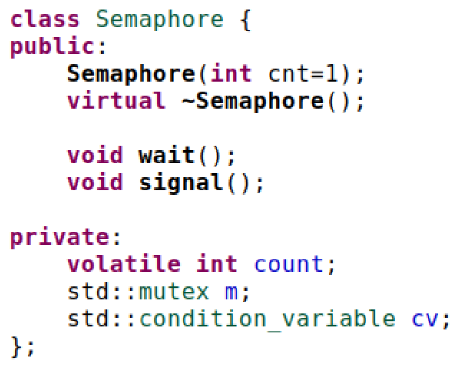
Semaphores

Semaphores

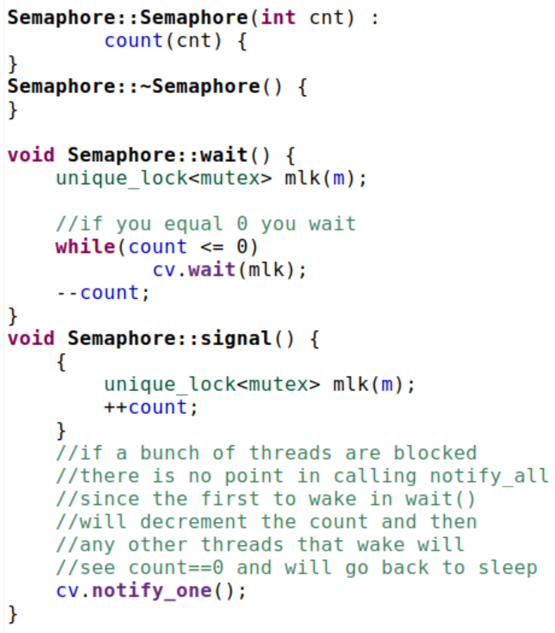
- Think of semaphores as bouncers at a nightclub. Fire codes limit the number of people that are allowed in the club at a time. If the club is full the bouncer makes people wait at the door, but as soon as someone leaves one of these waiting people can enter.
- It's simply a way to limit the number of consumers for a specific resource. For example, to limit the number of simultaneous calls to a database in an application.

Semaphores

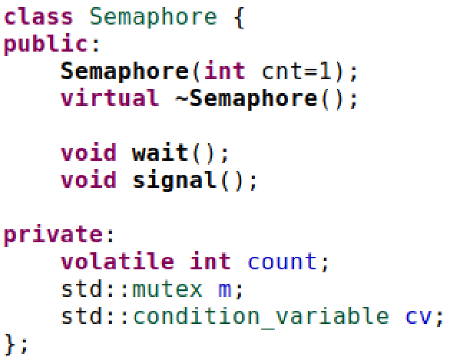
- Must initialize count. It corresponds to how many at once.



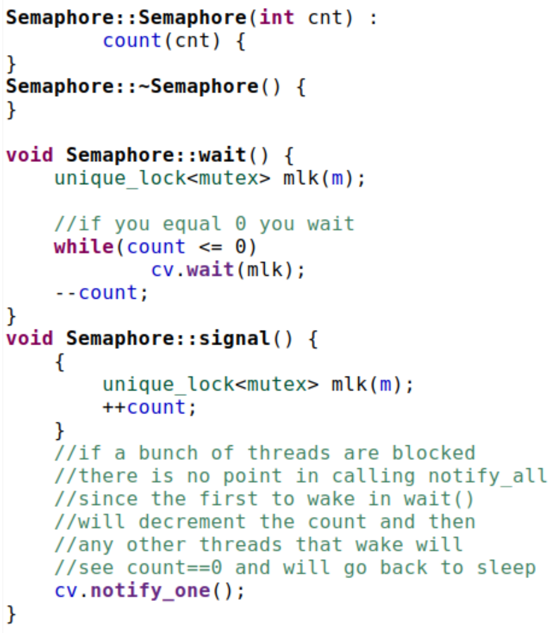
```
class Semaphore {
public:
    Semaphore(int cnt=1);
    virtual ~Semaphore();

    void wait();
    void signal();

private:
    volatile int count;
    std::mutex m;
    std::condition_variable cv;
};
```



```
Semaphore::Semaphore(int cnt) :
        count(cnt) {
}
Semaphore::~Semaphore() {
}

void Semaphore::wait() {
    unique_lock<mutex> mlk(m);

    //if you equal 0 you wait
    while(count <= 0)
            cv.wait(mlk);
    --count;
}
void Semaphore::signal() {
    {
        unique_lock<mutex> mlk(m);
        ++count;
    }
    //if a bunch of threads are blocked
    //there is no point in calling notify_all
    //since the first to wake in wait()
    //will decrement the count and then
    //any other threads that wake will
    //see count==0 and will go back to sleep
    cv.notify_one();
}
```



```
class Semaphore {
public:
    Semaphore(int cnt=1);
    virtual ~Semaphore();

    void wait();
    void signal();

private:
    volatile int count;
    std::mutex m;
    std::condition_variable cv;
};
```

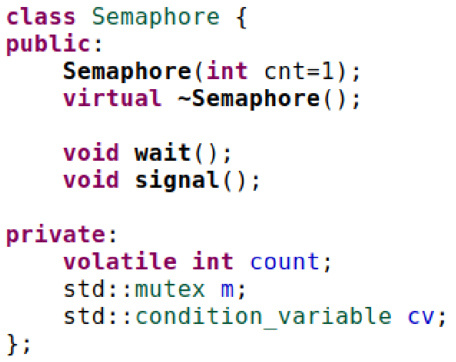


```
Semaphore::Semaphore(int cnt) :
        count(cnt) {
}
Semaphore::~Semaphore() {
}

void Semaphore::wait() {
    unique_lock<mutex> mlk(m);

    //if you equal 0 you wait
    while(count <= 0)
            cv.wait(mlk);
    --count;
}
void Semaphore::signal() {
    {
        unique_lock<mutex> mlk(m);
        ++count;
    }
    //if a bunch of threads are blocked
    //there is no point in calling notify_all
    //since the first to wake in wait()
    //will decrement the count and then
    //any other threads that wake will
    //see count==0 and will go back to sleep
    cv.notify_one();
}
```

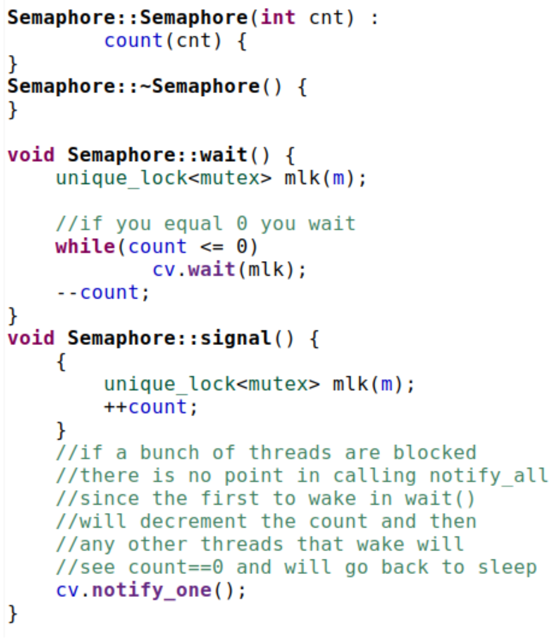
Semaphores

- Must initialize count. It corresponds to how many at once
- wait(): if count==0 then thread is blocked, otherwise thread decrements count and proceeds

