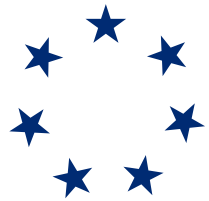
Lock Implementation

Instructor: Hengming Zou, Ph.D.

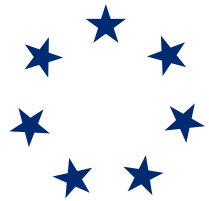
In Pursuit of Absolute Simplicity 求于至简，归于永恒

Content



- Lock Implementation
- Use interrupt enable and disable
- Use test-and-set instruction

Implementing Locks



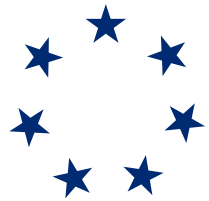
- So far we have used locks extensively
- We assumed that lock operations are atomic
- But how atomicity of lock is implemented?
- Lock must be implemented on hardware operations

Concurrent programs
High level synchronization operations provided by software (i.e. semaphore, lock, monitor)
Low-level atomic operations provide by hardware (i.e. load/store, interrupt enable/disable, test & set)

Use Interrupt Disable/Enable

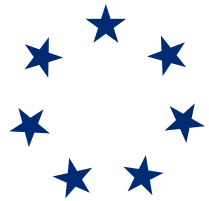
- On uniprocessor, operation is atomic as long as
  - context switch doesn’t occur in middle of operation
- How does thread get context switched out?
  - interrupt
- Prevent context switches at wrong time by preventing these events

Use Interrupt Disable/Enable



- With interrupt disable/enable to ensure atomicity,
  - why do we need locks?
- User program could call interrupt disable
  - before entering critical section
  - and call interrupt enable after leaving critical section
  - and make sure not to call yield in the critical section
- Could this work?
  - Theoretically, yes; practically, No.
- Therefore, it is better to leave the matter to the OS

Lock Implementation #1



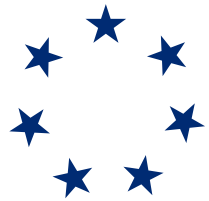
- Disable interrupts with busy waiting
- lock() {
- disable interrupts
- while (value != FREE) {
- enable interrupts
- disable interrupts
- }
- value = BUSY
- enable interrupts
- }

Why does lock() disable interrupts in the beginning of the function?

Why is it OK to disable interrupts in lock()'s critical section

Why wasn't it OK to disable interrupts while user code was running?

Lock Implementation #1

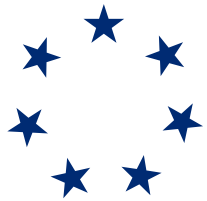


- unlock() {
- disable interrupts
- value = FREE
- enable interrupts
- }

Do we need to disable interrupts in unlock()?

Remember the x:=1 and x:=2 problem?

Read-Modify-Write Instructions



- Another atomic primitive
- Modern processors provide an easier way
  - with atomic read modify-write instructions
- Read-modify-write atomically
  - reads value from memory into a register
  - Then writes new value to that memory location

Read-Modify-Write Instructions

- test_and_set
  - atomically writes 1 to a memory location (set)
  - and returns the value that used to be there (test)

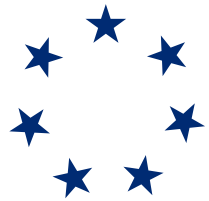
- test_and_set(*X*) {
- tmp = *X*
- *X* = 1
- return(tmp)
- }

Lock Implementation #2

- Test & set with busy waiting
  - (value is initially 0)
  - lock() {
  - while (test_and_set(value) == 1) {}
  - }

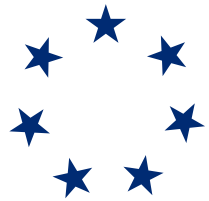
  - unlock() {
  - value = 0
  - }

Lock Implementation #2



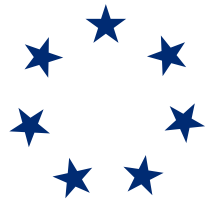
- If lock is free (value = 0)
  - test_and_set sets value to 1 and returns 0,
  - so the while loop finishes
- If lock is busy (value = 1)
  - test_and_set doesn’t change the value and returns 1,
  - so loop continues

Assembly Implementation



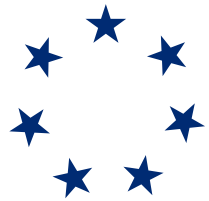
```
mutex_lock:
        TSL REGISTER, MUTEX     |copy mutex to register, set mutex to 1
        CMP REGISTER, #0        | was mutex zero?
        JNE ok                  |if zero, mutex was unlocked, so return
        CALL thread_yield       |mutex busy, schedule another thread
        JMP mutex_lock          |try again later
ok:     RET                     |return to caller; CR entered

mutex_unlock:
        MOVE MUTEX, #0          |store a 0 in mutex
        RET                     |return to caller
```

Strategy for Reducing Busy-Waiting



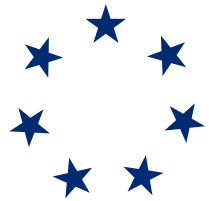
- In method 1 & 2, waiting thread uses lots of CPU time
  - just checking for lock to become free
- Better for it to sleep and let other threads run

