UNIVERSIDADE TECNOLÓGICA FEDERAL DO PARANÁ

Aula: Sincronização e Concorrência

Prof. Rodrigo Campiolo
Prof. Rogério A. Gonçalves[1]

[1]Universidade Tecnológica Federal do Paraná (UTFPR)
Departamento de Computação (DACOM)
Campo Mourão, Paraná, Brasil

Ciência de Computação

BCC34G - Sistemas Operacionais

Introdução

Programação Concorrente

- Consiste em um conjunto de processos sequenciais que executam concorrentemente.
- A concorrência implica em: disputa, execução simultânea e cooperação entre processos.
- A cooperação implica em compartilhar dados por meio de memória ou troca de mensagens.
- No entanto, acesso a dados compartilhados sem controle pode resultar em inconsistências ou comportamentos não esperados.

Problemas

Exemplo 1 - Produtor-Consumidor

- Há processos que produzem dados e armazenam em uma variável compartilhada.
- Há processos que consomem os dados armazenados na variável compartilhada.

Problemas

Exemplo 2 - Região com variáveis compartilhadas

- Considere o incremento de uma variável.

```
1   int x = 10;
2
3   // codigo compartilhado entre dois processos
4   x = x + 1;
```

- Execução concorrente dos processos P1 e P2:

Execução 1:

```
1  P1: MOV x, ACC
2  P1: INC ACC
3  P1: MOV ACC, x
4  % P1 escalonado
5  P2: MOV x, ACC
6  P2: INC ACC
7  P2: MOV ACC, x
8  % P2 escalonado
```

Execução 2:

```
1  P1: MOV x, ACC
2  P1: INC ACC
3  % P1 escalonado
4  P2: MOV x, ACC
5  P2: INC ACC
6  P2: MOV ACC, x
7  % P2 escalonado
8  P1: MOV ACC, x
```

Problema da Região Crítica

Região crítica

Trecho de código com alteração de dados compartilhados.

Condição de corrida (*race conditions*)

Resultado da execução de um trecho de código depende da ordenação de acesso as operações desse trecho.

Obs: Tradução alternativa *race condition* = condição de disputa

Problema da Região Crítica

Requisitos para a solução

- Exclusão mútua: dois ou mais processos não podem estar simultaneamente dentro da região crítica.
- Progressão: processos fora da região crítica não podem bloquear a entrada/execução de outro processo na região crítica.
- Espera limitada: processos não podem esperar infinitamente para entrar na região crítica.
- Processos não podem depender do número de processadores e de suas velocidades de execução como solução.

Mecanismos Básicos de Exclusão Mútua

Soluções em software puro

- Algoritmos de Dekker (1965) e de Petterson (1981).
- Executam na entrada e saída da Seção Crítica.
- Não há chamadas ao SO e interrupções especiais.
- Problemas: complexas, espera-ocupada (*busy-waiting*).

Soluções em Software Puro

```
                          flag[0]   = 0
                          flag[1]   = 0
                          turn      = 0

Processo P0:                          Processo P1:
      flag[0] = 1                             flag[1] = 1
      turn = 1                                turn = 0
      while( flag[1] && turn == 1 );          while( flag[0] && turn
                                              == 0 );
            // faz nada                               // faz nada

      Código-da-seção-crítica                 Código-da-seção-crítica

      flag[0] = 0                             flag[1] = 0
```

Figura 1: Solução de Petterson.

Mecanismos Básicos de Exclusão Mútua

Desabilitar interrupções

- Desabilita interrupções antes e ativa após o acesso à Seção Crítica.
- Solução simples para um processador.
- Usado em sistemas pequenos e dedicados (embarcados)
- Controle interno via SO.
- Problemas: pode diminuir a eficiência (interrupções pendentes), não adequado para máquinas paralelas.

Mecanismos Básicos de Exclusão Mútua

Spin-lock

- Baseada em instruções de máquina que realizam operações atômicas.
- A ideia central é proteger a Seção Crítica com uma variável (*lock*).
  - `lock = 0`: seção crítica livre.
  - `lock = 1`: seção crítica ocupada.
- Simplicidade de implementação.
- Recomendado para seções críticas pequenas.
- Desvantagens: espera ocupada (*busy-waiting*), possibilidade de postergação indefinida.

