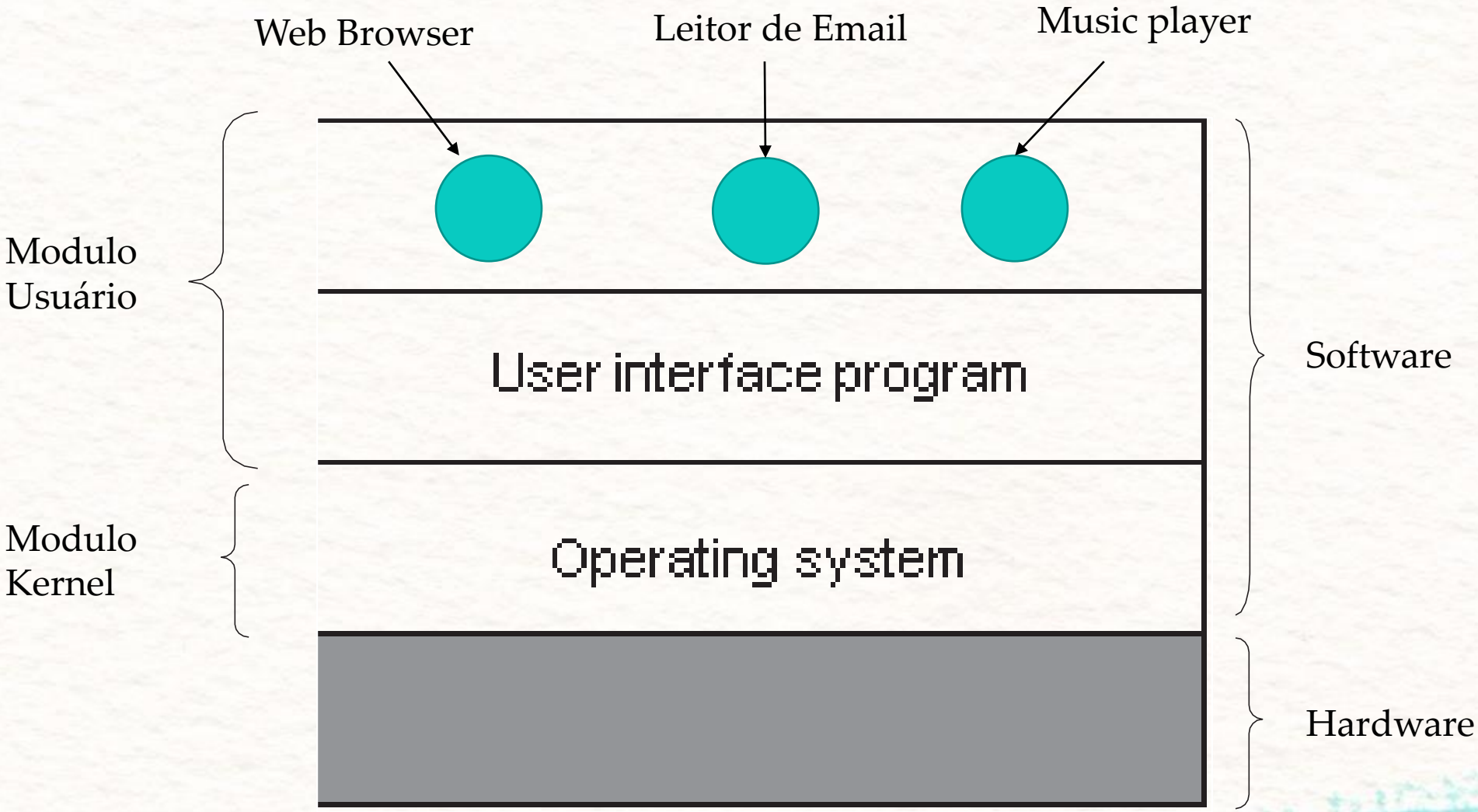
Fundamentos de Sistemas Operacionais

Prof. Me. Paulo Sérgio Germano

Onde o sistema operacional se encaixa



Componentes básicos de um S.O.

Um sistema operacional multiprogramado é composto basicamente por:

- Interpretador de comandos (Shell);
- Escalador de tarefas (Scheduler);
- Gerenciador de Processos.
- Gerenciamento de Memória;
- Gerenciador de Arquivos,
- Rotinas de Controle;
- Comutador de Contexto;
- Controle de Periféricos;

Interfaces com o S.O.

- SHELL – Linha de comando ou interfaces pouco elaboradas de pouco conteúdo gráfico e que permite execução de chamadas diretamente na linha de comando "prompt". (Linux, Unix, OS X)

- GUI (Grafic User Interface) – Interface mais intuitiva e gráfica, onde o uso da linha de comando é raro ou inexistente. (Windows, IOS, Android)

UNIX	Win32	Description
fork	CreateProcess	Create a new process
waitpid	WaitForSingleObject	Can wait for a process to exit
execve	(none)	CreateProcess = fork + execve
exit	ExitProcess	Terminate execution
open	CreateFile	Create a file or open an existing file
close	CloseHandle	Close a file
read	ReadFile	Read data from a file
write	WriteFile	Write data to a file
lseek	SetFilePointer	Move the file pointer
stat	GetFileAttributesEx	Get various file attributes
mkdir	CreateDirectory	Create a new directory
rmdir	RemoveDirectory	Remove an empty directory
link	(none)	Win32 does not support links
unlink	DeleteFile	Destroy an existing file
mount	(none)	Win32 does not support mount
umount	(none)	Win32 does not support mount, so no umount
chdir	SetCurrentDirectory	Change the current working directory
chmod	(none)	Win32 does not support security (although NT does)
kill	(none)	Win32 does not support signals
time	GetLocalTime	Get the current time

Modelo de Processos e Threads

Todos os computadores modernos costumam fazer várias coisas ao mesmo tempo. As pessoas acostumadas a trabalhar com computadores podem não estar totalmente cientes desse fato, portanto, alguns exemplos podem tornar o ponto mais claro. Primeiro considere um servidor da Web. Solicitações vêm de todas as partes pedindo páginas da Web. Quando uma solicitação chega, o servidor verifica se a página necessária está no cache. Se for, é enviado de volta; se não for, uma solicitação de disco é iniciada para buscá-lo. No entanto, do ponto de vista da CPU, as solicitações de disco levam a eternidade. Enquanto aguarda a conclusão de uma solicitação de disco, muitos outros pedidos podem vir. Se houver vários discos presentes, alguns ou todos os mais recentes poderão ser disparados para outros discos muito antes da primeira solicitação ser atendida. Claramente, alguma forma é necessária para modelar e controlar essa simultaneidade. Processos (e especialmente threads) podem ajudar aqui.

