CS194-24 Advanced Operating Systems Structures and Implementation Lecture 7

Parallelism and Synchronization

February 12^{th}, 2014
Prof. John Kubiatowicz
http://inst.eecs.berkeley.edu/~cs194-24

Goals for Today

- Threads/Concurrency (continued)
- Synchronization

Interactive is important!
Ask Questions!

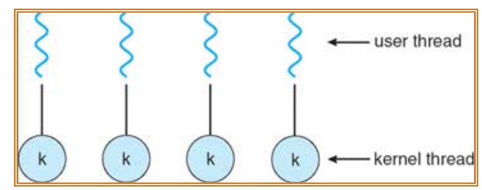
Note: Some slides and/or pictures in the following are adapted from slides ©2013

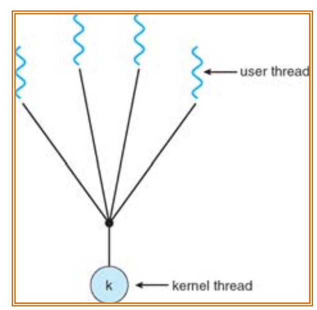
Recall: Scheduling Policy Goals/Criteria

- **Scheduling**: deciding which threads are given access to resources from moment to moment
- Minimize Response Time
  - Minimize elapsed time to do an operation (or job)
  - Response time is what the user sees:
    - » Time to echo a keystroke in editor
    - » Time to compile a program
- Maximize Throughput
  - Maximize operations (or jobs) per second
  - Two parts to maximizing throughput
    - » Minimize overhead (for example, context-switching)
    - » Efficient use of resources (CPU, disk, memory, etc)
- Minimize missed deadlines (Realtime)
  - Efficiency is important, but **predictability** is essential
  - In RTS, performance guarantees are:
    - » Task- and/or class centric
    - » Often ensured a priori

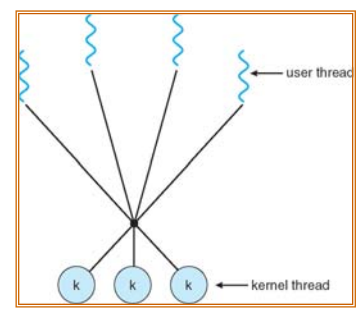
Recall: Threading models mentioned by Silbershatz book

Simple One-to-One Threading Model





Many-to-One



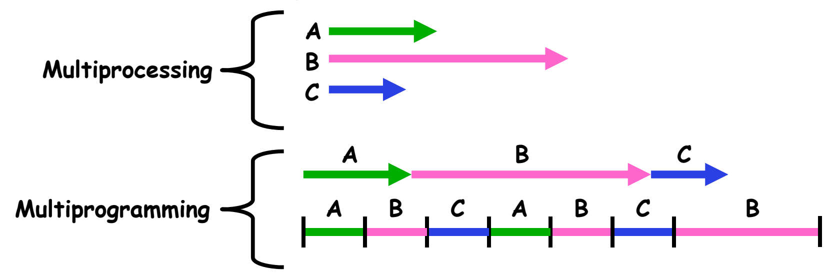
Many-to-Many

Thread Level Parallelism (TLP)

