Laboratório de Pesquisa em Redes e Multimídia

Sincronização de Processos: Semáforos

Aula 12

Profa. Patrícia Dockhorn Costa

Universidade Federal do Espírito Santo
Departamento de Informática

Tipos de Soluções (cont.)

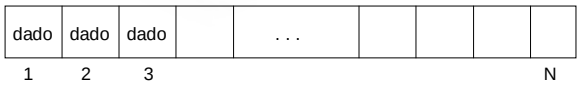
- Soluções de Hardware
  - Inibição de interrupções
  - Instrução TSL (apresenta *busy wait*)
- Soluções de software com *busy wait*
  - Variável de bloqueio
  - Alternância estrita
  - Algoritmo de Dekker
  - Algoritmo de Peterson
- Soluções de software com bloqueio
  - Sleep / Wakeup, Semáforos, Monitores

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).
- Fundamentalmente, a idéia é utilizar o semáforo (S) para permitir a cooperação entre processos por meio de *sinais*. Dessa maneira, um processo pode ser forçado a esperar num determidado lugar até que receba um sinal específico.
- Para receber um sinal, usa-se P(S)
- Para transmitir um sinal, usa-se V(S)

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.

Definição das Primitivas P(S) e V(S)

```
struct semaphore{
       int count;
       queueType q;}

void P(semaphore s)
{  s.count--;
   if(s.count<0){
        coloca o processo na fila q de s;
        bloquea o processo;
   }
}
void V(semaphore s)
{ s.count++;
  if (s.count <= 0){
        remove o processo P da fila q de s;
        coloca o processo P na fila de prontos;
   }
}
```

Semáforo pode ser inicializado com um valor não negativo

Definição das Primitivas Binárias P(S) e V(S)

```
struct binary_semaphore{
       enum {zero, one} value;
       queueType q;}
```

Semáforo pode ser inicializado com 0 ou 1

```
void PB(binary_semaphore s)
{
 if (s.value == 1)
    s.value = 0;
 else{
   coloca o processo na fila q
de s;
   bloquea o processo;
 }
}
```

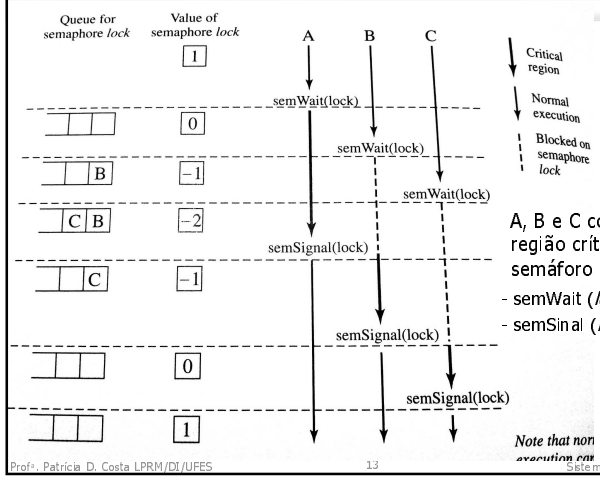
```
void VB(binary_semaphore s)
{
 if (s.q esta vazia())
    s.value = 1;
 else{
   remove o processo P da fila
q de s;
   coloca o processo P na fila
de prontos;
  }
}
```


Uso de Semáforos (1)

- Exclusão mútua (semáforos binários):

```
...
bynary_semaphore mutex = 1;
/* var. semáforo, inicializado com 1 */
```

Processo P_1	Processo P_2	...	Processo P_n
...	...		...
P(mutex)	P(mutex)		P(mutex)
// R.C.	// R.C.		// R.C.
V(mutex)	V(mutex)		V(mutex)
...	...		...



Uso de Semáforos (2)

- Alocação de Recursos (semáforos contadores)
- Para permitir que mais de um processo acesse a região crítica ao mesmo tempo (e.g., no caso de recursos compartilhados) basta inicializar o semáforo com o valor especificado. Neste caso:
  - s.count>=0: s.count representa o número de processos que podem executar o P(S) sem bloquear.
  - s.count<0: a magnitude de s.count representa o número de processos suspensos em s.queue.