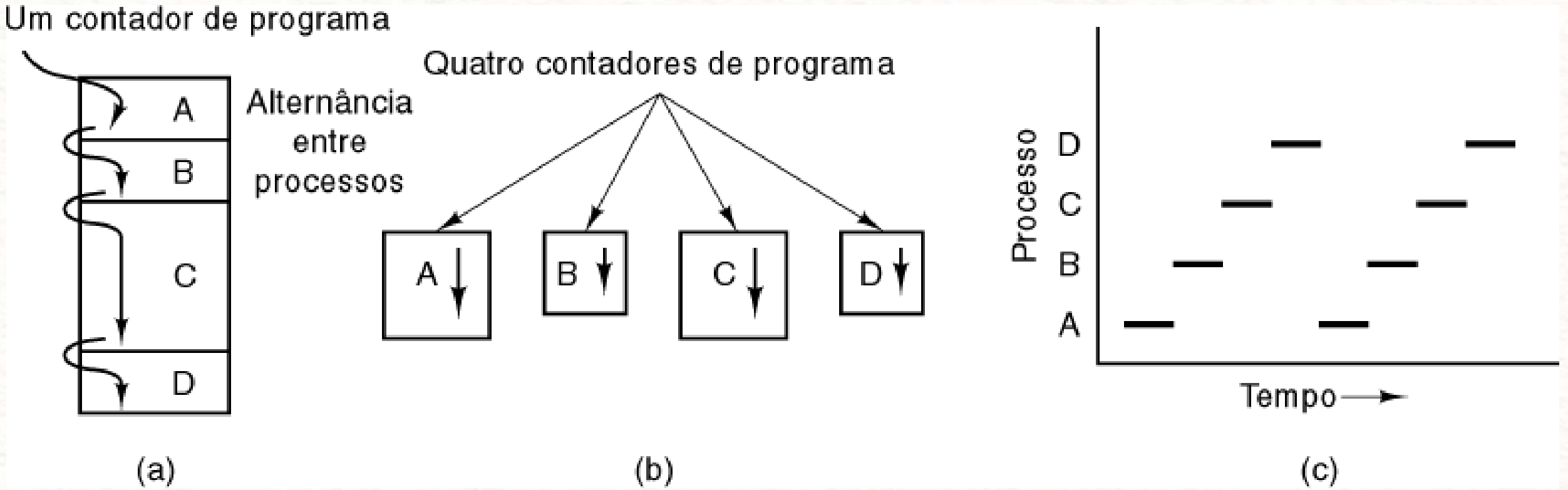
Modelo de Processos e Threads

- Multiprogramação

- Como vemos a multiprogramação:
  - processos sequenciais
  - Independentes

- Ocupação real do processador

Modelo de Processos e Threads



Criação de Processos

Os sistemas operacionais precisam, de alguma forma, criar processos. Em sistemas muito simples, ou em sistemas projetados para executar somente uma única aplicação (por exemplo, o controlador em um forno de micro-ondas), pode ser possível ter todos os processos que serão necessários quando o sistema chegar acima. Em sistemas de uso geral, no entanto, é necessário algum modo para criar e finalizar processos, conforme necessário, durante a operação. Vamos agora olhar para alguns dos problemas.
Quatro eventos principais fazem com que processos sejam criados:

1. Inicialização do sistema.
2. Execução de uma chamada de sistema de criação de processo por um “running process”.
3. Uma solicitação do usuário para criar um novo processo.
4. Início de um trabalho em lote (batch job)

Quando um sistema é inicializado, normalmente vários processos são criados. Alguns deles são processos em primeiro plano, isto é, processos que interagem com usuários (humanos) e executam trabalho para eles. Outros são executados em segundo plano e não estão associados a usuários específicos, mas possuem alguma função específica. Por exemplo, um processo em segundo plano pode ser projetado para aceitar e-mails recebidos, dormindo a maior parte do dia, mas de repente dando um salto quando os e-mails chegam. Outro processo em segundo plano pode ser projetado para aceitar solicitações recebidas de páginas da Web hospedadas nessa máquina, ativando quando uma solicitação chegar para atender à solicitação. Os processos que ficam em segundo plano para lidar com alguma atividade, como e-mail, páginas da Web, notícias, impressão e assim por diante, são chamados de daemons. Grandes sistemas comumente tem dezenas deles. No UNIX , o programa ps pode ser usado para listar os processos. No Windows, o gerenciador de tarefas pode ser usado.

Conclusão de Processos

Depois que um processo é criado, ele começa a ser executado e faz o que quer que seja seu trabalho. No entanto, nada dura para sempre, nem mesmo processos. Mais cedo ou mais tarde, o novo processo terminará, geralmente devido a uma das seguintes condições:

1. Saída normal (voluntária).

2. Saída de erro (voluntária).

3. Erro fatal (involuntário).

4. “Killed” Encerramento forçado por outro processo (involuntário).

A maioria dos processos termina porque eles fizeram o seu trabalho. Quando um compilador compilou o programa dado a ele, o compilador executa uma chamada de sistema para informar ao sistema operacional que está terminado. Esta chamada é “Exit” no UNIX e “ExitProcess” no Windows. Programas “Screen-oriented“ também suportam o encerramento voluntário. Processadores de texto, navegadores da Internet e programas semelhantes sempre têm um ícone ou item de menu no qual o usuário pode clicar para informar o processo para remover quaisquer arquivos temporários abertos e, em seguida, encerra-los.

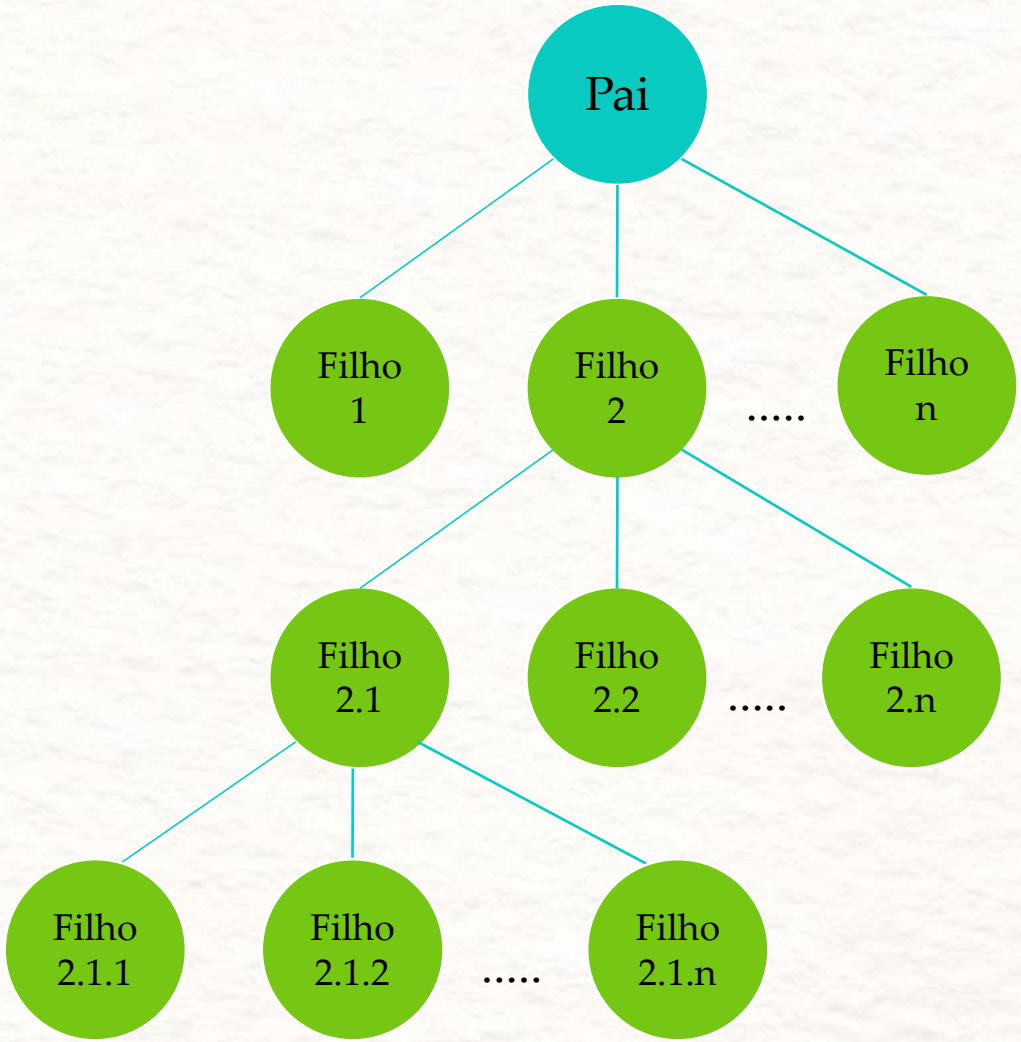
Hierarquia dos Processos

Em alguns sistemas, quando um processo cria outro processo, o processo pai e o processo filho continuam associados de determinadas maneiras. O processo filho pode criar mais processos, formando uma hierarquia de processos. Observe que, diferentemente de plantas e animais que usam a reprodução sexual, um processo tem apenas um dos pais (mas zero, um, dois ou mais filhos).

Ex.: O UNIX se inicializa quando é iniciado, logo após o computador ser inicializado. Um processo especial, chamado init, está presente na imagem de inicialização. Quando ele começa a funcionar, ele lê um arquivo informando quantos terminais existem. Então ele bifurca um novo processo por terminal. Esses processos esperam que alguém efetue login. Se um login for bem-sucedido, o processo de login executará um shell para aceitar comandos. Esses comandos podem iniciar mais processos e assim por diante. Assim, todos os processos em todo o sistema pertencem a uma única árvore, com init na raiz.

