Sistemas Operativos: Threads

Pedro F. Souto (pfs@fe.up.pt)

March 8, 2012

Sumário

Processos em Unix

- Em Unix e SOs dele derivados, como Linux, um processo dispõe essencialmente dum computador virtual:
  - a maioria dos recursos usados por um processo é reservado para seu uso exclusivo;
  - cada processo usa a sua própria memória, a qual, por omissão, não é partilhada com outros processos.
- Um processo pode ser visto como:
  - Uma máquina virtual para execução de programas

  em vez de
  - Um programa em execução

Comunicação entre Processos em Unix

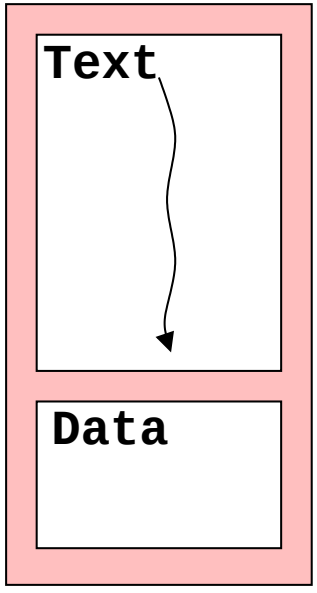
- A comunicação entre processos em Unix não é fácil:
  - o processo pai pode passar alguma informação antes de criar o processo filho, mas depois ...
  - o processo filho só pode retornar informação (e muito limitada) ao processo pai quando termina;
  - sincronização entre processos só entre o pai e os seus filhos.
- Suporte de memória partilhada entre processos pelo SO:
  - \+ facilita a cooperação entre processos;
  - – não é muito conveniente de usar;
  - – é relativamente ineficiente, já que requer a intervenção do SO para sincronização.

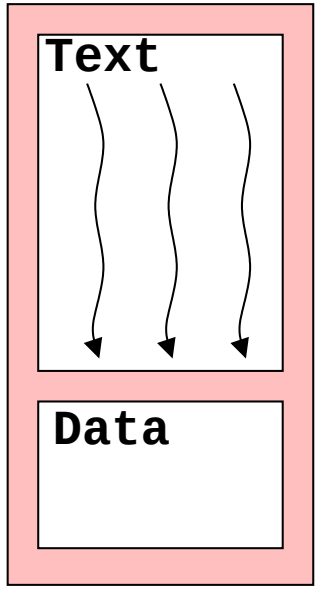
Threads

Threads abstraiem a execução duma sequência de instruções.

Grosso modo, enquanto que um processo corresponde à execução dum programa, um thread corresponde à execução duma função.

