando uma fila de mensagens ilimitada
        queue = new Vector<Object>();
    }

    // Implementa um send sem bloqueio
    public void send(Object item) {
        queue.addElement(item);
    }

    // Implementa um receive sem bloqueio
    public Object receive() {
        if (queue.size() == 0) {
            return null;
        } else {
            return queue.remove(0);
        }
    }
}
```

Troca de Mensagens: Produtor-Consumidor

- Produtor e consumidor compartilham:
  - `MessageQueue mailbox;`
- Código do produtor:

```
while (true) {
    Date message = new Date();
    mailbox.send(message);
}
```

- Código do consumidor:

```
while (true) {
    Date message = mailbox.receive();

    if (message != null) {
        // consome a mensagem
    }
}
```

Dúvidas?