Por outro lado, o Windows não possui um conceito de hierarquia de processos. Todos os processos são iguais. A única dica de uma hierarquia de processo é que, quando um processo é criado, o pai recebe um token especial (chamado de handle) que pode ser usado para controlar o filho. No entanto, é livre para passar esse token para algum outro processo, invalidando, assim, a hierarquia. Processos no UNIX não podem deserdar seus filhos.

Hierarquia dos Processos



Estado dos Processos

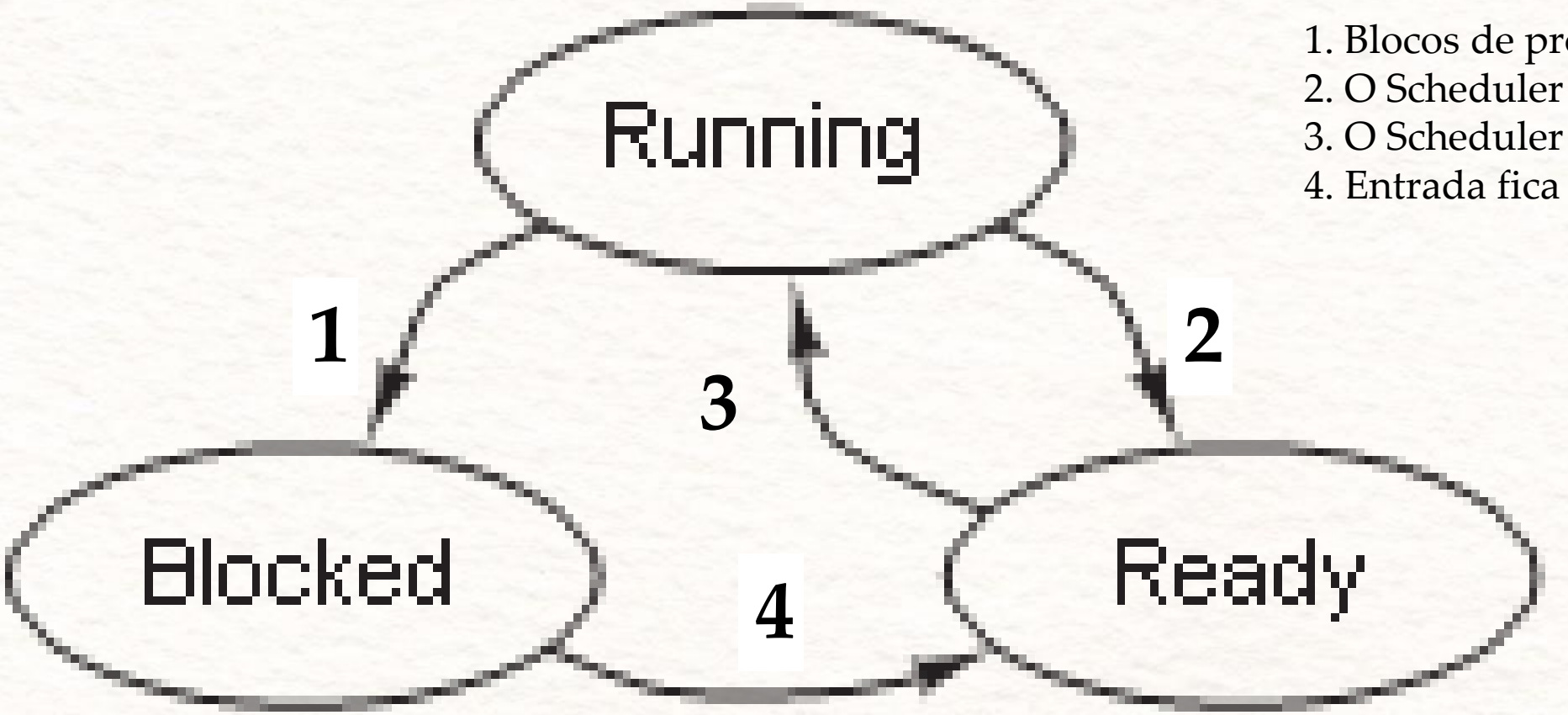
Embora cada processo seja uma entidade independente, com seu próprio contador de programa e estado interno, os processos geralmente precisam interagir com outros processos. Um processo pode gerar alguma saída que outro processo usa como entrada.

Quando um processo é bloqueado, ele faz isso porque, logicamente, ele não pode continuar, normalmente porque está aguardando uma entrada que ainda não está disponível. Também é possível que um processo que esteja conceitualmente pronto e seja capaz de rodar seja interrompido, porque o sistema operacional decidiu alocar a CPU para outro processo por algum tempo. Essas duas condições são completamente diferentes. No primeiro caso, a suspensão é inerente ao problema (você não pode processar a linha de comando do usuário até que ela tenha sido digitada). No segundo caso, é uma tecnicidade do sistema (não há CPUs suficientes para dar a cada processo seu próprio processador privado).

São três os estados que um processo pode estar:

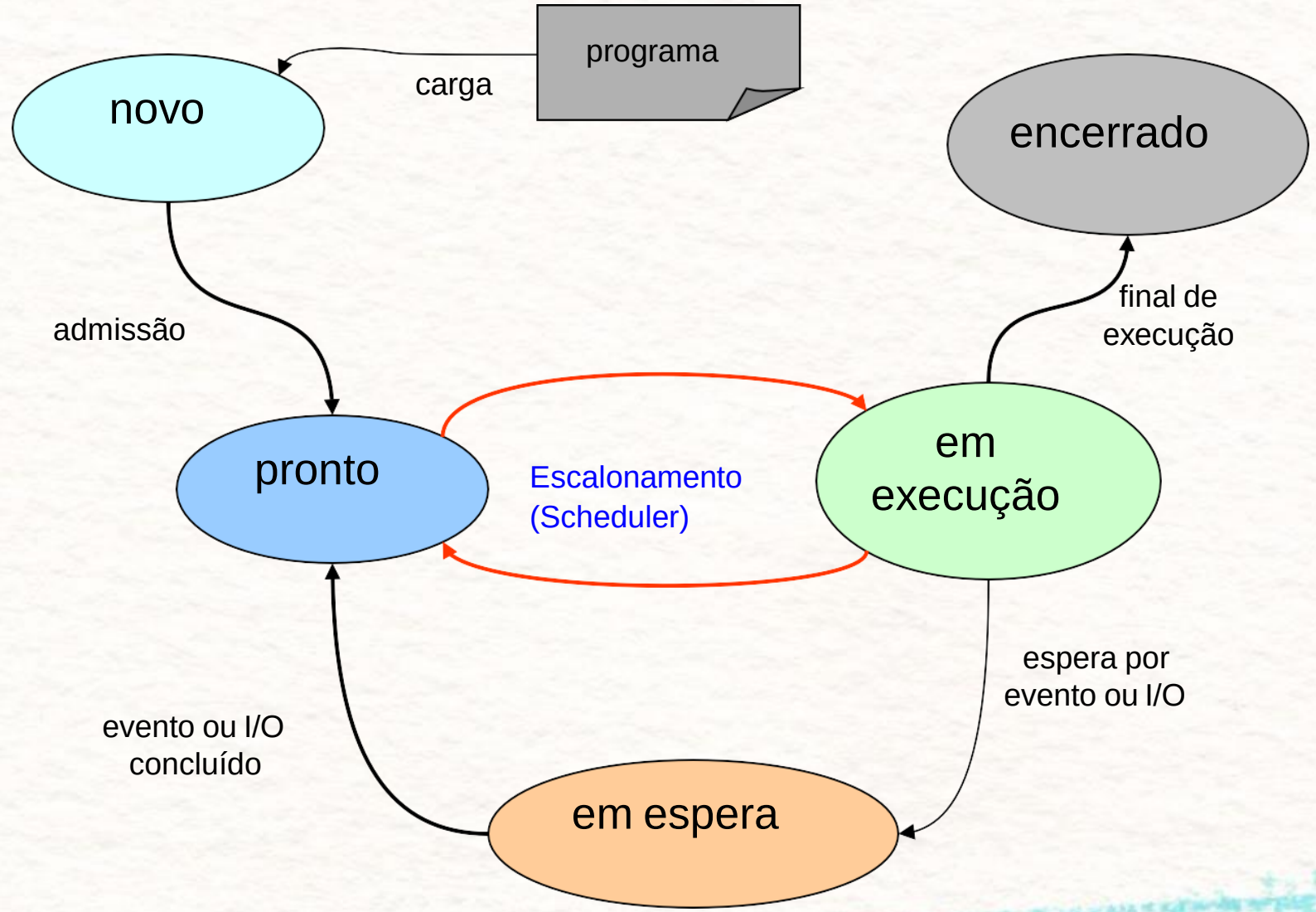
1. **Running** (na verdade usando a CPU naquele instante).
2. **Ready** (executável; temporariamente interrompido para permitir que outro processo seja executado).
3. **Blocked** (não é possível executar até que algum evento externo aconteça).

