SO
Sistemas Operacionais

Prof. Rodrigo Martins
rodrigo.martins@francomontoro.com.br

Cronograma

- Comunicação entre Processos

- Exercícios

Comunicação entre Processos

- Processos quase sempre precisam comunicar-se com outros processos.
  - Por exemplo, em um pipeline do interpretador de comandos, a saída do primeiro processo tem de ser passada para o segundo, e assim por diante até o fim da linha. Então, há uma necessidade por comunicação entre os processos, de preferência de uma maneira bem estruturada sem usar interrupções.

Comunicação entre Processos

- Há três questões importantes relacionadas a comunicação entre processos.

- A primeira é como um processo pode passar informações para outro.

- A segunda tem a ver com certificar-se de que dois ou mais processos não se atrapalhem.
  - Por exemplo, dois processos em um sistema de reserva de uma companhia aérea cada um tentando ficar com o último assento em um avião para um cliente diferente.

Comunicação entre Processos

- A terceira diz respeito ao sequenciamento adequado quando dependências estão presentes: se o processo A produz dados e o processo B os imprime, B tem de esperar até que A tenha produzido alguns dados antes de começar a imprimir.

Condições de corrida

- Quando dois ou mais processos estão lendo ou escrevendo alguns dados compartilhados e o resultado final depende de quem executa precisamente e quando, são chamadas de condições de corrida.

Condições de corrida

- Em alguns sistemas operacionais, processos que estão trabalhando juntos podem compartilhar de alguma memória comum que cada um pode ler e escrever.

- A memória compartilhada pode encontrar-se na memória principal ou ser um arquivo compartilhado; o local da memória compartilhada não muda a natureza da comunicação ou os problemas que surgem.

Exemplo de Condições de corrida

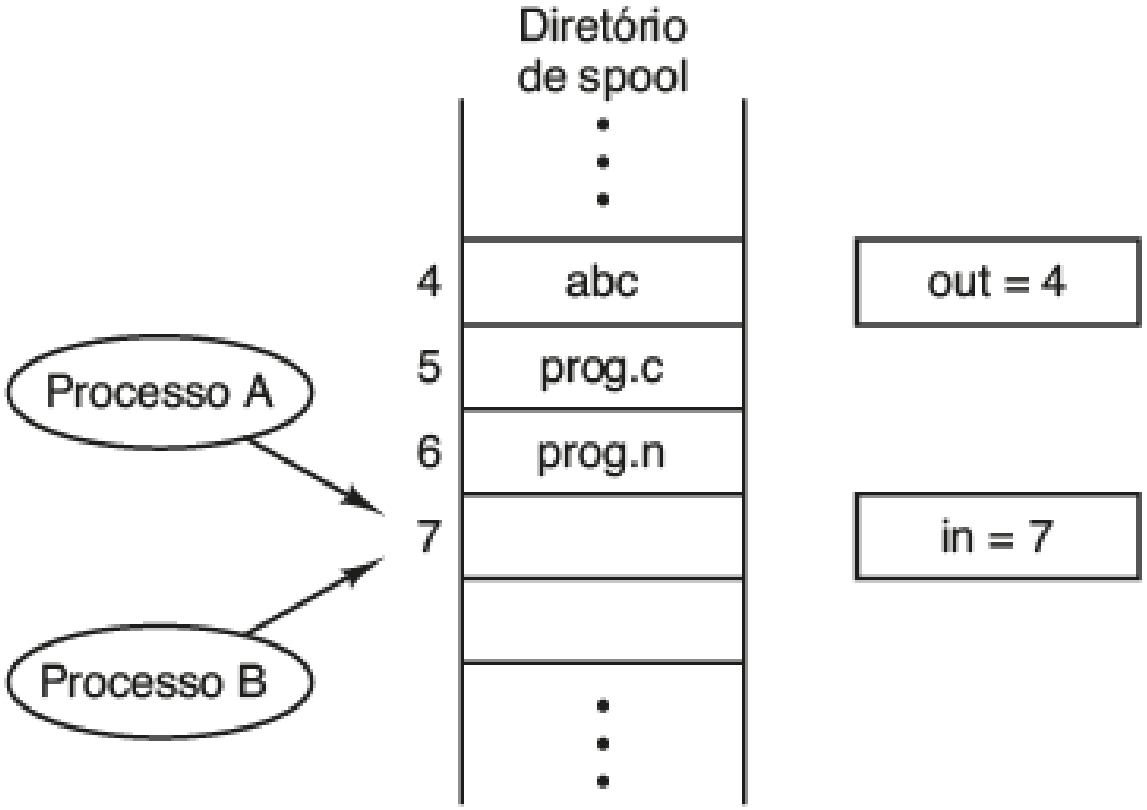
- Vamos considerar um exemplo simples, mas comum: um spool de impressão.

