Revisitando o Problema do Jantar dos Canibais: Uma Nova Proposta para Análise da Sincronização a partir de Mutexes e Semáforos

André Neves, Caio Eduardo, Douglas Brito, Gabriel Thiago, Rafael Passos, Tiago Costa e Vinicius Bueno
Universidade Federal de Alfenas

***Resumo*—Este trabalho apresenta uma solução imcompleta de uma nova proposta para análise e otimização da sincronização no problema do jantar dos canibais, utilizando visualização e os mecanismos de *mutexes* e *semáforos* através dos períodos de ondas sinusoidais. O objetivo é identificar falhas de sincronização, verificar a exclusão mútua adequada evitando a ocorrência de *deadlocks*, avaliar o uso do semáforo e analisar o comportamento de N canibais ao longo do tempo. A abordagem proposta permitiria uma compreensão intuitiva do sistema de sincronização, facilitando a identificação de problemas de concorrência e violações das regras do problema. Simulações foram implementadas, variando o número de canibais e registrando métricas como o uso do *mutex*, uso do *semáforo* e tempo de espera dos canibais. Esse estudo pode contribuir para a área de sistemas distribuídos, auxiliando no desenvolvimento de soluções mais robustas e eficientes para problemas de sincronização em cenários reais.**

I. Introdução

A sincronização entre processos é um desafio em sistemas operacionais, especialmente quando há a necessidade de compartilhar recursos entre threads concorrentes. No caso do problema dos canibais e do cozinheiro, temos uma tribo de canibais que janta ao redor de um caldeirão, onde cada canibal precisa agir de forma coordenada para evitar conflitos e garantir que todos tenham acesso às porções de missionário cozido. O problema do jantar dos canibais é um problema de concorrência em que um grupo de canibais compartilha um caldeirão com um número limitado de porções de comida. Os canibais podem se servir do caldeirão para comer uma porção de cada vez. No entanto, se o caldeirão estiver vazio, eles devem esperar até que o cozinheiro encha o caldeirão novamente.

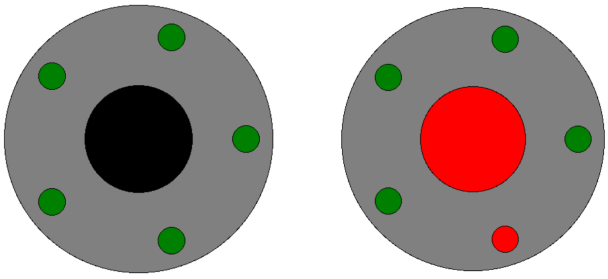

Figura 1. Ilustração do Jantar dos Canibais: Os cinco círculos verdes representam os canibais e o círculo preto ao centro representa o caldeirão vazio. Foi implementada uma simulação que periodicamente troca a cor do caldeirão para vermelho quando está cheio e enquanto um dos canibais está comendo.

Os conceitos fundamentais para a solução desse problema são os *semáforos*, que são estruturas de dados que permitem a coordenação entre *threads*, e os *mutexes*, que são mecanismos de exclusão mútua utilizados para evitar condições de corrida. Além disso, é importante compreender o que é deadlock, uma situação em que os processos ficam bloqueados permanentemente devido a uma interdependência circular de recursos.

O problema da sincronização surge na computação concorrente quando vários processos ou threads precisam de acessar recursos compartilhados ou coordenar sua execução de forma mutuamente exclusiva. Envolve a gestão do acesso a seções críticas do código para evitar condições de corrida, garantir a consistência dos dados e manter a integridade do sistema.

Os semáforos são uma primitiva de sincronização normalmente utilizada para controlar o acesso a recursos compartilhados. Funcionam como contadores que podem ser incrementados ou decrementados atomicamente. Dois tipos de semáforos são amplamente utilizados: os semáforos binários e os semáforos de contagem. Os semáforos binários têm dois valores possíveis (0 ou 1) e são frequentemente utilizados para exclusão mútua, permitindo que apenas um processo acesse um recurso de cada vez. Os semáforos de contagem podem ter valores superiores a 1 e são utilizados para cenários em que vários processos podem acessar um recurso simultaneamente, até um determinado limite.

Uma *deadlock* ocorre quando dois ou mais processos não conseguem prosseguir porque cada um deles está esperando por um recurso detido por outro processo, resultando em um padrão de espera circular. O deadlock normalmente acontece quando quatro condições necessárias são atendidas: exclusão mútua (os recursos não podem ser usados simultaneamente por vários processos), manter e esperar (os processos mantêm os recursos enquanto esperam por outros), sem preempção (os recursos não podem ser retirados à força dos processos) e espera circular (uma cadeia circular de processos, cada um esperando por um recurso mantido pelo próximo).

