X86 has hardware context switch support

* Will switch to kernel stack from user stack from syscall/interrupt for you (going from blue to red has hardware support, done automatically)

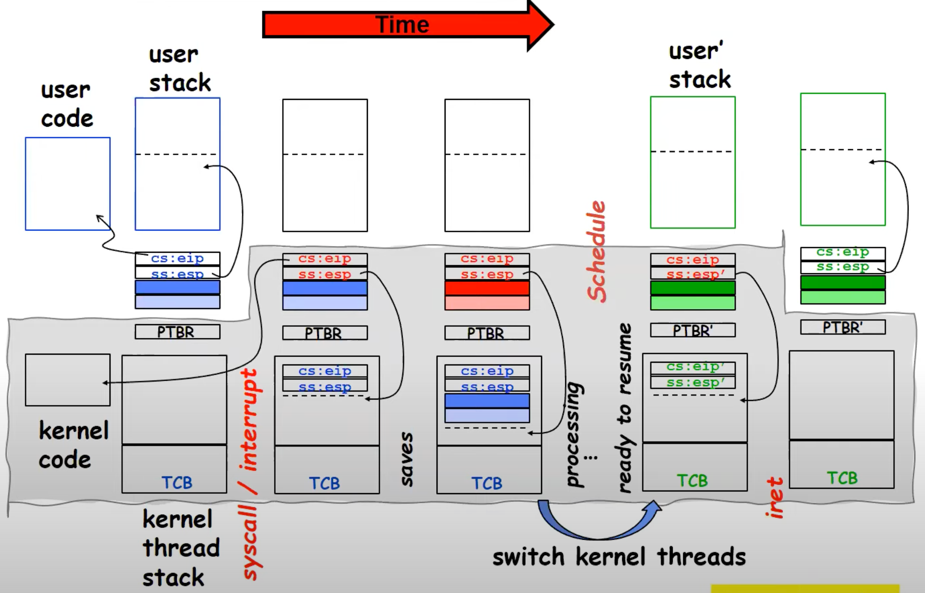
Recall:

* Kernel stack associated with user stack.
* Returning from switch and changing stacks executes a different thread
* Kernel stack, user stack, registers are restorable snapshot of thread 🡺 swap in virtual program counter (set hardware counter to this value)

A picture containing text, diagram, screenshot, plan

Description automatically generated

1. syscall/interrupt changes IP and SP to point to kernel code and kernel stack
2. Registers from user code pushed into kernel stack (by kernel)
3. Kernel loads its own registers, performs computation/processing
4. Restore user code registers… pop off kernel stack
5. Interrupt return syscall switches IP and SP to point to user code and user stack



🡸 Scheduling

(New PTBR and user code registers loaded in🡺 created on the fly by interrupt handler 🡺new IP and SP values set)

Dispatcher Problem: Infinite loop due to yield

* Timer Interrupt to Return Control
* Solution to dispatcher problem
  + User timer interrupt to force scheduling decisions

TimerInterrupt(){

DoPeriodicHouseKeeping();

run\_new\_thread();

}

* Interrupt controller disables everything as soon as interrupt taken 🡺 kernel enters interrupt routine 🡺 disables timer interrupts

A picture containing text, screenshot, font, line

Description automatically generated

Recall: Fixing banking problems with locks

* When one thread acquires lock, all other threads trying to acquire lock are put to sleep

Producer-Consumer with a Bounded Buffer

* Problem Defn:
  + Producer put things into shared buffer (finite buffer)
  + Consumer takes them out
  + Need synchronization to coordinate producer/consumer
  + Need synchronization to keep buffer consistent
* Don’t want producer and consumer to have to work in lockstep, so put a fixed-size buffer between them
  + Need to synchronize access to this buffer (don’t want buffer to get screwed up)
  + Producer needs to wait if buffer is full
  + Consumer needs to wait if buffer is empty
* Circular Buffer Data Structure

typdef struct buf{

int write\_index;

int read\_index;

<type> \*entries[BUFSIZE]

} buf\_t;

Insert: write and bump write ptr (enqueue)

Remove: read and bump read ptr (dequeue)

How to tell if Full (on insert) or Full (on remove)?

What needs to be atomic?

