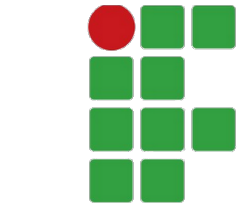
Sistemas Operacionais

Processos e Threads

Felipe Augusto Lima Reis

felipe.reis@ifmg.edu.br



INSTITUTO FEDERAL
Minas Gerais

Sumário

Conceitos

Processos

- "Processo é basicamente um programa em execução" [Tanenbaum and Bos, 2014];
  - "O programa é uma entidade passiva e por si só não é um processo" [Silberschatz et al., 2012]
  - O programa somente se torna um processo quando é executado;
  - Um programa pode ter zero, um ou mais processos em execução[1].
- O processo é também definido como uma unidade de trabalho em sistemas de tempo compartilhado[2] [Silberschatz et al., 2012].

[1] "Processo em execução" é redundante. O termo foi utilizado somente para facilitar a compreensão.

[2] Sistemas de tempo compartilhado são aqueles que dividem os recursos computacionais entre processos.

Tipo de Processos

- Sistemas operacionais consistem em coleções de processos:
  - Processos de sistema: executam código do sistema operacional;
  - Processos de usuário: executam código do usuário (aplicações);
- Processos de usuário e de sistema podem ser executados concorrentemente [Silberschatz et al., 2012].

Nomenclaturas

- Atividades de CPUs podem receber diversas nomenclaturas:
  - Jobs: atividades executadas em sistemas de *batch* (*batch systems*);
  - Tarefas[3]: atividades executadas em sistemas de tempo compartilhado;
- Genericamente, essas atividades são denominadas processos
  - No entanto, na prática, as palavras "job", "task" (tarefa) e processo são utilizadas intercambiavelmente;
  - Recomenda-se, entretanto, o uso da palavra processo [Silberschatz et al., 2012].

[3] Também denominados *tasks* ou programas de usuários.

Na literatura, por questões históricas, a expressão "job" é utilizada no lugar de processo. Quando algumas teorias de sistemas operacionais foram desenvolvidas, a maior parte das atividades eram relativas à execução de "jobs".

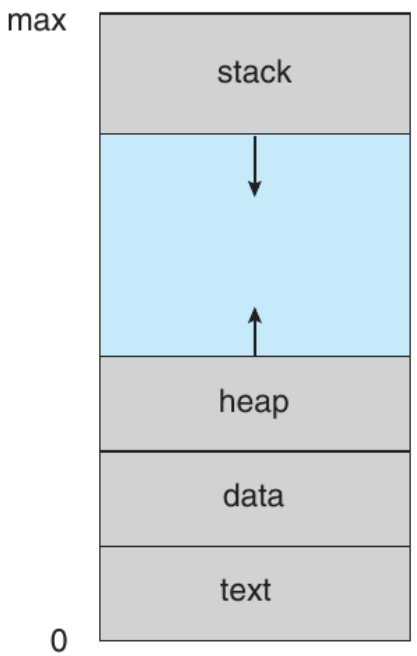
Organização em Memória

INSTITUTO FEDERAL
Minas Gerais

- Um processo em memória pode ser subdividido nas seguintes partes:
  - Text section: contém o código efetivo do programa;
    - Contém o contador do programa (*program counter*) e o conteúdo dos registradores do processador;
  - Data section: seção dos dados, contém as variáveis globais;
  - Pilha: contém os dados temporários (parâmetros de função, variáveis locais, endereços de retorno, etc);
  - Heap: contém os dados alocados dinamicamente (via estruturas de dados como listas, filas, pilhas, etc) [Silberschatz et al., 2012].

Organização em Memória

- Na memória, um processo pode ser visto da seguinte forma:



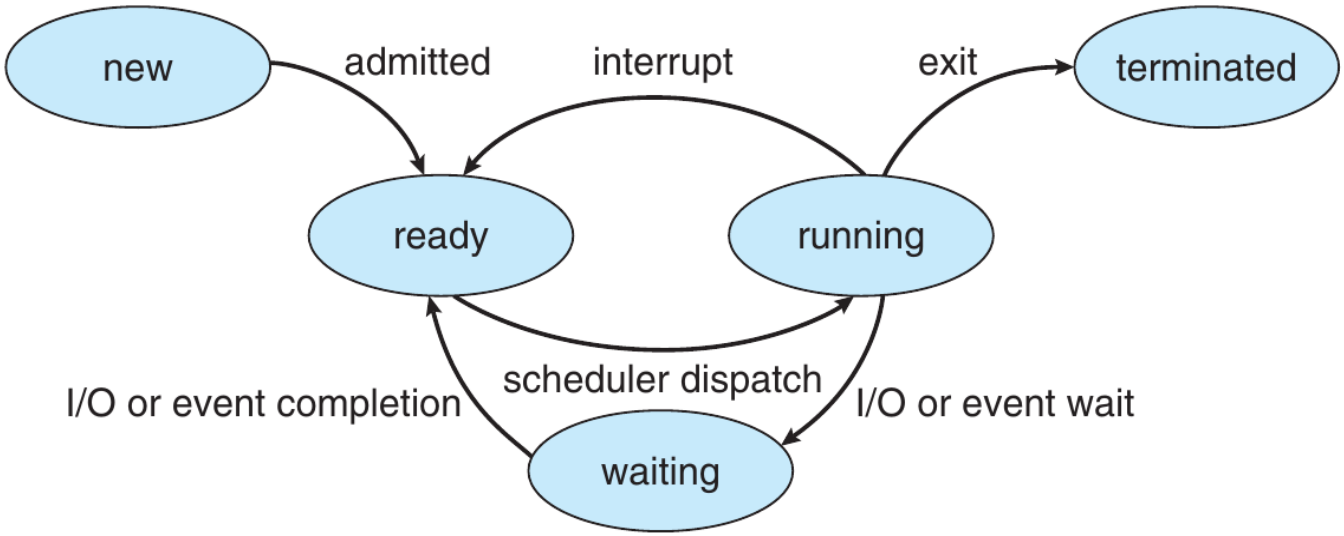
Organização de um processo na memória.
Fonte: [Silberschatz et al., 2012]

Estados dos Processos

- Os processos possuem os seguintes estados, correspondentes a sua situação atual:
  - Novo: processo em criação;
  - Em execução: processo possui instruções em execução;
  - Aguardando: processo está esperando um evento ou um recurso para continuar sua execução;
  - Pronto: processo aguarda ser atribuído a um processador;
  - Finalizado: processo terminou a execução de suas instruções [Silberschatz et al., 2012].

