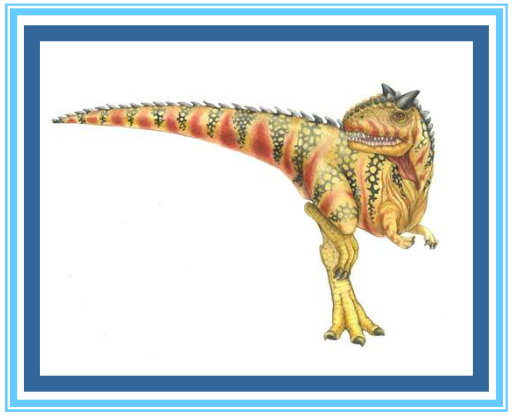
Capítulo 6: Sincronização de Processos



Sobre a apresentação (About the slides)

Os slides e figuras dessa apresentação foram criados por Silberschatz, Galvin e Gagne em 2009. Esse apresentação foi modificada por Cristiano Costa (cac@unisinos.br). Basicamente, os slides originais foram traduzidos para o Português do Brasil.

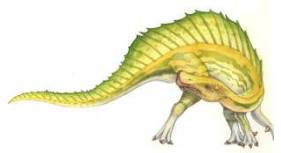
É possível acessar os slides originais em http://www.os-book.com

Essa versão pode ser obtida em http://www.inf.unisinos.br/~cac

The slides and figures in this presentation are copyright Silberschatz, Galvin and Gagne, 2009. This presentation has been modified by Cristiano Costa (cac@unisinos.br). Basically it was translated to Brazilian Portuguese.

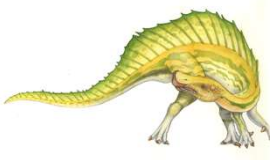
You can access the original slides at http://www.os-book.com

This version could be downloaded at http://www.inf.unisinos.br/~cac



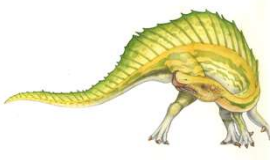
Módulo 6: Sincronização de Processos

- Fundamentos
- O problema das Regiões Críticas
- Solução de Peterson
- Hardware de Sincronização
- Semáforos
- Problemas Clássicos de Sincronização
- Monitores
- Exemplos de Sincronização
- Transações Atômicas



Objetivos

- Introduzir o problema da região crítica, em que as soluções podem ser usadas para garantir a consistência de dados compartilhados
- Apresentar soluções tanto de software quanto de hardware para o problema da região crítica
- Introduzir o conceito de transação atômica e descrever mecanismos de garantir atomicidade

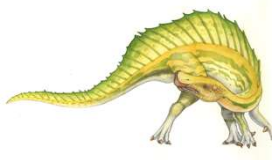


Fundamentos

- Acesso concorrente a dados compartilhados pode resultar em inconsistências.
- Manter a consistência de dados requer a utilização de mecanismos para garantir a execução ordenada de processos cooperantes.
- Suponha que seja desejado fornecer uma solução para o problema do produtor-consumidor que utilize todo o buffer. É possível fazer isso tendo um inteiro count que mantém o número de posições ocupadas no buffer. Inicialmente, count é inicializado em 0. Ele é incrementado pelo produtor após a produção de um novo item e decrementado pelo consumidor após a retirada.

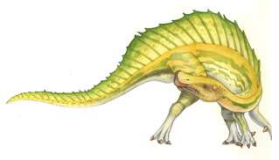
Produtor

```
while (true) {

        /*  produz um item e coloca em nextProduced  */
         while (count == BUFFER_SIZE)
                        ; // do nothing
         buffer [in] = nextProduced;
        in = (in + 1) % BUFFER_SIZE;
        count++;

}
```



Consumidor

```
while (true)  {
        while (count == 0)
              ; // não faz nada
       nextConsumed =  buffer[out];
       out = (out + 1) % BUFFER_SIZE;
      count -- ;

      /*  consome o item em nextConsumed
 }
```



Condição de Corrida

- count++ pode ser implementado como

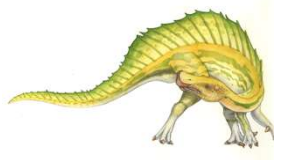
```
register1 = count
register1 = register1 + 1
count = register1
```

- count– pode ser implementado como

```
register2 = count
register2 = register2 - 1
count = register2
```

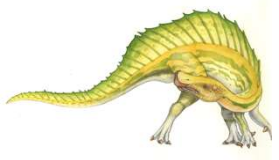
- Considere a seguinte ordem de execução com, inicialmente, “count = 5”:

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



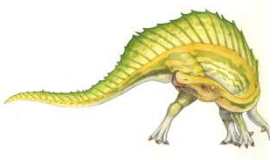
Solução para o Problema da Região Crítica

1. Exclusão Mútua - Se um processo P_i está executando sua região crítica, então nenhuma região crítica de outro processo pode estar sendo executada.
2. Progresso - Se nenhum processo está executando uma região crítica e existem processos que desejam entrar nas regiões críticas deles, então a escolha do próximo processo que irá entrar na região crítica não pode ser adiada indefinidamente.
3. Espera Limitada - Existe um limite para o número de vezes que outros processos são selecionados para entrar nas regiões críticas deles, depois que um processo fez uma requisição para entrar em sua região e antes que essa requisição seja atendida.
   - É assumido que cada processo executa em uma velocidade diferente de zero
   - Nenhuma hipótese é feita referente à velocidade relativa de execução dos N processos.



