Laboratório de Pesquisa em Redes e Multimídia

Sincronização de Processos (2)

Universidade Federal do Espírito Santo
Departamento de Informática

Tipos de Soluções (cont.)

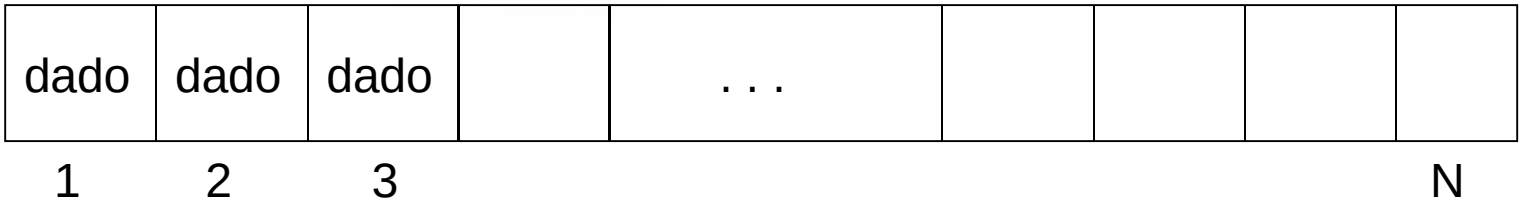
- Soluções de Hardware
  - Inibição de interrupções
  - Instrução TSL (apresenta *busy wait*)
- Soluções de software com *busy wait*
  - Variável de bloqueio
  - Alternância estrita
  - Algoritmo de Dekker
  - Algoritmo de Peterson
- Soluções de software com bloqueio
  - Sleep / Wakeup, Semáforos, Monitores

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*.
- 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 c/ *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)
  - V = *verhogen* (incrementar).
- Quando um processo executa uma operação ***P***, o valor do semáforo é **decrementado** (se o semáforo for maior que 0). O processo pode ser eventualmente bloqueado (semáforo for igual a 0) 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.

Semáforos (2)

- A operação *P* também é comumente referenciada como:
  - *down* ou *wait*
- *V* também é comumente referenciada
  - *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
     Then S := S – 1
  Else bloqueia processo (coloca-o na fila de S)


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

Uso de Semáforos (1)

- Exclusão mútua (semáforos binários):

```
...
Semaphore mutex = 1;        /*var.semáforo,
                            iniciado 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):

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

Processo P_1	Processo P_2	Processo P_3
...	...	...
P(S)	P(S)	P(S)
//usa recurso	//usa recurso	//usa recurso
V(S)	V(S)	V(S)
...	...	...

Uso de Semáforos (3)

- 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
```

Exemplo: Produtor - Consumidor c/ *Buffer* Limitado

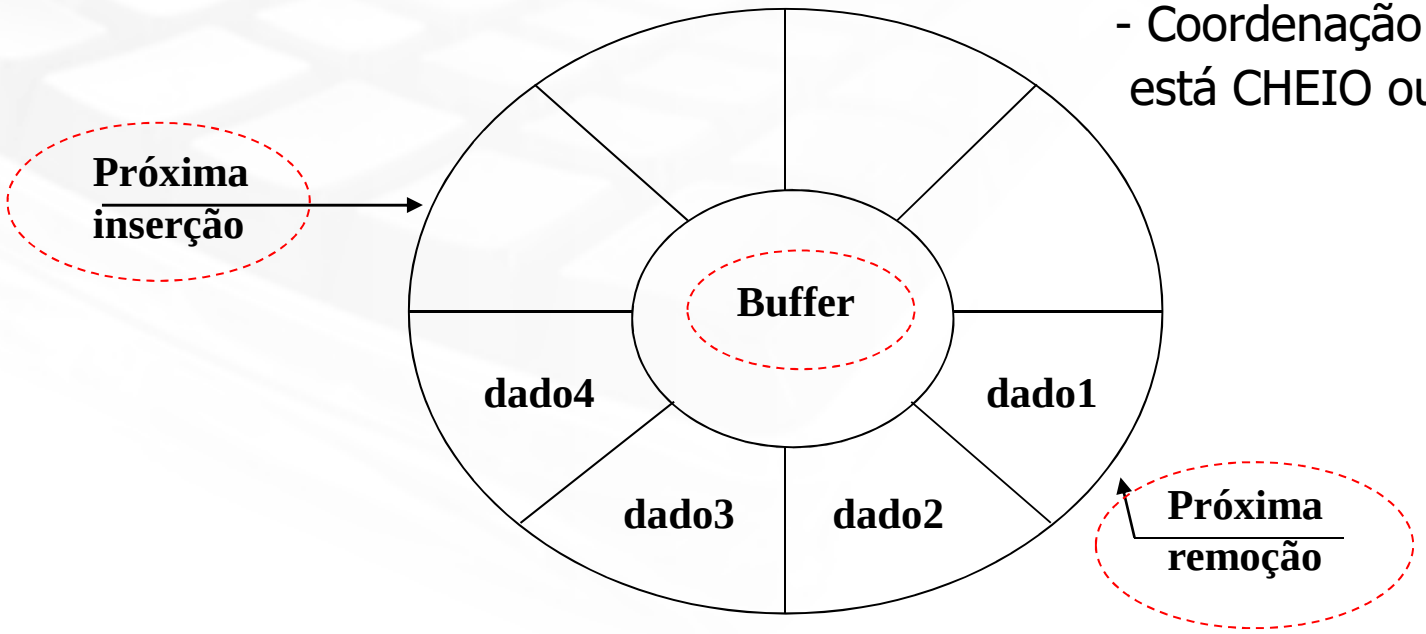
```
#define N 100                   /* number of slots in the buffer */
typedef int semaphore;          /* semaphores are a special kind of int */
semaphore mutex = 1;            /* controls access to critical region */
semaphore empty = N;            /* counts empty buffer slots */
semaphore full =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

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 veto
  - % 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* está vazio. Iniciado com "0".
  - *espera_vaga*: bloqueia produtor se o *buffer* está cheio. Iniciado com "N".

```
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);
... }
```

```
down(S):
  SE S > 0 ENTÃO S := S – 1
  SENÃO bloqueia processo


up(S):
  SE algum processo dorme na fila de S
  ENTÃO acorda processo
  SENÃO S := S + 1
```

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

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.

Referências

- A. S. Tanenbaum, "Sistemas Operacionais Modernos", 2a. Edição, Editora Prentice-Hall, 2003.
  - Seções 2.3.4 a 2.3.6
- Silberschatz A. G.; Galvin P. B.; Gagne G.; "Fundamentos de Sistemas Operacionais", 6a. Edição, Editora LTC, 2004.
  - Seção 7.4
- Deitel H. M.; Deitel P. J.; Choffnes D. R.; "Sistemas Operacionais", 3ª. Edição, Editora Prentice-Hall, 2005
  - Seção 5.6