Semaphores

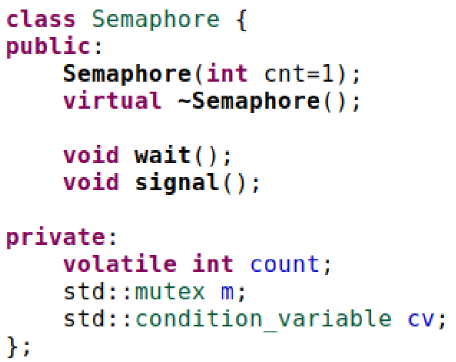


```
class Semaphore {
public:
    Semaphore(int cnt=1);
    virtual ~Semaphore();

    void wait();
    void signal();

private:
    volatile int count;
    std::mutex m;
    std::condition_variable cv;
};
```

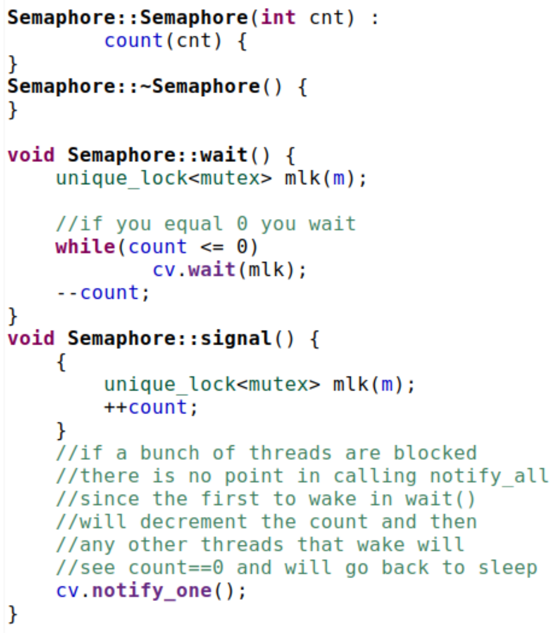


```
Semaphore::Semaphore(int cnt) :
        count(cnt) {
}
Semaphore::~Semaphore() {
}

void Semaphore::wait() {
    unique_lock<mutex> mlk(m);

    //if you equal 0 you wait
    while(count <= 0)
            cv.wait(mlk);
    --count;
}
void Semaphore::signal() {
    {
        unique_lock<mutex> mlk(m);
        ++count;
    }
    //if a bunch of threads are blocked
    //there is no point in calling notify_all
    //since the first to wake in wait()
    //will decrement the count and then
    //any other threads that wake will
    //see count==0 and will go back to sleep
    cv.notify_one();
}
```

- Must initialize count. It corresponds to how many at once
- wait(): if count==0 then thread is blocked, otherwise thread decrements count and proceeds
- signal(): increments count, and notifies one of the waiting threads (if any), thread wakes decrements count and proceeds



```
class Semaphore {
public:
    Semaphore(int cnt=1);
    virtual ~Semaphore();

    void wait();
    void signal();

private:
    volatile int count;
    std::mutex m;
    std::condition_variable cv;
};
```



```
Semaphore::Semaphore(int cnt) :
        count(cnt) {
}
Semaphore::~Semaphore() {
}

void Semaphore::wait() {
    unique_lock<mutex> mlk(m);

    //if you equal 0 you wait
    while(count <= 0)
            cv.wait(mlk);
    --count;
}
void Semaphore::signal() {
    {
        unique_lock<mutex> mlk(m);
        ++count;
    }
    //if a bunch of threads are blocked
    //there is no point in calling notify_all
    //since the first to wake in wait()
    //will decrement the count and then
    //any other threads that wake will
    //see count==0 and will go back to sleep
    cv.notify_one();
}
```

Semaphores

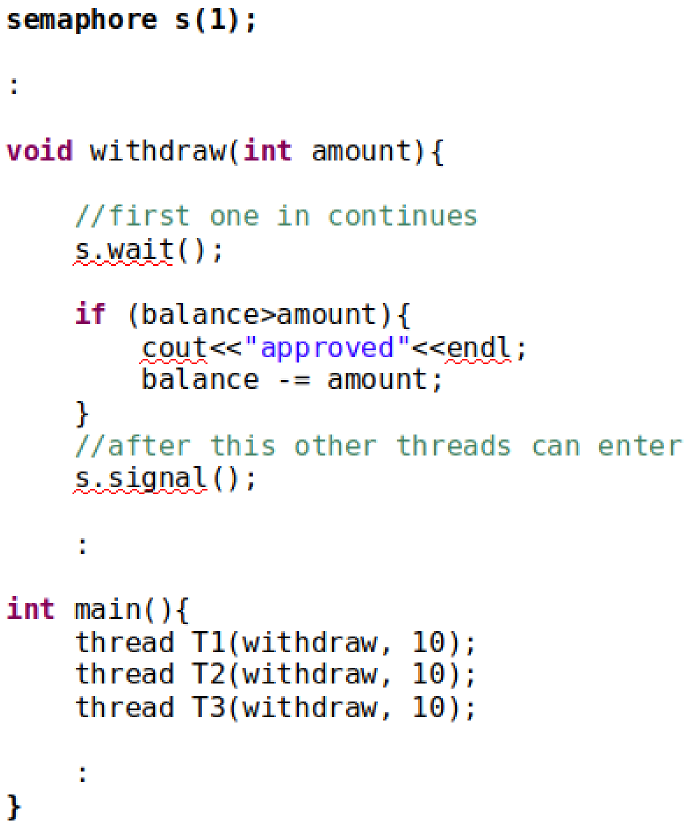
- Must initialize count. It corresponds to how many at once
- wait(): if count==0 then thread is blocked, otherwise thread decrements count and proceeds
- signal(): increments count, and notifies one of the waiting threads (if any), thread wakes decrements count and proceeds

Also, you Can signal and wait on different threads!
BTW for mutexes you must lock and unlock on the SAME thread.

Semaphores

An Example

Semaphores

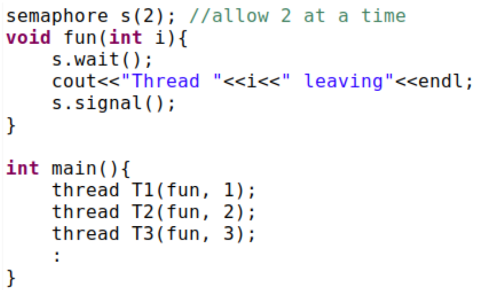


```
semaphore s(1);

:

void withdraw(int amount){

    //first one in continues
    s.wait();

    if (balance>amount){
        cout<<"approved"<<endl;
        balance -= amount;
    }
    //after this other threads can enter
    s.signal();

    :

int main(){
    thread T1(withdraw, 10);
    thread T2(withdraw, 10);
    thread T3(withdraw, 10);

    :
}
```

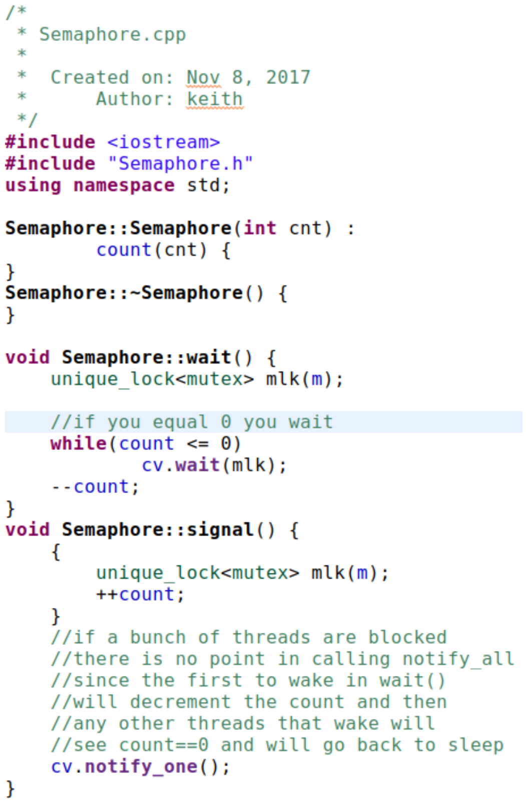
- An Example:
- Semaphore initialized to 1
- This is known as a binary semaphore
- It acts like a mutex.

