ACH 2147 — Desenvolvimento de Sistemas de Informação Distribuídos

Aula 06: Processos (parte 1)

Prof. Renan Alves

Escola de Artes, Ciências e Humanidades — EACH — USP

15/03/2024

Introdução às threads

Ideia básica

Construção de processadores virtuais em software, para representar processadores físicos.

Processador: Fornece um conjunto de instruções juntamente com a capacidade de executar automaticamente uma série dessas instruções.

Thread: Uma unidade de software mínima no qual um conjunto de instruções pode ser executado em um determinado contexto. Salvar o contexto de uma thread implica em parar a execução atual e salvar todos os dados necessários para continuar a execução em uma etapa posterior.

Processo: Uma unidade de software no contexto da qual um ou mais threads podem ser executados. Executar uma thread significa executar uma série de instruções no contexto dessa thread.

Troca de contexto

Contextos

- Contexto do processador: O conjunto mínimo de valores armazenados nos registradores de um processador usados para a execução de uma série de instruções (por exemplo, ponteiro de pilha, registradores de endereçamento, contador de programa).

Troca de contexto

Contextos

- Contexto do processador: O conjunto mínimo de valores armazenados nos registradores de um processador usados para a execução de uma série de instruções (por exemplo, ponteiro de pilha, registradores de endereçamento, contador de programa).
- Contexto da thread: O conjunto mínimo de valores armazenados em registradores e memória, usados para a execução de uma série de instruções (ou seja, contexto do processador, estado).

Troca de contexto

Contextos

- Contexto do processador: O conjunto mínimo de valores armazenados nos registradores de um processador usados para a execução de uma série de instruções (por exemplo, ponteiro de pilha, registradores de endereçamento, contador de programa).
- Contexto da thread: O conjunto mínimo de valores armazenados em registradores e memória, usados para a execução de uma série de instruções (ou seja, contexto do processador, estado).
- Contexto do processo: O conjunto mínimo de valores armazenados em registradores e memória, usados para a execução de uma thread (ou seja, contexto da thread, mas também, pelo menos, os valores de registradores MMU).

Troca de contexto

Observações

1. As threads compartilham o mesmo espaço de endereçamento. É possível até mesmo realizar a troca de contexto de thread de forma independente do sistema operacional.
2. A troca de processo geralmente é mais cara, pois precisa envolver o SO, ou seja, gerar uma interrupção de software para o kernel e atualização de mapas de memória (invalidação de cache).
3. As ações de criar e destruir são muito mais baratas para threads do que para processos.

