RESOLUÇÃO DO CONCURSO 4

SISTEMAS OPERATIVOS 2024/2025

Feito por:
a83933 Rodrigo Linhas
2º Ano da Licenciatura de Engenharia Informática (LEI)
Regente da UC: Amine Berqia

Índice

INTRODUÇÃO

Este trabalho tem o intuito de demonstrar a minha resolução perante os exercícios propostos no Concurso 4, juntamente com a(s) referencia(s) bibliográfica(s) consultada(s) no final do documento.

As soluções foram desenvolvidas em linguagem C, testadas em ambiente Linux e validadas com capturas de ecrã para garantir conformidade com os requisitos. Os códigos usam semáforos POSIX e mutexes para sincronização de processos e threads, garantindo exclusão mútua e evitando condições de corrida, deadlock e starvation, que seram explicados com mais detalhe.

A estrutura do documento segue a numeração dos exercícios, com explicações concisas, exemplos de código e saídas geradas.

Palavras chave: semaforos, deadlock, starvation, mutex

EXERCÍCIO I: PRODUTOR-CONSUMIDOR COM SEMÁFOROS

a) Explicação e Constrangimentos

Após ter compilado o programa dado pelo enunciado obtivemos o seguinte output:

```
rodrigo@rodrigo-Aspire-XC600:~/Desktop/uni/SO/C4$ gcc ex1.c -o ex1.exe
rodrigo@rodrigo-Aspire-XC600:~/Desktop/uni/SO/C4$ ./ex1.exe
Proc. C.
Object from buffer.
Cons. objet.0
Proc P.
Production .0
Object to buffer.
Production .1
Object to buffer.
Object from buffer.
Cons. objet.-1
Production .2
Object to buffer.
Production .3
Object to buffer.
Object from buffer.
Cons. objet.-2
Production .4
Object to buffer.
Production .5
Object to buffer.
^C
rodrigo@rodrigo-Aspire-XC600:~/Desktop/uni/SO/C4$
```

tive que parar a execução do mesmo uma vez que contem ciclos infinitos, mas dá para ter uma ideia da execução do programa. Também tenho que salientar que **deixei o programa a executar por aproximadamente 1 minuto (60 segundos).** Esta métrica será necessária para fins comparativos**.**

O problema central é a falta de partilha da variável `i` e a ausência de mecanismos de sincronização. Sem isso, o *buffer* não é devidamente administrado, permitindo condições inválidas (ex.: consumidor remover do buffer vazio).

O primeiro instinto para resolver este tipo de problema seria sem sombra de duvida o uso de *mutex_lock()* e *mutex_unlock()*, conteúdo visto no concurso anterior. **Contudo temos estes requisitos:**

- ***O produtor não pode colocar um objeto no buffer enquanto estiver cheio (O tamanho do buffer N é finito (N=10));***
- ***O consumidor não pode remover um objeto do buffer enquanto estiver vazio;***
- ***O produtor e o consumidor não devem utilizar o buffer ao mesmo tempo;***

Para solucionar o 1.º requisito basta criar uma variável global e implementa-la, **mas para o resto precisamos da adição de um novo recurso: os semáforos.**
Os semáforos são estruturas de sincronização que permitem controlar o acesso a recursos compartilhados entre processos ou threads. Estritamente necessários para solucionar estes requisitos.

Assim poderíamos criar 2 semáforos: *sem_empty* e *sem_full.*

- **`sem_empty`: Representa o número de espaços livres no buffer (inicializado com N=10).**
- **`sem_full`: Representa o número de itens disponíveis para consumo (inicializado com 0).**

Garatindo que o produtor só insira itens se houver espaço *sem_wait(&sem_empty)* e o consumidor só remova se houver itens *sem_wait(&sem_full)*.

