Aula 06

Concorrência, Threads e Locks

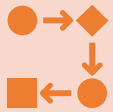
O Ponto de Partida: O Processo de Thread Única

Até agora, vimos um processo com um único ponto de execução (um Program Counter - PC).

O SO cria a ilusão de uma CPU e memória privadas para cada processo.

A Nova Abstração: A Thread

Uma thread é um novo ponto de execução dentro de um processo existente.

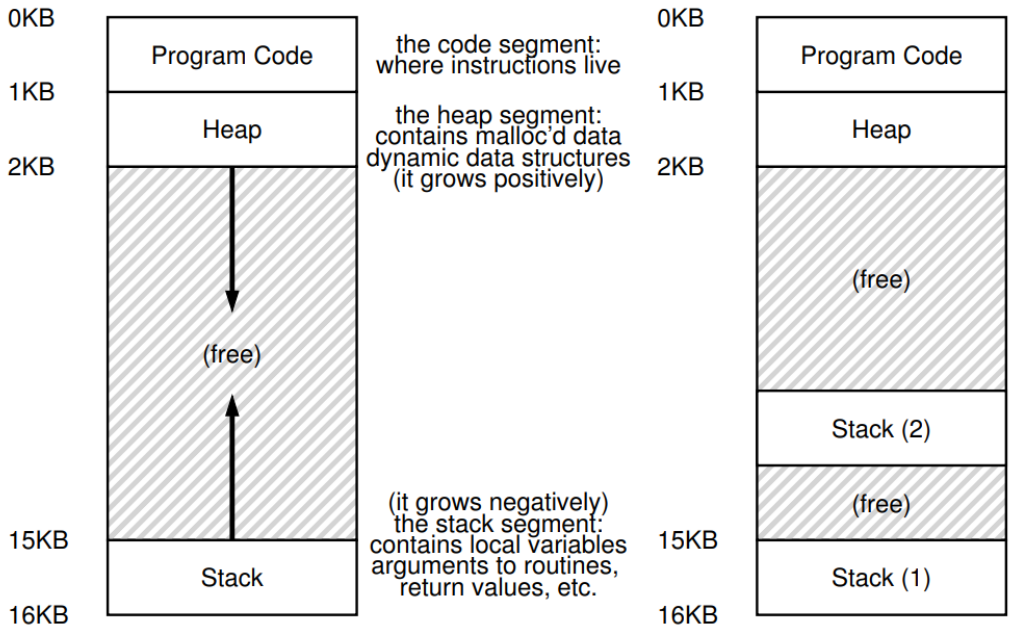


A Grande Diferença: Threads no mesmo processo compartilham o mesmo espaço de endereçamento (código e heap).

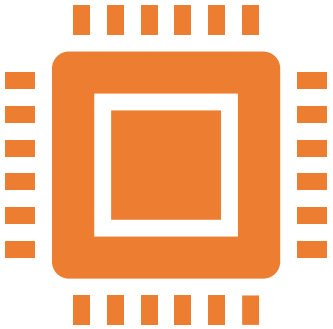
Isso torna a comunicação entre threads rápida, mas perigosa.

Anatomia de uma Thread

- Compartilhado: Código, Dados Globais, Heap.
- Privado por Thread: Program Counter (PC), Conjunto de Registradores, Pilha.



Por Que Usar Threads?



Paralelismo: Dividir tarefas para executá-las mais rápido em sistemas com múltiplos processadores.

Evitar Bloqueio (I/O): Manter o programa responsivo enquanto uma thread espera por I/O.

O Problema: Imprevisibilidade e Dados Compartilhados

O escalonador do SO decide qual thread executa e quando.	A ordem é não-determinística.
Exemplo:	Duas threads incrementam um contador 10 milhões de vezes.
Resultado Esperado:	20.000.000
Resultado Real:	19.345.221 (ou outro valor incorreto). Por quê?

O Coração do Problema: Operações Não-Atômicas

- A instrução `contador++` não é uma única operação para a CPU.
- Ela se decompõe em três passos: carregar, incrementar e armazenar.
  1. Carregar o valor de contador da memória para um registrador.
  2. Incrementar o valor no registrador.
  3. Armazenar o novo valor de volta na memória.

A Condição de Corrida (Race Condition) em Detalhe

- Thread 1 carrega contador (valor 50) e incrementa para 51.
- INTERRUPÇÃO! O SO troca para a Thread 2.
- Thread 2 carrega contador (ainda 50), incrementa para 51 e armazena.
- Thread 1 volta e armazena 51, perdendo uma atualização.
- Resultado: Duas operações resultaram em 51, não 52.

