Counting Semaphores

Discussed on whiteboard
30 November 2018

Slides Posted
10 December 2018

Aftab Hussain
University of California, Irvine
semaphore
an integer variable

binary semaphore
semaphore can be 0 or 1

counting semaphore
semaphore can have any integer value
An array that holds a list of suspended threads.
An array that holds a list of suspended threads.
Let’s call it `suspended_list`
Now, for example, we want to run the following code using multiple threads in parallel where:

```javascript
fn()
{
    //code preceding critical section
    ...
    ...
    ...
    //critical section code
    ...
    ...
    ...
    //code following critical section
    ...
    ...
    ...
}
```
Now, for example, we want to run the following code using multiple threads in parallel where:

```javascript
fn()
{
    //code preceding critical section
    ...
    ...
    ...
    //critical section code
    ...
    ...
    ...
    //code following critical section
    ...
    ...
    ...
}
```
Now, for example, we want to run the following code using multiple threads in parallel where:

```
fn()
{
    //code preceding critical section
    ...
    ...
    ...
    //critical section code
    ...
    ...
    ...
    //code following critical section
    ...
    ...
    ...
}
```

### some requirements

- ok to have this code run in parallel by multiple threads
- only $n$ number of threads should run this code at a time.
- ok to have this code run in parallel by multiple threads
Now, for example, we want to run the following code using multiple threads in parallel where:

```plaintext
fn()
{
//code preceding critical section
...
...
...

//critical section code
...
...
...

//code following critical section
...
...
...
}
```

some requirements:
- ok to have this code run in parallel by multiple threads
- only $n$ number of threads should run this code at a time.

let's call this the entry code

let's call this the exit code
Now, for example, we want to run the following code using multiple threads in parallel where:

```javascript
fn()
{

//code preceding critical section
...
...
...

//critical section code
...
...
...

//code following critical section
...
...
...
}
```

**some requirements**

- ok to have this code run in parallel by multiple threads
- only \( n \) number of threads should run this code at a time.
- ok to have this code run in parallel by multiple threads

let's call this the **entry code**

this is the key requirement

let's call this the **exit code**
Let’s use a counting semaphore to fulfill these requirements
int s;

semaphore

fn()
{
    // code preceding critical section
    P()
    // critical section code
    ...
    ...
    // code following critical section
    V()
}
In this example, we can use this semaphore to count threads (in different ways, as we shall soon see). Initial value of s is the number of threads we shall allow to execute in the critical section at the same time.

```c
int s;
fn()
{
    //code preceding critical section
    P()
    //critical section code
    ...
    ...
    //code following critical section
    V()
}
```
Two functions are used to implement a counting semaphore, in order to fulfill the controlled execution requirement of the critical section code.

- $P(s)$ decreases the value of the semaphore. It is placed in the entry code block.
- $V(s)$ increases the value of the semaphore. It is placed in the exit code block.
Two functions are used to implement a counting semaphore, in order to fulfill the controlled execution requirement of the critical section code.

They update the value of the counting semaphore.
Two functions are used to implement a counting semaphore, in order to fulfill the controlled execution requirement of the critical section code.

They update the value of the counting semaphore.

They are by convention named \textbf{P()} and \textbf{V}().\footnote{ref}
Two functions are used to implement a counting semaphore, in order to fulfill the controlled execution requirement of the critical section code.

They update the value of the counting semaphore.

They are by convention named $P()$ and $V()$. (ref)

$P()$ decreases the value of the semaphore. It is placed in the entry code block.
Two functions are used to implement a counting semaphore, in order to fulfill the controlled execution requirement of the critical section code.

They update the value of the counting semaphore.

They are by convention named \texttt{P()} and \texttt{V()}. (ref)

\texttt{P()} decreases the value of the semaphore. It is placed in the \textbf{entry code block}.

\texttt{V()} increases the value of the semaphore. It is placed in the \textbf{exit code block}. 
fn() {

  //code preceding critical section

  P()

  //critical section code

  ...
  ...

  //code following critical section

  V()
}

suspended_list
fn() {
    // code preceding critical section
    P()
    // critical section code
    ...
    ...
    // code following critical section
    V()
}

P() {
    s--;  
    if (s<0) 
    {   
        put thread t to suspended_list;  
        sleep(t);  
    }  
    else return;  
}
fn()
{
   //code preceding critical section
   P()
   //critical section code
   ...
   ...
   //code following critical section
   V()
}

P()
{
   s--;
   if (s<0)
   {
      put thread t to suspended_list;
      sleep(t);
   }
   else return;
}
fn()
{
    // code preceding critical section

    P()
    {
        s--;
        if (s<0)
        {
            put thread t to suspended_list;
            sleep(t);
        }
        else return;
    }

    // critical section code
...
...

