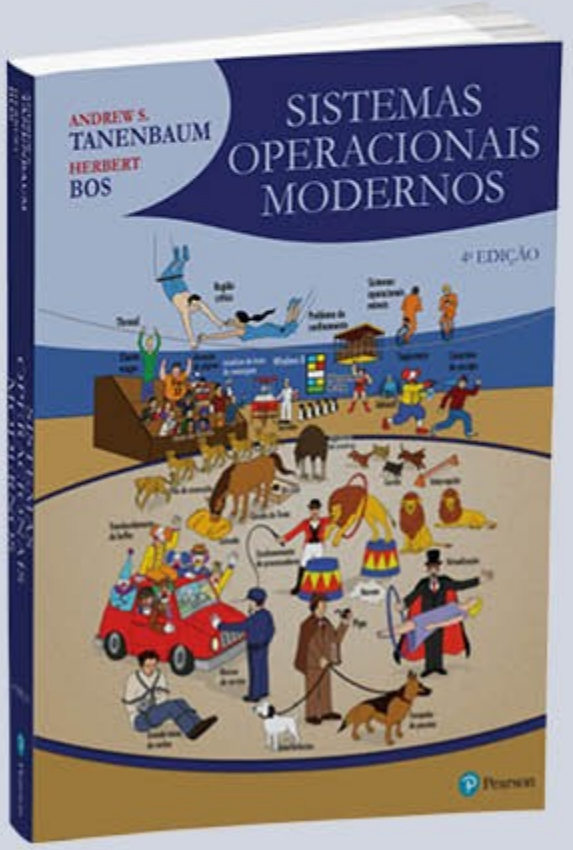
Capítulo 2:
Processos e threads



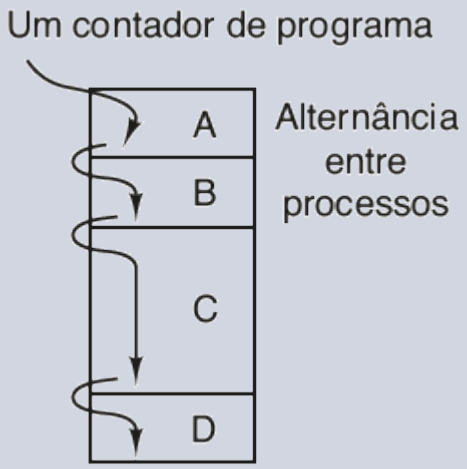
Processos

- O conceito mais central em qualquer sistema operacional é o **processo**: uma abstração de um programa em execução.
- Um processo é apenas uma instância de um programa em execução, incluindo os valores atuais do contador do programa, registradores e variáveis.
- Processos podem ser criados e terminados dinamicamente.
- Cada processo tem seu próprio espaço de endereçamento.

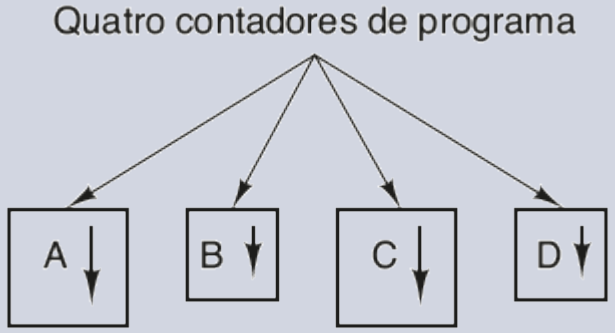
O modelo de processo

- Na **figura (a)** vemos um computador multiprogramando quatro programas na memória.
- Na **figura (b)** vemos quatro processos, cada um com seu próprio fluxo de controle e sendo executado independente dos outros.
- Na **figura (c)** vemos que, analisados durante um intervalo longo o suficiente, todos os processos tiveram progresso, mas a qualquer dado instante apenas um está sendo de fato executado.

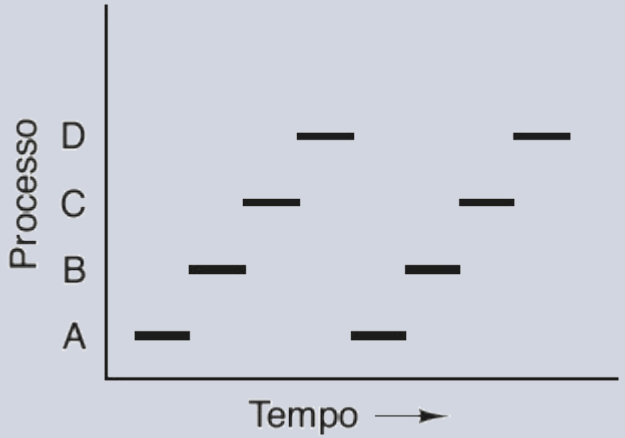
(a)



(b)



(c)



Criação de processos

- Sistemas operacionais precisam de alguma maneira para criar processos.
- Quatro eventos principais fazem com que os processos sejam criados:

  1. Inicialização do sistema.

  2. Execução de uma chamada de sistema de criação de processo por um processo em execução.

  3. Solicitação de um usuário para criar um novo processo.

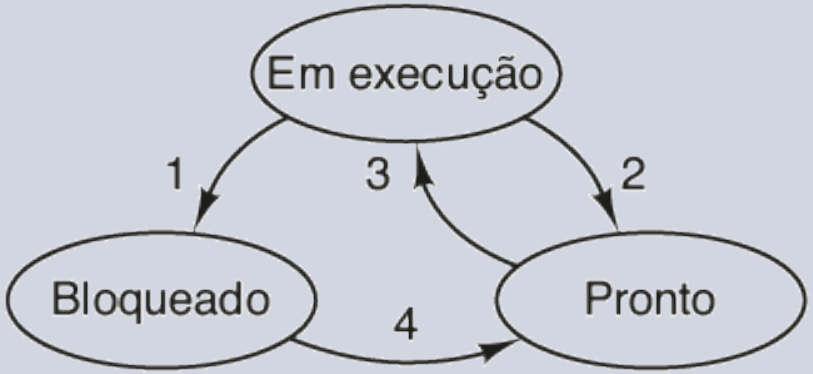
  4. Início de uma tarefa em lote.

