Relatório Lab 4 - Computação Concorrente

Prof: Silvana Rossetto

Aluno: Tales Moreira

DRE: 119047549

Atividade 1

Identificação do requisito lógico/condicional da aplicação

O objetivo deste programa é controlar a ordem de execução das threads de modo que as mensagens "HELLO" sejam impressas antes da mensagem "BYEBYE". Mais especificamente, as duas threads que imprimem "HELLO" (threads A) devem ser executadas antes da thread que imprime "BYEBYE" (thread B).

Thread A:

- Imprime "HELLO".
- Incrementa estado dentro de uma região crítica protegida pelo mutex.
- Se o estado for igual a 2 (ou seja, as duas threads A incrementaram o estado), sinaliza a condição usando `pthread_cond_signal(&cond);`.

Thread B:

- Tenta adquirir o mutex e verifica se o estado é menor que 2.
    - Se for, a thread entra em espera na variável de condição `cond` usando `pthread_cond_wait(&cond, &mutex);`.
- Após ser sinalizada, ou se o estado já for maior ou igual a 2, imprime "BYEBYE".
- Libera o mutex e termina.

Fluxo de Execução Esperado:

1. As duas threads A iniciam e imprimem "HELLO", incrementando o estado.
2. Quando o estado atinge 2, a thread A sinaliza a variável de condição.
3. A thread B, que possivelmente estava em espera, é acordada e imprime "BYEBYE".

Verificação da Ordem de Impressão:

Para verificar se o requisito lógico é sempre cumprido, podemos executar o programa várias vezes e observar a saída. Ao fazê-lo, percebe-se que a lógica é seguida.

Atividade 2:

1. Leitura e verificação do programa para atender aos requisitos

Requisitos da aplicação:

- **Requisito 1:** A ordem em que as threads 1 e 2 imprimem suas mensagens não importa, mas **ambas devem sempre imprimir suas mensagens antes da thread 3**.

- **Requisito 2:** A ordem em que as threads 4 e 5 imprimem suas mensagens não importa, mas **ambas devem sempre imprimir suas mensagens depois da thread 3**.

Análise do código:

O programa cria cinco threads:

1. **Thread 0:** Executa a função `chegada` e imprime "Oi José!".
2. **Thread 1:** Executa a função `chegada` e imprime "Oi Maria!".
3. **Thread 2:** Executa a função `permanencia` e imprime "Sentem-se por favor.".
4. **Thread 3:** Executa a função `saida` e imprime "Tchau José!".
5. **Thread 4:** Executa a função `saida` e imprime "Tchau Maria!".

Sincronização implementada:

- **Variáveis globais:**
  - `int chegadas = 0;` — Conta quantas threads de chegada finalizaram.
  - `int sentados = 0;` — Indica se a mensagem "Sentem-se por favor." foi impressa.
  - `pthread_mutex_t x_mutex;` — Mutex para garantir exclusão mútua.
  - `pthread_cond_t chegada_cond;` — Variável de condição para sincronizar a chegada das threads.
  - `pthread_cond_t sentado_cond;` — Variável de condição para sincronizar a saída das threads.
- **Função `chegada`:**
  - Imprime "Oi José!" ou "Oi Maria!" dependendo do `thread_id`.
  - Incrementa `chegadas`.
  - Se `chegadas == 2`, sinaliza `chegada_cond` para acordar a thread `permanencia`.
- **Função `permanencia`:**
  - Aguarda até que `chegadas == 2` usando `pthread_cond_wait`.
  - Após ser sinalizada, imprime "Sentem-se por favor.".
  - Incrementa `sentados` e realiza um `pthread_cond_broadcast` em `sentado_cond` para acordar as threads de saída.
- **Função `saida`:**
  - Aguarda até que `sentados == 1` usando `pthread_cond_wait`.
  - Após ser sinalizada, imprime "Tchau José!" ou "Tchau Maria!" dependendo do `thread_id`.

Verificação dos requisitos:

- **Requisito 1:** As threads de chegada (threads 0 e 1) sempre imprimem suas mensagens antes da thread `permanencia` (thread 2) devido à sincronização com `chegada_cond`.

- **Requisito 2:** As threads de saída (threads 3 e 4) só imprimem suas mensagens após a thread `permanencia` ter impresso "Sentem-se por favor.", garantido pela sincronização com `sentado_cond`.

Conclusão:

O programa atende aos requisitos especificados, pois a sincronização implementada assegura a ordem necessária de execução das threads.

2. Alteração do código para adicionar mensagens de Log

Código no GitHub anexado à tarefa.

Atividade 3:

2. Execução do programa sem a barreira (barreira comentada)

Passos:

- **Executar o programa com a barreira comentada.**

Comportamento Observado:

- As threads não estão sincronizadas.
- Cada thread executa suas iterações de forma independente.
- A saída do programa mostra os passos das threads intercalados de forma não ordenada.

Análise:

- Sem a barreira, não há garantia de que as threads concluam uma iteração antes de outra thread avançar para a próxima.
- As threads podem estar em diferentes passos ao mesmo tempo.
- Isso é esperado em programas concorrentes sem sincronização específica.

3. Execução do programa com a barreira (barreira descomentada)

Comportamento Observado:

- As threads estão sincronizadas.
- Todas as threads completam um passo antes de qualquer thread avançar para o próximo passo.
- A saída do programa mostra que as threads estão no mesmo passo ao mesmo tempo.

Análise:

- A barreira força as threads a esperarem umas pelas outras após cada passo.
- Nenhuma thread pode iniciar o passo `i+1` antes que todas as threads tenham completado o passo `i`.
- Isso garante que as threads avancem em conjunto através das iterações.

Conclusão e Avaliação dos Resultados

Sem a barreira:

- As threads operam de forma independente.
- Não há sincronização entre as threads.
- Possível inconsistente em aplicações que dependem da conclusão conjunta de etapas.

Com a barreira:

- As threads são sincronizadas após cada iteração.
- A barreira garante que todas as threads estão no mesmo estado antes de prosseguir.
- Útil em situações onde é necessário que todas as threads concluam uma etapa antes de avançar.

Atividade 4:

Código no GitHub anexado à tarefa.