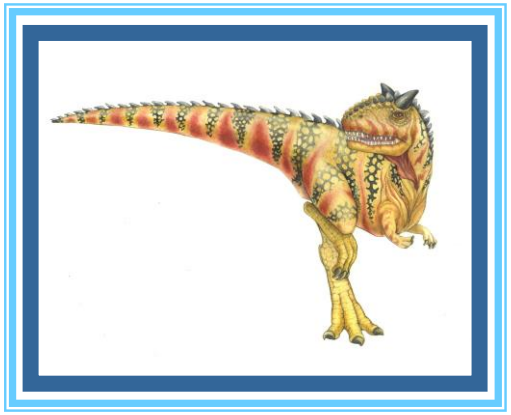
Capítulo 5: Sincronização de Processos



Capítulo 5: Sincronização de Processos

- Background (Fundo)
- O problema da Seção Crítica
- Solução de Peterson
- Sincronização de Hardware
- Bloqueios mutex
- Semáforos
- Clássicos Problemas de Sincronização
- Monitores
- Exemplos de Sincronização
- Abordagens Alternativas

Universidade de Brasília

Faculdade UnB Gama

Objetivos

- Apresentar o conceito de sincronização de processos
- Introduzir o problema da Seção-Crítica e quais soluções podem ser usadas para assegurar a consistência dos dados compartilhados
- Apresentar soluções tanto de software como hardware para o problema da Seção-Crítica
- Examinar vários problemas clássicos de sincronização de processos
- Explorar várias ferramentas que são usadas para resolver problemas de sincronização de processos

Fundo

- Processos podem ser executados simultaneamente
  - Alguns podem ser interrompidos por qualquer tempo, executando parcialmente o trabalho
- O acesso simultâneo a dados compartilhados pode gerar em incosistência de dados
- Manter dados consistentes requer mecanismos para garantir a execução ordenada de processos cooperantes
- Ilustração do problema:

  Suponha que nós esperamos fornecer uma solução para o problema consumidor-produtor que preenche **todos** os buffers. Podemos fazer isso com um **contador** inteiro que controla o número de buffers completos. Inicialmente, o **contador** é definido para 0. Ele é incrementado pelo produtor depois que esse produzir um novo buffer e é decrementado pelo consumidor depois desse consumir o buffer.

Produtor

```
while (true) {
        /* produce an item in next produced */

        while (counter == BUFFER_SIZE) ;
                /* do nothing */
        buffer[in] = next_produced;
        in = (in + 1) % BUFFER_SIZE;
        counter++;
}
```

Consumidor

```
while (true) {
        while (counter == 0)
                ; /* do nothing */
        next_consumed = buffer[out];
        out = (out + 1) % BUFFER_SIZE;
         counter--;
        /* consume the item in next consumed */
}
```

Universidade de Brasília

Faculdade UnB Gama

Condição Corrida

- **`counter++`** poderia ser implementando como

```
register1 = counter
register1 = register1 + 1
counter = register1
```

- **`counter--`** poderia ser implementando como

```
register2 = counter
register2 = register2 - 1
counter = register2
```

- Considere que essa intercalação de execução com "count = 5" inicialmente:

S0: producer execute `register1 = counter` {register1 = 5}
S1: producer execute `register1 = register1 + 1` {register1 = 6}
S2: consumer execute `register2 = counter` {register2 = 5}
S3: consumer execute `register2 = register2 - 1` {register2 = 4}
S4: producer execute `counter = register1` {counter = 6 }
S5: consumer execute `counter = register2` {counter = 4}

Universidade de Brasília

Faculdade UnB Gama

Problema da Seção-Crítica

- Conside sistema de ***n*** processos {p_0, p_1, … p_{n-1}}
- Cada processo tem o segmento de código de **seção-crítica**
  - Processos podem sofrer mudanças nas variáveis comuns, atualizando tabelas, escrevendo arquivos, etc
  - Quando um processo está na seção-crítica, nenhum outro poderá estar em sua respectiva seção-crítica
- ***Problema de seção-crítica*** é projetar/desenhar um protocolo para resolver isso.
- Cada processo deve pedir permissão para entrar na seção-crítica na **entry section (seção de entrada)**, pode sair da seção-crítica com a **exit section (seção de saída)**

Universidade de Brasília

Faculdade UnB Gama

Seção Crítica

- Estrutura geral do processo P_i

```
do {
    entry section
        critical section
    exit section
        remainder section
} while (true);
```

Algoritmo para o Processo P_i

```
do {

    while (turn == j);

            critical section

    turn = j;

            remainder section

} while (true);
```

Solução para Problema da Seção-Crítica

1. **Mutual Exclusion (Exclusão Mútua) -** Se o processo P_i está sendo executado em sua seção crítica, então nenhum outro processo pode executar em suas respectivas seções-críticas.
2. **Progress (Progresso) –** Se nenhum processo está sendo executado em sua seção-crítica e existem alguns processos que desejam entrar nas suas respectivas seções-críticas, então a seleção dos processos que irão entrar na seção-crítica a seguir não poderá ser adiada indefinidamente
3. **Bounded Waiting (Espera Limitada) –** Um limite deve existir no número de vezes que outros processos são permitidos a entrar nas suas seções-críticas depois que um processo solicitou entrar na seção-crífica e antes dessa requisição ser garantida.
   - Assumir que cada processo executa a um velocidade diferente de 0
   - Nenhuma suposição relativa à **velocidade relativa** dos ***n*** processos

Universidade de Brasília

Faculdade UnB Gama

Manipulação da Seção-Crítica no Sistema Operacional

Duas abordagens dependendo de que se o Kernel é preemptivo ou não preemptivo

- **Preemptive (Preemptivo) –** permite preempção do processo quando executando no modo kernel
- **Non-preemptive (Não preemptivo)** – funciona até sair do modo Kernel, bloquear ou render voluntariamente CPU
  - Essencialmente livre de condições de corrida no modo kernel

Solução de Peterson

- Boa descrição algorítmica da resolução do problema
- Solução em dois processos:
- Assumir que **`load`** e **`store`** instruções de linguagem de máquina são atômicas; isto é, não podem ser interrompidos
- Os dois processos compartilham duas variáveis:
  - **`int turn;`**
  - **`Boolean flag[2]`**
- A variável **`turn`** indica de quem é a vez de entrar na seção-crítica
- O array **`flag`** é usado para indicar se um processo está apto/pronto para entrar na seção crítica. **`flag[i] = true`** implica que o processo P_i está pronto!

