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)

[illegible]

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

✓

It will [illegible] kills performance.

Compare & Swap

atomic op, no int possible here

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

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) {

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

stay in loop until cas(.) says NLV

① 1st thrd → word = NLV; line 1 condition 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 funcs, each with own mem.

bad

☹ → - busy wait (must keep checking until available) {very bad, watch CPU usage spike}

yeah, this is always the case → - starvation & deadlock both possible

? 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 semaphores, souped up mutex, can allow more than 1 at a time
  - example

- then how semaphores implemented

tex (thread based <u>not</u> process based) need shared memory. in linux, threads are treated as processes with the same mem space.

10/19/16

①

```
#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 withdrawal 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
i.e? (Deadlock() for all other threads) Kill & restart.

---

better solution

how about a class?

.h

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

.cpp

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

}
```

autounlocks when it goes out of scope!

Already in C++11!

```
#include <mutex>
mutex gmutex;
lock.guard<std::mutex> lock(gmutex);
↑
```

when this goes out of scope it unlocks!

show in withdrawal problem.

show you do not have to unlock

but what if you want to lock accross functions

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

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

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

Semaphore init, semwait, semsignal

count

① may be initialized to a nonnegative int val
(corresponding to "how many at once"

② semWait - decrements count,
if count becomes negative then process ~~executing~~ is blocked
otherwise it proceeds   [blocking is not busy wait, give up TS]

③ sem Signal - increments count
if count is <= 0 then a blocked process is unblocked

BTW no out of the box semaphore In C++ 11

Bank ex.

```
semaphore s(1);
void withdraw (int amount) {
        semwait(s);
    if (balance > amount) {
            cout << "approved";
            balance -= amount;

    }
    semsignal (s);


Thread T1 (withdraw, 10)
Thread T2 (withdraw, 10)
Thread T3 (withdraw, 10)
```

can signal & wait on diff threads

good? what if UP → semsig → gi
& down semwait() → g--

```
semaphore s = 0
s.count = 0;
int gi = 0;

UP () {
    for (int i = 0; i < 10; i++)
        gi++;
        semsignal (s)
}

down () {
    for (int i = 0; i < 10; i++)
        semwait (s);
        gi--;
}
```

Semaphore

```
struct semaphore {
        int count;                    // # processes allowed in at a time
        queueType queue;   ← remove
                             FIFO no starvation   strong semaphore
                             dif order? starvation possible
}

void semwait (semaphore s)
        s.count--;
        if (s.count < 0){
            // place this process in s.queue
            // block it          std::thread::yield
        }
}

void semSignal ( semaphore s) {
        s.count ++;
        if (s.count <= 0) {
            // remove a process p from s.queue
            // place process p on ready list
        }
}
```