Transição de estados dos processos



1. Blocos de processo para entrada
2. O Scheduler escolhe outro processo
3. O Scheduler escolhe este processo
4. Entrada fica disponível

Transição de estados dos processos



Implementação de Processos

Para implementar o modelo de processo, o sistema operacional mantém uma tabela (uma matriz de estruturas), chamada de tabela de processos (bloco de processos), com uma entrada por processo. Esta entrada contém informações importantes sobre o estado do processo, incluindo seu contador de programa, ponteiro de pilha, alocação de memória, o status de seus arquivos abertos, suas informações contábeis e de agendamento e tudo mais sobre o processo que deve ser salvo quando o processo é comutado da execução para o estado pronto ou bloqueado, para que possa ser reiniciado mais tarde, como se nunca tivesse sido interrompido.

Todas as interrupções começam salvando os registradores, geralmente na entrada da tabela de processos para o processo atual. Em seguida, a informação inserida na pilha pela interrupção é removida e o ponteiro da pilha é definido para apontar para uma pilha temporária usada pelo manipulador de processo.

Quando ele fez seu trabalho, possivelmente tornando algum processo pronto, o agendador (scheduler) é chamado para ver quem deve ser executado em seguida.

Um processo pode ser interrompido milhares de vezes durante sua execução, mas a idéia-chave é que, após cada interrupção, o processo interrompido retorna exatamente ao mesmo estado em que se encontrava antes da interrupção.

Tabela de Processos

Process management	Memory management	File management
Registers	Pointer to text segment info	Root directory
Program counter	Pointer to data segment info	Working directory
Program status word	Pointer to stack segment info	File descriptors
Stack pointer		User ID
Process state		Group ID
Priority		
Scheduling parameters		
Process ID		
Parent process		
Process group		
Signals		
Time when process started		
CPU time used		
Children’s CPU time		
Time of next alarm		

Threads

Threads são fluxos de execução (linha de controle) que rodam dentro de um processo. Em processos tradicionais, há uma única linha de controle e um único contador de programa. Porém, alguns Sistemas Operacionais fornecem suporte para múltiplas linhas de controle dentro de um processo (sistemas *multithread*).

Ter múltiplas linhas de controle ou *threads* executando em paralelo em um processo equivale a ter múltiplos processos executando em paralelo em um único computador. Um exemplo tradicional do uso de múltiplas *thread* seria um navegador *web*, no qual pode ter uma *thread* para exigir imagens ou texto enquanto outro *thread* recupera dados de uma rede. É importante destacar que as *threads* existem no interior de um processo e compartilham entre elas os recursos do processo, como o espaço de endereçamento (código e dados).

Justificativa para o uso de Threads

A principal razão para ter threads é que, em muitos aplicativos, várias atividades estão acontecendo ao mesmo tempo. Algumas delas podem bloquear de tempos em tempos. Ao decompor esse aplicativo em vários segmentos sequenciais que são executados em quase paralelo, o modelo de programação se torna mais simples.

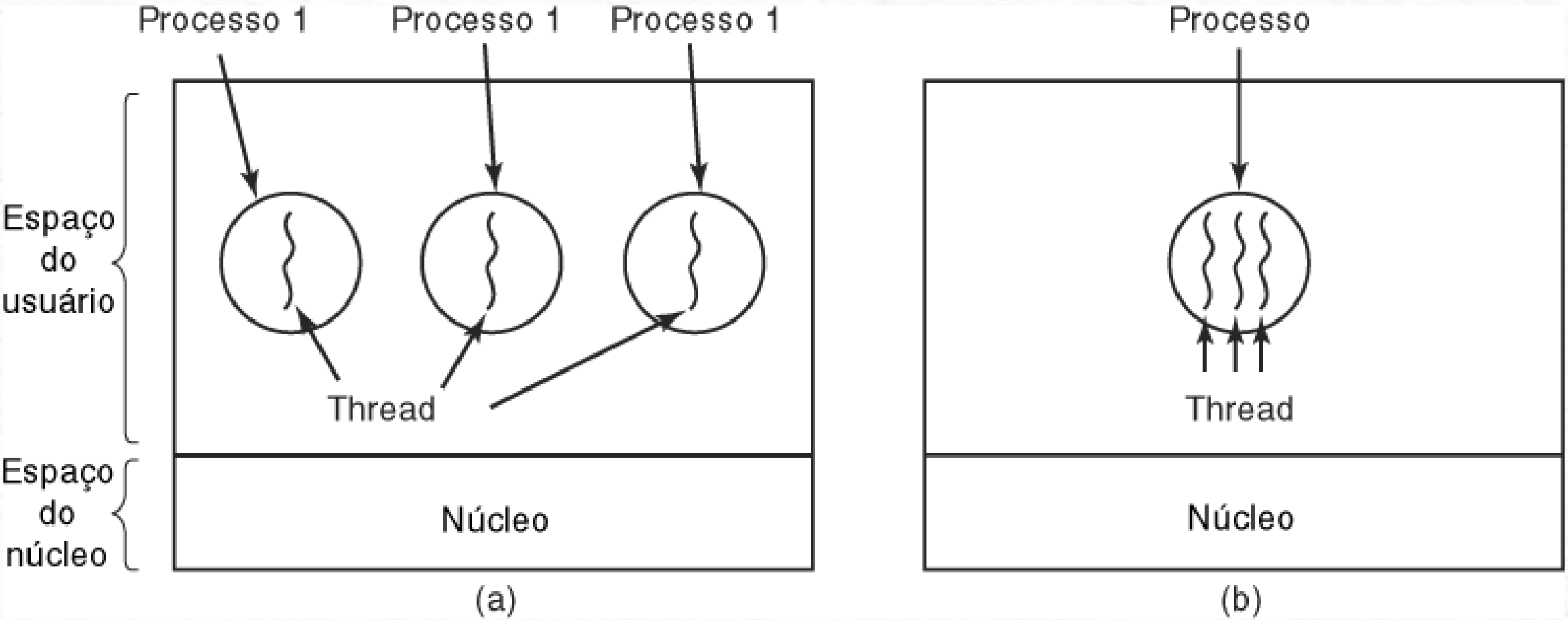
Somente agora, com os threads, adicionamos um novo elemento: a capacidade de as entidades paralelas compartilharem um espaço de endereço e todos os seus dados entre si. Essa capacidade é essencial para determinados aplicativos, e é por isso que ter vários processos (com seus espaços de endereço separados) não funcionará.

Um segundo argumento para ter threads é que, como eles são mais leves que os processos, eles são mais fáceis (ou seja, mais rápidos) de criar e destruir do que os processos. Em muitos sistemas, a criação de um encadeamento é 10 a 100 vezes mais rápida do que a criação de um processo. Quando o número de threads necessários muda dinamicamente e rapidamente, essa propriedade é útil ter.

Uma terceira razão para ter threads também é um argumento de desempenho. Os threads não geram ganho de desempenho quando todos eles estão ligados à CPU, mas quando há computação substancial e também E / S substancial, ter threads permite que essas atividades se sobreponham, acelerando assim o aplicativo.

Finalmente, os threads são úteis em sistemas com múltiplas CPUs, onde o paralelismo real é possível.

