Curso de Engenharia de Computação
Sistemas Operacionais

Processos e Threads – Parte III

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
MAUÁ

- **Conceitos**
  - **Comunicação** entre **processos** em execução é **comum**;
  - Por **exemplo**, na **linha** de **comando** do **Linux** quando se utiliza **pipes** para **redirecionar** a **saída** de um **processo** à **entrada** de **outro**;
  - **Problemas** que surgem na comunicação entre processos:
    - Como **efetuar** a **comunicação**?
    - **Como garantir que** dois ou mais **processos** possam **operar** sobre **recursos** em **comum sem** gerar **inconsistências** (exemplos: conta bancária, reservas aéreas):
    - Como **sincronizar** as **operações** entre dois processos?

Comunicação entre processos

INSTITUTO MAUÁ DE TECNOLOGIA

- **Condições de disputa**
  - **Processos** (e threads!) podem **compartilhar armazenamento** em comum (**memória**, **arquivo** em **disco**) e realizar nesse armazenamento **operações** de **leitura** e **escrita**;
  - Quando **dois ou mais processos** tentam **ler** ou **alterar** algum **recurso compartilhado** e o **resultado final depende** da **ordem** de como os **processos realizaram** tais **operações**, tem-se uma **condição de disputa** (*race condition*).

Comunicação entre processos

INSTITUTO MAUÁ DE TECNOLOGIA

- **Condições de disputa**
  - **Exemplo**
    - Os **processos** `p` e `q` ao serem executados em paralelo **acessam** a **variável compartilhada** `n`:

Algorithm 2.4: Assignment statements with one global reference	
integer n ← 0	
p	**q**
integer temp	integer temp
p1: temp ← n	q1: temp ← n
p2: n ← temp + 1	q2: n ← temp + 1

Dois cenários de execução!
Qual deles é o correto?

Process p	Process q	n	p.temp	q.temp
p1: temp←n	q1: temp←n	0	?	?
p2: n←temp+1	q1: temp←n	0	0	?
(end)	**q1: temp←n**	1		?
(end)	**q2: n←temp+1**	1		1
(end)	(end)	2		

Process p	Process q	n	p.temp	q.temp
p1: temp←n	q1: temp←n	0	?	?
p2: n←temp+1	**q1: temp←n**	0	0	?
p2: n←temp+1	q2: n←temp+1	0	0	0
(end)	**q2: n←temp+1**	1		0
(end)	(end)	1		

De: BEN-ARI, M. Principles of Concurrent and Distributed Programming. Harlow, England. Pearson Education Limited, 2015.

