FATEC

CENTRO PAULA SOUZA
COMPETÊNCIA EM EDUCAÇÃO PÚBLICA PROFISSIONAL
Faculdade de Tecnologia Prof. Jessen Vidal
São José dos Campos

GOVERNO DO ESTADO DE SÃO PAULO

Curso Superior de Tecnologia em Análise e Desenvolvimento de Sistemas

Disciplina: Sistemas Operacionais I

Aula 13: Sincronismo de Processos P2

Prof. Diogo Branquinho Ramos
diogo.branquinho@fatec.sp.gov.br
São José dos Campos - SP

Roteiro

- Solução de Peterson
- Semáfaros
- Solução com semáfaro para o buffer limitado

Estrita alternância: problema

```
while (true){
   while (turn != 0);
   regiao_critica(); //ráp.
   turn=1;
   regiao_nao_critica(); //ráp.
}
```

P_0

```
while (true){
   while (turn != 1);
   regiao_critica(); //ráp.
   turn=0;
   regiao_nao_critica(); //dev.
}
```

P_1

Efeito comboio

Solução de Peterson

- Os dois processos compartilham duas variáveis:
  - int turn;
  - boolean flag[2]
- A variável turn indica de quem é a vez de entrar na seção crítica.
- O array flag indica se um processo está pronto para entrar na seção crítica.
  - flag[i] = true implica que o processo P_i está pronto!

Solução de Peterson: algoritmo

Inicialização:
turn=0
i=0
j=1

P_i

```
while (true){
    flag[i] = true;
    turn = j;
    while(flag[j] && turn==j);
    secao_critica();
    flag[i] = false;
    secao_nao_critica();
}
```

P_j

```
while (true){
    flag[j] = true;
    turn = i;
    while(flag[i] && turn==i);
    secao_critica();
    flag[j] = false;
    secao_nao_critica();
}
```

Solução de Peterson

- **A exclusão mútua é preservada**
  - Mesmo que o outro processo esteja pronto, o laço da seção crítica só funciona para um processo.
- **O requisito do progresso é satisfeito**
- **O requisito de espera limitada é atendido**

- **Mas qual o problema desta solução?**

Problema das variáveis de travamento

- **turn e flag estão compartilhadas!**
  - Para entrar na seção crítica é preciso testar a variável de travamento.
    - Se for 0, muda para 1 e entra em sua seção crítica. Ao sair, retorna a variável para 0.
    - Se for 1, precisa esperar!
  - Paradoxo: se a variável é compartilhada, ela deveria estar em Seção Crítica e não fora controlando a entrada!

Alternativa: seção crítica usando *locks*

Edsger Dijkstra 1930-2002, Holanda.
Turing (1972). Físico e professor de Computação.
Contribuições:

- Problema do caminho mínimo;
- Desenvolveu o SO THE;
- Introduziu o conceito de Semáforo;
- Primeiro compilador para ALGOL60;
- Introduziu o conceito de pilha;
- Introduziu o conceito de *deadlock*;
- Introduziu o conceito de auto-estabilização em sistemas distribuídos.

```
while (true) {
        acquire lock
            critical section
        release lock
            remainder section
}
```

Semáforo

- **Semáforo**
  - Valor inteiro + operações modificadoras.
- **Operações modificadoras**
  - acquire() e release().
  - Dijkstra: P (proberen, testar) e V (verhogen, incrementar).
  - Tanenbaum: down e up.

```
acquire() {
    while value <= 0
       ; // no-op
    value--;
}

    release() {
       value++;
    }
```

Métodos precisam ser atômicos: quando um thread acessa o valor de um semáforo, nenhum outro pode acessar concorrentemente!

Semáforo

- **Semáforo contador:** valor inteiro pode variar por um domínio irrestrito.
- **Semáforo binário:** valor inteiro só pode variar entre 0 e 1.
  - Também conhecidos como locks mutex, pois oferecem exclusão mútua:

```
Semaphore S = new Semaphore();

S.acquire();

   // critical section

S.release();

   // remainder section
```

Semáforo com espera ocupada

- **Spinlock**
  - O processo “gira” enquanto espera pelo lock.
- **Vantagens**
  - Um thread pode “girar” em um processador enquanto o outro realiza sua seção crítica no outro processador.
  - Não há troca de contexto!
- **Desvantagem**
  - Desperdício de ciclos da CPU.