Modelo Clássico de Thread



Três processos cada um com um thread

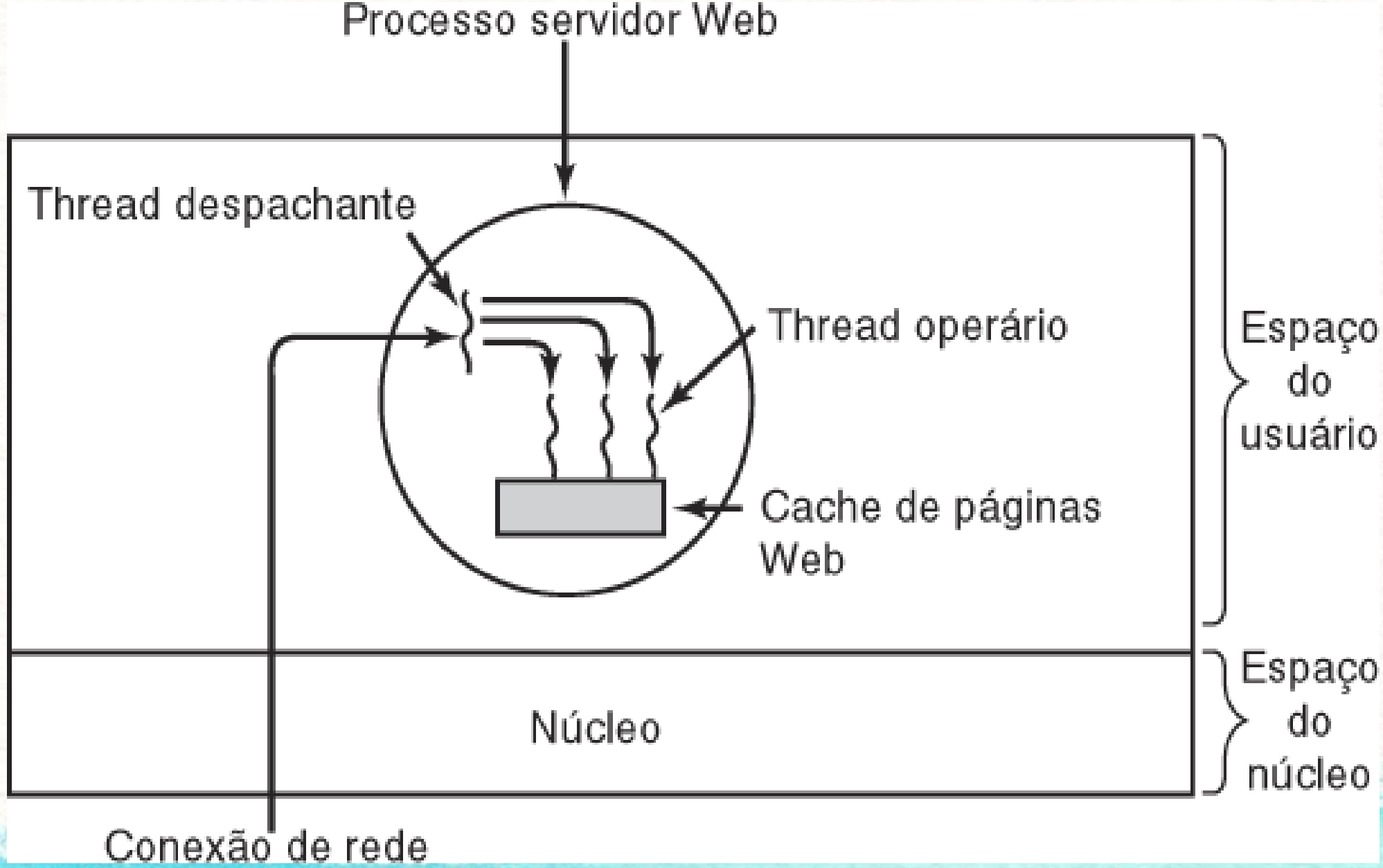
Um processo com três threads

Modelo de Thread

- Contexto do Processo
  - Itens compartilhados por todas as threads em um processo
- Contexto da Thread
  - Itens privativos de cada thread

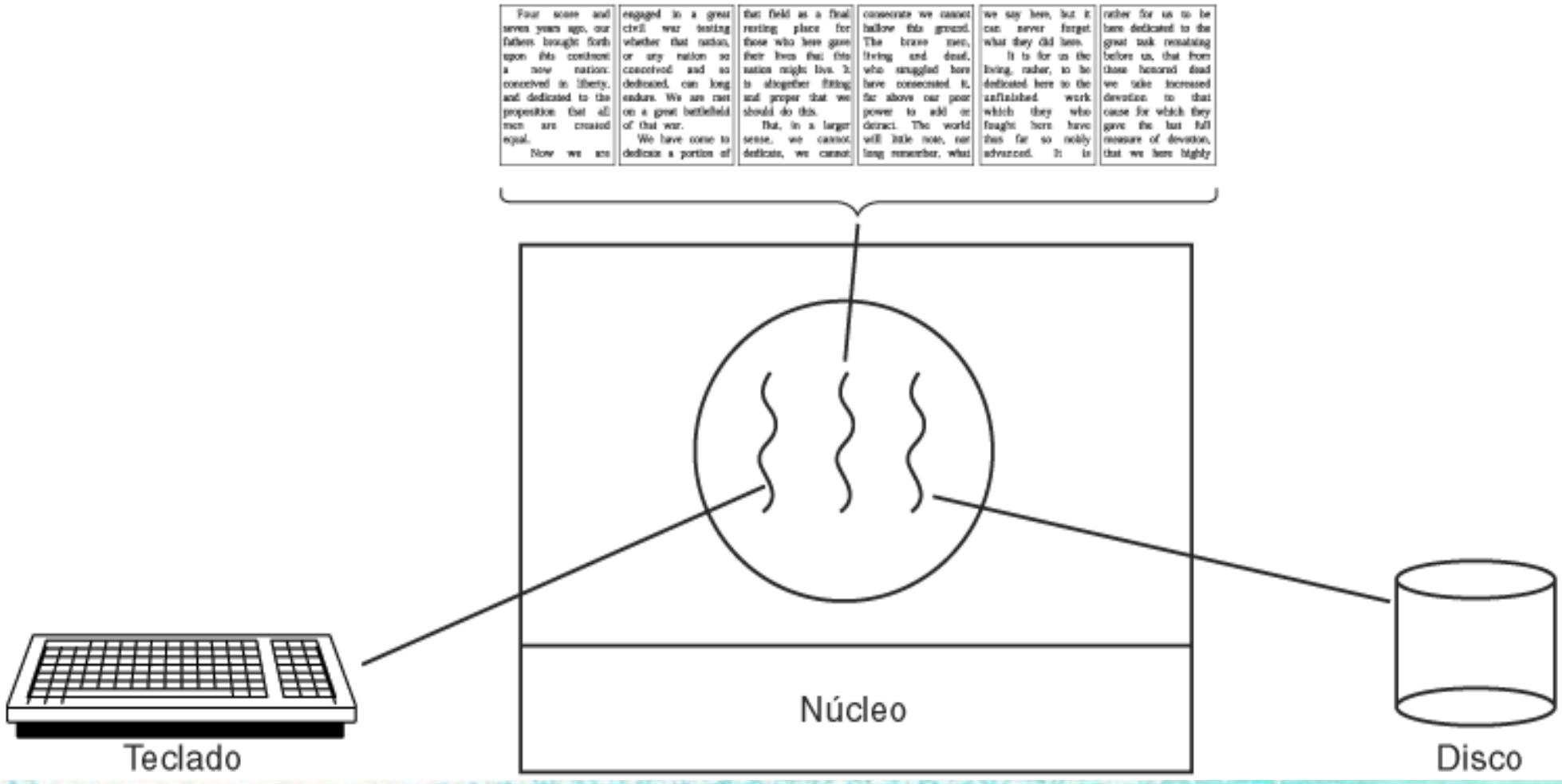
Itens por processo	Itens por thread
Espaço de endereçamento	Contador de programa
Variáveis globais	Registradores
Arquivos abertos	Pilha
Processos filhos	Estado
Alarmes pendentes	
Sinais e tratadores de sinais	
Informação de contabilidade	

