Lecture 6: Synchronization

Instructor: Mitch Neilsen

Office: N219D

Outline

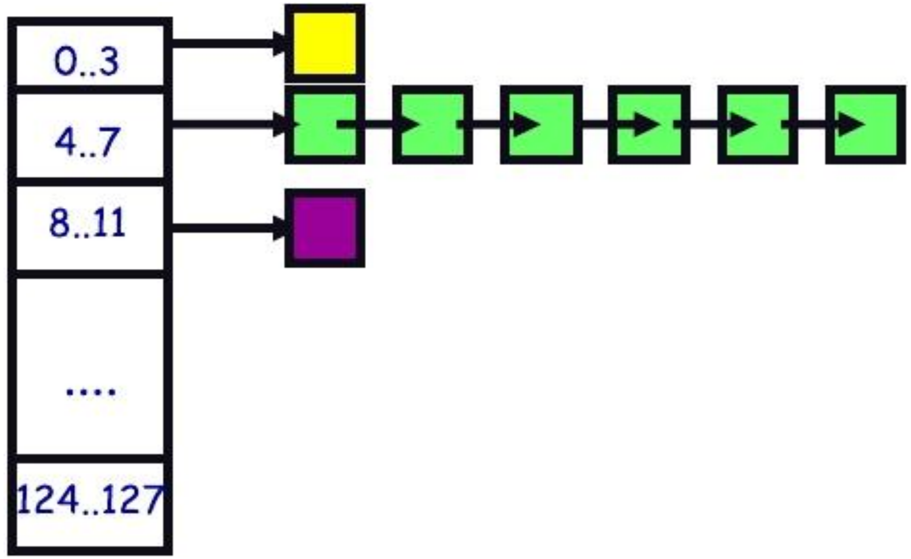
- Reading:
  - Ch. 4 - Threads
  - Ch. 5 - CPU Scheduling
  - Ch. 6 - Synchronization
- Project 1: Scheduling and Synchronization
  - Alarm Clock
  - Priority-based Scheduler
  - Synchronization and Priority Inheritance
  - [Extra Credit] MLFQ Scheduler

Quote of the Day

"Sometimes you are in sync with the times, sometimes you are in advance, sometimes you are late. "

-- Bernardo Bertolucci

Multilevel feeedback queues (BSD)



- **Every runnable process on one of 32 run queues**
  - Kernel runs process on highest-priority non-empty queue
  - Round-robins among processes on same queue
- **Process priorities dynamically computed**
  - Processes moved between queues to reflect priority changes
  - If a process gets higher priority than running process, run it
- **Idea: Favor interactive jobs that use less CPU**

Process priority (BSD model)

- p_nice **– user-settable weighting factor**
- p_estcpu **– per-process estimated CPU usage**
  - Incremented whenever timer interrupt found proc. running
  - Decayed every second when process runnable

$$\text{p\_estcpu} \leftarrow \left(\frac{2 \cdot \text{load}}{2 \cdot \text{load} + 1} \right) \text{p\_estcpu}$$

  - Load is sampled average of length of run queue plus short-term sleep queue over last minute

- **Set process priority by** **(lower p_usrpri = higher priority)**

$$\text{p\_usrpri} \leftarrow 50 + \left(\frac{\text{p\_estcpu}}{4} \right) + 2 \cdot \text{p\_nice}$$

(value clipped if over 127)

Sleeping process increases priority

- p_estcpu **not updated while asleep**
  - Instead p_slptime keeps count of sleep time
- **When process becomes runnable**

$$\text{p\_estcpu} \leftarrow \left(\frac{2 \cdot \text{load}}{2 \cdot \text{load} + 1}\right)^{\text{p\_slptime}} \cdot \text{p\_estcpu}$$

  - Approximates decay ignoring nice and past loads
- **Previous description based on [McKusick]**[a]