Uso de Semáforos (3)

- Alocação de Recursos (semáforos contadores):

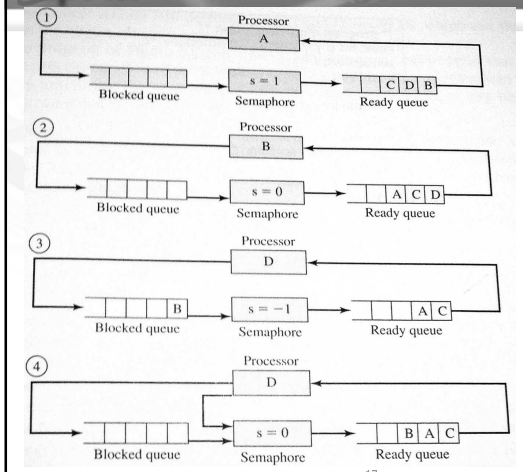
```
...
Semaphore S = 3;    /*var. semáforo, iniciado com
                    qualquer valor inteiro */

Processo P1         Processo P2         Processo P3
...                 ...                 ...
P(S)                P(S)                P(S)
//usa recurso       //usa recurso       //usa recurso
V(S)                V(S)                V(S)
...                 ...                 ...
```

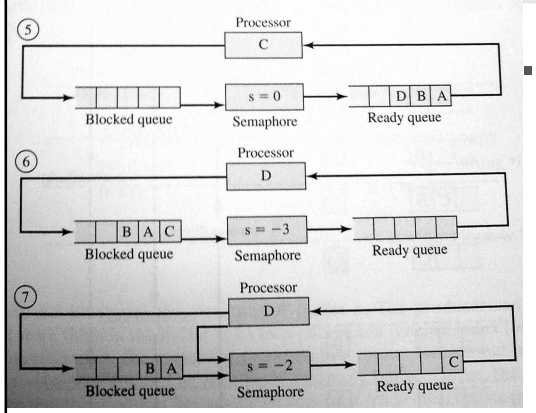

Uso de Semáforos (4)

- Relação de precedência entre processos:
  (Ex: executar *p1_rot2* somente depois de *p0_rot1*)

```
semaphore S = 0 ;
parbegin
      begin                     /* processo P0*/
            p0_rot1()
            V(S)
            p0_rot2()
      end
      begin                     /* processo P1*/
            p1_rot1()
            P(S)
            p1_rot2()
      end
parend
```



- Relação de precedência entre processos (A, B, C precisam do resultado de D)



- Relação de precedência entre processos (A, B, C precisam do resultado de D)

Uso de Semáforos (5)

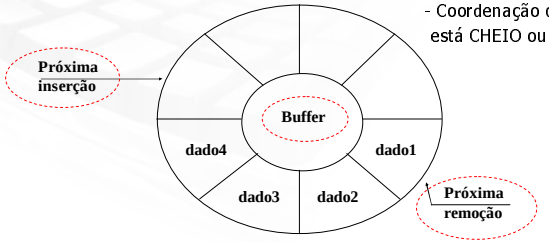
- Sincronização do tipo barreira:
  (*n-1* processos aguardam o *n-ésimo* processo para todos prosseguirem)

```
semaphore X = 0;
semaphore Y = 0;
semaphore Z = 0;
...
```

```
P1:          P2:          P3:
  ...          ...          ...
  V(X);        V(Y);        V(Z);
  V(X);        V(Y);        V(Z);
  P(Y);        P(X);        P(X);
  P(Z);        P(Z);        P(Y);
  do_A;        do_B;        do_C;
  ...          ...
```


Produtor - Consumidor c/ *Buffer* Circular (1)

Dois problemas p/ resolver:
- Variáveis compartilhada
- Coordenação quando o buffer está CHEIO ou VAZIO