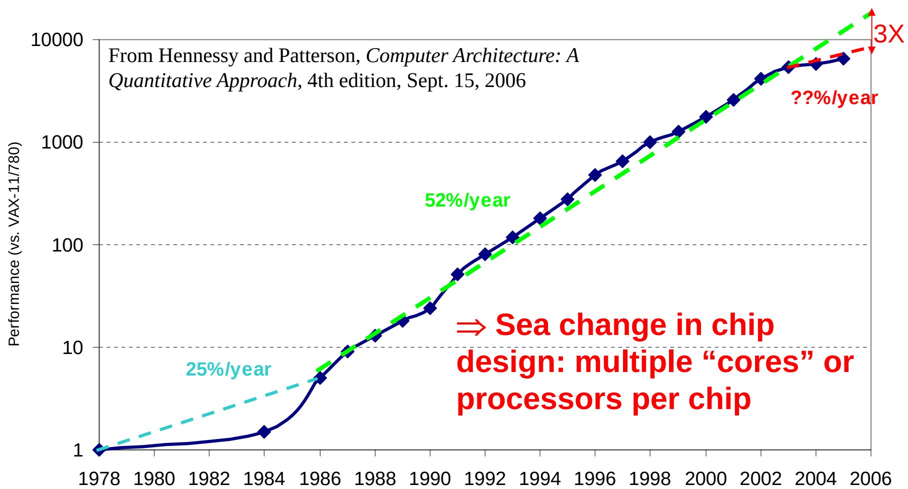
- In modern processors, Instruction Level Parallelism (ILP) exploits implicit parallel operations within a loop or straight-line code segment
- Thread Level Parallelism (TLP) explicitly represented by the use of multiple threads of execution that are inherently parallel
  - Threads can be on a single processor
  - Or, on multiple processors
- Concurrency vs Parallelism
  - Concurrency is when two tasks can start, run, and complete in overlapping time periods. It doesn't necessarily mean they'll ever both be running at the same instant.
    - » For instance, multitasking on a single-threaded machine.
  - Parallelism is when tasks literally run at the same time, eg. on a multicore processor.
- Goal: Use multiple instruction streams to improve
  - Throughput of computers that run many programs
  - Execution time of multi-threaded programs

Multiprocessing vs Multiprogramming

- Remember Definitions:
  - Multiprocessing ≡ Multiple CPUs
  - Multiprogramming ≡ Multiple Jobs or Processes
  - Multithreading ≡ Multiple threads per Process
- What does it mean to run two threads "concurrently"?
  - Scheduler is free to run threads in any order and interleaving: FIFO, Random, …
  - Dispatcher can choose to run each thread to completion or time-slice in big chunks or small chunks

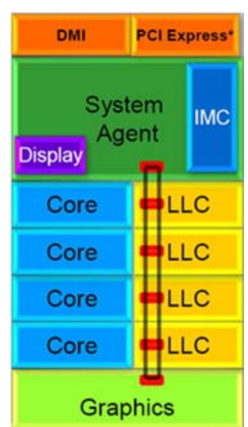


Recall: Slowdown in Joy's law of Performance



- VAX : 25%/year 1978 to 1986
- RISC + x86: 52%/year 1986 to 2002
- RISC + x86: ??%/year 2002 to present

Recall: Chip-scale features of SandyBridge



- Significant pieces:
  - Four OOO cores
    - » New Advanced Vector eXtensions (256-bit FP)
    - » AES instructions
    - » Instructions to help with Galois-Field mult
    - » 4 μ-ops/cycle
  - Integrated GPU
  - System Agent (Memory and Fast I/O)
  - Shared L3 cache divided in 4 banks
  - On-chip Ring bus network
    - » Both coherent and non-coherent transactions
    - » High-BW access to L3 Cache
- Integrated I/O
  - Integrated memory controller (IMC)
    - » Two independent channels of DDR3 DRAM
  - High-speed PCI-Express (for Graphics cards)
  - DMI Connection to SouthBridge (PCH)

Correctness for systems with concurrent threads

- If dispatcher can schedule threads in any way, programs must work under all circumstances
  - Can you test for this?
  - How can you know if your program works?
- Independent Threads:
  - No state shared with other threads
  - Deterministic ⇒ Input state determines results
  - Reproducible ⇒ Can recreate Starting Conditions, I/O
  - Scheduling order doesn't matter (if `switch()` works!!!)
- Cooperating Threads:
  - Shared State between multiple threads
  - Non-deterministic
  - Non-reproducible
- Non-deterministic and Non-reproducible means that bugs can be intermittent
  - Sometimes called "Heisenbugs"