Exemplos de Threads



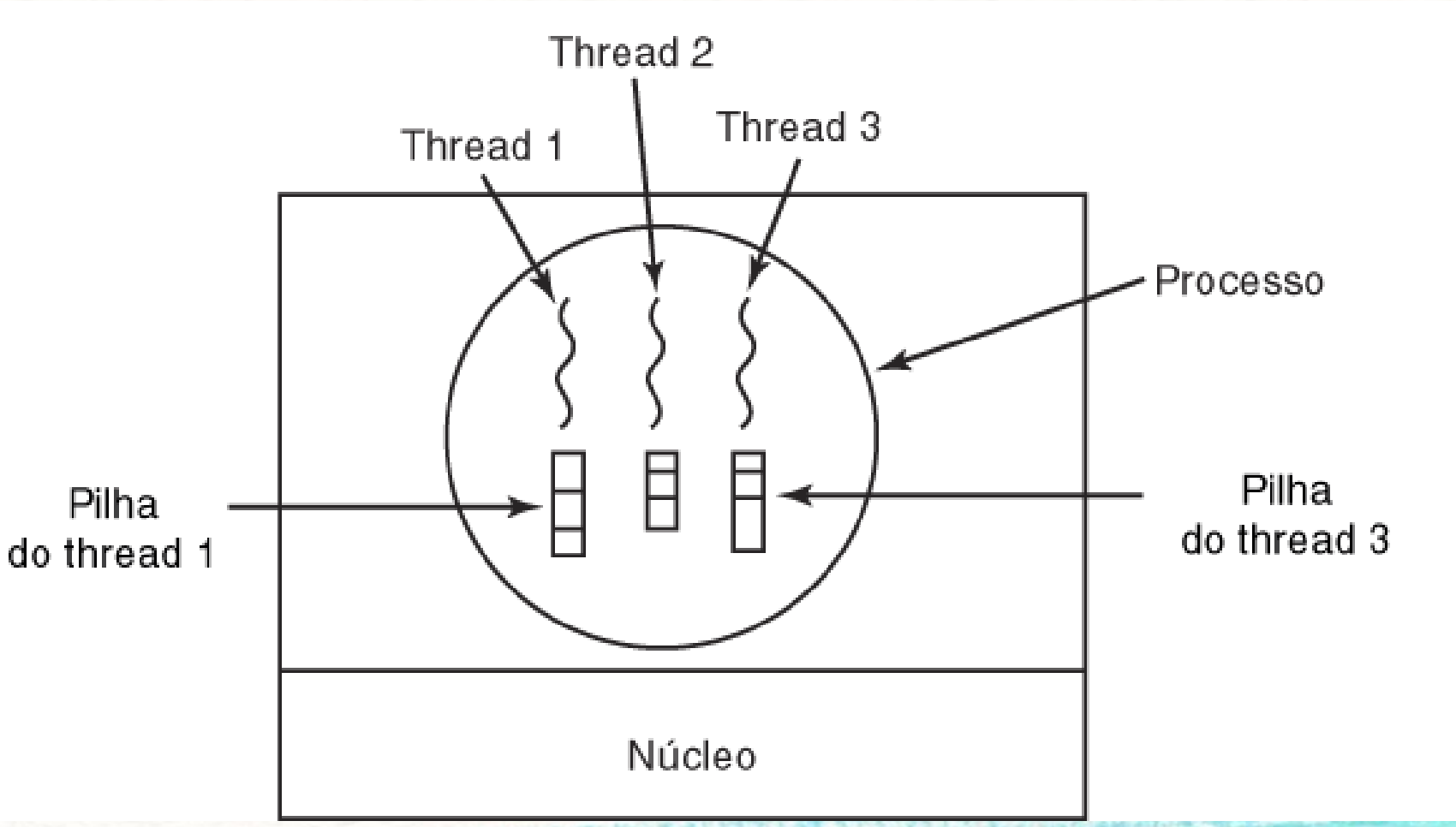
Exemplos de Threads

- Um processador de texto com três threads



Pilhas de Threads

Cada thread geralmente chama diferentes procedimentos e, portanto, tem um histórico de execução diferente. É por isso que **cada segmento precisa de sua própria pilha**.



Implementação de Threads

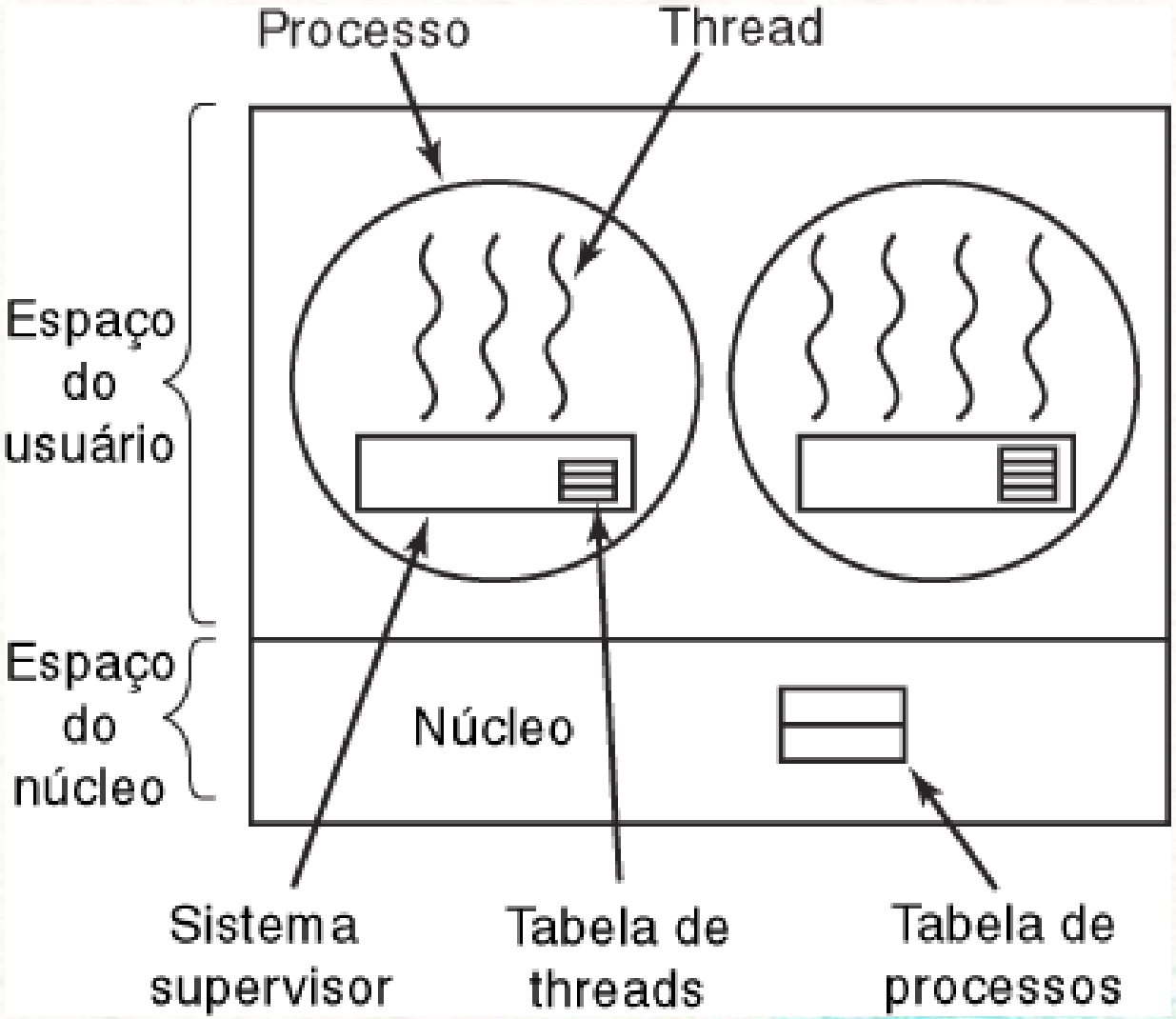
No espaço do Usuário

Vantagem

- Velocidade

Desvantagem

- Gerenciamento de threads no espaço do usuário
- O Bloqueio de uma thread bloqueia todo o processo