Estados dos Processos

- A relação entre estados de um processo pode ser vista na figura abaixo:



Relação entre estados de um processo.
Fonte: [Silberschatz et al., 2012]

Operações

Criação de Processos

- Processos podem criar inúmeros processos filhos durante sua execução
  - Esses novos processos formam uma árvore de processos;
  - Cada processo é identificado por um número único, denominado PID;
- De forma geral, todos os processos em execução são originados de um processo inicial, denominado *init*
  - Esse processo criado durante a rotina de *boot* inicializa outros processos (recursos do sistema);

Criação de Processos

- Quando um processos filhos são criados, temos as seguintes possibilidades:
  - Em relação à forma de execução:
    1. Pai e filhos continuam a executar simultaneamente;
    2. O pai espera até que alguns ou todos os seus filhos tenham terminado;
  - Em relação ao espaço de endereçamento:
    1. O processo filho é uma cópia do processo pai (tem o mesmo programa e dados);
    2. O processo filho possui um novo programa carregado nele [Silberschatz et al., 2012].

Término de Processos

- Ao finalizar a execução, o processo solicita ao SO que o exclua
  - Em sistemas POSIX é utilizada a chamada `exit()`, enquanto no Windows é utilizada a chamada `TerminateProcess()`[4];
  - Ao encerrar, o processo pode retornar um código para o processo pai[5];
- Processos também podem ser encerrados pelo processo pai
  - Isso ocorre quando o processo filho não é mais necessário, utilizou uma quantidade de recursos superiores à esperada ou quando o processo pai irá finalizar;
  - Alguns SOs implementam encerramento em cascata, para que não existam processos filhos orfãos [Silberschatz et al., 2012].

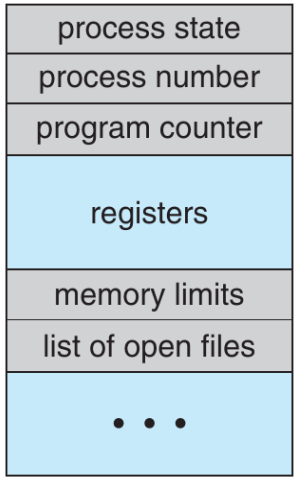
[4] Conforme informado em aulas anteriores, as linguagens podem encapsular tais comandos.

[5] Em C/C++ podem ser retornados valores inteiros na rotina principal. Tal comando pode indicar que o processo foi finalizado adequadamente (0) ou mediante erro (códigos diferentes de zero).

Troca de Contexto (Context Switching)

Process Control Block (PCB)

- O Bloco de Controle de Processos (*Process Control Block*) armazena pedaços de informações relacionados ao processo;
- O PCB é composto por informações importantes como estado do processo, contador do programa, registradores, informações de uso de memória e I/O, etc.



Process Control Block.
Fonte: [Silberschatz et al., 2012]

Interrupções

- Interrupção é um sinal que faz com que sistema operacional interrompa a tarefa atual e inicie a execução de uma rotina de *kernel* [Silberschatz et al., 2012];
- Interrupções podem ser divididas em 3 tipos:
  - Interrupção de Hardware: geradas por dispositivos externos (I/O), que desejam atenção do SO;
  - Interrupção de Software: programas que desejam realizar chamadas de sistema, monitoradas pelo SO;
  - Traps: correspondem a exceções ou falhas, geradas pela CPU, que requerem mudança para o modo kernel, para gerenciamento da condição excepcional[6] [Bower, 2015].

[6] Ex.: Exceção devido à divisão de um número por zero (*DivideByZeroException*).

Troca de Contexto

- Quando ocorre uma interrupção, o SO executa a operação de Troca de Contexto, correspondente às seguintes etapas:
  1. Armazenamento do contexto do processo corrente[7];
  2. Suspensão do processo atual;
  3. Restauração (ou carregamento) do novo processo;
  4. Execução do novo processo;
  5. Reinício do processo original, recuperando o contexto salvo[8];
- O contexto de um processo é definido pelo PCB [Silberschatz et al., 2012].

[7] Esta etapa é denominada *state save* ou "salvar estado".

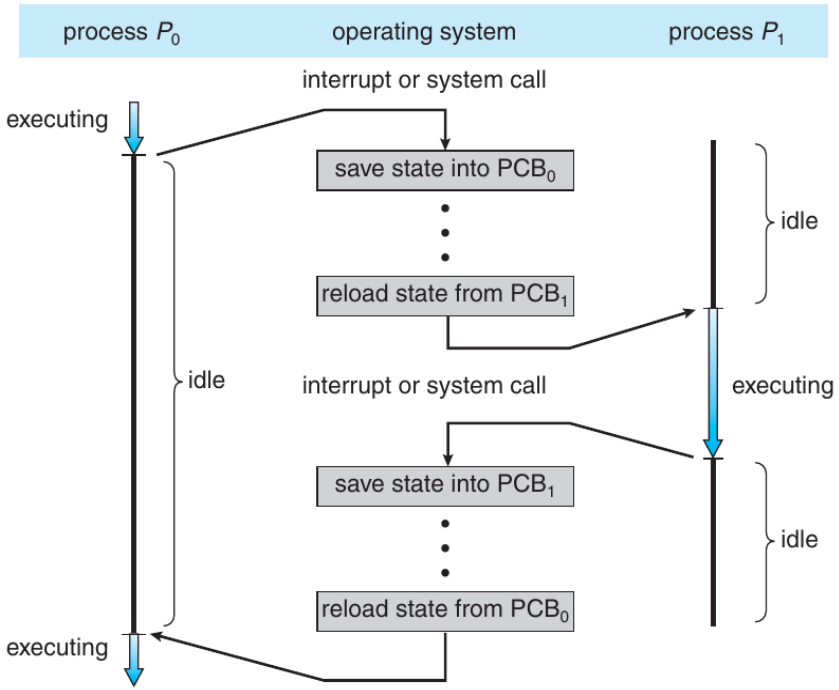
[8] Esta etapa é denominada *state restore* ou "recuperar estado".

Troca de Contexto

- A Troca de Contexto é um tipo de operação que não executa nenhum trabalho realmente útil durante a ação
  - A operação é necessária, porém não há processamento efetivo;
  - Desse modo, o procedimento deve ser o mais enxuto possível;
- A velocidade da mudança de contexto varia de acordo com o hardware e a quantidade de informação a ser copiada
  - Dependente da velocidade da memória, do número de registros a serem copiados e da existência de instruções especiais;
  - A velocidade típica é de alguns milissegundos [Silberschatz et al., 2012].

Troca de Contexto

- Quando um processo é interrompido, seu estado atual é salvo no PCB (e ao reiniciar, os dados são lidos da estrutura) [Silberschatz et al., 2012].