Um *mutex* (abreviatura de exclusão mútua) é um mecanismo de sincronização que garante que apenas um thread ou processo possa acessar um recurso compartilhado de cada vez. Ele fornece acesso exclusivo ao recurso protegido, impedindo modificações simultâneas que poderiam levar a inconsistências ou condições de corrida. Um mutex pode ser bloqueado por um thread, permitindo que ele acesse o recurso, e desbloqueado quando o thread termina, permitindo que outros threads adquiram o mutex e acessem o recurso.

II. Possíveis Pontos de Deadlock

Na solução proposta para o problema dos canibais e do cozinheiro, existem alguns pontos em que pode ocorrer deadlock. Um exemplo é quando dois ou mais canibais tentam se servir ao mesmo tempo, mas o caldeirão está vazio. Nesse caso, os canibais ficam bloqueados aguardando a chegada de novas porções de missionário, enquanto o cozinheiro aguarda que algum canibal consuma uma porção para que ele possa enchê-lo novamente.

Outro ponto de possível deadlock ocorre quando o cozinheiro está enchendo o caldeirão e é interrompido por algum motivo externo, como uma interrupção de tempo ou outra atividade. Nesse caso, os canibais ficam bloqueados aguardando a conclusão do processo de enchimento do caldeirão, enquanto o cozinheiro aguarda que algum canibal consuma uma porção.

No problema do jantar dos canibais, o impasse pode ocorrer em vários pontos. O problema envolve canibais e missionários que precisam atravessar um rio usando um barco que só pode transportar um número limitado de passageiros. O impasse pode ocorrer quando todos os canibais e missionários estão de um lado do rio e o barco está do outro lado. Se não houver passageiros suficientes para encher o barco com uma combinação de canibais e missionários quegaranta a segurança dos missionários, surge uma situação de impasse. Nenhum dos canibais ou missionários pode atravessar o rio sem violar a restrição de que deve haver sempre mais missionários do que canibais de cada lado. Esse impasse ocorre devido à incapacidade de formar uma combinação segura de passageiros para o barco.

III. Algoritmo

O código apresentado no *Anexo A* é uma solução para o problema clássico do jantar dos canibais usando threads e semáforos em C.

O código em questão implementa uma solução para o problema clássico do jantar dos canibais. O problema envolve um grupo de canibais que compartilham um caldeirão com um número limitado de porções de comida. Os canibais podem se servir do caldeirão e comer uma porção, mas se o caldeirão estiver vazio, eles devem esperar até que seja reabastecido.

A implementação utiliza *threads* para representar os canibais e o cozinheiro. Cada canibal é executado em um loop infinito e realiza as seguintes ações:

1) Chama a função *servir* para se servir do caldeirão, se houver porções disponíveis.
2) Chama a função *comer* para simular o ato de comer por um período de tempo.
3) Repete o processo indefinidamente.

A função *servir* é responsável por controlar o acesso ao caldeirão compartilhado. Ela aguarda o semáforo *servir_sem* e, em seguida, adquire o mutex para garantir a exclusão mútua. Se o caldeirão contiver pelo menos uma porção, uma porção é removida e uma mensagem é exibida. Se o caldeirão estiver vazio, o semáforo *encher_caldeirao_sem* é liberado, indicando ao cozinheiro que ele pode reabastecer o caldeirão. Após o acesso ao caldeirão, o mutex é liberado.

A função *comer* simplesmente imprime uma mensagem indicando que o canibal está comendo e, em seguida, adquire o mutex para verificar se ainda há porções disponíveis no caldeirão. Se houver, ele libera o semáforo *servir_sem*. Em seguida, o mutex é liberado e ocorre uma pausa simulando o ato de comer.

O cozinheiro é representado por outra *thread*, executando o procedimento *cozinheiro*. Ele é responsável por reabastecer o caldeirão quando estiver vazio. O cozinheiro aguarda o semáforo *encher_caldeirao_sem*, adquire o mutex para reabastecer o caldeirão com um número fixo de porções e, em seguida, libera o semáforo *servir_sem* para permitir que os canibais se sirvam. Após isso, ocorre uma pausa antes de repetir o processo.

A função principal (*main*) inicializa os semáforos, cria as *threads* para os canibais e o cozinheiro, aguarda a conclusão das *threads* e, finalmente, destrói os *semáforos* e o *mutex*.