[a] See library.stanford.edu for off-campus access

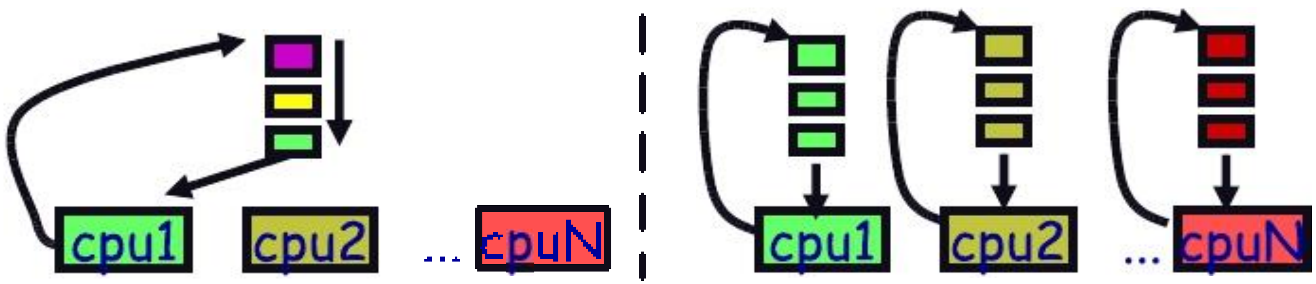
Pintos notes

- **Same basic idea for second half of project 1**
  - But 64 priorities, not 128
  - Higher numbers mean higher priority
  - Okay to have only one run queue if you prefer (less efficient, but we won't deduct points for it)
- **Have to negate priority equation:**

$$\text{priority} = 63 - \left\lceil \frac{\text{recent\_cpu}}{4} \right\rceil - 2 \cdot \text{nice}$$

Multiprocessor scheduling issues

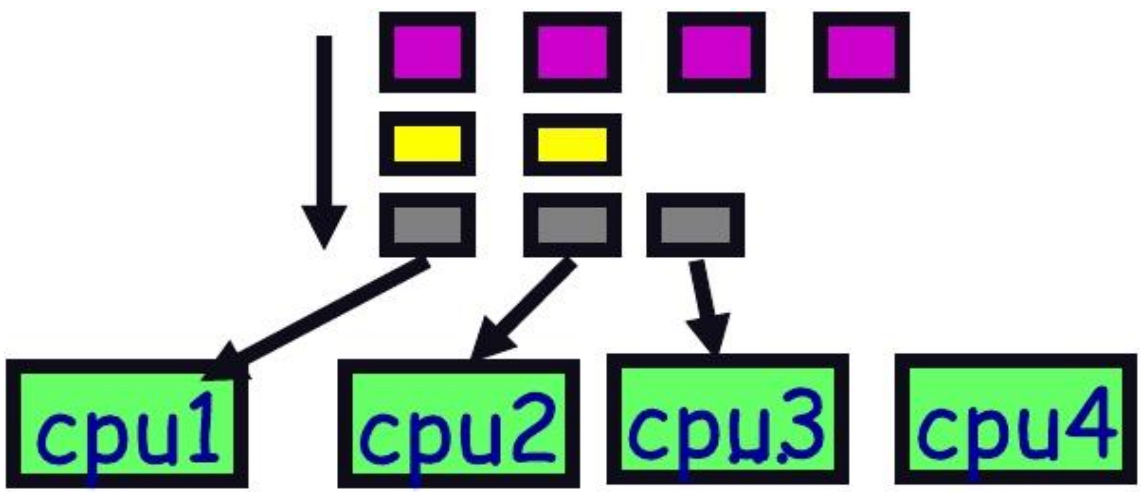
- **Must decide on more than which processes to run**
  - Must decide on **which CPU** to run which process
- **Moving between CPUs has costs**
  - More cache misses, depending on arch more TLB misses too
- ***Affinity scheduling*** **-** try to keep threads on same CPU



  - But also prevent load imbalances
  - Do *cost-benefit* analysis when deciding to migrate

Multiprocessor scheduling (cont)

