Laboratório de Pesquisa em Redes e Multimídia

Sincronização de Processos (2)

Aula 11

Profa. Patrícia Dockhorn Costa

Universidade Federal do Espírito Santo
Departamento de Informática

Concorrência

- Dificuldades:
  - Compartilhamento de recursos globais.
  - Gerência de alocação de recursos.
  - Localização de erros de programação (depuração de programas).
- Ação necessária:
  - Proteger os dados compartilhados (variáveis, arquivos e outros recursos globais).
  - Promover o acesso ordenado (controle de acesso) aos recursos compartilhados ⇒ *sincronização de processos*.

Condições de Corrida

- *Condições de corrida* são situações onde dois ou mais processos acessam dados compartilhados e o resultado final depende da ordem em que os processos são executados (o que, em última instância, é ditado pelo mecanismo de escalonamento do S.O.). Torna a depuração difícil.
- Condições de corrida são evitadas através da introdução de mecanismos de ***exclusão mútua***. A exclusão mútua garante que somente um processo estará usando os dados compartilhados num dado momento.
- A parte do programa (trecho de código) que em que os dados compartilhados são acessados é denominada de ***região (ou seção) crítica***.

5a. Tentativa

- Na tentativa anterior o processo assinalava a sua intenção de entrar na R.C. sem saber da intenção do outro, não havendo oportunidade dele mudar de idéia depois (i.e., mudar o seu estado para "*false*").
- A 5a. tentativa corrige este problema:
  - Após testar no *loop*, verifica se o outro processo também quer entrar na sua R.C. Em caso afirmativo, o processo com a posse da UCP declina da sua intenção, dando a vez ao parceiro.

5a. Tentativa (cont.)

```
/* Process 0*/:
...
flag[0] = true;
while (flag[1])
{
  flag[0]= false;
  /*delay*/
  flag[0] = true;
}
< critical section >
flag[0] = false;
...
```

```
/*Process 1*/:
...
flag[1] = true;
while (flag[0])
{
  flag[1]= false;
  /*delay*/
  flag[1] = true;
}
< critical section >
flag[1] = false;
...
```


5a. Tentativa (cont.)

- Esta solução é quase correta. Entretanto, existe um pequeno problema: a possibilidade dos processos ficarem cedendo a vez um para o outro "indefinidamente" (problema da "mútua cortesia")
  - Livelock
- Na verdade, essa é uma situação muito difícil de se sustentar durante um longo tempo na prática, devido às velocidades relativas dos processos. Entretanto, ela é uma possibilidade teórica, o que invalida a proposta como solução geral do problema.

5a. Tentativa – Exemplo

P_0 seta *flag[0]* para *true*.
P_1 seta *flag[1]* para *true*.
P_0 testa *flag[1]*.
P_1 testa *flag[0]*.
P_0 seta *flag[0]* para *false*.
P_1 seta *flag[1]* para *false*.
P_0 seta *flag[0]* para *true*.
P_1 seta *flag[1]* para *true*.

Tipos de Soluções (cont.)

- Soluções de Hardware
  - Inibição de interrupções
  - Instrução TSL (apresenta *busy wait*)
- Soluções de software com *busy wait*
  - Variável de bloqueio
  - Alternância estrita
  - Algoritmo de Dekker
  - Algoritmo de Peterson
- Soluções de software com bloqueio
  - Sleep / Wakeup, Semáforos, Monitores

Solução de Dekker

- Trata-se da primeira solução correta para o problema da exclusão mútua de dois processos (proposta na década de 60).
- O algoritmo combina as idéias de variável de bloqueio e *array* de intenção.
- É similar ao algoritmo da 5ª tentativa mas usa uma variável adicional (*vez/turn*) para realizar o desempate, no caso dos dois processos entrarem no *loop* de mútua cortesia.
- *Turn* indica que processo tem o *direito de insistir* em entrar na área de exclusão mútua.

Algoritmo de Dekker

```
boolean flag[2];
int turn;

void P0()
{
   flag[0] = true;
   while (flag[1])
   if (turn == 1)
   {
      flag[0] = false;
      while (turn == 1)
        /*faz nada*/
      flag[0] = true;
    }
   /*critical section*/
   turn = 1;
   flag[0] = false;
   /*resto*/
}

void main()
{
   flag[0] = false;
   flag[1] = false;
   turn = 1;
   parbegin(P0, P1);
}
```

```
void P1()
{
   flag[1] = true;
   while (flag[0])
   if (turn == 0)
   {
      flag[1] = false;
      while (turn == 0)
        /*faz nada*/
      flag[1] = true;
    }
   /*critical section*/
   turn = 0;
   flag[1] = false;
   /*resto*/
}
```


Algoritmo de Dekker (cont.)

