FACULDADE
SANTA TEREZINHA

SISTEMAS OPERACIONAIS

PROF CARLOS WAGNER
FACULDADE CEST

UNIDADE II – GERENCIAMENTO DE PROCESSOS (16h)

2.1 Modelos de processos

2.2 Escalonamento

2.3 Sincronização

2.4 Impasses

2.5 Gerenciamento de processos

2.3 - Sincronização

- **Conceitos**
- **O Problema da seção crítica**
- **Hardware de Sincronização**
- **Semáforos**
- **Problemas clássicos de sincronização**
- **Regiões críticas**
- **Monitores**

2.3 Sincronização – Buffer limitado

- **Dados compartilhados**

```
#define BUFFER_SIZE 10
typedef struct {
. . .
} item;
item buffer[BUFFER_SIZE];
int in = 0;
int out = 0;
int counter = 0;
```

2.3 Sincronização – Buffer limitado

- **Producer process**

```
item nextProduced;

while (1) {
        while (counter == BUFFER_SIZE)
        ; /* do nothing */
        buffer[in] = nextProduced;
        in = (in + 1) % BUFFER_SIZE;
        counter++;
}
```

- **Consumer process**

```
item nextConsumed;

while (1) {
        while (counter == 0)
                ; /* do nothing */
        nextConsumed = buffer[out];
        out = (out + 1) % BUFFER_SIZE;
        counter--;
        }
```

2.3 Sincronização – Buffer limitado

- O comando “**count++**” pode ser implementado em linguagem de máquina com o algoritmo abaixo:

```
register1 = counter
register1 = register1 + 1
counter = register1
```

- O comando “**count--**”pode ser implementado da seguinte forma:

```
register2 = counter
register2 = register2 – 1
counter = register2
```

2.3 Sincronização – Buffer limitado

- Se tanto o produtor quanto o consumidor tentarem atualizar o buffer simultaneamente, as instruções em linguagem assembly poderão ser intercaladas.

- A intercalação depende de como os processos do produtor e do consumidor são programados.

2.3 Sincronização – Buffer limitado

- Assuma que **counter** tem valor inicial 5. Uma intercalação de comandos é:

  produtor: **register1 = counter** (*register1 = 5*)
  produtor: **register1 = register1 + 1** (*register1 = 6*)
  consumidor: **register2 = counter** (*register2 = 5*)
  consumidor: **register2 = register2 – 1** (*register2 = 4*)
  produtor: **counter = register1** (*counter = 6*)
  consumdor: **counter = register2** (*counter = 4*)

- O valor de **count** pode ser tanto 4 ou 6, onde o resultado correto deve ser 5.

2.3 Sincronização – Condição de Corrida

- **Condição de corrida:** A situação em que vários processos acessam – e manipulam dados compartilhados simultaneamente. O valor final dos dados compartilhados depende de qual processo termina por último.

- Para evitar condições de corrida, os processos simultâneos devem ser sincronizados.

2.3 Sincronização – O Problema da Seção crítica

- n processos, todos competindo para usar alguns dados compartilhados
- Cada processo possui um segmento de código, denominado seção crítica, no qual os dados compartilhados são acessados.
- Problema – garantir que quando um processo estiver sendo executado em sua seção crítica, nenhum outro processo poderá ser executado em sua seção crítica.

2.3 Sincronização – O Problema da Seção crítica

Uma solução para o problema da seção crítica deve satisfazer aos três requisitos a seguir:

1. **Exclusão Mútua - Mutex.** Se o processo P_i estiver sendo executado em sua seção crítica, nenhum outro processo poderá estar sendo executado em suas seções críticas.
2. **Progresso**. Se nenhum processo estiver executando em sua seção crítica e existirem alguns processos que desejam entrar em sua seção crítica, então a seleção dos processos que entrarão em seguida na seção crítica não poderá ser adiada indefinidamente.
3. **Espera Limitada**. Deve existir um limite para o número de vezes que outros processos podem entrar em suas seções críticas após um processo ter feito uma solicitação para entrar em sua seção crítica e antes que essa solicitação seja atendida.
   - Suponha que cada processo seja executado a uma velocidade diferente de zero
   - Nenhuma suposição relativa à velocidade relativa dos n processos.