- **INÍCIO**
- **Definição das constantes:**
  - N: número máximo de porções no caldeirão
  - *NUM_CANIBAIS*: número de canibais
- **Declaração da variável compartilhada:**
  -
  - caldeirao: representa o estado do caldeirão
- **Declaração do mutex:**
  - *mutex*: mutex para garantir exclusão mútua
- **Declaração dos semáforos:**
  - *encher_caldeirao_sem*: semáforo para controlar o enchimento do caldeirão
  - *servir_sem*: semáforo para controlar o ato de se servir do caldeirão
- **Procedimento servir(canibal):**
  - Esperar pelo semáforo *servir_sem*
  - Bloquear o mutex
- Se o caldeirão contiver pelo menos uma porção então
  - Reduzir uma porção do caldeirão
  - Esperar 1 segundo
  - Imprimir mensagem informando que o canibal se serviu e quantas porções restam
- Se o caldeirão estiver vazio então
  - Liberar o semáforo *encher_caldeirao_sem*
- Desbloquear o mutex
- **Procedimento comer(canibal):**
  - Imprimir mensagem informando que o canibal está comendo
  - Bloquear o mutex
  - Se o caldeirão contiver pelo menos uma porção então
- Liberar o semáforo *servir_sem*
  - Desbloquear o mutex
  - Esperar 1 segundo
- **Procedimento canibal(arg):**
  - Enquanto verdadeiro faça
    - Chamar o procedimento servir com o argumento passado

    - ∗ Chamar o procedimento comer com o argumento passado
- **Procedimento encher():**
  - Esperar pelo semáforo *encher_caldeirao_sem*
  - Bloquear o mutex
  - Preencher o caldeirão com *N* porções
  - Imprimir mensagem informando que o cozinheiro encheu o caldeirão e quantas porções foram adicionadas
  - Liberar o semáforo *servir_sem*
  - Desbloquear o mutex
- **Procedimento dormir():**
  - Esperar 1 segundo
- **Procedimento cozinheiro(arg):**
  - Enquanto verdadeiro faça
    - ∗ Chamar o procedimento encher
    - ∗ Chamar o procedimento dormir
- **Função principal:**
  - Declaração das variáveis canibais[*NUM_CANIBAIS*] e *cozinheiro_thread* do tipo *pthread_t*
  - Inicializar os semáforos *encher_caldeirao_sem* e *servir_sem*
  - Para cada *i* de *0* até $NUM\_CANIBAIS - 1$ faça
    - ∗ Criar uma nova thread para o procedimento canibal, passando i como argumento
  - Criar uma nova thread para o procedimento cozinheiro
  - Para cada *i* de *0* até $NUM\_CANIBAIS - 1$ faça
    - ∗ Aguardar a conclusão da thread do canibal *i*
    - ∗ Aguardar a conclusão da thread do cozinheiro
    - ∗ Destruir os semáforos *encher_caldeirao_sem* e *servir_sem*
    - ∗ Destruir o mutex
  - *Retornar 0*
  - **FIM**

Essa implementação garante a exclusão mútua entre os canibais para evitar conflitos no acesso ao caldeirão e garante que o caldeirão seja reabastecido quando estiver vazio.

IV. Resultados

A mesma solução foi implementada em Python[1] com o propósito de plotar alguns gráficos importantes para a visualização dos dados do problema do jantar dos canibais. A visualização através dos gráficos permite uma compreensão mais clara e intuitiva do comportamento dos canibais e do funcionamento do sistema de sincronização.

Ao representar o uso do mutex e semáforo ao longo do tempo, os gráficos facilitam a identificação visual de possíveis problemas de sincronização, exclusão mútua inadequada ou violações das regras do problema. Dessa forma, torna-se mais fácil verificar se a sincronização está ocorrendo de forma correta, se a exclusão mútua está sendo respeitada e se o uso do semáforo está sendo adequado.

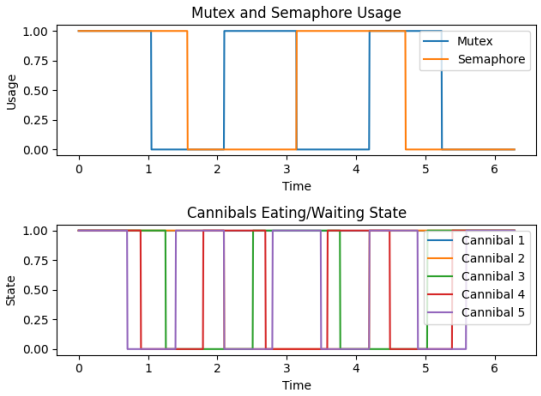

Figura 2. 1 ciclo: O gráfico do ciclo 1 mostra o comportamento dos 3 canibais ao longo do tempo, revelando os momentos em que eles estão adquirindo comida, bem como os períodos em que estão esperando sua vez de se alimentar. Através dessa análise, é possível identificar possíveis gargalos no sistema de sincronização e buscar formas de otimizá-lo.

