Problem: Atomic Read-Modify Write Instructions

* Kernel-Level lock 🡺 Requires syscall() hence going to kernel (which is slow operation)
* Can’t give lock implementation to users
* Doesn’t work on multiprocessors
  + Disabling interrupts only works for timer for a given processor (no critical section for other processors)

Alternative: Atomic Instruction sequences

* In memory system 🡺 doesn’t go to kernel
* These instruction read a value and write a new value atomically
* Hardware is responsible for implementing this correctly
  + On both uniprocessors and multiprocessors
  + Unlike disabling interrupts 🡺 Can be used on both uniprocessors and multiprocessors
* Examples of Read-Modify-Write
  + **Everything in these functions meant to be an atomic operation in processor**
  + test&set(&address) 🡺 only first thread sets value at memory to 1, other threads assign it to 1 when it is already 1
  + swap(&address, register) 🡺 swap the values of memory address and register
  + compare&swap(&address, reg1, reg2) 🡺 If memory address == reg1, then put reg2 in memory instead… otherwise don’t change memory
  + load-linked&store-condition(&address) 🡺
* Atomic add to linked-list function
  + I DON’T GET IT
* Implementing locks with test&setå

int mylock = 0

acquire(\*thelock){

while(test&set(thelock));

}

release(\*thelock){

\*thelock = 0;

A white background with black text

Description automatically generated with low confidence}

This is busy waiting! Thread consumes cycles while waiting.

BUT doesn’t go into kernel, only memory reads AND works on multiprocessor

Positives:

* User code can use the lock
* Works on a multiprocessor
* Machine can receive interrupts

Negatives:

* This is very inefficient as thread will consume cycles waiting
* Waiting thread may take cycles away from thread holding lock
* Priority inversion 🡺 If busy-waiting thread has higher priority than thread holding lock 🡺 no progress

Better locks using test&set

Can we build test&set locks without busy waiting?

* Mostly 🡺 Idea: only busy-wait to atomically check lock value

A screenshot of a computer code

Description automatically generated with low confidence

No kernel interrupt, syscall() to sleep still exists

Basically we replaced:

Enable interrupt 🡺 while(test&set(guard))

Disable interrupt 🡺 guard = 0

Busy waiting is extremely short (a single test and set call) 🡺 all code is in user mode though

Prior short solution with a single test&set call on page 1 can hang up entire system due to wasted CPU cycles from busy waiting

A screenshot of a computer code

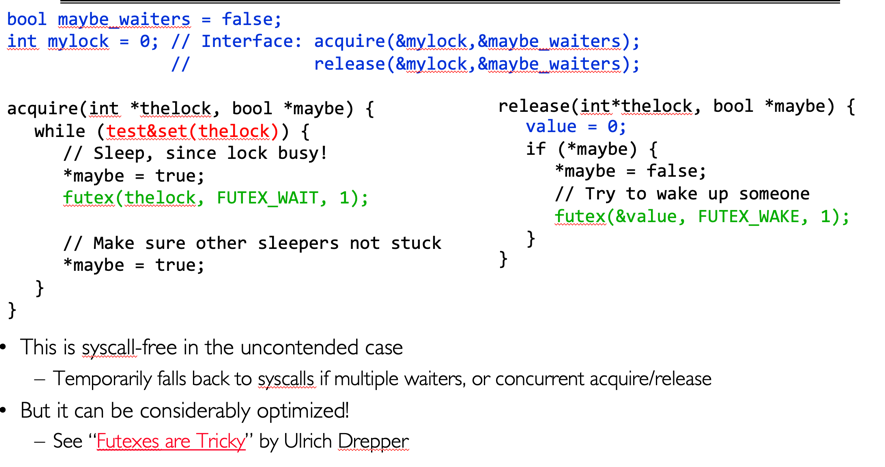
Description automatically generated with low confidence

Solution still requires putting threads to sleep

FUTEX: Fast Userspace Mutex 🡺

A picture containing text, screenshot, font

Description automatically generated\*More low level implementations of locks w/futexes\*