2.3 Sincronização – O Problema da Seção crítica

- Apenas 2 processos, P_0 e P_1
- Estrutura geral do processo P_i (outro processo P_j)

```
do {
        seção de entrada
              seção crítica
        seção de saída
              seção restante
} while (1)
```

- Os processos podem compartilhar algumas variáveis comuns para sincronizar suas ações.

2.3 Sincronização – Algoritmo 1

- Variáveis compartilhadas:
  - **int turn;**
    initially turn = 0
  - **turn - i => P_i** pode entrar em sua seção crítica
- Processo P_i

```
do {
        while (turn != i) ;
                seção crítica
        turn = j;
                seção restante
} while (1);
```

- Satisfaz exclusão mútua, mas o progresso

2.3 Sincronização – Algoritmo 2

- Variáveis compartilhadas:
  - **boolean flag[2];**
    initially flag [0] = flag [1] = false.
  - **flag [i] = true ⇑ P_i** pronto para entrar na sua seção crítica
- Processo P_i

```
do {
        flag[i] := true;
while (flag[j]) ;
            seção crítica
        flag [i] = false;
                    seção restante
} while (1);
```

- Satisfaz exclusão mútua, mas não o requisito de progresso

2.3 Sincronização – Algoritmo 3

- Variáveis combinadas compartilhadas dos algoritmos 1 e 2.
- Processo P_i

```
do {
        flag [i]:= true;
turn = j;
while (flag [j] and turn = j) ;
                seção crítica
        flag [i] = false;
                seção restante
} while (1);
```

- Atende a todos os três requisitos; resolve o problema da seção crítica para dois processos.

Seção crítica para n processos

- Antes de entrar na sua seção crítica, o processo recebe um número. O detentor do menor número entra na seção crítica.
- Se os processos P_i e P_j recebem o mesmo número, se i < j, então P_i é servido primeiro; caso contrário, P_j é servido primeiro.
- O esquema de numeração sempre gera números em ordem crescente de enumeração; ou seja, 1,2,3,3,3,3,4,5...

2.3 Sincronização – Algoritmo do padeiro

- Notação <🕒 ordem lexicográfica (número do ticket, número do ID do processo)
  - (a,b) <c,d) se a <c ou se a = c e b <d
  - max (a0,…, an-1) é um número, k, tal que k ai para i – 0, …, n – 1
- Dados compartilhados:

  boolean choosing[n];
  int number[n];

As estruturas de dados são inicializadas como falso e 0, respectivamente

2.3 Sincronização – Algoritmo do padeiro

```
do {
   choosing[i] = true;
   number[i] = max(number[0], number[1], …, number [n –
  1])+1;
   choosing[i] = false;
   for (j = 0; j < n; j++) {
      while (choosing[j]) ;
      while ((number[j] != 0) && (number[j,j] <
  number[i,i])) ;
   }
       Seção crítica
   number[i] = 0;
       Seção restante
} while (1);
```

2.3 Sincronização – Sincronização de hardware

- Testa e modifca o conteúdo de uma palavra atomicamente, isto é, como uma unidade impossível de interromper.

```
boolean TestAndSet(boolean &target) {
     boolean rv = target;
     target = true;

     return rv;
}
```

.