- **Want related processes scheduled together**
  - Good if threads access same resources (e.g., cached files)
  - Even more important if threads communicate often, otherwise must context switch to communicate
- ***Gang scheduling* - schedule all CPUs synchronously**
  - With synchronized quanta, easier to schedule related processes/threads together



Thread scheduling

- **With thread library, have two scheduling decisions:**
  - ***Local Scheduling*** – Thread library decides which user thread to put onto an available kernel thread
  - ***Global Scheduling*** – Kernel decides which kernel thread to run next
- **Can expose to the user**
  - E.g., pthread_attr_setscope allows two choices
  - PTHREAD_SCOPE_SYSTEM – thread scheduled like a process (effectively one kernel thread bound to user thread – Will return ENOTSUP in user-level pthreads implementation)
  - PTHREAD_SCOPE_PROCESS – thread scheduled within the current process (may have multiple user threads multiplexed onto kernel threads)

Thread dependencies

- **Say *H* at high priority, *L* at low priority**
  - *L* acquires lock *l*.
  - Scene 1: *H* tries to acquire *l*, fails, spins. *L* never gets to run.
  - Scene 2: *H* tries to acquire *l*, fails, blocks. *M* enters system at medium priority. *L* never gets to run.
  - Both scenes are examples of ***priority inversion***
- **Scheduling = deciding who should make progress**
  - Obvious: a thread's importance should increase with the importance of those that depend on it.
  - Naive priority schemes violate this.

Priority donation

- **Say higher number = higher priority**
- **Example 1:** L **(prio 2),** M **(prio 4),** H **(prio 8)**
  - L holds lock l
  - M waits on l, L's priority raised to $L' = \max(M, L) = 4$
  - Then H waits on l, L's priority raised to $\max(H, L') = 8$
- **Example 2: Same threads**
  - L holds lock l, M holds lock l_2
  - M waits on l, L's priority now $L' = 4$ (as before)
  - Then H waits on l_2. M's priority goes to $M' = \max(H, M) = 8$, *and* L's priority raised to $\max(M', L') = 8$
- **Example 3:** L **(prio 2),** $M_1, \ldots M_{1000}$ **(all prio 4)**
  - L has l, and $M_1, \ldots, M_{1000}$ all block on l. L's priority is $\max(L, M_1, \ldots, M_{1000}) = 4$.

Review: Thread Package API

- tid thread_create (void (*fn) (void *), void *arg);
  - Create a new thread that calls function fn with arg
- void thread_exit ();
- void thread_join (tid thread);
- **The execution of multiple threads is interleaved**
- **Can have *non-preemptive* *threads*:**
  - One thread executes exclusively until it makes a blocking call.
- **Or *preemptive* *threads*:**
  - May switch to another thread between any two instructions.
- **Using multiple CPUs is inherently preemptive**
  - Even if you don't take CPU_0 away from thread T, another thread on CPU_1 can execute between any two instructions of T.

Program A

```
int flag1 = 0, flag2 = 0;

void p1 ( ) {
    flag1 = 1;
    if (!flag2) { critical_section_1 (); }
}

void p2 ( ) {
    flag2 = 1;
    if (!flag1) { critical_section_2 (); }
}

int main ( ) {
    tid id = thread_create (p1, NULL);
    p2 (); thread_join (id);
}
```

**Even though the threads might deadlock,
can both critical sections run simultaneously?**

Program B

```
int data = 0, ready = 0;

void p1 ( ) {
    data = 2000;
    ready = 1;
}

void p2 ( ) {
    while (!ready)
        ;
    use (data);
}

int main () { ... }
```

Can use( ) **be called with value 0?**

Program C

```
int a = 0, b = 0;

void p1 ( ) { a = 1; }

void p2 ( ) {
    if (a == 1)
        b = 1;
}

void p3 ( ) {
    if (b == 1)
        use (a);
}

int main () { ... }
```

