Sistemas Operacionais

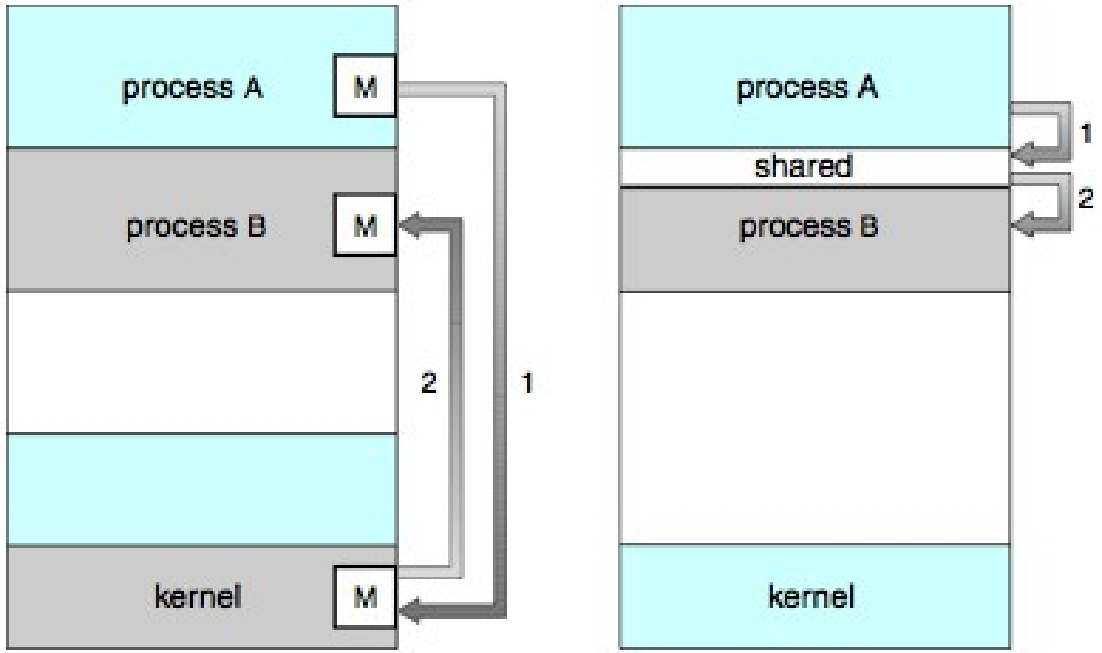
Comunicação entre processos

Comunicação entre Processos

- Os SO's hoje são multi-tarefa
- Alguns processos precisam cooperar para cumprir seus objetivos
- Sugestões de implementação?
- Exemplos
  - Um aplicativo imprimindo um documento

Comunicação entre Processos

- Duas formas:
  - Memória compartilhada
  - Passagem de mensagem



Passagem de mensagens

- Sistema de mensagem
  - Processos se comunicam entre si sem lançar mão de variáveis compartilhadas
- A comunicação utiliza 2 funções
  - send(destino, mensagem)
  - receive(origem, mensagem)
- Se P e Q quiserem se comunicar, eles precisam:
  - estabelecer um link de comunicação entre eles
  - trocar mensagens por meio de *send/receive*

Problema do Produtor Consumidor

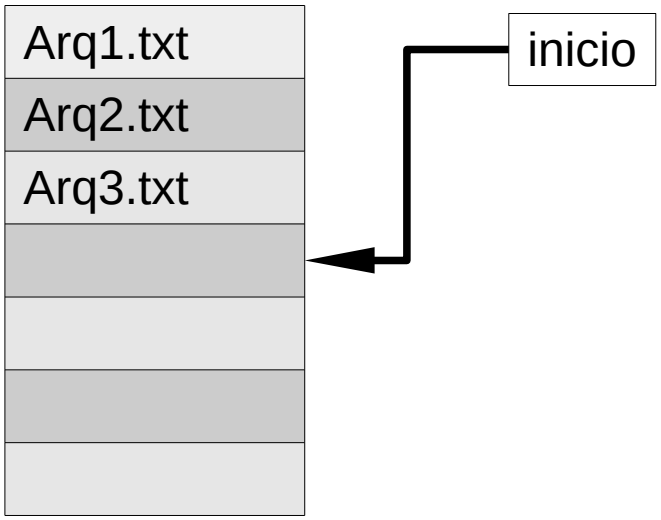
- Paradigma para processos em cooperação, processo produtor produz informações que são consumidas por um processo consumidor
  - Buffer ilimitado não impõe limite prático sobre o tamanho do buffer
  - Buffer limitado assume que existe um tamanho de buffer fixo
- Este problema é recorrente no mundo da computação
  - Lembrem do exemplo do aplicativo imprimindo