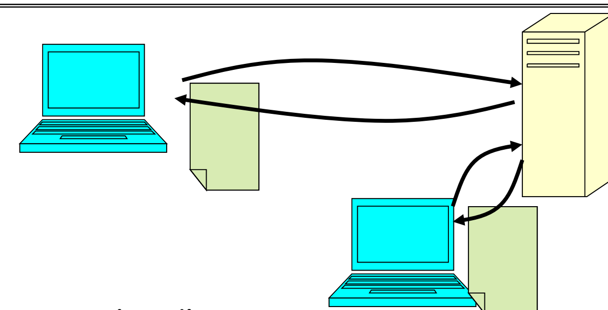
Interactions Complicate Debugging

- Is any program truly independent?
  - Every process shares the file system, OS resources, network, etc
  - Extreme example: buggy device driver causes thread A to crash "independent thread" B
- You probably don't realize how much you depend on reproducibility:
  - Example: Evil C compiler
    - » Modifies files behind your back by inserting errors into C program unless you insert debugging code
  - Example: Debugging statements can overrun stack
- Non-deterministic errors are really difficult to find
  - Example: Memory layout of kernel+user programs
    - » depends on scheduling, which depends on timer/other things
    - » Original UNIX had a bunch of non-deterministic errors
  - Example: Something which does interesting I/O
    - » User typing of letters used to help generate secure keys

Why allow cooperating threads?

- People cooperate; computers help/enhance people's lives, so computers must cooperate
  - By analogy, the non-reproducibility/non-determinism of people is a notable problem for "carefully laid plans"
- Advantage 1: Share resources
  - One computer, many users
  - One bank balance, many ATMs
    - » What if ATMs were only updated at night?
  - Embedded systems (robot control: coordinate arm & hand)
- Advantage 2: Speedup
  - Overlap I/O and computation
    - » Many different file systems do read-ahead
  - Multiprocessors – chop up program into parallel pieces
- Advantage 3: Modularity
  - More important than you might think
  - Chop large problem up into simpler pieces
    - » To compile, for instance, gcc calls cpp | cc1 | cc2 | as | ld
    - » Makes system easier to extend

Recall: High-level Example: Web Server



- Server must handle many requests
- Non-cooperating version:

```
serverLoop() {
    con = AcceptCon();
    ProcessFork(ServiceWebPage(),con);
}
```

- What are some disadvantages of this technique?

Threaded Web Server

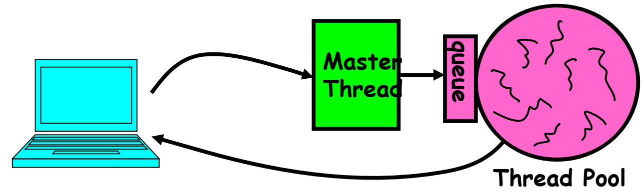
- Now, use a single process
- Multithreaded (cooperating) version:

```
serverLoop() {
      connection = AcceptCon();
      ThreadFork(ServiceWebPage(),connection);
   }
```

- Looks almost the same, but has many advantages:
  - Can share file caches kept in memory, results of CGI scripts, other things
  - Threads are *much* cheaper to create than processes, so this has a lower per-request overhead
- Question: would a user-level (say one-to-many) thread package make sense here?
  - When one request blocks on disk, all block…
- What about Denial of Service attacks or digg / Slash-dot effects?

Thread Pools

- Problem with previous version: Unbounded Threads
  - When web-site becomes too popular – throughput sinks
- Instead, allocate a bounded "pool" of worker threads, representing the maximum level of multiprogramming




```
master() {
   allocThreads(worker,queue);
   while(TRUE) {
      con=AcceptCon();
      Enqueue(queue,con);
      wakeUp(queue);
   }
}
```

```
worker(queue) {
   while(TRUE) {
      con=Dequeue(queue);
      if (con==null)
         sleepOn(queue);
      else
         ServiceWebPage(con);
   }
}
```


Administrivia

- Not much administrivia today
  - This lecture and next will have some duplication with CS162, but I believe it is necessary
- How are the topics going?
  - Too much/too little detail?
  - Too much/too little overlap with CS162?
- Midterm I: Wed 3/12 (Four weeks from today)
  - 2-Hour exam in 3 hours
  - Probably evening exam: 5-8pm?
  - Topics: Everything up to previous Monday fair game
- No class next Monday (2/17, Presidents' Day)

Other types of Linux Concurrency

- SoftIRQs and Tasklets
  - Utilized to defer work from interrupt handlers
  - Runs as soon as interrupts are finished
  - Tasklets built on top of SoftIRQs
  - When to choose them:
    - » SoftIRQs can run simultaneously on different processors
    - » No two Tasklets of same type run simultaneously
- Work Queues
  - Built on top of Kernel Threads
  - Simple mechanism for scheduling work to run in kernel
  - Units of parallelism can sleep!