Semáforo sem espera ocupada

- **Duas operações:**
  - block: coloca o processo que chama a operação na fila de espera apropriada.
  - wakeup: remove um dos processos na fila de espera e o coloca na fila de prontos.

  → A CPU pode ou não ser escalonada imediatamente.
- **Composição do novo semáforo**
  - Um valor inteiro;
  - Métodos modificadores;
  - Uma fila de espera de processos associados ao semáforo.
    - Quando um novo processo precisa esperar pelo semáforo.

Semáforo sem espera ocupada

- Implementação de acquire():

```
acquire(){
    value--;
    if (value < 0) {
       add this process to list
       block;
    }
}
```

O valor do semáforo pode ser negativo: a magnitude é o número de processos esperando.

- Implementação de release():

```
release(){
    value++;
    if (value <= 0) {
       remove a process P from list
       wakeup(P);
    }
}
```

Antes:

```
acquire() {
    while value <= 0
       ; // no-op
    value--;
}

    release() {
       value++;
    }
```

Problema de buffer limitado

- **Semáforo mutex**
  - Inicializado com o valor 1;
  - Provê exclusão mútua para os acessos ao buffer.
- **Semáforo full**
  - Inicializado com o valor 0;
  - Conta a quantidade de espaços ocupados no buffer.
- **Semáforo empty**
  - Inicializado com o valor N (tamanho do buffer);
  - Conta a quantidade de espaços vazios no buffer.

Problema de buffer limitado

```
public class BoundedBuffer implements Buffer
{
   private static final int BUFFER_SIZE = 5;
   private Object[] buffer;
   private int in, out;
   private Semaphore mutex;
   private Semaphore empty;
   private Semaphore full;

   public BoundedBuffer() {
      // buffer is initially empty
      in = 0;
      out = 0;
      buffer = new Object[BUFFER_SIZE];

      mutex = new Semaphore(1);
      empty = new Semaphore(BUFFER_SIZE);
      full = new Semaphore(0);
   }

   public void insert(Object item) {
      // Figure 6.9
   }

   public Object remove() {
      // Figure 6.10
   }
}
```

Problema de buffer limitado

- Método insert()

```
public void insert(Object item) {
    empty.acquire();
    mutex.acquire();

    // add an item to the buffer
    buffer[in] = item;
    in = (in + 1) % BUFFER_SIZE;

    mutex.release();
    full.release();
}
```

Travando o acesso: nenhum processo pode inserir no *buffer* enquanto outro estiver inserindo.

Liberando o acesso

Problema de buffer limitado

- Método remove()

```
public Object remove() {
    full.acquire();
    mutex.acquire();

    // remove an item from the buffer
    Object item = buffer[out];
    out = (out + 1) % BUFFER_SIZE;

    mutex.release();
    empty.release();

    return item;
}
```

Problema de buffer limitado

- **Estrutura do processo produtor**

```
public class Producer implements Runnable
{
   private Buffer buffer;

   public Producer(Buffer buffer) {
      this.buffer = buffer;
   }

   public void run() {
      Date message;

      while (true) {
         // nap for awhile
         SleepUtilities.nap();
         // produce an item & enter it into the buffer
         message = new Date();
         buffer.insert(message);
      }
   }
}
```

Problema de buffer limitado

- **Estrutura do processo consumidor**

```
public class Consumer implements Runnable
{
   private Buffer buffer;

   public Consumer(Buffer buffer) {
      this.buffer = buffer;
   }

   public void run() {
      Date message;

      while (true) {
         // nap for awhile
         SleepUtilities.nap();
         // consume an item from the buffer
         message = (Date)buffer.remove();
      }
   }
}
```

Problema de buffer limitado

- **Main**

```
public class Factory
{
    public static void main(String args[]) {
       Buffer buffer = new BoundedBuffer();

       // now create the producer and consumer threads
       Thread producer = new Thread(new Producer(buffer));
       Thread consumer = new Thread(new Consumer(buffer));

       producer.start();
       consumer.start();
    }
}
```