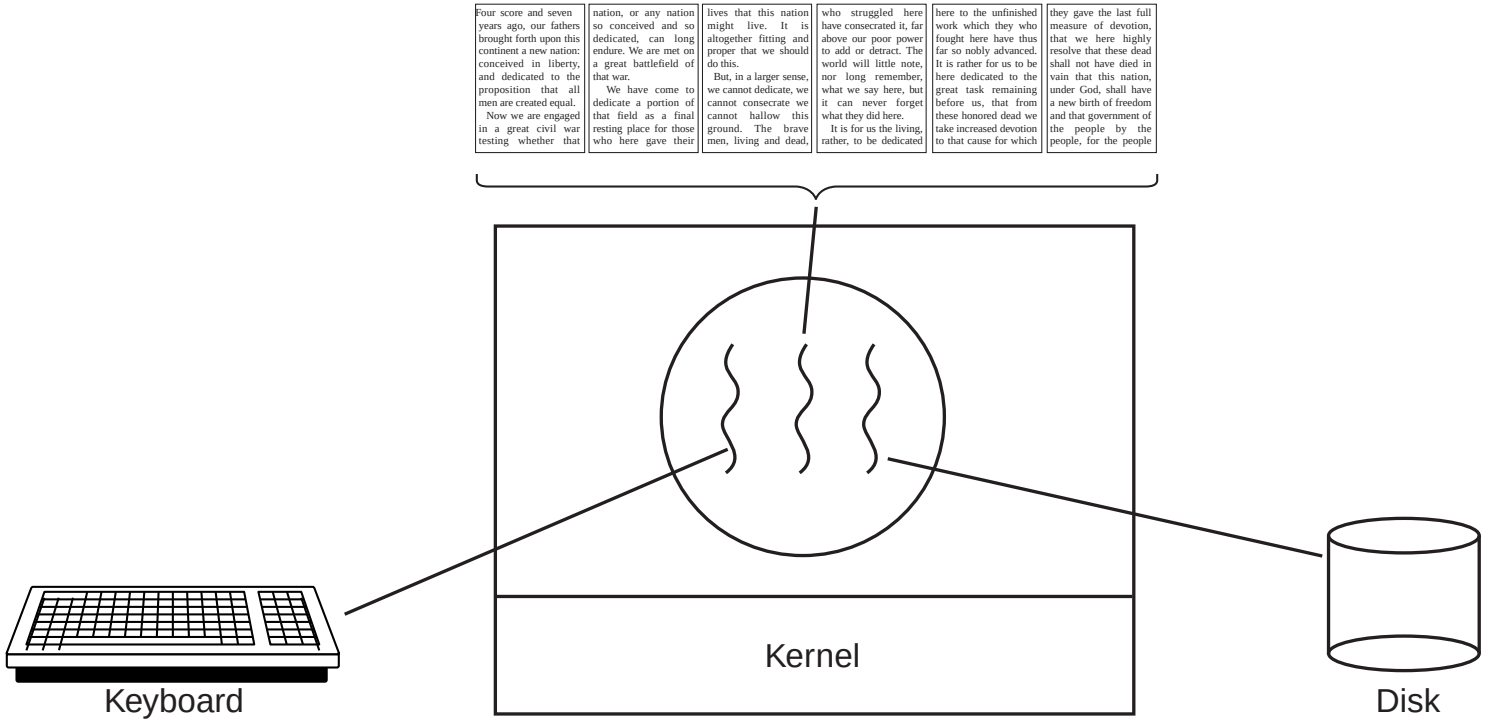
- Em SOs mais recentes, um processo pode incluir mais do que um *thread*.





Processador de Texto *Multithreaded*

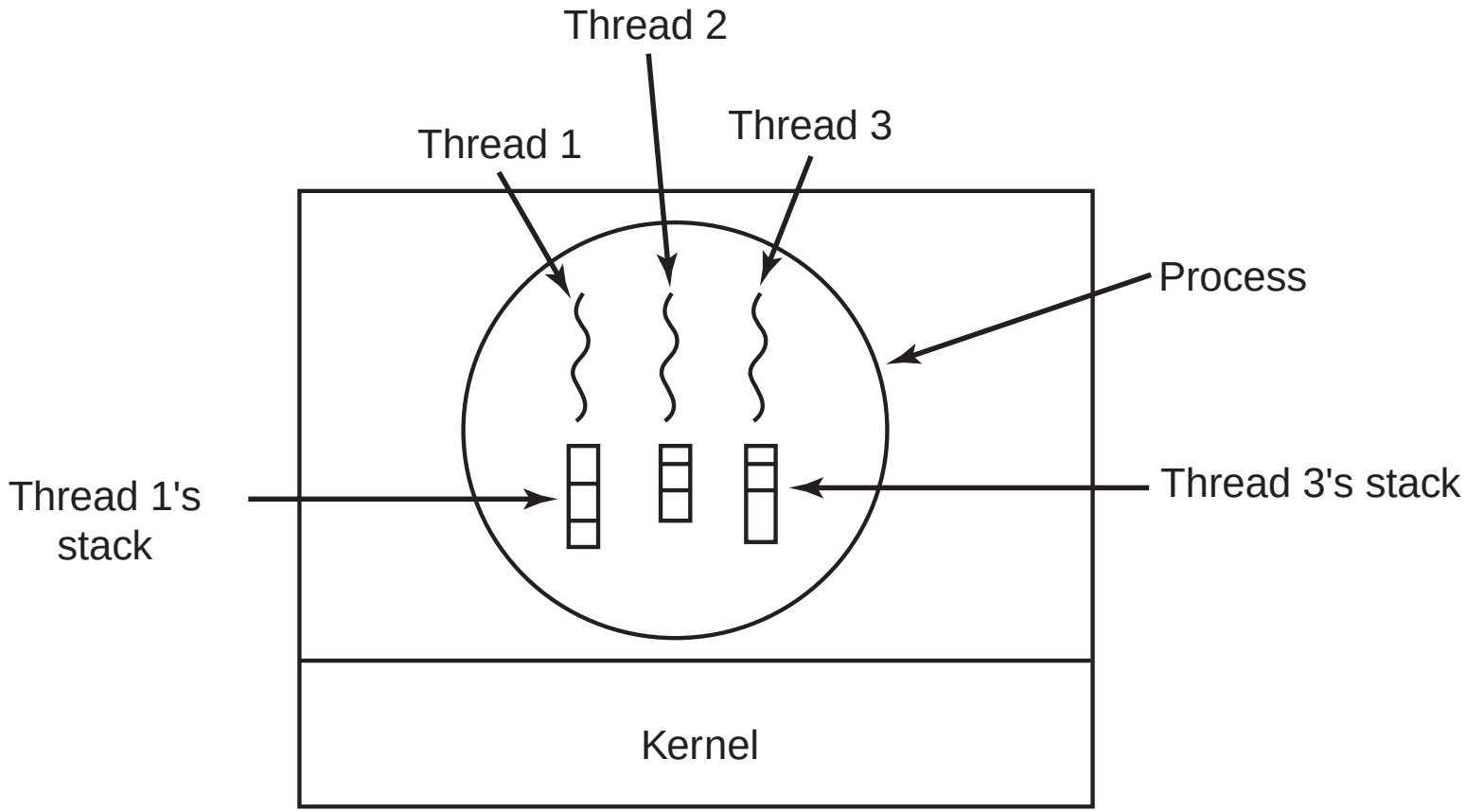
A ideia é usar um *thread* por tarefa.