Tentativas de Solução para a Região Crítica

- Inicialmente será considerada uma solução para dois processos
- É assumido que as instruções de máquina LOAD (carrega) e STORE (armazena) são atômicas; isto é, não podem ser interrompidas.
- Inicialmente são apresentadas duas tentativas e depois a solução

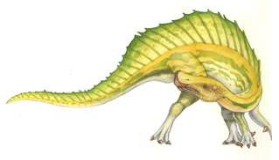


Tentativa 1

- Variáveis compartilhadas:
  - **var** *turn*: (0..1);
    inicialmente *turn* = 0
  - *turn* = *i* $\Rightarrow$ P_i pode entrar na sua região crítica
- Processo P_i

```
do {
    while turn != i
        ; /* não faz nada */
    // REGIÃO CRÍTICA
    turn = j;
    // SEÇÃO RESTANTE
} while (TRUE);
```

- Satisfaz exclusão mútua, mas não progresso.

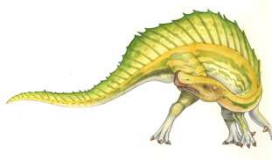


Tentativa 2

- Variáveis compartilhadas:
  - **var** *flag*: **array** [0..1] **of** *boolean*;
    inicialmente *flag* [0] = *flag* [1] = *false*.
  - *flag* [*i*] = *true* $\Rightarrow$ P_i pronto para entrar na região crítica
- Processo P_i

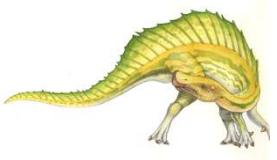
```
do {
    flag[i] = true;
    while flag[j]
        ; /* não faz nada */

        // REGIÃO CRÍTICA

    flag [i] = false;

        // SEÇÃO RESTANTE

} while (TRUE);
```

- Satisfaz exclusão mútua, mas não progresso.



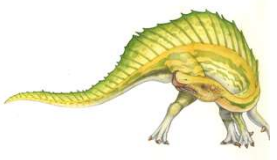
Solução de Peterson

- Os dois processos compartilham duas variáveis:
  - int turn;
  - Boolean flag[2]
- A variãvel turn indica de quem é a vez de entrar na região crítica.
- O vetor flag é usado para indicar se um processo está pronto para entrar na região crítica. flag[i] = true significa que processo P_i está pronto!



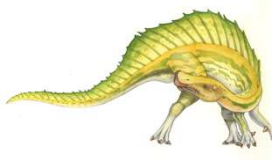
Algoritmo Para Processo P_i

```
do {
    flag[i] = TRUE;
    turn = j;
    while ( flag[j] && turn == j)
          ; /* não faz nada */

        // REGIÃO CRÍTICA

    flag[i] = FALSE;

        // SEÇÃO RESTANTE

} while (TRUE);
```



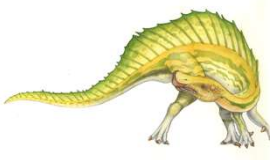
Sincronização por Hardware

- Muitos sistemas fornecem suporte de hardware para código de seção crítica
- Sistemas Monoprocessados – podem desabilitar interrupções
  - Código em execução pode executar sem preempção
  - Geralmente muito ineficiente em sistemas multiprocessados
    - Sistemas Operacionais que usam isso não escalam
- Arquiteturas modernas fornecem instruções atômicas especiais de hardware
    - Atômica = não interrompível
  - Testar uma posição de memória e setar um valor
  - Ou trocar conteúdos de duas posições na memória



Solução para problemas de regisão crítica usando Locks

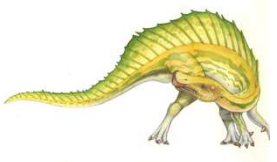
```
do {
    adquire lock
            região crítica
    libera lock
            região restante
} while (TRUE);
```



Instrução TestAndSet

- Definição:

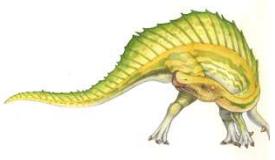
```
boolean TestAndSet (boolean *target)
{
    boolean rv = *target;
    *target = TRUE;
    return rv:
}
```



Solução usando TestAndSet

- Variável booleana compartilhada lock, inicializada em *FALSE*.
- Solução:

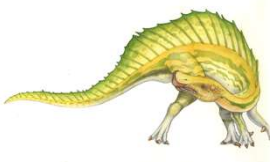
```
while (true) {
        while ( TestAndSet (&lock ))
                ;   /* não faz nada */

                //    REGIÃO CRÍTICA

        lock = FALSE;

                //      SEÇÃO RESTANTE

}
```



Instrução Swap

- Definição:

```
void Swap (boolean *a, boolean *b)
{
        boolean temp = *a;
        *a = *b;
        *b = temp:
}
```



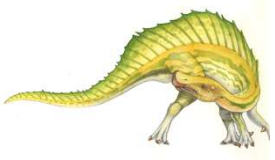
Solução usando Swap

- Variável booleana compartilhada lock, inicializada em *FALSE*; Cada processos tem uma variável booleana *key* local.
- Solução:

```
while (true)  {
        key = TRUE;
        while ( key == TRUE)
              Swap (&lock, &key );

                //    REGIÃO CRÍTICA

         lock = FALSE;

                //      SEÇÃO RESTANTE

 }
```

