Sistemas Operacionais

Othon Oliveira

Fatec – Faculdade de Informática — PE

14 de março de 2016

Contando os sinais de “acordar”

Dijkstra sugeriu uma variável inteira para contar o número de sinais de acordar salvos, para uso futuro.

Contando os sinais de "acordar"

Dijkstra sugeriu uma variável inteira para contar o número de sinais de acordar salvos, para uso futuro.

Semáforos

Essa proposta introduziu uma nova variável chamada de **semáforos**. Um semáforo poderia conter o valor 0, indicando que nenhum sinal de acordar foi salvo – ou algum valor positivo se um sinal de acordar estivessem pendentes.

Contando os sinais de "acordar"

Dijkstra sugeriu uma variável inteira para contar o número de sinais de acordar salvos, para uso futuro.

Semáforos

Essa proposta introduziu uma nova variável chamada de **semáforos**. Um semáforo poderia conter o valor 0, indicando que nenhum sinal de acordar foi salvo – ou algum valor positivo se um sinal de acordar estivessem pendentes.

Up e Down

Dijkstra propôs a existência de duas operações, *down* e *up* (generalização de sleep e wakeup respectivamente).

Ações sobre semáforos

Operações de semáforos

A operação *down* sobre um semáforo verifica se seu valor é maior que 0, se for decresce de 1 (equivale a dizer que "gasta" um sinal de acordar armazenado) e prosseguirá, caso seja 0 o processo será posto pra dormir.

Ações sobre semáforos

Operações de semáforos

A operação *down* sobre um semáforo verifica se seu valor é maior que 0, se for decresce de 1 (equivale a dizer que "gasta" um sinal de acordar armazenado) e prosseguirá, caso seja 0 o processo será posto pra dormir.

Ações atômicas

Todas essas tarefas são atômicas, ou seja indivisíveis. Garante que, uma vez iniciada uma operação de semáforo, nenhum outro processo pode ter acesso ao semáforo até que a operação seja terminada ou bloqueada.

Ações sobre semáforos

Operações de semáforos

A operação *down* sobre um semáforo verifica se seu valor é maior que 0, se for decresce de 1 (equivale a dizer que "gasta" um sinal de acordar armazenado) e prosseguirá, caso seja 0 o processo será posto pra dormir.

Ações atômicas

Todas essas tarefas são atômicas, ou seja indivisíveis. Garante que, uma vez iniciada uma operação de semáforo, nenhum outro processo pode ter acesso ao semáforo até que a operação seja terminada ou bloqueada.

Essas ações atômicas são absolutamente essenciais para resolver o problema da sincronização e evitar condições de disputa.

Ações sobre semáforos

Ações *up* e *down*

A operação *up* incrementa o valor de um semáforo. Se um ou mais processos estiverem dormindo naquele semáforo, incapacitados de terminar um operação *down* anterior, um deles seria escolhido pelo sistema (ex: aleatoriamente) e seria dada permissão para executar seu *down*.

Ações sobre semáforos

Ações *up* e *down*

A operação *up* incrementa o valor de um semáforo. Se um ou mais processos estiverem dormindo naquele semáforo, incapacitados de terminar um operação *down* anterior, um deles seria escolhido pelo sistema (ex: aleatoriamente) e seria dada permissão para executar seu *down*.

Indivisibilidade

A operação de incrementar o semáforo e acordar um processo é também indivisível. Nunca um processo é bloqueado a partir de um *up* (princípio 3 das condições de disputa)

O problema da perda do sinal de "acordar"

Indivisibilidade

É fundamental que seja inplementada de maneira indivisível, diferentemente do modo normal baseado em *up* e *down*, onde o sistema operacional desabilita suas interrupções, pondo-o para dormir, enquanto estiver testando o semáforo.

O problema da perda do sinal de "acordar"

Indivisibilidade

É fundamental que seja inplementada de maneira indivisível, diferentemente do modo normal baseado em *up* e *down*, onde o sistema operacional desabilita suas interrupções, pondo-o para dormir, enquanto estiver testando o semáforo.

O algoritmo em C

```
#define N 100 /* número lugares no buffer */
typedef int semaphore; /* semáforos são um tipo especial de int */
semaphore mutex = 1; /* controla o acesso à região crítica */
semaphore empty = N; /* conta os lugares vazios no buffer */
semaphore full = 0; /* conta os lugares preenchidos no buffer */
...
```

Implementando o código do produtor

```
void producer(void){
  int item;

  while(true){ /* True é a constante 1 */
    item = produzirItem(); /* gera o próximo item */
    down(&empty); /* decresce o contador de empty */
    down(&mutex); /* entra na região crítica */
    inserirItem(item); /* ponha um item no buffer */
    up(&mutex); /* o buffer está vazio? */
    up(&full); /* incrementa o contador de lugares preenchidos */
  }
}
```

O consumidor

```
void consumer(void){
  int intem;

  while(true){ /* True é a constante 1 */
    down(&full); /* decresce o contador de full */
    down(&mutex); /* entra na região crítica */
    item = removeItem(); /* pega um item do buffer */
    up(&mutex); /* deixa a região crítica */
    up(&empty); /* incrementa o cont. de lugares preenchidos */
    consomeItem(Item); /* faz algo com um item, ex: imprime */
  }
}
```