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

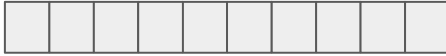
counting semaphore

semaphore can have any integer value

An array that holds a list of suspended threads.

An array that holds a list of suspended threads.

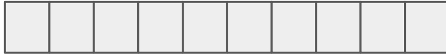
Let's call it *suspended_list*



suspended_list

Now, for example, we want to run the following code using multiple threads in parallel where:

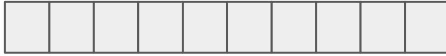
```
fn()
{

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

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

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



suspended_list

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
    ...
    ...
    ...
}
```



suspended_list

Now, for example, we want to run the following code using multiple threads in parallel where:

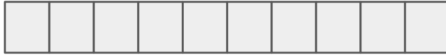
```
fn()
{

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

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

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

some requirements

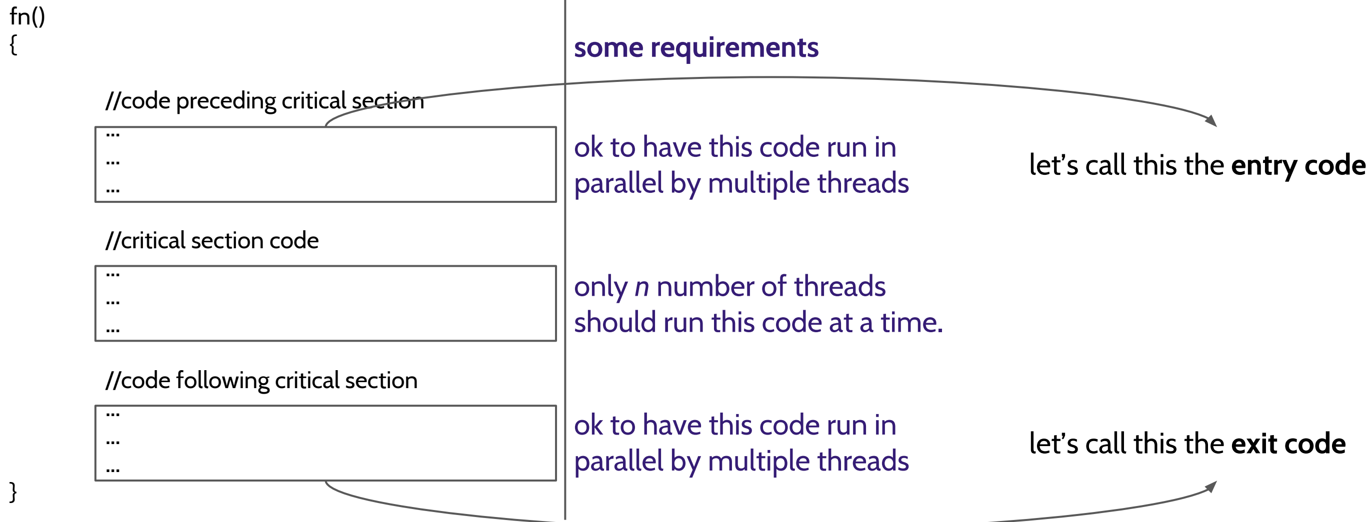
ok to have this code run in parallel by multiple threads

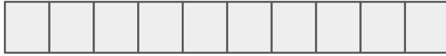
only *n* number of threads should run this code at a time.

ok to have this code run in parallel by multiple threads

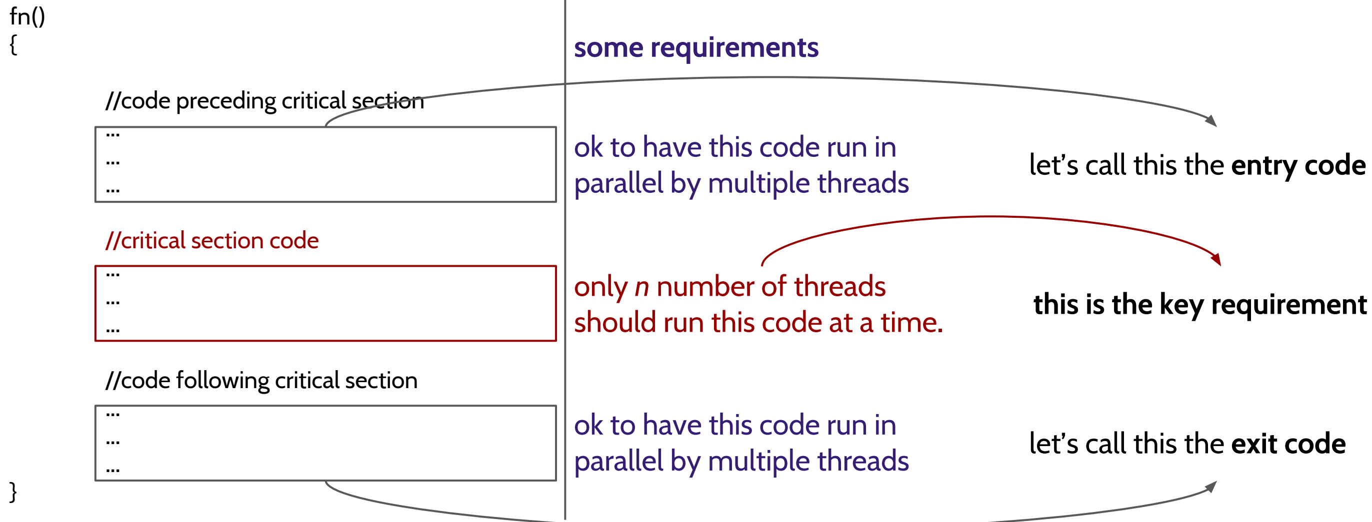


Now, for example, we want to run the following code using multiple threads in parallel where:





Now, for example, we want to run the following code using multiple threads in parallel where:

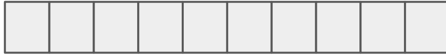


Let’s use a counting semaphore to fulfill these requirements

suspended_list

semaphore