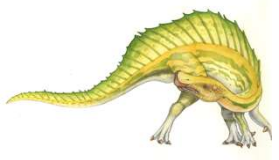
Exclusão Mútua de espera limitada com TestandSet()

```
do {
        waiting[i] = TRUE;
        key = TRUE;
        while (waiting[i] && key)
                key = TestAndSet(&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);
```



Semáforo

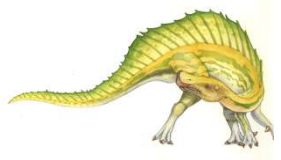
- Ferramenta de sincronização que não requer espera ocupada (busy waiting)
- Semáforo S – variável inteira
- Duas operações padrão modificam S: wait() e signal()
  - Originalmente chamadas P() e V()
- Menos Complicada
- Somente pode ser acessada via duas operações indivisíveis (atômicas)
  - wait (S) {
    while S <= 0
    ; // não faz nada
    S--;
    }
  - signal (S) {
    S++;
    }



Semáforo como uma Ferramenta Geral de Sincronização

- Semáforo Contador – valor nele armazenado pode ser qualquer número inteiro.
- Semáforo Binário – valor inteiro nele armazenado pode variar entre 0 e 1; pode ser implementado mais simplesmente.
  - Também conhecido como mutex locks
- É possível implementar semáforo contador S como um semáforo binário
- Fornece exclusão mútua:

```
    Semaphore mutex;    //  inicializado em 1
do {
    wait (mutex);
        // Região Crítica
    signal (mutex);
        // restante do código
} while (TRUE);
```

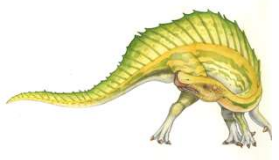


Implementação de Semáforo

- Deve garantir que dois processos não possam executar wait () e signal () no mesmo semáforo ao mesmo tempo

- Daí, a implementação se torna o problema da região crítica na qual o código do wait e signal são colocados em seções críticas.
  - Pode ter espera ocupada na implementação da região crítica
    - Código de implementação é menor
    - Pequena espera ocupada se região crítica está sendo usada raramente

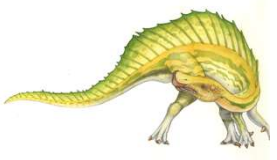
- Observe que aplicações podem perder muito tempo em regiões críticas e daí esta não é uma boa solução.



Implementação de Semáforo sem Espera Ocupada

- Associar uma fila de espera com cada semáforo. Cada entrada na fila de espera tem dois itens:
  - valor (de tipo inteiro)
  - ponteiro para o próximo registro na lista

- Duas operações:
  - block – coloca o processo que evoca a operação na fila de espera apropriada.
  - wakeup – remove um processo da fila de espera e coloca-o na fila de processos prontos (*ready queue*).



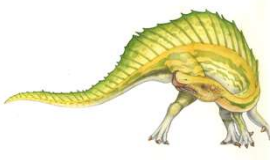
Implementação de Semáforo sem Espera Ocupada (Cont.)

- Implementação de wait:

```
wait(semaphore *S) {
        S->value--;
        if (S->value < 0) {
                adiciona esse processo em S->list;
                block();
        }
}
```

- Implementação de signal:

```
signal(semaphore *S) {
        S->value++;
        if (S->value <= 0) {
                remove o processo P de S->list;
                wakeup(P);
        }
}
```



Deadlock (Impasse) e *Starvation* (Abandono)

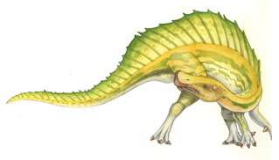
- Deadlock – dois ou mais processos estão esperando indefinidamente por um evento que pode ser causado somente por um dos processos esperando o evento
- Seja S e Q dois semáforos inicializados em 1

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

- Starvation – bloqueio indefinido. Um processo pode nunca ser removido da fila do semáforo em que está suspensa
- Priority Inversion – inversão de prioridade. Problema de escalonamento em que um processo de baixa prioridade mantém um *lock* necessário para um processo de maior prioridade

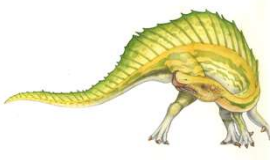
Problemas Clássicos de Sincronização

- Problema do Buffer de tamanho limitado (*Bounded-Buffer*)
- Problema dos Leitores e Escritores
- Problema do Jantar dos Filósofos (*Dining-Philosophers*)



Problema do Buffer de Tamanho Limitado

- *N* posições, cada um pode armazenar um item
- Semáforo mutex inicializado com o valor 1
- Semáforo full inicializado com o valor 0
- Semáforo empty inicializado com o valor N.



Problema do Buffer de Tamanho Limitado (Cont.)

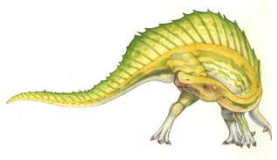
- A estrutura do processo produtor

```
do {

        //   produz um item

    wait (empty);
    wait (mutex);

        //  adiciona o item ao buffer

    signal (mutex);
    signal (full);
} while (TRUE)
```

Problema do Buffer de Tamanho Limitado (Cont.)