Comunicação entre processos

INSTITUTO MAUÁ DE TECNOLOGIA

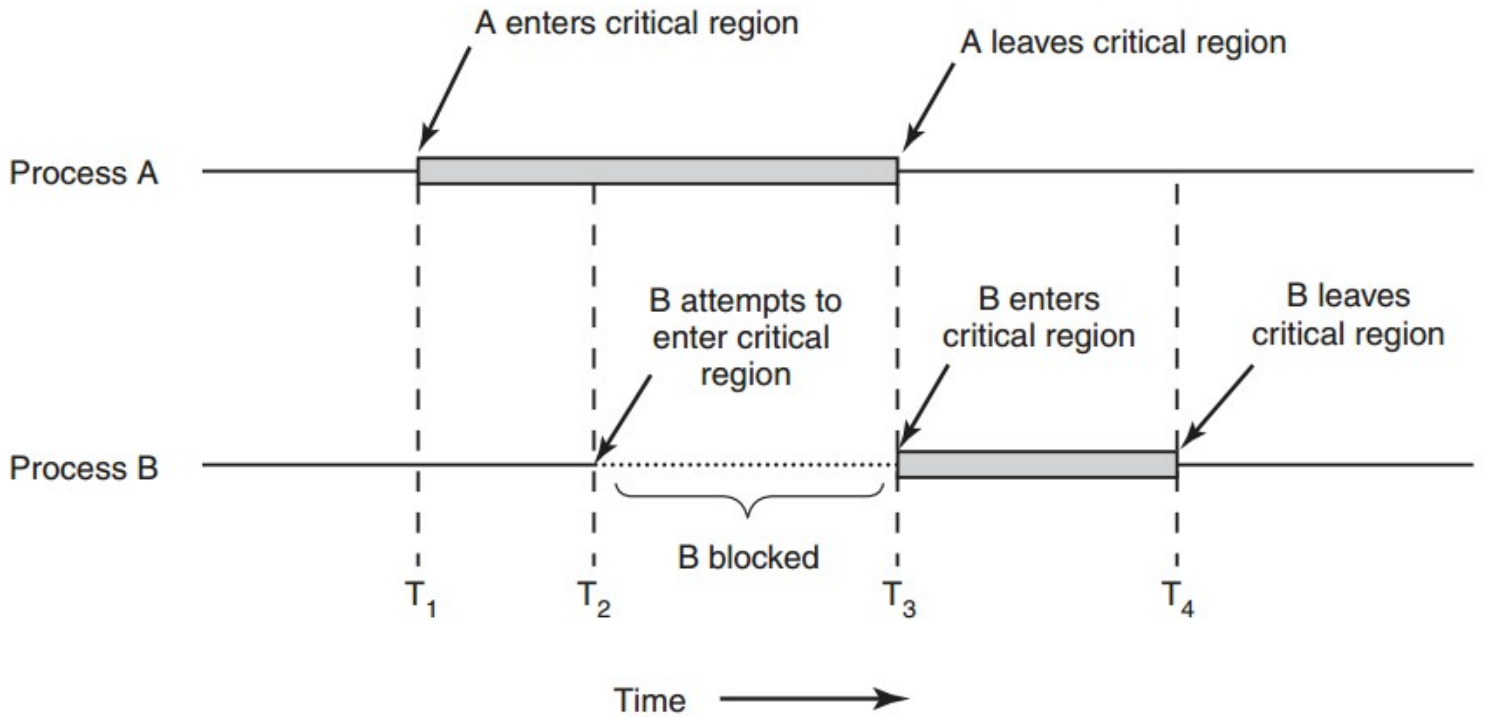
- **Regiões críticas**
  - Para **evitar condições** de **disputa**, é necessário **proibir** que **mais** de **um processo leia** ou **escreva** um certo **recurso compartilhado** ao **mesmo tempo**;
  - É necessário, portanto, **estabelecer** um **esquema** de **exclusão mútua**;
  - A parte do **programa** que **contém** o **recurso compartilhado** é denominado de **região crítica**.

Comunicação entre processos

INSTITUTO MAUÁ DE TECNOLOGIA

- **Regiões críticas**
  - Na realidade, as **condições** para **evitar problemas** de **leitura/escrita** de **dois** ou **mais processos** em um certo são as seguintes:
    - **(1) Não** se pode **permitir** que **dois processos** estejam executando **simultaneamente** dentro de suas **regiões críticas**;
    - **(2) Não** se pode **assumir nada** a **respeito** da **velocidade** ou **número** de **CPUs** (isso não deve afetar a corretude do algoritmo);
    - **(3) Nenhum processo** executando **fora** de sua **região crítica** poderá **bloquear** qualquer **outro processo**;
    - **(4) Nenhum processo** deverá **esperar indefinidamente** para **entrar** em sua **seção crítica**.

- Regiões críticas
  - Exclusão mútua por regiões críticas



Comunicação entre processos

INSTITUTO MAUÁ DE TECNOLOGIA

- **Exclusão mútua com espera ocupada**
  - A **exclusão mútua** pode ser **obtida** pela **técnica** denominada **espera ocupada** (*busy waiting*);
  - Existem diversas **propostas** para **implementar** a **espera ocupada**:
    - Desabilitando interrupções;
    - Variáveis de travamento;
    - Alternância estrita;
    - Solução de Peterson;
    - Instrução TSL.