Universidade de Brasília

Faculdade UnB Gama

Solução de Peterson(Cont.)

- Provável que três requisitos de CS (seção-crítica) sejam atendidos:
  1. A Exclusão Mútua é preservada:

     $\mathbf{P_i}$ entra na CS somente se:

     quer **`flag[j] = false`** ou **`turn = i`**
  2. Requisito de Progresso é satisfeito
  3. O requisito de Espera Limitada é atendido

Universidade de Brasília

Faculdade UnB Gama

Algoritmo do Processo P_i

```
do {
    flag[i] = true;
    turn = j;
    while (flag[j] && turn = = j);
            critical section
    flag[i] = false;
            remainder section
 } while (true);
```

Sincronização por Hardware

- Muitos sistemas fornecem hardware com suporte para a implementação do código da seção-crítica
- Todas soluções abaixo se baseiam na ideia de locking **(bloqueio)**
  - Regiões críticas protegidas via bloqueios
- Uniprocessadores – poderiam desabilitar interrupções
  - O código atualmente em execução seria executado sem preempção
  - Geralmente muito ineficiente em sistemas multiprocessadores
    - Sistemas operacionais que usam isso não são amplamento escalonáveis
- Máquinas modernas fornecem instruções atômicas especiais de hardware
    - **Atômico** = não-interruptível
  - Teste a palavra memória e defina um valor
  - Ou troque o conteúdo de duas palavras de memória

Solução para o Problema da Seção-Crítica usando Bloqueios

```
do {
        acquire lock
                critical section
        release lock
                remainder section
} while (TRUE);
```

test_and_set Instrução

Definição:

```
boolean test_and_set (boolean *target)
   {
        boolean rv = *target;
        *target = TRUE;
        return rv:
   }
```

1. Executado atomicamente
2. Returna o valor original do parâmetro passado
3. Define um valor do parâmetro passado para “VERDADEIRO (TRUE)”

Universidade de Brasília

Faculdade UnB Gama

Solução usando test_and_set()

- Bloqueio de variável boolean compartilhada, inicializada como FALSE
- Solução:

```
do {
    while (test_and_set(&lock))
        ; /* do nothing */

            /* critical section */

    lock = false;

            /* remainder section */

 } while (true);
```

Universidade de Brasília

Faculdade UnB Gama

compare_and_swap Instrução

Definição:

```
int compare _and_swap(int *value, int expected, int new_value) {
    int temp = *value;

    if (*value == expected)
       *value = new_value;
 return temp;
}
```

1. Executado atomicamente
2. Returna o valor original do parâmetro passado “value”
3. Define a variável “value” o valor do parâmetro passado “new_value” mas somente se “value” == “expected”. Ou seja, a troca ocorre somente sob essa condição.

Solução usando compare_and_swap

- “Lock” inteiro compartilhado inicializado com 0;
- Solução:

```
do {
     while (compare_and_swap(&lock, 0, 1) != 0)
      ; /* do nothing */
   /* critical section */
 lock = 0;
   /* remainder section */
} while (true);
```

Bounded-waiting (Espera Limitada) Mutual Exclusion (Exclusão Mútua) com test_and_set

```
do {
   waiting[i] = true;
   key = true;
   while (waiting[i] && key)
      key = test_and_set(&lock);
   waiting[i] = false;
   /* critical section */
   j = (i + 1) % n;
   while ((j != i) && !waiting[j])
      j = (j + 1) % n;
   if (j == i)
      lock = false;
   else
      waiting[j] = false;
   /* remainder section */
} while (true);
```

Universidade de Brasília

Faculdade UnB Gama

Bloqueios Mutex

- Soluções anteriores são complicadas e geralmente inacessíveis para programadores.
- Os designers de Sistemas Operacionais construiram ferramentas de software para resolver problema da seção-crítica
- O mais simples é o bloqueio mutex
- Proteger uma seção crítica primeiro **`acquire()`** um bloqueio, depois **`release ()`** o bloqueio
  - Variável boolean indicando se o bloqueio está disponível ou não
- Chamadas para **`acquire()`** e **`release()`** deve ser atômicas.
  - Geralmente implementado via instruções de hardware atômicas.
- Mas essa solução requer **busy waiting**
  - Esse bloqueio portanto, chamado de **spinlock**

Universidade de Brasília

Faculdade UnB Gama

acquire() e release()

```
acquire() {
    while (!available)
        ; /* busy wait */
    available = false;;
}
```

```
release() {
    available = true;
}
```

```
do {
    acquire lock
        critical section
    release lock
      remainder section
} while (true);
```

Universidade de Brasília

Faculdade UnB Gama

Semáforos

- Ferramentas de sincronização fornecem mais maneiras sofisticadas (que bloqueios Mutex) para o processo de sincronizar suas atividades.
- Semáforo **S** – variável inteira
- Só pode ser acessado por meio de duas operações indivisíveis (atômicas)
  - **`wait()`** and **`signal()`**
    - Originalmente chamada **`P()`** e **`V()`**
- Definição de **`wait()` operation**

```
wait(S) {
    while (S <= 0)
        ; // busy wait
    S--;
}
```

- Definição de **`signal()` operation**

```
signal(S) {
    S++;
}
```

Universidade de Brasília

Faculdade UnB Gama

Uso de Semáforos

- **Counting semaphore (semáforo contador)** – valor inteiro que pode variar em um domínio irrestrito.
- **Binary semaphore (semáforo binário)** valor inteiro com intervalo somente entre 0 e 1
  - Igual ao **mutex lock (bloqueio mutex)**
- Pode resolver vários problemas de sincronização
- Considere P_1 e P_2 que requer S_1 acontecer antes de S_2

  Criar o semáforo "**synch**" inicializado com 0

```
P1:
    S1;
    signal(synch);
P2:
    wait(synch);
    S2;
```