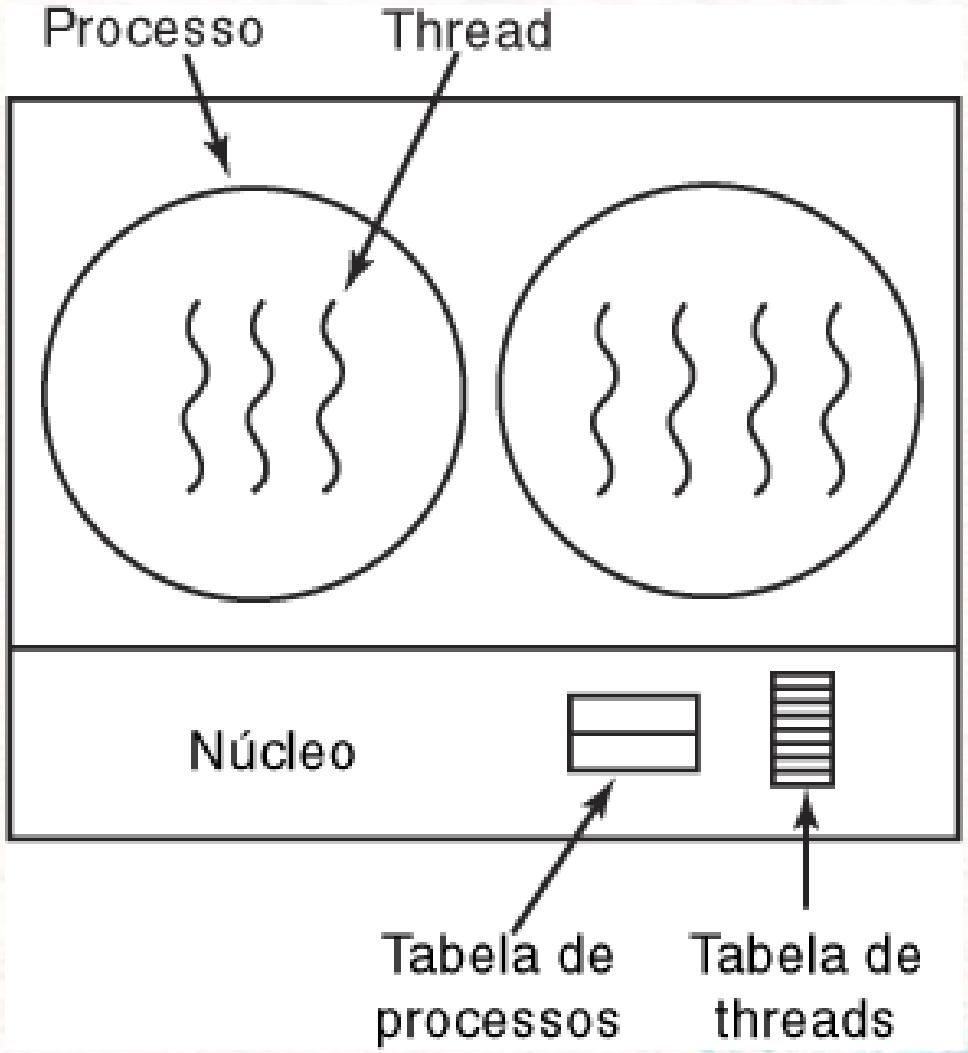
Implementação de Threads

No Kernel

Vantagem

- Não há tabela de threads em cada processo. Em vez disso, o kernel tem uma tabela de threads que controla todos os threads no sistema.
- A informação é a mesma que com threads de nível de usuário, mas agora mantida no kernel em vez de no espaço do usuário (dentro do sistema de tempo de execução).

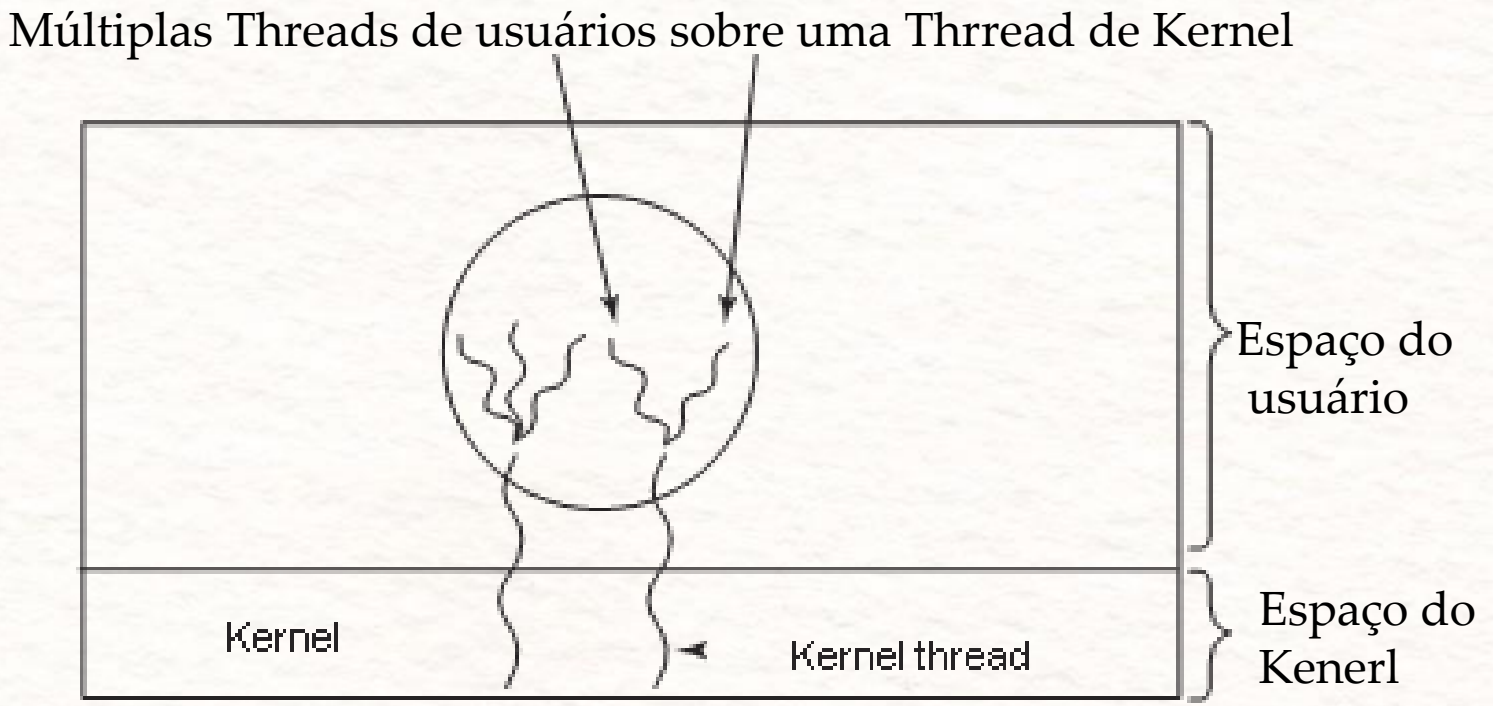
- Desvantagem
  - Custo consideravelmente maior do que uma chamada para um procedimento do sistema em tempo de execução.



