Questionário - 11

Arthur C. M. Barcella e Matheus P. Salazar

Por que não existem operações read(s) e write(s) para ler ou ajustar o valor atual de um semáforo?

Os semáforos são estruturas de dados usados para controlar o acesso a recursos compartilhados, tendo apenas dois estados possíveis, ocupado e desocupado.

Uma thread ou processo que desejar acessar um recurso compartilhado terá que ver o estado do semáforo para que assim consiga adquirir o semáforo ou não.

Dessa forma, caso a thread conseguisse ler ou ajustar o valor do semáforo acabaria gerando uma perda de sincronismo, pois poderia corromper os dados compartilhados ou o próprio valor do semáforo, impossibilitando a liberação de novas tarefas por parte do mesmo.

Mostre como pode ocorrer violação da condição de exclusão mútua se as operações down(s) e up(s) sobre semáforos não forem implementadas de forma atômica.

Processos: P1 e P2

Semáforo: s

P1 solicita down(s) para obter recurso.

P2 solicita down(s) para obter recurso.

P2 executa up(s) e libera o semáforo.

P1 executa up(s) e libera o semáforo.

Para compreendermos como pode ocorrer essa violação, vamos utilizar como exemplo o código à direita.

Note que entre a região de início e fim, é implementada as operações de Down e Up do semáforo impedindo que outras tarefas consigam acessar a posição de memória do “saldo”.

```
1   init (s, 1) ;
2
3   void depositar (semaphore s, int *saldo, int valor){
4
5   // Inicio
6   down (s) ;
7
8   (*saldo) += valor ;
9
10  up (s) ;
11  // Fim
12
13  }
```

Caso essa execução (entre início e fim) não seja atômica, um processo pode alterar o valor do semáforo “down” e então ser retirado da CPU pelo escalonador, impedindo de liberar o acesso ao saldo por outros processos, já que o respectivo processo não pode executar a função “up”.

```
1   init (s, 1) ;
2
3   void depositar (semaphore s, int *saldo, int valor){
4
5   // Inicio
6   down (s) ;
7
8   (*saldo) += valor ;
9
10  up (s) ;
11  // Fim
12
13  }
```

Em que situações um semáforo deve ser inicializado em 0, 1 ou n > 1?

O semáforo deve ser inicializado em 0 quando a tarefa que irá utilizar o recurso irá bloquear o acesso ao mesmo antes de utilizá-lo, levando o semáforo ao valor -1.

O semáforo deve ser iniciado em -1 caso o recurso compartilhado não esteja disponível no momento para ser acessado, para que a tarefa possa acessá-lo algum outro processo precisará liberar o semáforo (up).

Caso seja iniciado em um valor maior que 1 significará que o semáforo faz o controle de acesso a um recurso que pode ser acessado simultaneamente por um número de processos ou thread (perdendo o seu uso correto).

A implementação das operações down(s) e up(s) sobre semáforos deve ser atômica, para evitar condições de disputa sobre as variáveis internas do semáforo. Escreva, em pseudo-código, a implementação dessas duas operações, usando instruções TSL para evitar as condições de disputa. A estrutura interna do semáforo é indicada a seguir. Não é necessário detalhar as operações de ponteiros envolvendo a fila task_queue.

```
1   // Estrutura do semafaro:
2   struct semaphore{
3
4     // Indica se o semáforo está bloqueado ou não.
5     int lock = false;
6
7     //Contador inteiro que representa o número de recursos disponíveis.
8     //Como o recurso deve ser acessado por uma tarefa de cada vez, esse valor nunca será maior que 1.
9     int count;
10
11    // Fila de tarefas (task_t) que estão esperando para acessar o recurso compartilhado.
12    task_t *queue;
13  }
```

```
down(semaphore s) {

  // adquire o "lock" do semáforo, se o "lock" já estiver adquirido, a tarefa atual
  // será bloqueada até que o "lock" seja liberado
  TSL(s.lock);

 // decrementa o contador do semáforo
  s.count--;

  // se o contador do semáforo for negativo
  // a tarefa atual deve ser bloqueada
  if (s.count < 0) {

    // adiciona a tarefa atual na fila de espera
    add_to_queue(s.queue, current_task);

    // libera o lock do semáforo, se outra tarefa estiver esperando o lock
    // ela será acordada e adquirirá o lock
    TSL_UNLOCK(s.lock);

    // suspende a tarefa atual, a tarefa será acordada quando o contador do semáforo
    // for maior ou igual a zero
    block(current_task);

    // se o contador do semáforo for maior ou igual a zero, indicando que
    // o semafaro está liberado, a tarefa atual continuará a execução.
  } else {
    TSL_UNLOCK(s.lock);
  }
}
```

