

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) (remember)

Hardware enforced.

what if we disable interrupts?

- guarantees atomic code

Bad

- cannot have overlapping critical sections
- disables scheduling 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 work on single core performance.

Compare & Swap

atomic op, no race possible here

```
1 int cas ( int *word, int notlockedval, int lockedval)
2 {
3     int oldval = *word;
4     if (oldval == notlockedval) // if we can
5         *word = lockedval; // then do
6     return oldval;
7 }
```

lets say notlockedval = 0 } if *word == 0 we can
lockedval = 1 } 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;
7 }
```

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 (wasting 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.

4)

- bad

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

- starvation & deadlock both possible

NO \rightarrow 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)

mutex (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~~ must use recursive-mutex]

- unlock mutex when you are done!

solve withdrawal probs

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
it? (Deadlock) for all other threads Kill & restart.

better solution

how about a class?

.cpp

lockguard::lock_guard(mutex &amutex) {
g_mutex = &amutex; amutex.lock();

lockguard::~lock_guard() {
(&amutex).unlock();

}

(auto unlocks when it goes
out of scope)

class lock_guard

{ private:

mutex* g_mutex;

public:

lock_guard(mutex &amutex);

~lock_guard();

ready in C++!

#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
across functions

```
int get balance() {
```

```
    return balance;
```

```
}
```

show separate function ~~with~~ call in withdraw (so
its protected) is it OK? No! accessible ~~outside of lock~~ code

```
add balance (int i) {
```

```
    balance += i;
```

```
}
```


Mutex Types (partial) review

- mutex m \Rightarrow m.lock()
m.unlock() can be recursive-lock...

- lock_guard (mutex) lock(m)
auto-unlocks when it goes out of scope

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

- ① use a recursive-lock
see Baptiste Page

But also consider
that your design
may not be opti-
mal if you need to
do this. And it's
harder to track
locked regions

- unique_lock (mutex) lock(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