1. Um *thread* para interagir com o utilizador (teclado e rato);
2. Um *thread* para formatar o texto (em *background*);
3. Um *thread* para guardar o ficheiro periodicamente no disco.

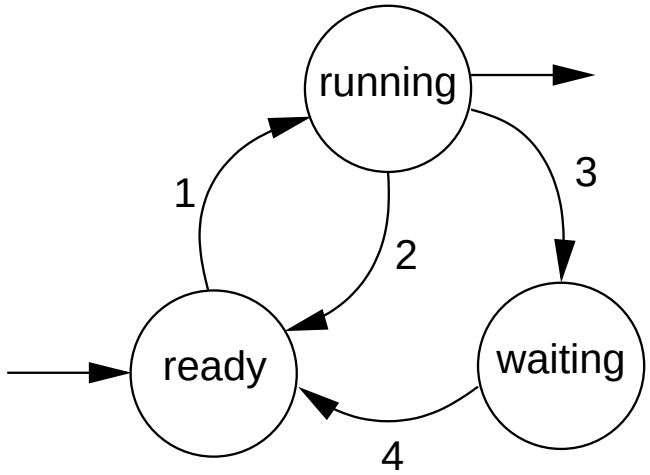
Partilha de recursos com *Threads*

- Threads dum mesmo processo podem partilhar a maior parte dos recursos, com excepção da stack e do estado do processador:



Estado dum *Thread*

- Tal como um processo, um *thread* pode estar num de 3 estados:



- A informação específica a manter por cada *thread* é relativamente reduzida:
  - o seu estado (pode estar bloqueado à espera dum evento);
  - o estado do processador (incluindo o `SP` e `PC`);
  - a stack.
- Operações tais como:
  - criação/terminação
  - comutação

  de *threads* dum mesmo processo são muito mais eficientes do que operações semelhantes sobre processos

Sumário

Uso de *Threads*

- *Threads* dum mesmo processo podem partilhar muitos recursos, incluindo o espaço de endereçamento:

  são particularmente apropriados para aplicações consistindo em actividades *concorrentes*.
- P.ex. servidor da *Web*:
  - Recebe e processa pedidos de páginas da Web.
  - As páginas da Web são ficheiros guardados em disco.
  - Mantém as páginas acedidas mais recentemente em memória, *cache*.
  - Se a página pedida não estiver na *cache*, o servidor tem que ir ao disco.