Término de processos

- Após um processo ter sido criado, ele começa a ser executado e realiza qualquer que seja o seu trabalho. No entanto, nada dura para sempre, nem mesmo os processos. Cedo ou tarde, o novo processo terminará, normalmente devido a uma das condições a seguir:
  1. Saída normal (voluntária).
  2. Erro fatal (involuntário).
  3. Saída por erro (voluntária).
  4. Morto por outro processo (involuntário).

Hierarquias de processos

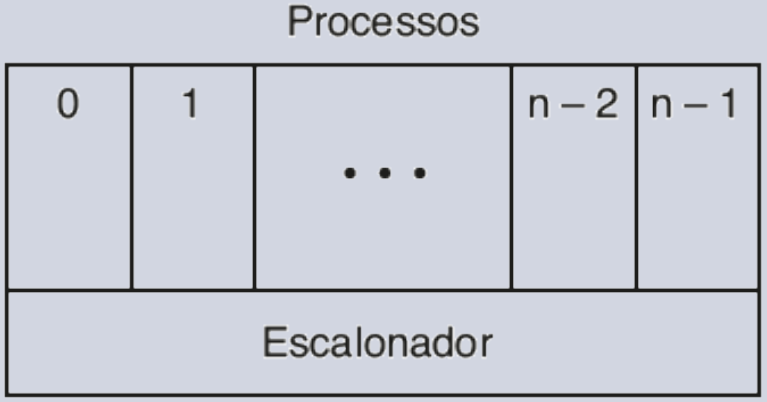
- Em alguns sistemas, quando um processo cria outro, o processo pai e o processo filho continuam a ser associados de certas maneiras.
- O processo filho pode em si criar mais processos, formando uma hierarquia de processos.
- **EXEMPLO**: Quando um usuário envia um sinal do teclado, o sinal é entregue a todos os membros do grupo de processos associados com o teclado no momento (em geral todos os processos ativos que foram criados na janela atual). Individualmente, cada processo pode pegar o sinal, ignorá-lo, ou assumir a ação predefinida, que é ser morto pelo sinal.



Estados de processos

- Embora cada processo seja uma entidade independente, com seu próprio contador de programa e estado interno, processos muitas vezes precisam interagir entre si.
- Um processo pode gerar alguma **saída** que outro processo usa como **entrada**.
- Na figura vemos um diagrama de estado mostrando os três estados nos quais um processo pode se encontrar:

1. Em execução (realmente usando a CPU naquele instante).
2. Pronto (executável, temporariamente parado para deixar outro processo ser executado).
3. Bloqueado (incapaz de ser executado até que algum evento externo aconteça).



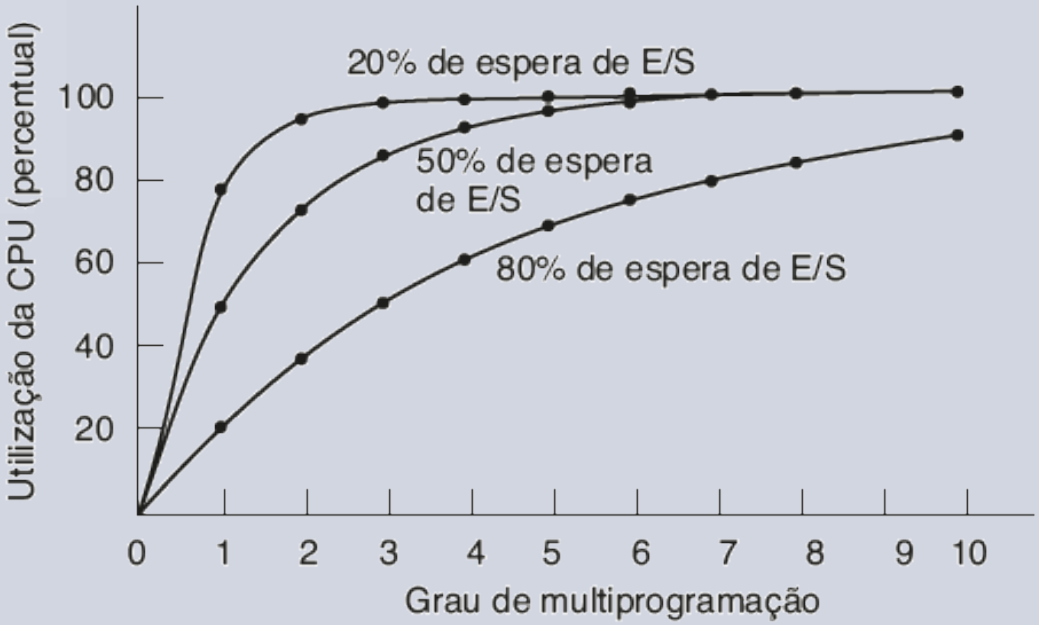
Estados de processos

- O nível mais baixo de um sistema operacional estruturado em processos controla interrupções e escalonamento. Acima desse nível estão processos sequenciais.
- Na figura o nível mais baixo do sistema operacional é o **escalonador**, com uma variedade de processos acima dele. Todo o tratamento de interrupções e detalhes sobre o início e parada de processos estão ocultos naquilo que é chamado aqui de escalonador, que, na verdade, não tem muito código. O resto do sistema operacional é bem estruturado na forma de processos. No entanto, poucos sistemas reais são tão bem estruturados como esse.