Comunicação entre processos

INSTITUTO MAUÁ DE TECNOLOGIA

- **Exclusão mútua com espera ocupada**
  - **Desabilitando interrupções**
    - Em sistemas de um **único processador**, a **solução** mais **simples** é fazer com que **cada processo** que **desabilite** todas **interrupções assim que entrar** na **seção crítica** e **reabilitá-las** logo **antes** de **sair** desta **seção**;
    - **Sem interrupções habilitadas**, **não ocorrem interrupções** de **relógio** – a **CPU** só **comuta** de **processo para processo** com **interrupção** de **relógio ou outras**;
    - **Portanto**, uma vez que um **processo desabilitou interrupções**, ele pode **examinar** e **atualizar** uma **memória compartilhada sem** que qualquer **outro processo intervenha**.

Comunicação entre processos

INSTITUTO MAUÁ DE TECNOLOGIA

- **Exclusão mútua com espera ocupada**
  - **Desabilitando interrupções**
    - **Problemas com esta abordagem**:
      - **Não** é **inteligente dar** aos **processos** a **capacidade** de **ligar/desligar interrupções** – e se um deles esquecer de restaurar as interrupções?
      - Em sistemas com **múltiplas CPUs** as **interrupções** afetadas são aquelas **da CPU** onde o **processo** em questão está **executando** – o **que acontece** com as **outras CPUs**? Os **processos** delas **poderão acessar** a **região crítica**;
      - Essa técnica é **utilizada raramente** …

Comunicação entre processos

INSTITUTO MAUÁ DE TECNOLOGIA

- **Exclusão mútua com espera ocupada**
  - **Variáveis de travamento**
    - **Variáveis** de **travamento** (*lock variables*) são **variáveis** que **indicam** se um **processo pode ou não executar** em uma **região crítica**;
    - A **dinâmica** é a seguinte: uma **variável** de **travamento** é **inicializada sempre** com **0**. Quando um **processo deseja entrar** em uma região **crítica** de **código**, ele **inspeciona** a **variável**. **Se** seu **valor** for **0**, então **o processo altera** seu **valor para 1** e **entra** na **seção crítica**; **senão**, se seu **valor for 1** o **processo aguarda até** que seu **valor retorne** novamente à **0**;
    - **Então**, o **valor 1 indica** que uma **seção crítica** está sendo **utilizada** por algum processo e o **valor 0** indica que **nenhum processo** está **utilizando** a **seção**.

INSTITUTO MAUÁ DE TECNOLOGIA

- **Exclusão mútua com espera ocupada**
  - **Variáveis de travamento**
    - **Problemas com esta abordagem**:
      - Pode acontecer o seguinte **cenário**: supor que um **processo consulte** uma **variável** de travamento e **identifique** que seu **valor** é **0**;
      - **Antes** de ele **alterar** o **valor** da **variável**, o **agendador agenda outro processo** que **inspeciona** a mesma **variável**, **identifica** que o **valor** é **0** e **prontamente altera** seu **valor para 1**;
      - Supondo que o **agendador retorne** ao **primeiro processo**, **este segue alterando** a **variável** de travamento **para 1** (lembra, para ele ela tinha o valor 0)
      - **Por fim**, tem-se **dois processos** que estarão na **mesma seção** crítica de código ao mesmo tempo! O que **não** é **aceitável**…

Comunicação entre processos

- **Exclusão mútua com espera ocupada**
  - **Alternância estrita**
    - **Observar** o **código** a seguir:

Processo 0

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

Processo 1

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

    - **Inicialmente**, o **Processo 0 inspeciona** a **variável** `turn` (que indica a **vez** do **processo**), **descobre** que é **0** (valor inicial) e entra na região crítica. O **Processo 1** também **verifica** que é **0** e **aguarda** em uma **repetição testando continuamente** se `turn` **alterou** para **1**.