Process Control Block.
Fonte: [Silberschatz et al., 2012]

Scheduling

Scheduling

INSTITUTO FEDERAL
Minas Gerais

- Sistemas operacionais buscam maximizar o uso as CPUs
  - Para isso, em sistemas de tempo compartilhado, a CPU troca o processo em execução constantemente;
  - Essa estratégia permite que usuários interajam com múltiplas aplicações ao mesmo tempo
    - Ex.: Usuário navega na internet enquanto escuta música;
- O Agendador de Processos (*Process Scheduler*) é responsável pela seleção de tarefas para execução na CPU
  - Para computadores com uma única CPU, somente um processo é executado por vez;
  - Demais processos devem aguardar a finalização do processo em execução [Silberschatz et al., 2012].

Scheduling

- O scheduler utiliza um algoritmo de agendamento[9], para escolher qual atividade será executada
  - São selecionadas tarefas cujo estado seja "pronto" (*ready*);
  - Processos nesse estado ficam armazenado em uma fila específica (*ready queue*)[10]
    - Essa fila segue uma estrutura semelhante às listas encadeadas;
    - Um item da fila contém o PCB e um apontamento para o próximo item da fila.

[9] Nomenclatura em inglês: *scheduling algorithm*.

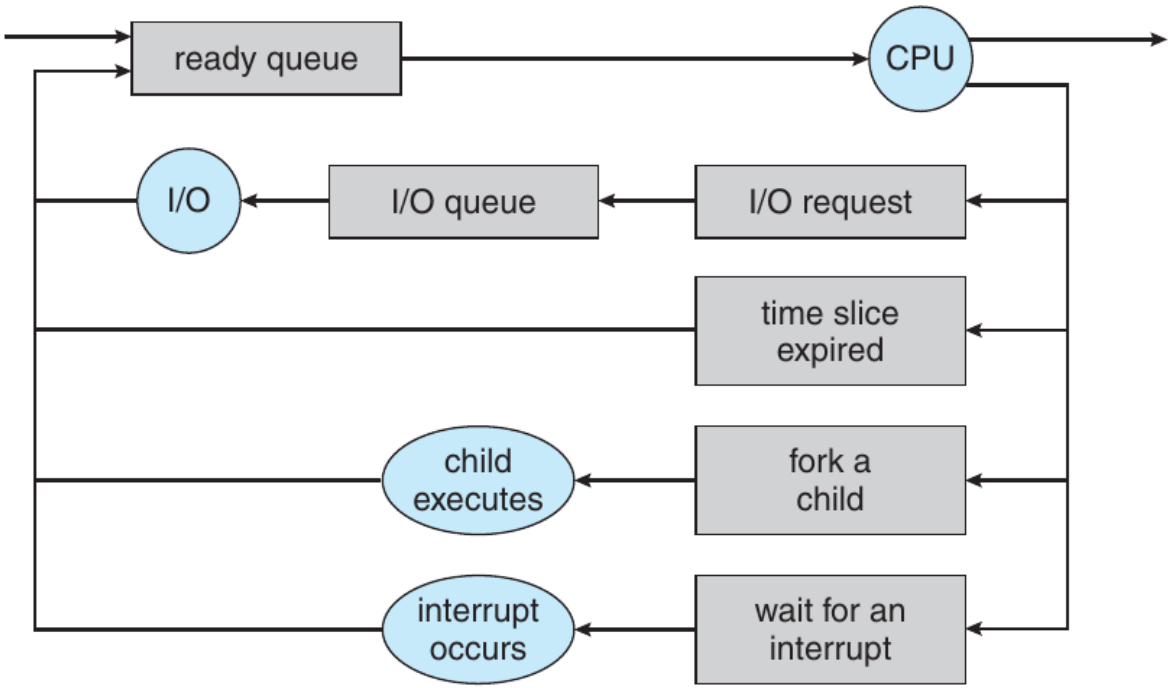
[10] De forma geral, todos os processos existentes ficam em uma fila genérica, denominada *job queue*. Outras filas podem ser utilizadas para tarefas específicas (ex.: fila de espera pelo retorno de um dispositivo).

Scheduling

- Um processo no estado de pronto será, eventualmente, executado pela CPU;
- Durante a execução, um processo pode:
  - Requisitar acesso a I/O (será adicionado à fila de espera do dispositivo solicitado);
  - Criar processos filhos e aguardar a finalização desses processos;
  - Ser interrompido pela CPU, devido ao esgotamento do tempo disponível para execução (*time-sharing*)
    - O processo é interrompido e colocado novamente na fila de processos prontos para execução (esse comportamento pode se repetir inúmeras vezes).

Scheduling

- A sequência de estados de um processo pode ser vista na figura abaixo:



Fila de processos prontos e alteração de estados do processo.
Fonte: [Silberschatz et al., 2012]

Schedulers

Tipos de Schedulers

- Para gerenciamento de processos, existem 3 tipos principais de schedulers:
  1. Long-Term Scheduler;
  2. Short-Term Scheduler;
  3. Medium-Term Scheduler.

Long-Term Scheduler

- Long-Term Schedulers[11] (Job Schedulers) são responsáveis por selecionar processos de um pool de processos (*job pool*) e adicioná-los à fila de prontos
  - Mesmo em um sistema de execução de *batch*, alguns processos precisarão esperar para executar;
- O Long-Term Scheduler é responsável por definir o grau de multiprogramação - quantos processos estão disponíveis para execução (em memória) [Shekhar, 2019].

[11] Tradução: Agendador/Escalonador de Longo Prazo.

Long-Term Scheduler

- Long-Term Schedulers são executados somente quando novos processos são criados (o que pode demorar muitos minutos)
  - Como a frequência de execução é menor, esse escalonador pode demorar mais tempo para escolher um processo, decidindo de forma mais eficiente [Silberschatz et al., 2012];
- Esse tipo de scheduler tem um efeito de longo prazo no sistema [Shekhar, 2019].

Long-Term Scheduler

- Long-Term Schedulers devem levar em consideração características dos processos, com relação ao seu uso de CPU e dependência de I/O;
  - *I/O-bound processes*: gastam mais tempo fazendo operações de I/O que processando;
  - *CPU-bound processes*: gastam mais tempo processando que fazendo operações de I/O;
- LTS devem combinar processos *I/O-bound* e *CPU-bound* para melhor desempenho
  - Se todos os processos dependerem de I/O, a CPU ficará ociosa;
  - Se todos os processos precisarem de processamento constante, dispositivos I/O ficarão ociosos [Silberschatz et al., 2012].

Short-Term Scheduler

- Short-Term Schedulers[12] (CPU Schedulers) são responsáveis por selecionar processos na fila de prontos e definir uma CPU para executá-los
  - O processo passará para o estado de "Em execução";
- O Short-Term Scheduler é responsável por selecionar processos para melhor desempenho do sistema operacional
  - Frequentemente são utilizadas filas de prioridade, onde processos são ordenados segundo sua prioridade de execução;
  - Processos com maior prioridade são executados primeiro;

[12] Tradução: Agendador/Escalonador de Curto Prazo.