Implementação de processos

- Para implementar o modelo de processos, o sistema operacional mantém uma tabela (um arranjo de estruturas) chamada de **tabela de processos**, com uma entrada para cada um deles.
- Um processo pode ser interrompido milhares de vezes durante sua execução, mas a ideia fundamental é que, após cada interrupção, o processo retorne precisamente para o mesmo estado em que se encontrava antes de ser interrompido.

Modelando a multiprogramação

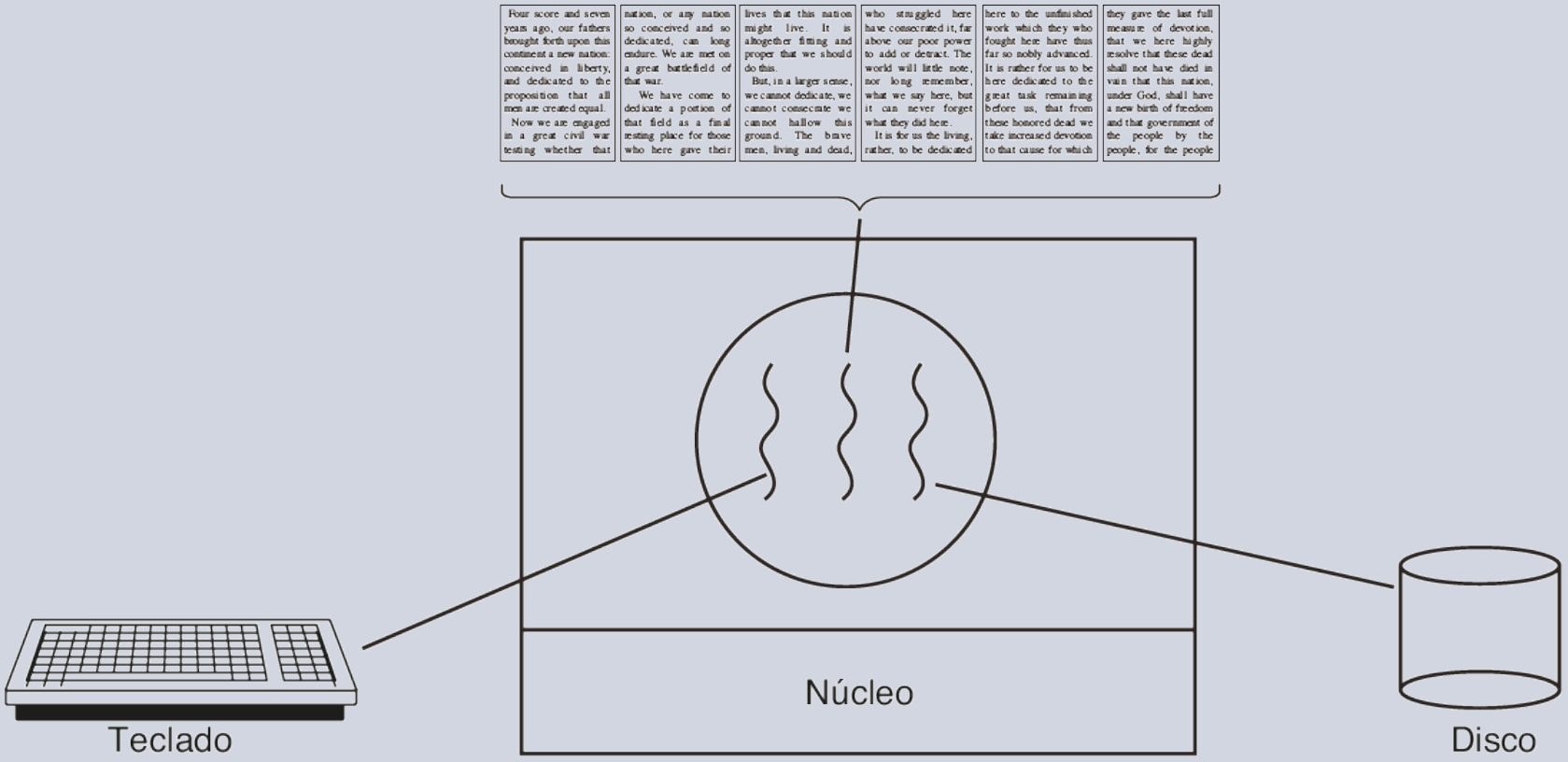


- Quando a multiprogramação é usada, a utilização da CPU pode ser aperfeiçoada. Colocando a questão de maneira direta, se o processo médio realiza computações apenas 20% do tempo em que está na memória, então com cinco processos ao mesmo tempo na memória, a CPU deve estar ocupada o tempo inteiro.
- A figura mostra a utilização da CPU como uma função de *n*, que é chamada de **grau de multiprogramação.**