Common Notions of Thread Creation

- **cobegin/coend**
```
cobegin
    job1(a1);
    job2(a2);
coend
```
  - Statements in block may run in parallel
  - cobegins may be nested
  - Scoped, so you cannot have a missing coend
- **fork/join**
```
tid1 = fork(job1, a1);
job2(a2);
join tid1;
```
  - Forked procedure runs in parallel
  - Wait at join point if it's not finished
- **future**
```
v = future(job1(a1));
… = …v…;
```
  - Future possibly evaluated in parallel
  - Attempt to use return value will wait
- **forall**
```
forall(I from 1 to N)
    C[I] = A[I] + B[I]
end
```
  - Separate thread launched for each iteration
  - Implicit join at end
- Threads expressed in the code may not turn into independent computations
  - Only create threads if processors idle
  - Example: Thread-stealing runtimes such as cilk

Overview of POSIX Threads

- Pthreads: The POSIX threading interface
  - System calls to create and synchronize threads
  - Should be relatively uniform across UNIX-like OS platforms
  - Originally IEEE POSIX 1003.1c
- Pthreads contain support for
  - Creating parallelism
  - Synchronizing
  - No explicit support for communication, because shared memory is implicit; a pointer to shared data is passed to a thread
    - » Only for HEAP! Stacks not shared

Forking POSIX Threads

Signature:
```
int pthread_create(pthread_t *,
                   const pthread_attr_t *,
                   void * (*)(void *),
                   void *);
```
Example call:
```
errcode = pthread_create(&thread_id; &thread_attribute
                         &thread_fun; &fun_arg);
```

- thread_id is the thread id or handle (used to halt, etc.)
- thread_attribute various attributes
  - Standard default values obtained by passing a NULL pointer
  - Sample attribute: minimum stack size
- thread_fun the function to be run (takes and returns void*)
- fun_arg an argument can be passed to thread_fun when it starts
- errorcode will be set nonzero if the create operation fails

Simple Threading Example (pThreads)

E.g., compile using gcc –lpthread

```
void* SayHello(void *foo) {
  printf( "Hello, world!\n" );
  return NULL;
}

int main() {
  pthread_t threads[16];
  int tn;
  for(tn=0; tn<16; tn++) {
    pthread_create(&threads[tn], NULL, SayHello, NULL);
  }
  for(tn=0; tn<16 ; tn++) {
    pthread_join(threads[tn], NULL);
  }
  return 0;
}
```

Shared Data and Threads

- Variables declared outside of main are shared
- Objects allocated on the heap may be shared (if pointer is passed)
- Variables on the stack are private: passing pointer to these around to other threads can cause problems

- Often done by creating a large "thread data" struct, which is passed into all threads as argument

```
char *message = "Hello World!\n";

pthread_create(&thread1, NULL,
               print_fun,(void*) message);
```


Loop Level Parallelism

- Many application have parallelism in loops

```
double stuff [n][n];
for (int i = 0; i < n; i++)
  for (int j = 0; j < n; j++)
    … pthread_create (…, update_stuff, …,
                      &stuff[i][j]);
```

- But overhead of thread creation is nontrivial
  - update_stuff should have a significant amount of work
- Common Performance Pitfall: Too many threads
  - The cost of creating a thread is 10s of thousands of cycles on modern architectures
  - Solution: Thread blocking: use a small # of threads, often equal to the number of cores/processors or hardware threads