Short-Term Scheduler

- Short-Term Schedulers são executados constantemente[13]
  - A escolha de processos deve ser o mais rápida possível;
  - Quanto mais tempo demorar para escolher, mais tempo de processamento é perdido [Silberschatz et al., 2012];
- Esse scheduler tem um efeito de curto prazo, em busca de obter o maior desempenho no momento [Shekhar, 2019].

[13] Por exemplo, a cada 100ms.

Medium-Term Scheduler

- Medium-Term Schedulers são utilizados para colocar processos novamente na fila de processos prontos para executar[14]
  - Esse etapa ocorre para processos que executaram, mas não terminaram (ou seja, foram interrompidos);
  - Os processos serão executados novamente em uma nova oportunidade;
- Esse scheduler tem um efeito de médio prazo no desempenho do sistema [Shekhar, 2019].

[14] Também pode ser utilizado para adicionar processos à fila de espera ou processos bloqueados.

Medium-Term Scheduler

- Medium-Term Schedulers também são responsáveis por mover processos da memória principal para secundária (e vice-versa)
  - Em alguns cenários pode ser vantajoso remover um processo da memória, reduzindo o grau de multiprogramação
    - Por exemplo, quando há pouco espaço disponível;
  - Posteriormente, o processo pode ser reintroduzido na memória;
  - Essa esquema é denominado swapping [Shekhar, 2019] [Silberschatz et al., 2012].

Comunicação

Conceitos

- Processos em execução concorrente podem ser classificados em [Silberschatz et al., 2012]:
  - Processos Independentes: não podem afetar ou serem afetados por outros processos do sistema;
    - Qualquer processo que não compartilhe dados com outro processo é definido como independente.
  - Processos em Cooperação: podem afetar ou serem afetados por outros processos do sistema;
    - Processos que compartilham seus dados com outros.

Motivação

- Alguns motivos para cooperação entre processos são:
  - Compartilhamento de Informações:
    - Muitos usuários podem precisar de uma mesma informação;
    - Compartilhar é mais eficiente que replicar (economiza memória, evita diversas cópias para sincronização, etc);
  - Desempenho computacional:
    - Subtarefas podem ser executadas em paralelo, aumentando o desempenho do sistema;

Motivação

- Alguns motivos para cooperação entre processos são:
  - Modularidade:
    - Módulos podem ser criados e separados em diferentes processos independentes (se um módulo falhar, outros módulos continuam ativos);
  - Conveniência:
    - Tarefas diferentes podem ser executadas em paralelo;
    - Ex.: Em um programa de exibição de filmes, o *player* do vídeo pode executar em um processo, enquanto os comentários são exibidos em um processo diferente.

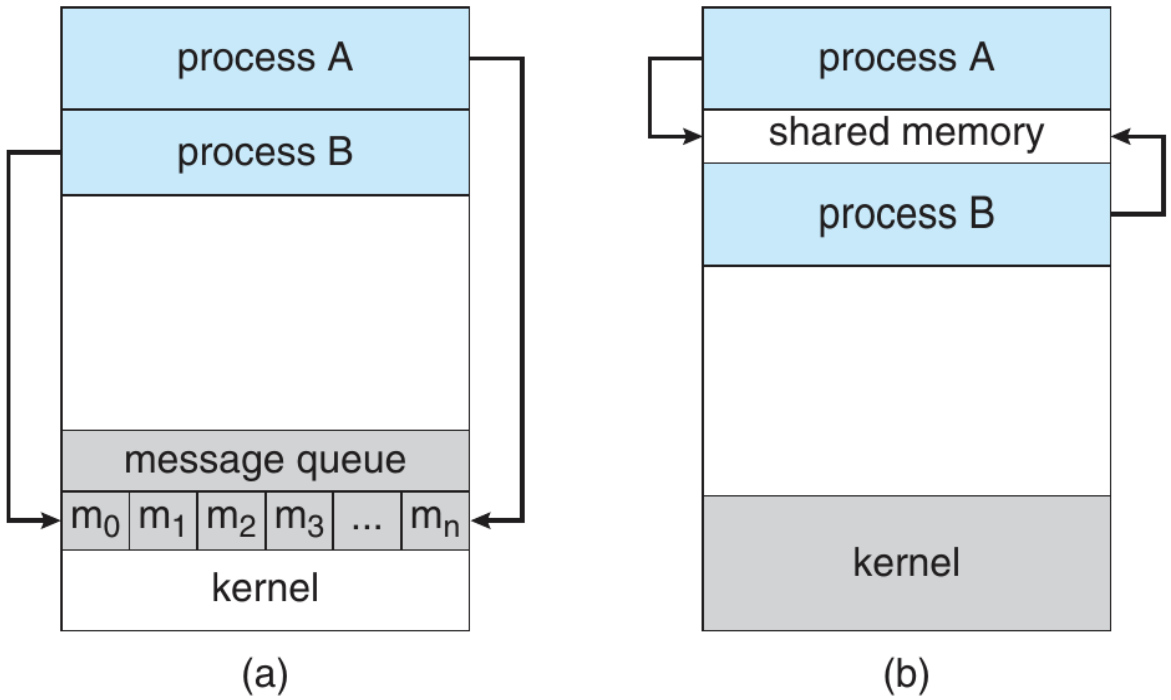
Comunicação entre Processos

- Comunicação entre Processos (IPC)[15] é o mecanismo que permite processos trocarem dados e informações [Silberschatz et al., 2012];
- Mecanismos de comunicação entre processos:
  - Memória Compartilhada:
    - Uma região de memória é compartilhada entre processos;
    - Processos podem ler e escrever informações nesse espaço;
  - Troca de Mensagens:
    - Mensagens são trocadas entre os processos (em geral, são transimitidas pequenas quantidades de informações).
    - Esse modelo evita conflitos entre os processos.

[15] IPC é o acrônimo de Interprocess Communication.

Comunicação entre Processos

- Os mecanismos de comunicação entre processos estão representados na figura abaixo:



Comunicação entre Processos. (a) Troca de Mensagens. (b) Memória compartilhada.
Fonte: [Silberschatz et al., 2012]

Comunicação entre Processos

- Sistemas Operacionais, em geral, implementam ambos os métodos de IPC
  - Memória Compartilhada:
    - Mais rápida que a troca de mensagens, uma vez que não requer execução de chamadas de sistema;
    - Pode causar conflitos, em caso de leitura/escrita de informações desatualizadas[16]
  - Troca de Mensagens:
    - Mais lento, uma vez que depende da mediação do *kernel*;
    - Evita conflitos entre os processos [Silberschatz et al., 2012].

