Mutual Exclusion (Mutex)

Protects critical sections that are longer than 1 line

Mutual Exclusion (Mutex)

- Enforcement
  - Only **1** thread in a critical section at a time
- Availability
  - If no thread in critical section, then any thread can enter
- Minimal Stay
  - Threads stay in critical section for minimal time
- Consistency
  - If resource must be protected anywhere, then it must be protected everywhere

Mutex – what it does

```
60  int balance=60;
61
62  void withdraw(int amt){
63
64      if(balance > amt){
65          cout<<"approved"<<endl;
66          balance-=amount;
67      }
68
69  }
70
71  int main(){
72      thread t1(withdraw,40);
73      thread t2(withdraw,25);
74      t1.join();
75      t2.join();
76  }
```

Mutex – what it does

```
60  int balance=60;
61
62  void withdraw(int amt){
63
64      if(balance > amt){
65          cout<<"approved"<<endl;
66          balance-=amount;
67      }
68
69  }
70
71  int main(){
72      thread t1(withdraw,40);
73      thread t2(withdraw,25);
74      t1.join();
75      t2.join();
76  }
```

As written, there is chance that a thread will be interrupted After line 64 but before line 66 executes.

Mutex – what it does

```
60  int balance=60;
61
62  void withdraw(int amt){
63
64      if(balance > amt){
65          cout<<"approved"<<endl;
66          balance-=amount;
67      }
68
69  }
70
71  int main(){
72      thread t1(withdraw,40);
73      thread t2(withdraw,25);
74      t1.join();
75      t2.join();
76  }
```

As written, there is chance that a thread will be interrupted After line 64 but before line 66 executes.

In this case the result will likely be an overdrawn account

Mutex – what it does

```
60  int balance=60;
61
62  void withdraw(int amt){
63
64      if(balance > amt){
65          cout<<"approved"<<endl;
66          balance-=amount;
67      }
68
69  }
70
71  int main(){
72      thread t1(withdraw,40);
73      thread t2(withdraw,25);
74      t1.join();
75      t2.join();
76  }
```

As written, there is chance that a thread will be interrupted
After line 64 but before line 66 executes.
In this case the result will likely be an overdrawn account

We want to ensure this never happens!

Mutex – what it does

```
60  int balance=60;
61
62  void withdraw(int amt){
63
64      if(balance > amt){
65          cout<<"approved"<<endl;
66          balance-=amount;
67      }
68
69  }
70
71  int main(){
72      thread t1(withdraw,40);
73      thread t2(withdraw,25);
74      t1.join();
75      t2.join();
76  }
```

First: Identify minimal critical section(s)

Mutex – what it does

```
60  int balance=60;
61  mutex m;
62  void withdraw(int amt){
63      m.lock();
64      if(balance > amt){
65          cout<<"approved"<<endl;
66          balance-=amount;
67      }
68      m.unlock();
69  }
70
71  int main(){
72      thread t1(withdraw,40);
73      thread t2(withdraw,25);
74      t1.join();
75      t2.join();
76  }
```

First: Identify minimal critical section(s)

Next: protect critical section(s) with a mutex

Mutex – what it does

```
60  int balance=60;
61  mutex m;
62  void withdraw(int amt){
63      m.lock();
64      if(balance > amt){
65          cout<<"approved"<<endl;
66          balance-=amount;
67      }
68      m.unlock();
69  }
70
71  int main(){
72      thread t1(withdraw,40);
73      thread t2(withdraw,25);
74      t1.join();
75      t2.join();
76  }
```

First: Identify minimal critical section(s)
Next: protect critical section(s) with a mutex

Only 1 thread can be executing code between lines 63 and 68 at a time.

Mutex – what it does

```
60  int balance=60;
61  mutex m;
62  void withdraw(int amt){
63      m.lock();
64      if(balance > amt){
65          cout<<"approved"<<endl;
66          balance-=amount;
67      }
68      m.unlock();
69  }
70
71  int main(){
72      thread t1(withdraw,40);
73      thread t2(withdraw,25);
74      t1.join();
75      t2.join();
76  }
```

t1

First: Identify minimal critical section(s)
Next: protect critical section(s) with a mutex

Only 1 thread can be executing code between lines 63 and 68 at a time.
Assume t1 is executing line 65

Mutex – what it does

```
60  int balance=60;
61  mutex m;
62  void withdraw(int amt){
63      m.lock();   <-- t2 is blocked here
64      if(balance > amt){
65          cout<<"approved"<<endl;  <-- t1
66          balance-=amount;
67      }
68      m.unlock();
69  }
70
71  int main(){
72      thread t1(withdraw,40);
73      thread t2(withdraw,25);
74      t1.join();
75      t2.join();
76  }
```

First: Identify minimal critical section(s)
Next: protect critical section(s) with a mutex

Only 1 thread can be executing code between lines 63 and 68 at a time.
Assume t1 is executing line 65
t2 is likely blocked at line 63, waiting for t1 to finish executing line 68.