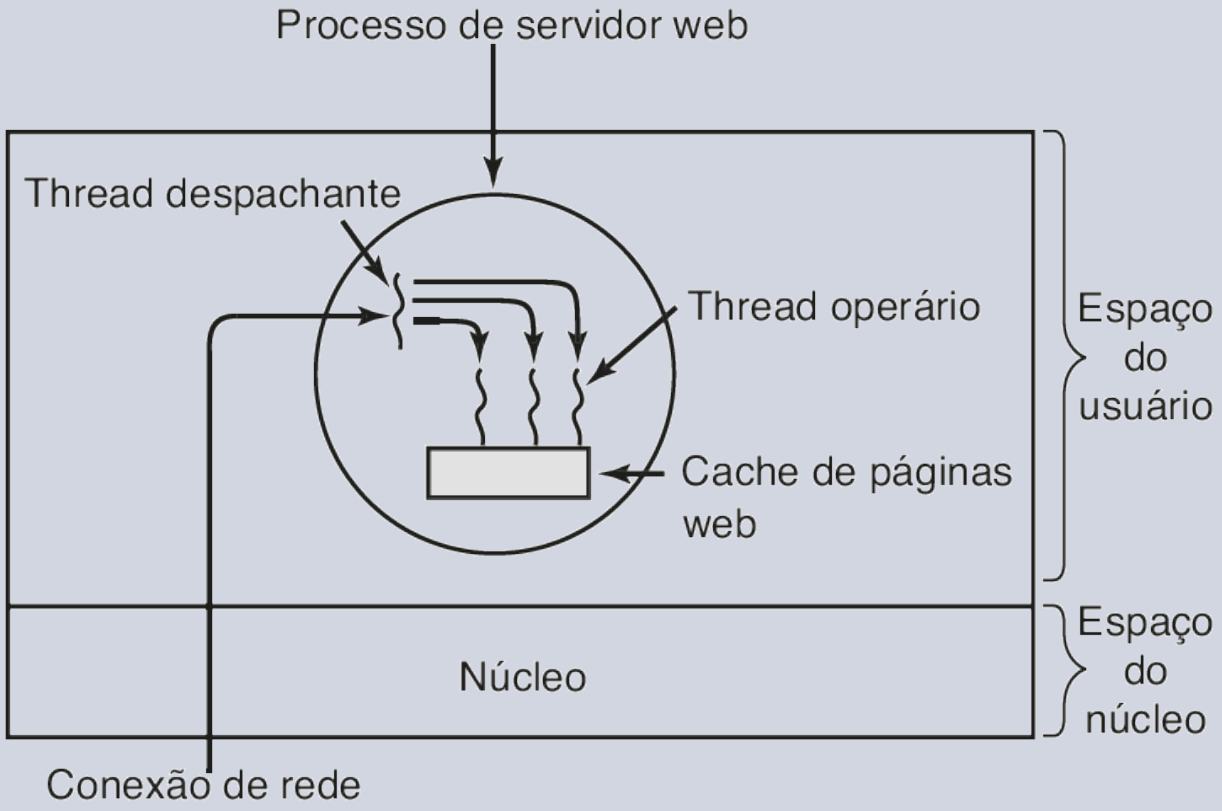
Threads

- Para algumas aplicações é útil ter múltiplos threads de controle dentro de um único processo.
- Esses threads são escalonados independentemente e cada um tem sua própria pilha, mas todos os threads em um processo compartilham de um espaço de endereçamento comum.
- Threads podem ser implementados no espaço do usuário ou no núcleo.

Um processador de texto com três threads

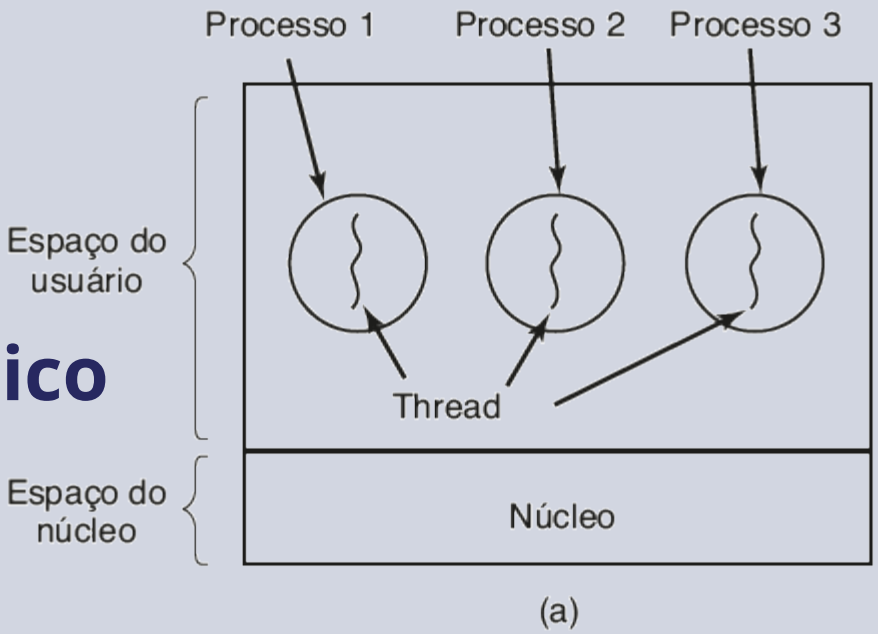


Um servidor web multithread

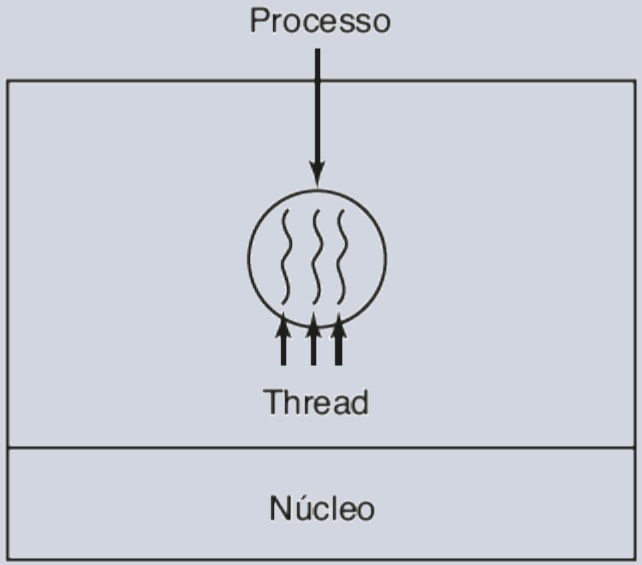


O modelo de thread clássico

- (A) vemos três processos tradicionais. Cada processo tem seu próprio espaço de endereçamento e um único thread de controle. Cada um deles opera em um espaço de endereçamento diferente.
- (B) vemos um único processo com três threads de controle. Embora em ambos os casos tenhamos três threads. Todos os três compartilham o mesmo espaço de endereçamento.