- Pode implementar um semáforo contador ***S*** como um semáforo binário

Implementação de Semáforos

- Deve garantir que os dois processos não podem executar o **`wait()`** e **`signal()`** no mesmo semáforo e ao mesmo tempo
- Portanto, a implementação torna-se um problema de seção crítica quando o código **`wait`** e **`signal`** são colocados numa seção crítica.
  - Poderia agora haver **busy waiting (espera ocupada)** numa implementação de seção crítica
    - Porém o código de implementação é curto
    - Pequena **busy waiting** – a seção crítica raramente ocupada
- Note que algumas aplicações podem gastar muito tempo na seção-crítica e sendo assim essa não é uma boa solução

Universidade de Brasília

Faculdade UnB Gama

Implementação do Semáforo sem Busy waiting

- Para cada semáforo, há uma fila de espera associada
- Cada entrada na fila de espera tem dois itens de dados:
  - Valor (do tipo inteiro)
  - Ponteiro para o próximo registro (nó) na lista
- Duas operações:
  - **block** – coloca o processo invocando as operações na fila apropriada de espera
  - **wakeup –** remove um dos processos da fila de espera e coloca ele na fila dos processos prontos
- ```
  typedef struct{
  int value;
  struct process *list;
  } semaphore;
  ```
  ```

Implementação do Semáforo sem Busy waiting (Cont.)

```
wait(semaphore *S) {
    S->value--;
    if (S->value < 0) {
        add this process to S->list;
        block();
    }
}

signal(semaphore *S) {
    S->value++;
    if (S->value <= 0) {
        remove a process P from S->list;
        wakeup(P);
    }
}
```

Universidade de Brasília

Faculdade UnB Gama

Deadlock e Starvation

- **Deadlock (Impasse)** – dois ou mais processos estão esperando indefinidamente para um evento que pode ser disparado somente por um dos processos de espera
- Sejam ***S*** e ***Q*** dois semáforos inicializados por 1

P_0	P_1
`wait(S);`	`wait(Q);`
`wait(Q);`	`wait(S);`
`...`	`...`
`signal(S);`	`signal(Q);`
`signal(Q);`	`signal(S);`

- **Starvation** – **indefinite blocking(bloqueio indefinido)**
  - Um processo nunca pode ser removido da fila de semáforos considerando que ele está suspenso
- **Priority Inversion (Inversão de prioridades)** – Problema de agendamento quando processo de baixa prioridade aguarda um bloqueio necessário por um processo de alta prioridade
  - Resolvido via **priority-inheritance protocol (protocolo de hierarquia de prioridades)**

Problemas Clássicos de Sincronização

- Problemas clássicos usados para testar esquema de sincronização recém-propostos
  - Bounded-Buffer Problem (Problema do Buffer Limitado)
  - Readers and Writers Problem (Problema de Leitores e Escritores)
  - Dining-Philosophers Problem (Problema dos Filósofos)

Universidade de Brasília

Faculdade UnB Gama

Bounded-Buffer Problem (Problema do Buffer Limitado)

- ***n*** buffers, cada um pode conter um item
- Semáforos **`mutex`** inicializados com o valor 1
- Semáforos **`full`** inicializados com o valor 0
- Semáforos **`empty`** inicializados com o valor n

Universidade de Brasília

Faculdade UnB Gama

Bounded-Buffer Problem (Problema do Buffer Limitado)(Cont.)

- A estrutura do processo produtor

```
do {
        ...
        /* produce an item in next_produced */
        ...
    wait(empty);
    wait(mutex);
        ...
        /* add next produced to the buffer */
        ...
    signal(mutex);
    signal(full);
} while (true);
```

Universidade de Brasília

Faculdade UnB Gama

Bounded-Buffer Problem (Problema do Buffer Limitado)(Cont.)

- A estrutura do processo produtor

```
do {
    wait(full);
    wait(mutex);
       ...
    /* remove an item from buffer to next_consumed */
       ...
    signal(mutex);
    signal(empty);
       ...
    /* consume the item in next consumed */
       ...
} while (true);
```

Problema de Leitores e Escritores

- Um conjunto de dados é compartilhado entre um número de processos concorrentes
  - Readers (Leitores) – leia apenas o conjunto de dados; eles ***não*** podem realizar qualquer alteração
  - Writers (Escritores) – podem tanto ler como escrever
- Problema – permitir que múltiplos leitores a ler no mesmo tempo
  - Apenas um único escritor pode acessar os dados compartilhados ao mesmo tempo
- Várias variações dos leitores e escritores são consideradas – todas envolvem alguma forma de prioridades
- Dados compartilhados
  - Conjunto de dados
  - Semáforos **`rw_mutex`** inicializados em 1
  - Semáforos **`mutex`** inicializados em 1
  - Inteiros**`read_count`** inicializados em 0

Problema de Leitores e Escritores (Cont.)

- A estrutura do processo escritor:

```
do {
        wait(rw_mutex);
           ...
        /* writing is performed */
           ...
     signal(rw_mutex);
} while (true);
```

Problema de Leitores e Escritores (Cont.)

- A estrutura do processo leitor:

```
do {
        wait(mutex);
        read_count++;
        if (read_count == 1)
        wait(rw_mutex);
    signal(mutex);

        ...
        /* reading is performed */
        ...