Semaphores

Another Example



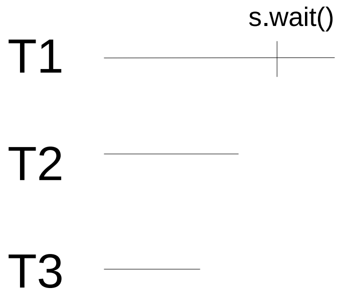
```
semaphore s(2); //allow 2 at a time
void fun(int i){
    s.wait();
    cout<<"Thread "<<i<<" leaving"<<endl;
    s.signal();
}

int main(){
    thread T1(fun, 1);
    thread T2(fun, 2);
    thread T3(fun, 3);
    :
}
```

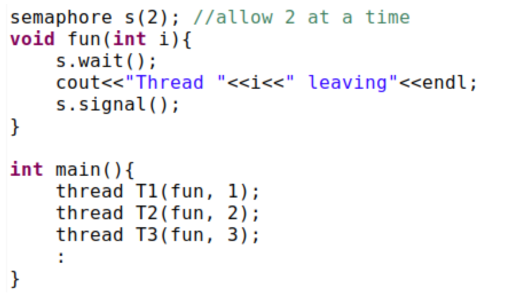


```
/*
 * Semaphore.cpp
 *
 *  Created on: Nov 8, 2017
 *      Author: keith
 */
#include <iostream>
#include "Semaphore.h"
using namespace std;

Semaphore::Semaphore(int cnt) :
        count(cnt) {
}
Semaphore::~Semaphore() {
}

void Semaphore::wait() {
    unique_lock<mutex> mlk(m);

    //if you equal 0 you wait
    while(count <= 0)
            cv.wait(mlk);
    --count;
}
void Semaphore::signal() {
    {
        unique_lock<mutex> mlk(m);
        ++count;
    }
    //if a bunch of threads are blocked
    //there is no point in calling notify_all
    //since the first to wake in wait()
    //will decrement the count and then
    //any other threads that wake will
    //see count==0 and will go back to sleep
    cv.notify_one();
}
```

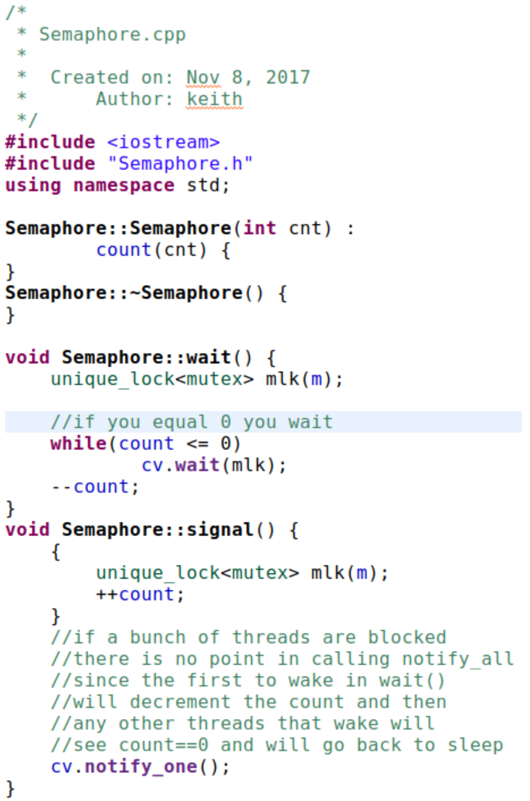
Assumming the order that threads start is (T1,T2,T3). Here is what happens:

T1 calls s.wait()





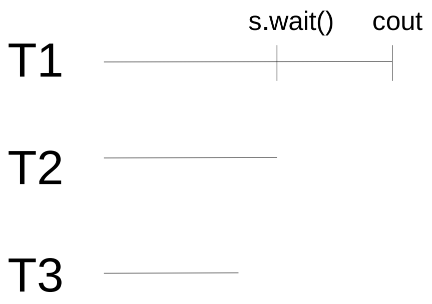
```
semaphore s(2); //allow 2 at a time
void fun(int i){
    s.wait();
    cout<<"Thread "<<i<<" leaving"<<endl;
    s.signal();
}

int main(){
    thread T1(fun, 1);
    thread T2(fun, 2);
    thread T3(fun, 3);
    :
}
```

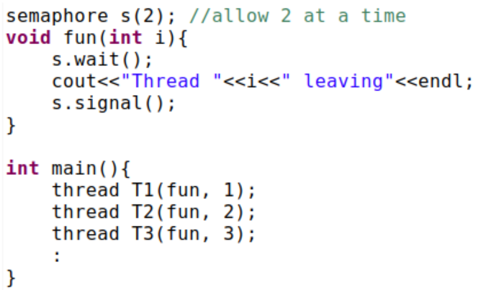


```
/*
 * Semaphore.cpp
 *
 *  Created on: Nov 8, 2017
 *      Author: keith
 */
#include <iostream>
#include "Semaphore.h"
using namespace std;

Semaphore::Semaphore(int cnt) :
        count(cnt) {
}
Semaphore::~Semaphore() {
}

void Semaphore::wait() {
    unique_lock<mutex> mlk(m);

    //if you equal 0 you wait
    while(count <= 0)
            cv.wait(mlk);
    --count;
}
void Semaphore::signal() {
    {
        unique_lock<mutex> mlk(m);
        ++count;
    }
    //if a bunch of threads are blocked
    //there is no point in calling notify_all
    //since the first to wake in wait()
    //will decrement the count and then
    //any other threads that wake will
    //see count==0 and will go back to sleep
    cv.notify_one();
}
```

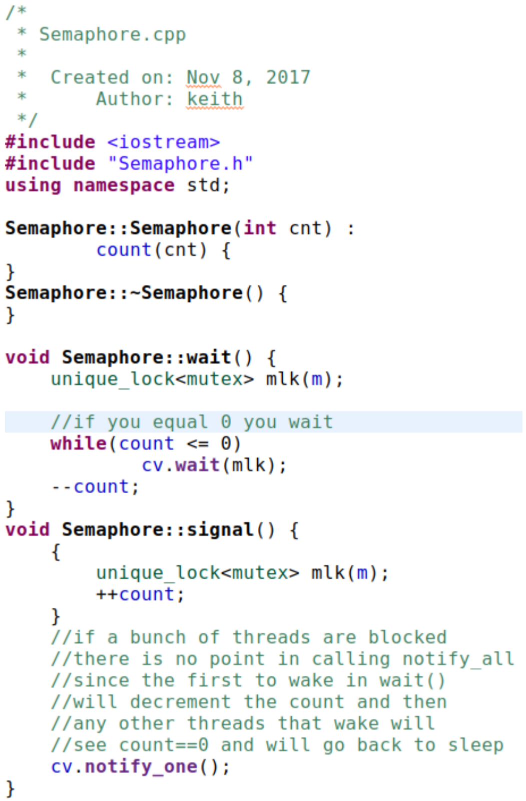
Assumming the order that threads start is (T1,T2,T3). Here is what happens:

T1 calls s.wait() after which s.count==1, T1 Then couts…





```
semaphore s(2); //allow 2 at a time
void fun(int i){
    s.wait();
    cout<<"Thread "<<i<<" leaving"<<endl;
    s.signal();
}

int main(){
    thread T1(fun, 1);
    thread T2(fun, 2);
    thread T3(fun, 3);
    :
}
```

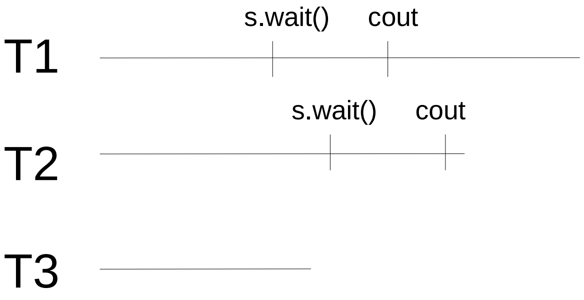


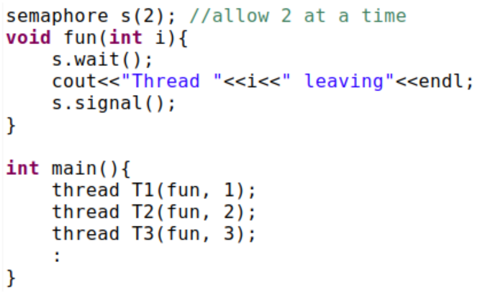
```
/*
 * Semaphore.cpp
 *
 *  Created on: Nov 8, 2017
 *      Author: keith
 */
#include <iostream>
#include "Semaphore.h"
using namespace std;

Semaphore::Semaphore(int cnt) :
        count(cnt) {
}
Semaphore::~Semaphore() {
}

void Semaphore::wait() {
    unique_lock<mutex> mlk(m);

    //if you equal 0 you wait
    while(count <= 0)
            cv.wait(mlk);
    --count;
}
void Semaphore::signal() {
    {
        unique_lock<mutex> mlk(m);
        ++count;
    }
    //if a bunch of threads are blocked
    //there is no point in calling notify_all
    //since the first to wake in wait()
    //will decrement the count and then
    //any other threads that wake will
    //see count==0 and will go back to sleep
    cv.notify_one();
}
```

Assumming the order that threads start is (T1,T2,T3). Here is what happens:

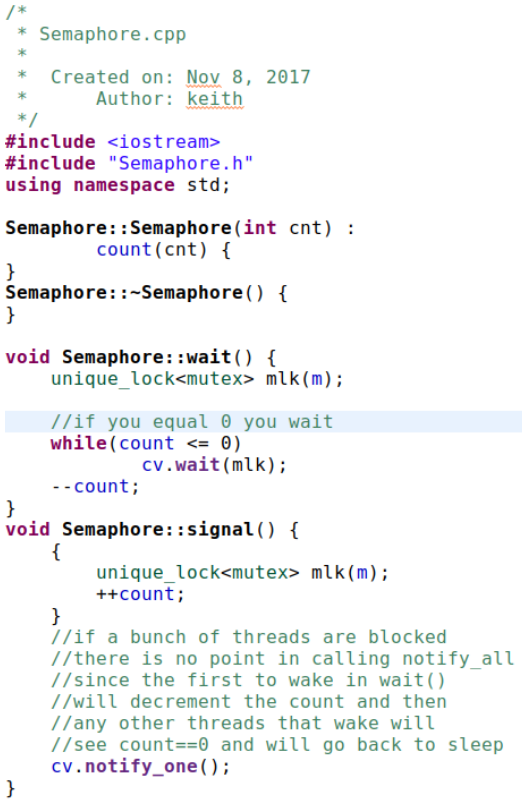
T1 calls s.wait() after which s.count==1, T1 Then couts its leaving

T2 calls s.wait() after which s.count==0, T2 Then couts its leaving





```
semaphore s(2); //allow 2 at a time
void fun(int i){
    s.wait();
    cout<<"Thread "<<i<<" leaving"<<endl;
    s.signal();
}

int main(){
    thread T1(fun, 1);
    thread T2(fun, 2);
    thread T3(fun, 3);
    :
}
```



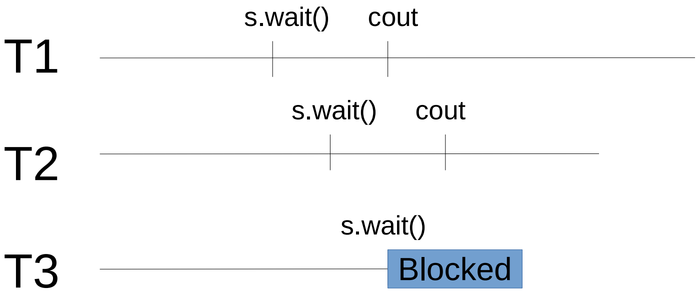
```
/*
 * Semaphore.cpp
 *
 *  Created on: Nov 8, 2017
 *      Author: keith
 */
#include <iostream>
#include "Semaphore.h"
using namespace std;

Semaphore::Semaphore(int cnt) :
        count(cnt) {
}
Semaphore::~Semaphore() {
}

void Semaphore::wait() {
    unique_lock<mutex> mlk(m);

    //if you equal 0 you wait
    while(count <= 0)
            cv.wait(mlk);
    --count;
}
void Semaphore::signal() {
    {
        unique_lock<mutex> mlk(m);
        ++count;
    }
    //if a bunch of threads are blocked
    //there is no point in calling notify_all
    //since the first to wake in wait()
    //will decrement the count and then
    //any other threads that wake will
    //see count==0 and will go back to sleep
    cv.notify_one();
}
```

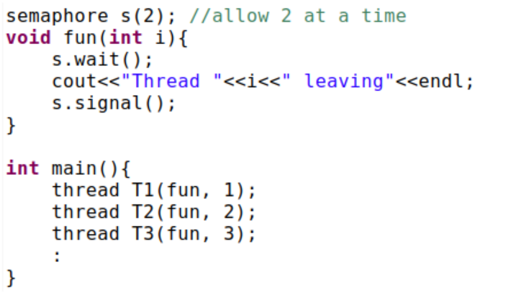
Assumming the order that threads start is (T1,T2,T3). Here is what happens:

T1 calls s.wait() after which s.count==1, T1 Then couts its leaving

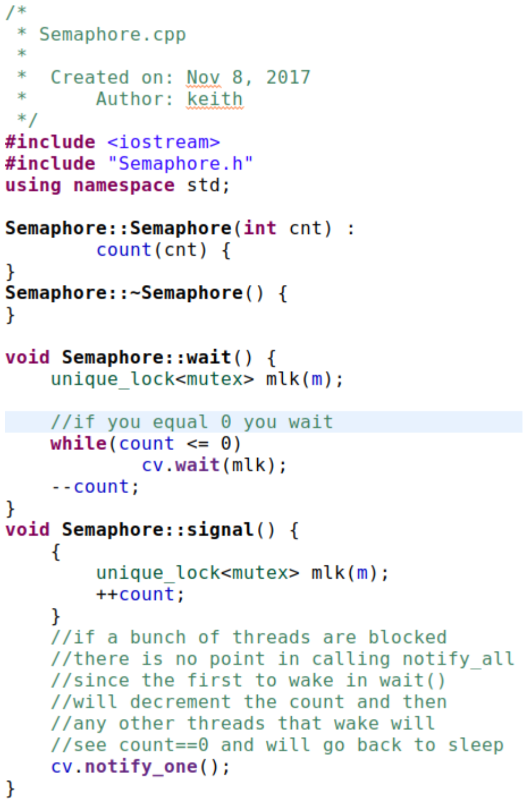
T2 calls s.wait() after which s.count==0, T2 Then couts its leaving

T3 calls s.wait() and Blocks...