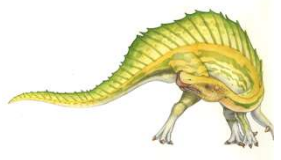
- A estrutura do processo consumidor

```
do {
        wait (full);
        wait (mutex);

            //  remove um item do buffer

        signal (mutex);
        signal (empty);

            //  consome o item removido

} while (TRUE);
```



Problema dos Leitores e Escritores

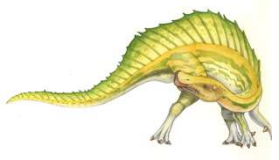
- Um conjunto de dados é compartilhada entre vários processos concorrentes
  - Leitores – somente lê um conjunto de dados; eles não realizam nenhuma atualização
  - Escritores – podem ler e escrever.
- Problema – permitir múltiplos leitores ler ao mesmo tempo. Somente um único escritor pode acessar os dados compartilhados ao mesmo tempo.
- Dados Compartilhados
  - Conjunto de dados
  - Semáforo mutex inicializado em 1.
  - Semáforo wrt inicializado em 1.
  - Inteiro readcount inicializado em 0.



Problema dos Leitores e Escritores (Cont.)

- A estrutura de um processo escritor

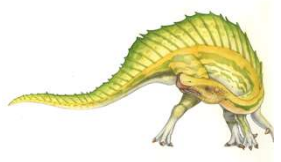
```
do {
        wait (wrt) ;

            //    writing is performed

        signal (wrt) ;
} while (TRUE);
```



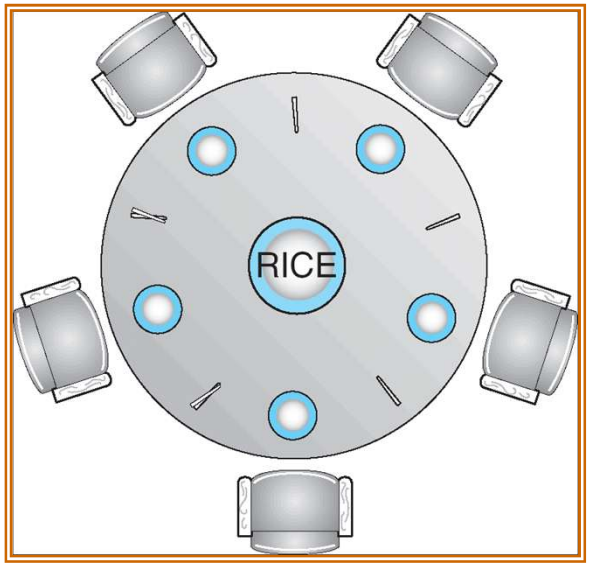
Problema dos Leitores e Escritores (Cont.)

- A estrutura de um processo leitor

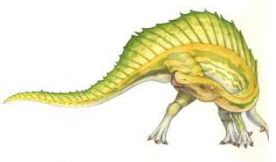
```
do {
        wait (mutex) ;
        readcount ++ ;
        if (readercount == 1)
                        wait (wrt) ;
        signal (mutex)

            // reading is performed

        wait (mutex) ;
        readcount  - - ;
        if (redacount  == 0)
                    signal (wrt) ;
        signal (mutex) ;
} while (TRUE);
```



Problema do Jantar dos Filósofos



- Dados Compartilhados
  - Tigela de Arroz (conjunto de dados)
  - Semáforo chopstick [5] inicializados em 1



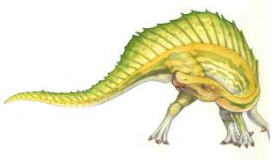
Problema do Jantar dos Filósofos (Cont.)

- A estrutura do Filósofo *i*:

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

            //  comer

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

            //  pensar

} while (TRUE);
```

Problemas com Semáforos

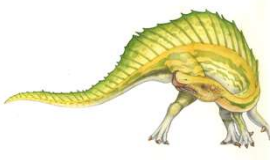
- Uso correto de operações em semáforos:
  - signal (mutex) …. wait (mutex)
  - wait (mutex) … wait (mutex)
  - Omissão de wait (mutex) ou signal (mutex) (ou ambos)



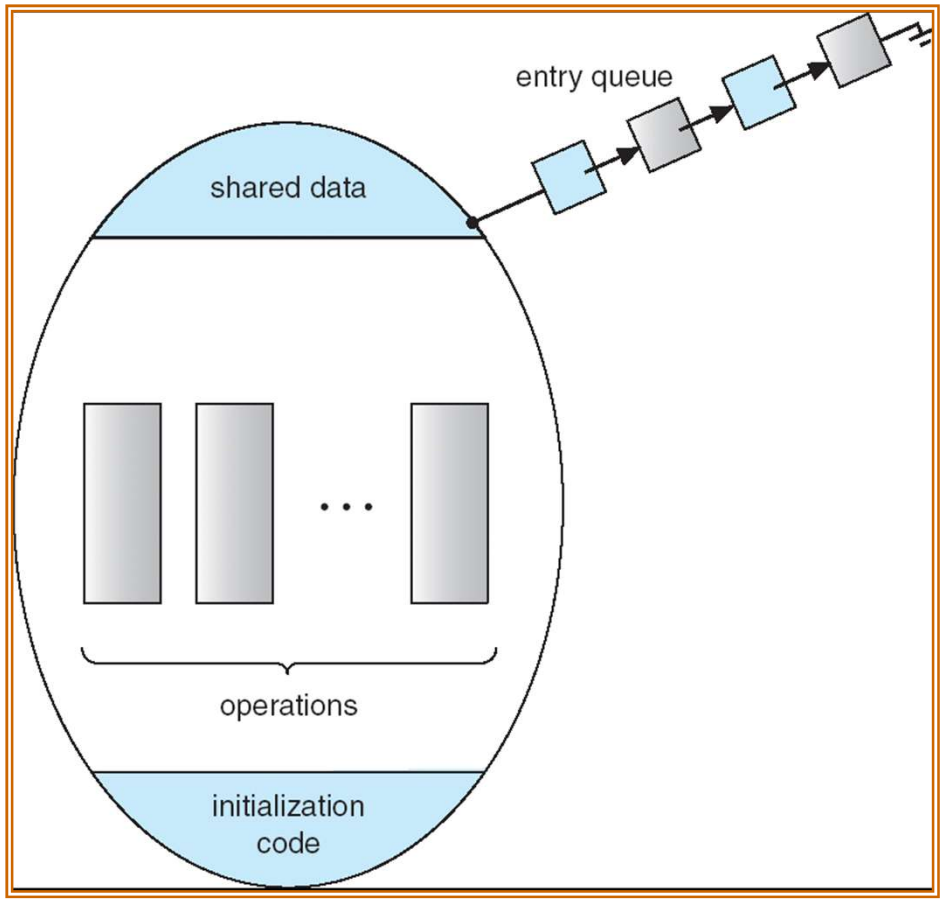
Monitores

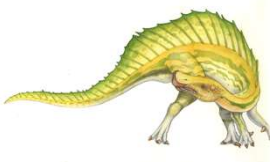
- Abstração de alto nível que fornece um mecanismo conveniente e eficiente para sincronização de processos
- Somente um processo por vez pode estar ativo dentro do monitor