Mecanismos Básicos de Exclusão Mútua - Spin-Lock

- Solução em alto nível:

```
1 /* Entrada da Secao Critica */
2 while (lock == 1);        // se verdadeiro, SC ocupada
3 lock = 1;
4
5 // Secao Critica ...
6
7 /* Saida da Secao Critica */
8 lock = 0;
```

Mecanismos Básicos de Exclusão Mútua

Spin-lock

- A manipulação da variável `lock` (comparação/leitura/escrita) deve ocorrer de forma atômica.
- Duas instruções comuns para implementar são:
  - **test-and-set (TSL)**: Copia o valor de memória para um registrador e armazena um valor diferente de 0 na memória.
  - **swap (SWAP ou XCHG)**: Troca o valor de duas posições (memória, registrador) atomicamente.

Spin-Lock - TSL

```
1  entradaSC:
2    TSL RX, lock          % copia lock para RX, armazena valor diferente 0 em lock
3    CMP RX, 0             % compara RX com 0
4    JNE entradaSC         % se diferente, ir para entrada da SC
5    RET                   % retorna
6
7  saidaSC:
8    MOV lock, 0           % atribui 0 a lock
9    RET
```

Execução 1: `lock = 0`

```
RX = 0 e lock = 2  (linha 2)
RX == 0 ?          (linha 3)
Verdade            (linha 4)
entrando SC        (linha 5)

Processo na SC

lock = 0           (linha 8)
saindo SC          (linha 9)
```

Execução 2: `lock = 1`

```
RX = 1 e lock = 5  (linha 2)
RX == 0 ?          (linha 3)
Falso              (linha 4)
indo entradaSC     (linha 1)

Processo repete até que lock seja 0
```

Spin-Lock - XCHG

```
1  entradaSC:
2    MOV  RX, 1          % atribui 1 a RX
3    XCHG RX, lock       % troca conteudo RX e lock
4    CMP RX, 0           % compara RX com 0
5    JNE entradaSC       % se diferente, ir para entrada da SC
6    RET                 % retorna
7
8  saidaSC:
9    MOV lock, 0         % atribui 0 a lock
10   RET                 % retorna
```

Execução 1: `lock = 0`

```
RX = 1              (linha 2)
RX = 0 e lock = 1   (linha 3)
RX == 0 ?           (linha 4)
Verdade             (linha 5)
entrando SC         (linha 6)

Processo na SC

lock = 0            (linha 9)
saindo SC           (linha 10)
```

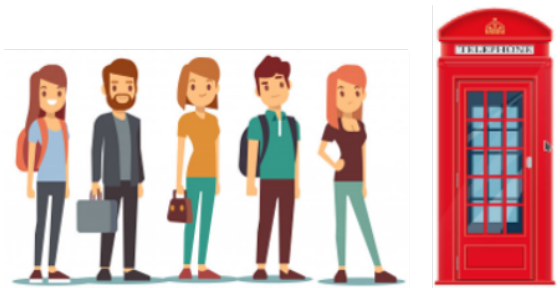
Execução 2: `lock = 1`

```
RX = 1              (linha 2)
RX = 1 e lock = 1   (linha 3)
RX == 0 ?           (linha 4)
Falso               (linha 5)
indo entradaSC      (linha 1)

Processo repete até que lock seja 0
```

Mecanismos de Exclusão Mútua - Mutex

Mutex

- Tipo abstrato de dados: **valor lógico** e **fila de processos**.
- A variável do tipo Mutex pode assumir: **livre** e **ocupado**.
- Suporta duas operações:
  - **lock**: solicitar entrada na seção crítica.
  - **unlock**: liberar seção crítica (indicar a saída).
- As operações **lock** e **unlock** devem ser atômicas.
- Como implementá-las?