Can use( ) **be called with value 0?**

Correct answers

- **Program A: I don't know**
- **Program B: I don't know**
- **Program C: I don't know**
- **Why?**
  - **It depends on your hardware**
  - If it provides ***sequential consistency***, then answers are all **No**
  - But not all hardware provides sequential consistency
- **Note: Examples and other slide content from [Adve & Gharachorloo]**

Sequential Consistency

- ***Sequential consistency***: The result of execution is as if all operations were executed in some sequential order, and the operations of each processor occurred in the order specified by the program. **[Lamport]**
- Boils down to two requirements:
  1. Maintaining ***program order*** on individual processors
  2. Ensuring ***write atomicity***
- **Without SC, multiple CPUs can be "worse" than preemptive threads**
  - May see results that cannot occur with any interleaving on 1 CPU
- **Why doesn't all hardware support sequential consistency?**

SC thwarts hardware optimizations

- **Complicates write buffers**
  - E.g., read flag(n) before flag($3 - n$) written through in Program A
- **Can't re-order overlapping write operations**
  - Concurrent writes to different memory modules
  - Coalescing writes to same cache line
- **Complicates non-blocking reads**
  - E.g., speculatively prefetch data in Program B
- **Makes cache coherence more expensive**
  - Must delay write completion until invalidation/update (Program B)
  - Can't allow overlapping updates if no globally visible order (Program C)

SC thwarts compiler optimizations

- **Code motion**
- **Caching value in register**
  - E.g., ready flag in Program B
- **Common subexpression elimination**
  - Could cause memory location to be read fewer times
- **Loop blocking**
  - Re-arrange loops for better cache performance
- **Software pipelining**
  - Move instructions across iterations of a loop to overlap instruction latency with branch cost

x86 consistency

- **x86 supports multiple consistency/caching models**
  - Memory Type Range Registers (MTRR) specify consistency for ranges of physical memory (e.g., frame buffer)
  - Page Attribute Table (PAT) allows control for each 4K page
- **Choices include:**
  - **WB**: Write-back caching (**the default**)
  - **WT**: Write-through caching (all writes go to memory)
  - **UC**: Uncacheable (for device memory)
  - **WC**: Write-combining – weak consistency & no caching
- **Some instructions have weaker consistency**
  - String instructions
  - Special "non-temporal" instructions that bypass cache

x86 WB consistency

- **Old x86s (e.g, 486, Pentium 1) had almost SC**
  - Exception: A read could finish before an earlier write to a different location
  - Which of Programs A, B, C might be affected?
- **Newer x86s let a processor read its own writes early**

x86 WB consistency

- **Old x86s (e.g, 486, Pentium 1) had almost SC**
  - Exception: A read could finish before an earlier write to a different location
  - Which of Programs A, B, C might be affected? **Just A**
- **Newer x86s let a processor read its own writes early**
  - E.g., both of these functions can return 2:

```
int flag1 = 0, flag2 = 0;

int p1 (void *ignored)
{
    register int f, g;
    flag1 = 1;
    f = flag1;
    g = flag2;
    return 2*f + g;
}

int p2 (void *ignored)
{
    register int f, g;
    flag2 = 1;
    f = flag2;
    g = flag1;
    return 2*f + g;
}
```

  - Older CPUs would wait at "f = ..." until store completes

x86 atomicity

- lock **prefix makes a memory instruction atomic**
  - Usually locks bus for duration of instruction (expensive!)
  - Can avoid locking if memory already exclusively cached
  - All lock instructions totally ordered
  - Other memory instructions cannot be re-ordered w. locked ones
- xchg **instruction is always locked (even w/o prefix)**
- **Special fence instructions can prevent re-ordering**
  - LFENCE – can't be reordered w. reads (or later writes)
  - SFENCE – can't be reordered w. writes
  - MFENCE – can't be reordered w. reads or writes