Óbvio que precisamos do *mutex* para não haver a *race condition* que muito foi batalhada no concurso anterior

Com tudo que revimos anteriormente, temos a seguinte estratégia para a continuação do exercício:

- Buffer circular com tamanho N = 10;
- Semáforo vazio (sem_empty) inicializado a N representa espaços livres;
- Semáforo full (sem_full) inicializado a 0 representa itens disponíveis;
- Mutex (sem_mutex) inicializado a 1 garante que produtor e consumidor não acedam ao buffer simultaneamente;
- Produtor: antes de inserir, faz sem_wait(&sem_empty); sem_wait(&sem_mutex); insere no buffer; sem_post(&sem_mutex); sem_post(&sem_full);
- Consumidor: antes de remover, faz sem_wait(&sem_full); sem_wait(&sem_mutex); remove do buffer; sem_post(&sem_mutex); sem_post(&sem_empty).

Que deixará o código muito mais eficiente e garantindo também todos os requisitos sejam cumpridos.

b) Código em C

Aplicando a estratégia que definimos temos a seguinte implementação:

```c
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <pthread.h>
#include <semaphore.h>

#define N 10

sem_t sem_empty, sem_full, sem_mutex;
int buffer[N];
int in = 0, out = 0;

void *producer() {
    int i = 0;
    while (1) {
        sem_wait(&sem_empty);
        sem_wait(&sem_mutex);
        buffer[in] = i;
        printf("Produção: %d no buffer[%d]\n", i, in);
        in = (in + 1) % N;
        i++;
        sem_post(&sem_mutex);
        sem_post(&sem_full);
        sleep(1);
    }
}

void *consumer() {
    int item;
    while (1) {
        sem_wait(&sem_full);
        sem_wait(&sem_mutex);
        item = buffer[out];
        printf("Consumo: %d do buffer[%d]\n", item, out);
        out = (out + 1) % N;
        sem_post(&sem_mutex);
        sem_post(&sem_empty);
        sleep(2);
    }
    return NULL;
}

int main(void) {
    pthread_t prod, cons;
    sem_init(&sem_empty, 0, N);
    sem_init(&sem_full, 0, 0);
    sem_init(&sem_mutex, 0, 1);
    pthread_create(&prod, NULL, producer, NULL);
    pthread_create(&cons, NULL, consumer, NULL);
    pthread_join(prod, NULL);
    pthread_join(cons, NULL);
    sem_destroy(&sem_empty);
    sem_destroy(&sem_full);
    sem_destroy(&sem_mutex);
    return 0;
}
```