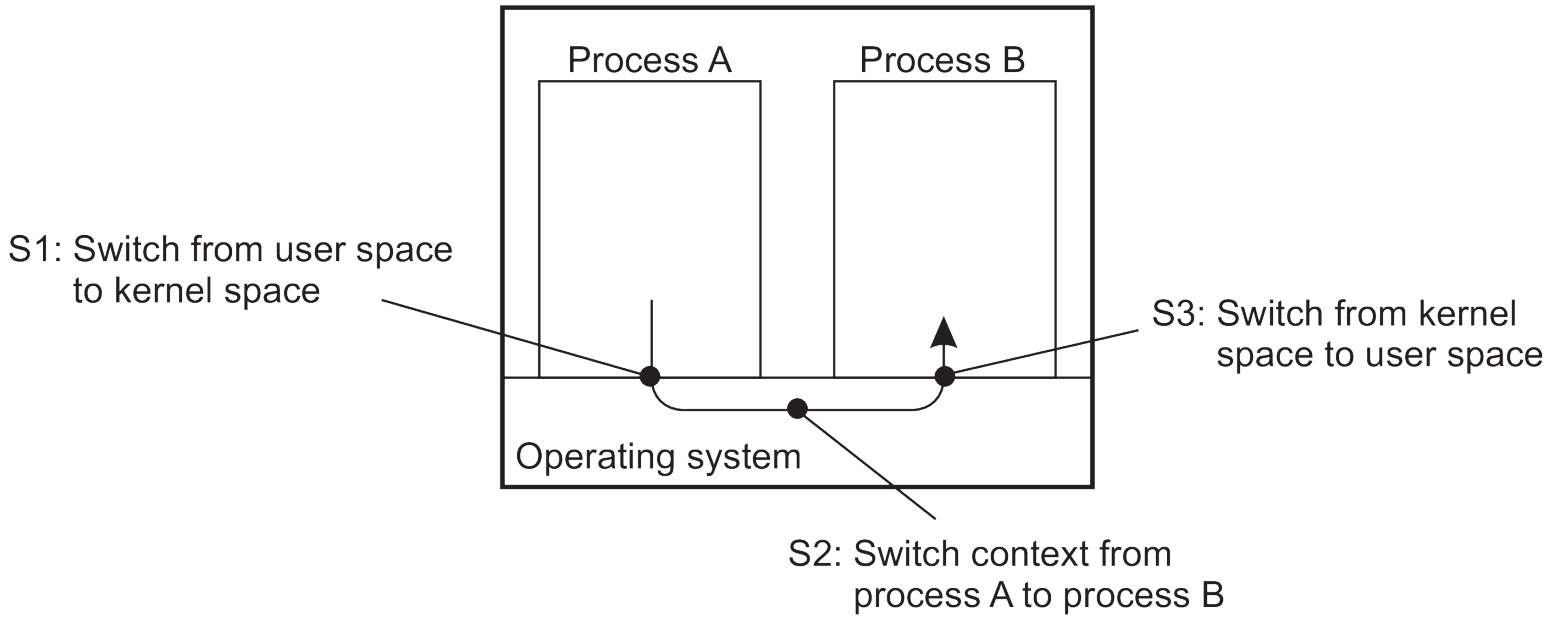
Motivos para usar threads

Algumas razões simples

- Evitar bloqueios desnecessários: um processo de thread única irá bloquear ao fazer E/S; em um processo multithread, o sistema operacional pode alternar a CPU para outra thread nesse processo.
- Explorar o paralelismo: as threads em um processo multithread podem ser agendadas para serem executadas em paralelo em um processador multinúcleos.
- Evitar trocas de processos: estruturar grandes aplicações não como uma coleção de processos, mas sim por meio de várias threads.
- Organização de software: mais fácil de organizar (alguns tipos de) sofware.

Evitar trocas de processos

Evitar trocas de contexto custosas



Trade-offs

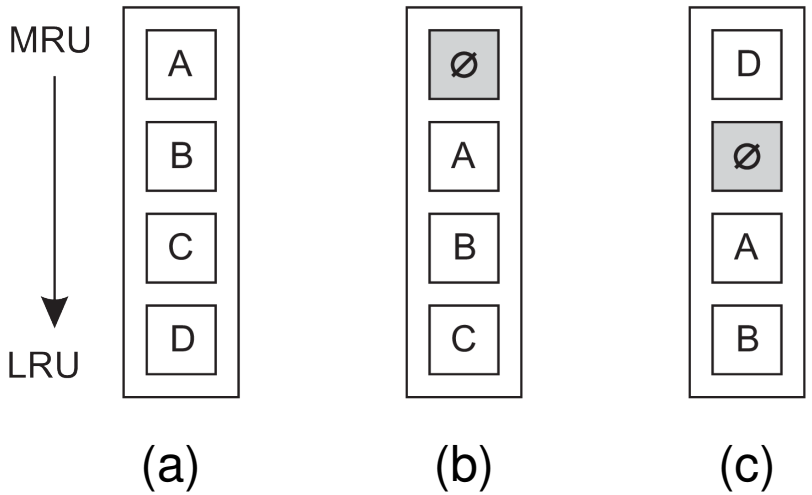
- As threads usam o mesmo espaço de endereço: mais propensas a erros
- Sem suporte do SO/HW para proteger as threads usando a memória umas das outras
- A troca de contexto de thread pode ser mais rápida do que a troca de contexto de processo

O custo de uma troca de contexto

Considere o tratamento de interrupção de clock simples

- custos diretos: a troca de fato e execução do código de tratamento
- custos indiretos: outros custos, principalmente causados por bagunçar o cache

O que uma troca de contexto pode causar: custos indiretos



(a) antes da troca de contexto

(b) após a troca de contexto

(c) após acessar o bloco *D*.

Um exemplo simples em Python

```
1  from multiprocessing import Process
2  from time import *
3  from random import *
4
5  def sleeper(name):
6      t = gmtime()
7      s = randint(1,20)
8      txt = str(t.tm_min)+':'+str(t.tm_sec)+' '+name+' is going to sleep for '+str(s)+' seconds'
9      print(txt)
10     sleep(s)
11     t = gmtime()
12     txt = str(t.tm_min)+':'+str(t.tm_sec)+' '+name+' has woken up'
13     print(txt)
14
15 if __name__ == '__main__':
16     p = Process(target=sleeper, args=('eve',))
17     q = Process(target=sleeper, args=('bob',))
18     p.start(); q.start()
19     p.join(); q.join()
```

```
40:23 eve is going to sleep for 14 seconds
40:23 bob is going to sleep for 4 seconds
40:27 bob has woken up
40:37 eve has woken up
```