Mutex buf\_lock;

Producer(item){

Acquire(&buf\_lock)

while(buffer full){}

enqueue(item);

release(&buf\_lock)

}

Consumer(){

Acquire(&buf\_lock)

while(buffer empty){}

item = dequeue();

release(&buf\_lock)

return item

}

Will we ever come out of the wait loop?

Producer acquires lock 🡺 Buffer is full so it waits 🡺 Consumer can’t dequeue because Producer has lock 🡺 Buffer stays full 🡺 Producer spins forever

Mutex buf\_lock;

Producer(item){

Acquire(&buf\_lock)

while(buffer full){ release(&buf\_lock); acquire(&buf\_lock);}

enqueue(item);

release(&buf\_lock)

}

Consumer(){

Acquire(&buf\_lock)

while(buffer empty){ release(&buf\_lock); acquire(&buf\_lock }

item = dequeue();

release(&buf\_lock)

return item

}

No more deadlock.. but its bad. If producer arrives with full buffer and no consumers 🡺 Producer spins for long time 🡺 Wasted cycles

Higher Primitives than Locks

* What is the right abstraction for synchronizing threads that share memory?
  + Want as high level primitive as possible
* Good primitives and practices are important!
  + Since execution isn’t entirely sequential, really hard to find bugs since they happen rarely
* Synchronization is way of coordinating multiple concurrent activities that are shared state

Recall: Semaphores

* Semaphores are a kind of generalized lock
* Defn: Semaphore has a non-negative integer value and supports the following two operations
  + Down or P(): an atomic operation that waits for semaphore to become positive and then decrements it by 1
    - Think of this as a wait() operations
  + Up or V(): an atomic operation that increments the semaphore by 1, waking up a waiting P if any
    - Think of this as the signal() operation
* Semaphores are like integers except…
  + No negative values
  + Only operations are allowed are P and V 🡺 Can’t read or write value
  + Operations must be atomic
    - Two P’s together can’t decrement value below 0
    - Thread going to sleep in P won’t miss wakeup from V – even if both happen at same time
  + Two uses:
    - Mutual exclusion: Binary Semaphore (initial value = 1)
      * Binary Semaphore or Mutex
      * Can be used for mutual exclusion

SemaP(&mysem);

//Critical Section

SemaV(&mysem);

* + - Schedule constraints (initial value = 0)
      * Allow thread 1 to wait for a signal from thread 2
        + Thread 2 schedules thread 1

Init val semaphore = 0

ThreadJoin{

semaP(&mysem);

}

ThreadFinish{

semaV(&mysem);

}

Revisted Bounded Buffer: Correctness constrains for solution

* Correctness Constraints:
  + Consumer must wait for producer to fill buffers, if non full
  + Producer must wait for consumer to empty buffers, if all full
  + Only on thread can manipulat buffer queue at a time
* Remember why need mutual exclusion
  + Computers are stupid
* General rule of thumb: Use separate semaphore for each constraint
  + Semaphore fullSlots = 0
  + Semaphore emptySlots = bufSize
  + Semaphore mutex = 1

Producer(item) {  
 semaP(&emptySlots); // Wait until space  
 semaP(&mutex); // Wait until machine free  
 Enqueue(item);  
 semaV(&mutex);  
 semaV(&fullSlots); // Tell consumers there is  
 // more coke  
}

Consumer() {  
 semaP(&fullSlots); // Check if there’s a coke  
 semaP(&mutex); // Wait until machine free  
 item = Dequeue();  
 semaV(&mutex);  
 semaV(&emptySlots); // tell producer need more  
 return item;  
}

Order of P’s is important!

semaP(&mutex); // Wait until machine free

semaP(&emptySlots); // Wait until space

Will cause a deadlock!

Where are we going with synchronization?

* A list of milk

  Description automatically generated with low confidenceWe are going to implement various higher-level synchronization primitives using atomic operations
  + Everything is pretty painful if only atomic primitives are load and store
  + Need to provide primitives at user-level
* Motivating example: “Too Much Milk”

Can fix the milk problem by putting a key on the refrigerator

* Lock it and take key if you are going go buy milk
* Fixes too much: roommate angry if only wants OJ
* Of course 🡺 We don’t know how to make a lock yet

Correctness Proprties:

* Need to be careful about concurrent programs
  + Write down correctness behavior first 🡺 Then code
* Correctness properties for “too much milk”
  + Never more than one person buys
  + Someone buys if needed
* First attempt: Restrict ourselves to use only atomic load and store operations as building block

Too Much Milk: Solution #1

* Use not to avoid buying too much milk
  + Leave Note before Buying (lock)
  + Remove Note after Buying (unlock)
  + Don’t buy if note (wait)
* Suppose a computer tries this:
  + if(noMilk){

if(noNote){

leaveNote

buyMilk

remove note;

}

}

A picture containing text, screenshot, font

Description automatically generatedResult?

