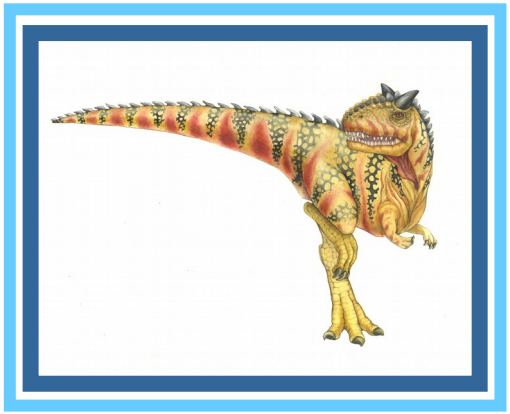
Capítulo 5: Sincronização entre Processos

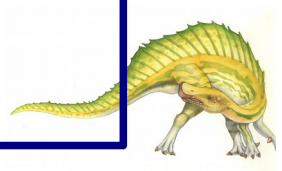


Semáforo

- Semáforo é uma ferramenta de sincronização entre processos mais sofisticada do que espera ocupada. É utilizado para controlar o acesso a seções críticas de processos evitando condições de corrida.

- Internamente, cada semáforo contém um **contador inteiro s.counter** e uma **fila de tarefas s.queue**, inicialmente vazia. Sobre essa variável podem ser aplicadas duas operações atômicas, descritas a seguir:
  - **Down** (ou **wait**): usado para solicitar acesso à seção crítica.
    - caso a seção esteja livre, a operação retorna imediatamente e a tarefa pode continuar sua execução;
    - caso contrário, a tarefa solicitante é suspensa e adicionada à fila do semáforo;

```
Down(s): // executa de forma atômica
s.counter ← s.counter − 1
if s.counter < 0 then
    insere a tarefa corrente no final de s.queue
    suspende a tarefa corrente
end if
```

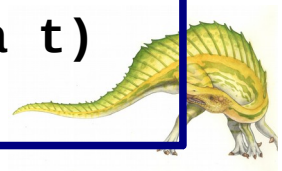


Semáforo

- Semáforo é uma ferramenta de sincronização entre processos mais sofisticada do que espera ocupada. É utilizado para controlar o acesso a seções críticas de processos evitando condições de corrida.

- Internamente, cada semáforo contém um **contador inteiro s.counter** e uma **fila de tarefas s.queue**, inicialmente vazia. Sobre essa variável podem ser aplicadas duas operações atômicas, descritas a seguir:
  - **Up** (ou **signal**): invocado para liberar a seção crítica associada a s; caso a fila do semáforo não esteja vazia, a primeira tarefa da fila é acordada, sai da fila do semáforo e volta à fila de tarefas *prontas*.

```
Up(s): // executa de forma atômica
s.counter ← s.counter + 1
if s.counter ≤ 0 then
    retira a primeira tarefa t de s.queue
    devolve t à fila de tarefas prontas (acorda t)
end if
```



Semáforo

- Um semáforo controla a quantidade de um certo recurso disponível.
  - Ele não controla quais são os recursos disponíveis.

- O **counter** do semáforo positivo, indica quantas instâncias do recurso estão disponíveis;
  - já o **counter** negativo indica quantas tarefas estão aguardando para usar o recurso.

- Erros comuns:
  - Alocar e não liberar um recurso que não é mais necessário;
  - Liberar um recurso que não foi alocado;
  - Alocar um recurso por um longo período de tempo sem utilizá-lo;
  - Utilizar um recurso sem alocá-lo com antecedência.

Semáforo

- Para evitar ***starvation*** (processo nunca obtém o recurso requisitado), os processos aguardando por um recurso podem ser enfileirados (FIFO).
  - Os processos na fila aguardando a liberação de um recurso podem possuir prioridades.
  - A implementação de FIFO não é obrigatória.
    - Os processos que estão aguradando podem ser aleatoriamente escolhidos.
    - Neste caso, pode ocorrer **starvation**?
- As operações **Up** and **Down** do semáforo devem ser atômicas.
  - Essas operações são implementadas pelo SO.

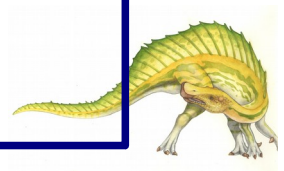
Semáforo

Analise a afirmativa a seguir.