    wait(mutex);
        read_count--;
        if (read_count == 0)
    signal(rw_mutex);
    signal(mutex);
} while (true);
```

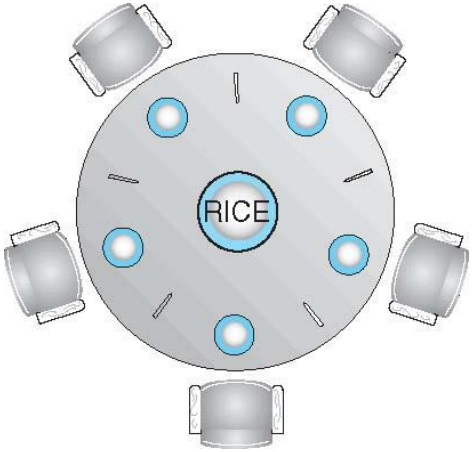
Universidade de Brasília

Faculdade UnB Gama

Variação de Problemas de Leitores e Escritores

- ***Primeira*** variação – nenhum leitor fica esperando, exceto se o escritor tenha tido permissão para usar o objeto compartilhado
- ***Segunda*** variação – uma vez que o escritor está pronto, ele executa a gravação o mais breve possível
- Ambos podem ter **starvation** levando a mais variações
- Problema é resolvido em alguns sistemas pelo fornecimento de bloqueios de leitores-escritores do kernel

Dining-Philosophers Problem (Jantar dos Filósofos)



- Filósofos passam suas vidas alterando entre pensar e comer
- Não interagem com seus vizinhos, ocasionalmente tentam pegar dois pauzinhos (um de cada vez) para comer na tijela
  - Precisam de ambos para comer, e depois soltam ambos quando finalizam a refeição
- No caso de 5 filósofos
  - Dados compartilhados
    - Tijela de arroz (conjunto de dados)
    - Semáforo ‘pauzinho[5]’ inicializado com 1

Dining-Philosophers Problem Algorithm (Problema do Filósofo – Algoritmo)

- A estrutura do filósofo *i*:

```
do {
    wait (chopstick[i] );
     wait (chopStick[ (i + 1) % 5] );

                //  eat

     signal (chopstick[i] );
     signal (chopstick[ (i + 1) % 5] );

                //  think

} while (TRUE);
```

- Qual é o problema com esse algoritmo?

Universidade de Brasília

Faculdade UnB Gama

Dining-Philosophers Problem Algorithm (Problema do Filósofo – Algoritmo)(Cont.)

- Manipulação de deadlock
  - Permite no máximo 4 filósofos estejam sentados à mesa simultaneamente.
  - Permite um filósofo pegar os gafos somente se eles estiverem disponíveis (colheita deve ser feita numa seção-crítica)
  - Uso de uma solução assimétrica – um filósofo de número ímpar pega o primeiro pauzinho esquerdo e depois o direito. Já o filósofo de número esquerdo ocorre o inverso: primeiro pega o pauzinho direito e depois o esquerdo.

Problemas com Semáforos

- Uso incorreto das operações de semáforo:
  - signal (mutex) …. wait (mutex)
  - wait (mutex) … wait (mutex)
  - Omissão de ***wait (mutex)*** ou ***signal (mutex)*** (ou ambos)
- Deadlock e starvation são possíveis.

Universidade de Brasília

Faculdade UnB Gama

Monitores

- Um alto nível de abstração que fornece mecanismos convenientes e eficazes para sincronização de processos
- *Tipo de dado abstrato,* variáveis internas somente acessíveis por código dentro de um procedure (procedimento)
- Apenas um processo de cada vez pode ser ativado dentro de um monitor
- Porém não é poderoso suficientes para modelar alguns esquemas de sincronização

```
monitor monitor-name
{
  // shared variable declarations
  procedure P1 (…) { …. }

