

Condition Variables

How do you signal between threads?

- What if you want to ensure that one thread starts before another (1 thread to wait until another thread says go)
- Or to check if a condition is true before proceeding?

What we want:

```
//a global  
bool isReady=false;
```

Thread 1

Efficiently waiting on isReady equal True

Thread 2

sets isReady=true and then wakes up thread one

How do you signal between threads?

- As an example, If I launch 2 threads;
 - 1 to deposit \$100
 - 1 to withdraw \$90
- Want the deposit to finish first! (otherwise overdraft)
- How to do this

How do you signal between threads?

- As an example, If I launch 2 threads;
 - 1 to withdraw \$90
 - 1 to deposit \$100
- Want the deposit to finish first! (otherwise overdraft)
- How to do this

```
mutex m;  
int balance = 0;
```

Thread 1

```
void withdraw(int amount){  
  
    lock_guard<mutex> lck(m);  
    balance-=amount;  
    if (balance < 0)  
        cout<<"You are overdrawn"<<endl;  
    else  
        cout<<"no worries"<<endl;  
}
```

Thread 2

```
void deposit(int amount){  
    //deposit is delayed!  
    std::this_thread::sleep_for(std::chrono::milliseconds(100));  
  
    lock_guard<mutex> lck(m);    //waiting for mutex  
    balance +=amount;  
}
```

Can use a spin lock

- Works but at a cost!
- High CPU usage!
- See [Simple Condition Variable](#) project, call up System Monitor and watch one cores usage spike.

```
mutex m;  
int balance = 0;  
bool deposit_made = false;
```

Thread 1

```
void withdraw(int amount){  
    //the bad idea, a busy wait  
    while (!deposit_made){}  
  
    lock_guard<mutex> lck(m);  
    balance-=amount;  
    if (balance <0)  
        cout<<"You are overdrawn"<<endl;  
    else  
        cout<<"no worries"<<endl;  
}
```

Thread 2

```
void deposit(int amount){  
    //deposit is delayed!  
    std::this_thread::sleep_for(std::chrono::milli  
  
    lock_guard<mutex> lck(m);    //waiting for mut  
    balance +=amount;  
    deposit_made=true;  
}
```

Get rid of that spinning

- Instead **would like to put the waiting thread to sleep** until the event occurs...
- And then wake it up
- No more spiking CPU core usage!

Use condition variables

- First the proper include (C++11 and above)

```
#include <condition_variable>
```

- And a new kind of lock object

```
//just like a lock_guard  
//PLUS you can manually unlock it!  
unique_lock<mutex> lck(m);
```

- Works just like a lock guard
- Except you can manually unlock it
- **AND IT'S THE ONLY KIND OF LOCK A CONDITION_VARIABLE CAN WAIT ON**

Use condition variables

```
mutex m;  
int balance = 0;  
bool deposit_made = false;  
condition_variable cv;
```

Thread 1

```
void withdraw(int amount){  
    //just like a lock_guard  
    //PLUS you can manually unlock it!  
    unique_lock<mutex> lck(m);  
  
    //MUST be a loop to handle  
    //spurious wakeups  
    while(!deposit_made)  
    {  
        cv.wait(lck);    //if here, release lock  
                          //and then sleep until  
                          //awakened  
        balance-=amount;  
        if (balance <0)  
            cout<<"You are overdrawn"<<endl;  
        else  
            cout<<"no worries"<<endl;  
    }  
}
```

Thread 2

```
void deposit(int amount){  
    std::this_thread::sleep_for(std::chrono::n  
    {  
        lock_guard<mutex> lck(m);  
        balance +=amount;  
  
        //must change condition while locked!  
        deposit_made=true;  
    }  
  
    //must release lock before notify  
    //notify_all wakes ALL threads  
    //waiting on this cv. One will  
    //acquire the mutex, check condition  
    //in while and move forward  
    //the others go back to sleep  
    cv.notify_all();  
}
```