Mecanismos de Exclusão Mútua - Mutex

```
1 lock(mutex):
2     if mutex == LIVRE:
3         mutex = OCUPADO
4     else:
5         bloqueia processo e insere na fila
6
7 unlock(mutex):
8     if fila vazia:
9         mutex = LIVRE
10    else:
11        libera processo no inicio da fila
```

Figura 2: Pseudocódigo para as operações lock e unlock.

```
1 Mutex mutex = LIVRE
2
3 lock(mutex)
4   ## codigo da Secao Critica ##
5 unlock(mutex)
```

Figura 3: Protegendo a seção crítica com o Mutex.

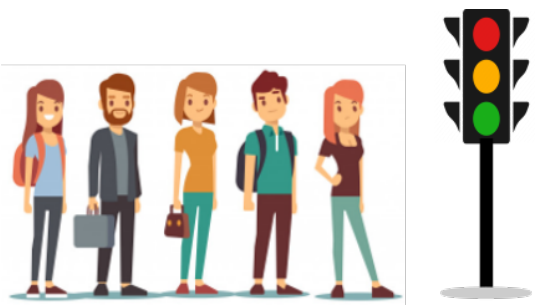
Mecanismos de Exclusão Mútua - Mutex

```c
pthread_t t[N];             // define N threads
pthread_mutex_t mutex;      // variavel mutex
int counter;                // variavel compartilhada

void* task(void *arg)
{
    pthread_mutex_lock(&mutex);    // inicio SC
    counter += 1;
    pthread_mutex_unlock(&mutex); // saida SC
}

int main(void)
{
    /* inicializa mutex - inicializado como LIVRE */
    pthread_mutex_init(&mutex, NULL);

    /* inicializa threads */ ...
    /* aguarda todas as threads finalizarem */ ...

    /* destroi mutex */
    pthread_mutex_destroy(&mutex);
    ...
}
```

Figura 4: Código em C com pthreads usando mutex.

Semáforos

- Tipo abstrato de dados: **valor inteiro** e **fila de processos**.
- Proposto por Dijkstra (1965).
- Suporta duas operações:
  - **P()**: decrementa valor inteiro, bloqueia processo se valor for negativo (insere fila).
  - **V()**: incrementa valor inteiro, libera processo no início da fila (se existir).
- **P()** - *proberen* (testar/tentar) e **V()** - *verhogen* (incrementar).
- As operações **P()** e **V()** devem ser atômicas.



Mecanismos de Exclusão Mútua - Semáforos

```
1 P(S):
2     S.valor = S.valor - 1
3     if S.valor < 0:          # valor negativo
4         bloqueia processo e insere na fila
5
6 V(S):
7     S.valor = S.valor + 1
8     if len(S.fila) > 0:      # fila nao vazia
9         libera processo no inicio da fila
```

Figura 5: Pseudocódigo para as operações P() e V().

```
1 Semaforo s
2 s.init_sem(1)  #inicializa com 1
3 ...
4
5 P(s)
6   ## codigo da Secao Critica ##
7 V(s)
```

Figura 6: Usando semáforo para proteção de seção crítica.

Mecanismos de Exclusão Mútua - Semáforos

Considerações e usos de semáforos

- Semáforos podem ser usados também para sincronização/sinalização entre processos.
- Exemplos: relações de precedência de execução, execução após um evento específico, barreiras.
- Em geral:
  - para exclusão mútua: semáforo inicializado com valor 1.
  - para sinalização: semáforo inicializado com valor 0.
- Mutex: mesmo processo executa o `lock()` e `unlock()`.
- Semáforo: processos distintos podem realizar o `P()` e `V()`.

Usando semáforos para sinalização

```
1  Semaforo s
2  s.init_sem (0)
3  ...
4  # compartilhar semaforo com as threads
```

Thread 1

```
1  # codigo da Thread 1
2  ...
3  ...
4  # thread aguardando Thread 2
5  P(s)
6  ... # codigo executa somente apos
       sinalizacao
```

Thread 2

```
1  # codigo da Thread 2
2  ...
3  ...
4  # sinaliza para Thread 1
5  V(s)
6  ...
7  # continua a execucao
```

Mecanismos de Exclusão Mútua - Monitores

Monitores

- Tipo abstrato de dados em nível de linguagem.
- Proposto Per Brinch Hansen (1973) e Hoare (1974).
- Gerenciar variáveis compartilhadas dentro do Monitor, isto é, encapsular variáveis e acesso somente via métodos.
- As operações do Monitor proveem exclusão mútua.
- Uso de **variáveis de condição** e operações **wait()** e **signal()**.
- **variável de condição**: associada a uma fila de processos bloqueados.
- **wait()**: bloqueia o processo e adiciona na fila.
- **signal()**: acorda o primeiro processo na fila.