- Quando um processo quer imprimir um arquivo, ele entra com o nome do arquivo em um diretório de spool especial. Outro processo, o daemon de impressão, confere periodicamente para ver se há quaisquer arquivos a serem impressos, e se houver, ele os imprime e então remove seus nomes do diretório.

- No próximo slide temos a figura onde os processos A e B decidem que querem colocar um arquivo na fila para impressão.

Exemplo de Condições de corrida

FIGURA 2.21 Dois processos querem acessar a memória compartilhada ao mesmo tempo.



Exemplo de Condições de corrida

- O **processo A** lê ***in*** e armazena o valor, **7**, em uma variável local chamada ***next_free_slot***. Logo em seguida uma interrupção de relógio ocorre e a CPU decide que o **processo A** executou por tempo suficiente, então, ele troca para o **processo B**. O **processo B** também lê ***in*** e recebe um **7**. Ele, também, o armazena em sua variável local ***next_free_slot***. Nesse instante**, ambos os processos** acreditam que a próxima vaga disponível é 7.

- O **processo B** agora continua a executar. Ele armazena o nome do seu arquivo na vaga **7** e atualiza ***in*** para ser um **8**. Então ele segue em frente para fazer outras coisas.

- Por fim, o **processo A** executa novamente, começando do ponto onde ele parou. Ele olha para ***next_free_ slot***, encontra um **7** ali e escreve seu nome de arquivo na vaga **7**, apagando o nome que o **processo B** recém colocou ali. Então ***calcula next_free_slot + 1***, que é **8**, e configura ***in*** para **8**.

- O **diretório de spool** está agora internamente consistente, então o **daemon** de impressão não observará nada errado, mas o **processo B jamais** receberá qualquer saída. O usuário B ficará em torno da impressora por anos, aguardando esperançoso por uma saída que nunca virá.

Como evitar as Condições de Corrida?

- A chave para evitar problemas e em muitas outras situações envolvendo memória compartilhada, arquivos compartilhados e tudo o mais compartilhado é encontrar alguma maneira de proibir mais de um processo de ler e escrever os dados compartilhados ao mesmo tempo.

- Colocando a questão em outras palavras, o que precisamos é de **exclusão mútua**, isto é, alguma maneira de se certificar de que se um processo está usando um arquivo ou variável compartilhados, os outros serão impedidos de realizar a mesma coisa.

Como evitar as Condições de Corrida?

- Durante parte do tempo, um processo está ocupado realizando computações internas e outras coisas que não levam a condições de corrida. No entanto, às vezes um processo tem de acessar uma memória compartilhada ou arquivos, ou realizar outras tarefas críticas que podem levar a corridas.

- Essa parte do programa onde a memória compartilhada é acessada é chamada de **região crítica ou seção crítica**.

Região Crítica

- Se conseguíssemos arranjar as coisas de maneira que jamais dois processos estivessem em suas regiões críticas ao mesmo tempo, poderíamos evitar as corridas.

- Embora essa exigência evite as condições de corrida, ela não é suficiente para garantir que processos em paralelo cooperem de modo correto e eficiente usando dados compartilhados.