Use condition variables

```
mutex m;  
int balance = 0;  
bool deposit_made = false;  
condition_variable cv;
```

Thread 1

```
void withdraw(int amount){  
    //just like a lock_guard  
    //PLUS you can manually unlock it!  
    unique_lock<mutex> lck(m);  
  
    //MUST be a loop to handle  
    //spurious wakeups  
    while(!deposit_made)  
    {  
        cv.wait(lck);    //if here, release lock  
                          //and then sleep until  
                          //awakened  
    }  
    balance-=amount;  
    if (balance < 0)  
        cout<<"You are overdrawn"<<endl;  
    else  
        cout<<"no worries"<<endl;  
}
```

Thread 2

```
void deposit(int amount){  
    std::this_thread::sleep_for(std::chrono::seconds(1));  
    {  
        lock_guard<mutex> lck(m);  
        balance+=amount;  
        //must change condition while locked!  
        deposit_made = true;  
    }  
  
    //must release lock before notify  
    //notify_all wakes ALL threads  
    //waiting on this cv. One will  
    //acquire the mutex, check condition  
    //in while and move forward  
    //the others go back to sleep  
    cv.notify_all();  
}
```

Can be any expression that evaluates to a bool
for instance;
(!deposit_made && !likes_dogs)

Use condition variables

```
mutex m;  
int balance = 0;  
bool deposit_made = false;  
condition_variable cv;
```

Thread 1

```
void withdraw(int amount){  
    //just like a lock_guard  
    //PLUS you can manually unlock it!  
    unique_lock<mutex> lck(m);  
  
    //MUST be a loop to handle  
    //spurious wakeups  
    while(!deposit_made){  
        cv.wait(lck); //if here, release lock  
                       //and then sleep until  
                       //awakened  
        balance-=amount;  
        if (balance <0)  
            cout<<"You are overdrawn"<<endl;  
        else  
            cout<<"no worries"<<endl;  
    }  
}
```

Thread 2

```
void deposit(int amount){  
    std::this_thread::sleep_for(std::chrono::n  
    {  
        lock_guard<mutex> lck(m);  
        balance +=amount;  
  
        //must change condition while locked!  
        cv.notify_all();  
    }  
    //must release lock before notify  
    //notify_all wakes ALL threads  
    //waiting on this cv. One will  
    //acquire the mutex, check condition  
    //in while and move forward  
    //the others go back to sleep  
    cv.notify_all();  
}
```

← Note the `unique_lock`

← This **MUST** be a while loop so
Thread can go back to sleep if it wakes
and the condition evaluates to false

Use condition variables

```
mutex m;  
int balance = 0;  
bool deposit_made = false;  
condition_variable cv;
```

Thread 1

```
void withdraw(int amount){  
    //just like a lock_guard  
    //PLUS you can manually unlock it!  
    unique_lock<mutex> lck(m);  
    //MUST be a loop to handle  
    //spurious wakeups  
    while(!deposit_made)  
        cv.wait(lck);  
    balance-=amount;  
    if (balance < 0)  
        cout<<"You are overdrawn"<<endl;  
    else  
        cout<<"no worries"<<endl;  
}
```

Lock_guard, unique_lock
Whatever...just lock it!

Boolean used in waiting
threads while loop

Thread 2

```
void deposit(int amount){  
    std::this_thread::sleep_for(std::chrono::n  
    {  
        lock_guard<mutex> lck(m);  
        balance +=amount;  
  
        //must change condition while locked!  
        deposit_made=true;  
    }  
  
    //must release lock before notify  
    //notify_all wakes ALL threads  
    //waiting on this cv. One will  
    //acquire the mutex, check condition  
    //in while and move forward  
    //the others go back to sleep  
    cv.notify_all();  
}
```

Use condition variables

```
mutex m;  
int balance = 0;  
bool deposit_made = false;  
condition_variable cv;
```

Thread 1

```
void withdraw(int amount){  
    //just like a lock_guard  
    //PLUS you can manually unlock it!  
    unique_lock<mutex> lck(m);  
  
    //MUST be a loop to handle  
    //curious wakeups  
    while(!deposit_made)  
        cv.wait(lck); //if here, release lock  
    //and then sleep until  
    //awakened  
    balance-=amount;  
    if (balance < 0)  
        cout<<"You are overdrawn"<<endl;  
    else  
        cout<<"no worries"<<endl;  
}
```

Lock_guard goes out of scope here
Make sure you release mutex
before signaling!

