Programação com *threads*

Gonzalo Travieso

2020

Outline

1 Processos e threads

2 Memória no programa

3 Compartilhamento

4 Disputa

Tópicos

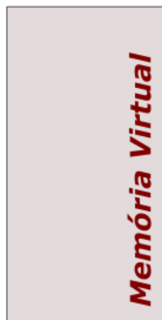
1 Processos e threads

2 Memória no programa

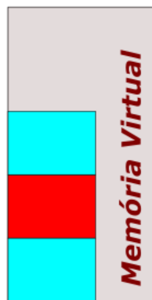
3 Compartilhamento

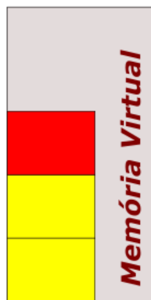
4 Disputa

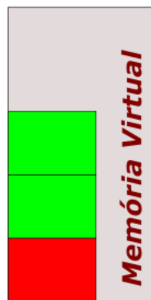
Processos



Compartilhamento entre processos



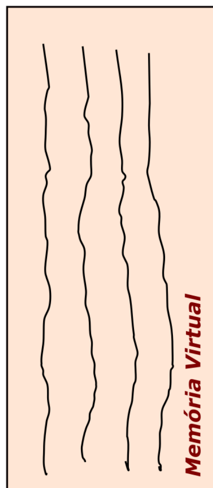




Threads

```
s = 0;
for (i = 0; i < 100; ++i) {
    s += a[i];
}
```

```
s = 0
i = 0
0 < 100 ?
lê s
calcula s + a[0]
escreve em s
i = 1
volta
1 < 100 ?
lê s
calcula s + a[1]
escreve em s
i = 2
volta
2 < 100 ?
lê s
calcula s + a[2]
escreve em s
i = 3
volta
...
```



Tópicos

1 Processos e threads

2 Memória no programa

3 Compartilhamento

4 Disputa

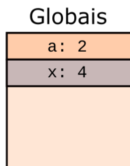
Pilha

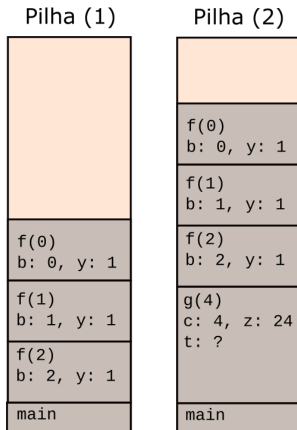
```
int const a = 2;
int x = 4;

int f(int b) {
    int y = 1;
    if (b > 0) {
       y = b * f(b-1); (1) (2)
    return y;
}

int g(int c) {
    int z = f(c);
    int t = f(c/2); (2)
    return z / t;
}

int main(int, char *[]) {
    std::cout << f(a) << std::endl; (1)
    std::cout << g(x) << std::endl; (2)
    return 0;
}
```





Heap

- Usada em espaço alocado dinamincamente.
  - `malloc` no C.
  - `new` no C++.
  - Estruturas de dados com alocação dinâmica interna, exemplo `std::vector<double> v(100)`.
- Usada também para criar espaço de memória que não está associado com o tempo de vida de uma função específica.
- Continuam válidos até serem desalocados.

Variáveis estáticas

- C e C++ permitem variáveis estáticas:

```
double exemplo(double x) {
  static int num_chamadas = 0;
  static double total = 0;
  ++num_chamadas;
  total += x;
  return total / num_chamadas;
}
```

- Variáveis estáticas são armazendas como variáveis globais.
- A única diferença é que elas são visíveis apenas no escopo onde são definidas.

Tópicos

1 Processos e threads

2 Memória no programa

3 Compartilhamento

4 Disputa

Objetos privados e compartilhados

- Um objeto acessado apenas por uma *thread* é um objeto privado.
- Um objeto acessado por mais do que uma *thread* é um objeto compartilhado.

Compartilhamento e escopo

- Variáveis globais estão sempre no escopo de todas as *threads* e porisso podem ser compartilhadas diretamente (inclusive de forma inadvertida).
- Variáveis locais das funções são próprias de cada execução da função. Elas somente poderão ser compartilhadas diretamente caso as threads tenham sido criadas no contexto da execução dessa função.
- Se threads distintas fazem separadamente chamadas para a mesma função, cada uma irá criar uma instância distinta das variáveis locais na sua pilha (cada thread tem uma pilha distinta).
- Variáveis estáticas têm apenas um cópia, portanto se várias threads fizerem chamadas distintas a uma função com variáveis estáticas, elas estarão compartilhando as variáveis estáticas.

Compartilhamento de referências

- Objetos na heap podem ser acessados de qualquer escopo que diponha de um ponteiro para eles. Portanto podem ser compartilhados por qualquer thread que tenha um ponteiro para eles. Eles só podem ser desalocados quando nenhuma thread precisar deles.
- Objetos locais de uma thread podem ser compartilhados com outra thread desde que ela disponha de um ponteiro para esse objeto. Mas nesse caso, é importante que o objeto continue existindo enquanto a outra thread precisar dele (quer dizer, a função que é dona do objeto não pode terminar sua execução enquanto a outra thread precisar do objeto).

Compartilhamento de leitura e escrita

- Se um objeto é compartilhado por várias threads, mas todas elas apenas lêm o seu valor, então esse objeto é compartilhado para leitura.
- Se pelo menos uma thread faz alteração (ou pode fazer alteração) no objeto compartilhado, então este objeto é compartilhado para escrita

Desempenho de compartilhamento

Lembrando o que discutimos ao falar de coerência de cache:

- Acesso a objetos não-compartilhados é eficiente.
- Acesso a objetos compartilhados para leitura é eficiente.
- Acesso a objetos compartilhados para escrita tem menor eficiência.

Tópicos

1 Processos e threads

2 Memória no programa

3 Compartilhamento

4 Disputa

Região crítica

- Podemos separar a operação das threads em dois tipos:
  - Quando as threads estão apenas acessando objetos privados ou compartilhados para leitura. Esta é a chamada região normal.
  - Quando as threads estão acessando objetos compartilhados para escrita. Esta é a região crítica.
- Lembre-se que a thread estará em região crítica se ela acessar objetos compartilhados para escrita, mesmo que ela esteja apenas fazendo leituras nesses objetos.
- Para cada conjunto de objetos que são compartilhados para escrita simultaneamente, teremos correspondentes regiões críticas nas threads que os compartilham.
- Threads que estão em suas regiões normais não interferem com as outras threads.
- Threads na região crítica podem interferir com outras threads.

Execução das threads

- O sistema operacional determina dinamicamente a execução das threds.
- Escolhe quando as threads serão colocadas para execução e por quanto tempo (escalonamento).
- Portanto, o programador não tem conhecimento, ao escrever o programa, a ordem em que as operações em regiões críticas das threads serão executadas.
- Por isso ele tem que assumir que qualquer ordem e velocidade relativa de execução entre as threads é possível.

Disputa

- Dizemos que um código tem disputa crítica ou simplesmente disputa se a ordem de execução das seções críticas das threads influencia no resultado final da execução.
- Como o programador não conhece a ordem de antemão, esse tipo de código tem que ser considerado errôneo.
- Boa parte da dificuldade da programação com memória compartilhada consiste em controlar corretamente a execução das seções críticas para evitar disputa.

Disputa: exemplo

Thread 1

```
// ...
y = 2 * x;
// ...
std::cout << y << std::endl;
```

Thread 2

```
// ...
x = 10;
// ...
x = -10;
// ...
```