2.3 Sincronização – Exclusão mútua com testa e seta

- Dado compartilhado:

```
boolean lock = false;
```

- Processo P_i

```
do {
    while (TestAndSet(lock)) ;
      seção crítica
    lock = false;
      seção restante
}
```

2.3 Sincronização – Hardware de Sincronização

- Troca atomicamente duas variáveis

```
void Swap(boolean &a, boolean &b) {
     boolean temp = a;
     a = b;
     b = temp;
}
```

2.3 Sincronização – Exclusão mútua com troca

- Dado **compartilhado** (inicializado com false):

```
boolean lock;
boolean waiting[n];
```

Processo P_i

```
do {
    key = true;
    while (key == true)
            Swap(lock,key);
            critical section
    lock = false;
        remainder section
}
```

2.3 Sincronização – Semáforos

- Ferramenta de sincronização que não requer espera ocupada.
- Semáforo S – variável inteira
- só pode ser acessado através de duas operações indivisíveis (atômicas)

```
wait (S):
    while S ≤ 0 do no-op;
    S--;

signal (S):
    S++;
```

2.3 Sincronização – Seção crítica de n processos

- Dados compartilhados

  semaphore mutex; //initially mutex = 1

- Processo P_i

```
do {
    wait(mutex);
        Seção crítica
    signal(mutex);
        Seção restante
} while (1);
```

2.3 Sincronização – Implementação de semáforo

- Defina um semáforo como um registro

```
typedef struct {
        int value;
        struct process *L;
} semaphore;
```

- Assume duas operações simples:
  - **Block:** suspende o processo que a invoca.
  - **Wakeup(P):** retoma a execução do processo bloqueado **P**

2.3 Sincronização – Implementação

- Operações de semáforos agora definidas como

```
wait(S):
        S.value--;
        if (S.value < 0) {
            adiciona este processo para S.L;
            block;
        }

signal(S):
        S.value++;
        if (S.value <= 0) {
            remove um process P de S.L;
            wakeup(P);
        }
```

2.3 Sincronização – Semáforos como ferramenta de sincronização geral

- Executa B em P_j somente depois de ter executado em P_i
- Usa *flag* de semáforo inicializada em 0
- Código:

P_i	P_j
⋮	⋮
A	`wait(flag)`
`signal(flag)`	B

2.3 Sincronização – Deadlock e Starvation

- **Deadlock (Bloqueio Mortal)** – dois ou mais processos aguardam indefinidamente por um evento que pode ser causado por apenas um dos processos em espera.
- Sejam S e Q dois semáforos inicializados em 1

```
        P0              P1
wait(S);        wait(Q);
wait(Q);        wait(S);
  ⋮               ⋮
signal(S);      signal(Q);
signal(Q)       signal(S);
```

- **Starvation (Inanição, fome)**: bloqueio indefinido. Um processo nunca pode ser removido da fila do semáforo na qual está suspenso.

2.3 Sincronização – Dois tipos de semáforos

- **Semáforo de contagem** – o valor inteiro pode variar em um domínio irrestrito.
- **Semáforo binário** – o valor inteiro pode variar apenas entre 0 e 1; pode ser mais simples de implementar.
- Pode implementar um semáforo de contagem S como um semáforo binário.

2.3 Sincronização – S como semáforo binário

- Estruturas de dados:

  binary-semaphore S1, S2;

  int C:

- Inicialização:

  S1 = 1

  S2 = 0

  C = initial value of semaphore S

2.3 Sincronização – Implementando S

- Operação wait

```
wait(S1);
C--;
if (C < 0) {
    signal(S1);
    wait(S2);
}
signal(S1);
```

- Operação signal

```
wait(S1);
C ++;
if (C <= 0)
   signal(S2);
else
   signal(S1);
```

2.3 Sincronização – Problemas clássicos

- Problema do buffer limitado
- Problema dos leitores e escritores
- Problema dos filósofos comensais

2.3 Sincronização – Problema do buffer limitado

- Dado compartilhado

  semaphore full, empty, mutex;

  Inicialmente:

  full = 0, empty = n, mutex = 1

2.3 Sincronização – Problema do buffer limitado
Processo Produtor

```
do {
        …
   produz um item em nextp
        …
   wait(empty);
   wait(mutex);
        …
   adiciona nextp ao buffer
        …
   signal(mutex);
   signal(full);
} while (1);
```