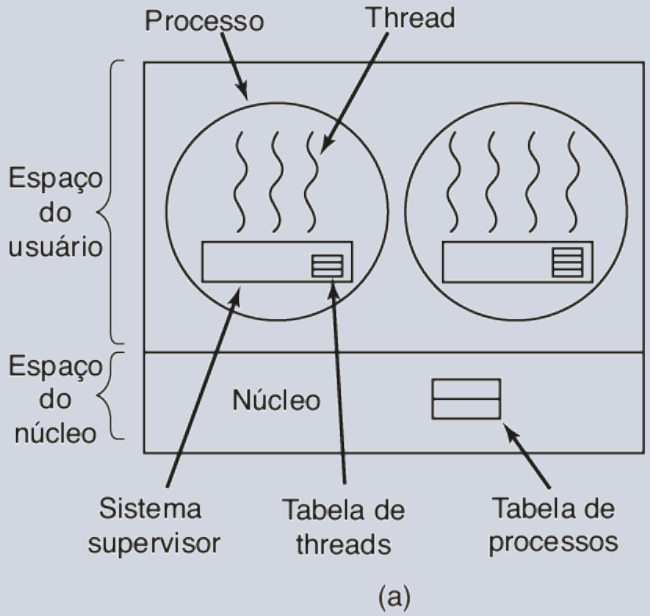
(a)

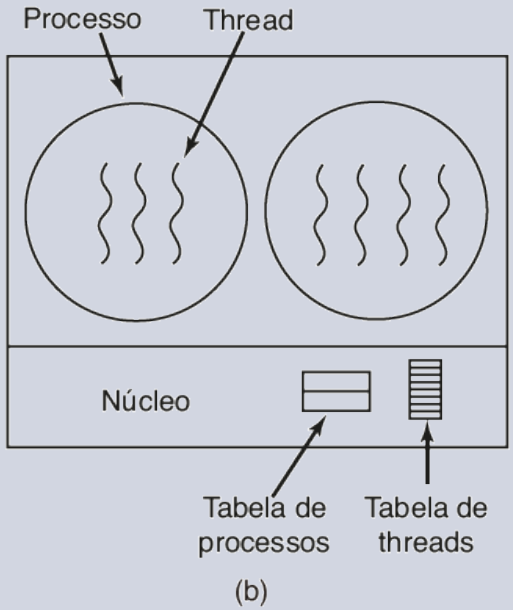


(b)

Implementando threads no espaço e no núcleo

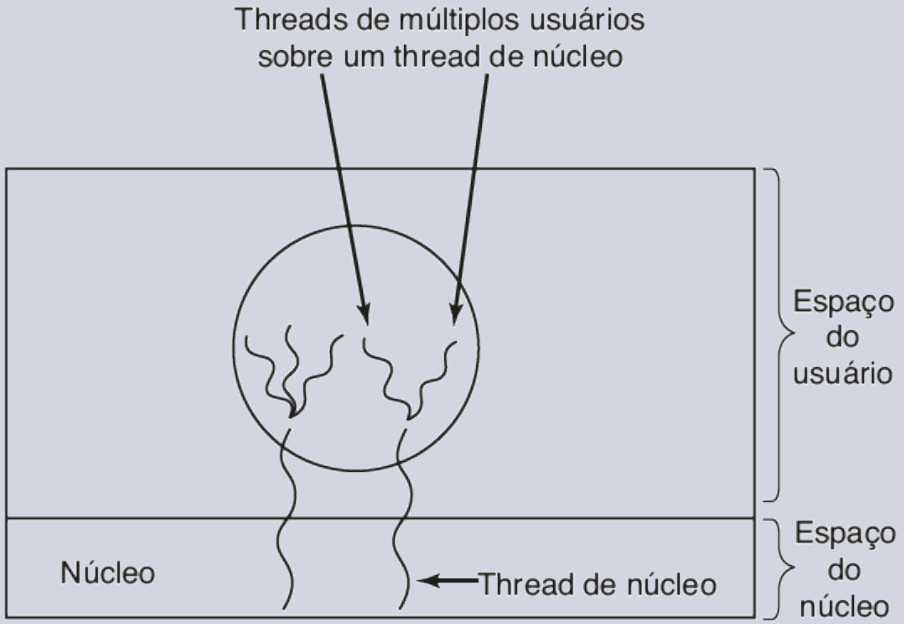
- Há dois lugares principais para implementar threads: no **espaço do usuário (figura a)** e no **núcleo (figura b)**. A escolha é um pouco controversa, e uma implementação híbrida também é possível.





