

22 - POSIX Threads II

Igor Montagner

Parte 0 - o programa `soma_global.c`

Vamos agora trabalhar em cima do arquivo `soma_global.c`. Para fins didáticos, estamos atualizando diretamente a variável `soma_total` dentro do `for`.

Exercício: Complete as partes faltantes e rode o programa. Ele dá os resultados esperados? Analise tanto em termos de tempo de execução (deve ser um quarto do sequencial) quanto em relação ao resultado final da soma.

Exercício: Os resultados acima estarão errados. Você consegue explicar por que?

Parte 1 - proteção de dados usando `mutex`

Vamos agora trabalhar agora para corrigir este erro! Lembrando da aula, as operações possíveis são as seguintes:

- `lock` - se tiver destravado, trava e continua; se não estiver espera.
- `unlock` - se tiver a trava, a destrava e permite que outras tarefas travem.

Note que não existe garantia de ordem! Ou seja, se tiverem vários processos esperando por um `mutex` qualquer um deles pode receber o acesso. Inclusive, uma thread pode esperar “para sempre” e nunca receber o acesso. Não é provável, mas é possível.

! Você pode precisar instalar o pacote `manpages-posix-dev` para obter as páginas do manual usadas neste roteiro.

Exercício: Identifique no seu código quais linhas compõe a região crítica e onde deveriam estar as diretivas `lock` e `unlock`. Coloque um comentário nestas linhas.

Exercício: O mutex precisa ser criado e inicializado. Onde isto deve ser feito? Como ele pode ser recebido pela função da thread?

Exercício: Consulte os seguintes manuais e use-os para consertar seu programa usando mutexes.

- `pthread_mutex_init`
- `pthread_mutex_lock`
- `pthread_mutex_unlock`

Exercício: Seu código arrumado funciona? Meça o tempo e compare com o original (2 threads).

Exercício: Seu programa ficou lento? Isto ocorre pois usar `mutex` é muito caro. Conserte seu código para minimizar o número de `lock` e `unlock`.

Como vemos no exemplo acima, usar primitivas de sincronização é caro! O modelo que fizemos originalmente (na parte 4 do roteiro anterior) é chamado de *fork-join* e será foco da primeira parte da disciplina de SuperComputação.

Parte 2 - semáforos

Usamos `mutex` quando precisamos criar regiões de **exclusão mútua** onde somente uma thread pode entrar por vez. Esta restrição é muito forte e não contempla outro caso muito comum em programação concorrente: sincronização de threads. Ou seja, o objetivo não é proteger o acesso a dados compartilhados, mas sim impor restrições no progresso das threads de maneira que elas estejam sempre em uma situação válida. Para isto trabalharemos com **semáforos**, que são um mecanismo de sincronização mais sofisticado e geral usado para que threads sincronizem seu progresso e possam executar **em paralelo**.

Definição: duas tarefas podem ser feitas em paralelo se

1. elas não compartilham absolutamente nenhuma informação.
2. elas compartilham informação mas possuem **mecanismos de sincronização** de tal maneira que **toda ordem de execução possível** de suas instruções resulte no mesmo resultado final.

Semáforos ajudam a criar programas no segundo caso. Nesta aula iremos olhar o caso mais simples de sincronização: duas threads combinam de só progredirem quando chegarem em um certo ponto do programa.

Rendez-vous

A expressão *Rendez-vous* significa, literalmente, *encontro* em francês. Ela é usada para marcar um horário para duas ou mais pessoas se encontrarem. No contexto de sincronização de tarefas, ele também é usado para nomear o problema de sincronização mais simples: duas threads rodando funções distintas precisam se sincronizar no meio de suas funções.

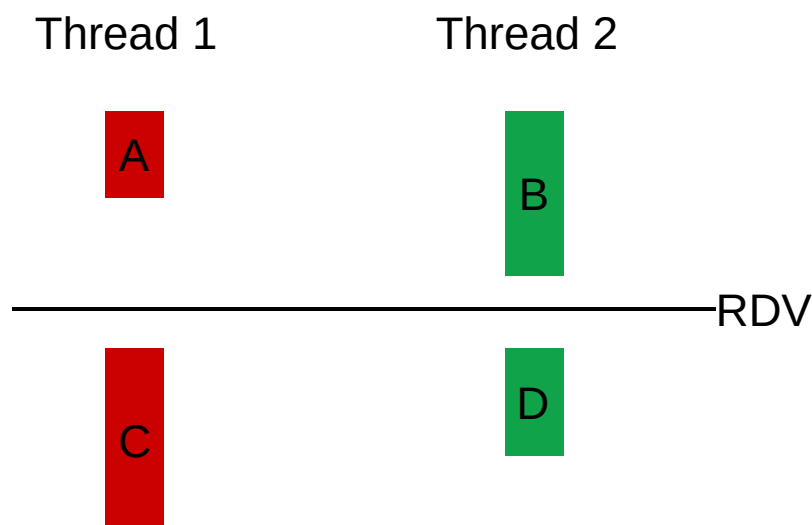


Figure 1: Tarefas sincronizadas usando um RDV

As partes A e B podem ser feitas em qualquer ordem, mas ambas obrigatoriamente devem ocorrer antes de iniciar a execução de C e D. Note que C e D também podem ser feitas em qualquer ordem.



Quando dizemos que duas tarefas podem ser feitas em qualquer ordem não quer dizer que elas possam ser feitas em paralelo! Apenas estamos dizendo que A inteira pode ocorrer antes ou depois de B inteira e os resultados serão os mesmos.

Exercício: Marque abaixo as ordens de execução possíveis para as partes A, B, C e D.

1. A C B D
2. A B C D
3. B D A C
4. B A D C
5. B A C D

Vamos fazer a solução do RDV no papel primeiro.

Inicialização: Preencha aqui quantos semáforos serão usados, seus nomes e valores iniciais.

Sua solução: Indique abaixo em quais quadrados azuis você usaria seus semáforos para resolver o RDV. Você pode usar mais de um semáforo em um mesmo quadrado e pode deixar os outros vazios.

Semáforos POSIX

A página `sem_overview` do manual contém um resumo do uso de semáforos. A partir de seu conteúdo responda as questões abaixo.

Exercício: Qual o tipo de variável usada para guardar um semáforo? Quais funções são usadas para criar e destruir cada tipo de semáforo?

Exercício: Quais as funções usadas para incrementar e decrementar um semáforo?

Exercício: Implemente (do zero) um programa que cria duas threads e as sincroniza usando *RDV*. Ambas deverão fazer um `print` antes e um depois do ponto de encontro.

Parte 3 - Entrega

Exercício: adapte o exercício da soma do vetor para calcular também a variância. Percebam que agora temos duas partes que tem uma relação de dependência:

1. Computar a soma (divida em duas partes)
2. Computar a variância (que depende da soma)

Use *RDV* para sincronizar as duas threads e você deverá implementar tudo na função

```
void *computa_media_variancia(void *)
```

Salve sua solução em um arquivo `var1.c` e entregue via blackboard.