INSTITUTO MAUÁ DE TECNOLOGIA

- **Exclusão mútua com espera ocupada**
  - **Alternância estrita**
    - Quando o **Processo 0 deixa** a **seção crítica** ele **altera** `turn` para **1**, **permitindo** que o **Processo 1 entre** na **região crítica** de código.
    - Supor que o **Processo 1 termine** sua **região crítica** tão **rápido de modo** que **ambos processos** ainda **estejam** em **suas regiões não-críticas** de **código com** `turn` com valor **0**;
    - Admitir que, a seguir, o **Processo 0 execute** seu **laço** de repetição **rapidamente** e **saia** da **região crítica** e **altere** `turn` para **1**. **Nesse ponto**, tanto **Processo 0** quanto **Processo 1** estão **executando** na **seção não-crítica**.

INSTITUTO MAUÁ DE TECNOLOGIA

- **Exclusão mútua com espera ocupada**
  - **Alternância estrita**
    - Neste ponto, **supor** que o **Processo 0 rapidamente termine** suma **região não-crítica** e **volte** para o **topo** do **seu laço** de **repetição**. Mas o **Processo 0 não** pode **entrar** em sua **região crítica** pois `turn` **vale 1** e **então** fica **aguardando** a sua **entrada até** que `turn` **volte** a **ser 0** – **depende** da **velocidade** do **Processo 1** em **alterar** `turn` em **0**;
    - Esse **mecanismo** de um **processo** ficar **continuamente testando** uma **variável até** que seu **valor mude**, é denominado de **espera ocupada** (*busy waiting*) e uma **variável** de **travamento** que é **utilizada** com **espera ocupada** é denominada *spin lock*.

Comunicação entre processos

INSTITUTO MAUÁ DE TECNOLOGIA

- **Exclusão mútua com espera ocupada**
  - **Alternância estrita**
    - **Problemas com esta abordagem**
      - Esta abordagem **não** é **aconselhável** quando um dos **processos** é **lento** – se o **processo lento** estiver na **região não-crítica** de código, ele **afetará** a **entrada** de **outro** processo **na região crítica** – **viola** a **regra** (3);
      - Esta **solução requer** que os dois **processos alternem-se estritamente** na **entrada** de suas **regiões críticas**.

INSTITUTO MAUÁ DE TECNOLOGIA

- **Exclusão mútua com espera ocupada**
  - **Solução de Peterson**
    - Utiliza **duas funções**, `enter_region(p)` e `leave_region(p)` que respectivamente **devem ser executadas** para **entrar** na **região crítica** e **sair** da **região crítica** de código;
    - O **parâmetro** `p` representa o **número** de um **processo**: pode ser **0** ou **1**;
    - O algoritmo de **Peterson** é **correto** no **acesso concorrente** à **região crítica** de código: por **utilizar duas variáveis** cuja alteração se dá em estados distintos, **sempre um dos processos** vai **entrar** na **região crítica antes** do **outro** – **mesmo** quando **há** um **acesso** quase **simultâneo**;
    - A única **crítica** é ainda o uso da **técnica** de **espera ocupada**.

Comunicação entre processos

INSTITUTO MAUÁ DE TECNOLOGIA

- **Exclusão mútua com espera ocupada**
  - **Solução de Peterson**

```
#define FALSE  0
#define TRUE   1
#define N      2                        /* number of processes */

int turn;                               /* whose turn is it? */
int interested[N];                      /* all values initially 0 (FALSE) */

void enter_region(int process);         /* process is 0 or 1 */
{
      int other;                        /* number of the other process */

      other = 1 – process;              /* the opposite of process */
      interested[process] = TRUE;       /* show that you are interested */
      turn = process;                   /* set flag */
      while (turn == process && interested[other] == TRUE)  /* null statement */ ;
}

void leave_region(int process)          /* process: who is leaving */
{
      interested[process] = FALSE;      /* indicate departure from critical region */
}
```