Terminologia Essencial da Concorrência

Seção Crítica:	Condição de Corrida:	Exclusão Mútua:	Atomicidade:
• Código que acessa um recurso compartilhado e precisa de proteção.	• Resultado depende da ordem de execução das threads.	• Garantir que apenas uma thread entre na seção crítica por vez.	• Sequência de operações indivisível.

A Solução: Locks (ou Mutexes)

- Locks são primitivas de sincronização que fornecem exclusão mútua.
- Uso padrão:

```
lock(&meu_lock);
// Seção Crítica
unlock(&meu_lock);
```

- Se uma thread tenta adquirir um lock ocupado, ela bloqueia (espera).

Como Avaliamos um Bom Lock?

Correção: Garante exclusão mútua?

Justiça (Fairness): Evita starvation?

Performance: Qual o custo (overhead) de usá-lo?

Tentativa #1: Desabilitar Interrupções

Ideia: Desligar interrupções antes da seção crítica.

Falhas: Requer privilégios de administrador e não funciona em sistemas multiprocessadores.

Inviável para uso geral.

Tentativa #2: Uma Flag Simples

Ideia: Usar uma variável flag para indicar se o lock está ocupado.

Falha: Condição de corrida entre testar a flag e definir a flag.

Duas threads podem achar que o lock está livre ao mesmo tempo

```
void lock(lock_t *mutex) {
 while (mutex->flag == 1) // 1. Testa a flag ;
 // Espera (spin)
 mutex->flag = 1; // 2. Define a flag para ocupado
}
void unlock(lock_t *mutex)
{ mutex->flag = 0; }
```

A Solução Real: Instruções Atômicas de Hardware

- Precisamos de suporte do hardware para resolver isso.
- Exemplo: Test-And-Set – instrução atômica que retorna o valor antigo e define um novo valor.
- A chave é a indivisibilidade da operação.

```
int TestAndSet(int *ponteiro_antigo, int novo_valor) {
  int valor_antigo = *ponteiro_antigo; // 1. Busca o valor antigo
  *ponteiro_antigo = novo_valor; // 2. Define o novo valor
  return valor_antigo; // 3. Retorna o valor antigo
}
```

Construindo um Spin Lock com Test-And-Set

- Código de spin lock usando Test-And-Set.
- A thread fica em loop chamando TestAndSet até obter o lock.
- Problema: Spinning desperdiça 100% da CPU enquanto espera.

```
void init(lock_t *lock) { lock->flag = 0; // 0: livre, 1: ocupado}

void lock(lock_t *lock) {
  // Continua testando e definindo até que o valor antigo seja 0
  while (TestAndSet(&lock->flag, 1) == 1)
    ; // Gira (spin) em um loop vazio
}

void unlock(lock_t *lock) {lock->flag = 0;}
```

Indo Além do Spinning: Suporte do SO

- Girar é ineficiente. Melhor abordagem é dormir se o lock estiver ocupado.
- Requer suporte do SO.
- Primitivas do SO: park() e unpark().

```
void lock(lock_t *lock) {
        while (TestAndSet(&flag, 1) == 1)
                yield(); // Cede a CPU em vez de girar
}
```

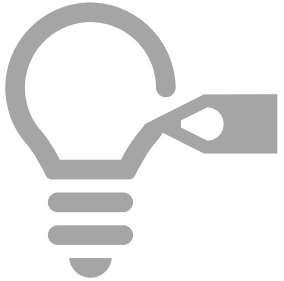
Lock Baseado em Fila (A Abordagem Moderna)

- Thread tenta adquirir o lock.
- Se ocupado, adiciona-se a uma fila de espera e chama park().
- unlock remove a próxima thread da fila e a acorda com unpark().
- Vantagens: eficiente e justo (FIFO).

O Desafio: Estruturas de Dados Thread-Safe

Objetivo:

Adicionar locks para garantir correção e alta performance.



Princípio:

'Evite otimização prematura'. Comece simples e correto.