O SO pode usar espera ocupada na atualização do semáforo pois as operações *Up* and *Down* são bem rápidas.

```
Up(s): // executa de forma atômica
s.counter ← s.counter + 1
if s.counter ≤ 0 then
    retira a primeira tarefa t de s.queue
    devolve t à fila de tarefas prontas (acorda t)
end if
```

```
Down(s): // executa de forma atômica
s.counter ← s.counter − 1
if s.counter < 0 then
    insere a tarefa corrente no final de s.queue
    suspende a tarefa corrente
end if
```



Semáforo

- Quem é responsável por gerenciar o *counter* e a fila dos processos que estão esperando para utilizar o recurso controlado pelo semáforo?

- Quando um processo solicita um recurso que não está disponível, quem é o responsável por alterar o estado do processo para *waiting* e inseri-lo na fila de processos aguardando pelo recurso?

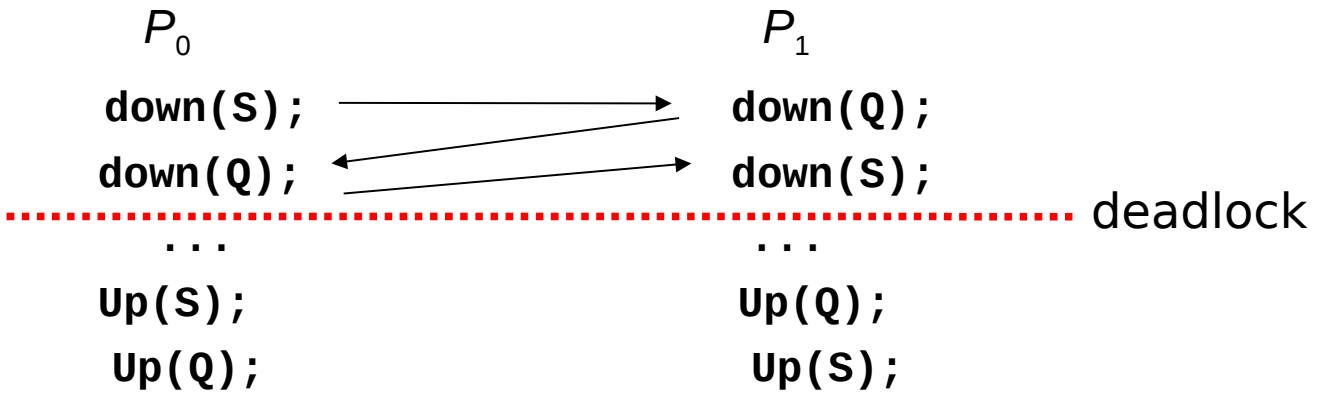
Semáforo

- Um mutex é essencialmente um semáforo binário (counter é igual 0 ou 1).
  - Apenas o processo que alocou o recurso é supostamente o que irá liberá-lo.

Estudar programa **printx_mutex.c** usando semáforo binário (mutex).

Deadlock e Starvation

- **Deadlock** – dois ou mais processos estão esperando por um evento que somente pode ser gerado por um dos processos que estão esperando.
- Considere ***S*** e ***Q*** dois semáforos inicializados em 1

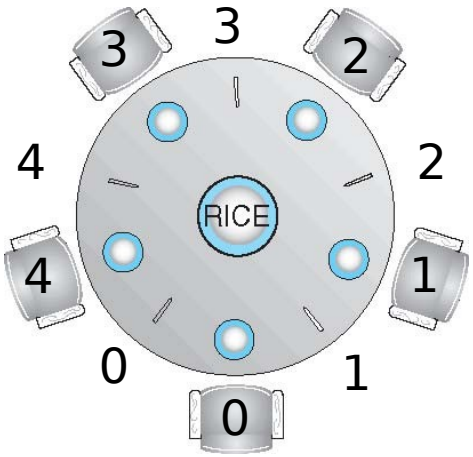


- **Starvation (inanição)** – bloqueio indefinido
  - Um processo nunca é removido da fila de suspenso do semáforo.

Problemas clássicos de sincronização

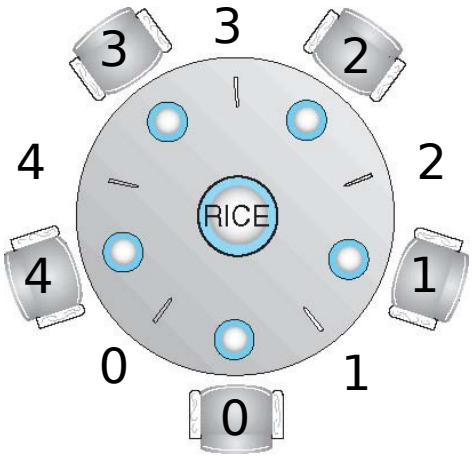
- Problema do jantar dos filósofos.
- Problema do buffer limitado.
- Problema dos leitores e escritores.