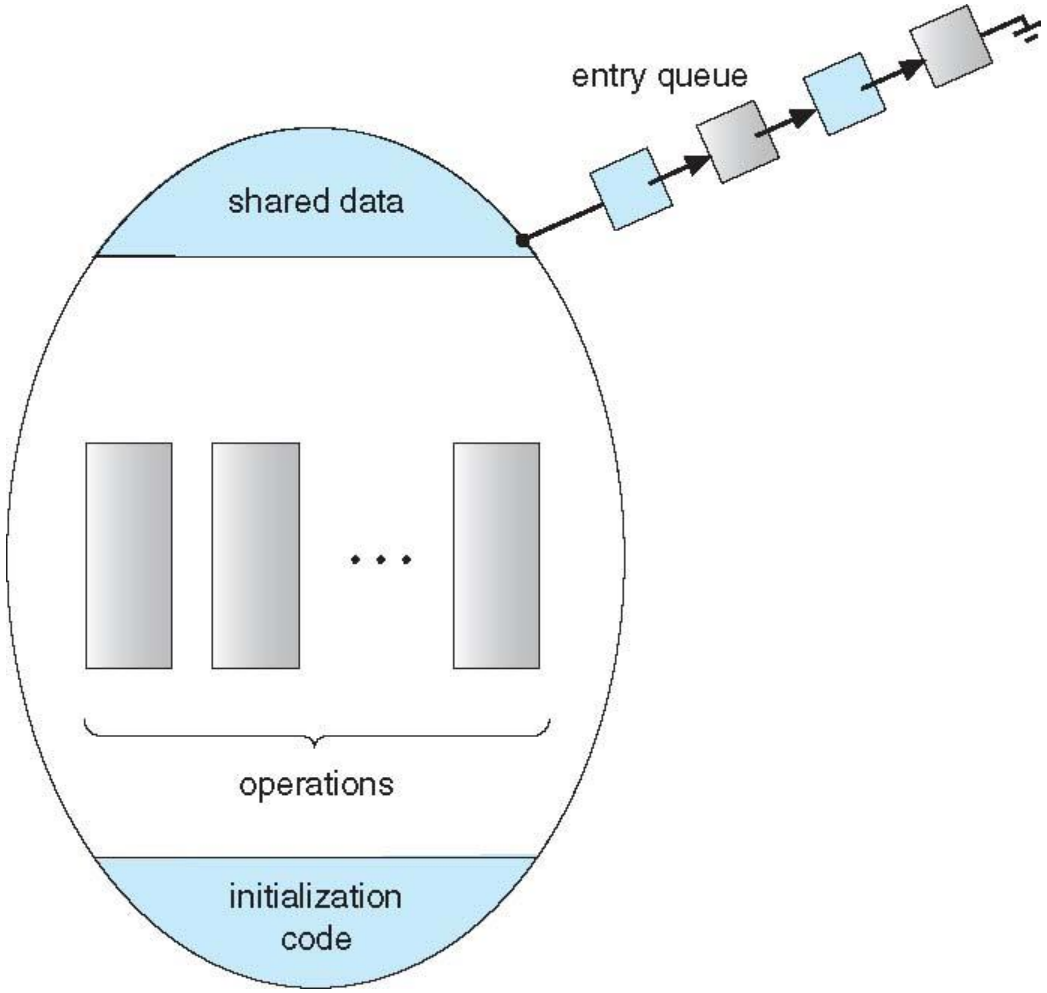
  procedure Pn (…) {……}

    Initialization code (…) { … }
  }
}
```

Universidade de Brasília

Faculdade UnB Gama

Vista Esquemática de um Monitor



Universidade de Brasília

Faculdade UnB Gama

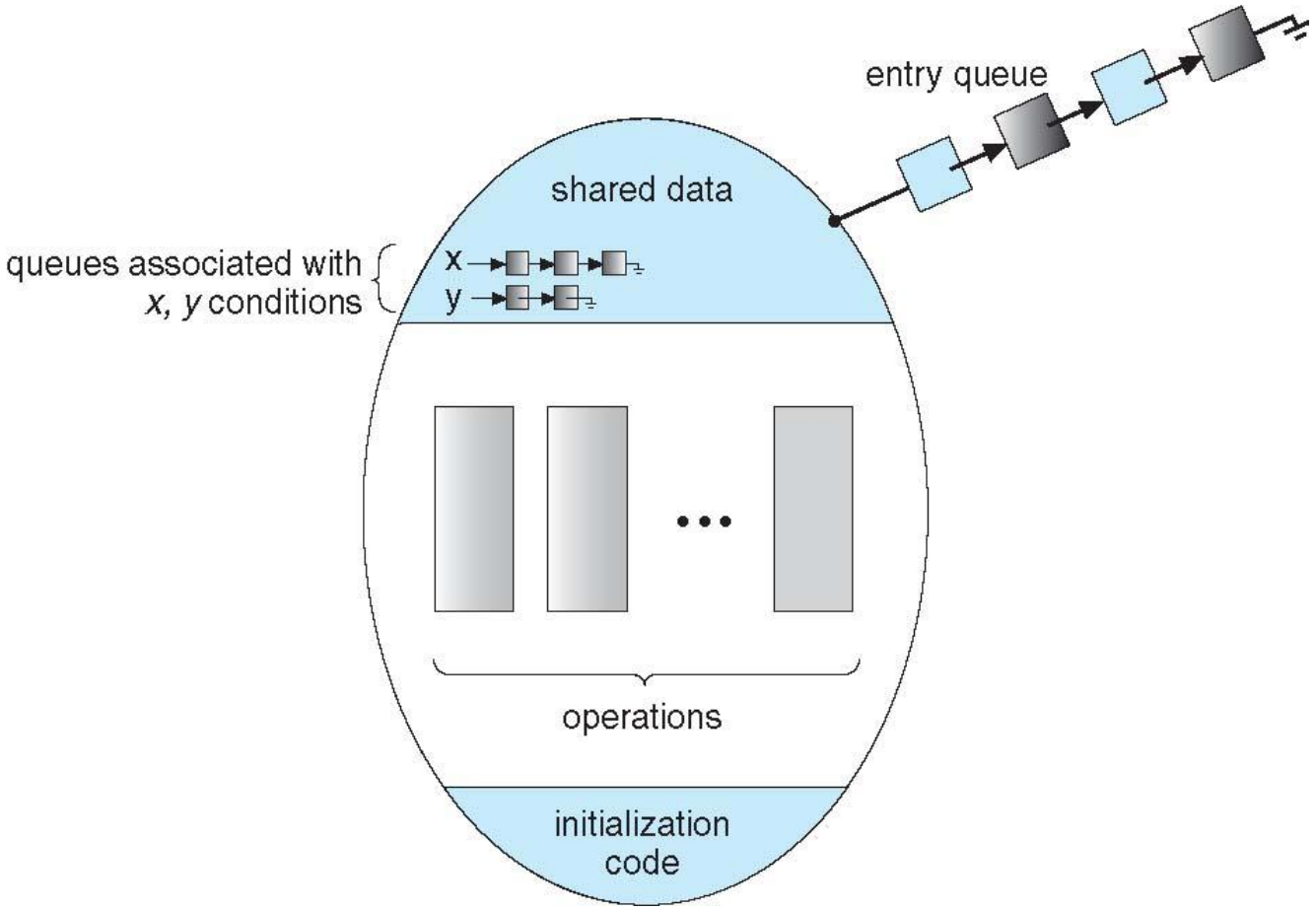
Variáveis de Condição

- **`Condição x, y;`**
- Duas operações são permitidas em variável de condição:
  - **`x.wait()`** – um processo que invoca a operação, é suspenso até **`x.signal()`**
  - **`x.signal()`** – retoma um dos processos (se houver) que invocou **`x.wait()`**
    - If no **`x.wait()`** on the variable, then it has no effect on the variable
    - Se não **`x.wait()`** na variável, então não tem efeito sobre a variável

Universidade de Brasília

Faculdade UnB Gama

Monitor com Variáveis de Condição



Universidade de Brasília

Faculdade UnB Gama

Opções de Variáveis de Condição

- Se o processo P invoca **`x.signal()`** e o processo Q está suspenso em **`x.wait()`**, o que acontecerá a seguir?
  - Tanto Q e P não poderá ser executado em paralelo. Se Q é retomado, então P deve esperar.
- As opções incluem:
  - **Signal and wait (sinalize e espere) –** P aguarda Q deixar o monitor ou outra condição
  - **Signal and continue (sinalize e continue) –** Q aguarda P deixa o monitor ou outra condição
  - Ambos possuem prós e contras – o implementar decide qual usar
  - Monitores implementados em Pascal Concorrente
    - P executa o sinal e imediatamente sairá do monitor, Q será retomado
  - Implementação em outras linguagens de programação incluindo Mesa, C#, Java

Universidade de Brasília

Faculdade UnB Gama

Solução do Monitor para Dining Philosophers (Problema dos Filósofos)

```
monitor DiningPhilosophers
{
   enum { THINKING; HUNGRY, EATING) state [5] ;
   condition self [5];

   void pickup (int i) {
          state[i] = HUNGRY;
          test(i);
          if (state[i] != EATING) self[i].wait;
}

   void putdown (int i) {
          state[i] = THINKING;
                  // test left and right neighbors
           test((i + 4) % 5);
           test((i + 1) % 5);
}
```

Universidade de Brasília

Faculdade UnB Gama

Solução para Dining Philosophers Problema dos Filósofos - (Cont.)