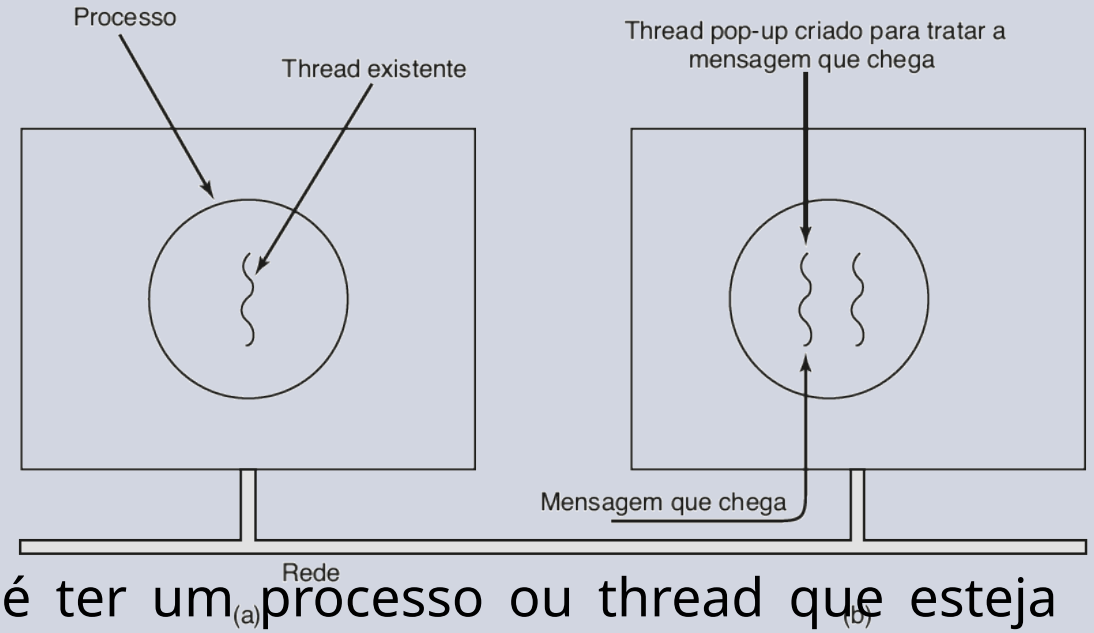
Implementações híbridas

- Várias maneiras foram investigadas para tentar combinar as vantagens de threads de usuário com threads de núcleo. Uma maneira é usar threads de núcleo e então multiplexar os de usuário em alguns ou todos eles, como mostrado na figura:



Threads pop-up

- Threads costumam ser úteis em sistemas distribuídos. Um exemplo importante é como mensagens que chegam são tratadas.

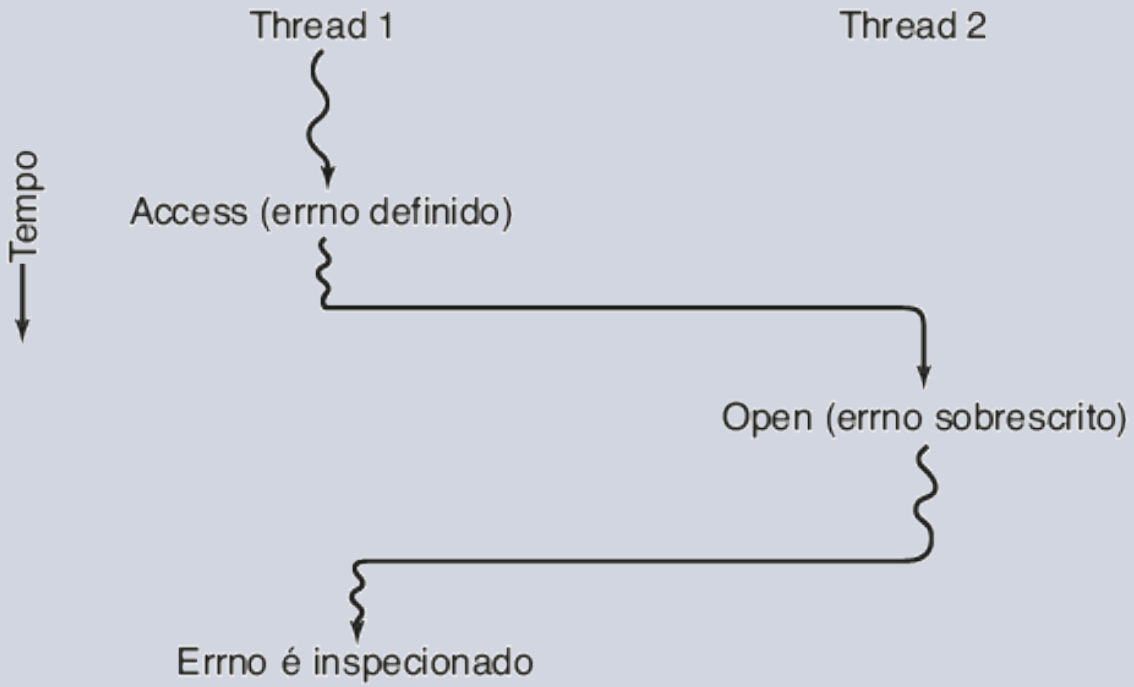


- A abordagem tradicional é ter um processo ou thread que esteja bloqueado em uma chamada de sistema *receive* esperando pela mensagem que chega. Quando uma mensagem chega, ela é aceita, aberta, seu conteúdo examinado e processada. No entanto, uma abordagem completamente diferente também é possível, na qual a chegada de uma mensagem faz o sistema criar um novo thread para lidar com a mensagem. Esse thread é chamado de **thread pop-up.**

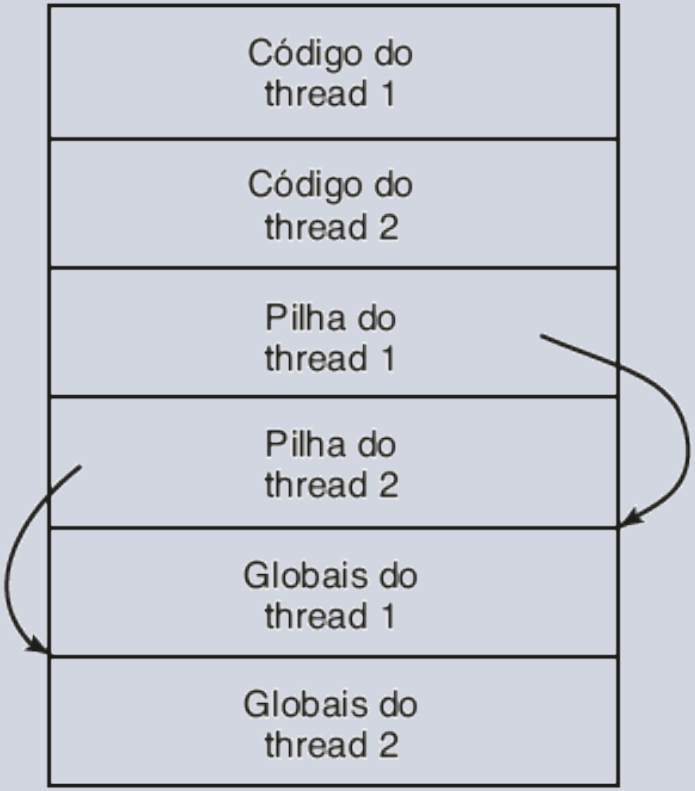
Convertendo código de um thread em código multithread