Some More Pthread Functions

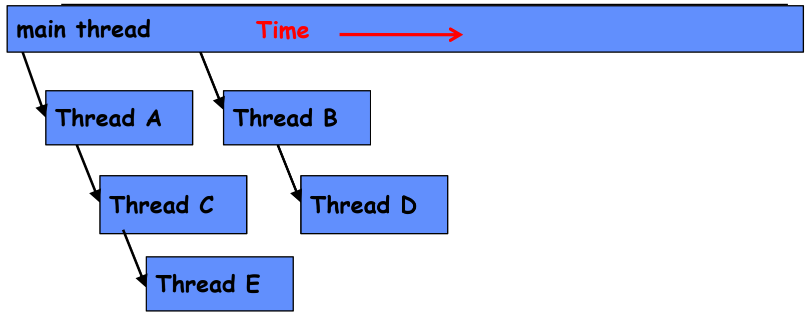
- pthread_yield();
  - Informs the scheduler that the thread is willing to yield its quantum, requires no arguments.
- pthread_exit(void *value);
  - Exit thread and pass value to joining thread (if exists)
- pthread_join(pthread_t *thread, void **result);
  - Wait for specified thread to finish. Place exit value into *result.

Others:

- pthread_t me; me = pthread_self();
  - Allows a pthread to obtain its own identifier pthread_t thread;
- pthread_detach(thread);
  - Informs the library that the threads exit status will not be needed by subsequent pthread_join calls resulting in better threads performance.

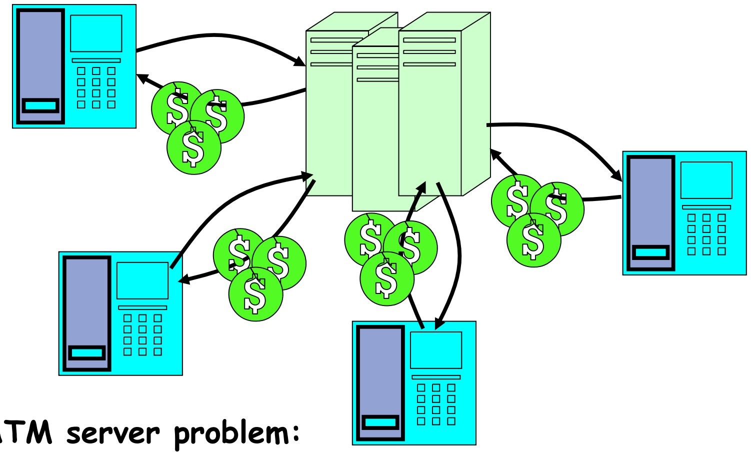
For more information consult the library or the man pages, e.g., man -k pthread..

Thread Scheduling



- Once created, when will a given thread run?
  - It is up to the Operating System or hardware, but it will run eventually, even if you have more threads than cores
  - But – scheduling may be non-ideal for your application
- Programmer can provide hints or affinity in some cases
  - E.g., create exactly P threads and assign to P cores
- Can provide user-level scheduling for some systems
  - Application-specific tuning based on programming model
  - Work in the ParLAB on making user-level scheduling easy to do (Lithe, PULSE)

Recall: ATM Bank Server



- ATM server problem:
  - Service a set of requests
  - Do so without corrupting database
  - Don't hand out too much money

ATM bank server example

- Suppose we wanted to implement a server process to handle requests from an ATM network:

```
BankServer() {
   while (TRUE) {
      ReceiveRequest(&op, &acctId, &amount);
      ProcessRequest(op, acctId, amount);
   }
}
ProcessRequest(op, acctId, amount) {
   if (op == deposit) Deposit(acctId, amount);
   else if …
}
Deposit(acctId, amount) {
   acct = GetAccount(acctId); /* may use disk I/O */
   acct->balance += amount;
   StoreAccount(acct); /* Involves disk I/O */
}
```

- How could we speed this up?
  - More than one request being processed at once
  - Event driven (overlap computation and I/O)
  - Multiple threads (multi-proc, or overlap comp and I/O)

Event Driven Version of ATM server

- Suppose we only had one CPU
  - Still like to overlap I/O with computation
  - Without threads, we would have to rewrite in event-driven style
- Example

```
BankServer() {
   while(TRUE) {
      event = WaitForNextEvent();
      if (event == ATMRequest)
         StartOnRequest();
      else if (event == AcctAvail)
         ContinueRequest();
      else if (event == AcctStored)
         FinishRequest();
   }
}
```

  - What if we missed a blocking I/O step?
  - What if we have to split code into hundreds of pieces which could be blocking?
  - This technique is used for graphical programming

Can Threads Make This Easier?