```
    void test (int i) {
            if ((state[(i + 4) % 5] != EATING) &&
            (state[i] == HUNGRY) &&
            (state[(i + 1) % 5] != EATING) ) {
                 state[i] = EATING ;
             self[i].signal () ;
            }
    }

        initialization_code() {
           for (int i = 0; i < 5; i++)
           state[i] = THINKING;
         }
}
```

Solução para Dining Philosophers Problema dos Filósofos - (Cont.)

- Cada filósofo *i* invoca a operação **`pickup()`** e **`putdown()`** na seguinte sequência:

```
DiningPhilosophers.pickup(i);

    EAT

DiningPhilosophers.putdown(i);
```

- Sem deadlock, porém **starvation** é possível

Implementação de Monitor Usando Semáforos

- Variáveis

```
semaphore mutex;  // (initially  = 1)
semaphore next;   // (initially  = 0)
int next_count = 0;
```

- Cada procedure (procedimento) ***F*** será substituído por:

```
wait(mutex);
      …
   body of F;
      …
if (next_count > 0)
  signal(next)
else
  signal(mutex);
```

- Mutual exclusion within a monitor is ensured

Universidade de Brasília

Faculdade UnB Gama

Implementação de Monitor – Variáveis de Condição

- Para cada condição ***x***, nós temos:

```
semaphore x_sem; // (initially  = 0)
int x_count = 0;
```

- A operação **x.wait** pode ser implementada como:

```
x_count++;
if (next_count > 0)
   signal(next);
else
   signal(mutex);
wait(x_sem);
x_count--;
```

Universidade de Brasília

Faculdade UnB Gama

Implementação de Monitores (Cont.)

- O operação **`x.signal`** pode ser implementada como:

```
if (x_count > 0) {
   next_count++;
   signal(x_sem);
   wait(next);
   next_count--;
}
```

Universidade de Brasília

Faculdade UnB Gama

Retomando Processos Dentro do Monitor

- Se vários processos enfileirados na condição x, e x.signal() executado, qual deve ser retomado?
- FCFS frequentemente não é adequado
- **conditional-wait** construct of the form x.wait(c)
- Construtor **conditional-wait** da forma x.wait(c)
  - Quando c é um **número prioritário**
  - Processos com baixo número (alta prioridade) está programado para ser o próximo

Universidade de Brasília

Faculdade UnB Gama

Alocação Única de Recursos

- Aloca um único recurso entre processos concorrentes usando números de prioridade que especificam o máximo tempo que um processo usar o recurso

```
R.acquire(t);
    ...
  access the resurce;
    ...

R.release;
```

- Quando R é uma instância do tipo **`ResourceAllocator`**

Universidade de Brasília

Faculdade UnB Gama

Um Monitor que Aloca um Único Recurso

```
monitor ResourceAllocator
{
   boolean busy;
   condition x;
   void acquire(int time) {
            if (busy)
               x.wait(time);
            busy = TRUE;
   }
   void release() {
            busy = FALSE;
            x.signal();
   }
initialization code() {
    busy = FALSE;
   }
}
```

Universidade de Brasília

Faculdade UnB Gama

Exemplos de Sincronização

- Solaris
- Windows
- Linux
- Pthreads

Universidade de Brasília

Faculdade UnB Gama

Sincronização no Solaris

- Implementa uma variedade de bloqueios que suportam multitarefa, multithreading (incluindo threads em tempo real) e multiprocessamento
- Usa **adaptive mutexes (mutexes adaptativos)** para eficiência ao proteger dados de segmentos de códigos curtos
  - Começa com um padrão de bloqueio de semáforo
  - Se o bloqueio for mantido e houver uma thread executando em outra CPU, rotaciona
  - Se o bloqueio for mantido por um segmento de estado não executado, bloqueie e aguarde o sinal de bloqueio sendo liberado
- Usa **variáveis de condição**
- Usa bloqueios de **readers-writers** (leitores-escritores) quando seções mais longas de código precisam de acesso aos dados
- Usa **turnstiles** para ordenar a lista de threads que aguardam para adquirir um bloqueio adaptativo de mutex ou reader-writer lock
  - Turnstiles são thread de bloqueio por bloqueio, não por-objeto
- Prioridade-herança por turnstile dá ao thread em execução a mais alta das prioridades dos threads em seu turnstile

Sincronização no Windows

- Usa máscaras de interrupção para proteger o acesso aos recursos globais em sistemas uniprocessamento
- Usa **spinlocks** em sistemas multiprocessamento
  - Spinlocking-thread nunca será preterita
- Também fornece objetos **dispatcher objects** user-land que podem agir mutexes, semáforos, eventos e temporizadores
  - **Eventos**
    - Um evento age muito como uma variável de condição
  - Temporizador notifica um ou mais threads quando o tempo acaba
  - Dispatcher objects - **signaled-state** (objeto disponível) ou **non-signaled state** (thread irá bloquear)

Sincronização no Linux

- Linux:
  - Antes do kernel Versão 2.6, desabilitava interrupções para implementar seções críticas curtas.
  - Version 2.6 e posteriormente, totalmente preventivo
- Linux fornece:
  - Semáforos
  - Inteiros atômicos
  - Spinlocks
  - Leitores-escritores
- No sistema de única-CPU, spinlocks substituídos por habilitar e desabilitar a preempção do kernel

Pthreads Sinocronização

- Pthreads API é um sistema operacional independente
- Fornece:
  - mutex locks (bloqueios mutex)
  - Variáveis de condição
- As extensões não-portáteis incluem:
  - bloqueio de leitores-escritores
  - spinlocks

Universidade de Brasília

Faculdade UnB Gama

Abordagens Alternativas

- Memória Transacional
- OpenMP
- Linguagens de Programação Funcional

Universidade de Brasília

Faculdade UnB Gama

Memória Transacional

- Uma **memória transacional** é uma sequência de operações de leitura-escrita para a memória que são realizadas automaticamente.