Região Crítica

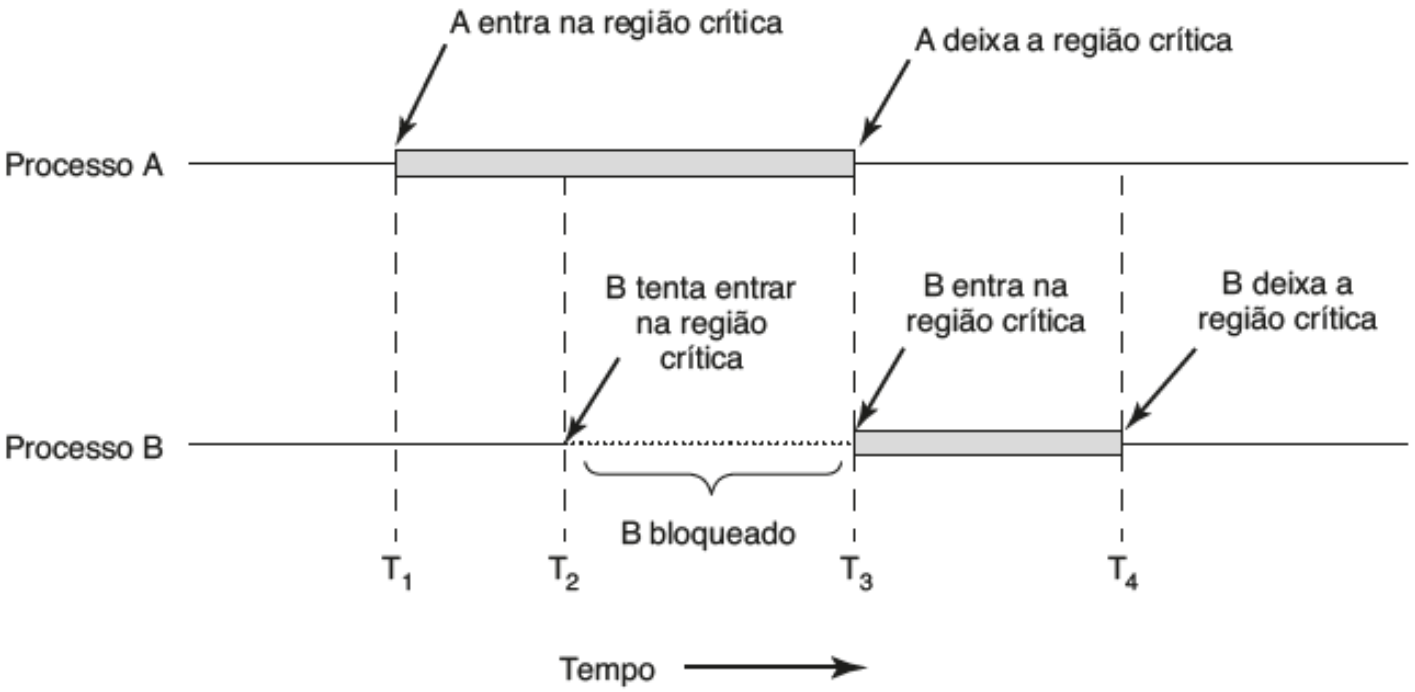
- Precisamos que quatro condições se mantenham para chegar a uma boa solução:

  1. Dois processos jamais podem estar simultaneamente dentro de suas regiões críticas.
  2. Nenhuma suposição pode ser feita a respeito de velocidades ou do número de CPUs.
  3. Nenhum processo executando fora de sua região crítica pode bloquear qualquer processo.
  4. Nenhum processo deve ser obrigado a esperar eternamente para entrar em sua região crítica.

- O comportamento que queremos é mostrado na Figura no próximo slide.

Região Crítica

FIGURA 2.22 Exclusão mútua usando regiões críticas.



Região Crítica

- Na figura anterior o **processo A** entra na sua região crítica no tempo **T1**. Um pouco mais tarde, no tempo T2, o **processo B** tenta entrar em sua região crítica, mas não consegue porque outro processo já está em sua região crítica e só permitimos um de cada vez. Em consequência, **B** é temporariamente suspenso até o tempo **T3**, quando **A** deixa sua região crítica, permitindo que **B** entre de imediato. Por fim, B sai (**em T4**) e estamos de volta à situação original sem nenhum processo em suas regiões críticas.

Exclusão mútua com espera ocupada

- Examinaremos várias propostas para realizar a exclusão mútua, de maneira que enquanto um processo está ocupado atualizando a memória compartilhada em sua região crítica, nenhum outro entrará na sua região crítica para causar problemas.

Desabilitando interrupções

- Em um sistema de processador único, a solução mais simples é fazer que cada processo desabilite todas as interrupções logo após entrar em sua região crítica e as reabilitar um momento antes de partir.

- Desabilitar interrupções é muitas vezes uma técnica útil dentro do próprio sistema operacional, mas não é apropriada como um mecanismo de exclusão mútua geral para processos de usuário.

Desabilitando interrupções

- A possibilidade de alcançar a **exclusão mútua** desabilitando interrupções — mesmo dentro do núcleo — está se tornando menor a cada dia por causa do número cada vez maior de chips multinúcleo mesmo em PCs populares.

- Em um sistema multinúcleo desabilitar as interrupções de uma CPU não evita que outras CPUs interfiram com as operações que a primeira está realizando. Em consequência, esquemas mais sofisticados são necessários.

