Sincronização de Processos (4) - Semáforos -

Aula 11

Semáforos (1)

- Mecanismo criado pelo matemático holandês E.W. Dijkstra, em 1965.
- É uma variável inteira que pode ser mudada por apenas duas operações primitivas (atômicas, não interrompíveis): *P* e *V*.
  - *P* = *proberen* (testar) e *V* = *verhogen* (incrementar).
- Quando um processo executa uma operação *P*, o valor do semáforo é decrementado e, em função disto, o processo pode ser bloqueado e inserido na fila de espera do semáforo.

Semáforos (2)

- Numa operação *V,* o semáforo é incrementado e, dependendo do seu novo valor, um sinal é enviado a um outro processo que aguarda na fila de espera deste semáforo.
- A operação *P* também é comumente referenciada como *UP* ou *WAIT*, e a operação *V* como *DOWN* ou *SIGNAL*.
- Semáforos que assumem somente os valores *0* (livre) e *1* (ocupado) 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):

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


Processo P1     Processo P2    ...    Processo Pn
...             ...                   ...
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):

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

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 (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
```

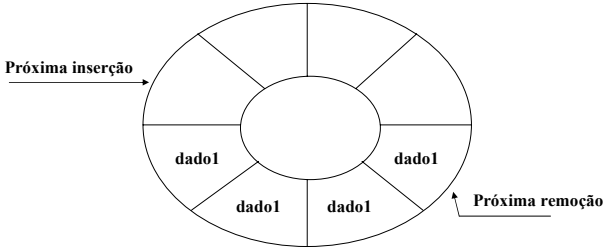

Uso de Semáforos (4)

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

```
Y,Z: semaphore initial 0;
...

P1:                      P2:
  ...                      ...
  V(Z);                    V(Y);
  P(Y);                    P(Z);
  A;                       B;
  ...                      ...
```

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



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 (o símbolo % 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".

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){
...
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* 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 */
}
```

Um Alocador de Recursos

- Semáforos contadores (S > 1) podem ser usados para implementar um controlador para um recurso formado por N unidades.
- O controlador é formado por dois procedimentos, *request* e *release*. Para *request*, o argumento *U* é de saída e para *release* o argumento *U* é de entrada.
- A variável *T* é o contador de unidades disponíveis e indica a posição do *array R* que contém a próxima unidade a ser alocada.
- O semáforo *counter* tranca as requisições quando T=0. *Mutex* garante acesso exclusivo à *T* e à *R*.


```
R: array[5] of integer;          /* Supõe-se R iniciado com [5,4,3,2,1] */
T: integer initial 5;            /* Sintaxe: linguagem V4 */
counter: semaphore initial 5;
mutex: semaphore initial 1;


procedure request(U:integer)
{ P(counter)
  P(mutex)
    U := R[T];
    T := T - 1;
  V(mutex)
};


procedure release(U:integer)
{ P(mutex)
    T := T + 1;
    R[T] := U;
  V(mutex);
  V(counter)
};
```

```
P(S):
  SE S > 0 ENTÃO S := S - 1
  SENÃO bloqueia processo


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


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.