```
monitor nome-monitor
{
    // declaração de variáveis compartilhadas
    procedure P1 (…) { …. }
                …

    procedure Pn (…) {……}

     Código de Inicialização ( ….) { … }
                …
    }
}
```



Visão Esquemática de um Monitor

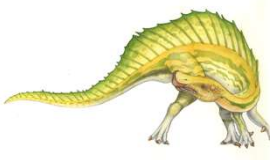




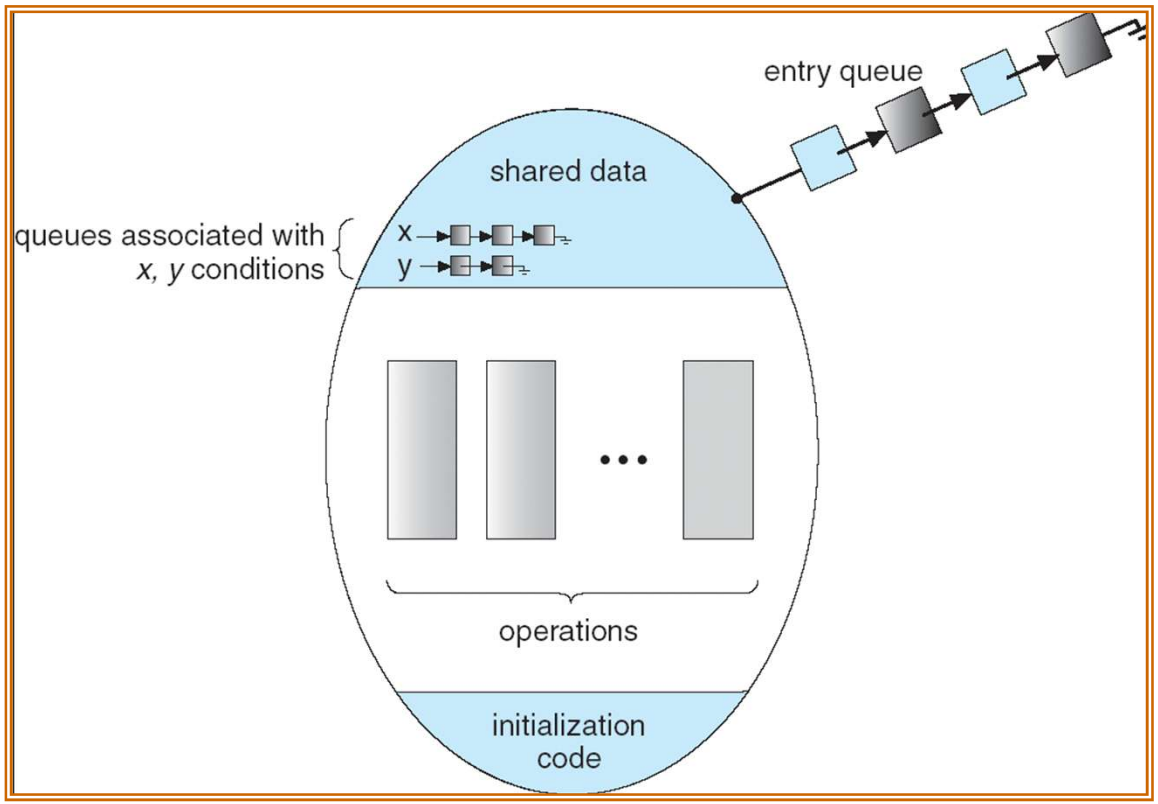
Variáveis Condicionais

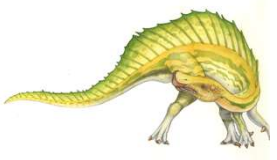
- condition x, y;

- Duas operações em variáveis condicionais:
  - x.wait () – um processo que evoca essa operação é suspenso.
  - x.signal () – reinicia um dos processos (se existe algum) que evocou x.wait ().



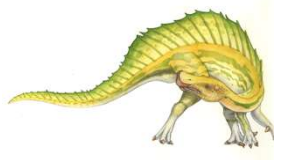
Monitor com Variáveis Condicionais





Solução para o problema dos Filósofos

```
monitor DP
  {
     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);
       }
```



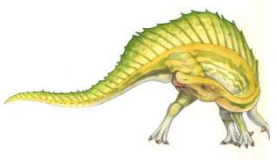
Solução para o problema dos Filósofos (Cont.)