Race Condition (Condição de Corrida)

- Condição em que dois processos lêem e escrevem um dado compartilhado e o resultado final depende da ordem em que os processos são executados.
- Lembre-se que em um sistema multitarefa um processo pode ser retirado da *execução* para *pronto* a qualquer momento.
- Essas condições são muito difíceis de debuggar, encontrar falhas, que podem ocorrer dependendo de fatores externos

Condição de corrida

```
ler(inicio)                    | ler(inicio)
pilha[inicio] = arquivo        | pilha[inicio] = arquivo
incrementa(inicio)             | incrementa(inicio)
gravaNovoInicio()              | gravaNovoInicio()
```



Região Crítica

- Uma solução para o problema da condição de corrida é **evitar que mais de um procecesso leia/escreva ao mesmo tempo.**
- Essa estratégia é chamada **exclusão mútua** (*mutual exclusion*) *mutex*
- É dever do programador, identificar estas situações e proteger seu código nestas situações

Região crítica

- Para que a solução seja implementada pelo S.O. 4 condições são necessárias:
  1. Dois processos não podem estar em sua região crítica simultaneamente
  2. A solução deve funcionar para qualquer número de CPUs e qualquer velocidade
  3. Nenhum processo fora da região crítica pode bloquear outros processos
  4. Nenhum processo deve esperar eternamente para entrar em sua região crítica

Algumas soluções possíveis

- Desabilitar interrupções
- Variáveis de trava(lock)
- Chaveamento origratório
- Solução de Peterson
- Dormir e acordar
- Semáforos
- Mutex
- Monitores

Desabilitar interrupções

- Ao entrar na região crítica ficam desabilitadas as interrupções
- Evita que ocorra troca de contexto por tempo.
- Em um sistema multiprocessado pode ser complexo ou impossível desabilitar a interrupção em todos os processadores
- Entregar o controle das interrupções ao usuário pode ser perigoso se o processo não habilitar novamente estas interrupções.

Variáveis de trava(lock)

- Uma única variável *lock* indica quando há algum processo em uma região crítica
- Se **`lock == 0`** não há processo na região crítica.
- Se **`lock == 1`** há processo na região crítica.
- A ideia é: se o seu processo quer entrar na região crítica ele deve ler *lock* e só poderá entrar quando **`lock == 0`**
- Quando ele entrar deve fazer **`lock = 1`**

Lock com espera ocupada

```
//enquanto o lock está ocupado
while (obter(lock)==1){
    //faz nada (espera ocupada)
}
lock=1;
ler(inicio);
pilha[inicio] = arquivo;
incrementa(inicio);
gravaNovoInicio();
lock=0; //libera o lock
```

Chaveamento obrigratório

- Alterna entre dois processos
- A variável turn indica qual processo tem direito a entrar na região crítica
- Viola a condição que um processo em região não crítica bloqueia um processo

```
while (TRUE) {
        while (turn !=0)            /* laço */ ;
        critical_region( );
        turn = 1;
        noncritical_region( );
}
```

```
while (TRUE) {
        while (turn !=1)            /* laço */ ;
        critical_region( );
        turn = 0;
        noncritical_region( );
}
```

Solução de Peterson

- Define duas funções *enter_region* e *leave_region*
- Usa um vetor de interessados em entrar na região crítica
- Usa uma variável *turn* para definir quem é a vez para usar a região crítica

```
#define FALSE  0
#define TRUE   1
#define N      2                        /* número de processos */

int turn;                               /* de quem é a vez? */
int interested[N];                      /* todos os valores 0 (FALSE) */

void enter_region(int process);         /* processo é 0 ou 1 */
{
    int other;                          /* número de outro processo */

    other = 1 – process;                /* o oposto do processo */
    interested[process] = TRUE;         /* mostra que você está interessado */
    turn = process;                     /* altera o valor de turn */
    while (turn == process && interested[other] == TRUE) /* comando nulo */ ;
}

void leave_region(int process)          /* processo: quem está saindo */
{
    interested[process] = FALSE;        /* indica a saída da região crítica */
}
```