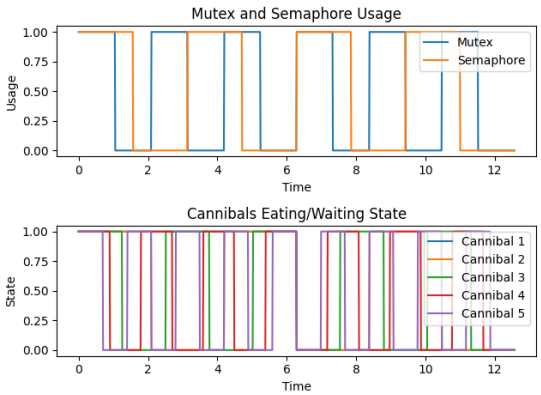

Figura 3. 2 ciclos: No gráfico do ciclo 2, é possível observar como o padrão de comportamento dos canibais se repete ao longo do tempo. Isso indica a existência de uma dinâmica cíclica no sistema, onde os canibais seguem um ritmo previsível na obtenção e consumo de comida. Essa repetição de padrões pode ser explorada para melhorar a eficiência da sincronização entre os canibais.

[1] Anexo B

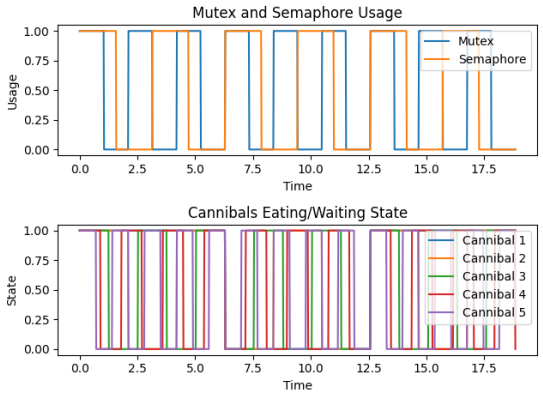

Figura 4. 3 ciclos: Ao analisar o gráfico do ciclo 3, é possível identificar possíveis variações no comportamento dos canibais. Essas variações podem ocorrer devido a fatores externos ou atrasos na comunicação entre os canibais. A compreensão dessas variações é fundamental para aprimorar o sistema de sincronização, minimizando os atrasos e garantindo uma distribuição justa e eficiente da comida.

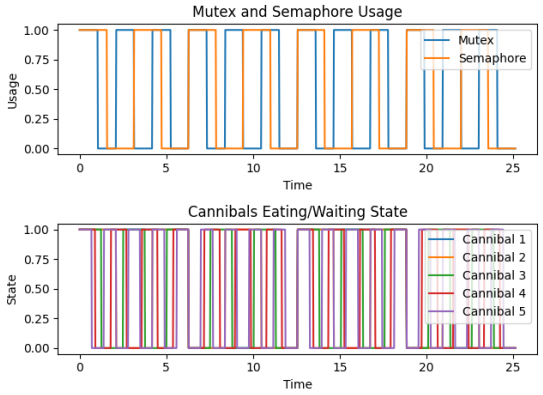

Figura 5. 4 ciclos: No gráfico do ciclo 4, pode-se observar a estabilidade e consistência no comportamento dos canibais ao longo do tempo. Isso indica que o sistema de sincronização foi otimizado e está funcionando de maneira eficiente, garantindo uma distribuição adequada da comida entre os canibais. A análise desses ciclos subsequentes permite verificar se as melhorias implementadas são efetivas e se o sistema continua operando de forma consistente.

Embora a diferença entre as implementações em Python e C seja principalmente de linguagem e abordagem, ambos os códigos realizam a mesma tarefa de simular o problema do jantar dos canibais e apresentam resultados semelhantes. No entanto, a escolha do Python, devido à sua facilidade de lidar com gráficos, proporciona uma vantagem adicional ao fornecer uma visualização clara e intuitiva dos dados, facilitando a compreensão e otimização do sistema de sincronização.

O código em C apresentado no anexo A foi modificado para extrapolar o problema do jantar dos canibais para um cenário com N canibais. Essa modificação permite realizar a contagem do uso de mutex e semáforos, bem como o tempo de espera dos n-1 canibais a cada iteração do problema. Além disso, o código varia o número de canibais de 1 a 10, incrementando de 1 em 1.

Essa modificação é valiosa, pois permite uma análise mais abrangente do desempenho do sistema de sincronização à medida que o número de canibais aumenta. Ao extrapolar o problema do jantar dos canibais para um cenário com N canibais, é possível obter dados estatísticos e científicos sobre o uso de mutex e semáforos, bem como o tempo de espera dos canibais.