Comunicação entre processos

- **Exclusão mútua com espera ocupada**
  - **Instrução TSL**
    - Neste caso, **utiliza-se alguma instrução** do **processador** para **implementar** o **acesso** à **memória**, criando um **travamento** em uma **certa posição** dela – válida para sistemas **multiprocessados**;
    - Pode-se imaginar uma **instrução como** `TSL RX,LOCK` (*Test and Set Lock*) que funciona assim: ela **lê** o **conteúdo** da **posição** de **memória travada** `LOCK` em `RX` e então **armazena** um **valor não zero** na **posição** de **memória** que está **travada** (`LOCK`). As **operações** de **ler** da memória travada e de **escrever** nela são **garantidas** a serem **indivisíveis** – **nenhum outro processador** pode **acessar** esta palavra de memória **até** a **instrução termine**, com a restauração do valor de `LOCK`.
    - A técnica é correta. O travamento ocorre a nível de bus – mas ainda é um exemplo da técnica de espera ocupada.

Comunicação entre processos

INSTITUTO MAUÁ DE TECNOLOGIA

- **Exclusão mútua com espera ocupada**
  - **Instrução TSL**
    - **Exemplo**

```
enter_region:
        TSL REGISTER,LOCK       | copy lock to register and set lock to 1
        CMP REGISTER,#0         | was lock zero?
        JNE enter_region        | if it was not zero, lock was set, so loop
        RET                     | return to caller; critical region entered

leave_region:
        MOVE LOCK,#0            | store a 0 in lock
        RET                     | return to caller
```

Comunicação entre processos

INSTITUTO MAUÁ DE TECNOLOGIA

- **Exclusão mútua com espera ocupada**
  - **Críticas finais** sobre esta técnica:
    - Há um **gasto desnecessário** de **CPU** (por conta da espera ao acesso);
    - Ocorre o **problema** da **inversão** de **prioridade**:
      - Supondo **dois processos**, `H`, com **alta prioridade** e `L` com **baixa prioridade**. O **agendador sempre escolhe executar** `H` **quando** ele está **pronto**;
      - Em um **dado momento**, `L` **está** na sua **região crítica** de código e **então** supor que `H` volta a ficar no **estado pronto** (**após** uma **interrupção** de E/S)
      - Só que `H` **fica** em **espera ocupada**, **mas** desde que `L` **não** é **agendado** para **continuar** (baixa prioridade) `L` **nunca** tem a **chance** de **sair** da **região crítica** de código e `H` fica **eternamente esperando entrar** lá.

Comunicação entre processos

INSTITUTO MAUÁ DE TECNOLOGIA

- **Dormir e acordar**
  - **Nesta técnica**, no lugar de os **processos** gastarem CPU aguardando a liberação de um recurso, eles **são suspensos** – "**dormem**" – **até** que possam ser **reativados** – "**acordar**" – e **acessar** o recurso;
  - **Primitivas** de **comunicação**
    - `sleep`: é uma **chamada** de **sistema** que faz com que o **processo chamador** seja **suspenso** até que outro processo o acorde;
    - `wakeup`: é uma **chamada** de **sistema** que **acorda** um **processo passado** como **parâmetro**.

Comunicação entre processos

INSTITUTO MAUÁ DE TECNOLOGIA

- **Dormir e acordar**
  - **O problema do Produtor-Consumidor**
    - **Também conhecido** como o problema do **Buffer Limitado**;
    - **Dois processos compartilham** um **buffer comum**, de **tamanho fixo**. Um deles, o **produtor**, **escreve dados** no **buffer**, e o outro, o **consumidor**, os **retira** (é possível generalizar o problema para ter *m* produtores e *n* consumidores, mas considerar o caso de um produtor e um consumidor simplifica as soluções.);
    - **Problemas**:
      - O **produtor** quer **colocar** um **novo item** no **buffer** mas este já **cheio**. A **solução** é que o **produtor durma** e seja **despertado** quando o **consumidor removeu um** ou **mais itens**;
      - O **consumidor** quer remover **um item** do **buffer** e percebe que o **buffer** está **vazio**. Ele vai **dormir** até que o **produtor coloque algo** no **buffer** e **acorde**.