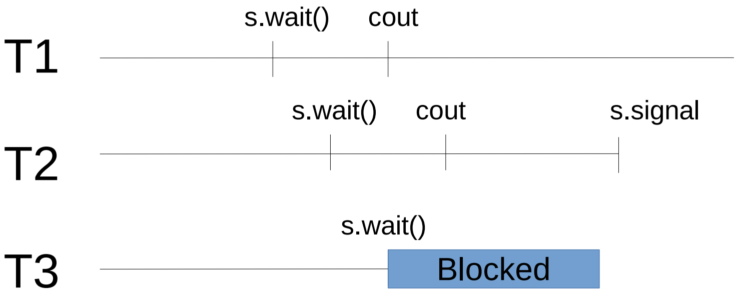
```
semaphore s(2); //allow 2 at a time
void fun(int i){
    s.wait();
    cout<<"Thread "<<i<<" leaving"<<endl;
    s.signal();
}

int main(){
    thread T1(fun, 1);
    thread T2(fun, 2);
    thread T3(fun, 3);
    :
}
```

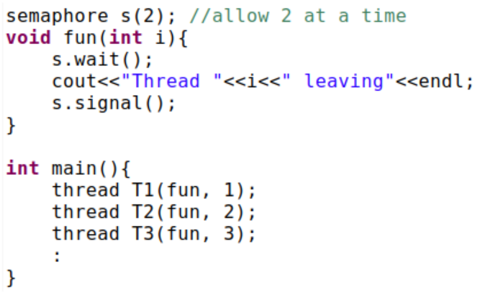


```
/*
 * Semaphore.cpp
 *
 *  Created on: Nov 8, 2017
 *      Author: keith
 */
#include <iostream>
#include "Semaphore.h"
using namespace std;

Semaphore::Semaphore(int cnt) :
        count(cnt) {
}
Semaphore::~Semaphore() {
}

void Semaphore::wait() {
    unique_lock<mutex> mlk(m);

    //if you equal 0 you wait
    while(count <= 0)
            cv.wait(mlk);
    --count;
}
void Semaphore::signal() {
    {
        unique_lock<mutex> mlk(m);
        ++count;
    }
    //if a bunch of threads are blocked
    //there is no point in calling notify_all
    //since the first to wake in wait()
    //will decrement the count and then
    //any other threads that wake will
    //see count==0 and will go back to sleep
    cv.notify_one();
}
```

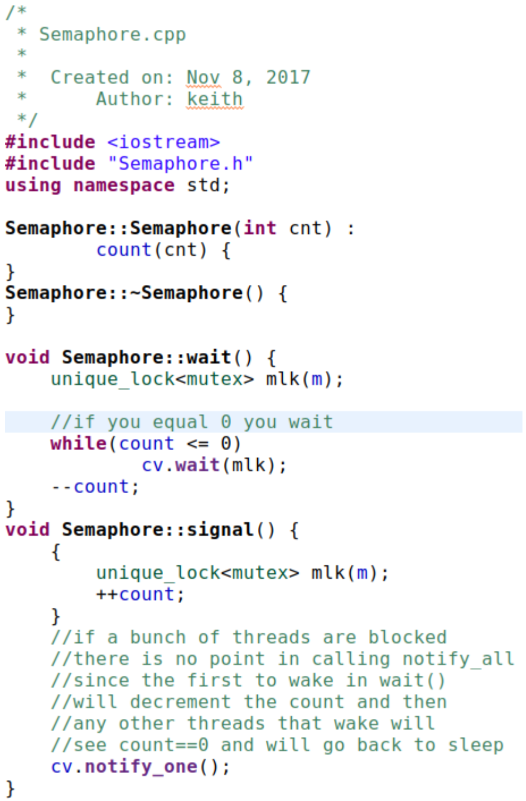
Assumming the order that threads start is (T1,T2,T3). Here is what happens:

T1 calls s.wait() after which s.count==1, T1 Then couts its leaving
T2 calls s.wait() after which s.count==0, T2 Then couts its leaving
T3 calls s.wait() and Blocks…
T2 (or T1) calls s.signal after which s.count==1,





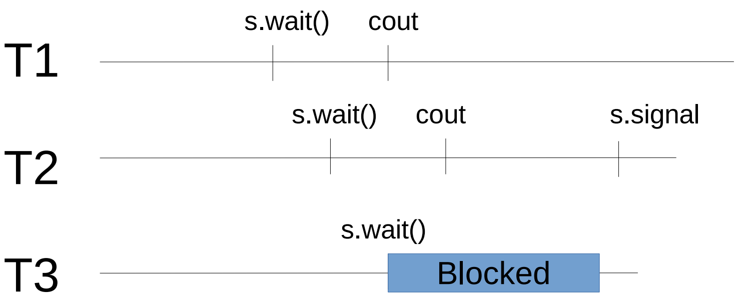
```
semaphore s(2); //allow 2 at a time
void fun(int i){
    s.wait();
    cout<<"Thread "<<i<<" leaving"<<endl;
    s.signal();
}

int main(){
    thread T1(fun, 1);
    thread T2(fun, 2);
    thread T3(fun, 3);
    :
}
```

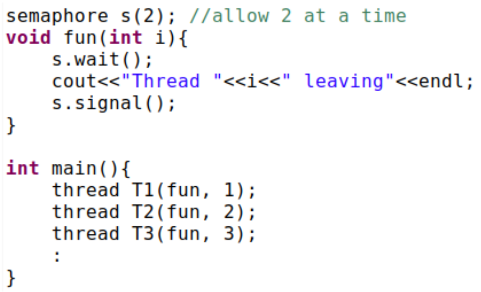


```
/*
 * Semaphore.cpp
 *
 *  Created on: Nov 8, 2017
 *      Author: keith
 */
#include <iostream>
#include "Semaphore.h"
using namespace std;

Semaphore::Semaphore(int cnt) :
        count(cnt) {
}
Semaphore::~Semaphore() {
}

void Semaphore::wait() {
    unique_lock<mutex> mlk(m);

    //if you equal 0 you wait
    while(count <= 0)
            cv.wait(mlk);
    --count;
}
void Semaphore::signal() {
    {
        unique_lock<mutex> mlk(m);
        ++count;
    }
    //if a bunch of threads are blocked
    //there is no point in calling notify_all
    //since the first to wake in wait()
    //will decrement the count and then
    //any other threads that wake will
    //see count==0 and will go back to sleep
    cv.notify_one();
}
```