```
void update()
 {
        /* read/write memory */
  }
```

Universidade de Brasília

Faculdade UnB Gama

OpenMP

- OpenMP é um conjunto de diretivas de compilador e a API que suporta programação paralela

```
void update(int value)
 {
      #pragma omp critical
      {
           count += value
      }
  }
```

O código contido dentro da diretiva **`#pragma omp critical`** é tratada como uma seção crítica e atomicamente executado.

Universidade de Brasília

Faculdade UnB Gama

Linguagens de Programação Funcional

- Linguagens de programação funcional oferecem um paradigma diferente do que as linguagens procedurais, pois elas não mantêm o estado
- Variáveis são tratadas como imutáveis e não podem alterar seu estado uma vez que é atribuído um valor
- Há um crescente interesse nas linguagens funcionais tal como Erlang e Scala pela sua abordagem no tratamento dos dados

Universidade de Brasília

Faculdade UnB Gama

O Problema Deadlock

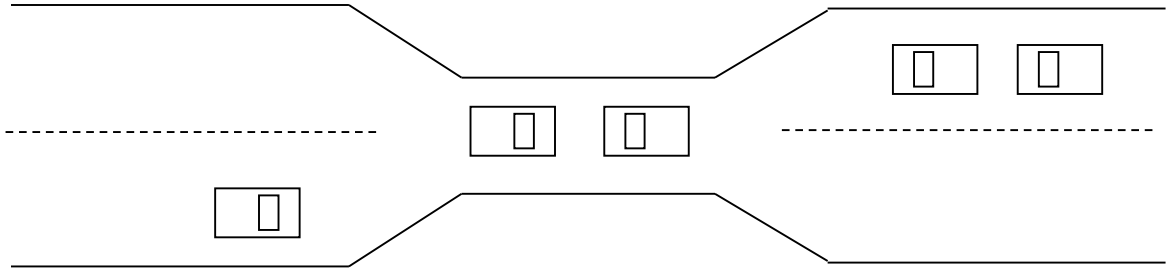
- Um conjunto de processos bloqueados cada um segurando um recurso e aguardando adquirir um recurso mantido por outro processo
- Exemplo
  - Sistemas têm 2 unidades de disco
  - P_1 e P_2 - cada um tem uma unidade de disco e cada um precisa da outra
- Exemplo
  - Semáforo *A* e *B*, inicializado em 1

P_0	P_1
wait (A);	wait(B)
wait (B);	wait(A)

Universidade de Brasília

Faculdade UnB Gama

Exemplo de Travessia de Ponte



- Tráfego apenas numa única direção
- Cada seção da ponte pode ser visto como um recurso
- Se um deadlock ocorrer, ele pode ser resolvido se um dos carros recuar (preempt resources e rollback)
- Vários carros podem precisar recuar se um deadlock ocorrer
- Starvation é possível
- Note – A maioria dos sistemas operacionais não evitam deadlock ou lidam com deadlocks

Exemplo Deadlock

```
/* thread one runs in this function */
void *do_work_one(void *param)
{
   pthread_mutex_lock(&first_mutex);
   pthread_mutex_lock(&second_mutex);
   /** * Do some work */
   pthread_mutex_unlock(&second_mutex);
   pthread_mutex_unlock(&first_mutex);
   pthread_exit(0);
}
/* thread two runs in this function */
void *do_work_two(void *param)
{
   pthread_mutex_lock(&second_mutex);
   pthread_mutex_lock(&first_mutex);
   /** * Do some work */
   pthread_mutex_unlock(&first_mutex);
   pthread_mutex_unlock(&second_mutex);
   pthread_exit(0);
}
```

Examplo Deadlock com Lock Ordering

