# Sincronização de Processos Sistemas Operacionais

#### Charles Tim Batista Garrocho

Instituto Federal de Minas Gerais – IFMG Campus Ouro Branco

garrocho.github.io

charles.garrocho@ifmg.edu.br

Sistemas de Informação



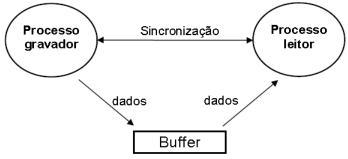
#### Conceitos Básicos

- Processo **Cooperativo** é aquele que pode afetar outros processos em execução, ou ser por eles afetados.
- Acesso concorrente a dados compartilhados pode resultar em inconsistências.
- Manter dados consistentes exige mecanismos para garantir a execução cooperativa de processos.



#### O Problema do Buffer Limitado

Também chamado de **Produtor e o Consumidor**, consiste em um conjunto de processos que compartilham um mesmo buffer. Os processos chamados **produtores** põem informação no buffer. Os processos chamados **consumidores** retiram informação deste buffer.



#### Buffer Limitado: Produtor e Consumidor

#### **PRODUTOR**

```
item nextProduced;
while (1) {
  while (counter == BUFFER_SIZE)
    /* buffer cheio */
    /* n\u00e3o faz coisa alguma */;
  buffer[counter] = nextProduced;
  counter++;
}
```

#### **CONSUMIDOR**

```
item nextConsumed;
while (1) {
  while (counter==0)
  /* buffer vazio */
  /* não faz coisa alguma */;
  nextConsumed = buffer[counter];
  counter--;
}
```



# Buffer Limitado (Memória Compartilhada)

#### Os comandos:

- o counter++; /\* produtor \*/
- counter-; /\* consumidor \*/

precisam ser executados de forma atômica.

isto é: não podem ser interrompidos no meio de sua execução.



# Buffer Limitado (Memória Compartilhada)

- Código para count++ em assembly:
- Código para count em assembly:

- a) MOV R1, \$counter
- b) INC R1
- c)MOV \$counter,R1

- x)MOV R2, \$counter
- y)DEC R2
- z)MOV \$counter, R2



# Buffer Limitado (Memória Compartilhada)

 Seja counter igual a 5 inicialmente e considerese a seguinte sequência de execução:

```
produtor: MOVR1, Scounter (R1 = 5)

produtor: NCR1 (R1 = 6)

consumidor: MOVR2, Scounter (R2 = 5)

consumidor: DECR2 (R2 = 4)

produtor: MOV Scounter, R1 (counter = 6)

consumidor: MOV Scounter, R2 (counter = 4)
```

 O valor de counter pode terminar como 4 ou 6, mas o correto seria 5!



## O Problema de Seção Crítica

- N processos competem para usar alguma estrutura de dados compartilhada.
- Cada processo tem um segmento de código comum onde a estrutura é acessada.
- Problema: garantir que quando um processo está executando aquele segmento de código, nenhum outro processo pode fazer o mesmo.

# O Problema de Seção Crítica

#### Requisitos da solução procurada:

- Exclusão mútua: se Pi está na seção crítica, nenhum outro processo pode entrar nela.
- Progresso: os processos participam na decisão sobre qual é o próximo processo a entrar na seção crítica.
- Espera limitada: se um processo deseja entrar na seção crítica, há um limite no num. de outros processos que podem entrar nela antes dele



# Hardware de Sincronização: TestAndSet

Para simplificar algoritmos, hardware pode prover operação atômica de leitura+escrita Exemplo: testa e modifica conteúdo da mem.

```
boolean TestAndSet(boolean* target)
{
   boolean old_value = *target;
   *target = true;
   return old_value;
}
```

# Hardware de Sincronização: TestAndSet

```
Variável compartilhada: boolean lock=false;
Processo i:
  do {
     while (TestAndSet(&lock));
    critical section
     lock = false;
  remainder section
  } while (1);
```



# Hardware de Sincronização: TestAndSet

Esses algoritmos garantem progresso, mas não espera limitada

Solução: cada processo registra sua intenção de entrar na região crítica separadamente

- boolean waiting[n]; // = false

Ao sair da região crítica, processo "passa a vez" diretamente ao próximo, se ele existir



### TestAndSet Com Espera Limitada

```
do {
  waiting[i] = true;
  while (waiting[i] && TestAndSet(&lock)); // Exclusao Mutua
  critical section
  j = next(i);
  while ((j!=i) && !waiting[j]) // procura o proximo
       j = next(j); // Espera limitada
  if (j==i) { lock = false; } // ninguém esperando
  else { waiting[j] = false; } // passa a vez diretamente
  waiting[i] = false; // Progresso
  remainder section
} while (1);
```



#### Semáforos

Variável inteira acessível apenas através de duas operações indivisíveis (atômicas):

```
wait(int* s) {
    while (*s<=0);
    (*s)--;
}</pre>
```

```
signal(int* s) {
    (*s)++;
}
```



#### Semáforos

## Estruturas de dados:

sem mutex = 1;

```
sem nbuffempty = TAM BUF;
  sem nbufffull = 0;
produtor:
do {
  // produz um item
  wait(nbuffempty);
  wait(mutex);
  // item → buffer
  signal(mutex);
  signal(nbufffull);
} while (1);
```

```
consumidor:
do {
   wait(nbufffull);
   wait(mutex);
   // item ← buffer
   signal(mutex);
   signal(nbuffempty);
   // processa item
} while (1);
```



# Transações Atômicas

Conceito originário da área de bancos de dados

Define um conjunto de operações que deve ser executado atomicamente:

- ° Ou todas operações executam, ou nenhuma
- ° P.ex.: saque de uma conta bancária

Operação normalmente baseada em logs

- ° Registra-se a operação que vai ser realizada
- ° Executa-se (ou não) a operação
- ° O log pode ser consultado no caso de falhas



#### Exercícios

Resolver a atividade prática de sincronização de processos que se encontra no site da disciplina.