Assuming sequential consistency

- **Let's for now say we have sequential consistency**
- **Example concurrent code: Producer/Consumer**
  - buffer stores **BUFFER_SIZE** items
  - **count** is number of used slots
  - **in** is next empty buffer slot to fill (if any)
  - **out** is oldest filled slot to consume (if any)

```
void producer (void *ignored) {
    for (;;) {
        /* produce an item and put in nextProduced */
        while (count == BUFFER_SIZE)
            ; // do nothing
        buffer [in] = nextProduced;
        in = (in + 1) % BUFFER_SIZE;
        count++;
    }
}

void consumer (void *ignored) {
    for (;;) {
        while (count == 0)
            ; // do nothing
        nextConsumed =  buffer[out];
        out = (out + 1) % BUFFER_SIZE;
        count--;
        /*  consume the item in nextConsumed */
    }
}
```

- **What can go wrong here?**

Data races

- count **may have wrong value**
- **Possible implementation of** count++ **and** count--

count++	count--
register←count	register←count
register←register + 1	register←register − 1
count←register	count←register

- **Possible execution (count one less than correct):**

register←count
register←register + 1
register←count
register←register − 1
count←register
count←register

Data races (continued)

- **What about a single-instruction add?**
  - E.g., i386 allows single instruction addl $1, count
  - So implement count++/-- with one instruction
  - Now are we safe?

Data races (continued)

- **What about a single-instruction add?**
  - E.g., i386 allows single instruction addl $1, count
  - So implement count++/-- with one instruction
  - Now are we safe?
- **Not atomic on multiprocessor!**
  - Will experience exact same race condition
  - Can potentially make atomic with lock prefix
  - But lock very expensive
  - Compiler won't generate it, assumes you don't want penalty
- **Need solution to *critical section* problem**
  - Place count++ and count-- in critical section
  - Protect critical sections from concurrent execution

Critical Section: Desired solution

- ***Mutual Exclusion***
  - Only one thread can be in critical section at a time
- ***Progress***
  - Say no process currently in critical section (C.S.)
  - One of the processes trying to enter will eventually get in
- ***Bounded waiting***
  - Once a thread *T* starts trying to enter the critical section, there is a bound on the number of times other threads get in
- **Note progress vs. bounded waiting**
  - If no thread can enter C.S., don't have progress
  - If thread *A* waiting to enter C.S. while *B* repeatedly leaves and re-enters C.S. *ad infinitum*, don't have bounded waiting

Peterson's solution

- **Still assuming sequential consistency**
- **Assume two threads,** T_0 **and** T_1
- **Variables**
  - int not_turn; – not this thread's turn to enter C.S.
  - bool wants[2]; – wants[i] indicates if T_i wants to enter C.S.
- **Code:**

```
for (;;) { /* code in thread i */
   wants[i] = true;
   not_turn = i;
   while (wants[1-i] && not_turn == i)
      /* other thread wants in and not our turn, so loop */;
   Critical_section ();
   wants[i] = false;
   Remainder_section ();
}
```

Does Peterson's solution work?

```
for (;;) { /* code in thread i */
  wants[i] = true;
  not_turn = i;
  while (wants[1-i] && not_turn == i)
    /* other thread wants in and not our turn, so loop */;
  Critical_section ();
  wants[i] = false;
  Remainder_section ();
}
```

- **Mutual exclusion – can't both be in C.S.**
  - Would mean wants[0] == wants[1] == true,
    so not_turn would have blocked one thread from C.S.
- **Progress – If T_{1-i} not in C.S., can't block T_i**
  - Means wants[1-i] == false, so T_1 won't loop
- **Bounded waiting – similar argument to progress**
  - If T_i wants lock and T_{1-i} tries to re-enter, T_{1-i} will set
    not_turn = 1 - i, allowing T_i in

Mutexes

- **Peterson expensive, only works for 2 processes**
  - Can generalize to *n*, but for some fixed *n*
- **Want to insulate programmer from implementing synchronization primitives**
- **Thread packages typically provide *mutexes*:**

```
void mutex_init (mutex_t *m, ..);
void mutex_lock (mutex_t *m);
int mutex_trylock (mutex_t *m);
void mutex_unlock (mutex_t *m);
```