O problema do jantar dos filósofos



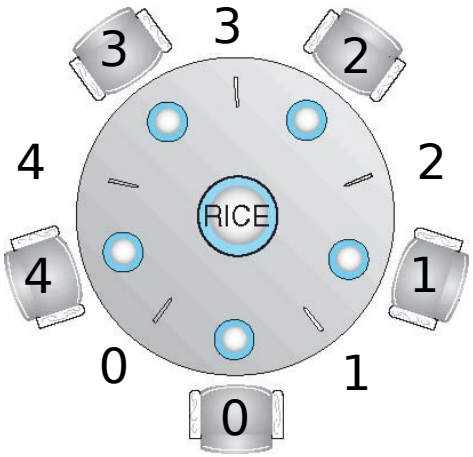
- Tarefa: filósofos passam o tempo alternando entre comendo e pensando. Para comer eles precisam pegar os palitos (chopsticks) da direita e da esquerda.
  - O filósofo somente pode comer se estiver com os dois palitos.
- Como implementar o problema de tal forma que não ocorra condição de corrida?
  - Existe uma variável compartilhada correspondente ao alimento (*rice*).
  - Além disso, existem os palitos do lado esquerdo e direito.

O problema do jantar dos filósofos



- Sugestão:
  - implementar 1 semáforo binário para o *rice* e 1 semáforo binário para cada palito, pois um palito pode ser utilizado somente por um filósofo de cada vez.
  - Cada filósofo é uma thread.
  - Estudar programa **dinningProblem.c**

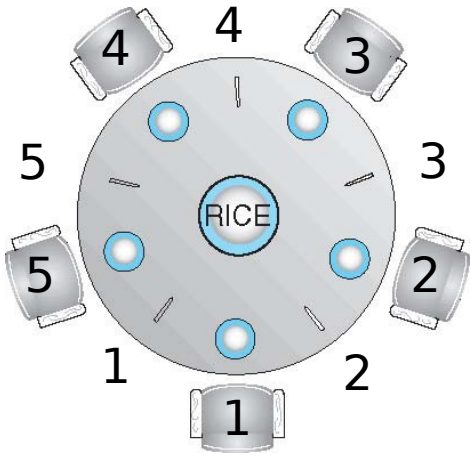
O problema do jantar dos filósofos



- Quando pode ocorrer *deadlock* ?
  - Um deadlock pode ocorrer quando cada filósofo está segurando um palito e esperando por outro de seu vizinho.

O problema do jantar dos filósofos

- As soluções abaixo resolvem o problema de deadlock no jantar dos filósofos?
  - Permitir no máximo 4 filósofos simultaneamente na mesa.
  - Permitir que um filósofo pegue e segure um palito apenas se os dois (esquerdo e direito) estiverem disponíveis.
  - O filósofo de número ímpar pega o palito da esquerda e depois o da direita. O filósofo de número par pega primeiro o palito da direita e depois o da esquerda.



O problema do buffer limitado

- Semáforos:

  empty =n → armazena a quantidade de posições vazias.

  full =0 → armazena a quantidade de posições preenchidas.

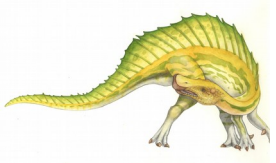
  mutex =1 → fornece exclusão mútua ao buffer

```
//Produtor
do {
   down(empty);//empty--;
   down(mutex);
  /* adiciona item*/
   up(mutex);
   up(full);//full++;
} while (true);
```

```
//Consumidor
  do {
      down(full);//full--;
      down(mutex);
      /*remove item*/
      up(mutex);
      up(empty);//empty++;
   } while (true);
```

Se **empty <=0**, produtor é bloqueado. Se **full <=0**, então consumidor é bloqueado.

Qual o problema de se utilizar apenas 1 semáforo e 1 mutex?