- Muitos programas existentes foram escritos para processos monothread. Convertê-los para multithreading é muito mais complicado do que pode parecer em um primeiro momento.
- O código de um thread em geral consiste em múltiplas rotinas, exatamente como um processo. Essas rotinas podem ter variáveis locais, variáveis globais e parâmetros. Variáveis locais e de parâmetros não causam problema algum, mas variáveis que são globais para um thread, mas não globais para o programa inteiro, são um problema. Essas são variáveis que são globais no sentido de que muitos procedimentos dentro do thread as usam (como poderiam usar qualquer variável global), mas outros threads devem logicamente deixá-las sozinhas.

Conflitos entre threads sobre o uso de uma variável global



Threads podem ter variáveis globais individuais

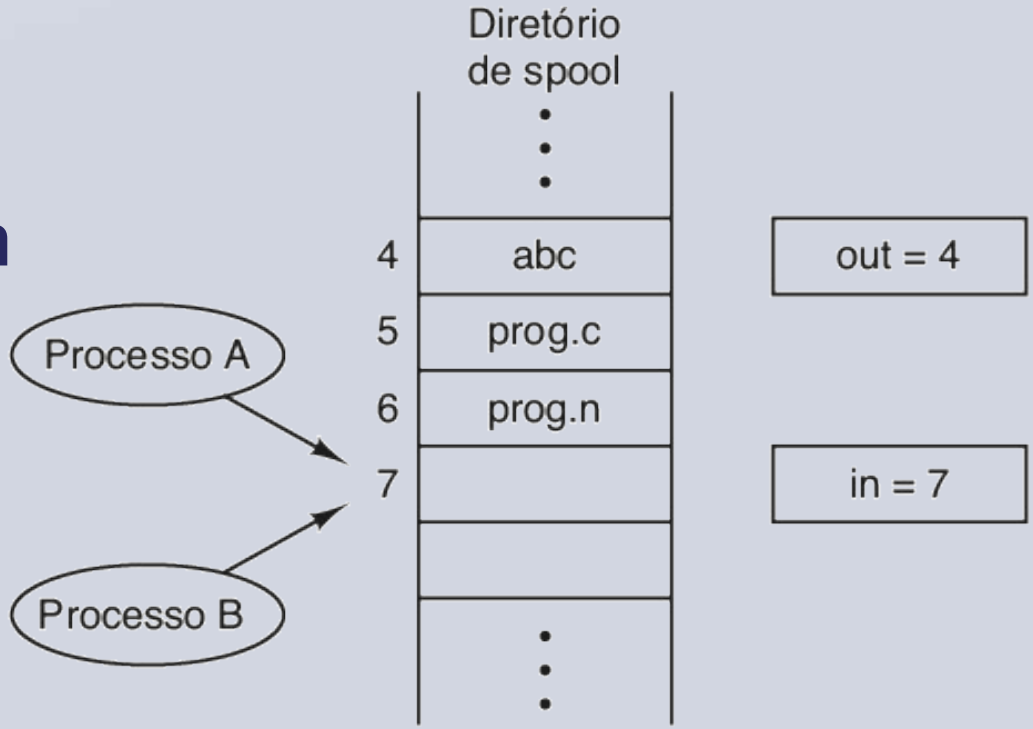


Comunicação entre processos

- Processos podem comunicar-se uns com os outros usando primitivas de comunicação entre processos, por exemplo, semáforos, monitores ou mensagens.
- Essas primitivas são usadas para assegurar que jamais dois processos estejam em suas regiões críticas ao mesmo tempo, uma situação que leva ao caos.
- Um processo pode estar sendo executado, ser executável, ou bloqueado, e pode mudar de estado quando ele ou outro executar uma das primitivas de comunicação entre processos. A comunicação entre threads é similar.