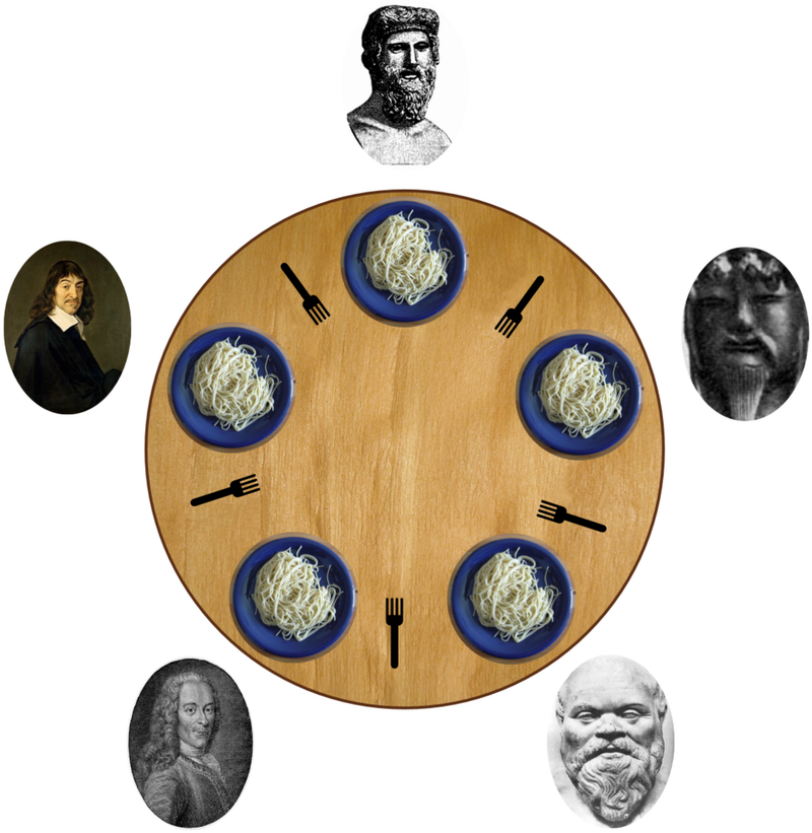
Obtendo o seguinte output com o tempo de execução de ≈ 60 segundos:

```
rodrigo@rodrigo-Aspire-XC600:~/Desktop/uni/SO/C4$ gcc ex1-resolvido.c -o ex1-resolvido.exe
rodrigo@rodrigo-Aspire-XC600:~/Desktop/uni/SO/C4$ ./ex1-resolvido.exe
Produção: 0 no buffer[0]
Consumo: 0 do buffer[0]
Produção: 1 no buffer[1]
Consumo: 1 do buffer[1]
Produção: 2 no buffer[2]
Produção: 3 no buffer[3]
Consumo: 2 do buffer[2]
Produção: 4 no buffer[4]
Produção: 5 no buffer[5]
Produção: 6 no buffer[6]
Consumo: 3 do buffer[3]
Produção: 7 no buffer[7]
Produção: 8 no buffer[8]
Consumo: 4 do buffer[4]
Produção: 9 no buffer[9]
Produção: 10 no buffer[0]
Consumo: 5 do buffer[5]
Produção: 11 no buffer[1]
Produção: 12 no buffer[2]
Consumo: 6 do buffer[6]
Produção: 13 no buffer[3]
Consumo: 7 do buffer[7]
Produção: 14 no buffer[4]
Produção: 15 no buffer[5]
Consumo: 8 do buffer[8]
Produção: 16 no buffer[6]
Produção: 17 no buffer[7]
Consumo: 9 do buffer[9]
Produção: 18 no buffer[8]
Produção: 19 no buffer[9]
Consumo: 10 do buffer[0]
Produção: 20 no buffer[0]
Consumo: 11 do buffer[1]
Produção: 21 no buffer[1]
Consumo: 12 do buffer[2]
Produção: 22 no buffer[2]
Consumo: 13 do buffer[3]
Produção: 23 no buffer[3]
Consumo: 14 do buffer[4]
Produção: 24 no buffer[4]
Consumo: 15 do buffer[5]
Produção: 25 no buffer[5]
Consumo: 16 do buffer[6]
Produção: 26 no buffer[6]
Consumo: 17 do buffer[7]
Produção: 27 no buffer[7]
Consumo: 18 do buffer[8]
Produção: 28 no buffer[8]
Consumo: 19 do buffer[9]
```

EXERCÍCIO II: JANTAR DOS FILÓSOFOS

1) Descrição do Problema

Vários filósofos sentam-se ao redor de uma mesa circular, para maior simplicidade vamos considerar apenas 5, cada um alternando entre pensar e comer. Entre cada par de filósofos há um garfo. Para comer, um filósofo precisa de dois garfos (o da sua esquerda e o da sua direita logicamente).
O problema é um bocado parecido com o que fizemos anteriormente no entanto realça o potencial de *deadlock* (todos seguram o garfo à esquerda e esperam pelo da direita) e *starvation* (alguns podem nunca obter dois garfos e assim não comem). Eis uma visualização gráfica do problema:



Situações como estas são perfeitamente normais em contexto de sistemas operativos, como na partilha de recursos para diferentes processos.

Como solucionar?

Para resolver o problema do Jantar dos Filósofos, evitando deadlock e starvation, utilizaremos a seguinte estratégia:

1. Controle Centralizado com mutex:

- Um *mutex* garante exclusão mútua durante a verificação e atualização dos estados dos filósofos, podemos pensar até que é um empregado a controlar o ponto de situação. Isso evita que múltiplos filósofos tentem utilizar garfos simultaneamente, prevenindo inconsistências.

2. Semáforos Individuais para Cada Filósofo:

- Cada filósofo possui um semáforo inicializado em 0. Ele é usado para bloquear o filósofo quando os garfos não estão disponíveis.

