Insper

13 - Sincronização

Super Computação 2018/2

Igor Montagner, Luciano Soares

Entrega: 24/09

Parte 1 - Sincronização em C++ threads

No roteiro 03 criamos threads e as executamos usando o pacote `<threads>` adicionado após C++11. Neste módulo do curso vamos deixar o OpenMP de lado e voltar a usar esta API.

Exercício: crie um arquivo *thread1.cpp* com uma função `main` que cria 4 threads. Cada thread deverá rodar uma função que imprime um id entre 0 e 3 passado pelo `main`.

Se for necessário, revise o conteúdo do roteiro 03.

Mutex

Você deve ter notado que, por vezes, os prints do seu programa aparecem embaralhados. Já sabemos que isto ocorre pois a saída padrão é um recurso que está sendo compartilhado por várias threads. Na expositiva vimos que devemos guardar a seção que utiliza recursos compartilhados usando um *Mutex*.

Exercício: C++11 possui um objeto do tipo `std::mutex`, mas ele não é muito usado. Em contrapartida, são usados `std::lock_guard` e `std::unique_lock`. Por que?

Exercício: conserte o problema dos prints embaralhados usando sua pesquisa acima. Qual a vantagem da técnica usada em comparação do uso direto de um `std::mutex`?

Dica: se você está com problemas em passar seu mutex para função da thread, isto ocorre pois ele não pode ser *copiado*, apenas passado como referência para outras threads. Para usar referências em conjunto com o construtor de `std::thread` você precisa passar seu mutex usando `std::ref` (documentação - seção Notes).

Dica 2: se você está usando um mutex global, pare de fazer isto e veja a dica 1.

Condition Variable

Neste módulo estamos tratando de paralelismo **por tarefas**. Agora nosso foco não é mais dividir os dados em várias partes e processá-los em paralelo, mas sim processar tarefas inteiras (possivelmente complexas) de maneira paralela **quando possível**. C++11 possui uma implementação de variáveis de condição fácil de usar. Nesta seção iremos sincronizar tarefas que dependem parcialmente umas das outras (ou seja, parte de uma tarefa depende de um resultado de outra enquanto o restante é independente). Ao executarmos as partes independentes de maneira paralela podemos obter grandes ganhos de desempenho em processos longos.

Trabalharemos primeiro no arquivo *rdv.cpp*. Queremos sincronizar as duas funções para que elas se sincronizem em um certo ponto e depois progridam de maneira independente.

Exercício: Foi feita uma tentativa de uso de variáveis de condição para resolver este problema, porém ela está dando resultados estranhos. Execute *rdv.cpp* diversas vezes e relate qual o problema que ocorre. Você consegue entender a causa deste problema?

Uma maneira de consertar o problema acima é pedir para a variável de condição checar *explicitamente* a condição que ela guarda. Ou seja, ela é capaz de checar se a condição é verdadeira quando *wait* é chamado e **não** bloquear. A função *notify_one* neste caso acorda uma thread para que ela cheque *explicitamente* se a condição é verdadeira.

Para isto devemos chamar uma versão de `wait` que aceita um *lambda* que faz a checagem da condição, como no exemplo abaixo. Ao ser notificada (e na primeira chamada a `wait`) a função é executada e, se a condição for verdadeira o código continua a ser executado.

```
cv.wait(lk, []{return cond == true; });
```

A sintaxe de *lambda* é um pouco estranha, então é sugerido olhar este link antes de prosseguir.

Exercício: Modifique *rdv.cpp* para ter o comportamento correto usando a técnica descrita acima.

Dica: antes de fazer este exercício leia a documentação sobre variáveis de condição.

Vamos agora trabalhar no arquivo *cond_var.cpp*, em que usaremos variáveis de condição para sincronizar o progresso de três funções cujas tarefas podem ser parcialmente feitas em paralelo.

Exercício: quanto tempo a execução sequencial das três funções demorará?

Exercício: algumas partes das tarefas poderiam ser feitas de modo concorrente. Desenhe abaixo um gráfico mostrando qual seria a execução de menor tempo das três tarefas acima.

Este tipo de processamento é chamado de **pipeline**: cada tarefa avisa as tarefas que dependem dela que seu trabalho foi concluído. Se duas tarefas são independentes elas podem rodar em paralelo. Note que usamos referências constantes (`int const&`) para os resultados calculados por outras tarefas. Isto significa que uma vez calculado este resultado não pode ser modificado por outras tarefas que dependem dele.

Vamos agora nos concentrar no uso de variáveis de condição para que as threads executem da maneira desenhada no exercício anterior. Precisamos nos atentar aos seguintes detalhes:

1. Cada variável de condição precisa de um *mutex* associado a ela e este *mutex* protege o acesso às variáveis usadas na checagem de sua condição.
2. Uma variável de condição é responsável por sinalizar que um recurso específico foi produzido. Se existe mais de um recurso possivelmente será necessário usar mais de uma variável de condição.
3. Não se esqueça que variáveis de condição não guardam *notify*s. Ou seja, se não tem ninguém esperando *notify_one* é perdido, o que pode implicar em um programa em *deadlock*.

Exercício: Modifique o programa *cond_var.cpp* para que a execução seja feita da maneira desenhada no exercício anterior.

Semaphores

C++ não possui uma implementação padrão de semáforos, mas é bastante fácil criar uma. Sua tarefa neste exercício será criar uma implementação de semáforos usando as primitivas de sincronização acima. Você deverá implementar uma classe *Semaphore* contendo os métodos `void acquire()` e `void release()`.

Exercício: Quais primitivas de sincronização são necessárias para implementar um semáforo? Como elas devem ser usadas?

Exercício: escreva, em pseudo-código, qual deverá ser o comportamento do método `void acquire()`.

Exercício: escreva, em pseudo-código, qual deverá ser o comportamento do método `void release()`.

Exercício: implemente a classe *Semaphore* e use sua implementação para criar uma versão mais simples do *rdv.cpp*.

Atenção: existem implementações prontas de semáforos em C++ pela internet. Entregar um código que não corresponda às respostas acima pode implicar em nota 0 nesta atividade.

Parte 2 - Produtor Consumidor

Na expositiva vimos o modelo de computação concorrente *Produtor-Consumidor*. Usando os conhecimentos adquiridos até agora, implemente este modelo usando C++11 threads. Seu programa deve possuir:

1. uma fila de tarefas contendo inteiros;
2. uma função `void produce()` que irá produzir inteiros aleatórios entre 0 e 100;
3. uma função `void consume()` que irá tirar inteiros da fila de tarefas e imprimir seu quadrado;

Para melhorar seus testes você deve incluir no seu código alguns `sleep` para que fique mais claro o que está acontecendo.