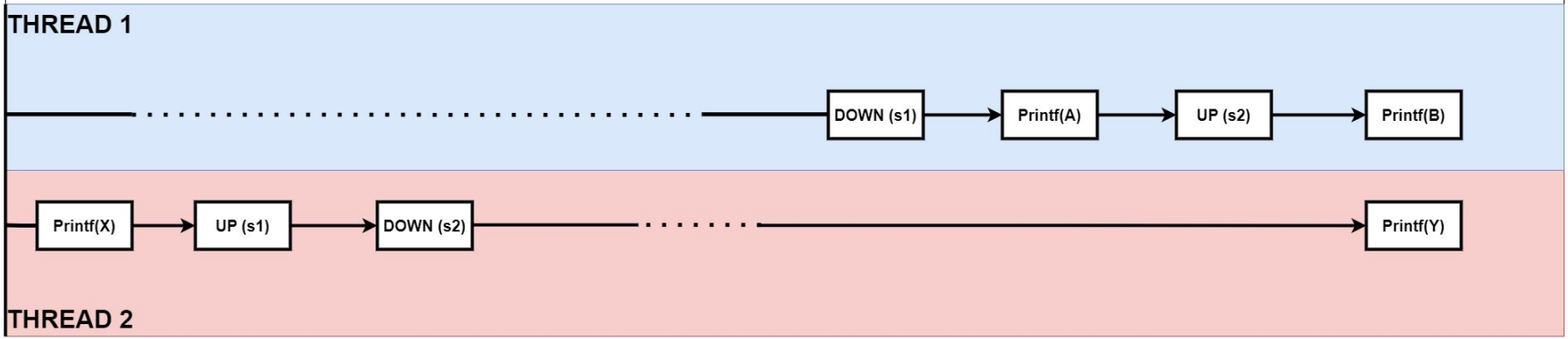
```
up(semaphore s) {

  // adquire o lock do semáforo, se o lock já estiver adquirido, a tarefa atual
  // será bloqueada até que o lock seja liberado
  TSL(s.lock);

  // incrementa o contador do semáforo
  // se o contador do semáforo for negativo, indica que há tarefas na fila de espera
  s.count++;

  // se o contador do semáforo for maior ou igual a zero, indica que o semáforo está liberado
  // e não há tarefas na fila de espera, se o contador do semáforo for negativo, indica que
  // há tarefas na fila de espera
  if (s.count <= 0) {

    // remove uma tarefa da fila de espera, a tarefa removida será acordada.
    // a tarefa será adicionada na fila de tarefas prontas e será executada.
    task_t *task = remove_from_queue(s.queue);
    unblock(task);

  }

  // libera o "lock" do semáforo
  TSL_UNLOCK(s.lock);
}
```

Desenhe o diagrama de tempo da execução e indique as possíveis saídas para a execução concorrente das duas threads cujos pseudo-códigos são descritos a seguir. Os semáforos s1 e s2 estão inicializados com zero (0).

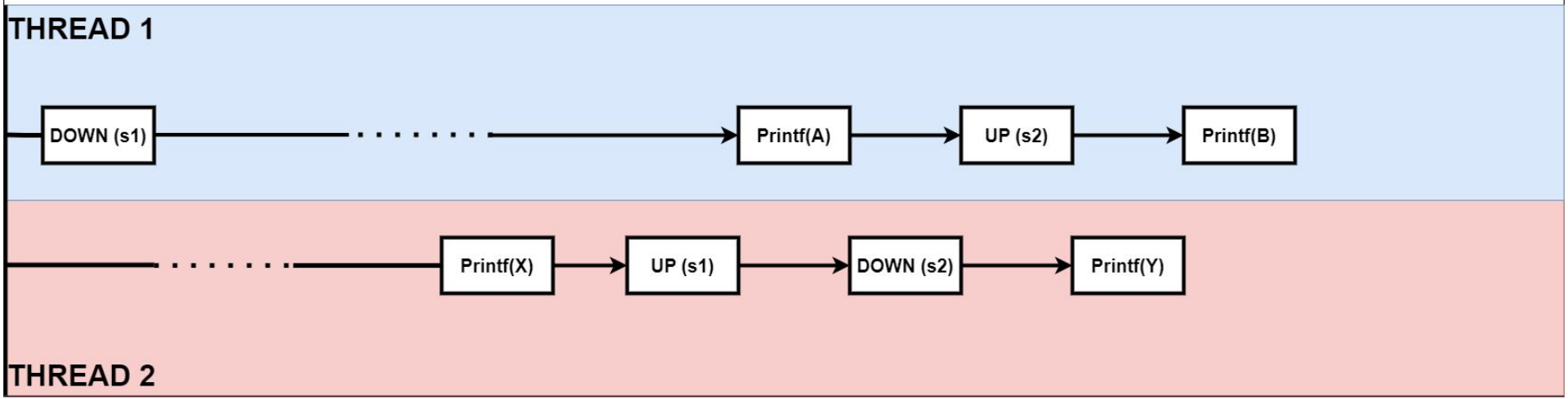
```
thread1 ()
{
    down(s1);
    printf("A");
    up(s2);
    printf("B");
}
```