3. Função de teste:

Agora vamos aprimorar o nosso empregado com as seguintes propriedades:

- Verifica se o filósofo pode utilizar ambos os garfos (esquerdo e direito).
- Condições para comer:
  - O filósofo está com fome, poderíamos definir uma constante para tal.
  - Os filósofos adjacentes não estão comendo.
  - Se as condições forem atendidas, o filósofo começa a comer, poderíamos definir uma constante para tal, e seu semáforo é libertado.

4. Prevenção de Deadlock:

- Nenhum filósofo utiliza um garfo sem verificar se o outro está disponível. Isso evita o cenário onde todos seguram um garfo e esperam infinitamente.
- O uso do *mutex* garante atomicidade na verificação dos estados.

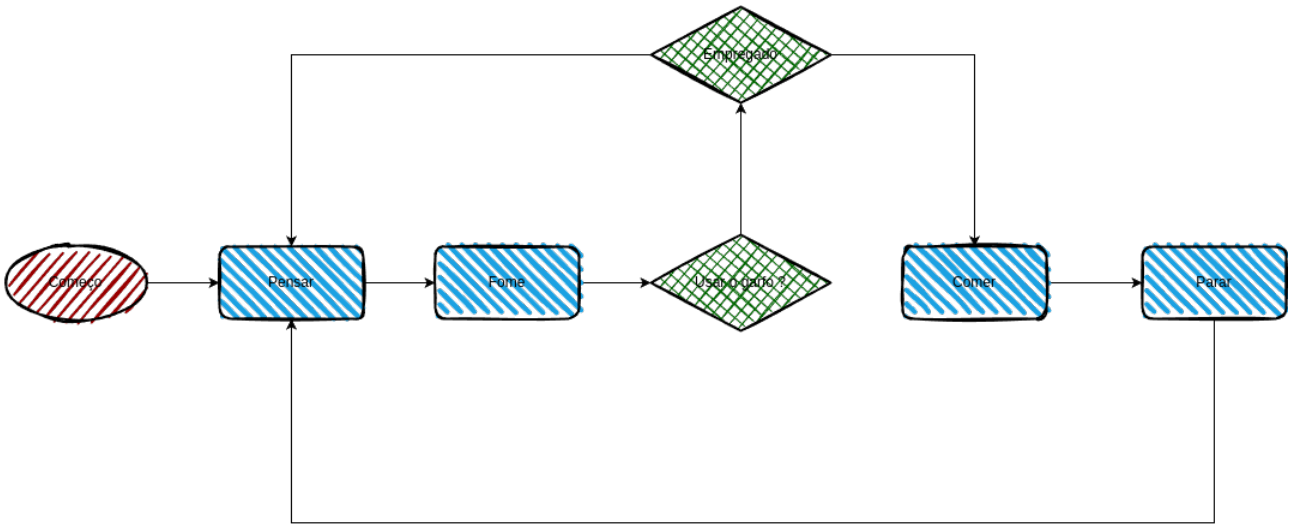
5. Prevenção de Starvation:

- Quando um filósofo termina de comer, ele verifica se os vizinhos podem comer. Isso prioriza filósofos que estão há mais tempo esperando.

6. Fluxo do Programa:

Pondo assim num fluxo de funcionamento do programa:

Pensar → Ficar com fome → Tentar utilizar garfos → Comer → Parar de usar os garfos.



2) Solução em C (prevenção de deadlock e starvation)

A partir da estratégia defina anteriormente temos a seguinte implementação:

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

#define N 5
#define THINKING 0
#define HUNGRY 1
#define EATING 2

int state[N];
sem_t mutex;
sem_t self[N];

void test(int i) {
    if (state[i] == HUNGRY && state[(i+4)%N] != EATING
            && state[(i+1)%N] != EATING) {
        state[i] = EATING;
        sem_post(&self[i]);
    }
}

void take_forks(int i) {
    sem_wait(&mutex);
    state[i] = HUNGRY;
    test(i);
    sem_post(&mutex);
    sem_wait(&self[i]);
}

void put_forks(int i) {
    sem_wait(&mutex);
    state[i] = THINKING;
    test((i+4)%N);
    test((i+1)%N);
    sem_post(&mutex);
}
```