Obs: Comentar sobre a operação *broadcast* (*notifyAll*).

Mecanismos de Exclusão Mútua - Monitores

Variáveis de condição: a e b
Filas: a.q, b.q, e
Operações: wait e notify (signal)

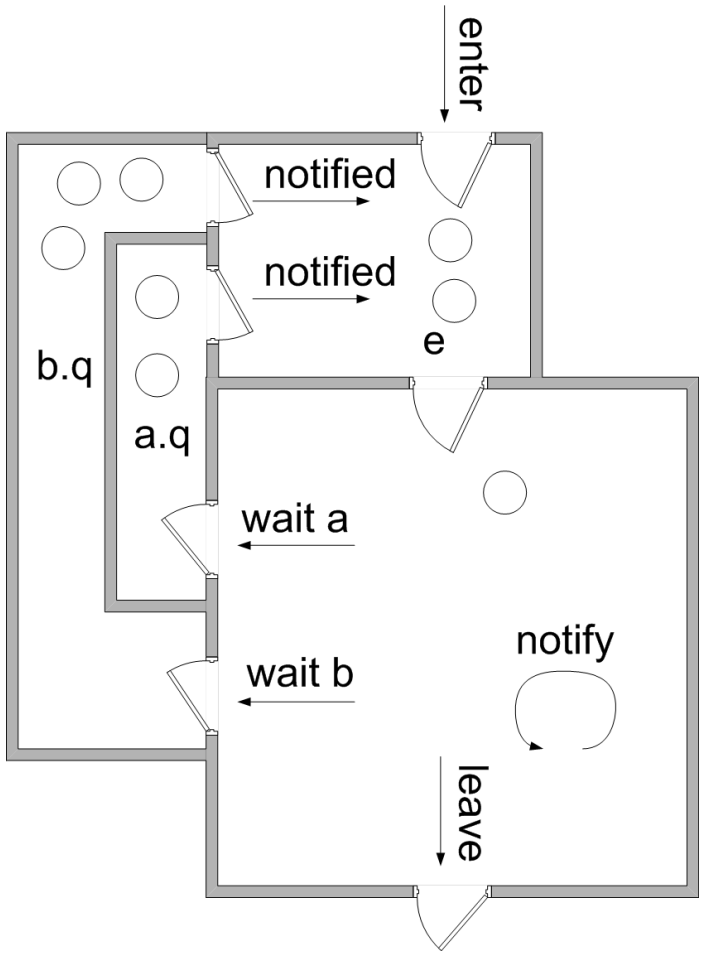


Figura 7: Ilustração de um monitor (Fonte: Wikipedia, 2019).

```java
public class ResourceMonitor {
    private String buffer;
    private boolean empty = true;

    public synchronized String take() {
        while (empty) {     // aguarda se buffer vazio
            try {
                wait();
            } catch (InterruptedException e) {}
        }
        /* remove valor de buffer */
        empty = true;
        notifyAll();        // notifica os produtores
        return buffer;
    }

    public synchronized void put(String buffer) {
        while (!empty) {    // aguarda se buffer cheio
            try {
                wait();
            } catch (InterruptedException e) {}
        }
        /* armazena valor em buffer */
        empty = false;
        this.buffer = buffer;
        notifyAll();        // notifica os consumidores
    }
}
```

Figura 8: Exemplo de monitor em Java para Problema do Produtor/Consumidor.