Servidor da Web com um Único *Thread*

```
while( TRUE ) {
  get_next_request(&buf);
  lookup_page_in_cache(buf, &page);
  if( page == NULL )
     read_page_from_disk(buf, &page);
  send_page(page);
}
```

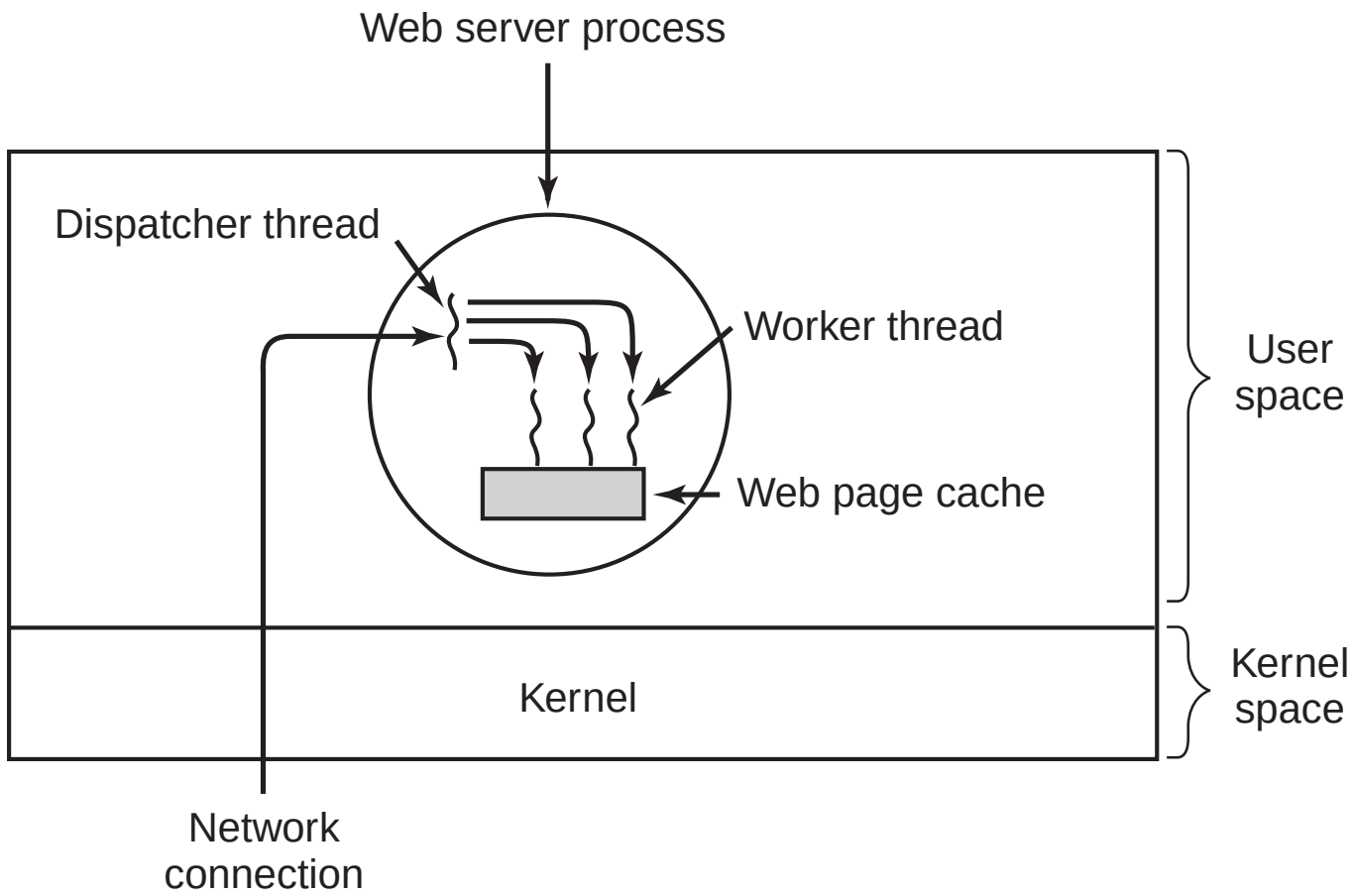
- Se a página pedida não estiver na *cache*, o servidor tem que ir ao disco, bloqueando.
- Enquanto a página não fôr trazida para memória, o servidor não pode processar outros pedidos.
- O número de pedidos que este servidor pode processar por unidade de tempo é muito baixo.

Servidor da Web com E/S Sem Bloqueio

- Alguns SOs suportam chamadas ao sistema de E/S que não bloqueiam o processo que as invoca. Posteriormente:
  - o processo pode interrogar o *kernel* sobre a conclusão da operação (*non-blocking I/O*);
  - alternativamente, o *kernel* pode notificar o processo da conclusão da operação (*asynchronous I/O*).
- Se a página pedida não estiver na *cache*, o servidor pode executar uma operação de E/S sem-bloqueio/assíncrona.
- Depois, pode receber e processar a mensagem seguinte.
- O servidor tem que manter informação sobre o estado de processamento de cada pedido pendente.
- Este tipo de solução diz-se *event driven*, ou baseada numa *finite-state machine*.