Variáveis do tipo trava

- Uma segunda tentativa é uma **solução de software**.
- Considere ter uma única variável (de trava) compartilhada, **inicialmente 0**.
- Quando um processo quer entrar em sua **região crítica**, ele primeiro testa a trava. **Se** a trava é **0**, o processo a configura para **1** e entra na **região crítica**.
- **Se** a trava já é **1**, o processo apenas espera até que ela se torne **0**. Desse modo, um **0** significa que nenhum processo está na região crítica, e um **1** significa que algum processo está em sua região crítica.

Variáveis do tipo trava

- Infelizmente, essa ideia **contém uma falha fatal**.
- Suponha que um processo lê a trava e vê que ela é **0**.
- Antes que ele possa configurar a trava para **1**, outro processo está escalonado, executa e configura a trava para **1**. Quando o primeiro processo executa de novo, ele também configurará a trava para **1**, e dois processos estarão nas suas regiões críticas ao mesmo tempo.

Alternância explícita

- Uma **terceira abordagem** para o problema da **exclusão mútua** é mostrada na figura ao lado. É um trecho de programa escrito na linguagem C.

FIGURA 2.23 Uma solução proposta para o problema da região crítica. (a) Processo 0. (b) Processo 1. Em ambos os casos, certifique-se de observar os pontos e vírgulas concluindo os comandos while.

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

(a)

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

(b)

Alternância explícita

- Na Figura do slide anterior, a variável do tipo inteiro **turn**, inicialmente **0**, serve para controlar de quem é a vez de entrar na região crítica e examinar ou atualizar a memória compartilhada.

- Inicialmente, o **processo 0** inspeciona turn, descobre que ele é **0** e entra na sua **região crítica**. O **processo 1** também encontra lá o valor **0** e, portanto, espera em um laço fechado testando continuamente **turn** para ver quando ele vira **1**.

Alternância explícita

- Quando o **processo 0** deixa a região crítica, ele configura **turn** para **1**, a fim de permitir que o **processo 1** entre em sua região crítica.

- Suponha que o **processo 1** termine sua região rapidamente, de modo que ambos os processos estejam em suas regiões não críticas, com **turn** configurado para **0**.

- Agora o **processo 0** executa todo seu laço rapidamente, deixando sua região crítica e configurando **turn** para **1**. Nesse ponto, **turn** é **1** e ambos os processos estão sendo executados em suas **regiões não críticas**.

Alternância explícita

- De repente, o **processo 0** termina sua **região não crítica** e volta para o topo do seu laço. Infelizmente, não lhe é permitido entrar em sua região crítica agora, pois **turn** é **1** e o **processo 1** está ocupado com sua região não crítica. Ele espera em seu laço **while** até que o **processo 1** configura **turn** para **0**. Ou seja, chavear a vez não é uma boa ideia quando um dos processos é muito mais lento que o outro.

- Essa solução exige que os **dois processos** alternem-se estritamente na entrada em suas regiões críticas. Apesar de evitar todas as corridas, esse algoritmo **não é um sério candidato a uma solução**.

Alternância explícita

- Testar continuamente uma variável até que algum valor apareça é chamado de **espera ocupada**.

- Em geral ela deve ser evitada, já que desperdiça tempo da CPU. Apenas quando há uma expectativa razoável de que a espera será curta, a espera ocupada é usada. Uma trava que usa a espera ocupada é chamada de **trava giratória (spin lock)**.

Solução de Peterson

- Em 1981, G. L. Peterson descobriu uma maneira muito mais simples de realizar a exclusão mútua. O algoritmo de Peterson é mostrado na próxima figura.

A solução de Peterson para realizar a exclusão mútua.

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

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

void enter_region(int process);         /* processo e 0 ou 1 */
{
      int other;                        /* numero do outro processo */

