Universidade Federal do Rio Grande do Norte
Instituto Metrópole Digital
IMD0036 – Sistemas Operacionais

Programação Concorrente: Implementação

Prof. Gustavo Girão
girao@imd.ufrn.br

Roteiro

- Mutexes em Pthreads
- Variáveis condicionais
- Exercícios
- Implementação de Semáforos
- Jantar dos Filósofos

Mutexes em Pthreads

- Declaração
  - **pthread_mutex_t** **nome*
- Inicialização
  - **pthread_mutex_init**(*pthread_mutex_t* *nome, *pthread_mutexattr_t* *attr);
  - Inicializa o mutex considerando os atributos em *attr*
- Destruição
  - **pthread_mutex_destroy**(*pthread_attr_t* *nome)
- Lock
  - *int* **pthread_mutex_lock**(*pthread_mutex_t* *nome);
    - ✧ Atômico
- Unlock
  - *int* **pthread_mutex_unlock**(*pthread_mutex_t* *nome);
  - Atômico

Variáveis Condicionais

- Declaração
  - **pthread_cond_t** **nome*
- Inicialização
  - **pthread_cond_init**(*pthread_cond_t* *nome, *pthread_condattr_t* *attr);
- Destruição
  - **pthread_cond_destroy**(*pthread_cond_t* *nome)

Variáveis Condicionais

- *int* **pthread_cond_wait**(*pthread_cond_t* *cond, *pthread_mutex_t* *mutex)
  - Libera o mutex e bloqueia esperando um sinal na variável *cond*
- *int* **pthread_cond_signal**(*pthread_cond_t* *cond)
  - Libera uma thread bloqueada na variável *cond*
  - Se houver mais de uma thread esperando, o escalonador define a thread a ser desbloqueada
- *int* **pthread_cond_broadcast**(*pthread_cond_t* *cond)

Exercício 1

- Crie um código em C que
  - Declare uma matriz de 100 linhas e 10 colunas
  - Crie 10 threads
  - Cada thread deve somar todos os elementos de uma coluna e imprimir o valor total
  - Garanta a exclusão mútua no acesso à matriz

Produtor Consumidor em Pthreads

- Exemplo simples utilizando apenas um buffer de uma posição
- ex4_prodcons_mutex.c

Semáforos

- Biblioteca semaphore.h
- Declaração
  - **sem_t** **nome*
- Inicialização
  - **sem_init**(*sem_t* *nome, *int* pshared, *unsigned int* value);
    - ✧ pshared determina o compartilhamento:
      - Se for 0, o semáforo é compartilhado entre as threads do processo
      - Caso contrário, o semáforo é compartilhado entre processos e deve ser inserido em uma região de memória compartilhada
    - ✧ Value é o valor inicial do semáforo
- Destruição
  - **sem_destroy**(*sem_t* *nome)

Semáforos

- **sem_wait**(*sem_t* *nome)
  - Se refere a um "down" em um semáforo
  - Se o valor atual for zero, a thread é bloqueada
  - Caso contrário, o valor do semáforo é decrementado

- **sem_post**(*sem_t* *nome)
  - Se refere a um "up" em um semáforo
  - Incrementa o valor de um semáforo
  - Se o valor anterior era zero, uma thread bloqueada nesse semáforo é desbloqueada