As variáveis acumuladoras são utilizadas para contabilizar a quantidade de vezes que o mutex é utilizado e a quantidade de vezes que o semáforo é usado como controle de acesso às porções de comida. Essas contagens fornecem informações sobre a eficiência e a correta aplicação dos mecanismos de sincronização.

Além disso, o código também registra o tempo de espera dos $n-1$ canibais em cada iteração. Essa informação é relevante para compreender o impacto do número de canibais no tempo de espera geral do sistema. Ela pode ser utilizada para avaliar o desempenho do sistema sob diferentes cargas de trabalho e identificar possíveis gargalos ou oportunidades de otimização.

Ao variar o número de canibais de 1 a 10, de forma incremental, o código permite examinar a evolução do uso de mutex e semáforos, bem como o tempo de espera, à medida que a complexidade do problema aumenta. Essa análise sistemática possibilita uma compreensão mais completa do comportamento do sistema de sincronização em diferentes cenários, fornecendo insights valiosos para sua otimização e ajuste de acordo com as necessidades específicas.

Portanto, ao modificar o código em C para extrapolar o problema do jantar dos canibais para N canibais, contabilizar o uso de mutex e semáforos, e registrar o tempo de espera dos canibais, é possível realizar uma análise mais abrangente do sistema de sincronização, permitindo avaliar seu desempenho sob diferentes cargas de trabalho e obter informações relevantes para sua otimização.

Os dados gerados pelo código em C[2] foram salvos em um arquivo. Em seguida, o Python foi utilizado para realizar a análise desses dados e gerar os seguintes gráficos:

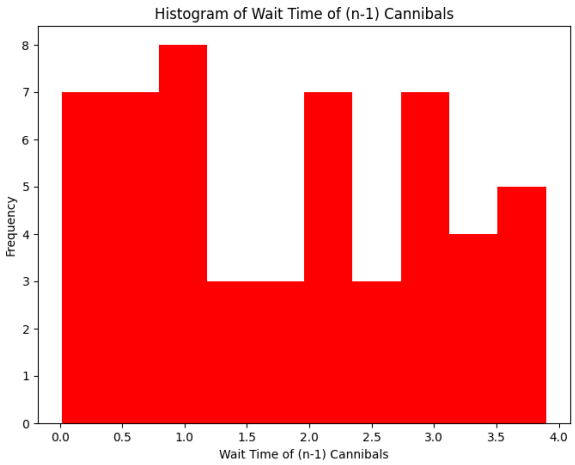

Figura 6. Gráfico do Número de Canibais versus Tempo de Espera dos N-1 Canibais: Nesse gráfico, é possível observar como o tempo de espera dos n-1 canibais varia em relação ao número de canibais. O tempo de espera médio é representado no eixo vertical, enquanto o número de canibais é representado no eixo horizontal. Esse gráfico oferece insights sobre o desempenho do sistema de sincronização e a eficiência do mecanismo utilizado.

[2] Anexo C

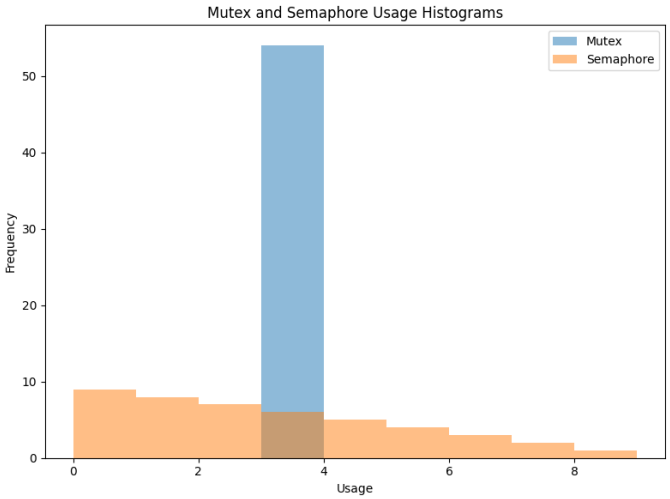

Figura 7. Histogramas de Uso de Mutex e Semáforos em relação ao número de canibais: Os histogramas são gráficos que mostram a distribuição do uso de mutex e semáforos para cada número de canibais. Cada barra no histograma representa a quantidade de vezes que o mutex ou semáforo foi utilizado para um determinado número de canibais. Esses histogramas oferecem uma visão mais detalhada da frequência de uso dos mecanismos de sincronização para diferentes cenários.

V. Conclusão

Neste trabalho, apresentamos o problema de sincronização dos canibais e do cozinheiro e discutimos os conceitos de semáforos, deadlock e mutex. Descrevemos uma possível solução de sincronização, que utiliza mutex e semáforos para garantir a correta coordenação entre os canibais e o cozinheiro. Através desse algoritmo, é possível evitar condições de corrida e deadlock, assegurando um jantar harmonioso para a tribo de canibais.