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

void leave_region(int process)          /* processo: quem esta saindo */
{
      interested[process] = FALSE;      /* indica a saida da regiao critica */
}
```

Solução de Peterson

- Antes de usar as variáveis compartilhadas, cada processo chama **enter_region** com seu próprio número de processo, **0** ou **1**, como parâmetro.

- Essa chamada fará que ele espere, se necessário, até que seja seguro entrar. Após haver terminado com as variáveis compartilhadas, o processo chama **leave_region** para indicar que ele terminou e para permitir que outros processos entrem, se assim desejarem.

Solução de Peterson

- **Vamos ver como essa solução funciona**.

- Inicialmente, nenhum processo está na sua região crítica. Agora o **processo 0** chama **enter_region**. Ele indica o seu interesse alterando o valor de seu elemento de arranjo e alterando **turn** para **0**.

- Como o **processo 1** não está interessado, **enter_region** retorna imediatamente. Se o **processo 1** fizer agora uma chamada para **enter_region**, ele esperará ali até que **interested[0]** mude para **FALSE**, um evento que acontece apenas quando o **processo 0** chamar **leave_region** para deixar a região crítica.

Solução de Peterson

- **Agora considere o caso em que ambos os processos chamam enter_region quase simultaneamente.**

- Ambos armazenarão seu número de processo em **turn**. O último a armazenar é o que conta; o primeiro é sobrescrito e perdido. Suponha que o **processo 1** armazene por último, então **turn** é **1**.

- Quando ambos os processos chegam ao comando **while**, o **processo 0** o executa zero vez e entra em sua região crítica. O **processo 1** permance no laço e não entra em sua região crítica até que o **processo 0** deixe a sua.

Sleep e Wakeup

- Vamos examinar algumas primitivas de **comunicação entre processos** que bloqueiam em vez de desperdiçar tempo da CPU quando eles não são autorizados a entrar nas suas regiões críticas.

- Uma das mais simples é o **par sleep** e **wakeup**.

- **Sleep** é uma chamada de sistema que faz com que o processo que a chamou bloqueie, isto é, seja suspenso até que outro processo o desperte.

- **Wakeup** tem um parâmetro, o processo a ser desperto. Alternativamente, tanto sleep quanto wakeup cada um tem um parâmetro, um endereço de memória usado para parear sleeps com wakeups.

O problema do produtor-consumidor

- **Um exemplo de como essas primitivas podem ser usadas, vamos considerar o problema produtor-consumidor.**

- Dois processos compartilham de um buffer de tamanho fixo comum. Um deles, o produtor, insere informações no buffer, e o outro, o consumidor, as retira dele.

- O **problema** surge quando o **produtor** quer **colocar** um **item novo** no **buffer**, mas ele já está **cheio**. A **solução** é o produtor ir dormir, para ser desperto quando o consumidor tiver removido um ou mais itens. De modo similar, se o consumidor quer remover um item do buffer e vê que este está vazio, ele vai dormir até o produtor colocar algo no buffer e despertá-lo.

O problema do produtor-consumidor

- O código , produtor e consumidor, é mostrado na próxima figura.

FIGURA 2.27 O problema do produtor-consumidor com uma condição de corrida fatal.

```
#define N 100                                   /* numero de lugares no buffer */
int count = 0;                                  /* numero de itens no buffer */

void producer(void)
{
     int item;

     while (TRUE) {                             /* repita para sempre */
          item = produce_item( );               /* gera o proximo item */
          if (count == N) sleep( );             /* se o buffer estiver cheio, va dormir */
          insert_item(item);                    /* ponha um item no buffer */
          count = count + 1;                    /* incremente o contador de itens no buffer */
          if (count == 1) wakeup(consumer);     /* o buffer estava vazio? */
     }
}


void consumer(void)
{
     int item;

     while (TRUE) {                             /* repita para sempre */
          if (count == 0) sleep( );             /* se o buffer estiver cheio, va dormir */
          item = remove_item( );                /* retire o item do buffer */
          count = count – 1;                    /* descresca de um contador de itens no buffer */
          if (count == N – 1) wakeup(producer); /* o buffer estava cheio? */
          consume_item(item);                   /* imprima o item */
     }
}
```

O problema do produtor-consumidor

- A essência do problema aqui é que um chamado de despertar enviado para um processo que (ainda) não está dormindo é perdido.

- Ao construir exemplos com três ou mais processos nos quais o bit de espera pelo sinal de acordar é insuficiente. Poderíamos fazer outra simulação e acrescentar um segundo bit de espera pelo sinal de acordar, ou talvez 8 ou 32 deles, mas em princípio o problema ainda está ali.

Semáforos

- Quando E. W. Dijkstra (1965) sugeriu usar uma variável inteira para contar o número de sinais de acordar salvos para uso futuro. Em sua proposta, um novo tipo de variável, que ele chamava de semáforo, foi introduzido. Um semáforo podia ter o valor 0, indicando que nenhum sinal de despertar fora salvo, ou algum valor positivo se um ou mais sinais de acordar estivessem pendentes.

Semáforos

- Dijkstra propôs ter **duas operações** nos semáforos, hoje normalmente chamadas de **down** e **up** (generalizações de **sleep** e **wakeup**, respectivamente).

- A operação **down** em um semáforo confere para ver se o valor é maior do que 0. Se for, ele decrementará o valor (isto é, gasta um sinal de acordar armazenado) e apenas continua. Se o valor for 0, o processo é colocado para dormir sem completar o down para o momento.

Semáforos

- A operação **up** incrementa o valor de um determinado semáforo. Se um ou mais processos estiverem dormindo naquele semáforo, incapaz de completar uma operação **down** anterior, um deles é escolhido pelo sistema (por exemplo, ao acaso) e é autorizado a completar seu down.

- Desse modo, após um up com processos dormindo em um semáforo, ele ainda estará em 0, mas haverá menos processos dormindo nele. A operação de incrementar o semáforo e despertar um processo também é indivisível. Nenhum processo é bloqueado realizando um up, assim como nenhum processo é bloqueado realizando um wakeup no modelo anterior.

Semáforos

- No exemplo da figura, usamos semáforos de duas maneiras diferentes. Essa diferença é importante o suficiente para ser destacada. O semáforo mutex é usado para exclusão mútua. Ele é projetado para garantir que apenas um processo de cada vez esteja lendo ou escrevendo no buffer e em variáveis associadas. Essa exclusão mútua é necessária para evitar o caos.

FIGURA 2.28 O problema do produtor-consumidor usando semáforos.