O problema do buffer limitado

```
//Produtor
do {
   down(empty);//empty--;
   down(mutex);
   /* adiciona item*/
   up(mutex);
   up(full);//full++;
} while (true);
```

```
//Consumidor
do {
    down(full);//full--;
    down(mutex);
    /*remove item*/
    up(mutex);
    up(empty);//empty++;
 } while (true);
```

As threads são bloqueadas para valores <=0. Por isso estão sendo utilizados dois semáforos. **Não** é possível fazer algo do tipo: “se full==MAX então coloca a thread em espera”.

empty → pode fazer o produtor entrar em estado de espera.

full → pode fazer o consumidor entrar em estado de espera.

O problema do buffer limitado

- Estudar programa: boundedBuffer.c

O problema dos leitores e escritores

- Um conjunto de dados é compartilhado entre vários processos concorrentes.
  - Leitores – podem ler e não fazem atualizações.
  - Escritores – podem ler e atualizar os dados.
- Problema
  - múltiplos leitores podem acessar o dado ao mesmo tempo.
  - quando um escritor deseja atualizar o dado, o acesso deve ser exclusivo.
- Existem diferentes variações do problema, cada um com prioridades diferentes entre leitores e escritores.

first readers - writers

```
//Reader
do {
   Down(mutex);
   read_count++;
   if (read_count == 1)
      Down(rw_mutex);
   Up(mutex);
   /* reading is performed */
   Down(mutex);
   read_count--;
   if (read_count == 0)
      Up(rw_mutex);
   Up(mutex);
} while (true);
```

Exclusão mútua para "read_count" (número de leitores)

O primeiro leitor bloqueia os escritores

O último leitor libera os escritores

.

```
//Writer
do {
      Down(rw_mutex);
   /*writing is performed */
      Up(rw_mutex);
   } while (true);
```

O problema dos leitores e escritores

- Programa: readers_writers.c

- Observe que o escritor somente pode escrever quando não há nenhum leitor. Variações no algoritmo podem ser feitas de tal forma que o escritor tenha uma maior prioridade.

Erros no uso de semáforos

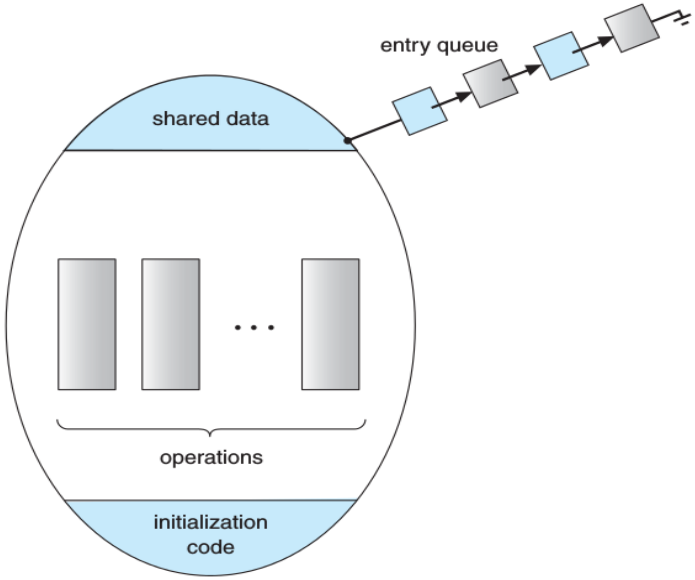
- Erros no uso de semáforos
  - Up (mutex) …. Down (mutex)
  - Down (mutex) … Down (mutex)
  - Omissão de Down (mutex) e/ou Up (mutex)
- Mesmo com semáforos, deadlocks e starvation (inanição) podem acontecer.

Variáveis de condição

- Estudar os programas:
  - conditionVariables1.c
  - conditionVariables2.c

Monitores

- Tipo abstrato de dados (TAD) que possui
  - Dado compartilhado
  - Procedimentos para manipular os dados
  - Mecanismo de sincronização que controla o acesso concorrente ao dado compartilhado.
- Apenas um processo de cada vez pode executar dentro do monitor. Se outro processo tenta acessar o monitor, ele é colocado em estado de espera.



“monitor-like” em Java

```
class Conta
{
   private float saldo = 0;

   public synchronized void depositar (float valor)
   {
      if (valor >= 0)
         saldo += valor ;
      else
         System.err.println("valor negativo");
   }

   public synchronized void retirar (float valor)
   {
      if (valor >= 0)
         saldo -= valor ;
      else
         System.err.println("valor negativo");
   }
}
```

End of Chapter 5