

RELATÓRIO DA AVALIAÇÃO DA DISCIPLINA DE SISTEMAS OPERACIONAIS: SINCRONIZAÇÃO

Aluno: Pablo Durkheim Fernandes do Nascimento

A avaliação consiste em resolver apenas uma das questões disponibilizada em uma lista, para este relatório foi escolhido a questão **02 Variável de condição a partir de um semáforo.**

Objetivo: implementar a API para variáveis de condição descrita no item 1 da lista. A diferença entre as questões 1 e 2 é o ponto de partida: na primeira questão a API de mutex está disponível, enquanto que na segunda a API de semáforo está disponível.

Resolução:

```
#include <stdio.h>
#include <pthread.h>
#include <semaphore.h>

struct condvar {
    sem_t semaforo;    // Semáforo que controla o acesso à seção crítica (inicializado como 1,
    indicando que está "acordado")
    sem_t wait_sem;    // Semáforo usado para sincronizar as threads (inicializado como 0,
    indicando que está "dormindo")
    int waiting_threads; // Número de threads atualmente esperando
};

void condvar_init(struct condvar *c) {
    sem_init(&c->semaforo, 0, 1); // Inicializa o semáforo como 1 (acordado, permitindo o
    acesso à seção crítica)
    sem_init(&c->wait_sem, 0, 0); // Inicializa o semáforo de espera como 0 (dormindo,
    nenhuma thread está esperando inicialmente)
    c->waiting_threads = 0;    // Inicialmente, nenhum thread está esperando
}

void condvar_wait(struct condvar *c) {
    sem_wait(&c->semaforo);    // Adquire o semáforo, bloqueando outras threads de
    entrarem na seção crítica (torna-se "dormindo")
    c->waiting_threads++;    // Incrementa o número de threads esperando
    sem_post(&c->semaforo);    // Libera o semáforo para outras threads

    sem_wait(&c->wait_sem);    // Aguarda até que seja acordado por outra thread (torna-se
    "dormindo")
}

void condvar_signal(struct condvar *c) {
```

```

    if (c->waiting_threads > 0) {
        sem_post(&c->wait_sem);    // Acorda uma thread que está esperando (torna-se
"acordado")
        c->waiting_threads--;    // Decrementa o número de threads esperando
    }
}

void condvar_broadcast(struct condvar *c) {
    while (c->waiting_threads > 0) {
        sem_post(&c->wait_sem);    // Acorda todas as threads que estão esperando (tornam-se
"acordadas")
        c->waiting_threads--;    // Decrementa o número de threads esperando
    }
}

// Função para uso em uma thread
void *thread_func(void *arg) {
    struct condvar *cv = (struct condvar *)arg;

    //sleep(2); // Simula algum trabalho antes de entrar na seção crítica

    printf("Thread entrou na seção crítica.\n");

    // Realiza algum trabalho na seção crítica

    printf("Thread saiu da seção crítica.\n");

    condvar_signal(cv); // Acorda uma thread que pode estar esperando

    return NULL;
}

int main() {
    pthread_t threads[3];
    struct condvar cv;
    condvar_init(&cv);

    // Criação de 3 threads
    for (int i = 0; i < 3; i++) {
        pthread_create(&threads[i], NULL, thread_func, &cv);
    }

    // Aguarda o término de cada thread criada
    for (int i = 0; i < 3; i++) {

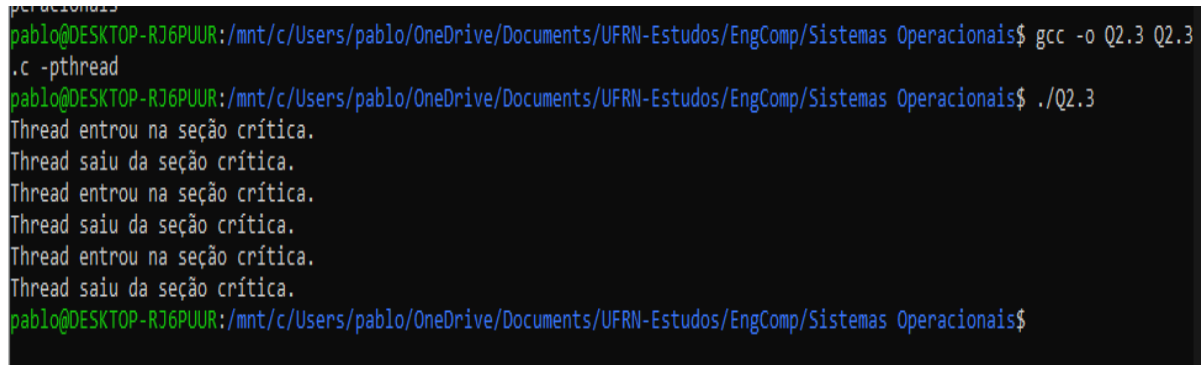
```

```

        pthread_join(threads[i], NULL);
    }

    return 0;
}

