Curso de Engenharia de Computação
Sistemas Operacionais

Processos e Threads – Parte IV

Slides da disciplina Sistemas Operacionais
Curso de Engenharia de Computação
Instituto Mauá de Tecnologia – Escola de Engenharia Mauá
Prof. Marco Antonio Furlan de Souza

Comunicação entre processos

INSTITUTO MAUÁ DE TECNOLOGIA

- **Semáforos**
  - **Semáforo** é uma **técnica** que utiliza uma **variável inteira** e **duas operações**, `up` e `down`, para **implementar exclusão mutual** de **processos**;
  - O **valor numérico** da **variável** de **semáforo** representa o **número** de **vezes** que os **processos** foram **suspensos** (dormir);
  - A **operação** `down` **verifica se** o **valor** da **variável** em **questão** é **maior** que **zero**. **Se for**, então o **variável** é **decrementada** e `down` **continua** sua execução; **senão**, o **processo** é **suspenso** (`sleep`) e a **operação** `down` **não continua** sua execução;
  - É **importante garantir** que a **alteração** do **valor** e a **execução** de `sleep` sejam **realizados** de forma **atômica** (como uma unidade).

Comunicação entre processos

INSTITUTO MAUÁ DE TECNOLOGIA

- **Semáforos**
  - A **operação** `up` **incrementa** o **valor** do **semáforo** em questão e, **se um** ou **mais processos** estavam **dormindo** por causa deste semáforo, **um deles** é **escolhido** (**aleatoriamente**) e **então** ele **poderá completar** sua **operação** `down`;
  - As **operações** de **incrementar** o **semáforo** e então **acordar** um **processo** também **deverão** ser **executadas** de forma **atômica**;
  - **Vantagem** do uso de semáforos
    - **Resolve** o **problema** do `wakeup` **perdido** da **técnica** original de **Acordar-Dormir**.

Comunicação entre processos

INSTITUTO MAUÁ DE TECNOLOGIA

- **Semáforos**
  - **Problema do Produtor-Consumidor com Semáforos**
    - **Solução**

```
#define N 100                                   /* number of slots in the buffer */
typedef int semaphore;                          /* semaphores are a special kind of int */
semaphore mutex = 1;                            /* controls access to critical region */
semaphore empty = N;                            /* counts empty buffer slots */
semaphore full = 0;                             /* counts full buffer slots */

void producer(void)
{
        int item;

        while (TRUE) {                          /* TRUE is the constant 1 */
                item = produce_item( );         /* generate something to put in buffer */
                down(&empty);                   /* decrement empty count */
                down(&mutex);                   /* enter critical region */
                insert_item(item);              /* put new item in buffer */
                up(&mutex);                     /* leave critical region */
                up(&full);                      /* increment count of full slots */
        }
}
```

Comunicação entre processos

INSTITUTO MAUÁ DE TECNOLOGIA

- **Semáforos**
  - **Problema do Produtor-Consumidor com Semáforos**
    - **Solução** (cont.)

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

        while (TRUE) {                          /* infinite loop */
                down(&full);                    /* decrement full count */
                down(&mutex);                   /* enter critical region */
                item = remove_item( );          /* take item from buffer */
                up(&mutex);                     /* leave critical region */
                up(&empty);                     /* increment count of empty slots */
                consume_item(item);             /* do something with the item */
        }
}
```

- **Semáforos**
  - **Problema do Produtor-Consumidor com Semáforos**
    - Esta solução utiliza **três semáforos**:
      - `full`: para contar o número de entradas que estão cheias;
      - `empty`: para contar o número de entradas que estão vazias;
      - `mutex`: para evitar que o consumidor e produtos não acessem o buffer ao mesmo tempo.
    - Os semáforos `full` e `empty` são utilizados para **sincronização**: eles garantem que o **produtor para** de **executar** quando o **buffer** estiver **cheio** e que o **consumidor para** de **executar** quando o **buffer** estiver **vazio**;
    - Semáforos podem ser **binários** (`mutex`) ou não (`full`, `empty`).

Comunicação entre processos

INSTITUTO MAUÁ DE TECNOLOGIA

- **Mutexes**
  - É uma **versão simplificada** de um **semáforo**;
  - Trata-se de uma **variável compartilhada** que pode estar em um dos seguintes **estados**:
    - Destravada (*unlocked*);
    - Travada (*locked*).
  - O **número zero** é utilizado para um **mutex destravado** e qualquer valor **diferente de zero** é utilizado para um **mutex travado**.

Comunicação entre processos

INSTITUTO MAUÁ DE TECNOLOGIA

- **Mutexes**
  - **Procedimentos utilizados com mutexes**
    - O **procedimento** `mutex_lock()` é **executado** por uma thread ou processo que precisa **acessar** uma **região crítica**.
    - Se o **mutex** estava **destravado** (região crítica disponível), esta **chamada funciona** e a **thread entra** na **região crítica**;
    - Se o **mutex** estava **travado**, a **thread** fica **bloqueada** até que a **região crítica** seja **liberada** por **outra thead**, que deverá **invocar** `mutex_unlock();`
    - Se **várias threads** estão **bloqueadas** pelo **mutex**, **uma** delas é **escolhida aleatoriamente** e poderá **adquirir** a **trava**.

Comunicação entre processos

INSTITUTO MAUÁ DE TECNOLOGIA

- **Mutexes**
  - **Procedimentos utilizados com mutexes**
    - Mutexes **podem ser implementados fora** do **kernel**, por meio de **instruções TSL** que permitem **acesso** ao **mutex** de forma **exclusiva**;
    - Nessa **implementação** no **espaço** de usuário, é **importante** que se **execute** alguma **função** que permita a **seleção** de outra **thread** que esteja **bloqueada** de modo a **evitar** a **espera ocupada**:

```
mutex_lock:
        TSL REGISTER,MUTEX      | copy mutex to register and set mutex to 1
        CMP REGISTER,#0         | was mutex zero?
        JZE ok                  | if it was zero, mutex was unlocked, so return
        CALL thread_yield       | mutex is busy; schedule another thread
        JMP mutex_lock          | try again
ok:     RET                     | return to caller; critical region entered

mutex_unlock:
        MOVE MUTEX,#0           | store a 0 in mutex
        RET                     | return to caller
```

Comunicação entre processos

INSTITUTO MAUÁ DE TECNOLOGIA

- **Monitores**
  - Ao se utilizar **semáforos** e **mutexes** deve-se tomar **cuidado** na **ordem** das **requisições** de **travamento** e **destravamento** das **variáveis** de **travamento** pois pode-se **incorrer** a *deadlocks* (situações onde nenhuma das threads consegue retomar o recurso);
  - **Monitor** é uma **primitiva** de **programação** – deve fazer **parte** da **sintaxe** da **linguagem** – e que **permite** que uma **única thread** esteja **ativa** na **área protegida** pelo **monitor** por vez;
  - A **linguagem** de **programação Java implementa** uma **forma** modificada de **monitor** por meio da **palavra reservada** `synchronized`, que pode ser **aplicada** a uma **declaração** de **método**;
  - Assim, o **método** fica **protegido** da **execução concorrente** do método por uma ou mais threads – **sem** a palavra `synchronized`, **não** há **garantias** sobre a **execução intercalada** do **método** por mais de uma thread.

Comunicação entre processos

INSTITUTO MAUÁ DE TECNOLOGIA

- **Passagem de mensagem**
  - Este **método** de **comunicação** entre **processos** utiliza **duas primitivas**, `send` e `receive`, e que devem ser implementados como chamadas de sistema;
  - Normalmente essas primitivas são **implementadas** como **funções** que necessitam de **dois parâmetros**: para/de quem a mensagem se refere e a mensagem propriamente dita:
    - `send(destination, &message);`
    - `receive(source, &message);`

Comunicação entre processos

INSTITUTO MAUÁ DE TECNOLOGIA

- **Passagem de mensagem**
  - Este método adiciona um **novo problema**: **perda** de **dados** na **transmissão** – é necessário **adicionar** um **mecanismo** de **reconhecimento de recebimento** (*acknowledgement message*) para o **receptor avisar** ao **remetente** que **recebeu** a **mensagem**;
  - Além disso, surgem **problemas** de **autenticação** – como um **cliente** pode **saber** se ele está se **comunicando** com um **servidor real** e não um impostor?

Comunicação entre processos

INSTITUTO MAUÁ DE TECNOLOGIA

- **Passagem de mensagem**
  - **Solução do problema do Produtor-Consumidor com passagem de mensagem**
    - **Assumir** que todas as **mensagens** tenham o **mesmo tamanho** e que **mensagens enviadas** mas ainda **não recebidas** são automaticamente **buferizadas** pelo **sistema operacional**;
    - Na solução `N` **mensagens** são **utilizadas** em um **buffer compartilhado** da memória;
    - O **consumidor inicia enviando** `N` **mensagens** vazias ao produtor;
    - **Sempre** que o **produtor possui** um **item para enviar**, ele pega uma **mensagem vazia** do **consumidor** e **envia** de **volta** uma **mensagem cheia**;
    - **Desta forma** o **número** de **mensagens** no sistema fica **constante.**

Comunicação entre processos

INSTITUTO MAUÁ DE TECNOLOGIA

- **Passagem de mensagem**
  - **Solução do problema do Produtor-Consumidor com passagem de mensagem**

```
#define N 100                                   /* number of slots in the buffer */

void producer(void)
{
        int item;
        message m;                              /* message buffer */

        while (TRUE) {
                item = produce_item( );         /* generate something to put in buffer */
                receive(consumer, &m);          /* wait for an empty to arrive */
                build_message(&m, item);        /* construct a message to send */
                send(consumer, &m);             /* send item to consumer */
        }
}
```

Comunicação entre processos

INSTITUTO MAUÁ DE TECNOLOGIA

- **Passagem de mensagem**
  - **Solução do problema do Produtor-Consumidor com passagem de mensagem**