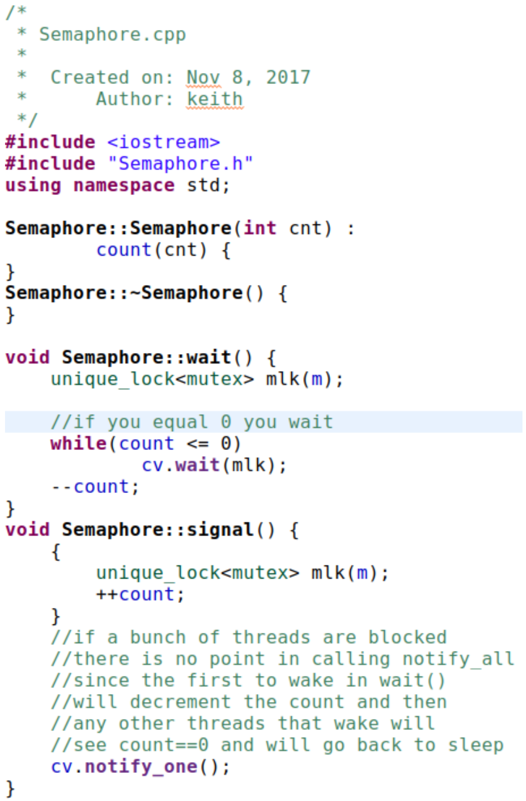
Assumming the order that threads start is (T1,T2,T3). Here is what happens:

T1 calls s.wait() after which s.count==1, T1 Then couts its leaving
T2 calls s.wait() after which s.count==0, T2 Then couts its leaving
T3 calls s.wait() and Blocks…
T2 (or T1) calls s.signal after which s.count==1
T3 awakes as soon as T2 signals, decrements count (count==0), works...





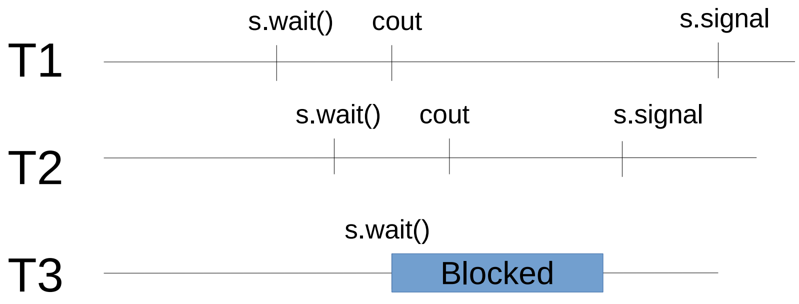
```
semaphore s(2); //allow 2 at a time
void fun(int i){
    s.wait();
    cout<<"Thread "<<i<<" leaving"<<endl;
    s.signal();
}

int main(){
    thread T1(fun, 1);
    thread T2(fun, 2);
    thread T3(fun, 3);
    :
}
```

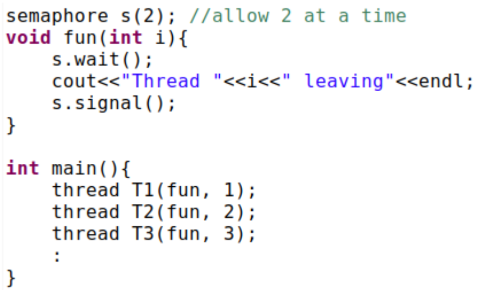


```
/*
 * Semaphore.cpp
 *
 *  Created on: Nov 8, 2017
 *      Author: keith
 */
#include <iostream>
#include "Semaphore.h"
using namespace std;

Semaphore::Semaphore(int cnt) :
        count(cnt) {
}
Semaphore::~Semaphore() {
}

void Semaphore::wait() {
    unique_lock<mutex> mlk(m);

    //if you equal 0 you wait
    while(count <= 0)
            cv.wait(mlk);
    --count;
}
void Semaphore::signal() {
    {
        unique_lock<mutex> mlk(m);
        ++count;
    }
    //if a bunch of threads are blocked
    //there is no point in calling notify_all
    //since the first to wake in wait()
    //will decrement the count and then
    //any other threads that wake will
    //see count==0 and will go back to sleep
    cv.notify_one();
}
```

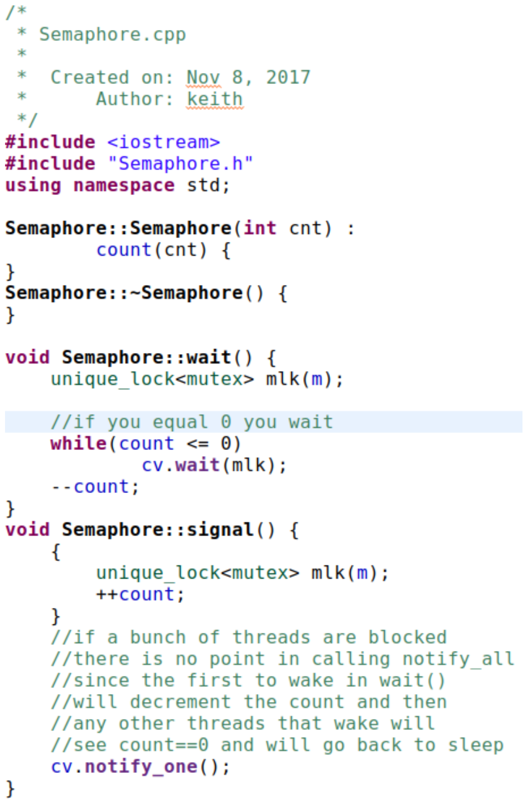
Assumming the order that threads start is (T1,T2,T3). Here is what happens:

T1 calls s.wait() after which s.count==1, T1 Then couts its leaving
T2 calls s.wait() after which s.count==0, T2 Then couts its leaving
T3 calls s.wait() and Blocks…
T2 (or T1) calls s.signal after which s.count==1
T3 awakes as soon as T2 signals, decrements count (s.count==0), works…
T1 calls s.signal after which s.count==1





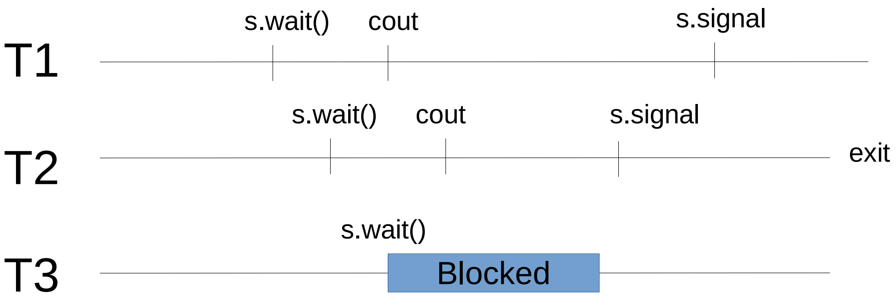
```
semaphore s(2); //allow 2 at a time
void fun(int i){
    s.wait();
    cout<<"Thread "<<i<<" leaving"<<endl;
    s.signal();
}

int main(){
    thread T1(fun, 1);
    thread T2(fun, 2);
    thread T3(fun, 3);
    :
}
```