Um (outro) exemplo simples em Python

```
1  from multiprocessing import Process
2  from threading import Thread
3  from time import *
4  from random import *
5
6  shared_x = randint(10,99)
7
8  def sleeping(name):
9      global shared_x
10     t = gmtime(); s = randint(1,20)
11     txt = str(t.tm_min)+':'+str(t.tm_sec)+' '+name+' is going to sleep for '+str(s)+' seconds'
12     print(txt)
13     sleep(s)
14     t = gmtime(); shared_x = shared_x + 1
15     txt = str(t.tm_min)+':'+str(t.tm_sec)+' '+name+' has woken up, seeing shared x being '
16     print(txt+str(shared_x) )
17
18 def sleeper(name):
19     sleeplist = list()
20     print(name, 'sees shared x being', shared_x)
21     for i in range(3):
22         subsleeper = Thread(target=sleeping, args=(name+' '+str(i),))
23         sleeplist.append(subsleeper)
24     for s in sleeplist: s.start()
25     for s in sleeplist: s.join()
26     print(name, 'sees shared x being', shared_x)
27
28 if __name__ == '__main__':
29     p = Process(target=sleeper, args=('eve',))
30     q = Process(target=sleeper, args=('bob',))
31     p.start(); q.start()
32     p.join(); q.join()
```

Um (outro) exemplo simples em Python

```
eve sees shared x being 47
bob sees shared x being 47
31:8 eve 0 is going to sleep for 16 seconds
31:8 eve 1 is going to sleep for 2 seconds
31:8 eve 2 is going to sleep for 18 seconds
31:8 bob 0 is going to sleep for 14 seconds
31:8 bob 1 is going to sleep for 20 seconds
31:8 bob 2 is going to sleep for 2 seconds
31:10 eve 1 has woken up, seeing shared x being 48
31:10 bob 2 has woken up, seeing shared x being 48
31:22 bob 0 has woken up, seeing shared x being 49
31:24 eve 0 has woken up, seeing shared x being 49
31:26 eve 2 has woken up, seeing shared x being 50
eve sees shared x being 50
31:28 bob 1 has woken up, seeing shared x being 50
bob sees shared x being 50
```

Threads e sistemas operacionais

Problema principal

Um kernel de SO deveria fornecer threads, ou elas deveriam ser implementadas a nível de usuário?

Solução de espaço do usuário

- Todas as operações podem ser completamente tratadas dentro de um único processo $\Rightarrow$ as implementações podem ser extremamente eficientes.
- Todos os serviços fornecidos pelo kernel são feitos em nome do processo no qual uma thread reside $\Rightarrow$ se o kernel decidir bloquear uma thread, o processo inteiro será bloqueado.
- Threads são usadas quando há muitos eventos externos: as threads bloqueiam por evento $\Rightarrow$ se o kernel não puder distinguir threads, como ele pode dar suporte a sinalização de eventos para elas?

Threads e sistemas operacionais

Solução de kernel

A ideia principal é que o kernel contém a implementação do funcionamento das threads. Isso significa que todas as operações de thread (criar, apagar, sincronismo, etc) se tornam chamadas de sistema:

- Operações que bloqueiam uma thread já não são um problema: o kernel agenda outra thread disponível dentro do mesmo processo.
- lidar com eventos externos é simples: o kernel (que captura todos os eventos) agenda a thread associada ao evento.
- O problema é (ou costumava ser) a perda de eficiência, pois cada operação de thread requer uma interrupção de software para o kernel.