Estudo de Caso 1: Contador Concorrente

- **Abordagem simples**: um lock global para proteger o contador.
- **Resultado**: funciona, mas escala mal com mais threads.

```c
typedef struct __counter_t {
  int value;
  pthread_mutex_t lock;
} counter_t;

void init(counter_t *c) {
  c->value = 0;
  pthread_mutex_init(&c->lock, NULL);
}

void increment(counter_t *c) {
  pthread_mutex_lock(&c->lock);
  c->value++;
  pthread_mutex_unlock(&c->lock);
}

void decrement(counter_t *c) {
  pthread_mutex_lock(&c->lock);
  c->value--;
  pthread_mutex_unlock(&c->lock);
}

int get(counter_t *c) {
  pthread_mutex_lock(&c->lock);
  int rc = c->value;
  pthread_mutex_unlock(&c->lock);
  return rc;
}
```

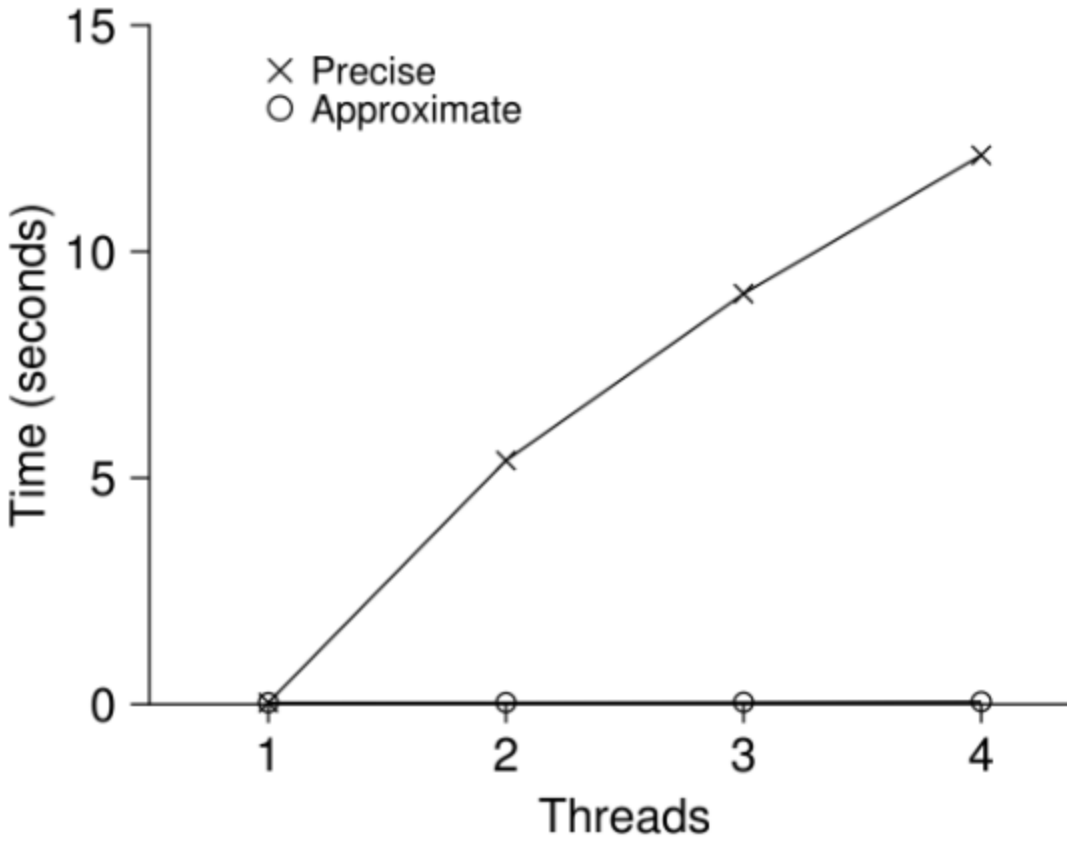
Contador Escalável: O Contador Aproximado