Test and Set

- Utiliza uma instrução especial do processador
- Assembly **`TSL RX, LOCK`**

```
enter_region:
        TSL REGISTER,LOCK       | copia lock para o registrador e põe lock em 1
        CMP REGISTER,#0         | lock valia zero?
        JNE enter_region        | se fosse diferente de zero, lock estaria ligado, portanto, continue no laço de repetição
        RET                     | retorna a quem chamou; entrou na região crítica


leave_region:
        MOVE LOCK,#0            | coloque 0 em lock
        RET                     | retorna a quem chamou
```

Dormir e acordar

- As soluções anteriores são corretas mas empregam a “espera ocupada” também chamada “espera ociosa”
- Se um processo não consegue entrar na região crítica ele fica preso em um loop consumindo tempo de CPU
- Pode gerar um problema de inversão de prioridades
- Para resolver este problema podemos usar chamadas do SO **`sleep`** e **`wakeup`**

sleep() e wakeup()

- A diferença mais importante desta estratégia para as outras é que estas permitem que o S.O. bloqueie o processo em sleep
- Assim o processo não consumirá CPU enquanto espera para entrar em sua região crítica

Problema do produtor consumidor

```
#define N 100
int count = 0;

void producer(void)
{
        int item;

        while (TRUE) {
                item = produce_item( );
                if (count == N) sleep( );
                insert_item(item);
                count = count + 1;
                if (count == 1) wakeup(consumer);
        }
}
```

```
void consumer(void)
{
        int item;

        while (TRUE) {
                if (count == 0) sleep( );
                item = remove_item( );
                count = count – 1;
                if (count == N – 1) wakeup(producer);
                consume_item(item);
        }
}
```

Semáforos

- Um semáforo é uma estratégia que permite que 2 ou mais processos se alternem no uso de um recurso compartilhado
- Corrigindo o problema do sinal wakeup perdido, os semáforos tem um contador interno para os sleep e wakeup.
- Para manter a clareza nos semáforos as instruções são chamadas `down` (equivalente ao `sleep`) e `up` (equivalente ao `wakeup`).

Produtor consumidor usando semáforos

```
#define N 100
typedef int semaphore;
semaphore mutex = 1;
semaphore empty = N;
semaphore full = 0;

void producer(void)
{
        int item;

        while (TRUE) {
                item = produce_item( );
                down(&empty);
                down(&mutex);
                insert_item(item);
                up(&mutex);
                up(&full);
        }
}
```

```
void consumer(void)
{
        int item;

        while (TRUE) {
                down(&full);
                down(&mutex);
                item = remove_item( );
                up(&mutex);
                up(&empty);
                consume_item(item);
        }
}
```

Mutex

- Mutex são tipos especiais de semáforos que tem apenas uma variável binária, permitindo que apenas 1 processo acesse a região crítica.
- É equivalente a um semáforo iniciado em 1.
- A implementação é mais simples e mais rápida.
- Muito usado para garantir exclusão mútua a uma variável compartilhada.

Monitores

- A última estratégia para problema da condição de corrida (*race condition*) são os monitores.
- Um monitor é um artifício de algumas linguagens de programação
- Permite que se defina um conjunto de métodos e propriedades dentro de um pacote ou módulo.
- O importante aqui é que ***Somente 1 processo pode estar ativo em um monitor por vez.***

Monitores

```
monitor ProducerConsumer
    condition full, empty;
    integer count,

    procedure insert(item:integer);
    begin
        if count= N then wait (full);
        insert_item(item);
        count:= count + 1;
        if count= 1 then signal (empty)
    end ;

    function remove:integer;
    begin
        if count= 0 then wait (empty);
        remove =remove_item;
        count:= count– 1;
        if count= N – 1 then signal (full)
    end;

    count:= 0;
end monitor;
```

```
procedure producer;
begin
    while true do
    begin
        item = produce_item;
        ProducerConsumer.insert(item)
    end
end ;

procedure consumer;
begin
    while true do
    begin
        item = ProducerConsumer.remove;
        consume_item(item)
    end
end ;
```

Atividade

- Descreva como dois processos podem se comunicar, usando memória compartilhada.
- O que é um condição de corrida?
- Descreva uma situação em que pode haver uma condição de corrida.
- O que é uma região crítica de um processo? Por que ela precisa ser protegida de execução simultânea?
- O que é espera ocupada?
- Descreva as seguintes estratégias de comunicação/sincronização entre processos.
  - Monitores
  - Semáforos
  - Mutex
  - Test Set and Lock (variável de lock)

Endereço de entrega: https://goo.gl/tLT4Gh