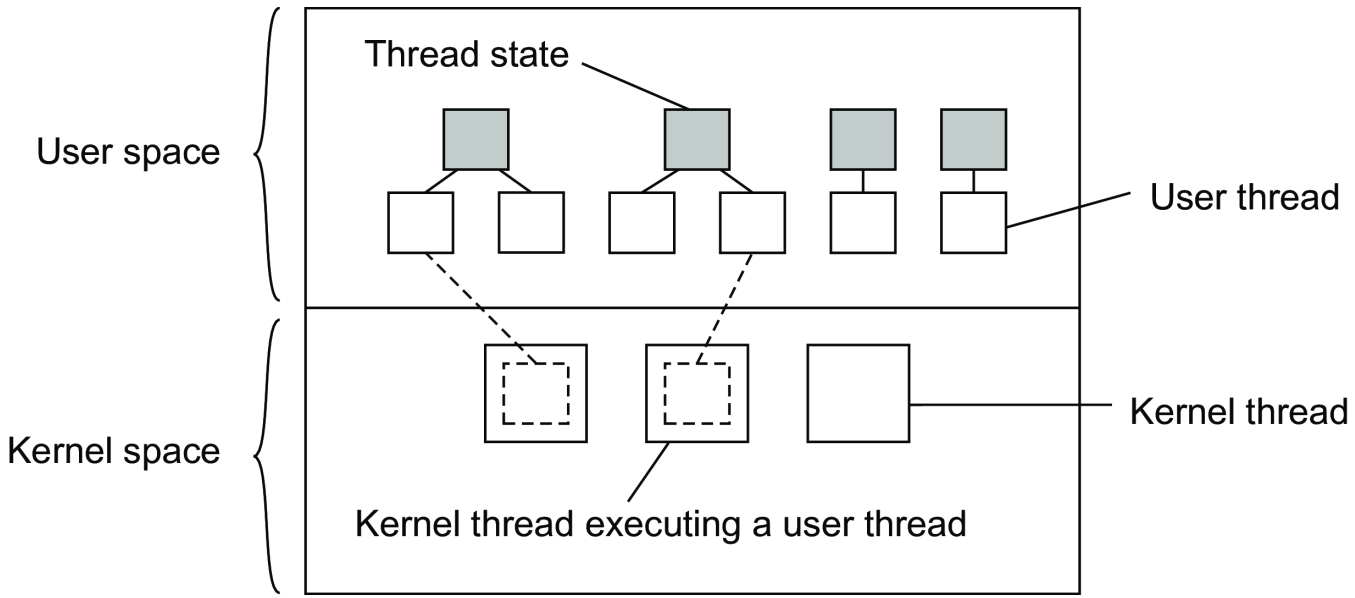
Conclusão – mas

Tentar aliar as threads de nível de usuário às threads de nível de kernel. No entanto, o ganho de desempenho geralmente não compensa a complexidade aumentada.

Combinação de threads de usuário e de kernel

Ideia básica

Introduzir uma abordagem de encadeamento de dois níveis: threads de kernel que podem executar threads de usuário.



Threads de usuário e de kernel combinadas

Princípio operacional

Threads de usuário e de kernel combinadas

Princípio operacional

- Thread de usuário faz uma chamada de sistema $\Rightarrow$ a thread de kernel (que está executando essa thread de usuário) bloqueia. A thread de usuário permanece vinculada à thread de kernel.

Threads de usuário e de kernel combinadas

Princípio operacional

- Thread de usuário faz uma chamada de sistema $\Rightarrow$ a thread de kernel (que está executando essa thread de usuário) bloqueia. A thread de usuário permanece vinculada à thread de kernel.
- O kernel pode agendar outra thread de kernel com uma thread de usuário executável vinculada a ela. Nota: esta thread de usuário pode alternar para qualquer outra thread de usuário executável atualmente no espaço do usuário.

Threads de usuário e de kernel combinadas

Princípio operacional

- Thread de usuário faz uma chamada de sistema ⇒ a thread de kernel (que está executando essa thread de usuário) bloqueia. A thread de usuário permanece vinculada à thread de kernel.
- O kernel pode agendar outra thread de kernel com uma thread de usuário executável vinculada a ela. Nota: esta thread de usuário pode alternar para qualquer outra thread de usuário executável atualmente no espaço do usuário.
- Uma thread de usuário chama uma operação de nível de usuário bloqueante ⇒ faz a troca de contexto para uma thread de usuário executável (então vinculada à mesma thread de kernel).

Threads de usuário e de kernel combinadas

Princípio operacional

- Thread de usuário faz uma chamada de sistema $\Rightarrow$ a thread de kernel (que está executando essa thread de usuário) bloqueia. A thread de usuário permanece vinculada à thread de kernel.
- O kernel pode agendar outra thread de kernel com uma thread de usuário executável vinculada a ela. Nota: esta thread de usuário pode alternar para qualquer outra thread de usuário executável atualmente no espaço do usuário.
- Uma thread de usuário chama uma operação de nível de usuário bloqueante $\Rightarrow$ faz a troca de contexto para uma thread de usuário executável (então vinculada à mesma thread de kernel).
- Quando não há threads de usuário para agendar, uma thread de kernel pode permanecer ociosa e até mesmo ser removida (destruída) pelo kernel.

Usando threads no lado do cliente

Cliente Web multithread

Ocultando latências de rede:

- O navegador da Web examina uma página HTML recebida e descobre que mais arquivos precisam ser buscados.
- Cada arquivo é buscado por uma thread separada, cada uma fazendo uma solicitação HTTP (bloqueadora).
- Conforme os arquivos chegam, o navegador os exibe.