Servidor da Web com Múltiplos *Threads*

- Um *thread*, o *dispatcher*, recebe os pedidos e passa-os a outros *threads*, os *worker*.
- Cada *worker thread* processa um pedido de cada vez: pode usar operações de E/S que bloqueiem.



Servidor da Web com Múltiplos Threads

- Código do *dispatcher thread*:

```
while( TRUE ) {
    get_next_request(&buf);
    handoff_work(buf);
}
```

- Código dos *worker threads*:

```
while( TRUE ) {
    wait_for_work(&buf);
    lookup_page_in_cache(buf, &page);
    if( page == NULL )
        read_page_from_disk(buf, &page);
    send_page(page);
}
```

Servidor da Web: Comparação

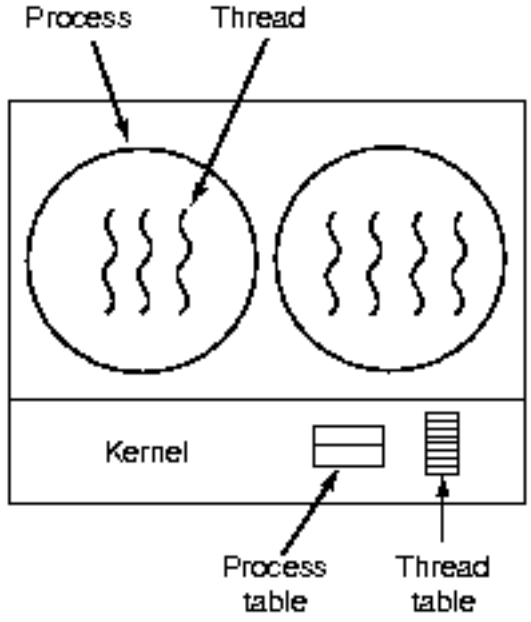
Modêlo	**Paralelismo**	**Programação**
Thread único	Não	Fácil.
Event-driven	Sim	Trabalhosa.
Multi-threaded	Sim	Nem sempre fácil.

Sumário

Implementação de *threads*

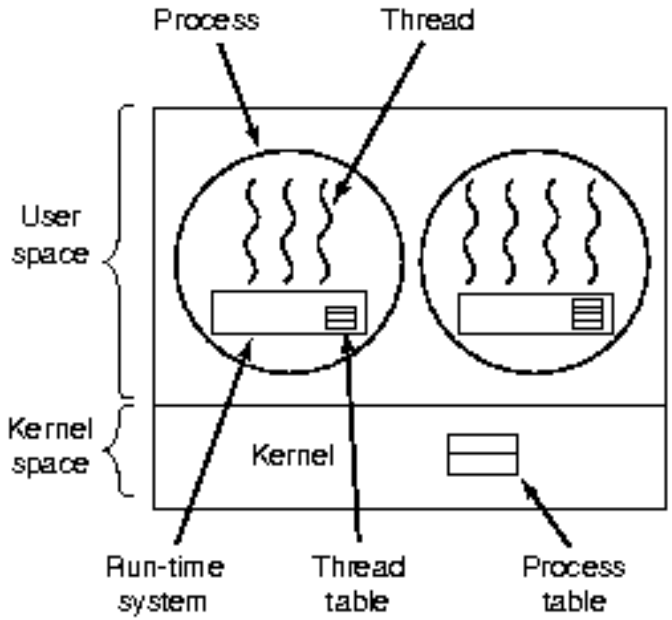
- *Threads* podem ser implementados:
  1. directamente pelo SO (*kernel-level threads*);
  2. por código que executa em *user-level*, i.e. acima do SO, (*user-level threads*).

Kernel-level Threads



- O *kernel* suporta processos com múltiplos *threads*:
  os *threads* são as *"entidades"* que disputam o CPU.
- O SO mantém uma tabela de *threads* com a informação específica a cada *thread*.
  - O PCB dum processo aponta para a sua tabela de *threads*.
- Todas as operações de gestão de *threads*, p.ex. criar um *thread*, requerem a execução de chamadas ao sistema.