Problemas Clássicos

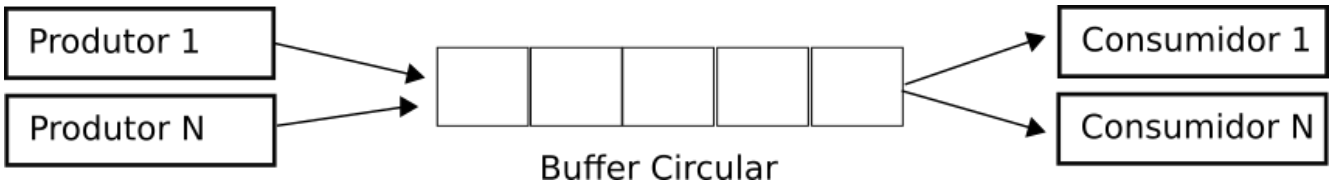
Problemas

- Produtores/Consumidores.
- Leitores/Escritores.
- Jantar dos Filósofos.
- Barbeiro Dorminhoco.

Problemas Clássicos

Produtores/Consumidores

- Dois tipos de processos (threads):
  - Produtores: produzem e armazenam itens em um buffer.
  - Consumidores: retiram e consumem itens de um buffer.
- A variável buffer é compartilhada.
- Restrições:
  - Produtores devem aguardar se buffer cheio.
  - Consumidores devem aguardar se buffer vazio.
  - Exclusão mútua para manipulação do buffer.



Problemas Clássicos

Leitores/Escritores

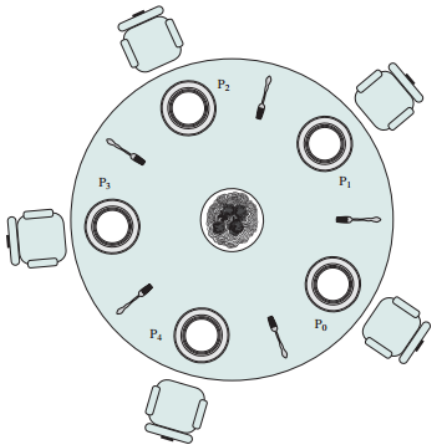
- Dois tipos de processos (threads):
  - Leitores: Realizam a leitura de um recurso compartilhado.
  - Escritores: Realizam a escrita em um recurso compartilhado.
- Restrições:
  - Os leitores podem acessar o recurso simultaneamente.
  - Os escritores devem acessar o recurso exclusivamente.

X

Problemas Clássicos

Jantar dos Filósofos

- Os filósofos (processos) estão em uma mesa circular.
- Os filósofos pensam, sentem fome e comem.
- Mesa: 5 pratos de macarrão e um garfo (recurso) entre cada prato.
- Para comer, os filósofos precisam de dois garfos.
- Ao sentir fome, os filósofos pegam os garfos e, se conseguirem, comem.
- Ao terminar de comer, os filósofos devolvem os dois garfos.



Problemas Clássicos

Barbeiro Dorminhoco

- Barbearia com 1 cadeira para atender e N para esperar.
- O barbeiro (processo) está atendendo ou dormindo (não há clientes).
- Os clientes (processos) chegam na barbearia e, se o barbeiro está dormindo, o acordam. Se está atendendo, os clientes esperam em uma das cadeiras (se disponível), caso contrário, vão embora.
- O barbeiro volta a dormir quando não há mais clientes.

Leitura e Atividades

Leitura recomendada

1. Comunicação entre processos (Seção 2.3). Sistemas Operacionais Modernos, Tanenbaum and Bos (2016).
2. Coordenação entre tarefas (Capítulo 10). Sistemas Operacionais: Conceitos e Mecanismos, Maziero (2019).
3. Mecanismos de coordenação (Capítulo 11). Sistemas Operacionais: Conceitos e Mecanismos, Maziero (2019).
4. Problemas clássicos (Capítulo 12). Sistemas Operacionais: Conceitos e Mecanismos, Maziero (2019).

Atividades

1. Resolver a lista de exercícios *L05 - Sincronização e Concorrência*, disponibizada na plataforma Moodle.
2. Implementar soluções usando semáforos e monitores para os problemas clássicos de concorrência.

Referências I

Maziero, C. A. (2019). *Sistemas operacionais: conceitos e mecanismos*. online. Disponível em http://wiki.inf.ufpr.br/maziero/lib/exe/fetch.php?media=so:so-livro.pdf.

Tanenbaum, A. S. and Bos, H. (2016). *Sistemas operacionais modernos*. Pearson Education do Brasil, 4 edition.