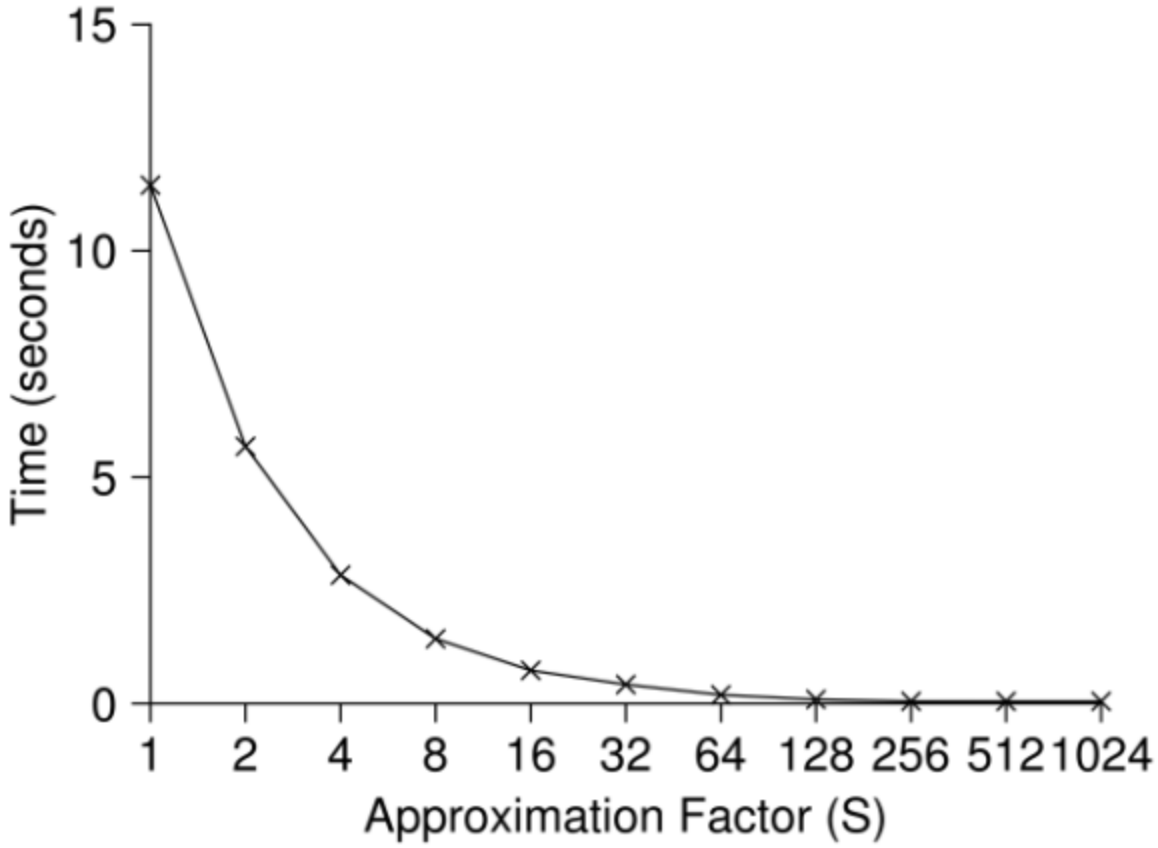
- Contador local por CPU + contador global.
- Threads atualizam contadores locais sem contenção.
- Periodicamente, valores locais são somados ao contador global.
- Resultado: alta performance e escalabilidade com pequena imprecisão.

```c
typedef struct __counter_t {
  int global; // global count
  pthread_mutex_t glock; // global lock
  int local[NUMCPUS]; // per-CPU count
  pthread_mutex_t llock[NUMCPUS]; // ...   and
locks
  int threshold; // update freq
} counter_t;

// init: record threshold, init locks, init
values
// of all local counts and global count
void init(counter_t *c, int threshold) {
  c->threshold = threshold;
  c->global = 0;
  pthread_mutex_init(&c->glock, NULL);
  int i;
  for (i = 0; i < NUMCPUS; i++) {
    c->local[i] = 0;
    pthread_mutex_init(&c->llock[i],   NULL);
}
}
```

```c
void update(counter_t *c, int threadID, int amt) {
  int cpu = threadID % NUMCPUS;
  pthread_mutex_lock(&c->llock[cpu]);
  c->local[cpu] += amt;
  if (c->local[cpu] >= c->threshold) {
    // transfer to global (assumes amt > 0)
    pthread_mutex_lock(&c->glock);
    c->global += c->local[cpu];
    pthread_mutex_unlock(&c->glock);
    c->local[cpu] = 0;
  }
  pthread_mutex_unlock(&c->llock[cpu]);
}

// get: just return global amount (approximate)
  int get(counter_t *c) {
  pthread_mutex_lock(&c->glock);
  int val = c->global;
  pthread_mutex_unlock(&c->glock);
  return val; // only approximate!
}
```