```
void transaction(Account from, Account to, double amount)
{
   mutex lock1, lock2;
   lock1 = get_lock(from);
   lock2 = get_lock(to);
   acquire(lock1);
      acquire(lock2);
         withdraw(from, amount);
         deposit(to, amount);
      release(lock2);
   release(lock1);
}
```

Transações 1 e 2 são executadas simultaneamente. Transação 1 transfere $25 da conta A para conta B, e Transação 2 transfere $50 da conta B para a conta A

Universidade de Brasília

Faculdade UnB Gama

Características do Deadlock

Deadlocks podem aumentar se quatro condições se mantêm simultanemente.

- **Mutual exclusion**: somente um processo por vez pode usar um recurso
- **Hold and wait**: um processo que mantém pelo menos um recurso aguardando para adquirir recursos adicionais mantidos por outros processos
- **No preemption**: um recurso que pode ser liberado apenas por voluntariamente por outro processo que o mantém, após esse processo ter completado sua tarefa
- **Circular wait**: there exists a set $\{P_0, P_1, \ldots, P_n\}$ of waiting processes such that P_0 is waiting for a resource that is held by P_1, P_1 is waiting for a resource that is held by P_2, …, P_{n-1} is waiting for a resource that is held by P_n, and P_n is waiting for a resource that is held by P_0.
- **Circular wait**: existe um conjunto $\{P_0, P_1, \ldots, P_n\}$ de processos em espera tal como P_0 está aguardando por um recurso que é mantido por P_1, P_1 está aguardando por um recurso que é mantido por P_2, …, P_{n-1} está aguardando um recurso que é mantido por P_n, e P_n está aguardando um recurso que é mantido por P_0.

Universidade de Brasília Faculdade UnB Gama

Gráfico de Alocação de Recursos

Um conjunto de vértices *V* e arestas *E.*

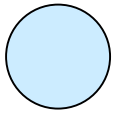
- V está particionado em dois tipos:
  - $P = \{P_1, P_2, \ldots, P_n\}$, o conjunto consistindo de todos os processos no sistema
  - $R = \{R_1, R_2, \ldots, R_m\}$, o conjunto consistindo de todos os tipos de recursos no sistema
- **request edge** – direção aresta $P_i \rightarrow R_j$
- **assignment edge** – direção aresta $R_j \rightarrow P_i$

Universidade de Brasília

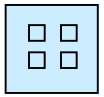
Faculdade UnB Gama

Gráfico de Alocação de Recursos(Cont.)

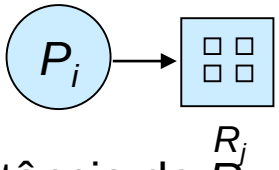
- Processo



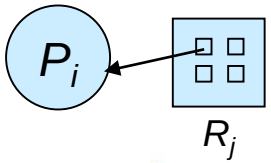
- Tipo de Recurso com 4 instâncias



- P_i solicita instância de R_j



- P_i está aguardando a instância de R_j



Universidade de Brasília

Faculdade UnB Gama

Exemplo de Gráfico de Alocação de Recursos

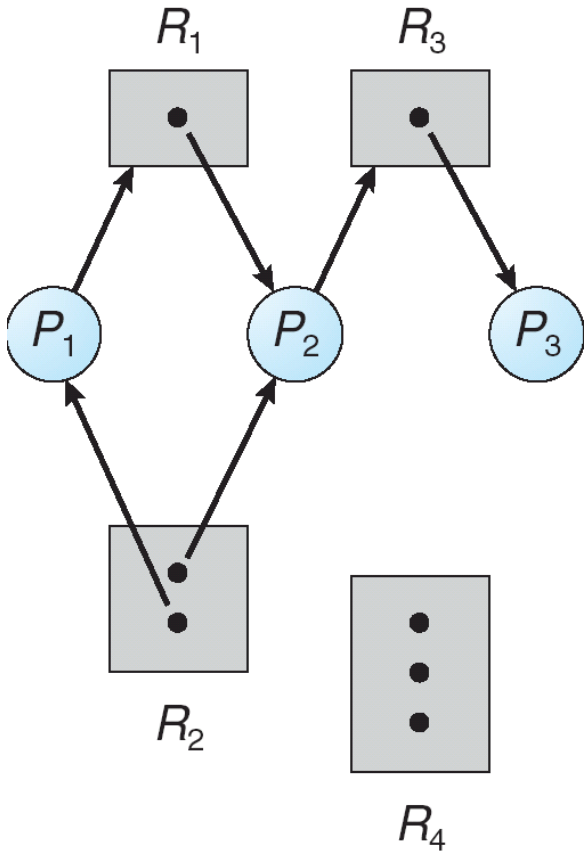


Gráfico de Alocação de Recursos com um Deadlock

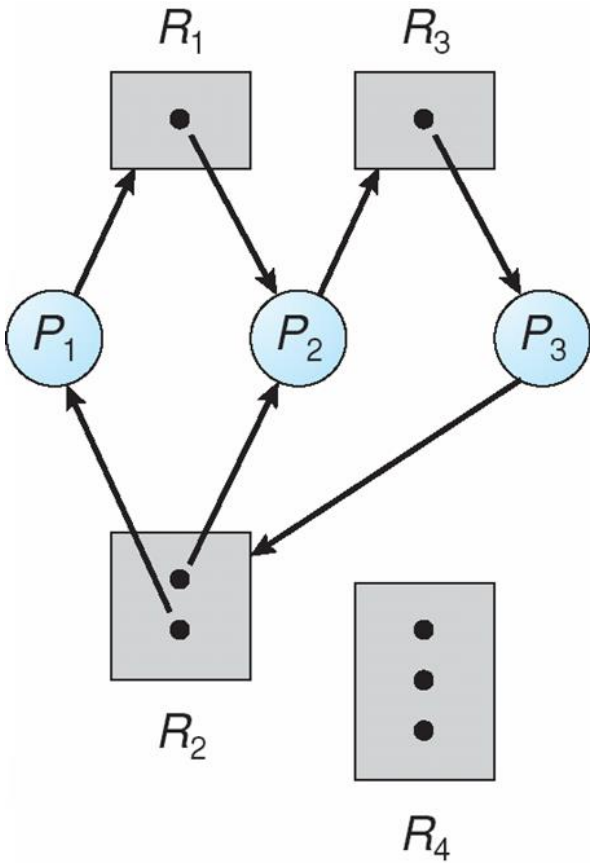
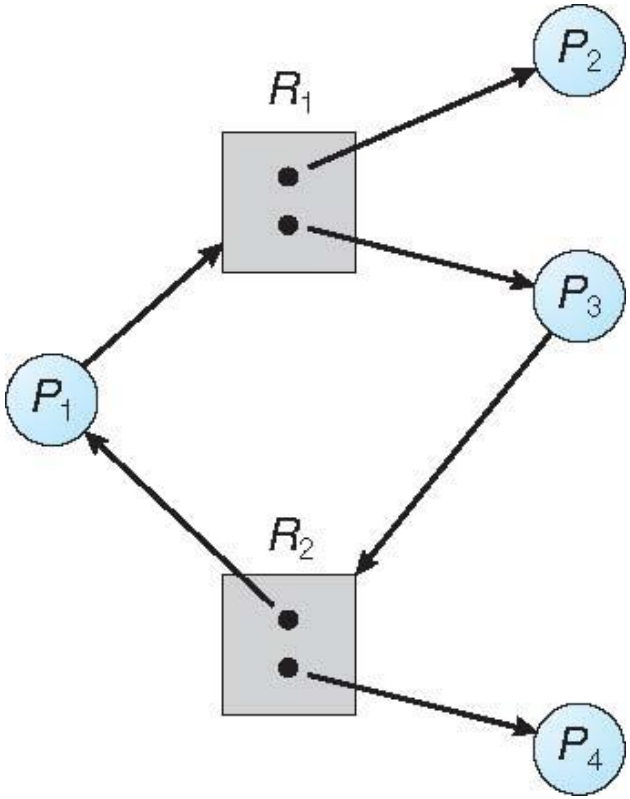


Gráfico com um Círculo porém sem Deadlock



Fatores Básicos

- Se o gráfico não contém círculos ⇒ sem deadlock
- If graph contains a cycle ⇒
- Se o gráfico contém um círculo ⇒
  - Se há uma única instância por tipo de recurso, possível um deadlock
  - Se há várias instâncias por tipo de recursos, possivelmente ocorrerá um deadlock

Métodos para Manipular Deadlocks

- Garantir que o sistema ***nunca*** irá entrar no estado de deadlock:
  - Deadlock prevenção
  - Deadlock evitação
- Permite que o sistema entre num estado de deadlock e depois recupere
- Ignorar o problema e fazer de conta que o deadlock nunca irá ocorrer no sistema; uso na maioria dos sistemas operacionais, incluindo UNIX

Fim do Capítulo 5