Produtor Consumidor c/ *Buffer* Circular (2)

- *Buffer* com capacidade N (vetor de N elementos).
- Variáveis *proxima_insercao* e *proxima_remocao* indicam onde deve ser feita a próxima inserção e remoção no *buffer*.
- Efeito de *buffer* circular é obtido através da forma como essas variáveis são incrementadas. Após o valor *N-1* elas voltam a apontar para a entrada zero do vetor
  - % representa a operação "resto da divisão"
- Três semáforos, duas funções diferentes: exclusão mútua e sincronização.
  - *mutex*: garante a exclusão mútua. Deve ser iniciado com "1".
  - *espera_dado*: bloqueia o consumidor se o *buffer* estiver vazio. Iniciado com "0".
  - *espera_vaga*: bloqueia produtor se o *buffer* estiver cheio. Iniciado com "N".

Produtor - Consumidor c/ *Buffer* Circular (3)

```
struct tipo_dado buffer[N];
int proxima_insercao = 0;
int proxima_remocao = 0;
...
semaphore mutex = 1;
semaphore espera_vaga = N;
semaphore espera_dado = 0;

----------------------------------------
void produtor(void){
...
down(espera_vaga);
down(mutex);
buffer[proxima_insercao] = dado_produzido;
proxima_insercao = (proxima_insercao + 1) % N;
up(mutex);
up(espera_dado);
... }

----------------------------------------
void consumidor(void){
...
down(espera_dado);
down(mutex);
dado_a_consumir := buffer[proxima_remocao];
proxima_remocao := (proxima_remocao + 1) % N;
up(mutex);
up(espera_vaga);
... }
```


Produtor - Consumidor c/ *Buffer* Limitado

```
#define N 100          /* number of slots in the buffer */
typedef int semaphore; /* semaphores are a special kind of int */
semaphore mutex = 1;   /* controls access to critical region */
semaphore empty = N;   /* counts empty buffer slots */
semaphore full = 0;    /* counts full buffer slots */

void producer(void){
  int item;
  produce_item(&item); /* generate something to put in buffer */
  P(&empty);           /* decrement empty count */
  P(&mutex);           /* enter critical region */
  enter_item(item);    /* put new item in buffer */
  V(&mutex);           /* leave critical region */
  V(&full);            /* increment count of full slots */
}

void consumer(void){
  int item;
  P(&full);            /* decrement full count */
  P(&mutex);           /* enter critical region */
  remove_item(&item);  /* take item from buffer */
  V(&mutex);           /* leave critical region */
  V(&empty);           /* increment count of empty slots */
  consume_item(item);  /* do something with the item */
}
```


Deficiência dos Semáforos (1)

- Exemplo: suponha que os dois *down* do código do produtor estivessem invertidos. Neste caso, *mutex* seria diminuído antes de *empty*. Se o *buffer* estivesse completamente cheio, o produtor bloquearia com *mutex = 0*. Portanto, da próxima vez que o consumidor tentasse acessar o *buffer* ele faria um *down* em *mutex*, agora zero, e também bloquearia. Os dois processos ficariam bloqueados eternamente.
- Conclusão: erros de programação com semáforos podem levar a resultados imprevisíveis.

Deficiência dos Semáforos (2)

- Embora semáforos forneçam uma abstração flexível o bastante para tratar diferentes tipos de problemas de sincronização, ele é inadequado em algumas situações.
- Semáforos são uma abstração de alto nível baseada em primitivas de baixo nível, que provêem atomicidade e mecanismo de bloqueio, com manipulação de filas de espera e de escalonamento. Tudo isso contribui para que a operação seja lenta.
- Para alguns recursos, isso pode ser tolerado; para outros esse tempo mais longo é inaceitável.
  - Ex: (Unix) Se o bloco desejado é achado no *buffer cache*, *getblk()* tenta reservá-lo com P(). Se o *buffer* já estiver reservado, não há nenhuma garantia que ele conterá o mesmo bloco que ele tinha originalmente.