```c
void *philosopher(void *num) {
    int i = *(int *)num;
    while (1) {
        printf("Filósofo nº %d pensando\n", i+1);
        sleep(1);
        printf("Filósofo nº %d tem fome\n", i+1);
        take_forks(i);
        printf("Filósofo nº %d comendo\n", i+1);
        sleep(1);
        put_forks(i);
    }
}

int main(void) {
    int i;
    pthread_t thread_id[N];
    sem_init(&mutex, 0, 1);
    for (i = 0; i < N; i++) {
        sem_init(&self[i], 0, 0);
    }
    int phil[N] = {0,1,2,3,4};
    for (i = 0; i < N; i++) {
        pthread_create(&thread_id[i], NULL, philosopher, &phil[i]);
    }
    for (i = 0; i < N; i++) {
        pthread_join(thread_id[i], NULL);
    }
    return 0;
}
```

Obtendo o seguinte output, por razões óbvias tive de parar a execução uma vez que contem um ciclo infinito contudo já terá informação suficiente que demonstra que o programa está a correr conforme (VER A PRÓXIMA PÁGINA):

```
rodrigo@rodrigo-Aspire-XC600:~/Desktop/uni/SO/C4$ gcc ex2.c -o ex2.exe
rodrigo@rodrigo-Aspire-XC600:~/Desktop/uni/SO/C4$ ./ex2.exe
Filósofo n° 1 pensando
Filósofo n° 5 pensando
Filósofo n° 2 pensando
Filósofo n° 3 pensando
Filósofo n° 4 pensando
Filósofo n° 1 tem fome
Filósofo n° 1 comendo
Filósofo n° 5 tem fome
Filósofo n° 4 tem fome
Filósofo n° 4 comendo
Filósofo n° 2 tem fome
Filósofo n° 3 tem fome
Filósofo n° 1 pensando
Filósofo n° 2 comendo
Filósofo n° 4 pensando
Filósofo n° 5 comendo
Filósofo n° 1 tem fome
Filósofo n° 4 tem fome
Filósofo n° 5 pensando
Filósofo n° 2 pensando
Filósofo n° 3 comendo
Filósofo n° 1 comendo
Filósofo n° 5 tem fome
Filósofo n° 2 tem fome
Filósofo n° 4 comendo
Filósofo n° 3 pensando
Filósofo n° 1 pensando
Filósofo n° 2 comendo
Filósofo n° 4 pensando
Filósofo n° 3 tem fome
Filósofo n° 5 comendo
Filósofo n° 1 tem fome
Filósofo n° 2 pensando
Filósofo n° 3 comendo
Filósofo n° 4 tem fome
Filósofo n° 5 pensando
Filósofo n° 1 comendo
Filósofo n° 2 tem fome
Filósofo n° 3 pensando
Filósofo n° 4 comendo
Filósofo n° 5 tem fome
Filósofo n° 3 tem fome
Filósofo n° 4 pensando
Filósofo n° 1 pensando
Filósofo n° 5 comendo
Filósofo n° 3 comendo
Filósofo n° 4 tem fome
Filósofo n° 1 tem fome
Filósofo n° 3 pensando
Filósofo n° 5 pensando
Filósofo n° 4 comendo
Filósofo n° 2 comendo
Filósofo n° 3 tem fome
```

REFERÊNCIAS BIBLIOGRÁFICAS

- Silberschatz, A., Galvin, P. B., & Gagne, G. (2018). Operating System Concepts (10th ed.). Wiley.
- Tanenbaum, A. S., & Bos, H. (2015). Modern Operating Systems (4th ed.). Pearson.
- Man page sem_init, sem_wait, sem_post, sem_destroy.
- https://youtu.be/VSkvwzqo-Pk?si=Iz0rH6dFtnYhUDSp
- https://en.wikipedia.org/wiki/Dining_philosophers_problem
- https://en.wikipedia.org/wiki/Producer%E2%80%93consumer_problem