```
fn()
{

    //code preceding critical section

    P()

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

    //code following critical section

    V()

}
```



suspended_list

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.

```
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.

> 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 **P()** and **V()**. (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 **P()** and **V()**. (ref)

> **P()** decreases the value of the semaphore. It is placed in the **entry code block**.

> **V()** increases the value of the semaphore. It is placed in the **exit code block**.

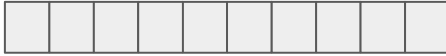
suspended_list

```
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;
}
```



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;
}
```

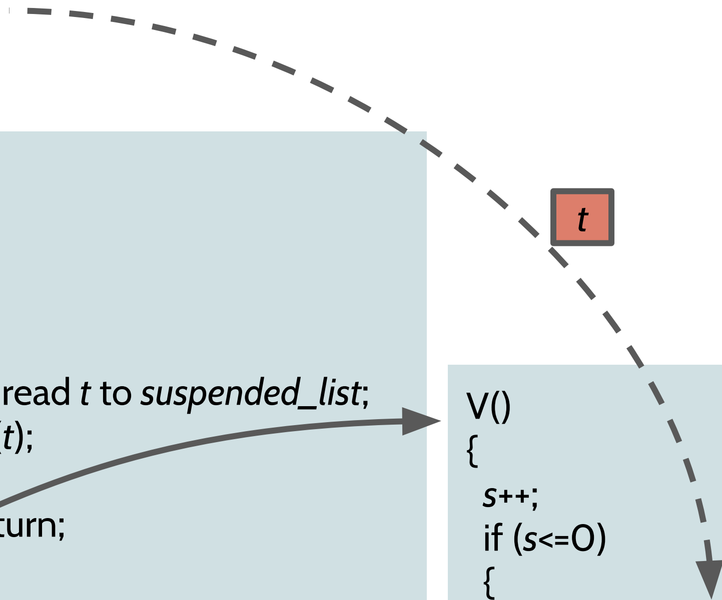
suspended_list

```
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;
}
```

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

suspended_list




```
fn()
{

    //code preceding critical section

    P()

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

    //code following critical section

    V()

}
```

```
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.

suspended_list

s=2

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

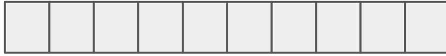
suspended_list

s=2

```
fn()
{
```

T1 **Enters function fn()**

```
    //code preceding critical section

    P()

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

    //code following critical section

    V()

}
```



suspended_list

s=1

fn()
{

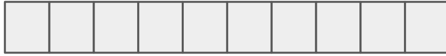
//code preceding critical section

//critical section code

//code following critical section

}

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



suspended_list

s=1

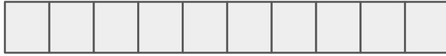
```
fn()
{

    //code preceding critical section

    P()

    //critical section code
    ...
    ...   T1  Enters critical section.
    ...

    //code following critical section

    V()

}
```