Comunicação entre processos

INSTITUTO MAUÁ DE TECNOLOGIA

- **Dormir e acordar**
  - **O problema do Produtor-Consumidor**
    - **Exemplo de implementação**

```
#define N 100                                   /* number of slots in the buffer */
int count = 0;                                  /* number of items in the buffer */

void producer(void)
{
        int item;

        while (TRUE) {                          /* repeat forever */
                item = produce_item();          /* generate next item */
                if (count == N) sleep();        /* if buffer is full, go to sleep */
                insert_item(item);              /* put item in buffer */
                count = count + 1;              /* increment count of items in buffer */
                if (count == 1) wakeup(consumer);   /* was buffer empty? */
        }
}
```

Comunicação entre processos

INSTITUTO MAUÁ DE TECNOLOGIA

- **Dormir e acordar**
  - **O problema do Produtor-Consumidor**
    - **Exemplo de implementação (cont.)**

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

        while (TRUE) {                          /* repeat forever */
                if (count == 0) sleep( );       /* if buffer is empty, got to sleep */
                item = remove_item( );          /* take item out of buffer */
                count = count – 1;              /* decrement count of items in buffer */
                if (count == N – 1) wakeup(producer);   /* was buffer full? */
                consume_item(item);             /* print item */
        }
}
```

Comunicação entre processos

INSTITUTO MAUÁ DE TECNOLOGIA

- **Dormir e acordar**
  - **O problema do Produtor-Consumidor**
    - A solução **possui** os mesmos **problemas** de **condição** de **disputa** como já apresentado;
    - Para **controlar** o **número** de **itens** no **buffer**, utiliza-se a **variável** `count`;
    - **Código do produtor**: se o **número máximo** de **itens** que o **buffer** pode **comportar** é `N`, o **produtor primeiro testa** para verificar se `count` é `N`; **se for**, o **produtor dormirá**; **senão** o **produtor adicionará** um **item** ao **buffer** e **incrementará** a variável `count`;
    - **Código do consumidor**: ele **primeiramente testa** a **variável** `count` **para** verificar se é **zero**. **Se for**, então **ele** dormirá; **senão** ele **removerá** um **item** e decrementará a variável `counter`.

Comunicação entre processos

INSTITUTO MAUÁ DE TECNOLOGIA
MAUÁ

- **Dormir e acordar**
  - **O problema do Produtor-Consumidor**
    - A **condição** de **disputa** neste problema ocorre porque o **acesso** à **variável** `count` é **irrestrito**;
    - Pode **ocorrer** a seguinte **situação**: o **buffer** está **vazio** e o **consumidor** acabou de **ler** o **valor** de **count** para **checar** se é **0**. **Nesse instante**, o **agendador decide parar** a execução do **consumidor temporariamente** e **iniciar** a execução do **produtor**;
    - O **produtor insere** um **item** no **buffer**, **incrementa** `count` e **verifica** que **agora** é **1**. **Depois**, o **produtor executa** `wakeup` para **acordar** o **consumidor**. **Mas** o **consumidor não** está **logicamente dormindo** e o sinal `wakeup` é **perdido**.
    - Quando o **consumidor** volta a **executar**, ele **testa** o **último valor** de `count` lido, que é **0** e então **volta** a **dormir**. **Mais cedo** ou **mais tarde** o **produtor preencherá** o **buffer** e também **dormirá**;
    - **Por fim ambos dormirão eternamente**…

Referências bibliográficas

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