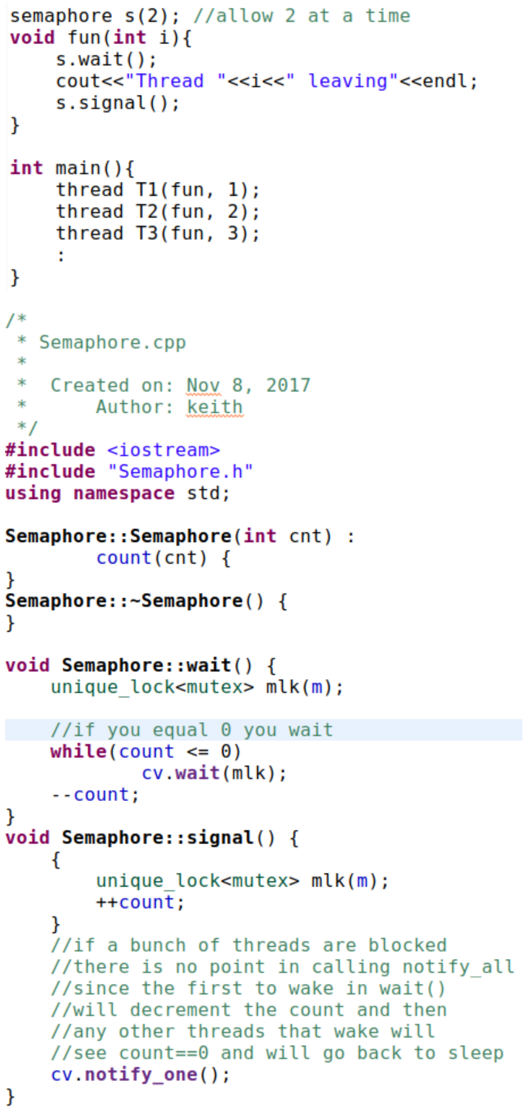


```
/*
 * Semaphore.cpp
 *
 *  Created on: Nov 8, 2017
 *      Author: keith
 */
#include <iostream>
#include "Semaphore.h"
using namespace std;

Semaphore::Semaphore(int cnt) :
        count(cnt) {
}
Semaphore::~Semaphore() {
}

void Semaphore::wait() {
    unique_lock<mutex> mlk(m);

    //if you equal 0 you wait
    while(count <= 0)
            cv.wait(mlk);
    --count;
}
void Semaphore::signal() {
    {
        unique_lock<mutex> mlk(m);
        ++count;
    }
    //if a bunch of threads are blocked
    //there is no point in calling notify_all
    //since the first to wake in wait()
    //will decrement the count and then
    //any other threads that wake will
    //see count==0 and will go back to sleep
    cv.notify_one();
}
```

Assumming the order that threads start is (T1,T2,T3). Here is what happens:

T1 calls s.wait() after which s.count==1, T1 Then couts its leaving
T2 calls s.wait() after which s.count==0, T2 Then couts its leaving
T3 calls s.wait() and Blocks…
T2 (or T1) calls s.signal after which s.count==1
T3 awakes as soon as T2 signals, decrements count (s.count==0), works…
T1 calls s.signal after which s.count==1
T2 exits

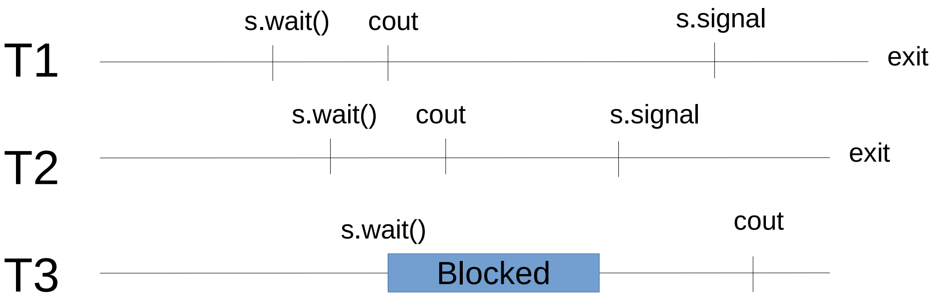


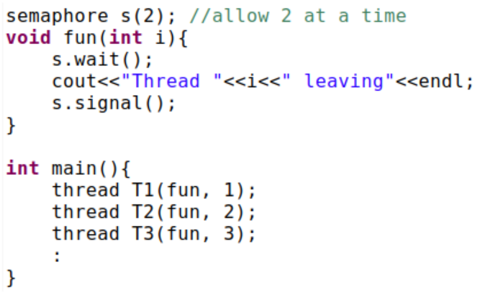


```
semaphore s(2); //allow 2 at a time
void fun(int i){
    s.wait();
    cout<<"Thread "<<i<<" leaving"<<endl;
    s.signal();
}

int main(){
    thread T1(fun, 1);
    thread T2(fun, 2);
    thread T3(fun, 3);
    :
}

/*
 * Semaphore.cpp
 *
 *  Created on: Nov 8, 2017
 *      Author: keith
 */
#include <iostream>
#include "Semaphore.h"
using namespace std;

Semaphore::Semaphore(int cnt) :
        count(cnt) {
}
Semaphore::~Semaphore() {
}

void Semaphore::wait() {
    unique_lock<mutex> mlk(m);

    //if you equal 0 you wait
    while(count <= 0)
            cv.wait(mlk);
    --count;
}
void Semaphore::signal() {
    {
        unique_lock<mutex> mlk(m);
        ++count;
    }
    //if a bunch of threads are blocked
    //there is no point in calling notify_all
    //since the first to wake in wait()
    //will decrement the count and then
    //any other threads that wake will
    //see count==0 and will go back to sleep
    cv.notify_one();
}
```

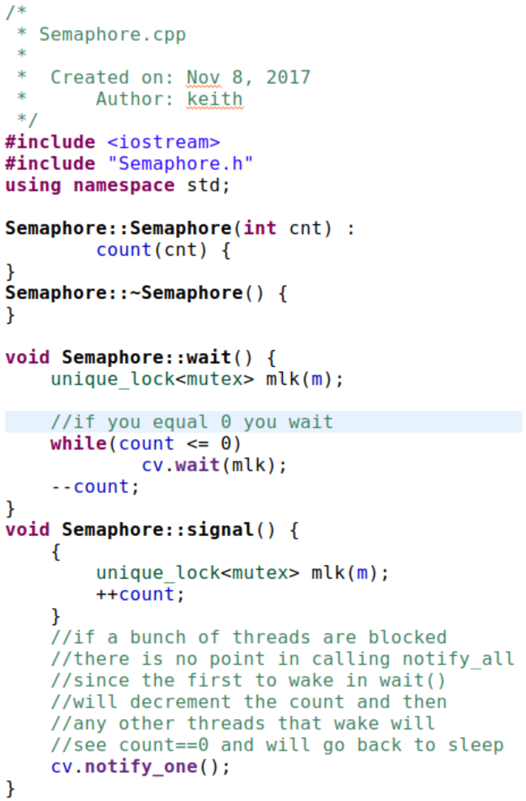
Assumming the order that threads start is (T1,T2,T3). Here is what happens:

T1 calls s.wait() after which s.count==1, T1 Then couts its leaving
T2 calls s.wait() after which s.count==0, T2 Then couts its leaving
T3 calls s.wait() and Blocks…
T2 (or T1) calls s.signal after which s.count==1
T3 awakes as soon as T2 signals, decrements count (s.count==0), works…
T1 calls s.signal after which s.count==1
T2 exits
T1 exits, T3 couts





