Threads

Wedson Almeida Filho
Samuel Xavier de Souza

O que são *threads*?

- Similar a processos
- Cada thread tem o seu contexto
  - Registradores da CPU, incluindo *program counter* (PC)
  - Pilha da chamada
- Entretanto, algumas propriedades são compartilhadas
  - Memória
  - Tabela de descritores de arquivos
  - Tabela de sinais

Motivação

- Execução concorrente e possivelmente paralela
  - Por exemplo, servidor pode servir múltiplos clientes concorrentemente
  - Navegador pode baixar múltiplas imagens que compõem uma página concorrentemente
  - Processador de texto pode analisar ortografia enquanto continua respondendo ao usuário
- Desempenho
  - Mais barato criar uma thread do que um processo
  - Mais barato alternar entre threads do mesmo processo do que entre processos
  - Comunicação
- Compartilhamento de recursos
  - Arquivos abertos
  - Caches em memória

Criando *threads* com biblioteca pthreads

```c
#include <stdio.h>
#include <stdlib.h>
#include <pthread.h>

void *thread(void *arg)
{
   printf("Ola da thread\n");
}

int main(void)
{
   pthread_t th;
   int rc = pthread_create(&th, NULL, thread, NULL);
   if (rc != 0) {
       fprintf(stderr, "Erro ao criar thread: %d\n", rc);
       exit(1);
   }
   return 0;
}
```

Qual é a saída do programa anterior?

- Nenhuma
- Por quê?

Esperando a thread acabar

```c
#include <stdio.h>
#include <stdlib.h>
#include <pthread.h>

void *thread(void *arg)
{
   printf("Ola da thread\n");
}

int main(void)
{
   pthread_t th;
   int rc = pthread_create(&th, NULL, thread, NULL);
   if (rc != 0) {
           fprintf(stderr, "Erro ao criar thread: %d\n", rc);
           exit(1);
   }
   pthread_join(th, NULL);
   return 0;
}
```

Prática

- Definir uma variável global com tipo `uint64_t` com valor inicial zero
- Criar duas threads
  - Cada thread tem um laço que se repete 1.000.000 de vezes
  - Cada iteração do laço incrementa a variável global (`g = g + 1`)
- Esperar as duas threads acabarem
- Mostrar o valor final da variável global

Solução

```c
#include <stdio.h>
#include <pthread.h>
#include <stdint.h>

uint64_t valor = 0;

void *thread(void *arg)
{
	size_t i = 1000000;
	while (i--) {
		valor++;
	}
}

int main(void)
{
	pthread_t th1, th2;
	pthread_create(&th1, NULL, thread, NULL);
	pthread_create(&th2, NULL, thread, NULL);
	pthread_join(th1, NULL);
	pthread_join(th2, NULL);
	printf("Valor: %lu\n", valor);
	return 0;
}
```

Resultado da execução

```
$ ./03-inc
Valor: 1126235
$ ./03-inc
Valor: 1069826
$ ./03-inc
Valor: 1054619
$ ./03-inc
Valor: 1040078
$ ./03-inc
Valor: 1170460
$ ./03-inc
Valor: 1086741
```

O que acontece se compilar com otimização -O3?

- Por exemplo: `gcc -o 03-inc 03-inc.c -O3`

Resultado da execução

```
$ ./03-inc
Valor: 2000000
$ ./03-inc
Valor: 2000000
$ ./03-inc
Valor: 2000000
```

Instruções sem otimização

```
Dump of assembler code for function thread:
   0x00000000000011a9 <+0>:     endbr64
   0x00000000000011ad <+4>:     push   %rbp
   0x00000000000011ae <+5>:     mov    %rsp,%rbp
   0x00000000000011b1 <+8>:     mov    %rdi,-0x18(%rbp)
   0x00000000000011b5 <+12>:    movq   $0xf4240,-0x8(%rbp)
   0x00000000000011bd <+20>:    jmp    0x11d1 <thread+40>
   0x00000000000011bf <+22>:    mov    0x2e52(%rip),%rax        # 0x4018 <valor>
   0x00000000000011c6 <+29>:    add    $0x1,%rax
   0x00000000000011ca <+33>:    mov    %rax,0x2e47(%rip)        # 0x4018 <valor>
   0x00000000000011d1 <+40>:    mov    -0x8(%rbp),%rax
   0x00000000000011d5 <+44>:    lea    -0x1(%rax),%rdx
   0x00000000000011d9 <+48>:    mov    %rdx,-0x8(%rbp)
   0x00000000000011dd <+52>:    test   %rax,%rax
   0x00000000000011e0 <+55>:    jne    0x11bf <thread+22>
   0x00000000000011e2 <+57>:    nop
   0x00000000000011e3 <+58>:    pop    %rbp
   0x00000000000011e4 <+59>:    ret
```