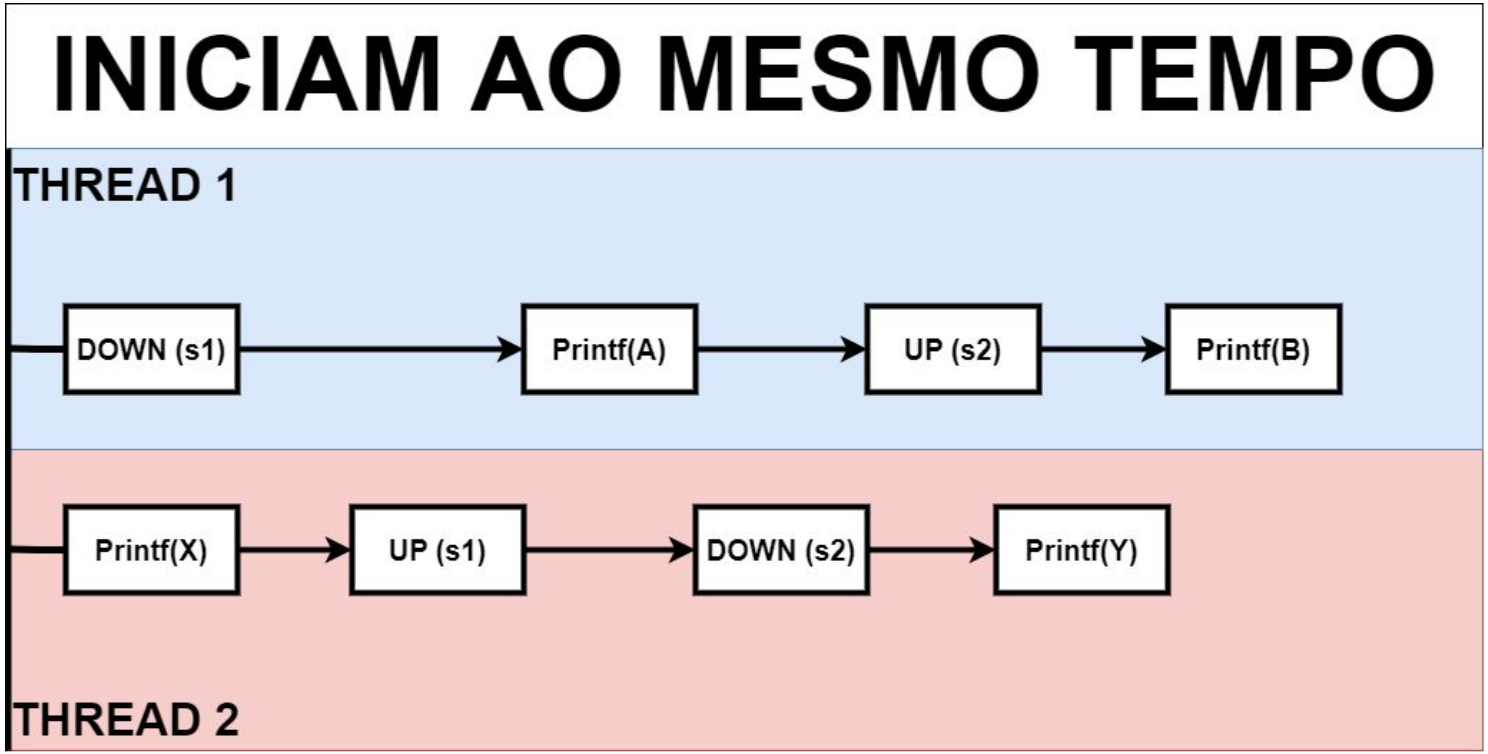
```
thread2 ()
{
    printf("X");
    up(s1);
    down(s2);
    printf("Y");
}
```

THREAD 2 INICIA ANTES DE THREAD 1



THREAD 1 INICIA ANTES DE THREAD 2




INICIAM AO MESMO TEMPO
THREAD 1
DOWN (s1)
Printf(A)
UP (s2)
Printf(B)
Printf(X)
UP (s1)
DOWN (s2)
Printf(Y)
THREAD 2

Questionário - 11

Arthur C. M. Barcella e Matheus P. Salazar