- Threads yield overlapped I/O and computation without "deconstructing" code into non-blocking fragments
  - One thread per request
- Requests proceeds to completion, blocking as required:

```
Deposit(acctId, amount) {
  acct = GetAccount(actId); /* May use disk I/O */
  acct->balance += amount;
  StoreAccount(acct);      /* Involves disk I/O */
}
```

- Unfortunately, shared state can get corrupted:

```
        Thread 1                      Thread 2
load r1, acct->balance
                               load r1, acct->balance
                               add r1, amount2
                               store r1, acct->balance
add r1, amount1
store r1, acct->balance
```

Atomic Operations

- Atomic Operation: an operation that always runs to completion or not at all
  - It is *indivisible:* it cannot be stopped in the middle and state cannot be modified by someone else in the middle
  - Fundamental building block – if no atomic operations, then have no way for threads to work together
- On most machines, memory references and assignments (i.e. loads and stores) of words are atomic
- Many instructions are not atomic
  - Double-precision floating point store often not atomic
  - VAX and IBM 360 had an instruction to copy a whole array
- Some architectures can turn non-atomic instruction into atomic ones
  - E.g. x86 – use "lock" prefix on an instruction
- Synchronization: using atomic operations to ensure cooperation between threads
  - For now, only loads and stores are atomic
  - We are going to show that its hard to build anything useful with only reads and writes

Using Locks for Synchronization

- Lock: prevents someone from doing something
  - Lock before entering critical section and before accessing shared data
  - Unlock when leaving, after accessing shared data
  - Wait if locked
    - » Important idea: all synchronization involves waiting
- Mutual Exclusion: ensuring that only one thread does a particular thing at a time
  - One thread *excludes* the other while doing its task
- Critical Section: piece of code that only one thread can execute at once. Only one thread at a time will get into this section of code.
  - Critical section is the result of mutual exclusion
  - Critical section and mutual exclusion are two ways of describing the same thing.

Using Locks to Protect Shared Variable

- Consider Previous Example:

```
Deposit(acctId, amount) {
  Acquire(depositlock);
  acct = GetAccount(actId);
  acct->balance += amount;
  StoreAccount(acct);
  Release(depositlock);
}
```

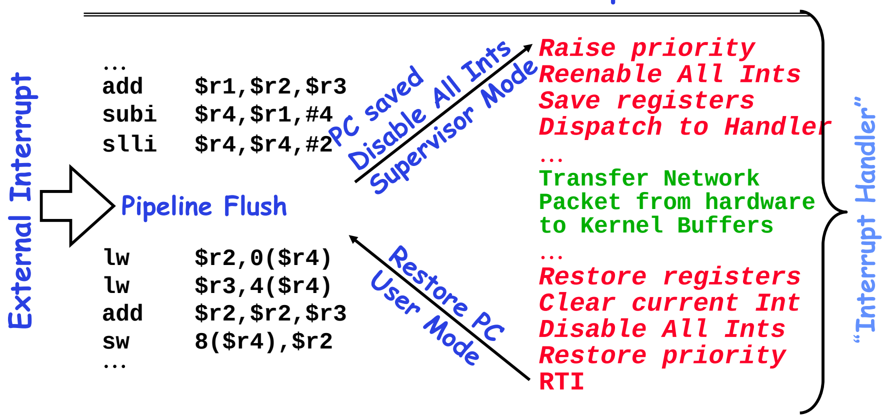
Critical Section (lines Acquire(depositlock) through Release(depositlock))

- Locking Behavior:
  - Only one critical section can be running at once!
    - » Second Acquire() before release ⇒ second thread waits
  - As soon as Release() occurs, another Acquire() can happen
  - If many threads request lock acquisition at same time:
    - » Might get livelock, depending on what happens on Release()
- Result of using locks: three instructions in critical section become Atomic! (cannot be separated)

Review: Ways of entering the kernel/ changing the flow of control

- The Timer Interrupt:
  - Callbacks scheduled to be called when timer expires
  - Cause of scheduler events – change which process of thread is running
- System Calls
  - Controlled function call into kernel from user space
  - User-level code stops, kernel-level code
  - What about asynchronous system calls?
- Normal Interrupts
  - Entered via hardware signal
  - Typically *Asynchronous* to the instruction stream
  - Often structured in some sort of hierarchy (some interrupts higher priority than others
- Exceptions:
  - Instruction execution fails for some reason
  - Typically *Synchronous* to the instruction stream