2.3 Sincronização – Problema do buffer limitado
Processo Consumidor

```
do {
   wait(full)
   wait(mutex);
            …
   remove um item do buffer para nextc
            …
   signal(mutex);
   signal(empty);
            …
   consome o item em nextc
            …
} while (1);
```

2.3 Sincronização – Problema leitores-escritores

Dados compartilhados

```
semaphore mutex, wrt;
```

Inicialmente

```
mutex = 1, wrt = 1, readcount = 0
```

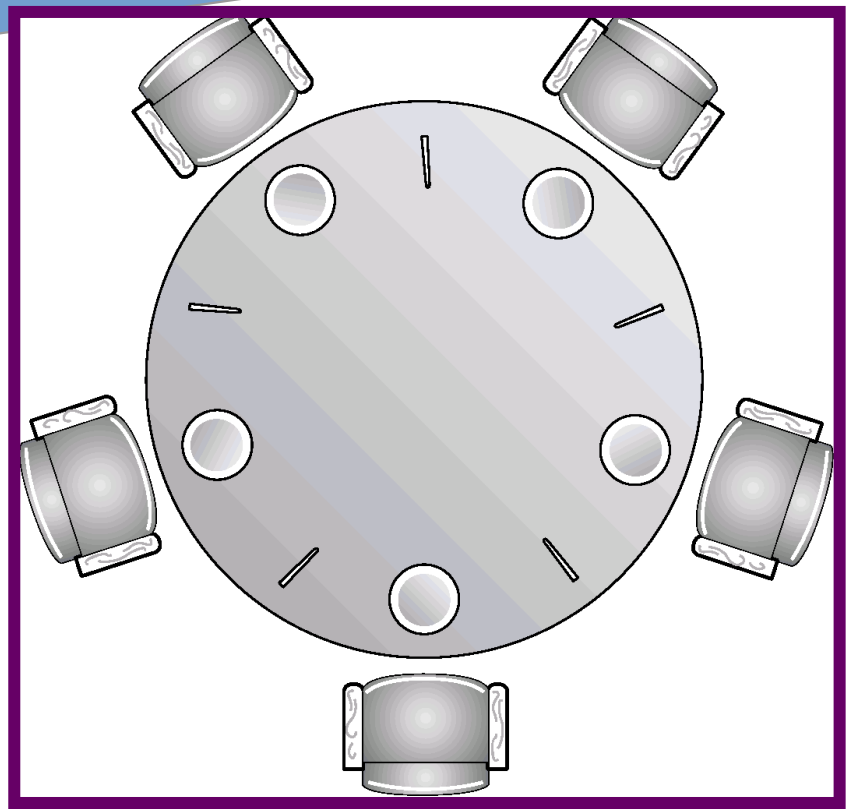
2.3 Sincronização – Problema leitores-consumidores. Processo escritor

```
wait(wrt);
          …
  a escrita é realizada
          …
signal(wrt);
```

2.3 Sincronização – Problema leitores-consumidores. Processo leitor

```
wait(mutex);
readcount++;
if (readcount == 1)
   wait(rt);
signal(mutex);
 …
 reading is performed
 …
wait(mutex);
readcount--;
if (readcount == 0)
   signal(wrt);
signal(mutex):
```

2.3 Sincronização – Problema dos filósofos comensais



Dados compartilhados

semaphore chopstick[5];

Inicialmente todos os valores são 1

2.3 Sincronização – Problema dos filósofos comensais

```
Philosopher i:
        do {
        wait(chopstick[i])
        wait(chopstick[(i+1) % 5])
         …
        eat
         …
        signal(chopstick[i]);
        signal(chopstick[(i+1) % 5]);
         …
        think
         …
        } while (1);
```

Contato:

“

wagner.costa@cest.edu.br

”