Mutex – what it does

```
60  int balance=60;
61  mutex m;
62  void withdraw(int amt){
63      m.lock();
64      if(balance > amt){
65          cout<<"approved"<<endl;
66          balance-=amount;
67      }
68      m.unlock();
69  }
70
71  int main(){
72      thread t1(withdraw,40);
73      thread t2(withdraw,25);
74      t1.join();
75      t2.join();
76  }
```

First: Identify minimal critical section(s)
Next: protect critical section(s) with a mutex

Only 1 thread can be executing code between lines 63 and 68 at a time.
Assume t1 is executing line 65
t2 is blocked at line 63, waiting for t1 to finish executing line 68.
Once t1 executes line 68, t2 is free to acquire the mutex and proceed

Mutex – how to implement

- Hardware enforced
  - Disable interrupts
  - Guarantees atomic code because your code cannot be interrupted

Mutex – how to implement

- Hardware enforced
  - Disable interrupts
  - Guarantees atomic code because your code cannot be interrupted
- But…
  - Cannot have overlapping critical sections
  - Cannot switch to other, non-related, processes
  - Will not work on multi-core system unless you disable interrupts on all cores (big performance hit)
  - Kills performance on core

Mutex – how to implement

- Hardware enforced
  - Disable interrupts
  - Guarantees atomic code because your code cannot be interrupted
- But…
  - Cannot have overlapping critical sections
  - Cannot switch to other, non-related, processes
  - Will not work on multi-core system unless you disable interrupts on all cores (big performance hit)
  - Kills performance on core
- So… Cannot use disabled interrupts solution

Mutex – how to implement

```
1   //a special atomic function
2   int compare_and_swap(int *word, int notlockedval, int lockedval){
3       //save original
4       int oldval=*word;
5
6       if(oldval==notlockedval)    //if we can
7           *word=lockedval         //then do
8       return oldval;
9   }
10
11  const int notlockedval=0;
12  const int lockedval=1;
13
14  int word=notlockedval;  //start unlocked
15
16  void withdraw(int amt){
17
18      //stay in below loop until compare_and_swap returns notlockedval
19      while (compare_and_swap(&word,notlockedval, lockedval ) == lockedval){}
20      if(balance > amt){
21          cout<<"approved"<<endl;
22          balance-=amount;
23      }
24      word= notlockedval;
25  }
```

Compare and Swap – an atomic operation (once started cannot be interrupted)

Mutex – how to implement

```
1   //a special atomic function
2   int compare_and_swap(int *word, int notlockedval, int lockedval){
3       //save original
4       int oldval=*word;
5
6       if(oldval==notlockedval)    //if we can
7           *word=lockedval         //then do
8       return oldval;
9   }
10
11  const int notlockedval=0;
12  const int lockedval=1;
13
14  int word=notlockedval;  //start unlocked
15
16  void withdraw(int amt){
17
18      //stay in below loop until compare_and_swap returns notlockedval
19      while (compare_and_swap(&word,notlockedval, lockedval ) == lockedval){}
20      if(balance > amt){
21          cout<<"approved"<<endl;
22          balance-=amount;
23      }
24      word= notlockedval;
25  }
```

Compare and Swap – an atomic operation (once started cannot be interrupted)

1st thread => word=notlockedval => line 19 returns false, go to 20

Mutex – how to implement

```
1   //a special atomic function
2   int compare_and_swap(int *word, int notlockedval, int lockedval){
3       //save original
4       int oldval=*word;
5
6       if(oldval==notlockedval)    //if we can
7           *word=lockedval         //then do
8       return oldval;
9   }
10
11  const int notlockedval=0;
12  const int lockedval=1;
13
14  int word=notlockedval;  //start unlocked
15
16  void withdraw(int amt){
17
18      //stay in below loop until compare_and_swap returns notlockedval
19      while (compare_and_swap(&word,notlockedval, lockedval ) == lockedval){}
20      if(balance > amt){
21          cout<<"approved"<<endl;
22          balance-=amount;
23      }
24      word= notlockedval;
25  }
```

Compare and Swap – an atomic operation (once started cannot be interrupted)