Referências

[1] Dijkstra, E. W. (1968). Cooperating sequential processes. In Programming languages (pp. 43-112). Academic Press.
[2] Silberschatz, A., Galvin, P. B., & Gagne, G. (2014). Operating system concepts. John Wiley & Sons.
[3] Tanenbaum, A. S., & Bos, H. (2006). Modern operating systems. Pearson Education.
[4] Bach, M. J. (1996). The design of the UNIX operating system. Prentice Hall.
[5] Tanenbaum, A. S. (2007). Distributed systems: principles and paradigms. Pearson Education.
[6] Silberschatz, A., Korth, H. F., & Sudarshan, S. (2013). Database system concepts. McGraw-Hill.

Glossário

Sincronização: Processo de coordenação entre threads ou processos para garantir a correta execução de suas ações e evitar problemas como condições de corrida e deadlock.

Condição de Corrida: Situação em que múltiplas threads ou processos tentam acessar e modificar um recurso compartilhado simultaneamente, levando a resultados inconsistentes e imprevisíveis.

Deadlock: Situação em que dois ou mais processos ficam bloqueados permanentemente, esperando uns pelos outros para liberar recursos que cada um deles precisa para prosseguir com sua execução.

Mutex: Abreviação de "exclusão mútua". Mecanismo de sincronização que permite que apenas um thread ou processo acesse um recurso compartilhado de cada vez, garantindo a consistência dos dados e evitando condições de corrida.

Semáforo: Primitiva de sincronização que pode ser utilizada para controlar o acesso a recursos compartilhados. Pode ser usado como um contador para permitir ou bloquear o acesso a um recurso, evitando problemas como condições de corrida.

Caldeirão: Recurso compartilhado no problema do jantar dos canibais. Representa o local onde as porções de comida são armazenadas e compartilhadas pelos canibais.

Canibal: Entidade que participa do problema do jantar dos canibais. Os canibais se servem do caldeirão para obter porções de comida.

Cozinheiro: Entidade que participa do problema do jantar dos canibais. O cozinheiro é responsável por encher o caldeirão quando ele estiver vazio.

Exclusão Mútua: Propriedade que garante que apenas uma entidade (thread ou processo) pode acessar um recurso compartilhado de cada vez.

Concorrência: Situação em que múltiplas threads ou processos estão em execução simultaneamente, realizando suas tarefas concorrentemente.

Thread: Unidade básica de execução em um programa. Pode ser considerada como uma "sub-rotina"que compartilha o mesmo espaço de memória com outras threads de um mesmo processo.

Processo: Programa em execução em um sistema operacional. Pode conter uma ou mais threads que compartilham recursos e executam concorrentemente.

Recursos Compartilhados: Recursos (como variáveis, memória, dispositivos, etc.) que podem ser acessados e modificados por múltiplas threads ou processos.

Coordenação: Ato de organizar e controlar a execução de threads ou processos para garantir a correta sincronização de suas ações.

Anexo A: Solução em C para o problema clássico dos canibais e do cozinheiro usando threads e semáforos.

```c
#include <stdio.h>
#include <stdlib.h>
#include <pthread.h>
#include <semaphore.h>
#include <unistd.h> // To the usleep function

#define N 5
// Max number of portin in cauldron
#define NUM_CANIBAIS 6

int caldeirao = 0; // Shared variable to represent the state of the cauldron

pthread_mutex_t mutex = PTHREAD_MUTEX_INITIALIZER; // Mutex to ensure mutual exclusion
sem_t encher_caldeirao_sem;                        // Semaphore light to control the filling of the cauldron
sem_t servir_sem;                                  // Semaphore light to control the pouring of the cauldron

void servir(long int canibal)
{

  sem_wait(&servir_sem);

  pthread_mutex_lock(&mutex);
  if (caldeirao > 0)
  {
    caldeirao--;
    sleep(1);
    printf("Canibal %ld se serviu do caldeir o. Restam %d por   o(es).\n", canibal, caldeirao);
  }
  if (caldeirao == 0)
  {
    sem_post(&encher_caldeirao_sem);
  }
  pthread_mutex_unlock(&mutex);
}

void comer(long int canibal)
{
  printf("Canibal %ld est  comendo.\n", canibal);
  pthread_mutex_lock(&mutex);
  if (caldeirao > 0)
    sem_post(&servir_sem);
  pthread_mutex_unlock(&mutex);
  sleep(1);
  // Wait 1 second before the next action
}

void *canibal(void *arg)
{
  while (1)
  {
    servir((long)arg);
    comer((long)arg);
  }
}

void encher()
```

```
{
  sem_wait(&encher_caldeirao_sem);
  pthread_mutex_lock(&mutex);
  caldeirao = N;
  printf("Cozinheiro encheu o caldeir o com %d por   o(es).\n", caldeirao);
  sem_post(&servir_sem);
  pthread_mutex_unlock(&mutex);
}

void dormir()
{
  sleep(1);
  return;
}

void *cozinheiro(void *arg)
{
  while (1)
  {
    encher();
    dormir();
  }
}

int main()
{
  pthread_t canibais[NUM_CANIBAIS];
  pthread_t cozinheiro_thread;

  sem_init(&encher_caldeirao_sem, 0, 1);
  sem_init(&servir_sem, 0, caldeirao);

  for (long i = 0; i < NUM_CANIBAIS; i++)
  {
    pthread_create(&canibais[i], NULL, canibal, (void *)i);
  }

  pthread_create(&cozinheiro_thread, NULL, cozinheiro, NULL);

  for (int i = 0; i < NUM_CANIBAIS; i++)
  {
    pthread_join(canibais[i], NULL);
  }
  pthread_join(cozinheiro_thread, NULL);

  sem_destroy(&encher_caldeirao_sem);
  sem_destroy(&servir_sem);
  pthread_mutex_destroy(&mutex);

  return 0;
}
```