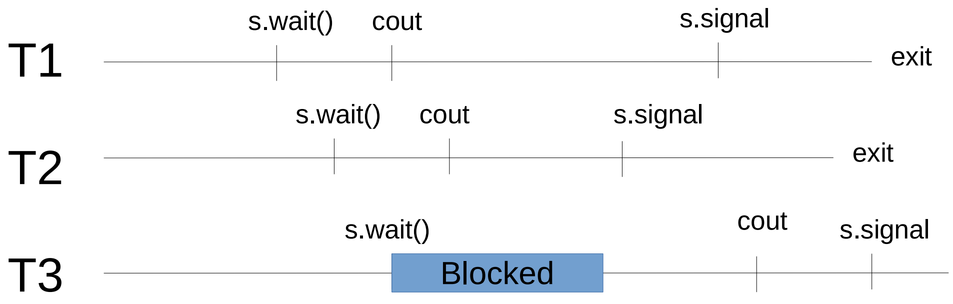
```
semaphore s(2); //allow 2 at a time
void fun(int i){
    s.wait();
    cout<<"Thread "<<i<<" leaving"<<endl;
    s.signal();
}

int main(){
    thread T1(fun, 1);
    thread T2(fun, 2);
    thread T3(fun, 3);
    :
}
```

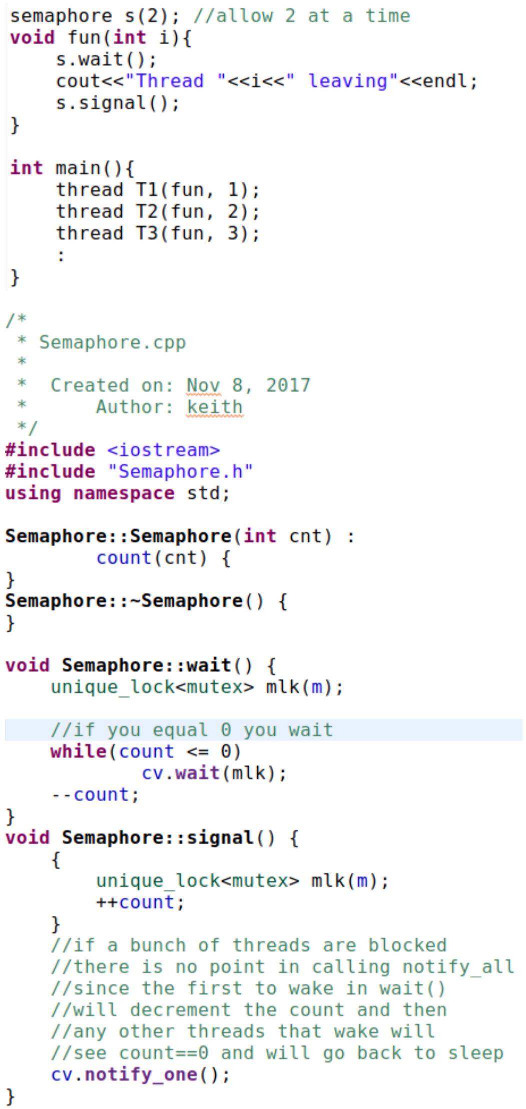


```
/*
 * Semaphore.cpp
 *
 *  Created on: Nov 8, 2017
 *      Author: keith
 */
#include <iostream>
#include "Semaphore.h"
using namespace std;

Semaphore::Semaphore(int cnt) :
        count(cnt) {
}
Semaphore::~Semaphore() {
}

void Semaphore::wait() {
    unique_lock<mutex> mlk(m);

    //if you equal 0 you wait
    while(count <= 0)
            cv.wait(mlk);
    --count;
}
void Semaphore::signal() {
    {
        unique_lock<mutex> mlk(m);
        ++count;
    }
    //if a bunch of threads are blocked
    //there is no point in calling notify_all
    //since the first to wake in wait()
    //will decrement the count and then
    //any other threads that wake will
    //see count==0 and will go back to sleep
    cv.notify_one();
}
```

Assumming the order that threads start is (T1,T2,T3). Here is what happens:

T1 calls s.wait() after which s.count==1, T1 Then couts its leaving
T2 calls s.wait() after which s.count==0, T2 Then couts its leaving
T3 calls s.wait() and Blocks…
T2 (or T1) calls s.signal after which s.count==1
T3 awakes as soon as T2 signals, decrements count (s.count==0), works…
T1 calls s.signal after which s.count==1
T2 exits
T1 exits, T3 couts
T3 calls s.signal after which s.count==2



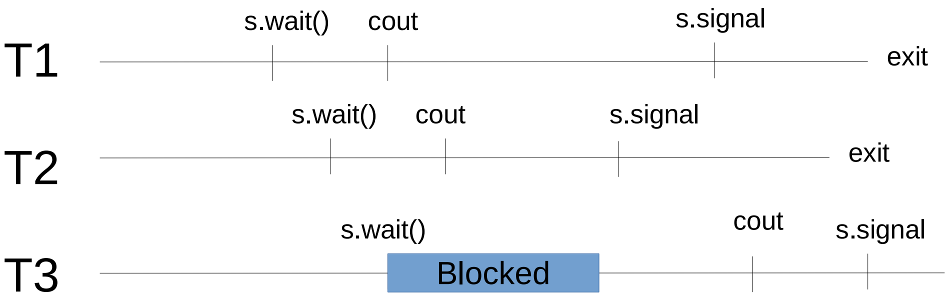


```
semaphore s(2); //allow 2 at a time
void fun(int i){
    s.wait();
    cout<<"Thread "<<i<<" leaving"<<endl;
    s.signal();
}

int main(){
    thread T1(fun, 1);
    thread T2(fun, 2);
    thread T3(fun, 3);
    :
}

/*
 * Semaphore.cpp
 *
 *  Created on: Nov 8, 2017
 *      Author: keith
 */
#include <iostream>
#include "Semaphore.h"
using namespace std;

Semaphore::Semaphore(int cnt) :
        count(cnt) {
}
Semaphore::~Semaphore() {
}

void Semaphore::wait() {
    unique_lock<mutex> mlk(m);

    //if you equal 0 you wait
    while(count <= 0)
            cv.wait(mlk);
    --count;
}
void Semaphore::signal() {
    {
        unique_lock<mutex> mlk(m);
        ++count;
    }
    //if a bunch of threads are blocked
    //there is no point in calling notify_all
    //since the first to wake in wait()
    //will decrement the count and then
    //any other threads that wake will
    //see count==0 and will go back to sleep
    cv.notify_one();
}
```

Assumming the order that threads start is (T1,T2,T3). Here is what happens:

T1 calls s.wait() after which s.count==1, T1 Then couts its leaving
T2 calls s.wait() after which s.count==0, T2 Then couts its leaving
T3 calls s.wait() and Blocks…
T2 (or T1) calls s.signal after which s.count==1
T3 awakes as soon as T2 signals, decrements count (s.count==0), works…
T1 calls s.signal after which s.count==1
T2 exits
T1 exits, T3 couts
T3 calls s.signal after which s.count==2
T3 exits

Notice that the Semaphores count is once again 2



Semaphores

Go to project
410_Semaphore_ConditionVar_Mutex_Thread_Producer_Consumer