  - Only one thread acquires m at a time, others wait
  - **All global data should be protected by a mutex!**
- **OS kernels also need synchronization**
  - May or may not look like mutexes

Same concept, many names

- **Most popular application-level thread API: *pthreads***
  - Function names in this lecture all based on *pthreads*
  - Just add pthread_ prefix
  - E.g., pthread_mutex_t, pthread_mutex_lock, . . .
- **Same abstraction in Pintos under different name**
  - Data structure is struct lock
  - void lock_init (struct lock *);
  - void lock_acquire (struct lock *);
  - bool lock_try_acquire (struct lock *);
  - void lock_release (struct lock *);
- **Extra Pintos feature:**
  - Release checks that the lock was acquired by the same thread
  - bool lock_held_by_current_thread (struct lock *lock);

Improved producer

```
mutex_t mutex = MUTEX_INITIALIZER;

void producer (void *ignored) {
    for (;;) {
        /* produce an item and put in nextProduced */

        mutex_lock (&mutex);
        while (count == BUFFER_SIZE) {
          mutex_unlock (&mutex);   // <--- Why?
          thread_yield ();
          mutex_lock (&mutex);
        }

        buffer [in] = nextProduced;
        in = (in + 1) % BUFFER_SIZE;
        count++;
        mutex_unlock (&mutex);
    }
}
```

Improved consumer

```
void consumer (void *ignored) {
    for (;;) {
        mutex_lock (&mutex);
        while (count == 0) {
            mutex_unlock (&mutex);
            thread_yield ();
            mutex_lock (&mutex);
        }

        nextConsumed =  buffer[out];
        out = (out + 1) % BUFFER_SIZE;
        count--;
        mutex_unlock (&mutex);

        /*  consume the item in nextConsumed */
    }
}
```

Condition variables

- **Busy-waiting in application is a bad idea**
  - Thread consumes CPU even when can’t make progress
  - Unnecessarily slows other threads and processes
- **Better to inform scheduler of which threads can run**
- **Typically done with condition variables**
- void cond_init (cond_t *, ...);
  - Initialize
- void cond_wait (cond_t *c, mutex_t *m);
  - Atomically unlock m and sleep until c signaled
  - Then re-acquire m and resume executing
- void cond_signal (cond_t *c);
  void cond_broadcast (cond_t *c);
  - Wake one/all threads waiting on c

Improved producer

```
mutex_t mutex = MUTEX_INITIALIZER;
cond_t nonempty = COND_INITIALIZER;
cond_t nonfull = COND_INITIALIZER;

void producer (void *ignored) {
    for (;;) {
        /* produce an item and put in nextProduced */

        mutex_lock (&mutex);
        while (count == BUFFER_SIZE)
          cond_wait (&nonfull, &mutex);

        buffer [in] = nextProduced;
        in = (in + 1) % BUFFER_SIZE;
        count++;
        cond_signal (&nonempty);
        mutex_unlock (&mutex);
    }
}
```

Improved consumer

```
void consumer (void *ignored) {
    for (;;) {
        mutex_lock (&mutex);
        while (count == 0)
           cond_wait (&nonempty, &mutex);

        nextConsumed =   buffer[out];
        out = (out + 1) % BUFFER_SIZE;
        count--;
        cond_signal (&nonfull);
        mutex_unlock (&mutex);

        /*   consume the item in nextConsumed */
    }
}
```

Condition variables (continued)

- **Why must** cond_wait **both release mutex & sleep?**
- **Why not separate mutexes and condition variables?**

```
while (count == BUFFER_SIZE) {
    mutex_unlock (&mutex);
    cond_wait (&nonfull);
    mutex_lock (&mutex);
}
```

Condition variables (continued)

- **Why must** cond_wait **both release mutex & sleep?**
- **Why not separate mutexes and condition variables?**

```
while (count == BUFFER_SIZE) {
    mutex_unlock (&mutex);
    cond_wait (&nonfull);
    mutex_lock (&mutex);
}
```