```
void consumer(void)
{
        int item, i;
        message m;

        for (i = 0; i < N; i++) send(producer, &m);  /* send N empties */
        while (TRUE) {
                receive(producer, &m);              /* get message containing item */
                item = extract_item(&m);            /* extract item from message */
                send(producer, &m);                 /* send back empty reply */
                consume_item(item);                 /* do something with the item */
        }
}
```

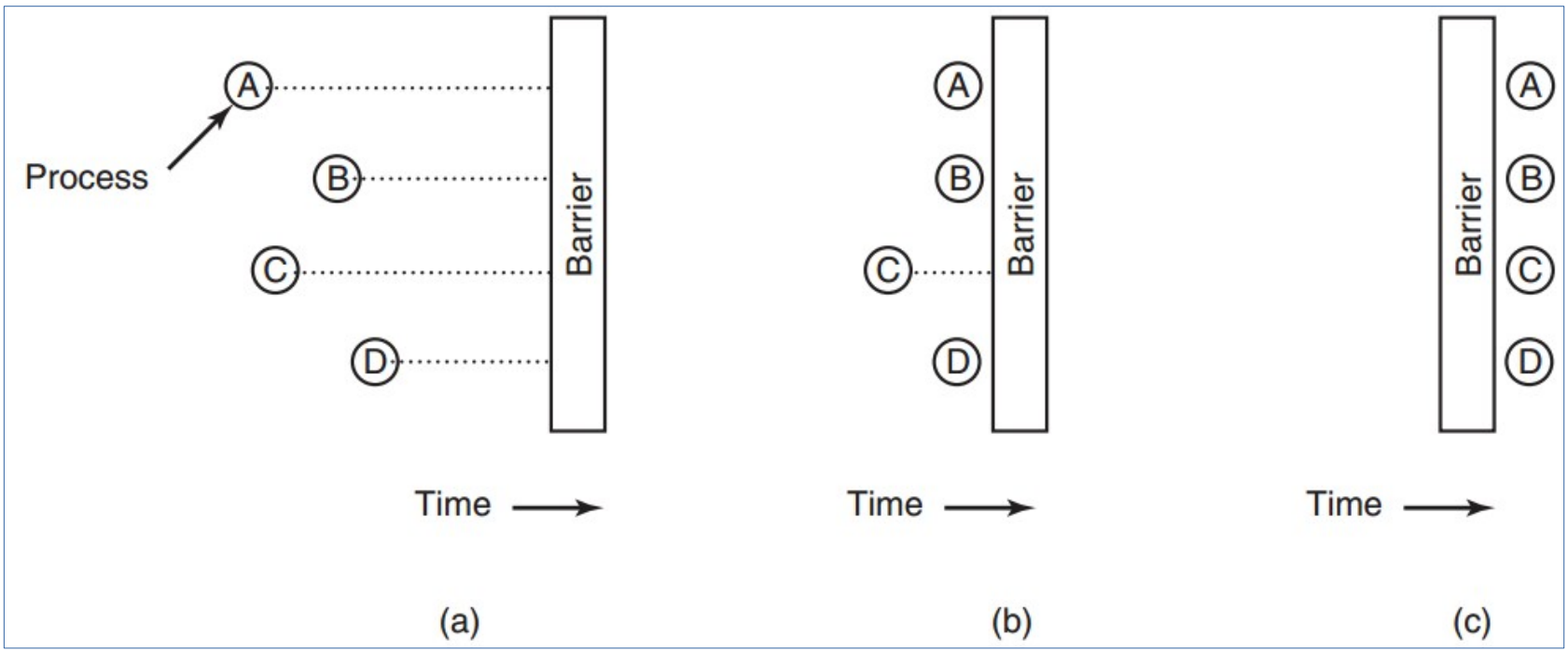
Comunicação entre processos

INSTITUTO MAUÁ DE TECNOLOGIA

- **Barreiras**
  - Este mecanismo de sincronização é **destinado** a **grupos** de **processos**;
  - **Algumas aplicações** são **divididas** em **fases** e têm a **regra** de que **nenhum processo** pode **prosseguir** para a **próxima fase** até que **todos** os **processos** estejam **prontos** para prosseguir para a **próxima fase**;
  - Esse **comportamento** pode ser **conseguido** colocando uma **barreira** no **final** de cada **fase**. Quando um **processo atinge** a **barreira**, fica **bloqueado** até que todos os **processos** tenham **atingido** a barreira;
  - Isso permite grupos de processos para sincronizar.

Comunicação entre processos

- **Barreiras**
  - **Exemplo de funcionamento**



Comunicação entre processos

INSTITUTO MAUÁ DE TECNOLOGIA

- **Read-Copy-Update**
  - A técnica *Read-Copy-Update* (**RCU**) foi desenvolvida para **permitir** que, em alguns casos, **dois processos** distintos possam **acessar dados compartilhados lendo** e **alterando valores simultaneamente**;
  - A **ideia** é **garantir** que o **processo leitor** ou **leia** uma **versão antiga** dos dados **ou** leia uma **versão nova** dos dados, **nunca** uma **mistura** dos **dois**.

Referências bibliográficas

TANENBAUM, Andrew S. **Sistemas operacionais modernos**. 3. ed. São Paulo: Pearson, 2013. 653 p.