```
        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 o problema dos Filósofos (Cont.)

- Cada filósofo evoca as operações pickup() e putdown() na seguinte seqüência:

```
dp.pickup (i)

    COMER

dp.putdown (i)
```



Implementação de Monitor com Semáforos

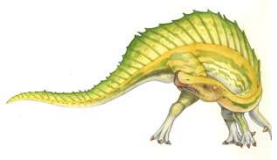
- Variáveis

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

- Cada procedure ***F*** será substituída por

```
wait(mutex);
    …
        corpo de F;

    …
if (next-count > 0)
   signal(next)
else
   signal(mutex);
```

- Exclusão mutua no monitor é garantida.



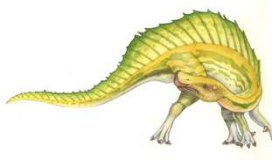
Implementação de Monitor

- Para cada variável condicional ***x***, nós temos:

```
semaphore x-sem; // (inicialmente  = 0)
int x-count = 0;
```

- A operação x.wait pode ser implementada como:

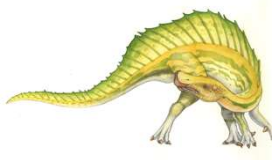
```
x-count++;
if (next-count > 0)
      signal(next);
else
      signal(mutex);
wait(x-sem);
x-count--;
```



Implementação de Monitor (Cont.)

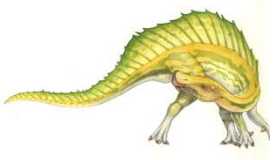
- A operação x.signal pode ser implementada como:

```
if (x-count > 0) {
    next-count++;
    signal(x-sem);
    wait(next);
    next-count--;
}
```



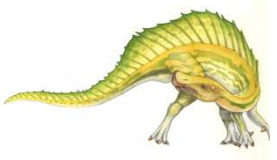
Um Monitor para alocar recursos únicos

```
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;
        }
}
```



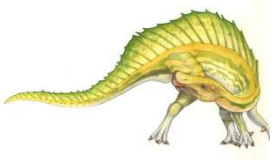
Exemplos de Sincronização

- Solaris
- Windows XP
- Linux
- Pthreads



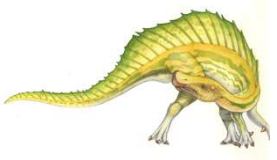
Sincronização no Solaris

- Implementa uma variedade de travas (*locks*) para suportar multitarefa, múltiplas threads (incluindo threads tempo real), e multiprocessamento
- Usa mutex adaptativos para eficiência quando está protegendo dados de segmentos com código curto
- Usa variáveis condicionais e travas leitores-escritores quando seções longas de código necessitam acessar dados
- Usa turnstiles para ordenar a lista de threads esperando para adquirir um mutex adaptativo ou uma trava leitor-escritor



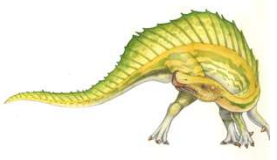
Sincronização no Windows XP

- Usa máscaras de interrupção para proteger acesso aos recursos globais em sistemas monoprocessados
- Usa spinlocks em sistemas multiprocessados
- Também fornece dispatcher objects os quais podem agir como mutexes ou semáforos
- Dispatcher objects podem também fornecer eventos
  - Um evento age de forma parecida com variáveis condicionais



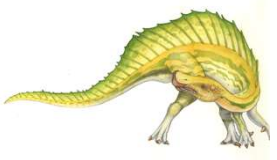
Sincronização no Linux

- Linux:
  - Antes do kernel versão 2.6, desabilita interrupções para implementar seções críticas curtas
  - Versão 2.6 e posterior, totalmente preemptável
- Linux fornece:
  - semáforos
  - *spinlocks*



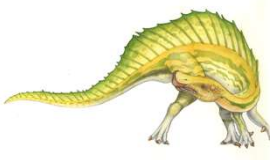
Sincronização em Pthreads

- Pthreads API é independente de SO
- Ela fornece:
  - travas mutex
  - variáveis condicionais
- Extensões não portáveis incluem:
  - travas de leitura-escrita
  - *spinlocks*



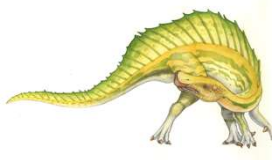
Transações Atômicas

- Modelo de Sistema
- Recuperação baseada em registro de operações (*log*)
- Pontos de Teste (*Checkpoints*)
- Transações Atômicas Concorrentes