1st thread => word=notlockedval => line 19 returns false, go to 20

1st thread => Interrupted at line 21

Mutex – how to implement

```
1   //a special atomic function
2   int compare_and_swap(int *word, int notlockedval, int lockedval){
3       //save original
4       int oldval=*word;
5
6       if(oldval==notlockedval)    //if we can
7           *word=lockedval         //then do
8       return oldval;
9   }
10
11  const int notlockedval=0;
12  const int lockedval=1;
13
14  int word=notlockedval;  //start unlocked
15
16  void withdraw(int amt){
17
18      //stay in below loop until compare_and_swap returns notlockedval
19      while (compare_and_swap(&word,notlockedval, lockedval ) == lockedval){}
20      if(balance > amt){
21          cout<<"approved"<<endl;
22          balance-=amount;
23      }
24      word= notlockedval;
25  }
```

Compare and Swap – an atomic operation (once started cannot be interrupted)

1st thread => word=notlockedval => line 19 returns false, go to 20

1st thread => Interrupted at line 21

2nd thread => line 19 returns true, so it busy waits (spins) burning CPU time

Mutex – how to implement

```
1   //a special atomic function
2   int compare_and_swap(int *word, int notlockedval, int lockedval){
3       //save original
4       int oldval=*word;
5
6       if(oldval==notlockedval)    //if we can
7           *word=lockedval         //then do
8       return oldval;
9   }
10
11  const int notlockedval=0;
12  const int lockedval=1;
13
14  int word=notlockedval;  //start unlocked
15
16  void withdraw(int amt){
17
18      //stay in below loop until compare_and_swap returns notlockedval
19      while (compare_and_swap(&word,notlockedval, lockedval ) == lockedval){}
20      if(balance > amt){
21          cout<<"approved"<<endl;
22          balance-=amount;
23      }
24      word= notlockedval;
25  }
```

Compare and Swap – an atomic operation (once started cannot be interrupted)

1st thread => word=notlockedval => line 19 returns false, go to 20

1st thread => Interrupted at line 21

2nd thread => line 19 returns true, so it busy waits (spins) burning CPU time

1st thread swapped back and finishes, line 24 set word=notlockedval

Mutex – how to implement

```
1   //a special atomic function
2   int compare_and_swap(int *word, int notlockedval, int lockedval){
3       //save original
4       int oldval=*word;
5
6       if(oldval==notlockedval)    //if we can
7           *word=lockedval         //then do
8       return oldval;
9   }
10
11  const int notlockedval=0;
12  const int lockedval=1;
13
14  int word=notlockedval;  //start unlocked
15
16  void withdraw(int amt){
17
18      //stay in below loop until compare_and_swap returns notlockedval
19      while (compare_and_swap(&word,notlockedval, lockedval ) == lockedval){}
20      if(balance > amt){
21          cout<<"approved"<<endl;
22          balance-=amount;
23      }
24      word= notlockedval;
25  }
```

Compare and Swap – an atomic operation (once started cannot be interrupted)

1st thread => word=notlockedval => line 19 returns false, go to 20

1st thread => Interrupted at line 21

2nd thread => line 19 returns true, so it busy waits (spins) burning CPU time

1st thread swapped back and finishes, line 24 set word=notlockedval

2nd thread swapped back, line 19 returns false, goes to 20 and finishes

Mutex – how to implement

```
1   //a special atomic function
2   int compare_and_swap(int *word, int notlockedval, int lockedval){
3       //save original
4       int oldval=*word;
5
6       if(oldval==notlockedval)    //if we can
7           *word=lockedval         //then do
8       return oldval;
9   }
10
11  const int notlockedval=0;
12  const int lockedval=1;
13
14  int word=notlockedval;  //start unlocked
15
16  void withdraw(int amt){
17
18      //stay in below loop until compare_and_swap returns notlockedval
19      while (compare_and_swap(&word,notlockedval, lockedval ) == lockedval){}
20      if(balance > amt){
21          cout<<"approved"<<endl;
22          balance-=amount;
23      }
24      word= notlockedval;
25  }
```

Compare and Swap

<u>Good:</u>

Simple

Easily verified

Multiprocessor/multiprocess as long as can share memory

Mutex – how to implement

```
1   //a special atomic function
2   int compare_and_swap(int *word, int notlockedval, int lockedval){
3       //save original
4       int oldval=*word;
5
6       if(oldval==notlockedval)    //if we can
7           *word=lockedval         //then do
8       return oldval;
9   }
10
11  const int notlockedval=0;
12  const int lockedval=1;
13
14  int word=notlockedval;  //start unlocked
15
16  void withdraw(int amt){
17
18      //stay in below loop until compare_and_swap returns notlockedval
19      while (compare_and_swap(&word,notlockedval, lockedval ) == lockedval){}
20      if(balance > amt){
21          cout<<"approved"<<endl;
22          balance-=amount;
23      }
24      word= notlockedval;
25  }
```

