Gerenciamento de Eventos em Ambiente Multithread - Uso de Condições em Código Sincronizado

STATUS **DESENVOLVIDA**

Objetivo do Projeto

Este relatório documenta os testes realizados para avaliar diferentes cenários de sincronização no gerenciamento de eventos em um ambiente multithread. Três cenários foram testados: sem sincronização, com sincronização usando wait() e notify(), e com múltiplos produtores e consumidores.

🗒️ Cenários teste

Cenário 1: Sem Uso de Condições Sincronizadas

Objetivo: Observar o comportamento do sistema sem sincronização, analisando os efeitos da concorrência.

Modificação no Código

Removida a sincronização e o uso de wait() e notify(). O produtor adiciona eventos sem verificação de limite. O consumidor consome eventos sem verificação de disponibilidade.

Resultados

Condição de Corrida: O tamanho do armazenamento pode exceder o limite ou ficar negativo. Inconsistência de Dados: Eventos podem ser perdidos ou consumidos incorretamente. Exceções: Possíveis NullPointerException quando o consumidor tenta acessar uma lista vazia.

=======

Cenário 2: Com Uso de Condições Sincronizadas

Objetivo: Garantir a consistência do armazenamento e evitar condições de corrida usando wait() e notify().

Modificação no Código

Uso de synchronized nos métodos set() e get().

Implementação de wait() para impedir que o produtor exceda o limite do armazenamento. Implementação de wait() para evitar que o consumidor acesse uma lista vazia. Uso de notify() para acordar threads aguardando operações pendentes.

Resultados

Controle de Concorrência: O tamanho do armazenamento nunca excede o limite e nunca fica negativo. Consistência de Dados: Todos os eventos são corretamente armazenados e consumidos. Sem Exceções: Nenhuma falha devido à sincronização adequada.

=======

Cenário 3: Aumentando o Número de Produtores e Consumidores

Objetivo: Avaliar a escalabilidade do sistema com múltiplas threads operando simultaneamente.

Modificação no Código

Criados três produtores e três consumidores, cada um executando em uma thread separada. Mantida a sincronização com wait() e notify().

Resultados

Escalabilidade: O sistema continua funcionando corretamente com múltiplas threads. Controle de Concorrência: Nenhuma inconsistência no armazenamento. Consistência de Dados: Todos os eventos foram adicionados e consumidos sem erros.

🔧 Conclusão:

🔗

Os testes demonstraram a importância da sincronização em programas multithreaded. O Cenário 1 evidenciou problemas de concorrência, enquanto o Cenário 2 solucionou essas questões por meio de wait() e notify(). No Cenário 3, verificamos que a solução sincronizada é escalável e funciona corretamente com vários produtores e consumidores. Recomendação: Para sistemas que envolvem manipulação concorrente de recursos compartilhados, é essencial utilizar técnicas adequadas de sincronização para garantir a consistência e evitar condições de corrida.

💻 Desenvolvedor

🔗

Marcio Fonseca
👨‍💻

=======