* Still too much milk
* Thread can get context switched after checking milk but before buying milk

Possible Solution:

A computer code on a white background

Description automatically generated with low confidence

Solution #3 (labeled notes):

A picture containing text, receipt, font, screenshot

Description automatically generated

Correct Solution:

A picture containing text, receipt, font, white

Description automatically generated

A picture containing text, font, white, screenshot

Description automatically generated

Solution #3 works but…

* Complicated, asymmetrical code (A code different from B) 🡺 What if many threads?
* While A is waiting, it is consuming CPU time
* Must be better way!
  + Have hardware provide higher-level primitives than atomic load & store
  + Build even higher-level programming abstractions on this hardware support

Recall our target lock interface:

* acquire(&milklock) – wait until lock is free, then grab
* release(&milklock) – Unlock, waking up anyone waiting
* These must be atomic operations – if two threads are waiting for the lock and both see it’s free, only one succeeds to grab the lock
* Then, our milk problem is easy,

acquire(&milklock);

if (nomilk)

buy milk;

release(&milklock);

Remember, sleep is different from waiting! Sleep means thread is placed on sleep queue and doesn’t consume CPU time.

How to implement locks?

* Hardware lock instruction
  + Is this a good idea?
  + What about putting task to sleep?
    - Putting thread to sleep requires knowledge of current OS, how threads look on stack, where to place thread
    - OS specific version of sleep required
  + How can we build multi-instruction atomic operations?
  + Naïve use of Interrupt Enable/Disable
    - Recall: dispatcher gets control two ways 🡺 Internal, External Events
    - On a uniprocessor can avoid context-switching by:
      * Avoiding internal events
      * Preventing external events by disabling interrupts
    - Naïve Implementation:

LockAcquire { disableInts}

LockRelease { enableInts}

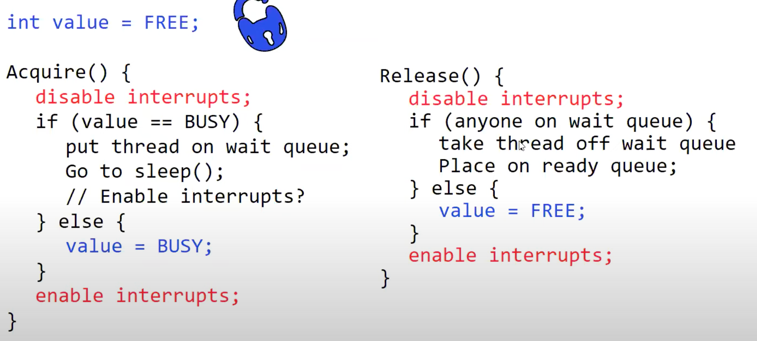
* + - Problems with this approach:
      * Can’t let user do this (can crash machine):
        + LockAcquire();

While(TRUE) {;}

* + - Real-time system
      * No guarantees on time 🡺 Critical sections might be arbitrarily long
    - What happens with I/O or other important events
    - Only one lock allowed 🡺 THE LOCK that disables interrupts

Can still use disable interrupts… (no longer THE LOCK but can be used IN A LOCK)

Key idea: maintain a lock variable and impose mutual exclusion only during operations on that variable



Only disable interrupts for a short time

* Acquire() waits just long enough to see what state of lock is (and possibly alter state of lock)
* Release() waits just long enough to see if anyone is on wait queue and remove them off queue (otherwise set value to FREE and enable interrupts)
* Using interrupt to implement lock
* Short critical section

Why do we need to disable interrupts at all?

* Avoid interruption between checking and setting lock value
* Otherwise two threads could think they both have lock
* What about re-enabling ints when going to sleep?
  + Before putting thread on wait queue?
  + After putting the thread on the wait queue?