```



```

pablo@DESKTOP-RJ6PUUR:/mnt/c/Users/pablo/OneDrive/Documents/UFRN-Estudos/EngComp/Sistemas Operacionais$ gcc -o Q2.3 Q2.3.c -pthread
pablo@DESKTOP-RJ6PUUR:/mnt/c/Users/pablo/OneDrive/Documents/UFRN-Estudos/EngComp/Sistemas Operacionais$ ./Q2.3
Thread entrou na seção crítica.
Thread saiu da seção crítica.
Thread entrou na seção crítica.
Thread saiu da seção crítica.
Thread entrou na seção crítica.
Thread saiu da seção crítica.
pablo@DESKTOP-RJ6PUUR:/mnt/c/Users/pablo/OneDrive/Documents/UFRN-Estudos/EngComp/Sistemas Operacionais$

```

O código acima funciona da seguinte forma: é implementado uma variável de condição utilizando semáforos em um ambiente multi-threading em C. A estrutura **condvar** inclui dois semáforos e uma variável de controle. O semáforo **semaforo** regula o acesso à seção crítica, enquanto o **wait_sem** sincroniza as threads. A função **condvar_wait** bloqueia a thread atual até ser acordada, enquanto **condvar_signal** acorda uma thread que está esperando. A função **condvar_broadcast** acorda todas as threads esperando. A função **thread_func** simula a entrada e saída de uma seção crítica, e a função principal cria três threads que executam essa função. De forma mais detalhada, temos:

Struct condvar:

- Esta estrutura representa uma variável de condição e contém dois semáforos (semaforo e wait_sem) e uma variável inteira (waiting_threads).
- semaforo controla o acesso à seção crítica, inicializado como 1 (acordado).
- wait_sem é usado para sincronizar as threads, inicializado como 0 (dormindo).
- waiting_threads mantém o número de threads atualmente esperando.

Função condvar_init:

- Inicializa a variável de condição.
- Inicializa semaforo como 1, indicando acesso permitido à seção crítica.
- Inicializa wait_sem como 0, indicando que nenhuma thread está esperando inicialmente.
- waiting_threads é inicializado como 0, pois inicialmente, nenhuma thread está esperando.

Função condvar_wait:

- Bloqueia a thread atual até ser acordada por outra thread.
- Adquire o semaforo, bloqueando outras threads de entrar na seção crítica (torna-se "dormindo").
- Incrementa waiting_threads, indicando que a thread está esperando.
- Libera o semaforo para outras threads.
- Aguarda até ser acordado por outra thread (torna-se "dormindo").

Função condvar_signal:

- Acorda uma thread que está esperando.

- Verifica se há threads esperando (`waiting_threads > 0`).
- Se verdadeiro, sinaliza (`sem_post`) para acordar uma thread.
- Decrementa `waiting_threads` indicando que uma thread foi acordada.

Função `condvar_broadcast`:

- Acorda todas as threads que estão esperando.
- Enquanto há threads esperando (`waiting_threads > 0`), sinaliza (`sem_post`) para acordar todas as threads.
- Decrementa `waiting_threads` para cada thread acordada.

Função `thread_func`:

- Função que simula a entrada e saída de uma seção crítica.
- A thread entra na seção crítica, realiza algum trabalho e sai.
- Após sair, chama `condvar_signal` para acordar outra thread que possa estar esperando.

Função `main`:

- Inicializa uma variável de condição (`cv`) e cria três threads usando a função `pthread_create`.
- As threads executam a função `thread_func`.
- A função principal espera até que todas as threads tenham terminado usando `pthread_join`.

Como a questão escolhida é relacionada com a questão 01 que é sobre mutex, resolvi propor vantagens e desvantagens em relação ao uso do próprio semáforo e mutex.

Variáveis de Condição com Semáforos:

Prós:

Flexibilidade: O uso de semáforos oferece maior flexibilidade, permitindo a implementação de estratégias mais complexas de sincronização além do que é possível com mutexes sozinhos.

Notificação Seletiva: As variáveis de condição permitem notificar seletivamente threads específicas, o que pode ser útil em cenários onde diferentes threads precisam ser acordadas em diferentes situações.

Possibilidade de Broadcast: A capacidade de acordar todas as threads esperando (broadcast) pode ser útil em certas situações para otimizar a sincronização.

Contras:

Complexidade: A implementação com semáforos pode ser mais complexa do que o uso direto de mutexes, especialmente em casos simples de exclusão mútua.

Mutexes:

Prós:

Simplicidade: O uso de mutexes por si só é mais simples e direto para garantir a exclusão mútua, sendo mais fácil de entender e menos propenso a erros.