- **Can end up stuck waiting when bad interleaving**

```
PRODUCER                                CONSUMER
while (count == BUFFER_SIZE);
mutex_unlock (&mutex);
                                          mutex_lock (&mutex);
                                          ...
                                          count--;
                                          cond_signal (&nonfull);
cond_wait (&nonfull);
```

Other thread package features

- **Alerts – cause exception in a thread**
- **Timedwait – timeout on condition variable**
- **Shared locks – concurrent read accesses to data**
- **Thread priorities – control scheduling policy**
  - Mutex attributes allow various forms of *priority donation* (will be familiar concept after lab 1)
- **Thread-specific global data**
- **Different synchronization primitives (in a few slides)**
  - Monitors
  - Semaphores

Implementing synchronization

- **User-visible mutex is straight-forward data structure**

```
typedef struct mutex {
    bool is_locked;            /* true if locked */
    thread_id_t owner;         /* thread holding lock, if locked */
    thread_list_t waiters; /* threads waiting for lock */

    lower_level_lock_t lk; /* Protect above fields */
};
```

- **Need lower-level lock** lk **for mutual exclusion**
  - Internally, mutex_* functions bracket code with lock(mutex->lk) . . . unlock(mutex->lk)
  - Otherwise, data races! (E.g., two threads manipulating waiters)
- **How to implement** lower_level_lock_t**?**
  - Could use Peterson's algorithm, but typically a bad idea (too slow and don't know maximum number of threads)

Approach #1: Disable interrupts

- **Only for apps with** $n : 1$ **threads (1 kthread)**
  - Cannot take advantage of multiprocessors
  - But sometimes most efficient solution for uniprocessors
- **Have per-thread "do not interrupt" (DNI) bit**
- lock (lk)**: sets thread's DNI bit**
- **If timer interrupt arrives**
  - Check interrupted thread's DNI bit
  - If DNI clear, preempt current thread
  - If DNI set, set "interrupted" (I) bit & resume current thread
- unlock (lk)**: clears DNI bit *and* checks I bit**
  - If I bit is set, immediately yields the CPU

Approach #2: Spinlocks

- **Most CPUs support atomic read-[modify-]write**
- **Example:** int test_and_set (int *lockp);
  - Atomically sets *lockp = 1 and returns old value
  - Special instruction – can't be implemented in portable C
- **Use this instruction to implement *spinlocks*:**

```
#define lock(lockp)     while (test_and_set (lockp))
#define trylock(lockp)  (test_and_set (lockp) == 0)
#define unlock(lockp)   *lockp = 0
```

- **Spinlocks implement mutex's** lower_level_lock_t
- **Can you use spinlocks instead of mutexes?**
  - Wastes CPU, especially if thread holding lock not running
  - Mutex functions have short C.S., less likely to be preempted
  - On multiprocessor, sometimes good to spin for a bit, then yield

Synchronization on x86

- **Test-and-set only one possible atomic instruction**
- **x86** xchg **instruction, exchanges reg with mem**
  - Can use to implement test-and-set

```
_test_and_set:
          movl    8(%esp), %edx     # %edx = lockp
          movl    $1, %eax          # %eax = 1
          xchgl   %eax, (%edx)      # swap (%eax, *lockp)
          ret
```

- **CPU locks memory system around read and write**
  - Recall xchgl always acts like it has lock prefix
  - Prevents other uses of the bus (e.g., DMA)
- **Usually runs at memory bus speed, not CPU speed**
  - Much slower than cached read/buffered write

Kernel Synchronization

- **Should kernel use locks or disable interrupts?**
- **Old UNIX had non-preemptive threads, no mutexes**
  - Interface designed for single CPU, so count++ etc. not data race
  - . . . *Unless* memory shared with an interrupt handler

```
int x = splhigh (); // Disable interrupts
// Touch data shared with interrupt handler
splx (x);           // Restore previous state
```