```
#define N 100                                   /* numero de lugares no buffer */
typedef int semaphore;                          /* semaforos sao um tipo especial de int */
semaphore mutex = 1;                            /* controla o acesso a regiao critica */
semaphore empty = N;                            /* conta os lugares vazios no buffer */
semaphore full = 0;                             /* conta os lugares preenchidos no buffer */

void producer(void)
{
     int item;

     while (TRUE) {                             /* TRUE e a constante 1 */
          item = produce_item();                /* gera algo para por no buffer */
          down(&empty);                         /* decresce o contador empty */
          down(&mutex);                         /* entra na regiao critica */
          insert_item(item);                    /* poe novo item no buffer */
          up(&mutex);                           /* sai da regiao critica */
          up(&full);                            /* incrementa o contador de lugares preenchidos */
     }
}


void consumer(void)
{
     int item;

     while (TRUE) {                             /* laco infinito */
          down(&full);                          /* decresce o contador full */
          down(&mutex);                         /* entra na regiao critica */
          item = remove_item();                 /* pega item do buffer */
          up(&mutex);                           /* sai da regiao critica */
          up(&empty);                           /* incrementa o contador de lugares vazios */
          consume_item(item);                   /* faz algo com o item */
     }
}
```

Semáforos

- O outro uso dos **semáforos** é para a **sincronização**.
- Os semáforos **full** e **empty** são necessários para garantir que determinadas sequências ocorram ou não.
- Nesse caso, eles asseguram que o produtor pare de executar quando o buffer estiver cheio, e que o consumidor pare de executar quando ele estiver vazio.

Mutexes

- Quando a capacidade do semáforo de fazer contagem não é necessária, uma versão simplificada, chamada **mutex**, às vezes é usada.

- Mutexes são bons somente para gerenciar a exclusão mútua de algum recurso ou trecho de código compartilhados.

Mutexes

- O código para mutex_lock e mutex_unlock para uso com um pacote de threads de usuário são mostrados na figura abaixo:

FIGURA 2.29 Implementação de *mutex_lock* e *mutex_unlock*.

```
mutex_lock:
    TSL REGISTER,MUTEX      | copia mutex para o registrador e atribui a ele o valor 1
    CMP REGISTER,#0         | o mutex era zero?
    JZE ok                  | se era zero, o mutex estava desimpedido, portanto retorne
    CALL thread_yield       | o mutex esta ocupado; escalone um outro thread
    JMP mutex_lock          | tente novamente
ok: RET                     | retorna a quem chamou; entrou na regiao critica

mutex_unlock:
    MOVE MUTEX,#0           | coloca 0 em mutex
    RET                     | retorna a quem chamou
```

Monitores (1)

- Para tornar mais fácil escrever programas corretos, Brinch Hansen (1975) e Hoare (1974) propuseram uma primitiva de sincronismo de mais alto nível, chamada de **monitor**.

- Suas propostas diferiam ligeiramente, conforme descrito a seguir. Um **monitor** é um conjunto de rotinas, variáveis e estruturas de dados, todas agrupadas em um tipo especial de módulo ou pacote.