Desempenho Comparado





Estudo de Caso 2: Lista Ligada Concorrente

- **Abordagem Simples:**
  - Assim como no contador, a abordagem padrão é usar um único lock para proteger toda a lista durante operações de inserção ou busca.

```c
// basic node structure
typedef struct __node_t {
  int key;
  struct __node_t *next;
} node_t;

// basic list structure
typedef struct __list_t {
  node_t *head;
  pthread_mutex_t lock;
} list_t;

void List_Init(list_t *L) {
  L->head = NULL;
  pthread_mutex_init(&L->lock, NULL);
}
```

```c
int List_Insert(list_t *L, int key) {
  pthread_mutex_lock(&L->lock);
  node_t *new = malloc(sizeof(node_t));
  if (new == NULL) {
    perror("malloc");
    pthread_mutex_unlock(&L->lock);
    return -1; // fail
  }
  new->key = key;
  new->next = L->head;
  L->head = new;
  pthread_mutex_unlock(&L->lock);
  return 0; // success
}
```

```c
int List_Lookup(list_t *L, int key) {
  pthread_mutex_lock(&L->lock);
  node_t *curr = L->head;
  while (curr) {
    if (curr->key == key) {
      pthread_mutex_unlock(&L->lock);
      return 0; // success
    }
    curr = curr->next;
  }
  pthread_mutex_unlock(&L->lock);
  return -1; // failure
}
```

Estudo de Caso 2: Lista Ligada Concorrente

- **Abordagem Complexa:**
  - Um lock para cada nó da lista

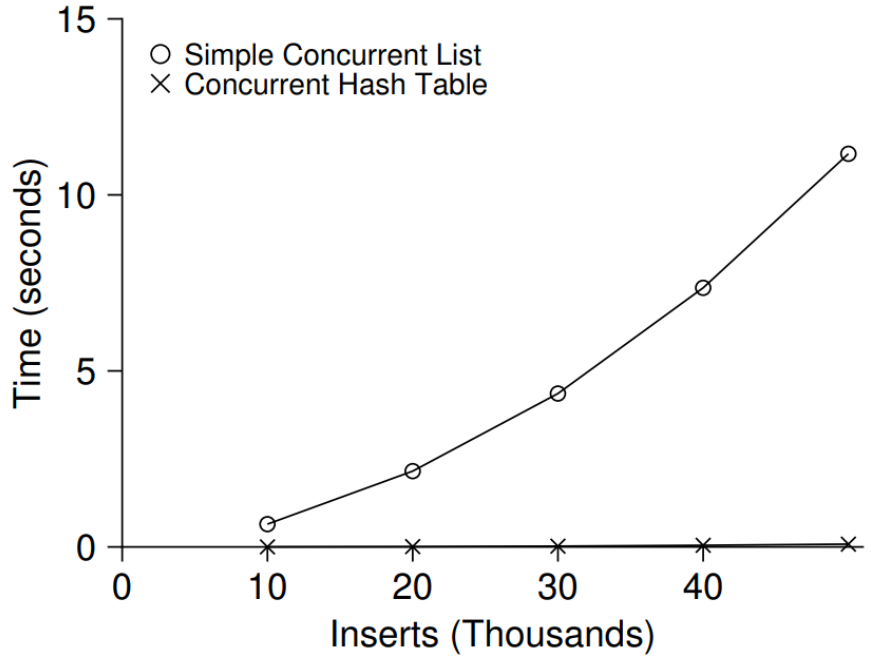
```
void List_Init(list_t *L) {
L->head = NULL;
pthread_mutex_init(&L->lock, NULL);
}

int List_Insert(list_t *L, int key) {
// synchronization not needed
node_t *new = malloc(sizeof(node_t));
if (new == NULL) {
perror("malloc");
return -1;
}
new->key = key;
// just lock critical section
pthread_mutex_lock(&L->lock);
new->next = L->head;
L->head = new;
pthread_mutex_unlock(&L->lock);
return 0; // success
}

int List_Lookup(list_t *L, int key) {
int rv = -1;
pthread_mutex_lock(&L->lock);
node_t *curr = L->head;
while (curr) {
if (curr->key == key) {
rv = 0;
break;
}
curr = curr->next;
}
pthread_mutex_unlock(&L->lock);
return rv; // now both success and failure
}
```

Estudo de Caso 3: Tabela Hash Concorrente

- Tabela hash é naturalmente paralelizável.
- Array de buckets (listas).
- Usar um lock por bucket em vez de um único lock global.

Performance da Tabela Hash

- Operações em buckets diferentes ocorrem em paralelo sem disputa por locks.
- Desempenho quase constante com mais threads e mais trabalho.



Usando Locks (Mutexes)

- `pthread_mutex_lock(&lock)` e `pthread_mutex_unlock(&lock)`.
- Inicializar:
  - PTHREAD_MUTEX_INITIALIZER ou pthread_mutex_init().
- Verificar códigos de retorno:
  - todas as funções pthread podem falhar.

Resumo e Diretrizes Finais

- Mantenha sincronização simples.
- Sempre inicialize locks e variáveis de condição.
- Verifique códigos de retorno.
- Cada thread tem sua própria pilha.
- Use variáveis de condição para sinalizar, não flags manuais.