User-level Threads



- O *kernel* não sabe da existência dos *threads*:
  - são implementados inteiramente por uma biblioteca em *user-space*;
  - podem ser implementados num SO que não suporta *threads*.

Implementação de *User-level Threads*

- A biblioteca de *threads* oferece funções que permitem:
  - criar/terminar *threads*;
  - sincronizar entre *threads*;
  - ceder o CPU a outros *threads* (`yield`);
- A biblioteca executa a comutação entre *threads* e mantém uma tabela de *threads*.
- Funções que encapsulam chamadas ao sistema que podem bloquear têm que ser alteradas:

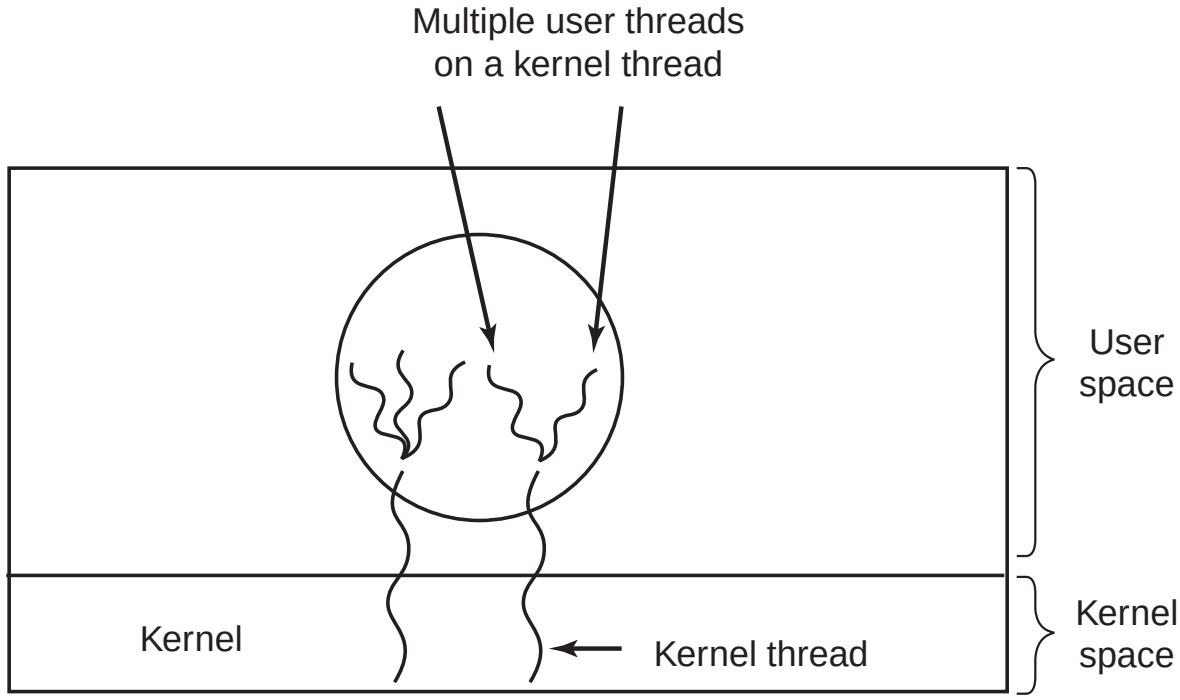
  para evitar que todos os *threads* bloqueiem.
- Algumas dificuldades:
  - como executar chamadas ao sistema sem bloquear?
  - e *page-faults*?
  - como evitar que um *thread* monopolize o CPU?

User-level vs. Kernel-level Threads

+ O SO não precisa suportar *threads*.
+ Evita a intervenção do *kernel* em muitas operações, p.ex. criação/terminação de *threads* e comutação de *threads*.
– *Page-fault* por um *thread* bloqueia os restantes *threads* do processo.
– Incapazes de explorar paralelismo em arquitecturas multiprocessador.