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

suspended_list

s=1

```
fn()
{

        //code preceding critical section

        P()

        //critical section code
        ...
        ...        T1
        ...

        //code following critical section

        V()

}
```



suspended_list

s=1

```
fn()
{
```

T2 **Enters function fn()**

//code preceding critical section

P()

//critical section code

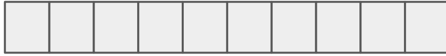
...
...
...

T1

//code following critical section

V()

```
}
```



suspended_list

s=0

fn()
{

//code preceding critical section

//critical section code

//code following critical section

}

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

suspended_list

s=0

```
fn()
{

    //code preceding critical section

    P()

    //critical section code

    ...
    ...
    ...

    //code following critical section

    V()

}
```

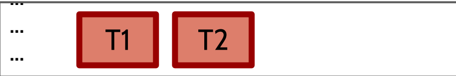
Enters critical section.

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

suspended_list

s=0

```
fn()
{
        //code preceding critical section
        P()

        //critical section code
```




```
        //code following critical section
        V()

}
```

suspended_list

s=0

```
fn()
{
```

T3 **Enters function fn()**

//code preceding critical section

P()

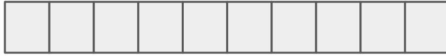
//critical section code

...
...   T1   T2
...

//code following critical section

V()

```
}
```



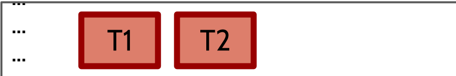
suspended_list

s=-1

fn()
{

//code preceding critical section

//critical section code



//code following critical section

}

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

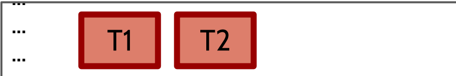
suspended_list

s=-1

fn()
{

//code preceding critical section

//critical section code



//code following critical section

}

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

suspended_list

s=-1 // notice that if s is negative, |s| gives us the **number of sleeping threads**, or threads in suspended_list

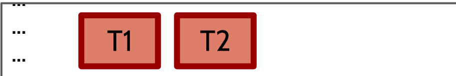
fn()
{

//code preceding critical section

P()

//critical section code

...
...
...



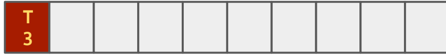
//code following critical section

V()

}

So now we have T1 and T2 running in the critical section.

Say T1 finishes running critical section code.



suspended_list

s=-1

```
fn()
{

        //code preceding critical section

        P()

        //critical section code
        ...
        ...
        ...              T2

        //code following critical section

        V()      T1      Leaves critical section.
}
```

suspended_list

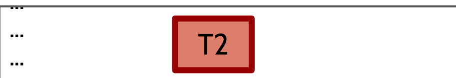
s=0

```
fn()
{
```

//code preceding critical section

P()

//critical section code



//code following critical section

V()

```
}
```

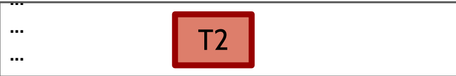
```
V()
{
  s++; //s becomes 0
  if (s<=0)
  {
    pick a thread t from
    suspended_list;
    wakeup(t);
  }
  else return;
}
```

suspended_list

T3

s=0

```
fn()
{

    //code preceding critical section
```



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

T2

```
    //code following critical section
```

V() T1

```
}
```

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

suspended_list

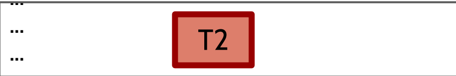
T3

Note:

T3 is put to a ready state only. It is upto the OS scheduler to decide whether or not to let it execute the critical section immediately.

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.

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;
}
```

```
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)

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;
}
```

Note:

Based on this setup, threads can gain access to these functions parallely and update the value of s simultaneously causing a race on s.

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

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;
}
```

Note:

Hence we need to guard these two functions using locks.

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

End of demonstration of how a
counting semaphore may be used.

Refs:

https://en.wikipedia.org/wiki/Semaphore_(programming)
https://www.youtube.com/watch?v=eoGkJWgxurQ //(not in English)