  - C.f., Pintos intr_disable / intr_set_level
- **Used arbitrary pointers like condition variables**
  - int [t]sleep (void *ident, int priority, ...);
    put thread to sleep; will wake up at priority (~cond_wait)
  - int wakeup (void *ident);
    wake up all threads sleeping on ident (~cond_broadcast)

Kernel locks

- **Nowadays, should design for multiprocessors**
  - Even if first version of OS is for uniprocessor
  - Someday may want multiple CPUs and need *preemptive* threads
  - That's why Pintos uses locks
- **Multiprocessor performance needs fine-grained locks**
  - Want to be able to call into the kernel on multiple CPUs
- **If kernel has locks, should it ever disable interrupts?**

Kernel locks

- **Nowadays, should design for multiprocessors**
  - Even if first version of OS is for uniprocessor
  - Someday may want multiple CPUs and need *preemptive* threads
  - That's why Pintos uses locks
- **Multiprocessor performance needs fine-grained locks**
  - Want to be able to call into the kernel on multiple CPUs
- **If kernel has locks, should it ever disable interrupts?**
  - Yes! Can't sleep in interrupt handler, so can't wait for lock
  - So even modern OSes have support for disabling interrupts
  - Often uses DNI trick, which is cheaper than masking interrupts in hardware

Semaphores [Dijkstra]

- **A *Semaphore* is initialized with an integer** N
- **Provides two functions:**
  - sem_wait (S) (originally called P, called ***sema_down*** in Pintos)
  - sem_signal (S) (originally called V, called ***sema_up*** in Pintos)
- **Guarantees** sem_wait **will return only** N **more times than** sem_signal **called**
  - Example: If $N == 1$, then semaphore is a mutex with sem_wait as lock and sem_signal as unlock
- **Semaphores allow elegant solutions to some problems**

Semaphore

A semaphore is a structure consisting of 2 parts:

```
struct semaphore {
    int count;  // number of resources available
    queue Q;  // queue of process/thread ids of blocked
}
```

Shorthand notation:

semaphore S = 1 → S.count = 1, S.Q = { }

Operations on Semaphores

There are two basic semaphore operations:

sem_wait(S):

    if (S.count > 0) then S.count = S.count -1;

    else block calling process in S.Q;

sem_signal(S):

    if (S.Q is non-empty) then wakeup a process in S.Q;

    else S.count = S.count + 1;

Semaphore Example: Mutual Exclusion

Semaphore S = 1;

Thread A:

```
sem_wait(S);
  (do work in critical section CS);
sem_signal(S);
```

Thread B:

```
sem_wait(S);
  (do work in CS);
sem_signal(S);
```

Semaphore Example: Order Execution

Semaphore S = 0;

Thread A → Thread B:

Thread A:

(do work);

sem_signal(S);

Thread B:

sem_wait(S);

(do work);

Semaphore producer/consumer

- **Can re-write producer/consumer to use three semaphores**
- **Semaphore** mutex **initialized to 1**
  - Used as mutex, protects buffer, in, out. . .
- **Semaphore** full **initialized to 0**
  - To block consumer when buffer empty
- **Semaphore** empty **initialized to N**
  - To block producer when queue full

```
void producer (void *ignored) {
    for (;;) {
        /* produce an item and put in nextProduced */
        sem_wait (&empty);
        sem_wait (&mutex);
        buffer [in] = nextProduced;
        in = (in + 1) % BUFFER_SIZE;
        sem_signal (&mutex);
        sem_signal (&full);
    }
}

void consumer (void *ignored) {
    for (;;) {
        sem_wait (&full);
        sem_wait (&mutex);
        nextConsumed =  buffer[out];
        out = (out + 1) % BUFFER_SIZE;
        sem_signal (&mutex);
        sem_signal (&empty);
        /*  consume the item in nextConsumed */
    }
}
```

Summary

- Read Ch. 1-6
- Processes and Threads (Ch. 4)
- Process Scheduling (Ch. 5)
- Synchronization (Ch. 6)
- Project 1 – Scheduling and Synchronization