Modelo de Sistema

- Garante que operações ocorrem como uma única unidade lógica de trabalho, de forma que é executada inteira ou não é executada
- Relacionada com a área de bancos de dados
- Desafio é garantir atomicidade apesar de defeitos no sistema computacional
- Transação – coleção de instruções ou operações que executam uma única função lógica
  - Aqui a preocupação é com mudanças no meio de armazenamento estável – disco
  - Transação é uma série de operações de leitura e escrita
  - Terminada por uma operação commit (transação terminou normalmente) ou abort (transação falhou)
  - Transações abortadas devem ser desfeitas uma a uma (roll back) de forma a eliminar quaisquer mudanças realizadas



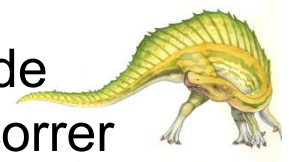
Tipos de Meios de Armazenamento

- Armazenamento Volátil – informação armazenada nele não sobrevive a travamentos do sistema
  - Exemplos: memória principal, cache
- Armazenamento Não-volátil – informação geralmente sobrevive aos travamentos
  - Exemplo: disco e fita
- Armazenamento Estável – informação nunca é perdida
  - Não é realmente possível, aproximações são obtidas via replicação ou sistemas RAID para dispositivos com modos de defeitos independentes

Objetivo é garantir a atomicidade da transação nas quais defeitos causam perda de informação em meios de armazenamento volátil

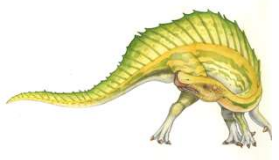
Recuperação Baseada em Registro de Operações (*Log*)

- Armazenar em meio de armazenamento estável informações sobre todas as modificações realizadas por uma transação
- Mais comum é *write-ahead logging* (registro de operação antecipado)
  - Registro de operações em meio de armazenamento estável, cada entrada no registro descreve uma simples operação de escrita na transação, incluindo
    - Nome da Transação
    - Nome do item de dados
    - Valor antigo
    - Novo valor
  - <T_i starts> é escrito no registro de operações quando a transação T_i inicia
  - <T_i commits> é escrito no registro de operações quando a transação T_i termina normalmente
- Entradas no registro de operações devem estar no meio de armazenamento estável antes da operação nos dados ocorrer



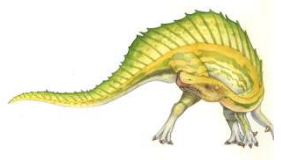
Algoritmo de Recuperação Baseado em Registro de Operações (*Log*)

- Usando o registro de operações, o sistema pode tratar cada erro de memória volátil
  - Undo(T_i) restaura o valor de todos os dados atualizados por T_i
  - Redo(T_i) seta valores de todos os dados na transação T_i para valores novos
- Undo(T_i) e redo(T_i) devem ser idempotentes
  - Múltiplas execuções devem ter o mesmo resultada do que uma execução
- Se o sistema falhar, o estado de todos os dados atualizados deve ser restaurado via registro de operações
  - Se registro de operações contém <T_i starts> sem <T_i commits>, undo(T_i)
  - Se registro de operações contém <T_i starts> e <T_i commits>, redo(T_i)



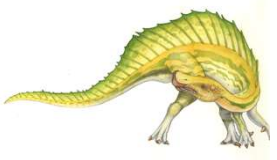
Pontos de Teste (*Checkpoints*)

- Registro de Operações pode se tornar longo e recuperação pode ser demorada
- Pontos de Teste encurtam o registro de operações e o tempo de recuperação.
- Esquema dos Pontos de Teste:
  1. Armazenar todos as entradas do registro de operações atualmente em armazenamento volátil para meio de armazenamento estável
  2. Armazenar todos os dados modificados do meio de armazenamento volátil para o estável
  3. Armazenar uma entrada no registro de operações <checkpoint> no meio de armazenamento estável
- Agora recuperação somente inclui Ti, considerando que Ti iniciou sua execução antes do ponto de teste mais recente, e todas as transações após Ti. Todas outras transações já estão no meio de armazenamento estável



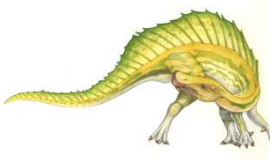
Transações Concorrentes

- Deve ser equivalente a execução seqüencial – execução serial
- Pode realizar todas as transações em uma região crítica
  - Ineficiente, muito restritiva
- Algoritmos de Controle de Concorrência fornecem execução serial



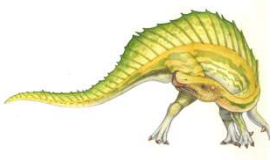
Execução Serial de Transações

- Considere dois itens de dados A e B
- Considere transações T_0 e T_1
- Execute T_0, T_1 atomicamente
- Seqüência de execução é chamada de escalonamento
- Manter a ordem de execução de transações atômicas é chamada de escalonamento serial
- Para N transações, existem N! escalonamentos seriais válidos



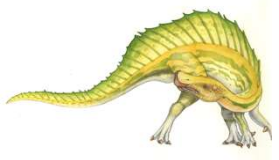
Escalonamento 1: T_0 então T_1

T_0	T_1
read(A)	
write(A)	
read(B)	
write(B)	
	read(A)
	write(A)
	read(B)
	write(B)