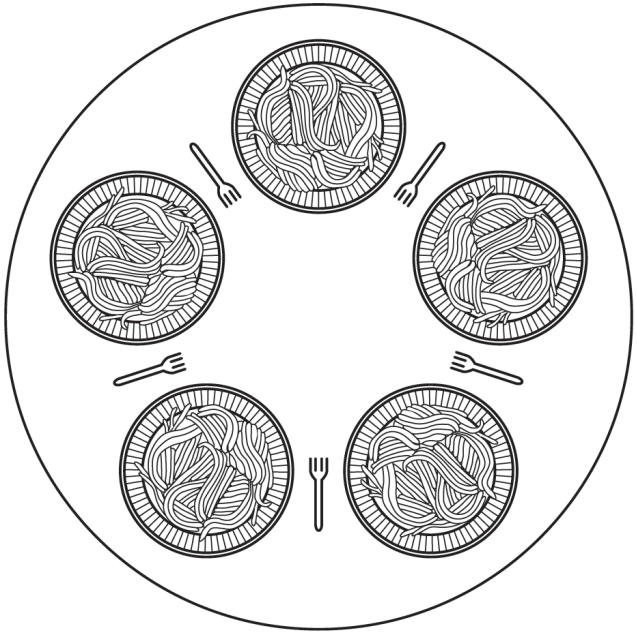
Exercício 2

- Baseado no exemplo anterior e no código apresentado na ultima aula, Implemente o produtor consumidor com um buffer de 10 posições
- Você vai precisar de:
  - Um vetor de inteiros (o buffer)
  - Uma variável inteira que guarda o próximo lugar do buffer em que o PRODUTOR irá escrever
  - Uma variável inteira que guarda o próximo lugar do buffer do qual o CONSUMIDOR irá ler
- Garanta, através de semáforos, que o produtor-consumidor garante exclusão mútua e sincronizaçãp

```
#define N 100                                   /* número de lugares no buffer */
typedef int semaphore;                          /* semáforos são um tipo especial de int */
semaphore mutex = 1;                            /* controla o acesso à região crítica */
semaphore empty = N;                            /* conta os lugares vazios no buffer */
semaphore full = 0 ;                            /* conta os lugares preenchidos no buffer */

void producer(void)
{
      int item;

      while (TRUE) {                            /* TRUE é a constante 1 */
            item = produce_item( );             /* gera algo para pôr no buffer */
            down(&empty);                       /* decresce o contador empty */
            down(&mutex);                       /* entra na região crítica */
            insert_item(item);                  /* põe novo item no buffer */
            up(&mutex);                         /* sai da região crítica */
            up(&full);                          /* incrementa o contador de lugares preenchidos */
      }
}


void consumer(void)
{
      int item;

      while (TRUE) {                            /* laço infinito */
            down(&full);                        /* decresce o contador full */
            down(&mutex);                       /* entra na região crítica */
            item = remove_item( );              /* pega o item do buffer */
            up(&mutex);                         /* deixa a região crítica */
            up(&empty);                         /* incrementa o contador de lugares vazios */
            consume_item(item);                 /* faz algo com o item */
      }
}
```

Jantar dos filósofos

- Cinco filósofos estão jantando em uma circular, lado a lado
- Quando não está comendo, o filósofo está pensando
- Precisa de dois garfos (o seu, na esquerda e o do vizinho da direita) para comer



Jantar dos filósofos

- Solução óbvia

```
#define N 5                                     /* número de filósofos */

void philosopher(int i)                         /* i: número do filósofo, de 0 a 4 */
{
        while (TRUE) {
                think( );                       /* o filósofo está pensando */
                take_fork(i);                   /* pega o garfo esquerdo */
                take_fork((i+1) % N);           /* pega o garfo direito; % é o operador módulo */
                eat( );                         /* hummm! Espaguete*/
                put_fork(i);                    /* devolve o garfo esquerdo à mesa */
                put_fork((i+1) % N);            /* devolve o garfo direito à mesa */
        }
}
```

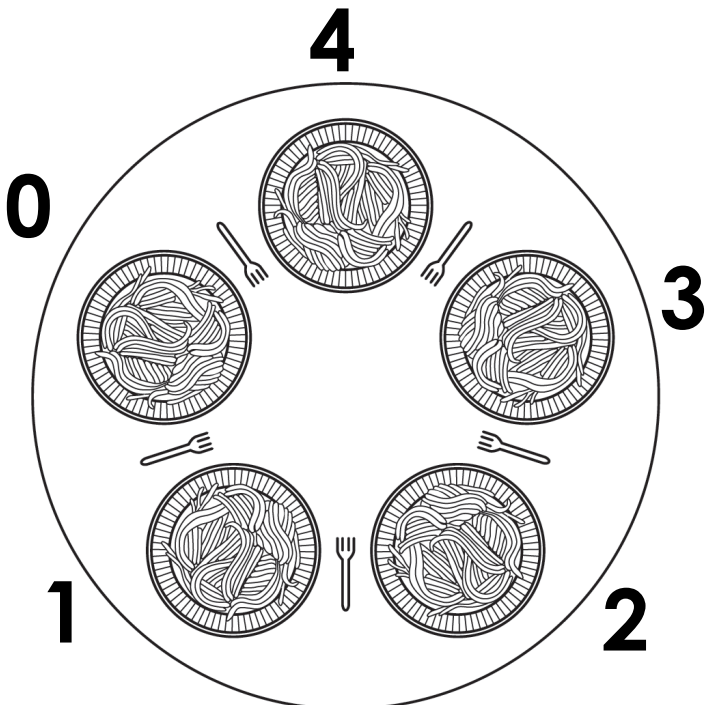
Errada!
Imagine que todos os filósofos resolvam comer ao mesmo tempo...