- Quando *P0* quer entrar na sua R.C. ele coloca seu *flag* em *true*. Ele então vai checar o *flag* de *P1*.
- Se o *flag* de *P1* for *false*, então *P0* pode entrar imediatamente na sua R.C.; do contrário, ele consulta a variável *turn*.
- Se *turn = 0* então *P0* sabe que é a sua vez de insistir e, deste modo, fica em *busy wait* testando o estado de *P1*.
- Em certo ponto, *P1* notará que é a sua vez de declinar. Isso permite ao processo *P0* prosseguir.
- Após *P0* usar a sua R.C. ele coloca o seu *flag* em *false* para liberá-la, e faz *turn = 1* para transferir o direito para *P1*.

Algoritmo de Dekker (cont.)

- Algoritmo de Dekker resolve o problema da exclusão mútua
- Uma solução deste tipo só é aceitável se houver um número de CPUs igual (ou superior) ao número de processos que se devam executar no sistema. Porquê?
  - Poderíamos nos dar 'ao luxo' de consumir ciclos de CPU,
  - Situação rara na prática (em geral, há mais processos do que CPUs)
  - Isto significa que a solução de Dekker é pouco usada.
- Contudo, a solução de Dekker mostrou que é possível resolver o problema inteiramente por software, isto é, sem exigir instruções máquina especiais.
- Devemos fazer uma modificação significativa do programa se quisermos estender a solução de 2 para N processos:
  - flag[] com N posições; variável turn passa a assumir valores de 1..N; alteração das condições de teste em todos os processos

Solução de Peterson

- Proposto em 1981, é uma solução simples e elegante para o problema da exclusão mútua, sendo facilmente generalizado para o caso de *n* processos.
- O truque do algoritmo consiste no seguinte:
  - Ao marcar a sua intenção de entrar, o processo já indica (para o caso de empate) que a vez é do outro.

Algoritmo de Peterson

```
boolean flag[2];
int turn;

void P0()
{
   while (true)
   {
      flag[0] = true;
      turn = 1;
      while (flag[1] && turn ==1)
         /*faz nada*/
      /*critical section*/
      flag[0] = false;
      /*resto*/
   }
}

void P1()
{
   while (true)
   {
      flag[1] = true;
      turn = 0;
      while (flag[1] && turn == 0)
         /*faz nada*/
      /*critical section*/
      flag[1] = false;
      /*resto*/
   }
}

void main()
{
   flag[0] = false;
   flag[1] = false;
   parbegin(P0, P1);
}
```


Solução de Peterson (cont.)

- Exclusão mútua é atingida.
  - Uma vez que *P0* tenha feito *flag[0] = true*, *P1* não pode entrar na sua R.C.
  - Se *P1* já estiver na sua R.C., então *flag[1] = true* e *P0* está impedido de entrar.
- Bloqueio mútuo (deadlock) é evitado.
  - Supondo *P0* bloqueado no seu *while*, isso significa que *flag[1] = true* e que *turn = 1*
  - se flag[1] = true e turn = 1, então P1 por sua vez entrará na sua seção crítica
  - Assim, *P0* só pode entrar quando **ou** *flag[1]* tornar-se *false* **ou** *turn* passar a ser 0.

Tipos de Soluções (cont.)

- Soluções de Hardware
  - Inibição de interrupções
  - Instrução TSL (apresenta *busy wait*)
- Soluções de software com *busy wait*
  - Variável de bloqueio
  - Alternância estrita
  - Algoritmo de Dekker
  - Algoritmo de Peterson
- Soluções de software com bloqueio
  - Sleep / Wakeup, Semáforos, Monitores

A alteração do valor p/ "trancado" APÓS o teste permite que dois processos executem a R.C. ao mesmo tempo!
O TESTE e a ALTERAÇÃO necessitam ser feitos de forma **indivisível**...

A Instrução TSL (1)

- TSL = "Test and Set Lock"
- Solução de hardware para o problema da exclusão mútua em ambiente com vários processadores.
  - O processador que executa a TSL bloqueia o barramento de memória, impedindo que outras CPUs acessem a MP até que a instrução tenha terminado.
- A instrução TSL faz o seguinte:
  - Lê o conteúdo de um endereço de memória (variável compartilhada "*lock*", usada para proteger a R.C.) para um registrador e armazena um valor diferente de zero (normalmente 1) nesse endereço.

A Instrução TSL (2)

- Se *lock = 0* ⇒ R.C. livre;
  Se *lock = 1* ⇒ R.C. ocupada.
  (*Lock* é iniciada com o valor 0).
- A instrução TSL é executada de forma atômica.
  - As operações de leitura e armazenamento da variável *lock* são garantidamente indivisíveis, sem interrupção.
  - Nenhuma outra CPU pode acessar *lock* enquanto a instrução não tiver terminado.

A Instrução TSL (3)

```
enter_region:
  tsl register, lock |copia lock para registrador e o
                     |configura como 1
  cmp register, #0   |lock era zero?
  jne enter_region   |se não era zero, o bloqueio estava
                     |configurado, então inicia um laço
  ret                |retorna para aquele que fez
                     |a chamada; entrada na região crítica

leave_region:
  move lock, #0      |armazena um 0 no bloqueio(lock)
  ret                |retorna para aquele que fez
                     |a chamada
```


A Instrução TSL (5)

- Vantagens da TSL:
  - Simplicidade de uso (embora sua implementação em hardware não seja trivial).
  - Não dá aos processos de usuário o poder de desabilitar interrupções.
  - Presente em quase todos os processadores atuais.
  - Funciona em máquinas com vários processadores.
- Desvantagens:
  - Espera ocupada (*busy wait*).
  - Possibilidade de postergação infinita (starvation)
    - "processo azarado" sempre pega a variável *lock* com o valor 1

Tipos de Soluções (cont.)