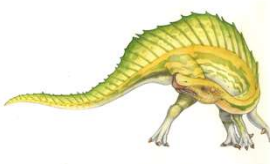
Escalonamento Não-serial

- Escalonamento Não-serial permite execuções sobrepostas
  - Execução resultante não é necessariamente incorreta
- Considere escalonamento S, operações O_i, O_j
  - Conflito existe se o mesmo item de dados for acessado, com pelo menos uma escrita
- Se O_i, O_j são consecutivos e operações ocorrem em diferentes transações & O_i e O_j não conflitam
  - Então S' com ordem trocada O_j O_i é equivalente a S
- Se S pode se tornar S' via troca de operações não conflitantes
  - S permite execução serial com conflito (*conflict serializable*)



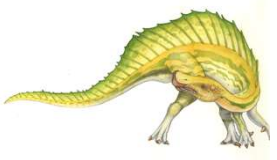
Escalonamento 2: Execução Serial Concorrente

T_0	T_1
read(A)	
write(A)	
	read(A)
	write(A)
read(B)	
write(B)	
	read(B)
	write(B)



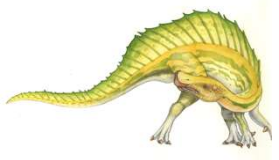
Protocolo de Travamento (*Locking*)

- Garante execução serial associando uma trava com cada item de dado
  - Segui o protocolo de travamento para controle de acesso
- Travas
  - Compartilhada – T_i tem uma trava em modo compartilhado (S) no item Q, T_i pode ler Q mas não pode escrever em Q
  - Particular – T_i tem uma trava em modo exclusivo (X) em Q, T_i pode ler e escrever em Q
- Necessita que cada transação no item Q adquira a trava apropriada
- Se a trava está ocupada, novos pedidos devem esperar
  - Similar ao algoritmo dos leitores e escritores



Protocolo de Travamento em duas fases

- Em Inglês: *Two-phase Locking Protocol*
- Geralmente garante execução serial com conflito (*conflict serializability*)
- Cada transação distribui requisições de travar (*lock*) e destravar (unlock) em duas fases
  - Crescimento – obtendo travas
  - Decrescimento – liberando travas
- Não impede impasse (*deadlock*)

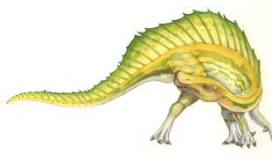


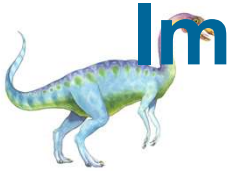
Protocolos baseados em Marcas de Tempo (*Timestamp*)

- Seleciona ordem entre transações com antecedência – ordenação por marca de tempo (*timestamp-ordering)*

- Transação T_i associada com a marca de tempo $TS(T_i)$ antes de T_i iniciar
  - $TS(T_i) < TS(T_j)$ se Ti entrou no sistema antes de T_j
  - TS pode ser gerada pelo *clock* do sistema ou como um contador lógico incrementado a cada entrada na transação

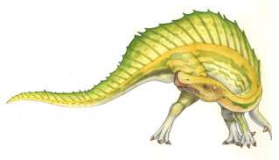
- Marcas de Tempo determinam ordem da execução serial
  - Se $TS(T_i) < TS(T_j)$, sistema deve garantir a produção de escalonamento equivalente ao escalonamento serial no qual T_i aparece antes que T_j





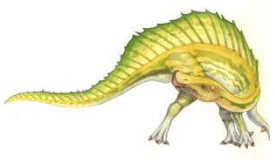
Implementação de Protocolos baseados em Marcas de Tempo

- Item de dados Q obtém duas marcas de tempo (*timestamps*)
  - *W-timestamp*(Q) – maior marca de tempo de qualquer transação que executa write(Q) com sucesso
  - *R-timestamp*(Q) – maior marca de tempo de read(Q) com sucesso
  - Atualizada sempre que read(Q) ou write(Q) são executadas
- Protocolo de Ordenação por Marca de Tempo (*Timestamp-ordering protocol*) garante que quaisquer read e write conflitantes são executados na ordem das marcas de tempo
- Suponha que Ti executa read(Q)
  - Se $TS(T_i) < W\text{-}timestamp(Q)$, T_i necessita ler o valor de Q that que já foi sobrescrito
    - operação read rejeitada e T_i desfeito (*roll back*)
  - Se $TS(T_i) \geq W\text{-}timestamp(Q)$
    - read executado, R-timestamp(Q) alterado para $\max(R\text{-}timestamp(Q), TS(T_i))$



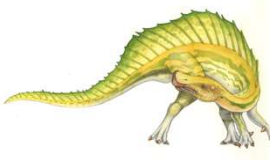
Protocolo de Ordenação por Marca de Tempo

- Suponha que T_i executa write(Q)
  - Se $TS(T_i) < R\text{-}timestamp(Q)$, valor Q produzido por T_i foi necessário previamente e T_i assumiu que ele nunca seria produzido
    - operação write rejeitada, T_i desfeito (*roll back*)
  - Se $TS(T_i) < W\text{-}timestamp(Q)$, T_i tentando escrever valor obsoleto de Q
    - operação write rejeitada e T_i desfeito (*roll back*)
  - Outro caso, write executado
- Qualquer transação T_i desfeita recebe uma nova marca de tempo e reinicia
- Algoritmo garante execução serial de conflito e é livre de impasses (*deadlocks*)



Possível Escalonamento no Protocolo

T_2	T_3
read(B)	
	read(B)
	write(B)
read(A)	
	read(A)
	write(A)



Fim do Capítulo 6