[16] Essa situações serão descritas na seção Sincronização de Processos.

Comunicação entre Processos

- Segundo [Silberschatz et al., 2012], algumas pesquisas indicam que o mecanismo de Troca de Mensagens é mais eficiente que o uso de Memória Compartilhada
  - Troca de mensagens, apesar de mais lento, não necessita de mecanismos de coerência de informações[17];
  - Com o aumento do número de processadores e das aplicações paralelas, os problemas de coerência de informação podem se tornar mais frequentes.

[17] Esses mecanismos serão descritas na seção Sincronização de Processos.

Comunicação Cliente-Servidor

Comunicação Cliente-Servidor

INSTITUTO FEDERAL
Minas Gerais

- Em sistemas cliente-servidor, são necessárias estratégias diferentes para comunicação entre processos;
- As estratégias mais comuns são:
  - Sockets;
  - Remote Procedure Calls;
  - Pipes.

Sockets

INSTITUTO FEDERAL
Minas Gerais

- Sockets são terminais para comunicação
  - Dois processos utilizam um par de sockets para se comunicar (servidor e cliente);
  - Um socket é identificado por um IP + número da porta;
  - O servidor espera por solicitações de clientes, ouvindo uma porta específica;
  - Ao receber a solicitação, o servidor aceita a conexão e troca dados/informações com o cliente;
  - Alguns serviços comuns, como HTTP, FTP, SMTP, POP3 e SSH são implementados utilizando sockets;
  - Sockets são serviços de baixo nível, portanto, somente permitem uma sequência de bytes não estruturados.

Remote Procedure Calls (RPC)

- Remote Procedure Calls (RPC) são estruturas para comunicação em conexões de rede
  - Utiliza um mecanismo de comunicação para serviços remotos;
  - RPCs são bem estruturados, não constituindo apenas de uma sequência de bytes;
  - Cada mensagem é endereçada a um *daemon*[18] RPC em uma porta específica, identificando uma função e contendo parâmetros para execução;
  - A função é executada e o retorno é enviado ao cliente em uma mensagem separada [Silberschatz et al., 2012].

[18] Daemon é um programa que é executado como um processo em background.

Pipes

- Pipes são mecanismos onde a saída de um processo é direcionada para a entrada de outro
  - São estruturas de comunicação implementadas nos primeiros sistemas UNIX;
  - Fornece, um fluxo de dados unilateral entre dois processos;
  - São implementados usando Filas (FIFO);
  - Um processo escreve um item na fila e outro processo lê esse item [GeeksForGeeks, 2020] [Silberschatz et al., 2012].

Sincronização

Sincronização

- Processos em Cooperação são aqueles que compartilham seu espaço de endereçamento ou seus dados com outros processos;
- Esses processos podem, em algum momento, fazer acesso concorrente a um dado ou um recurso
  - Devido ao mecanismo de *scheduling*, o acesso a um recurso pode ser interrompido antes de sua completa execução;
  - Tal condição pode gerar inconsistências;
- Mecanismos de Sincronização são utilizados para evitar essas inconsistências nos dados compartilhados.

Ex.: Problema de Sincronização

- Considere a execução de dois processos em cooperação, P1 e P2, de uma fonte de dados compartilhada:
  1. P1: Lê uma variável `x`, de um endereço compartilhado, e copia para uma variável local (`x=1`);
  2. P1: Incrementa a variável (`x++`);
  3. P1: Escreve o resultado na variável compartilhada (`x=2`).
  4. P2: Lê uma variável `x`, de um endereço compartilhado, e copia para uma variável local (`x=2`);
  5. P2: Decrementa a variável (`x--`);
  6. P2: Escreve o resultado na variável compartilhada (`x=1`).
- O que aconteceria se o mecanismo de *scheduling* interrompesse a execução dos processos?

Ex.: Problema de Sincronização

- Considere a execução de dois processos em cooperação, P1 e P2, de uma fonte de dados compartilhada:
  1. P1: Lê uma variável `x`, de um endereço compartilhado, e copia para uma variável local (`x=1`);
  2. P1: Incrementa a variável (`x++`);
  3. P1: Escreve o resultado na variável compartilhada (`x=2`).
  4. P2: Lê uma variável `x`, de um endereço compartilhado, e copia para uma variável local (`x=2`);
  5. P2: Decrementa a variável (`x--`);
  6. P2: Escreve o resultado na variável compartilhada (`x=1`).
- O que aconteceria se o mecanismo de *scheduling* interrompesse a execução dos processos?

Ex.: Problema de Sincronização

- Considere a execução de dois processos em cooperação, P1 e P2, de uma fonte de dados compartilhada:
  1. P1: Lê uma variável `x`, de um endereço compartilhado, e copia para uma variável local (`x=1`);
  2. P1: Incrementa a variável (`x++`);
  3. P1: Escreve o resultado na variável compartilhada (`x=2`).
  4. P2: Lê uma variável `x`, de um endereço compartilhado, e copia para uma variável local (`x=2`);
  5. P2: Decrementa a variável (`x--`);
  6. P2: Escreve o resultado na variável compartilhada (`x=1`).
- O que aconteceria se o mecanismo de *scheduling* interrompesse a execução dos processos?

Ex.: Problema de Sincronização

INSTITUTO FEDERAL
Minas Gerais

- Considere a execução de dois processos em cooperação, P1 e P2, de uma fonte de dados compartilhada:
  1. P1: Lê uma variável `x`, de um endereço compartilhado, e copia para uma variável local (`x=1`);
  2. P1: Incrementa a variável (`x++`);
  3. P1: Escreve o resultado na variável compartilhada (`x=2`).
  4. P2: Lê uma variável `x`, de um endereço compartilhado, e copia para uma variável local (`x=2`);
  5. P2: Decrementa a variável (`x--`);
  6. P2: Escreve o resultado na variável compartilhada (`x=1`).
- O que aconteceria se o mecanismo de *scheduling* interrompesse a execução dos processos?

Ex.: Problema de Sincronização

- Considere uma nova execução de dois processos, P1 e P2, interrompida pelo mecanismo de *scheduling*:
  1. P1: Lê uma variável `x`, de um endereço compartilhado, e copia para uma variável local (`x=1`);
  2. P1: Incrementa a variável (`x++`);
  3. P2: Lê uma variável `x`, de um endereço compartilhado, e copia para uma variável local (`x=1`);
  4. P2: Decrementa a variável (`x--`);
  5. P1: Escreve o resultado na variável compartilhada (`x=2`).
  6. P2: Escreve o resultado na variável compartilhada (`x=0`).
- Conforme pode ser observado, o resultado final da execução é diferente do exemplo anterior, o que é um erro!

Suponha que o mecanismo de *scheduling* foi executado após o processamento dos itens 2 e 4.