Implementação de Threads

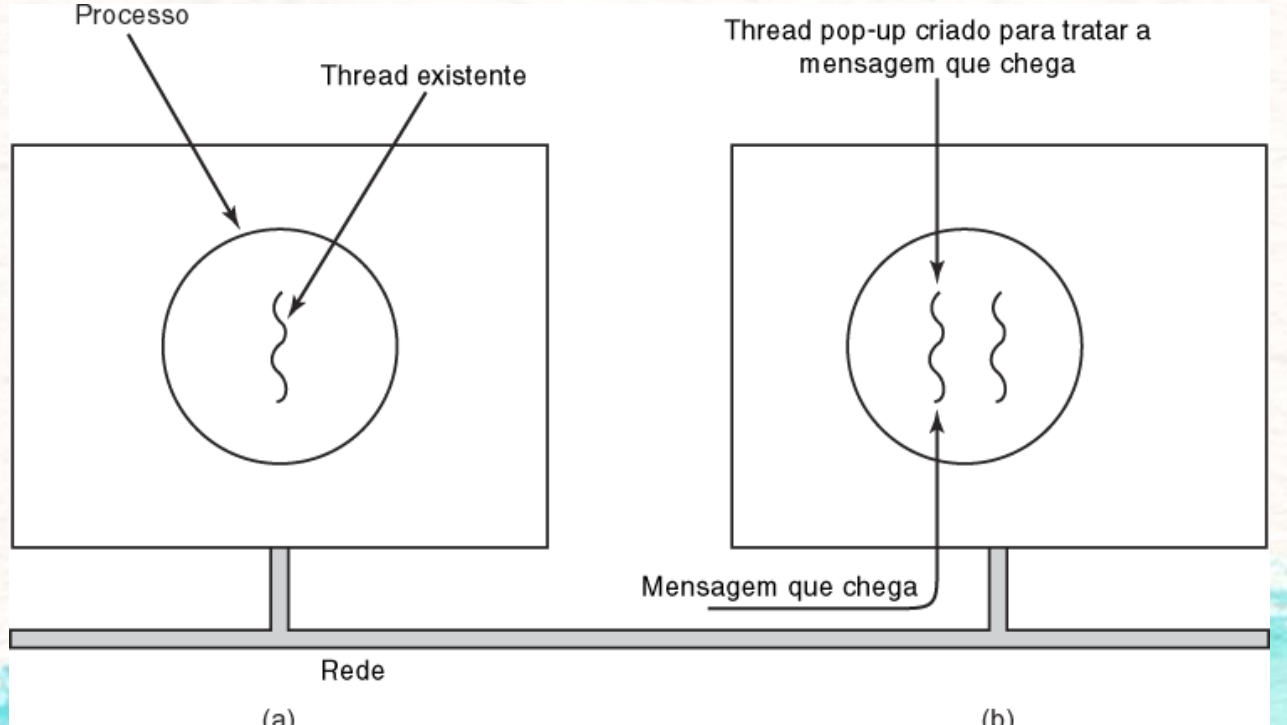
Hybrido

Nesse modelo, cada encadeamento no nível do kernel tem um conjunto de encadeamentos no nível do usuário que se alternam usando-o.



Threads de pop up

Threads são frequentemente úteis em sistemas distribuídos. Um exemplo importante é como as mensagens recebidas, por exemplo, solicitações de serviço, são tratadas.
Ex.: a chegada de uma mensagem faz com que o sistema crie um novo thread para manipular a mensagem. Tal encadeamento é chamado de encadeamento pop-up.
Uma das principais vantagens dos encadeamentos pop-up é que, como eles são novos, eles não têm histórico - registros, pilha, seja o que for - que deve ser restaurado. Cada um começa fresco e cada um é idêntico a todos os outros. Isso possibilita a criação de tal thread rapidamente. O resultado do uso de encadeamentos pop-up é que a latência entre a chegada da mensagem e o início do processamento pode ser muito curta



Comunicação entre Processos

- Processos Independentes.
  - Não afetam nem são afetados por outros processos.

- Processos Cooperados.
  - Compartilham:
    - Memória;
    - Arquivos;
    - Dispositivos de E/S;
    - Etc.

Pipeline

Os processos freqüentemente precisam se comunicar uns com outros processos. Por exemplo, em um pipeline de shell, a saída do primeiro processo deve ser passada para o segundo processo e assim por diante na linha. Assim, há necessidade de comunicação entre processos, preferencialmente de maneira bem estruturada, não utilizando interrupções.

Existem três questões:

1. Como um processo pode passar informações para outro.

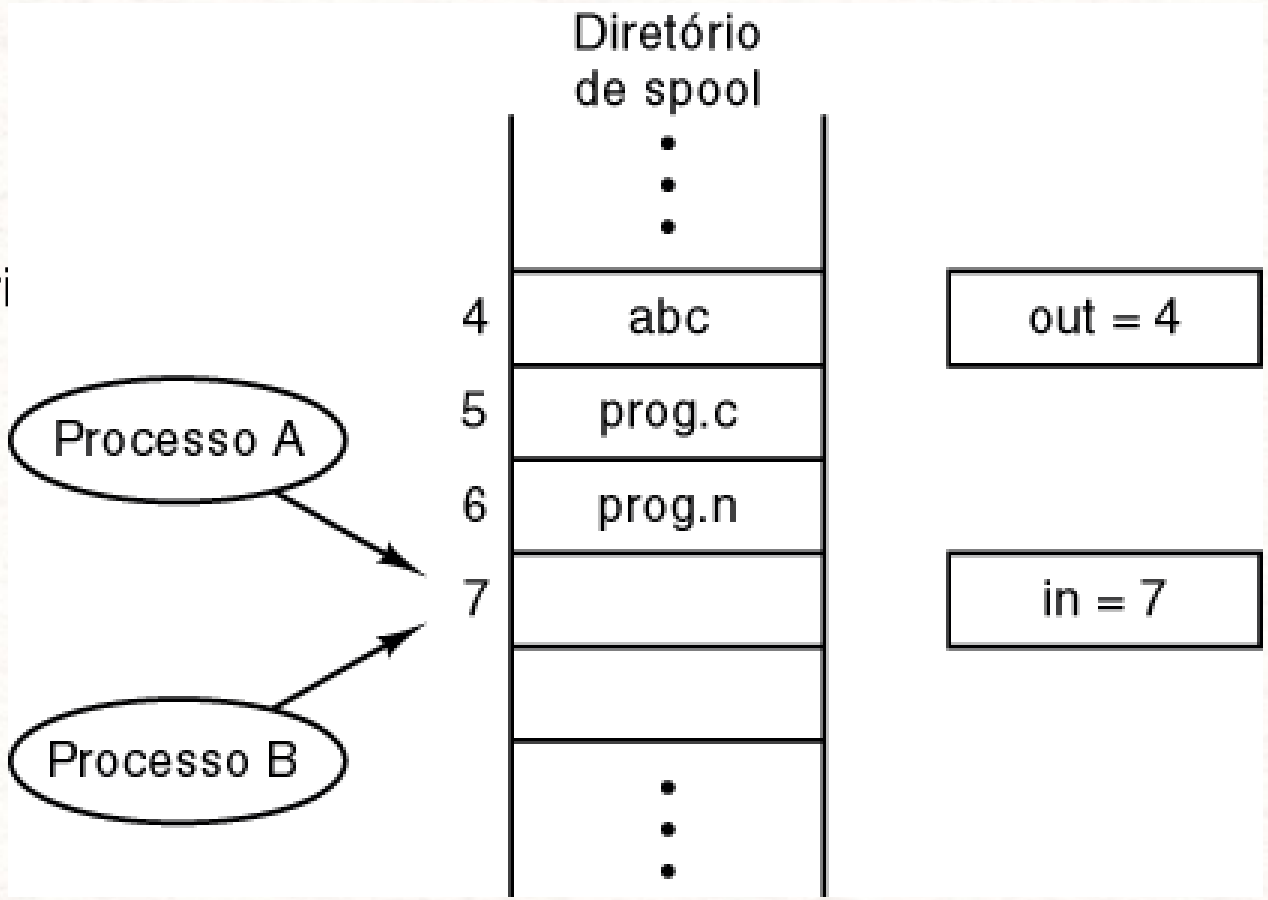
2. Tem a ver com a certeza de que dois ou mais processos não se encaixam uns nos outros, por exemplo, dois processos em um sistema de reservas de companhias aéreas, cada um tentando pegar o último lugar em um avião para um cliente diferente.

3. Diz respeito ao sequenciamento adequado quando as dependências estão presentes: se o processo A produz dados e o processo B os imprime, B tem que esperar até que A tenha produzido alguns dados antes de começar a imprimir.

Também é importante mencionar que dois desses problemas se aplicam igualmente aos threads. A primeira delas - a passagem de informações - é fácil para os threads, pois eles compartilham um espaço de endereço comum (threads em diferentes espaços de endereços que precisam se comunicar estão sob o título de processos de comunicação).

Comunicação entre Processos

- Interprocess Comunication (IPC)
- Dois processos querem escrever simultaneamente em uma memóri compartilhada



Spoll (simultaneous peripheral operations online) - É um espaço livre de memória secundária gerenciado pelo sistema operacional para a comunicação com dispositivos periféricos como impressora. É necessário devido à diferença na velocidade de operação da CPU e dos dispositivos periféricos. Dispositivos periféricos são lentos em sua operação.

Referências Bibliográficas

- TANENBAUM, Andrew S., BOSS, Herbert. **Sistemas Operacionais Modernos,** Pearson - 4ª ed., 2016.
- SILBERSCHATZ, A., GALVIN, P.B., GAGNE, G. **Fundamentos de Sistemas Operacionais**, Ed. LTC, 8ª ed., 2011
- DEITEL, H.M.; DEITEL, P.J.; CHOFFNES, D.R. **– Sistemas Operacionais**. Prentice Hall, Tradução da 3ª ed., 2005
- Link para o livro "Advanced Linux Programming“http://richard.esplins.org/static/downloads/linux_book.pdf