Jantar dos filósofos

- Cada filósofo é uma thread que precisa
  - Pensa (filosofa)
  - Verificar se tem dois garfos disponíveis
    - Os garfos tem que ser adjacentes a ele
  - Caso estejam disponíveis, ele come
  - Põe os garfos de volta à mesa
  - Repete o processo....
- **Como implementamos isso com semáforos?**

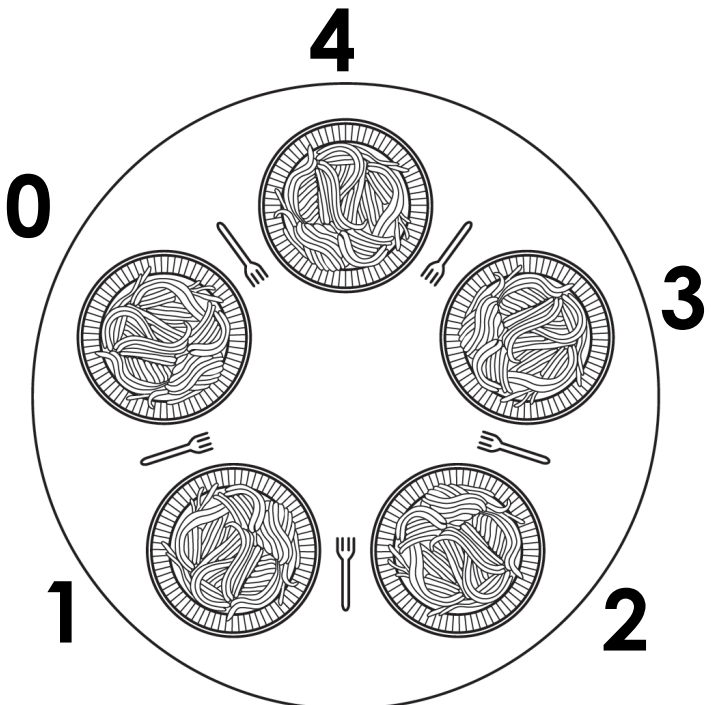
Jantar dos filósofos

- Imagine uma solução em que cada filósofo é representado por um número e pode estar:
  - Comendo (EATING)
  - Filosofando (THINKING)
  - Tentando comer (HUNGRY)



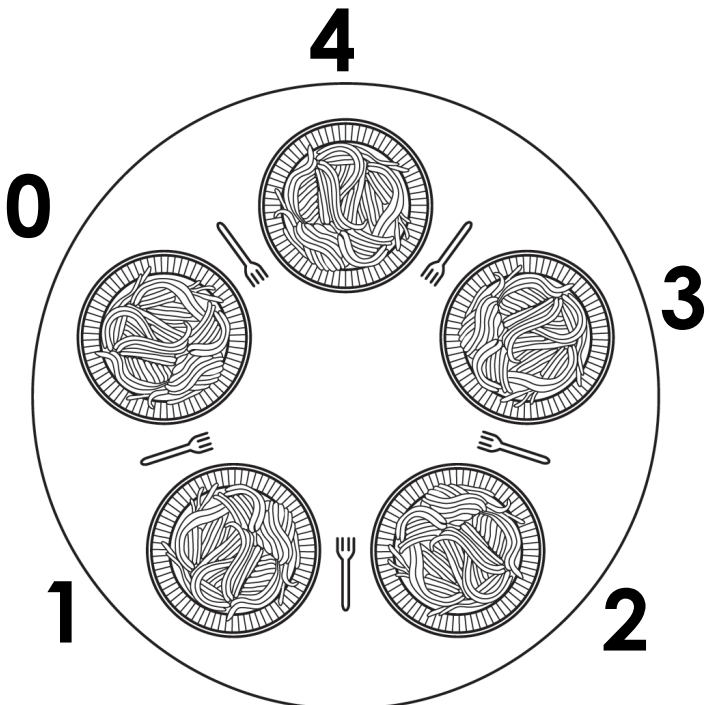
Jantar dos filósofos

- Por precisarem de dois garfos, em uma mesa com cinco filósofos, no máximo dois deles estarão comento ao mesmo tempo



Jantar dos filósofos

- Isso significa que:
  - 1) Se um filósofo está comendo, seus vizinhos não estão comendo (estarão filosofando ou com fome)
  - 2) Quando pelo menos 1 de seus vizinho esta comendo, ele não pode comer



Jantar dos filósofos

- As funções pensar e comer são meras abstrações
- O importante é coordenar as ações de pegar os garfos e colocá-los de volta na mesa

- Take_forks
  - Precisa estabelecer que o filósofo não está pensando e nem comendo. Logo, está com fome (HUNGRY)
  - Precisa testar se seus vizinhos estão comendo
  - As duas ações anteriores precisam ser feitas com exclusão mútua
  - Caso esteja com fome e possa comer
    - Dá UP no mutex
  - Por fim, verifica se os garfos foram pegos. Em caso negativo, bloqueia e espera (com fome....)