Ex.: Problema de Sincronização

- Considere uma nova execução de dois processos, P1 e P2, interrompida pelo mecanismo de *scheduling*:
  1. P1: Lê uma variável `x`, de um endereço compartilhado, e copia para uma variável local (`x=1`);
  2. P1: Incrementa a variável (`x++`);
  3. P2: Lê uma variável `x`, de um endereço compartilhado, e copia para uma variável local (`x=1`);
  4. P2: Decrementa a variável (`x--`);
  5. P1: Escreve o resultado na variável compartilhada (`x=2`).
  6. P2: Escreve o resultado na variável compartilhada (`x=0`).
- Conforme pode ser observado, o resultado final da execução é diferente do exemplo anterior, o que é um erro!

Suponha que o mecanismo de *scheduling* foi executado após o processamento dos itens 2 e 4.

Ex.: Problema de Sincronização

- Considere uma nova execução de dois processos, P1 e P2, interrompida pelo mecanismo de *scheduling*:
  1. P1: Lê uma variável `x`, de um endereço compartilhado, e copia para uma variável local (`x=1`);
  2. P1: Incrementa a variável (`x++`);
  3. P2: Lê uma variável `x`, de um endereço compartilhado, e copia para uma variável local (`x=1`);
  4. P2: Decrementa a variável (`x--`);
  5. P1: Escreve o resultado na variável compartilhada (`x=2`).
  6. P2: Escreve o resultado na variável compartilhada (`x=0`).
- Conforme pode ser observado, o resultado final da execução é diferente do exemplo anterior, o que é um erro!

Suponha que o mecanismo de *scheduling* foi executado após o processamento dos itens 2 e 4.

Ex.: Problema de Sincronização

- Considere uma nova execução de dois processos, P1 e P2, interrompida pelo mecanismo de *scheduling*:
  1. P1: Lê uma variável `x`, de um endereço compartilhado, e copia para uma variável local (`x=1`);
  2. P1: Incrementa a variável (`x++`);
  3. P2: Lê uma variável `x`, de um endereço compartilhado, e copia para uma variável local (`x=1`);
  4. P2: Decrementa a variável (`x--`);
  5. P1: Escreve o resultado na variável compartilhada (`x=2`).
  6. P2: Escreve o resultado na variável compartilhada (`x=0`).

- Conforme pode ser observado, o resultado final da execução é diferente do exemplo anterior, o que é um erro!

Suponha que o mecanismo de *scheduling* foi executado após o processamento dos itens 2 e 4.

Ex.: Problema de Sincronização

INSTITUTO FEDERAL
Minas Gerais

- Considere uma nova execução de dois processos, P1 e P2, interrompida pelo mecanismo de *scheduling*:
  1. P1: Lê uma variável `x`, de um endereço compartilhado, e copia para uma variável local (`x=1`);
  2. P1: Incrementa a variável (`x++`);
  3. P2: Lê uma variável `x`, de um endereço compartilhado, e copia para uma variável local (`x=1`);
  4. P2: Decrementa a variável (`x--`);
  5. P1: Escreve o resultado na variável compartilhada (`x=2`).
  6. P2: Escreve o resultado na variável compartilhada (`x=0`).
- Conforme pode ser observado, o resultado final da execução é diferente do exemplo anterior, o que é um erro!

Suponha que o mecanismo de *scheduling* foi executado após o processamento dos itens 2 e 4.

Ex.: Problema de Sincronização

INSTITUTO FEDERAL
Minas Gerais

- Considere uma nova execução de dois processos, P1 e P2, interrompida pelo mecanismo de *scheduling*:
  1. P1: Lê uma variável `x`, de um endereço compartilhado, e copia para uma variável local (`x=1`);
  2. P1: Incrementa a variável (`x++`);
  3. P2: Lê uma variável `x`, de um endereço compartilhado, e copia para uma variável local (`x=1`);
  4. P2: Decrementa a variável (`x--`);
  5. P1: Escreve o resultado na variável compartilhada (`x=2`).
  6. P2: Escreve o resultado na variável compartilhada (`x=0`).
- Conforme pode ser observado, o resultado final da execução é diferente do exemplo anterior, o que é um erro!

Suponha que o mecanismo de *scheduling* foi executado após o processamento dos itens 2 e 4.

Sincronização

- Conforme exemplo anterior, a ordem de execução de processos que possuem informação compartilhada pode resultar em valores incorretos, caso não exista o devido cuidado
  - Situações semelhantes ao exemplo são denominadas condições de corrida;
- Para evitar erros, é necessário estabelecer mecanismos de sincronização e coordenação
  - Ao escrever o código fonte de uma aplicação, devem ser identificadas seções cuja paralelização pode causar erros;

Problema da Seção Crítica

- Seções críticas são trechos do código-fonte onde a paralelização pode causar erros, devido ao compartilhamento de informações
  - Com isso, dois processos não devem ser capazes de acessar a seção crítica em um mesmo instante;
  - Quando um processo A entrar em uma seção crítica do software, outros não poderão acessar esse trecho de código
    - Demais processos devem aguardar até que A tenha finalizado esta seção;
  - O Problema da Seção Crítica busca projetar protocolos para que processos possam usar para cooperação.

Problema da Seção Crítica

- Segundo [Silberschatz et al., 2012], o problema da seção crítica deve satisfazer os seguintes requisitos:
  1. Exclusão mútua: se o processo P_i está executando sua seção crítica, nenhum outro processo pode estar executando em suas seções críticas;
  2. Progresso: se nenhum processo está executando sua seção crítica e alguns processos desejam executar essa seção, a decisão será tomada apenas pelos processos interessados (e não pode ser adiada indefinidamente);
  3. Espera limitada: existe um limite no número de vezes que outros processos podem entrar em suas seções críticas depois que um processo fez uma solicitação de acesso à seção.

Mutex Locks

- Uma das inúmeras soluções[19] para o problema é denominada Mutex Locks[20]
  - Mutex é um corresponde à abreviação de exclusão mútua (*mutual exclusion*);
- Os Mutex são utilizados para evitar condições de corrida em seções críticas do código;
- Para que um processo entre em uma seção crítica, deverá adquirir (*acquire*) um lock[21], que será liberado (*release*) apenas ao fim da seção crítica [Silberschatz et al., 2012].

[19] Dentre as principais soluções, destacam-se a Solução de Perterson, os Semáforos e os Mutex Locks.

[20] Denominado também apenas como Mutex ou apenas como Locks.

[21] Pode ser traduzido como tranca, cadeado ou fechadura. No entanto, é comum o uso do conceito em inglês.