    // code following critical section

    V()
    {
        s++;
        if (s<=0)
        {
            pick a thread t from suspended_list;
            wakeup(t);
        }
        else return;
    }
}
fn() {
    //code preceding critical section
    P()
    //critical section code
    ...
    ...
    //code following critical section
    V()
}

P() {
    s--;
    if (s<0) {
        put thread t to suspended_list;
        sleep(t);
    }
    else return;
}

V() {
    s++;
    if (s<=0) {
        pick a thread t from suspended_list;
        wakeup(t);
    }
    else return;
}
Let’s see a running example of how these two functions can help us control execution of the critical section code.
Say initial value of $s$ is 2.
This means we shall allow two threads to execute in the critical section simultaneously.
fn() {
    //code preceding critical section
    P()
    //critical section code
    ...
    ...
    //code following critical section
    V()
}
fn()
{
    //code preceding critical section
    P()
    //critical section code
    ...
    ...
    //code following critical section
    V()
}
fn()
{
    //code preceding critical section
    P()
    //critical section code
    ...
    ...
    //code following critical section
    V()
}

P()
{
    s--; // s becomes 1
    if (s<0)
    {
        put thread t to suspended_list;
        sleep(t);
    }
    else return;
}
fn() {
    // code preceding critical section
    P()
    // critical section code
    T1 Enters critical section.
    ...  
    ...  
    // code following critical section
    V()
    }

s=1

P()
{
    s--; // s becomes 1
    if (s<0)
    {
        put thread t to suspended_list;
        sleep(t);
    }
    else return; // T1 returns
}
fn() {

// code preceding critical section

P()

// critical section code

... T1

... T1

// code following critical section

V()
}
s=1

fn() {
    //code preceding critical section
    P()
    //critical section code
    ...
    ...
    //code following critical section
    V()
}
fn() {
    //code preceding critical section
    P()
    //critical section code
    ...
    ...
    //code following critical section
    V()
}

P() {
    s--; // s becomes 0
    if (s<0) {
        put thread t to suspended_list;
        sleep(t);
    }
    else return;
}
fn()
{
  //code preceding critical section
  P()
  //critical section code
  ...
  T1  T2  Enters critical section.
  ...
  //code following critical section
  V()
  s=0
}

P()
{
  s--; // s becomes 0
  if (s<0)
  {
    put thread t to suspended_list;
    sleep(t);
  }
  else return; //T2 returns
}
s=0

fn()
{
    //code preceding critical section
    P()
    //critical section code
    ... T1 T2 ...
    //code following critical section
    V()
}
Enters function fn()

//code preceding critical section

P()

//critical section code

... T1 T2 ...

//code following critical section

V()
fn() {
    //code preceding critical section
    P()
    //critical section code
    ...
    T1 T2
    //code following critical section
    V()
}

s=-1

P()
{
    s--; // s becomes -1
    if (s<0)
    {
        put thread t to suspended_list;
        sleep(t);
    }
    else return;
}
s=-1

fn()
{
    //code preceding critical section
    P()
    //critical section code
    ...
    ...
    //code following critical section
    V()
}
s = -1  // notice that if s is negative, |s| gives us the **number of sleeping threads**, or threads in suspended_list

```c
fn()
{
    // code preceding critical section
    P()
    // critical section code
    T1  T2
    // code following critical section
    V()
}
```
So now we have T1 and T2 running in the critical section.
Say T1 finishes running critical section code.
$s = -1$

```c
fn()
{
    //code preceding critical section
    P()
    //critical section code
    ...
    ... T2
    //code following critical section
    V() T1 Leaves critical section.
}
```
fn()
{
    //code preceding critical section
    P()
    //critical section code
    ...
    ...
    ...
    //code following critical section
    V()

    s = 0
    V()
    {
        s++;
        //s becomes 0
        if (s <= 0)
        {
            pick a thread t from suspended_list;
            wakeup(t);
        }
        else return;
    }
}
```c
fn()
{
    //code preceding critical section
    P()
    //critical section code
    ...
    ...
    ...
    //code following critical section
    V()
    }

s=0

V()
{
    s++; //s becomes 0
    if (s<=0)
    {
        pick a thread t from suspended_list;
        wakeup(t);
    }
    else return;
}```
T3 is put to a ready state only. It is up to the OS scheduler to decide whether or not to let it execute the critical section immediately.

Note:

```
s=0
fn()
{
    //code preceding critical section
    P()
    //critical section code
    ...
    ...
    //code following critical section
    V()
}
```

```
V()
{
    s++;// s becomes 0
    if (s<=0)
    {
        pick a thread t from suspended_list;
        wakeup(t);
    }
    else return;
}
```
T1 exits.
Couple of points.
fn() {
  //code preceding critical section
  P() {
    s--;
    if (s<0)
      {
        put thread t to suspended_list;
        sleep(t);
      }
    else return;
  }
  //critical section code
  ...
  ...
  //code following critical section
  V() {
    s++;
    if (s<=0)
      {
        pick a thread t from suspended_list;
        wakeup(t);
      }
    else return;
  }
}

Note:
The thread picked from suspended_list can follow any algorithm (e.g. FIFO)
Based on this setup, threads can gain access to these functions parallely and update the value of $s$ simultaneously causing a race on $s$. 

Note:
fn() {
    //code preceding critical section
    P() {
        //critical section code
        ...
        ...
        //code following critical section
        V() {
            //code following critical section
            ...
        }
    }
}

P() {
    s--; 
    if (s<0) 
    { 
        put thread t to suspended_list; 
        sleep(t); 
    } 
    else return;
}

V() {
    s++; 
    if (s<=0) 
    { 
        pick a thread t from suspended_list; 
        wakeup(t); 
    } 
    else return;
}

Hence we need to guard these two functions using locks.
End of demonstration of how a counting semaphore may be used.