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 12: Sincronismo de Processos P1

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

Roteiro

- Motivação
- Implementando sincronismo
- Problema de seção crítica

Metáfora

Hora	Pessoa A	Pessoa B
6:00	Olha a geladeira: sem leite	-
6:05	Sai para a padaria	-
6:10	Chega na padaria e compra leite	Olha a geladeira: sem leite
6:15	Sai da padaria	Sai para a padaria
6:20	Chega em casa: guarda o leite	Chega na padaria e compra leite
6:25	-	Sai da padaria
6:30	-	Chega em casa: Ah! Não!

Motivação para implementar sincronismo

- **Problemas no acesso concorrente**
  - Tal acesso a recursos compartilhados pode resultar em inconsistência.
- **Manutenção da consistência**
  - Requer mecanismos para garantir a execução ordenada dos processos em cooperação.
- **Produtor-consumidor manipulando buffer compartilhado**
  - Um contador inteiro que acompanha o preenchimento do buffer.
    - Incrementado pelo produtor
    - Decrementado pelo consumidor

Produtor

```
while (count == BUFFER_SIZE)
   ; // do nothing

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

Consumidor

```
while (count == 0)
   ; // do nothing

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

- **Rotinas corretas separadamente, porém o funcionamento fica incorreto quando executadas juntas!**

Condição de corrida

- **count++ em mais baixo nível:**

  register1 = count

  register1 = register1 + 1

  count = register1

- **count-- em mais baixo nível:**

  register2 = count

  register2 = register2 – 1

  count = register2

Condição de corrida

- **Considere esta execução intercalando com "count = 5" inicialmente:**

S0: produtor executa register1 = count {register1 = 5}
S1: produtor executa register1 = register1 + 1 {register1 = 6}
S2: consumidor executa register2 = count {register2 = 5}
S3: consumidor executa register2 = register2 - 1 {register2 = 4}
S4: produtor executa count = register1 {count = 6 }
S5: consumidor executa count = register2 {count = 4}

- **Condição de corrida (*race condition*)**
  - Quando há acesso simultâneo aos dados compartilhados e o resultado final depende da ordem de execução.
  - Estado incorreto: dois processos estão manipulando a variável **count** de maneira concorrente.

Problema de seção crítica: elementos

- **Seção crítica**
  - Código acessando recursos compartilhados.
- **Seção restante**
  - Código acessando recursos exclusivos.
- **Seção de entrada**
  - Código que permite a entrada na seção crítica.
- **Seção de saída**
  - Código executado após a saída da seção crítica relacionado ao acesso à seção crítica.

Estrutura de um processo típico

```
while (true) {
        entry section
            critical section
        exit section
            remainder section
}
```

Solução do problema de seção crítica: princípios

- **Exclusão mútua**
  - Somente um processo pode estar executando sua seção crítica.
- **Progresso**
  - Se não existem processos executando em sua seção crítica e alguns processos querem entrar em sua seção crítica, a seleção dos processos que entrarão na seção crítica em seguida não poderá ser adiada indefinidamente!
  - Só participam dessa seleção os processos que não estão em sua seção restante.

Solução do problema de seção crítica: princípios

- **Espera limitada**
  - Para entrar em sua seção crítica, o processo pode ter de esperar.
    - Entre a solicitação e a entrada na seção crítica, deve haver um limite sobre o número de vezes que outros processos têm permissão para entrar em suas seções críticas nesse intervalo.

Solução mais simples

- **Inibição das interrupções**
  - Ao entrar na região crítica, o processo desabilita interrupções, o que impede o escalonamento que usa a interrupção de tempo.
  - Desvantagem
    - Fornece a processos de usuários o poder de desabilitar interrupções! E se tal processo não habilitá-las novamente? E se tivermos mais de um processador?
  - Vantagem
    - É conveniente que o *kernel* possa inibir interrupções do sistema enquanto realizam tarefas críticas.

Estrita alternância

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

P_0

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

P_1

- **turn é a variável que estabelece quem vai entrar na seção crítica:**
  - 0 para P_0; 1 para P_1;
  - Espera ocupada: espera executando um loop.

Estrita alternância

```
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

Estrita alternância

- **Efeito comboio**
  - Suponha que P_1 execute rapidamente sua seção crítica: turn=0;
  - P_0 executa seu loop rapidamente retornando à seção não-crítica: turn=1;
  - Mesmo com P_1 em seu trecho não-crítico, P_0 não pode executar sua seção crítica, pois turn vale 1!

→ Ou seja, essa solução não é adequada quando temos um processo mais lento que o outro.

→ Nenhum processo que não esteja em sua seção crítica pode bloquear outro processo!