Mutex Locks

- O mecanismo do Mutex pode ser visto na figura abaixo:
  - Quando um processo solicita o lock, este fica aguardando até a liberação do recurso.

```
acquire() {
    while (!available)
       ; /* busy wait */
    available = false;;
}

do {

    acquire lock

        critical section

    release lock

        remainder section

} while (true);
```

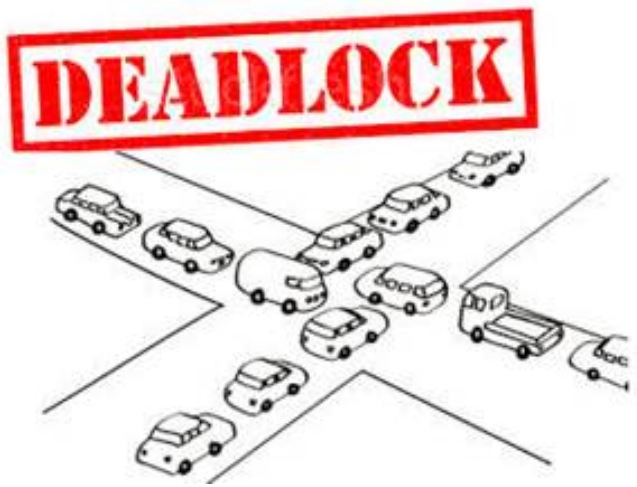
Mutex Locks.
Fonte: [Silberschatz et al., 2012]

Algumas linguagens, como Java, possuem comandos próprios para implementação simplificada de locks.

Deadlocks

Deadlocks

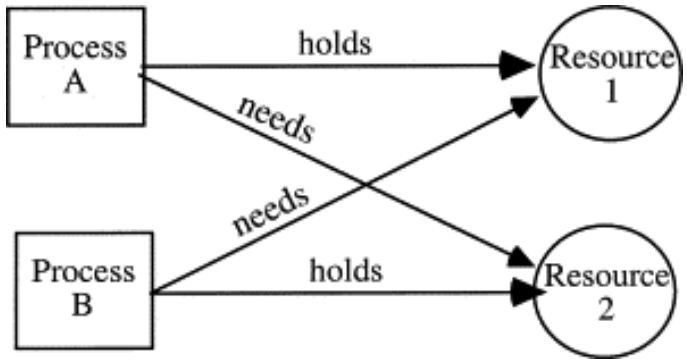
- Deadlock corresponde a uma situação de dependência cíclica, causada por *locks* em diferentes recursos necessários à conclusão dos processos.



Deadlock.
Fonte: [Stefanetti, 2018]

Deadlocks

- Um exemplo de *deadlock* pode ser visto na figura abaixo:
  - Processos A e B precisam de dois recursos, R_1 e R_2, para finalizarem sua execução;
  - Proc. *A* possui o controle de R_1 e aguarda a liberação de R_2;
  - Proc. *B* possui o controle de R_2 e aguarda a liberação de R_1;
  - Ambos os processos não liberam seu recurso, gerando um impasse [Keller, 2003].



Deadlock.
Fonte: [Keller, 2003]

Deadlocks

INSTITUTO FEDERAL
Minas Gerais

- Apesar de alguns softwares conseguirem identificar *deadlocks*, em geral, cabe ao programador a tarefa de prevenção;
- Os deadlocks derivam obrigatoriamente das seguintes condições [Silberschatz et al., 2012]:
  - Exclusão Mútua: um recurso está bloqueado para uso exclusivo de um único processo;
  - Segurar e Esperar: o processo segura um recurso enquanto espera por outros recursos;
  - Sem Preempção: recursos somente podem ser liberados voluntariamente (não podem ser preemptados);
  - Espera Circular: um processo aguarda recursos que estão em posse de processos que aguardam recursos sob a posse de terceiros (formando um ciclo).

Deadlocks

- Para lidar com *deadlocks*, um SO pode utilizar as seguintes abordagens: [Silberschatz et al., 2012]
  - Prevenir deadlocks, evitando que o sistema entre nesse estado;
  - Recuperar deadlocks, criando mecanismos para detecção e recuperação do problema;
  - Ignorar deadlocks, fingindo que o problema não ocorre e deixando a solução para o usuário.

Prevenção de Deadlocks

INSTITUTO FEDERAL
Minas Gerais

- A prevenção de *deadlocks* busca evitar ao menos uma das 4 condições obrigatórias: [Silberschatz et al., 2012]
  - Exclusão Mútua: deve-se evitar a restrição de acesso a um recurso por um único processo;
  - Segurar e Esperar: um processo somente pode solicitar um recurso se não tiver nenhum recurso bloqueado; ou deve solicitar acesso a todos os recursos de uma só vez;
  - Sem Preempção: se um processo solicita um recurso, deve obrigatoriamente liberar os recursos que possui;
  - Espera Circular: recursos devem ser numerados e divididos em tipos e cada processo deve solicitar o recursos somente em uma sequência[22].

[22] Ex.: Um processo pode requisitar acesso ao SSD e em seguida ao HD, mas nunca ao HD e em seguida ao SSD.

Threads

Threads

- Cada processo pode ser composto de uma ou múltiplas threads
  - Threads estendem do conceito de processos e permitem a execução de mais de uma tarefa por vez;
- A thread pode ser definida como a unidade básica de utilização de uma CPU [Silberschatz et al., 2012];
- Em sistemas multithreading, o PCB também inclui informações sobre threads de um processo.

Simultaneous multithreading (SMT) é uma tecnologia que permite a execução de duas ou mais threads por núcleo do processador. Alguns fabricantes adotam nomes específicos, como Hyperthreading (Intel).

Threads

- Threads podem ser traduzidas como “fio” (ou sequência) de execução de um processo
  - Tradicionalmente, os processos continham somente uma thread;
  - No entanto, sistemas modernos permitem a execução de múltiplas threads em um mesmo processo;
  - Tal característica possibilita a execução de mais de uma tarefa por vez (dentro de um processo);
  - Devido a essas características, [Tanenbaum and Bos, 2014] descrevem threads como “mini processos”.

Nomenclatura

- Um processo tradicional contém uma única thread
  - Esses processos são denominados *heavyweight process*[23];
- Processos modernos, em geral, contém múltiplas threads
  - Alguns autores utilizam o termo *lightweight*[24] para diferenciar dos processos tradicionais.
- Utiliza-se, frequentemente, a nomenclatura em inglês single-thread e multi-thread para designar processos de uma e múltiplas threads, respectivamente.

[23] Tradução direta: processos peso pesados.