Múltiplas chamadas de solicitação-resposta para outras máquinas (RPC)

- Um cliente faz várias chamadas ao mesmo tempo, cada uma por uma thread diferente.
- Ele então espera até que todos os resultados tenham sido retornados.
- Observação: se as chamadas forem para servidores diferentes, podemos ter um aumento de velocidade linear.

Clientes multithread: ajuda mesmo?

Paralelismo de nível de thread: TLP (Thread-Level Parallelism)

Seja c_i a fração de tempo em que exatamente i threads estão sendo executadas simultaneamente

$$TLP = \frac{\sum_{i=1}^{N} i \cdot c_i}{1 - c_0}$$

com N o número máximo de threads (que podem ser) executadas ao mesmo tempo.

Clientes multithread: ajuda mesmo?

Paralelismo de nível de thread: TLP (Thread-Level Parallelism)

Seja c_i a fração de tempo em que exatamente i threads estão sendo executadas simultaneamente

$$TLP = \frac{\sum_{i=1}^{N} i \cdot c_i}{1 - c_0}$$

com N o número máximo de threads (que podem ser) executadas ao mesmo tempo.

Medições práticas

Um navegador da Web típico tem um valor TLP entre 1,5 e 2,5 $\Rightarrow$ as threads são usadas principalmente para organizar logicamente os navegadores.

Usando threads no lado do servidor

Melhorar o desempenho

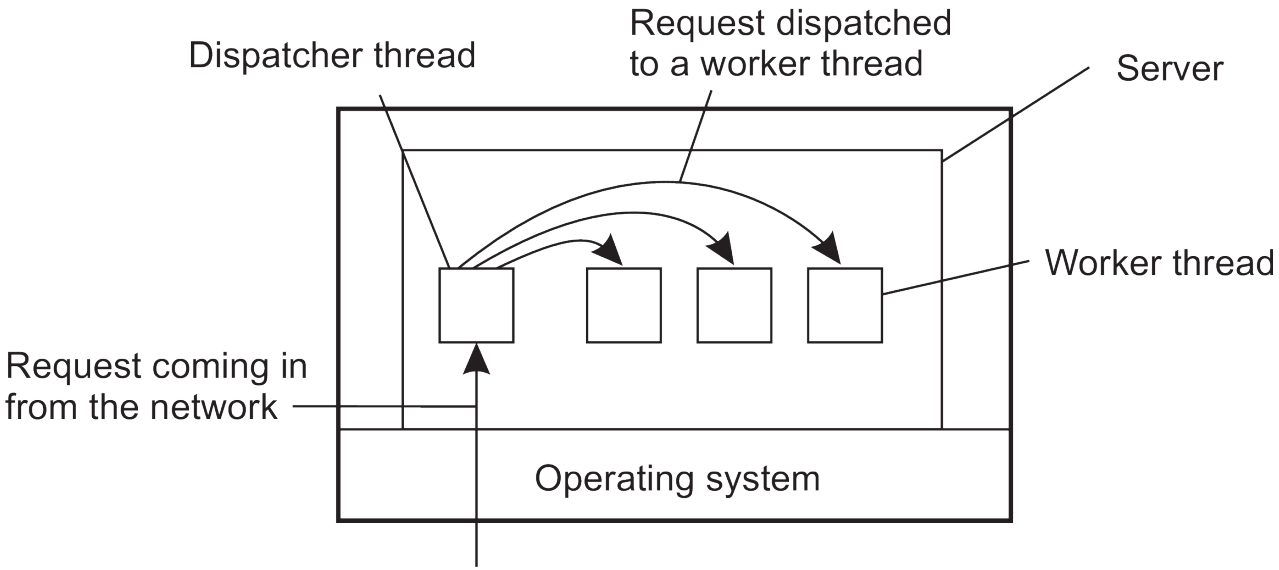
- Iniciar uma thread é mais barato do que iniciar um novo processo.
- Ter um servidor de thread única reduz a escalabilidade de um sistema multiprocessador.
- Como com os clientes: ocultar a latência da rede reagindo à próxima solicitação enquanto a anterior está sendo respondida.

Estrutura melhor

- A maioria dos servidores têm alta demanda de E/S. Usar chamadas de bloqueantes simples, bem conhecidas simplifica a estrutura.
- Programas multithreaded tendem a ser menores e mais fáceis de entender devido ao fluxo de controle simplificado.

Porque multithreading é popular: organização

Modelo dispatcher/worker (despachante/operário)



Visão geral

Modelo	Características
Multithreading	Paralelismo, chamadas de sistema bloqueantes
Processo de única thread	Sem paralelismo, chamadas de sistema bloqueantes
Máquina de estado finito	Paralelismo, chamadas de sistema não bloqueantes