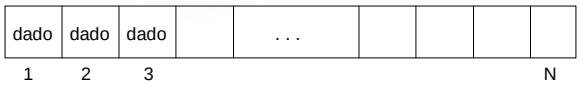
- Soluções de Hardware
  - Inibição de interrupções
  - Instrução TSL (apresenta *busy wait*)
- Soluções de software com *busy wait*
  - Variável de bloqueio
  - Alternância estrita
  - Algoritmo de Dekker
  - Algoritmo de Peterson
- Soluções de software com bloqueio
  - Sleep / Wakeup, Semáforos, Monitores

As Primitivas *Sleep* e *Wakeup*

- A idéia desta abordagem é bloquear a execução dos processos quando a eles não é permitido entrar em suas regiões críticas
- Isto evita o desperdício de tempo de CPU, como nas soluções com *busy wait*.
- Introduz primitivas de comunicação interprocesso que bloqueiam ao invés de esperar.
- Sleep()
  - Bloqueia o processo e espera por uma sinalização, isto é, suspende a execução do processo que fez a chamada até que um outro o acorde.
- Wakeup()
  - Sinaliza (acorda) o processo anteriormente bloqueado por *Sleep*().

O Problema do Produtor e Consumidor com *Buffer* Limitado

- Processo produtor gera dados e os coloca em um *buffer* de tamanho N.
- Processo consumidor retira os dados do *buffer*, na mesma ordem em que foram colocados, um de cada vez.
- Se o *buffer* está **cheio**, o produtor deve ser bloqueado
- Se o *buffer* está **vazio**, o consumidor é quem deve ser bloqueado.
- Apenas um único processo, produtor ou consumidor, pode acessar o *buffer* num certo instante.



Uso de *Sleep* e *Wakeup* para o Problema do Produtor e Consumidor

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

void producer(void) {
 while (true){
  produce_item();               /* generate next item */
  if (count == N) sleep();      /* if buffer is full, go to sleep */
  enter_item();                 /* put item in buffer */
  count = count + 1;            /* increment count of items in buffer*/
  if (count == 1) wakeup(consumer);   /* was buffer empty? */
 }
}

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

Uma Condição de Corrida ...

- *Buffer* está vazio. Consumidor testa o valor de *count*, que é zero, mas não tem tempo de executar *sleep*, pois o escalonador selecionou agora produtor. Este produz um item, insere-o no *buffer* e incrementa *count*. Como *count = 1*, produtor chama *wakeup* para acordar consumidor. O sinal não tem efeito (é perdido), pois o consumidor ainda não está logicamente adormecido. Consumidor ganha a CPU, executa *sleep* e vai dormir. Produtor ganha a CPU e, cedo ou tarde, encherá o *buffer*, indo também dormir. Ambos dormirão eternamente.

Tipos de Soluções (cont.)

- Soluções de Hardware
  - Inibição de interrupções
  - Instrução TSL (apresenta *busy wait*)
- Soluções de software com *busy wait*
  - Variável de bloqueio
  - Alternância estrita
  - Algoritmo de Dekker
  - Algoritmo de Peterson
- Soluções de software com bloqueio
  - Sleep / Wakeup, Semáforos, Monitores

Semáforos (1)

- Mecanismo criado pelo matemático holandês E.W. Dijkstra, em 1965.
- O semáforo é uma variável inteira que pode ser mudada por apenas duas operações primitivas (atômicas): *P* e *V*.
  - *P* = *proberen* (testar) e *V* = *verhogen* (incrementar).
- A idéia é utilizar o semáforo (S) para contar o número de *wakeups* salvos para o futuro.
- Se S=0, indica que nenhum *wakeup* foi salvo
- Se S>0, indica que há *wakeups* pendentes

Semáforos (2)

- Quando um processo executa uma operação *P*, o valor do semáforo é decrementado. O processo pode ser eventualmente bloqueado e inserido na fila de espera do semáforo.
- Numa operação *V*, o semáforo é incrementado e, eventualmente, um processo que aguarda na fila de espera deste semáforo é acordado.
- A operação *P* também é comumente referenciada como *DOWN* ou *WAIT*, e a operação *V* como *UP* ou *SIGNAL*.
- Semáforos que assumem somente os valores *0* e *1* são denominados *semáforos binários* ou *mutex*. Neste caso, P e V são também chamadas de *LOCK* e *UNLOCK*, respectivamente.

Semáforos (3)

```
P(S):
  If (S > 0)
     S = S – 1;
  Else bloqueia processo (coloca-o na fila de S)


V(S):
 If (algum processo dorme na fila de S)
    acorda processo;
 Else S = S + 1;
```