Compare and Swap

<u>Good:</u>

Simple

Easily verified

Multiprocessor/multiprocess as long as can share memory

<u>Bad</u>

Busy wait (line 19) must keep checking until available, CPU usage spikes, see Spinlock project)

Starvation and Deadlock both possible

Compare_and_swap has to be atomic, this C++ code is not

Mutex – Using implementation in C++ 11

- Mutexes are thread based in C++ 11, not process based!

```
#include <mutex>
std::mutex g_mutex;          //generally a global value
:
g_mutex.lock();              //if available then proceed, otherwise thread blocks
g_mutex.unlock();            //unlocks the mutex, other waiting threads can acquire
```

General rules

Unlock a mutex when you are done (else waiting threads will wait forever)

Do not lock() a mutex twice from same thread without intermediate unlock(). Otherwise thread will block waiting to acquire a mutex that it has.

Mutex – Solve withdrawal problem

```
std::mutex mymutex;
void withdraw(int amt){

    mymutex.lock();
    if(balance > amt){
        cout<<"approved"<<endl;
        balance-=amount;
    }
    mymutex.unlock();
}
```

Mutex – Solve withdrawal problem

```
std::mutex mymutex;
void withdraw(int amt){

    mymutex.lock();
    if(balance > amt){
        cout<<"approved"<<endl;
        balance-=amount;
    }
    mymutex.unlock();
}
```

Mutex – Solve withdrawal problem

```
std::mutex mymutex;
void withdraw(int amt){

    mymutex.lock();
    if(balance > amt){
        cout<<"approved"<<endl;
        balance-=amount;
    }
    mymutex.unlock();
}
```

But wait! What if you throw an exception here?

Mutex – Solve withdrawal problem

```
std::mutex mymutex;
void withdraw(int amt){

    mymutex.lock();
    if(balance > amt){
        cout<<"approved"<<endl;
        balance-=amount;
    }
    mymutex.unlock();
}
```

But wait! What if you throw an exception here?

- You will never unlock the mutex.
- All threads waiting to enter the critical section will be blocked forever
- Process will never join() those threads
- Process will be blocked forever
- Have to kill and restart process

Mutex – A better idea

- Use a self unlocking mutex- As soon as the mutex goes out of scope it unlocks.

```
std::mutex mymutex;
void withdraw(int amt){

    lock_guard<std::mutex> lock(mymutex); //locks mymutex here
    if(balance > amt){
        cout<<"approved"<<endl;
        balance-=amount;
    }
    //unlocks mymutex here when the lock_guard goes out of scope
}

lock_guard<std::mutex> lock(mymutex);
```

Mutex – A better idea

- Use a self unlocking mutex- As soon as the mutex goes out of scope it unlocks.

```
std::mutex mymutex;
void withdraw(int amt){

    lock_guard<std::mutex> lock(mymutex); //locks mymutex here
    if(balance > amt){
        cout<<"approved"<<endl;
        balance-=amount;
    }
    //unlocks mymutex here when the lock_guard goes out of scope
}

lock_guard<std::mutex> lock(mymutex);
```

Mutex – A better idea

- Use a self unlocking mutex- As soon as the mutex goes out of scope it unlocks.

```
std::mutex mymutex;
void withdraw(int amt){

    lock_guard<std::mutex> lock(mymutex); //locks mymutex here
    if(balance > amt){
        cout<<"approved"<<endl;
        balance-=amount;
    }
    //unlocks mymutex here when the lock_guard goes out of scope
}

lock_guard<std::mutex> lock(mymutex);
```

But wait! What if you throw an exception here?

Mutex – A better idea

- Use a self unlocking mutex- As soon as the mutex goes out of scope it unlocks.

```
std::mutex mymutex;
void withdraw(int amt){

    lock_guard<std::mutex> lock(mymutex); //locks mymutex here
    if(balance > amt){
        cout<<"approved"<<endl;
        balance-=amount;
    }
    //unlocks mymutex here when the lock_guard goes out of scope
}

lock_guard<std::mutex> lock(mymutex);
```

But wait! What if you throw an exception here?

- No worries, the lock_guard will unlock as soon as it goes out of scope

Summary

- Mutexes function as a traffic cop, they allow 1 thread in a critical section at a time, other threads are blocked.
- Prefer a lock_guard over a raw mutex since it automatically unlocks when it goes out of scope
- To use: Identify minimal critical sections, then wrap critical section with an auto unlocking lock_guard
- Mutexes are going to be global variables, they will NOT be local variables