Anexo B: Solução em Python, gerador de ondas sinusoidais quadradas que mostram o estado de N canibais ao longo de N ciclos.

```
import threading
import time
import math
import matplotlib.pyplot as plt
import numpy as np

class Cannibal:
    def __init__(self, name, left_portion, right_portion, semaphore, mutex):
        self.name = name
        self.left_portion = left_portion
        self.right_portion = right_portion
        self.eating = False
        self.semaphore = semaphore
        self.mutex = mutex

    def get_portions(self):
        while True:
            if not self.left_portion.is_eaten() and not self.right_portion.is_eaten()
                :
                self.mutex.acquire()
                self.left_portion.take()
                self.right_portion.take()
                self.mutex.release()
                self.semaphore.acquire()
                break
            else:
                time.sleep(0.1)

    def release_portions(self):
        self.mutex.acquire()
        self.left_portion.release()
        self.right_portion.release()
        self.mutex.release()

class FoodPortion:
    def __init__(self):
        self.eaten = False

    def is_eaten(self):
        return self.eaten

    def take(self):
        self.eaten = True

    def release(self):
        self.eaten = False

def simulate_dinner(num_cannibals, cycles):
    # Create food portions
    food_portions = [FoodPortion() for _ in range(5)]

    # Create semaphore
    semaphore = threading.Semaphore(2)

    # Create mutex
    mutex = threading.Lock()

    # Create cannibals
    cannibals = []
    for i in range(num_cannibals):
```

```
        name = f'Cannibal {i+1}'
        left_portion = food_portions[i]
        right_portion = food_portions[(i+1) % num_cannibals]
        cannibal = Cannibal(name, left_portion, right_portion, semaphore, mutex)
        cannibals.append(cannibal)

    # Simulate cycles
    for _ in range(cycles):
        for cannibal in cannibals:
            cannibal.eating = True
            time.sleep(1)
            cannibal.eating = False
            time.sleep(1)

    # Configure the plot
    fig, (ax1, ax2) = plt.subplots(2, 1)

    # Plot the mutex and semaphore usage
    time_wave = np.linspace(0, 2 * np.pi * cycles, 1000)
    mutex_waveform = np.ones_like(time_wave)
    mutex_waveform[np.sin(3 * time_wave) < 0] = 0
    semaphore_waveform = np.ones_like(time_wave)
    semaphore_waveform[np.sin(2 * time_wave) < 0] = 0
    ax1.plot(time_wave, mutex_waveform, label='Mutex')
    ax1.plot(time_wave, semaphore_waveform, label='Semaphore')
    ax1.set_xlabel('Time')
    ax1.set_ylabel('Usage')
    ax1.set_title('Mutex and Semaphore Usage')
    ax1.legend()

    # Generate square waves for each cannibal
    waveforms = []
    for i in range(num_cannibals):
        waveform = np.zeros_like(time_wave)
        waveform[np.sin((i + 0.5) * time_wave) >= 0] = 1
        waveforms.append(waveform)

    # Plot the cannibals' eating/waiting states
    eating_waveform = np.zeros_like(time_wave)
    for waveform in waveforms:
        eating_waveform += waveform
    for i, waveform in enumerate(waveforms):
        ax2.plot(time_wave, waveform, label=f'Cannibal {i+1}')
    ax2.set_xlabel('Time')
    ax2.set_ylabel('State')
    ax2.set_title('Cannibals Eating/Waiting State')
    ax2.legend()

    # Display the plots
    plt.tight_layout()
    plt.show()

# Example usage with n cannibals and n cycles
num_cannibals = 3
cycles = 3
simulate_dinner(num_cannibals, cycles)
```