Instruções com otimização

```
Dump of assembler code for function thread:
   0x0000000000001250 <+0>:     endbr64
   0x0000000000001254 <+4>:     addq   $0xf4240,0x2db9(%rip)        # 0x4018 <valor>
   0x000000000000125f <+15>:    ret
```

Solução com variável atômica

```c
#include <stdio.h>
#include <pthread.h>
#include <stdatomic.h>
#include <stdint.h>

_Atomic uint64_t valor = 0;

void *thread(void *arg)
{
	size_t i = 1000000;
	while (i--) {
		valor++;
	}
}

int main(void)
{
	pthread_t th1, th2;
	pthread_create(&th1, NULL, thread, NULL);
	pthread_create(&th2, NULL, thread, NULL);
	pthread_join(th1, NULL);
	pthread_join(th2, NULL);
	printf("Valor: %lu\n", valor);
	return 0;
}
```

Instruções sem otimização

```
Dump of assembler code for function thread:
   0x00000000000011a9 <+0>:     endbr64
   0x00000000000011ad <+4>:     push   %rbp
   0x00000000000011ae <+5>:     mov    %rsp,%rbp
   0x00000000000011b1 <+8>:     sub    $0x30,%rsp
   0x00000000000011b5 <+12>:    mov    %rdi,-0x28(%rbp)
   0x00000000000011b9 <+16>:    mov    %fs:0x28,%rax
   0x00000000000011c2 <+25>:    mov    %rax,-0x8(%rbp)
   0x00000000000011c6 <+29>:    xor    %eax,%eax
   0x00000000000011c8 <+31>:    movq   $0xf4240,-0x10(%rbp)
   0x00000000000011d0 <+39>:    jmp    0x11eb <thread+66>
   0x00000000000011d2 <+41>:    movl   $0x1,-0x1c(%rbp)
   0x00000000000011d9 <+48>:    mov    -0x1c(%rbp),%eax
   0x00000000000011dc <+51>:    cltq
   0x00000000000011de <+53>:    lock xadd %rax,0x2e31(%rip)        # 0x4018 <valor>
   0x00000000000011e7 <+62>:    mov    %rax,-0x18(%rbp)
   0x00000000000011eb <+66>:    mov    -0x10(%rbp),%rax
   0x00000000000011ef <+70>:    lea    -0x1(%rax),%rdx
   0x00000000000011f3 <+74>:    mov    %rdx,-0x10(%rbp)
   0x00000000000011f7 <+78>:    test   %rax,%rax
   0x00000000000011fa <+81>:    jne    0x11d2 <thread+41>
   0x00000000000011fc <+83>:    nop
   0x00000000000011fd <+84>:    mov    -0x8(%rbp),%rdx
   0x0000000000001201 <+88>:    sub    %fs:0x28,%rdx
   0x000000000000120a <+97>:    je     0x1211 <thread+104>
   0x000000000000120c <+99>:    call   0x1080 <__stack_chk_fail@plt>
   0x0000000000001211 <+104>:   leave
   0x0000000000001212 <+105>:   ret
```

Instruções com otimização

```
Dump of assembler code for function thread:
   0x0000000000001250 <+0>:     endbr64
   0x0000000000001254 <+4>:     mov    $0xf4240,%eax
   0x0000000000001259 <+9>:     nopl   0x0(%rax)
   0x0000000000001260 <+16>:    lock addq $0x1,0x2daf(%rip)        # 0x4018 <valor>
   0x0000000000001269 <+25>:    sub    $0x1,%rax
   0x000000000000126d <+29>:    jne    0x1260 <thread+16>
   0x000000000000126f <+31>:    ret
```

E se houver mais de uma operação?

- É possível sincronizar de outras formas
- Por exemplo com exclusão mútua
- Na biblioteca pthreads, `pthread_mutex_t`

Exemplo com mutex

```c
#include <stdio.h>
#include <pthread.h>
#include <stdint.h>

uint64_t valor = 0;
pthread_mutex_t mutex = PTHREAD_MUTEX_INITIALIZER;

void *thread(void *arg)
{
	size_t i = 1000000;
	while (i--) {
		pthread_mutex_lock(&mutex);
		valor++;
		pthread_mutex_unlock(&mutex);
	}
}

int main(void)
{
	pthread_t th1, th2;
	pthread_create(&th1, NULL, thread, NULL);
	pthread_create(&th2, NULL, thread, NULL);
	pthread_join(th1, NULL);
	pthread_join(th2, NULL);
	printf("Valor: %lu\n", valor);
	return 0;
}
```