Menos Overhead: Em alguns casos, a utilização de mutexes pode ter menos overhead do que a implementação de variáveis de condição com semáforos.

Contras:

Limitações na Comunicação: Mutexes, por si só, não oferecem uma forma eficiente de comunicação entre threads. Se a comunicação específica é necessária, pode ser necessário complementar com outras estruturas.

Menos Flexibilidade: Mutexes podem ser menos flexíveis em situações que exigem mais do que simples exclusão mútua, como notificações seletivas ou broadcast.

Otimização: como a otimização depende do contexto/objetivo do que se quer aprimorar, listei possíveis formas de otimizar:

Uso de Spinlocks em Vez de Semáforos:

- Em alguns casos, o uso de spinlocks pode ser mais eficiente quando a espera é curta. Isso evita a mudança de contexto para o sistema operacional, resultando em menos overhead.

Minimizar Mudanças de Contexto:

- Reduzir as mudanças de contexto adquirindo o semáforo diretamente em vez de usar `sem_post` seguido imediatamente por `sem_wait`. Isso pode ser feito mantendo o semáforo adquirido enquanto aguarda na `wait_sem` e liberando-o apenas após o `wait_sem` ser adquirido.

Utilização de `sem_trywait`:

- Em situações em que não é desejado bloquear indefinidamente, `sem_trywait` pode ser uma alternativa para tentar adquirir o semáforo sem esperar.

Uso de `sem_post` Condicional:

- Na função `condvar_broadcast`, em vez de iterar até que todas as threads sejam acordadas, considerar usar um valor especial ou uma flag para indicar às threads que elas devem sair da espera. Isso pode economizar iterações desnecessárias.

Aprimoramento da Lógica de Espera:

- Avaliar a lógica de espera nas funções `condvar_wait`, `condvar_signal`, e `condvar_broadcast`. Dependendo do contexto, a lógica pode ser otimizada para reduzir o tempo que as threads passam esperando.

Ajuste do Número Inicial de Threads Esperando:

- Ajustar o número inicial de threads esperando para minimizar o tempo gasto na espera da semaforização.

Avaliação da Necessidade de Dormir na Seção Crítica:

- Se a simulação de trabalho antes da seção crítica não é necessária, considerar remover o `sleep(2)` para melhorar a eficiência. (isso foi feito)

Dificuldades encontradas: não ficou claro nas questões se era para criar ou não a API do semáforo/mutex onde tinha “Considere um ambiente onde a seguinte API para semáforos está disponível”, porque mostrava partes de código dando a entender que era para criar do zero a API que supostamente deveria estar disponível, além disso, acabei chegando atrasado nas aulas sobre semáforos enquanto vinha do interior do estado e acabei optando por resolver essa questão porque eu iria entender o que perdi, por isso, houve um pouco de dificuldade no início e tive que pesquisar bastante, principalmente essa parte de otimização.

Nova defesa: na primeira defesa, o professor recomendou uma revisão na função **condvar_wait** devido a possibilidade de “corrida de dados” na seção crítica, pelo que entendi poderia ocorrer um caso em que a variável **waiting_threads** fosse incrementada ou decrementada sem um certo controle podendo até ficar negativa, para evitar isso seria necessário usar um mutex seguindo o formato do pdf disponibilizado pelo professor **void condvar_wait(struct condvar *c, pthread_mutex_t *m)**. Para resolver o problema acrescentei **pthread_mutex_lock(m)** para adquirir o mutex e garantir o acesso a seção crítica e ao afinal **pthread_mutex_unlock(m)** para liberar o mutex, além disso nas funções **condvar_signal** e **condvar_broadcast** acrescentei **sem_wait(&c->semaforo)** e **sem_post(&c->semaforo)** para não ocorrer possíveis problemas semelhantes ao caso de **waiting_threads**.

Alteração no código:

```
// Bloqueia a thread atual até que seja acordada por outra thread
void condvar_wait(struct condvar *c, pthread_mutex_t *m) {
```

```
    sem_wait(&c->semaforo);
    c->waiting_threads++;
    sem_post(&c->semaforo);
    pthread_mutex_unlock(m);
    sem_wait(&c->wait_sem);
    pthread_mutex_lock(m);
```

```
}
```

```
// Acorda uma thread que está esperando
void condvar_signal(struct condvar *c) {
```

```
    sem_wait(&c->semaforo);
    if (c->waiting_threads > 0) {
        sem_post(&c->wait_sem);
        c->waiting_threads--;
    }
    sem_post(&c->semaforo);
```

```
}
```

```
// Acorda todas as threads que estão esperando
```

```
void condvar_broadcast(struct condvar *c) {
```

```
    sem_wait(&c->semaforo);
    while (c->waiting_threads > 0) {
        sem_post(&c->wait_sem);
        c->waiting_threads--;
    }
    sem_post(&c->semaforo);
```

```
}
```