Anexo C: Solução em C, simulador do jantar dos canibais com um número variável de canibais

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

#define N 10
#define MAX_CANIBALS 10

typedef struct {
    int num_canibals;
    int count_mutex;
    int count_semaphore;
    double wait_time;
} ExperimentResult;

int caldeirao = 0;
pthread_mutex_t mutex = PTHREAD_MUTEX_INITIALIZER;
sem_t encher_caldeirao_sem;
sem_t servir_sem;

void* canibal(void* arg) {
    long id = (long)arg;
    double wait_time = 0.0;
    int count_mutex = 0;
    int count_semaphore = 0;

    int num_cycles = *((int*)arg);
    for (int cycle = 0; cycle < num_cycles; cycle++) {
        sem_wait(&servir_sem);

        pthread_mutex_lock(&mutex);
        if (caldeirao > 0) {
            caldeirao--;
            printf("Canibal %ld took a portion. Remaining portions: %d.\n", id,
                caldeirao);
        }
        pthread_mutex_unlock(&mutex);

        if (caldeirao == 0) {
            sem_post(&encher_caldeirao_sem);
        }

        printf("Canibal %ld is eating.\n", id);
        sleep(1);

        sem_post(&servir_sem);

        double start_time = (double)clock() / CLOCKS_PER_SEC;
        for (int i = 0; i < MAX_CANIBALS; i++) {
            if (i != id) {
                sem_wait(&servir_sem);
                double end_time = (double)clock() / CLOCKS_PER_SEC;
                wait_time += end_time - start_time;
                sem_post(&servir_sem);
            }
        }

        pthread_mutex_lock(&mutex);
```

```
60          count_mutex++;
61          count_semaphore = id - 1;
62          pthread_mutex_unlock(&mutex);
63      }
64
65      ExperimentResult* result = malloc(sizeof(ExperimentResult));
66      result->num_canibals = id;
67      result->count_mutex = count_mutex;
68      result->count_semaphore = count_semaphore;
69      result->wait_time = wait_time;
70
71      pthread_exit(result);
72  }
73
74  void* cozinheiro(void* arg) {
75      while (1) {
76          sem_wait(&encher_caldeirao_sem);
77
78          pthread_mutex_lock(&mutex);
79          caldeirao = N;
80          printf("The cook filled the caldeirao with %d portions.\n", caldeirao);
81          pthread_mutex_unlock(&mutex);
82      }
83  }
84
85  void write_data(const char* filename, ExperimentResult** results, int num_results) {
86      FILE* file = fopen(filename, "w");
87      if (file == NULL) {
88          printf("Error opening file.\n");
89          return;
90      }
91
92      for (int i = 0; i < num_results; i++) {
93          fprintf(file, "%d, %d, %d, %.3f\n", results[i]->num_canibals, results[i]->
                count_mutex, results[i]->count_semaphore, results[i]->wait_time);
94      }
95
96      fclose(file);
97  }
98
99  int main() {
100     int start_canibals = 1;
101     int end_canibals = 10;
102     int increment = 1;
103     int num_cycles = 3; // Define the cycles number
104
105     pthread_t canibals[MAX_CANIBALS];
106     pthread_t cozinheiro_thread;
107     ExperimentResult* results[MAX_CANIBALS];
108     int num_results = 0;
109
110     sem_init(&encher_caldeirao_sem, 0, 0);
111     sem_init(&servir_sem, 0, 1);
112
113     for (int num_cannibals = start_canibals; num_cannibals <= end_canibals;
            num_cannibals += increment) {
114         for (long i = 0; i < num_cannibals; i++) {
115             int* cycles = malloc(sizeof(int));
116             *cycles = num_cycles;
117             pthread_create(&canibals[i], NULL, canibal, cycles);
118         }
119
120         pthread_create(&cozinheiro_thread, NULL, cozinheiro, NULL);
121
```

```
122         for (int i = 0; i < num_cannibals; i++) {
123             ExperimentResult* result;
124             pthread_join(canibals[i], (void**)&result);
125             results[num_results++] = result;
126         }
127 
128         pthread_cancel(cozinheiro_thread);
129         if (num_cannibals >= 10) {
130             break;
131         }
132     }
133 
134     sem_destroy(&encher_caldeirao_sem);
135     sem_destroy(&servir_sem);
136     pthread_mutex_destroy(&mutex);
137 
138     write_data("output.txt", results, num_results);
139 
140     for (int i = 0; i < num_results; i++) {
141         free(results[i]);
142     }
143 
144     return 0;
145 }
```