Mutual Exclusion

Enforcement
only 1 thread in critical section of shared object.

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, then every access to that resource is protected
(cannot have 1 access that is not protected)

Hardware enforced.

what if we disable interrupts?
- guarantees atomic code

Bad
- cannot have overlapping critical sections
- disables switching to other non-related (does not use shared resource) processes
- will not work multiprocessor (can't disable interrupts on 2 cores at a time.
- kills performance on single core

Compare & Swap

```
1  int cas( int *word, int notlockedval, int lockedval)
2  {
3      int oldval = *word;
4      if (oldval == notlockedval)      // if we can
           *word = lockedval;           // then do
5      return oldval;
   }
```

atomic op, no int possible here

lets say notlockedVal = 0
lockedVal = 1
} if *word == 0 we can lock it otherwise not.

```
const int NLV = 0;
const int LV = 1;
int word = NLV; //start off unlocked
void withdraw (int amount) {
```

Stay in loop until cas() says NLV

```
1      while( cas( &word, NLV, LV) == LV){}
2      if (balance > amount){
3          cout << "Approved";
4          balance -= amount;
5      }
6      word = NLV;
   }
```

① 1st thrd → word = NLV; line 1 cond from false goto 2

② interrupt at 3, 2nd thrd calls 1

③ cas returns LV == LV so it busy waits (burning CPU time)

④ 1st thrd swapped back in & finishes, line 6 word = NLV

⑤ 2nd thrd line 1, cas returns NLV != LV so it proceeds.

good

- simple
- easily verified
- multiprocessor, multiprocess as long as can share memory
- can have lots of CAS fines, each with own mem.

bad

- busy wait (must keep checking until available)
  {very bad, watch CPU usage spike}
- starvation & deadlock both possible

yeah, this is always the case

? can I run this program as written & use the CAS to offer effective mutual exclusion?

NO → CAS must be atomic & my CAS is not

~~this function is part of the part of the HW~~

It has to be part of hardware & atomic

ie (no interrupts)

First

- talk about mutexes, C++11 construct for synchronization between threads. (1 at a time
  - with examples of how to solve withdrawel problem.
- then condition vars
- then semaphores, sorped up mutex, can allow more than 1 at a time
  - example
- then how semaphores implemented

ex (thread based not process based) need shared memory. in linux, threads are treated as processes with the same mem space.

```
#include <mutex>

std::mutex g_mutex;
void lock();      // if avail will proceed, otherwise locks
void unlock();    // unlocks (once per call);
bool trylock();   // locks if possible or returns false
                  // no blocking
```

- do not call lock multiple times from same thread! trylock [if you must use recursive_mutex]
- unlock mutex when you are done!!

---

solve withdrawel prob

```
mutex g_mutex;
void withdraw(int amount){
    g_mutex.lock();
    if (balance > amount){
        cout << "approved";
        balance -= amount;
    }
    g_mutex.unlock();
}
```

what happens if you throw an exception ∴ never unlock ¿L? (Deadlock() for all other threads) Kill ∴ restart.

---

better solution

how about a class?

.h

```
class lock_guard
{ private:
    mutex*g_mutex;
  public:
    lock_guard(mutex &a_mutex);
  ~lockguard();
```

.cpp

```
lockguard::lock_guard(mutex &a_mutex){
g_mutex=&a_mutex; a_mutex.lock();
}
lock_guard::~lock_guard(){
    (*a_mutex).unlock();
}
```

(auto unlocks when it goes out of scope!!)

ready in C++11 !

```
#include <mutex>
  mutex gmutex;
  lock_guard<std::mutex> lock(gmutex);
```

↑ when this goes out of scope it unlocks!

show in withdraw problem.

show you do not have to unlock

but what if you want to lock accross functions

```
int getbalance() {
    return balance;
}
```

---

show seperate function ~~call~~ call in withdraw (so its protected) is it OK? No, accessable outside of locked code

```
addbalance(int i) {
  balance += i;
}
```

Mutex Types (partial) review

- mutex m => m.lock()
  m.unlock()
  can be recursive_lock...
- lock_guard<mutex> lck(m)
  auto unlocks when it goes out of scope

what happens if you need to lock the same mutex more than once before unlock?

- ① use of recursive_lock
  see Baptiste Page

But also consider that your design may not be optimal if you need to do this. And its harder to track locked regions

- unique_lock<mutex> lck(m)
  like a lock_guard auto unlocks when it goes out of scope, can be recursive if the lock it wraps is.
  Used with condition variables

Condition Variable

used to signal between threads. [thread(s) block until signaled]

overview

bool isReady = false; // global

T1 (waiting on isReady == true) ← blocked in meantime

need some locking

T2 (sets isReady = true & then notifies any other threads waiting.

std::condition_variable

Notifying

notify_one → OS wakes 1 thread (can't tell which one)

notify_all → OS wakes them all (1 will acquire mutex & continue) the rest wait until they get mutex & continue

Waiting

wait — wait until notified

wait_for, wait_until } timed versions of above ignore for now

```
#include <mutex>
#include <condition_variable>

mutex m;
condition_variable c;
bool ready = false;
void get() {
    unique_lock<mutex> lck(m);   <- acquire
    while (!ready)
        c.wait(lck)              <- release & wait for signal
}

void set() {
    unique_lock<mutex> lck(m);
    ready = true;                // all access to ready protected
    cv.notify_all;               // wake up all other threads.
}
```

unique_lock verses lockguard?
lockguard is locked until destruct

unique_lock is all that u can unlock, create unlocked then lock
unique_lock a little heavier, always use lock_guard unless you need unique lock abilities