Jantar dos filósofos

- Put_forks
  - Estabelece que o filósofo agora vai filosofar (THINKING)
  - Verifica se algum dos seus vizinhos está com fome
    - Já que agora, ele irá liberar seus dois garfos
    - Em caso positivo, o mutex do filósofo será liberado
  - É importante notar que a função put_forks tem o potencial de liberar o mutex de DOIS filósofos

Jantar dos filósofos

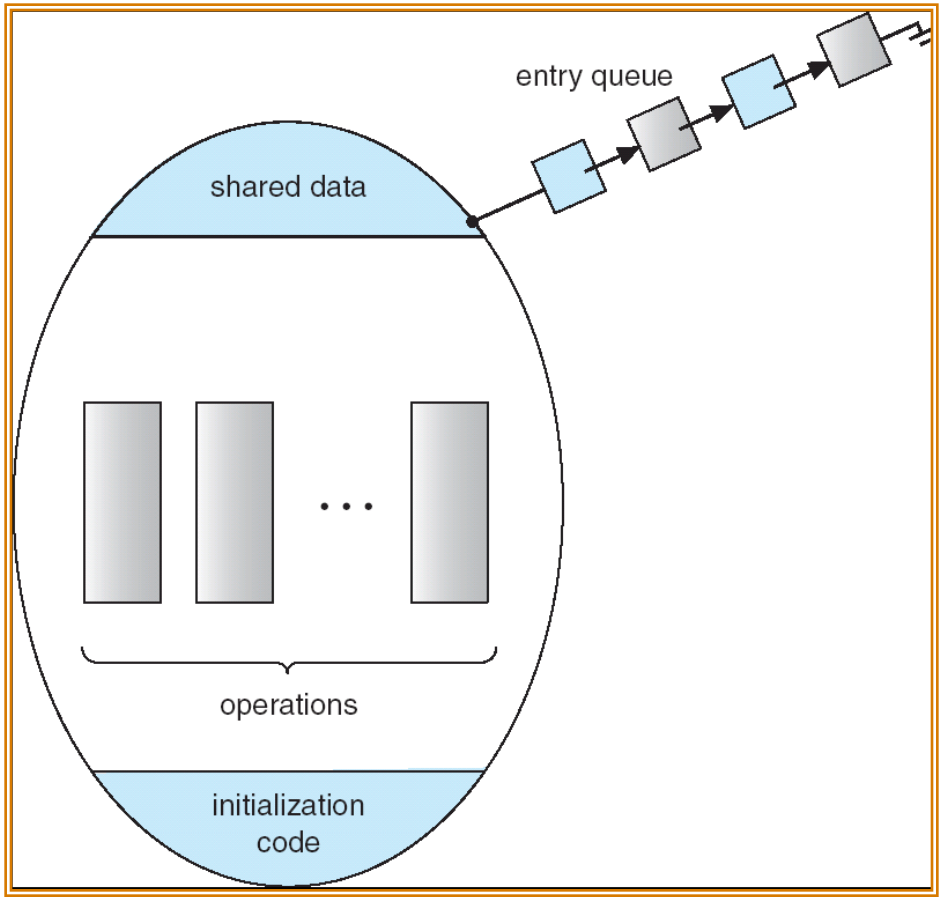
- Test
  - Verifica:
    - Se o filósofo sendo verificado está com fome (ou seja, tentou comer e não conseguiu) e;
    - Se os vizinhos desse filósofo não estão comendo
  - Caso as duas condições sejam verdadeiras, libera o mutex desse filósofo

Monitores

- Abstração de alto nível que fornece um mecanismo conveniente e eficiente para sincronização de processos
- Somente um processo por vez pode estar ativo dentro do monitor

```
monitor nome-monitor
{
   // declaração de variáveis compartilhadas
   procedure P1 (…) { …. }
            …
   procedure Pn (…) {……}

    Código de Inicialização ( ….) { … }
            …
   }
}
```

Visão Esquemática de um Monitor



Referências

- OLIVEIRA, Rômulo Silva de; CARISSIMI, Alexandre da Silva; TOSCANI, Simão Sirineo. **Sistemas operacionais**. 4. ed. Porto Alegre: Bookman, 2010. ISBN: 9788577805211.
  - **Capitulo 3**

- TANENBAUM, Andrew S.. **Sistemas operacionais modernos**. 3. ed. São Paulo: Prentice Hall, 2009. 653 p. ISBN: 9788576052371.
  - **Capítulo 2, Seção 2.3**

- SILBERCHATZ, A.; Galvin, P.; Gagne, G.; **Fundamentos de Sistemas Operacionais**, LTC, 2015. ISBN: 9788521629399
  - **Capítulo 6**

Próxima Aula

- Deadlocks