```
monitor example
        integer i;
        condition c;

        procedure producer( );
        .
        .
        .
        end;

        procedure consumer( );
        .
        .
        .
        end;
end monitor;
```

Monitores (2)

```
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;
```

- Delineamento do problema do produtor-consumidor com monitores
  - somente um procedimento está ativo por vez no monitor
  - o buffer tem N lugares

Monitores (3)

```
public class ProducerConsumer {
    static final int N = 100;                   // constante com o tamanho do buffer
    static producer p = new producer( );        // instância de um novo thread produtor
    static consumer c = new consumer( );// instância de um novo thread consumidor
    static our_monitor mon = new our_monitor( );  // instância de um novo monitor

    public static void main(String args[ ]) {
        p.start( );                             // inicia o thread produtor
        c.start( );                             // inicia o thread consumidor
    }

    static class producer extends Thread {
        public void run( )  {                   // o método run contém o código do thread
            int item;
            while (true) {                      // laço do produtor
                item = produce_item( );
                mon.insert(item);
            }
        }
        private int produce_item( ) { ... }     // realmente produz
    }

    static class consumer extends Thread {
        public void run( )  {                   // método run contém o código do thread
            int item;
            while (true) {                      // laço do consumidor
                item = mon.remove( );
                consume_item (item);
            }
        }
        private void consume_item(int item) { ... }     // realmente consome
    }
    static class our_monitor {                  // este é o monitor
        private int buffer[ ] = new int[N];
        private int count = 0, lo = 0, hi = 0; // contadores e índices

        public synchronized void insert(int val) {
            if (count == N) go_to_sleep( );     // se o buffer estiver cheio, vá dormir
            buffer [hi] = val;                  // insere um item no buffer
            hi = (hi + 1) % N;                  // lugar para colocar o próximo item
            count = count + 1;                  // mais um item no buffer agora
            if (count == 1) notify( );          // se o consumidor estava dormindo, acorde-o
        }
        public synchronized int remove( ) {
            int val;
            if (count == 0) go_to_sleep( );     // se o buffer estiver vazio, vá dormir
            val = buffer [lo];                  // busca um item no buffer
            lo = (lo + 1) % N;                  // lugar de onde buscar o próximo item
            count = count - 1;                  // um item a menos no buffer
            if (count == N - 1) notify( );      // se o produtor estava dormindo, acorde-o
            return val;
        }
        private void go_to_sleep( ) { try{wait( );} catch(InterruptedException exc) {};}
    }
}
```

- Solução para o problema do produtor-consumidor em Java

Monitores (4)

```
#define N 100                                   /* número de lugares no buffer */

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

     while (TRUE) {
          item = produce_item( );               /* gera alguma coisa para colocar no buffer */
          receive(consumer, &m);                /* espera que uma mensagem vazia chegue */
          build_message(&m, item);              /* monta uma mensagem para enviar */
          send(consumer, &m);                   /* envia item para consumidor */
     }
}

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

     for (i = 0; i < N; i++) send(producer, &m);  /* envia N mensagens vazias */
     while (TRUE) {
          receive(producer, &m);                /* pega mensagem contendo item */
          item = extract_item(&m);              /* extrai o item da mensagem */
          send(producer, &m);                   /* envia a mensagem vazia como resposta */
          consume_item(item);                   /* faz alguma coisa com o item */
     }
}
```

- Solução para o problema do produtor-consumidor em Java (parte 2)

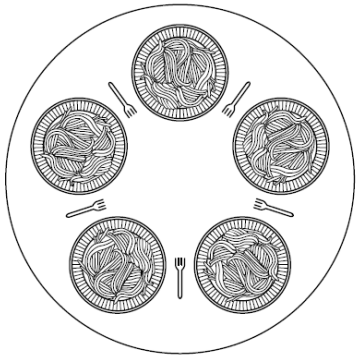
Troca de Mensagens

```
#define N 100                                   /* número de lugares no buffer */

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

     while (TRUE) {
          item = produce_item( );               /* gera alguma coisa para colocar no buffer */
          receive(consumer, &m);                /* espera que uma mensagem vazia chegue */
          build_message(&m, item);              /* monta uma mensagem para enviar */
          send(consumer, &m);                   /* envia item para consumidor */
     }
}

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

     for (i = 0; i < N; i++) send(producer, &m);  /* envia N mensagens vazias */
     while (TRUE) {
          receive(producer, &m);                /* pega mensagem contendo item */
          item = extract_item(&m);              /* extrai o item da mensagem */
          send(producer, &m);                   /* envia a mensagem vazia como resposta */
          consume_item(item);                   /* faz alguma coisa com o item */
     }
}
```

- O problema do produtor-consumidor com N mensagens

Jantar dos Filósofos (1)

- Em 1965, Dijkstra propôs e resolveu um problema de sincronização que chamou de problema da janta dos filósofos.
- Cinco filósofos estão sentados ao redor de uma mesa circular. Cada filósofo tem um prato de espaguete. O espaguete é tão escorregadio que o filósofo precisa de dois garfos para comê-lo. Entre cada par de pratos há um garfo. A disposição da mesa está ilustrada na Figura.
- Quando um filósofo sente fome, ele tenta pegar os garfos da esquerda e da direita, um de cada vez, em qualquer ordem. Se conseguir pegar os dois garfos, ele come por algum tempo e, então, coloca os garfos na mesa e continua a pensar. A pergunta fundamental é: você consegue escrever um programa para cada filósofo que faça o que deve fazer e nunca entre em impasse (deadlock)?



Jantar dos Filósofos (2)

- Uma solução errada para o problema do jantar dos filósofos