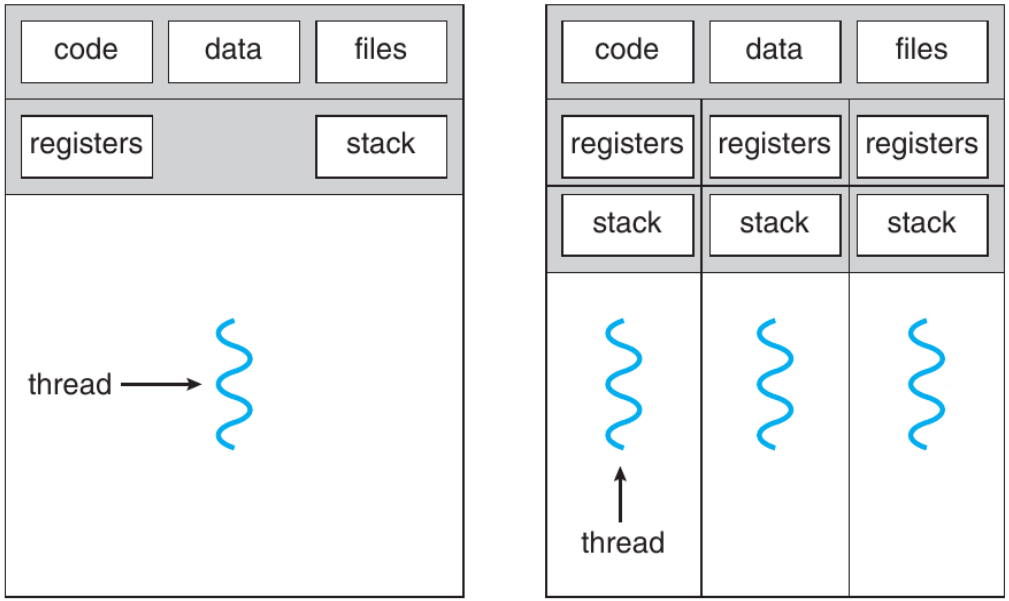
[24] Tradução direta: peso leve.

Threads

- Uma thread é composta de:
  - Identificador (ID) da thread;
  - Contador de programa;
  - Conjunto de registradores;
  - Pilha;
- Uma thread compartilha com outras threads, em um mesmo processo:
  - Seção de código;
  - Seção de dados;
  - Recursos dos sistemas operacionais (ex.: arquivos abertos) [Silberschatz et al., 2012].

Organização em Memória

- Processos single e multi-threading podem ser vistos na figura abaixo.



Processos single-thread e multi-thread.
Fonte: [Silberschatz et al., 2012]

Motivação

- Threads podem ser utilizadas para subdividir um processo
  - Ex. 1: Browser
    - Cada aba pode ser colocada em um processo diferente;
    - Na aba, threads podem ser utilizadas para diferentes tarefas como: execução do player de video, exibição de comentários, gerenciamento de caixas de texto, etc;
    - Essa subdivisão permite o processamento de forma paralela e permite uso mais amigável da aplicação;
  - Ex. 2: Servidor web
    - Um servidor web deve monitorar portas do SO (ex.: 80, 443, 8080, etc), para retornar páginas web;
    - Essa tarefa deve ser feita por um único processo;
    - Para múltiplos clientes, cada solicitação pode ser respondida em uma thread diferente.

Vantagens

- Sistemas multithreading possuem as seguintes vantagens: [Silberschatz et al., 2012]
  - Responsividade
    - Permite que um programa continue sua execução mesmo se parte dele estiver bloqueado ou executando operações demoradas;
    - Possibilita o aumento da capacidade de resposta ao usuário, uma vez que não é necessário aguardar a finalização de tarefas;
    - Melhora a usabilidade do sistema;
  - Compartilhamento de Recursos
    - Threads nativamente compartilham informações[25], ao contrário de processos, que precisam ser programados explicitamente;

[25] Somente dentro de um mesmo processo.

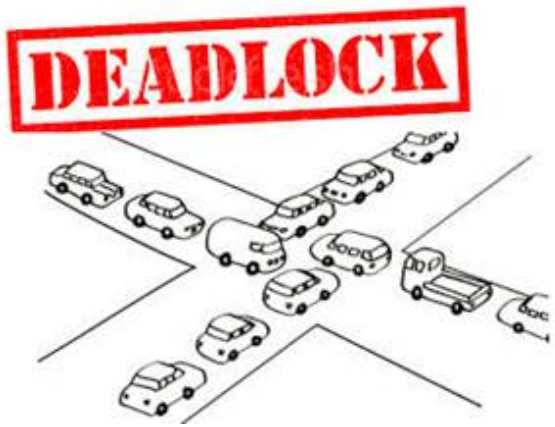
Vantagens

- Sistemas multithreading possuem as seguintes vantagens: [Silberschatz et al., 2012]
  - Economia
    - Threads não necessitam de alocação de memória e criação de processos, tarefas computacionalmente caras;
    - Threads, de forma geral, são mais rápidas de serem criadas e gerenciadas[26];
  - Escalabilidade
    - Múltiplas threads podem ser executadas de formar paralela, permitindo maior desempenho computacional;
    - Processos de thread única podem ser executados em apenas um processador, independentemente da quantidade disponível.

[26] No Solaris, processos são cerca de 30x mais lentos de serem criados que threads [Silberschatz et al., 2012].

Desvantagens

- Sistemas multithreading também são sujeitos a problemas de deadlocks
  - Todos os mecanismos de prevenção e recuperação de deadlocks podem ser estendidos para o gerenciamento de threads.



Deadlock.
Fonte: [Stefanetti, 2018]

Referências I

Bower, T. (2015).

Basics of how operating systems work.
[Online]; acessado em 06 de Setembro de 2021. Disponível em:
http://faculty.salina.k-state.edu/tim/ossg/Introduction/OSworking.html.

GeeksForGeeks (2020).

Ipc technique pipes.
[Online]; acessado em 07 de Setembro de 2021. Disponível em:
https://www.geeksforgeeks.org/ipc-technique-pipes/.

Keller, L. S. (2003).

Operating systems.
In Meyers, R. A., editor, Encyclopedia of Physical Science and Technology (Third Edition), pages 169–191. Academic Press, New York, third edition edition.

Shekhar, A. (2019).

What is long-term, short-term, and medium-term scheduler?
[Online]; acessado em 06 de Setembro de 2021. Disponível em:
https://afteracademy.com/blog/what-is-long-term-short-term-and-medium-term-scheduler.

Silberschatz, A., Galvin, P. B., and Gagne, G. (2012).

Operating System Concepts.
Wiley Publishing, 9th edition.

Referências II

Stefanetti, R. (2018).

Detect deadlocks in old navs.
[Online]; acessado em 08 de Setembro de 2021. Disponível em:
https://robertostefanettinavblog.com/2018/04/11/detect-deadlocks-in-old-navs-2017/.

Tanenbaum, A. S. and Bos, H. (2014).

Modern Operating Systems.
Prentice Hall Press, USA, 4th edition.