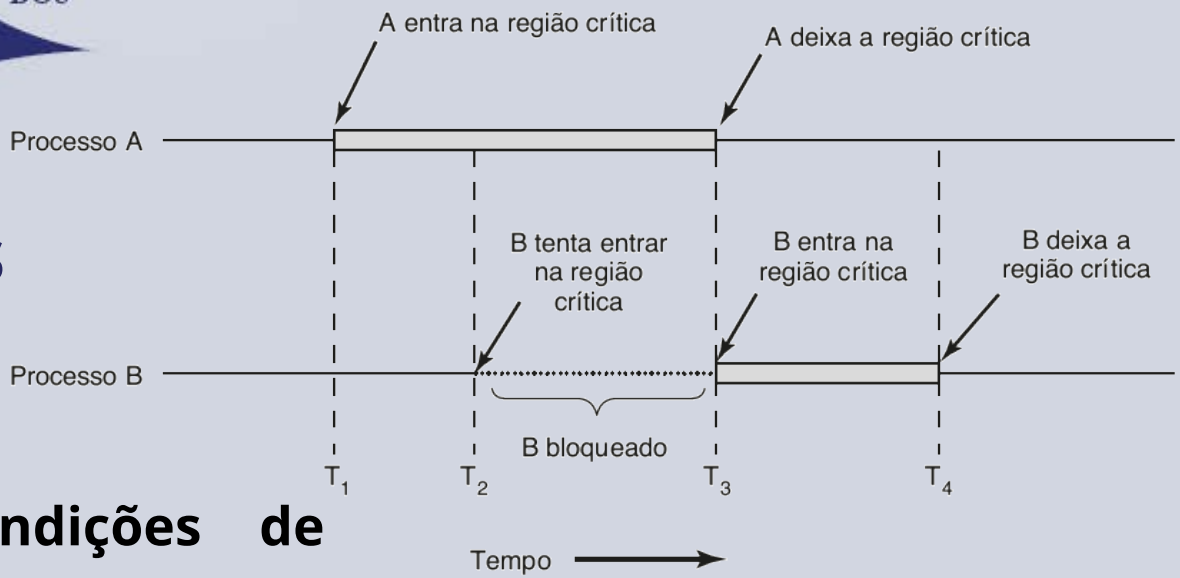
Condições de corrida

- Em alguns sistemas operacionais, processos que estão trabalhando juntos podem compartilhar de alguma memória comum que cada um pode ler e escrever.



Veja na figura dois processos querem acessar a memória compartilhada ao mesmo tempo.

Regiões críticas

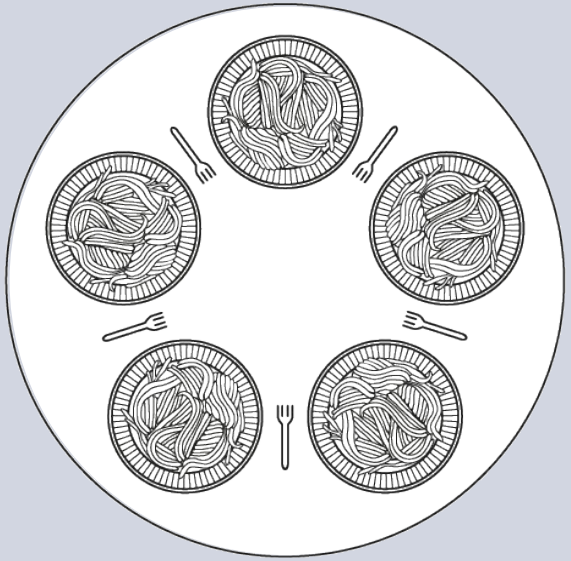


Como evitar as condições de corrida? A chave para evitar problemas aqui e em muitas outras situações envolvendo memória compartilhada, arquivos compartilhados e tudo o mais compartilhado é encontrar alguma maneira de proibir mais de um processo de ler e escrever os dados compartilhados ao mesmo tempo.

Colocando a questão em outras palavras, o que precisamos é de **exclusão mútua**, isto é, alguma maneira de se certificar de que se um processo está usando um arquivo ou variável compartilhados, os outros serão impedidos de realizar a mesma

Escalonamento

- Quando um computador é multiprogramado, ele frequentemente tem múltiplos processos ou threads competindo pela CPU ao mesmo tempo. Essa situação ocorre sempre que dois ou mais deles estão simultaneamente no estado pronto.
- Se apenas uma CPU está disponível, uma escolha precisa ser feita sobre qual processo será executado em seguida. A parte do sistema operacional que faz a escolha é chamada de **escalonador**, e o algoritmo que ele usa é chamado de **algoritmo de escalonamento**.



Problemas clássicos de IPC

O problema do jantar dos filósofos

- Cinco filósofos estão sentados em torno de uma mesa circular. Cada filósofo tem um prato de espaguete. O espaguete é tão escorregadio que um filósofo precisa de dois garfos para comê-lo. Entre cada par de pratos há um garfo.
- Quando um filósofo fica suficientemente faminto, ele tenta pegar seus garfos à esquerda e à direita, um de cada vez, não importa a ordem. Se for bem-sucedido em pegar dois garfos, ele come por um tempo, então larga os garfos e continua a pensar.
- A questão fundamental é: você consegue escrever um programa para cada filósofo que faça o que deve fazer e jamais fique travado?

Problemas clássicos de IPC

O problema dos leitores e escritores

- Imagine, por exemplo, um sistema de reservas de uma companhia aérea, com muitos processos competindo entre si desejando ler e escrever.
- É aceitável ter múltiplos processos lendo o banco de dados ao mesmo tempo, mas se um processo está atualizando (escrevendo) o banco de dados, nenhum outro pode ter acesso, nem mesmo os leitores.
- A questão é: como programar leitores e escritores?

Pesquisas sobre processos e threads

- Como um todo, processos, threads e escalonamento, não são mais os tópicos quentes de pesquisa que já foram um dia. A pesquisa seguiu para tópicos como **gerenciamento de energia**, **virtualização**, **nuvens** e **segurança**.
- Uma área de pesquisa particularmente ativa lida com a gravação e a reprodução da execução de um processo (VIENNOT et al., 2013). A reprodução ajuda os desenvolvedores a procurar erros difíceis de serem encontrados e especialistas em segurança a investigar incidentes.
- De modo similar, questões de segurança. Muitos incidentes demonstraram que os usuários precisam de uma proteção melhor contra agressores (e, ocasionalmente, contra si mesmos). Uma abordagem é controlar e restringir com cuidado os fluxos de

Pesquisas sobre processos e threads

- O escalonamento de dispositivos móveis em busca da eficiência energética (YUAN e NAHRSTEDT, 2006), escalonamento com tecnologia *hyperthreading* (BULPIN e PRATT, 2005) e escalonamento *bias-aware* (KOUFATY, 2010).
- Com cada vez mais computação em smartphones com restrições de energia, alguns pesquisadores propõem migrar o processo para um servidor mais potente na nuvem, quando isso for útil (GORDON et al, 2012).
- No entanto, poucos projetistas de sistemas andam preocupados com a falta de um algoritmo de escalonamento de threads decente, então esse tipo de pesquisa parece ser mais um interesse de pesquisadores do que uma demanda de projetistas.