Implementação Híbrida

A ideia é multiplexar *user-level threads* sobre *kernel-level threads*

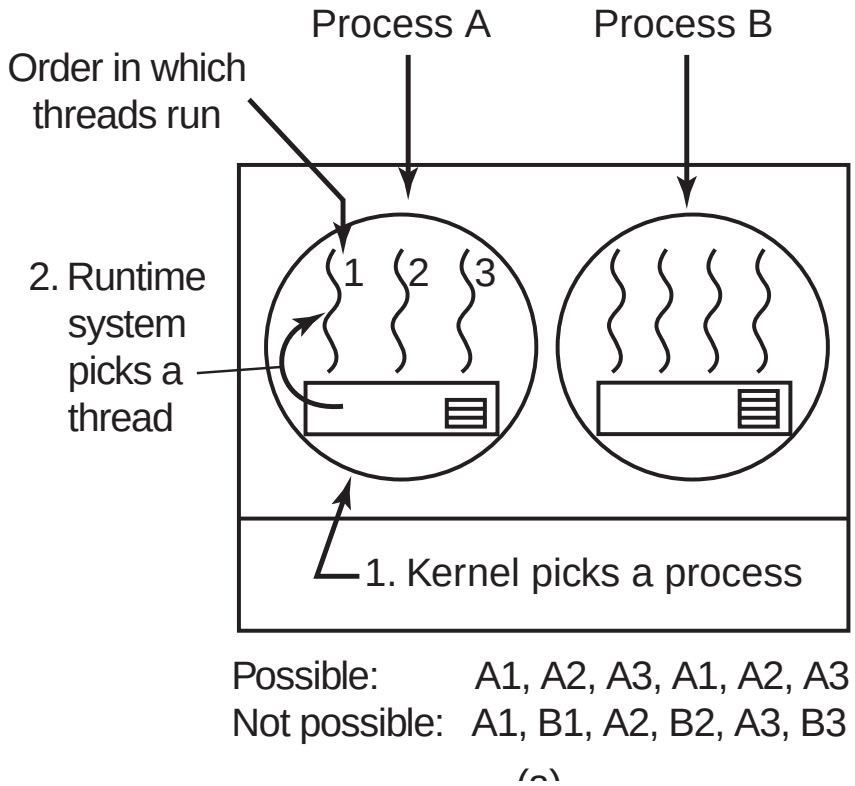


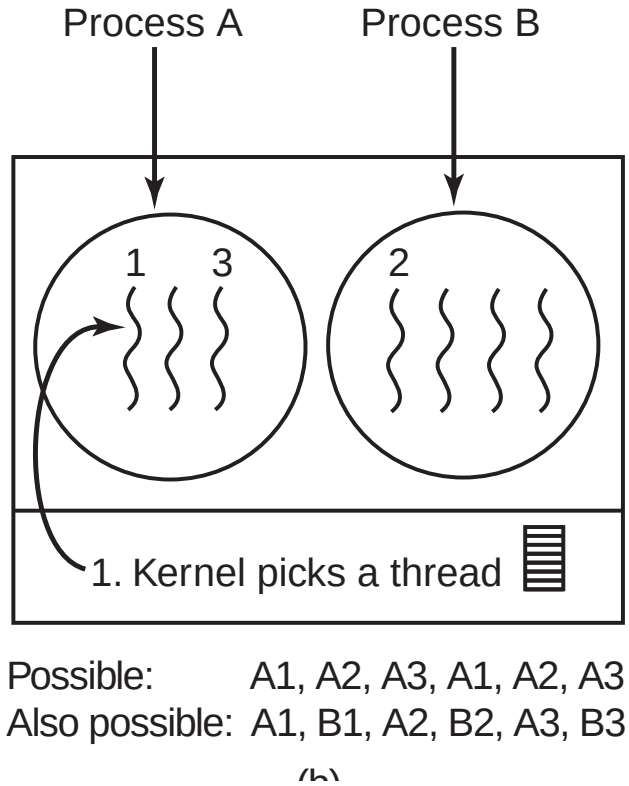
- O *kernel* não está a par dos *user-level threads*.
- A biblioteca de *user-level threads* atribui estes aos *kernel-level threads*.
- O número de *user-level threads* pode ser muito maior do que o de *kernel-level threads*

Sumário

Escalonamento de *Threads*

User-level *vs.* *kernel-level*





- *Quantum* de cada processo de 50 ms.
- Cada *thread* executa em *bursts* de 5 ms.

Sumário

Leitura Adicional

Sistemas Operativos

- Secção 3.4: *Modelo Multitarefa*

Modern Operating Systems, 2nd. Ed.

- Secção 2.2: *Threads*

Operating Systems Concepts

- Secção 4.1: *Overview* (of threads)
- Secção 4.2: *Multithreading Models*