Lock Implementation #3



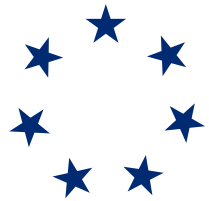
- Interrupt disable, no busy-waiting
- Waiting thread gives up processor
  - so that other threads (e.g. thread with lock) can run more quickly
- Someone wakes up thread when the lock is free

Lock Implementation #3



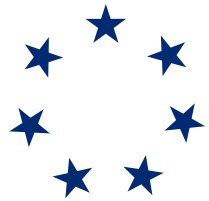
- lock() {
- disable interrupts
- if (value == FREE) {
- value = BUSY
- } else {
- add thread to queue of threads waiting for this lock
- switch to next run-able thread
- }
- enable interrupts
- }

Lock Implementation #3



- unlock() {
- disable interrupts
- value = FREE
- if (any thread is waiting for this lock) {
- move waiting thread from waiting queue to ready queue
- value = BUSY
- }
- enable interrupts
- }

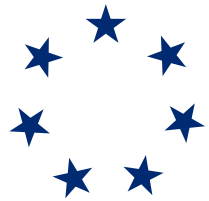
Issue Lock Implementation #3



- When should lock() re-enable interrupts before calling switch?
- lock() {
- disable interrupts
- if (value == FREE) {
- value = BUSY
- } else {
- add thread to queue of threads waiting for this lock ←
- switch to next run-able thread ← Only three places
- }
- enable interrupts
- }

Is this code correct?

Interrupt Disable/Enable Pattern



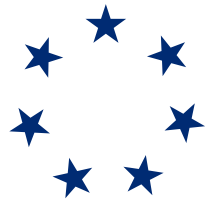
- Enable interrupts before adding thread to wait queue?
- lock() {
  - disable interrupts
  - ...
  - if (lock is busy) {
    - enable interrupts
    - add thread to lock wait queue
    - switch to next run-able thread
  - }

Remember the signal loss in PC?

When could this fail?

Will this work?

Interrupt Disable/Enable Pattern

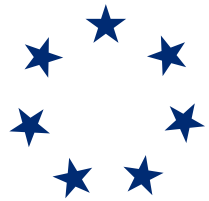


- Enable interrupts after adding thread to wait queue,
  - but before switching to next thread?
- lock() {
- disable interrupts
- ...
- if (lock is busy) {
- add thread to lock wait queue
- enable interrupts
- switch to next run-able thread
- }
- }

When could this fail?

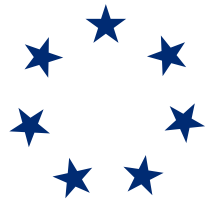
Will this work?

Interrupt Disable/Enable Pattern



- But this fails if interrupt happens after thread enable interrupts
  - lock() adds thread to wait queue
  - lock() enables interrupts, interrupt causes preemption,
- Preemption moves thread to ready queue
  - Now thread is on two queues (wait and ready)!
- Also, switch is likely to be a critical section
  - Add thread to wait queue and switch must be atomic

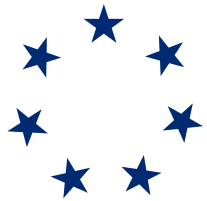
Solution



- Waiting thread leaves interrupts disabled
  - when it calls switch
- Next thread to run has the responsibility of
  - re-enabling interrupts before returning to user code
- When waiting thread wakes up
  - it returns from switch with interrupts disabled

- Caveat:
  - All threads promise to have interrupts disabled when they call switch
  - All threads promise to re-enable interrupts after returned to from switch

Solution



Thread A

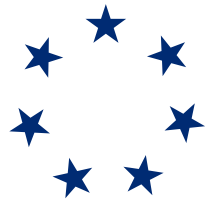
enable interrupts
}
<user code runs>
lock() {
disable interrupts
...
switch

back from switch
enable interrupts
}

Thread B
yield() {
disable interrupts
switch

back from switch
enable interrupts
}
<user code runs>
unlock() (move thread A to ready queue)
yield() {
disable interrupts
switch

Lock Implementation #4



- Test and set, minimal busy-waiting
- Can’t implement locks using test & set without some busy-waiting
  - but can minimize it
- Idea:
  - use busy waiting only to atomically execute lock code
  - Give up CPU if busy
- Solution:
  - Use an extra variable: guard

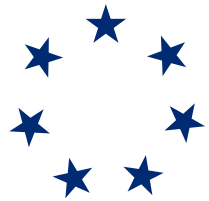
Lock Implementation #4

```
lock() {
    while(test_and_set(guard)) {
    }
    if (value == FREE) {
        value = BUSY
    } else {
        add thread to queue of threads waiting for this lock
        switch to next run-able thread
    }
    guard = 0
}
```

Lock Implementation #4

- unlock() {
- while (test_and_set(guard)) {
- }
- value = FREE
- if (any thread is waiting for this lock) {
- move waiting thread from waiting queue to ready queue
- value = BUSY
- }
- guard = 0
- }

Problems with Interrupt Approach



- Interrupt disable works on a uniprocessor
  - by preventing current thread from being switched out
- But this doesn't work on a multi-processor
- Disabling interrupts on one processor doesn't
  - prevent other processors from running
- Not acceptable (or provided) to modify interrupt disable
  - to stop other processors from running
- How about test & set on multi-processor system?


Thoughts Change Reality
意念改变现实