Wake up every thread waiting
On condition_variable cv

Thread 2

```
void deposit(int amount){  
    std::this_thread::sleep_for(std::chrono::n  
    {  
        lock_guard<mutex> lck(m);  
        balance +=amount;  
  
        //must change condition while locked!  
        deposit_made=true;  
    }  
  
    //must release lock before notify  
    //notify_all wakes ALL threads  
    //waiting on this cv. One will  
    //acquire the mutex, check condition  
    //in while and move forward  
    //the others go back to sleep  
    cv.notify_all();  
}
```

Problem – Spurious wakeup

- **Spurious wakeup** – sometimes the receiver wakes up, although no notification happened. [POSIX Threads](#) and the [Windows API](#) are vulnerable to this.
- That's why we have the while loop.
- If thread awakes without condition being changed it just goes back to sleep.


```
//MUST be a loop to handle
//spurious wakeups
while(!deposit_made)←
    cv.wait(lck);    //if here, release lock
                    //and then sleep until
                    //awakened
balance-=amount;
if (balance <0)
    cout<<"You are overdrawn"<<endl;
else
    cout<<"no worries"<<endl;
}
```

This **MUST** be a while loop so
Thread can go back to sleep if it wakes
and the condition evaluates to false

```
balance +=amount;
//must change condition while locked!
//must release lock before notify
//notify_all wakes ALL threads
//waiting on this cv. One will
//acquire the mutex, check condition
//in while and move forward
//the others go back to sleep
cv.notify_all();
}
```

Problem – Lost Wakeup

- **Lost wakeup**– the sender sends its notification before the receiver waits on the cv. So the notification is lost ☹ (but not in this code)
- What happens if Thread2 notifies when Thread 1 is at any arrow below?
- (Keep in mind who holds the lock)
- No worries here



```
void withdraw(int amount){
    //just like a lock_guard
    //PLUS you can manually unlock it!
    unique_lock<mutex> lck(m);

    //MUST be a loop to handle
    //spurious wakeups
    while(!deposit_made)
    {
        cv.wait(lck);    //if here, release lock
                        //and then sleep until
                        //awakened
        balance-=amount;
        if (balance <0)
            cout<<"You are overdrawn"<<endl;
        else
            cout<<"no worries"<<endl;
    }
```

Thread 1

```
void deposit(int amount){
    std::this_thread::sleep_for(std::chrono::n
    {
        lock_guard<mutex> lck(m);
        balance +=amount;

        //must change condition while locked!
        deposit_made=true;
    }

    //must release lock before notify
    //notify_all wakes ALL threads
    //waiting on this cv. One will
    //acquire the mutex, check condition
    //in while and move forward
    //the others go back to sleep
    cv.notify_all();
}
```

Thread 2

Condition variables – notifying

- **notify_one();** wake just 1 thread waiting
- **notify_all();** wake them all, 1 will acquire the mutex, Go about its business, then the next acquires mutex, etc
- I use notify_all for convenience in this class because I often have more than 1 thread waiting on the condition_variable and I want them all to awake.
- If I used notify_one instead and had multiple threads, one would wake up and the rest would sleep.

Condition variables – waiting

- **'wait'** releases the mutex until condition_variable is signaled
- **cv.wait** wait until notified
- **cv.wait_for** timed version of wait
- **cv.wait_until** “

Other reading

- https://en.cppreference.com/w/cpp/thread/condition_variable
- See the condition variables projects on the course website