A screenshot of a computer code

Description automatically generated with low confidence

Problem is that semaphore are dual-purpose:

* They are used both for mutexes and scheduling constraints
* Note flipping the P’s in bounded buffer problem gives deadlock which is not obvious

Cleaner idea: Use locks for mutual exclusion and condition variables for scheduling constraints

Monitor 🡺 a lock and zero or more condition variables for managing concurrent access to shared data

* Most other use actual locks and condition variables
* Monitor is a paradigm for concurrent programming
* Complicated synchronization done easily

Condition Variables: a queue of threads waiting for something inside a critical section

* Key idea: allowing sleeping inside critical section by atomically releasing lock at time we go to sleep
* Contrast to semaphore: Can’t wait inside critical section
* Operations:
  + Wait(&lock): Atomically release lock and go to sleep
  + Signal(): Wake up one waiter, if any
  + Broadcast(): Wake up all waiters

Monitor with Condition Variables

* Lock 🡺 Lock provides mutual exclusion to shared data
  + Always acquire before accessing shared data
  + Always release after finishing with shared data
  + Lock initially free
* Condition variable 🡺 a queue of threads waiting for something inside a critical section
  + Key idea: make it possible to go to sleep inside critical section by atomically releasing lock at a time we go to sleep

Example (infinite buffer):

* Consumer still goes to sleep if queue is empty

A screenshot of a computer code

Description automatically generated with low confidence

cond\_wait takes both buf\_CV and buf\_lock

cond\_wait needs to be inside critical section after lock acquired so that other threads don’t modify state and hence make condition false

while loop is to recheck queue in case someone signals thread

A picture containing text, font, screenshot

Description automatically generated

Need to be precise about Hoare vs Mesa monitor

(Between being woken up and OS acquiring lock for us, some other thread and grab item on queue 🡺 always have to check)

A picture containing text, screenshot, font, line

Description automatically generated

Messy from implementation perspective

Bad from cache standpoint 🡺 signaler thread loses CPU and hence cache state

A picture containing text, screenshot, font, line

Description automatically generatedA picture containing text, screenshot, font

Description automatically generated

A screenshot of a computer program

Description automatically generated with medium confidence

* Condition variables and checking is in lock 🡺 Don’t have to worry about state changing while checking
* When thread is waiting, it sleeps 🡺 not busy waiting

For most OS, when a thread is woken up signal(), it is simply placed on ready queue

It may or may not acquire the lock immediately

* Another thread could be scheduled and “sneak” into empty queue
* Need a loop to recheck condition on wakeup

Readers/Writers Problem

Motivation: Consider a shared databse

* Two class of users:
  + Readers – never modify database
  + Writers – read and modify database
* Is a single lock on the whole database sufficient?
  + Likely to have multiple readers, one writer
* Correctness Constraint
  + Readers can access DB when no writers
  + Writers can access DB when no readers or writers
  + Only one thread manipulates state variables

Reader()

* Wait until no writers
* Access DB
* Check out – wake up a waiting writer

Writer()

* Wait until no readers of writers
* Access DB
* Check out – wake up waiter readers or writers

State Variables

* int AR 🡺 active reader
* int WR 🡺 waiting reader
* int AW 🡺 active writer
* int WW 🡺 waiting writer
* Condition okToRead = NIL
* Condition okToWrite = NIL

Reader(){

//CHECK INTO SYSTEM

acquire(&lock)

while (AW + WW > 0){

WR++;

cond\_wait(&okToRead, &lock);

WR--;

}

AR++;

release(&lock);

//ACCESS DB

acquire(&lock)

AR--;

if(AR == 0 && WW > 0)

cond\_signal(&okToWrite)

release(&lock)

}

Writer(){

acquire(&lock)

while (AW + AR > 0){

WW++;

cond\_wait(&okToRead, &lock);

WW--;

}

AW++;

release(&lock)

if(WW > 0){

cond\_signal(&okToWrite); //wake up one writer

} else if (WR > 0){

cond\_broadcast(&okToRead); //wake up all readers

}

release(&lock);

}