```
#define N 5                                     /* número de filósofos */

void philosopher(int i)                         /* i: número do filósofo, de 0 a 4 */
{
    while (TRUE) {
        think( );                               /* o filósofo está pensando */
        take_fork(i);                           /* pega o garfo esquerdo */
        take_fork((i+1) % N);                   /* pega o garfo direito; % é o operador modulo */
        eat( );                                 /* hummm! Espaguete! */
        put_fork(i);                            /* devolve o garfo esquerdo à mesa */
        put_fork((i+1) % N);                    /* devolve o garfo direito à mesa */
    }
}
```

Jantar dos Filósofos (3)

- Uma solução para o problema do jantar dos filósofos (parte 1)

```
#define N           5                   /* número de filósofos */
#define LEFT        (i+N–1)%N           /* número do vizinho à esquerda de i */
#define RIGHT       (i+1)%N             /* número do vizinho à direita de i */
#define THINKING    0                   /* o filósofo está pensando */
#define HUNGRY      1                   /* o filósofo está tentando pegar garfos */
#define EATING      2                   /* o filósofo está comendo */
typedef int semaphore;                  /* semáforos são um tipo especial de int */
int state[N];                           /* arranjo para controlar o estado de cada um */
semaphore mutex = 1;                    /* exclusão mútua para as regiões críticas */
semaphore s[N];                         /* um semáforo por filósofo */

void philosopher(int i)                 /* i: o número do filósofo, de 0 a N–1 */
{
    while (TRUE) {                      /* repete para sempre */
        think( );                       /* o filósofo está pensando */
        take_forks(i);                  /* pega dois garfos ou bloqueia */
        eat( );                         /* hummm! Espaguete! */
        put_forks(i);                   /* devolve os dois garfos à mesa */
    }
}
```

Jantar dos Filósofos (4)

- Uma solução para o problema do jantar dos filósofos (parte 2)

```
void take_forks(int i)          /* i: o número do filósofo, de 0 a N–1 */
{
    down(&mutex);               /* entra na região crítica */
    state[i] = HUNGRY;          /* registra que o filósofo está faminto */
    test(i);                    /* tenta pegar dois garfos */
    up(&mutex);                 /* sai da região crítica */
    down(&s[i]);                /* bloqueia se os garfos não foram pegos */
}

void put_forks(i)               /* i: o número do filósofo, de 0 a N–1 */
{
    down(&mutex);               /* entra na região crítica */
    state[i] = THINKING;        /* o filósofo acabou de comer */
    test(LEFT);                 /* vê se o vizinho da esquerda pode comer agora */
    test(RIGHT);                /* vê se o vizinho da direita pode comer agora */
    up(&mutex);                 /* sai da região crítica */
}

void test(i)                    /* i: o número do filósofo, de 0 a N–1 */
{
    if (state[i] == HUNGRY && state[LEFT] != EATING && state[RIGHT] != EATING) {
        state[i] = EATING;
        up(&s[i]);
    }
}
```

- O problema da janta dos filósofos é útil para modelar processos que estão competindo pelo acesso exclusivo a um número limitado de recursos, como dispositivos de E/S.

Exercícios

1. Cite e explique as três questões importantes relacionadas a comunicação entre processos.
2. Cite e exemplifique condições de corrida.
3. Como evitar as condições de corrida?
4. Cite e explique os 2 tipos primitivos de comunicação entre processos que bloqueiam em vez de desperdiçar tempo da CPU quando eles não são autorizados a entrar nas suas regiões críticas.
5. Cite e explique a técnica semáforos, proposto por E. W. Dijkstra em 1965.
6. Explique a técnica Mutex.
7. Explique a técnica Monitor.

Referência desta Aula

- TANENBAUM, A. S. Sistemas Operacionais Modernos, Prentice-Hall do Brasil, 4ª edição, 2016.

Fim
Obrigado

Rodrigo