Recall: Network Interrupt



- Disable/Enable All Ints ⇒ Internal CPU disable bit
  - RTI reenables interrupts, returns to user mode
- Raise/lower priority: change interrupt mask
- Software interrupts can be provided entirely in software at priority switching boundaries

Naïve use of Interrupt Enable/Disable

- How can we build multi-instruction atomic operations?
  - Scheduler gets control in two ways.
    - » Internal: Thread does something to relinquish the CPU
    - » External: Interrupts cause dispatcher to take CPU
  - On a uniprocessor, can avoid context-switching by:
    - » Avoiding internal events (although virtual memory tricky)
    - » Preventing external events by disabling interrupts
- Consequently, naïve Implementation of locks:

```
LockAcquire { disable Ints; }
LockRelease { enable Ints; }
```

- Problems with this approach:
  - Can't let user do this! Consider following:

```
LockAcquire();
While(TRUE) {;}
```

  - Real-Time system—no guarantees on timing!
    - » *Critical Sections might be arbitrarily long*
  - What happens with I/O or other important events?
    - » "Reactor about to meltdown. Help?"

Implementation of Locks by Disabling Interrupts?

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

```
int value = FREE;
```

```
Acquire() {
   disable interrupts;
   if (value == BUSY) {
      put thread on wait queue;
      Go to sleep();
      // Enable interrupts?
   } else {
      value = BUSY;
   }
   enable interrupts;
}
```

```
Release() {
   disable interrupts;
   if (anyone on wait queue) {
      take thread off wait queue
      Place on ready queue;
   } else {
      value = FREE;
   }
   enable interrupts;
}
```


Typical Linux Interfaces

- Disabling and Enabling Interrupts on the Linux Kernel:

```
local_irq_disable();
/* interrupts are disabled ... */
local_irq_enable();
```

  - These operations often single assembly instructions
    - » The *only* work for local processor!
    - » If competing with another processor, but use other form of synchronization
  - Dangerous if called when interrupts already disabled
    - » Then, when you code reenables, you will change semantics
- Saving and restoring interrupt state first:

```
unsigned long flags;

local_irq_save(flags);     // Save state
/* Do whatever, including disable/enable*/
local_irq_restore(flags);  // Restore
```

- State of the system

```
in_interrupt();  // In handler or bottom half
in_irq();        // Specifically in handler
```

Linux Interrupt control (Con't)

- No more global `cli()`!
  - Used to be that `cli()/sti()` could be used to enable and disable interrupts on all processors
  - First deprecated (2.5), then removed (2.6)
    - » Could serialize device drivers across all processors!
    - » Just a bad idea
  - Better option?
    - » Fine-grained spin-locks between processors (more later)
    - » Local interrupt control for local processor
- Disabling specific interrupt (nestable)

```
disable_irq(irq);         // Wait current handlers
disable_irq_nosync(irq);  // Don't waitcurrent handler
enable_irq(irq);          // Reenable line
synchronize_irq(irq);     // Wait for current handler
```

  - Not great for buses with multiple interrupts per line, such as PCI! More when we get into device drivers.

How to implement locks? Atomic Read-Modify-Write instructions

- Problem with previous solution?
  - Can't let users disable interrupts! (Why?)
  - Doesn't work well on multiprocessor
    - » Disabling interrupts on all processors requires messages and would be very time consuming
- Alternative: atomic instruction sequences
  - These instructions read a value from memory and write a new value atomically
  - Hardware is responsible for implementing this correctly
    - » on both uniprocessors (not too hard)
    - » and multiprocessors (requires help from cache coherence protocol)
  - Unlike disabling interrupts, can be used on both uniprocessors and multiprocessors

Atomic Instructions

```
• test&set (&address) {      /* most architectures */
      result = M[address];
      M[address] = 1;
      return result;
  }
• swap (&address, register) { /* x86 */
      temp = M[address];
      M[address] = register;
      register = temp;
  }
• compare&swap (&address, reg1, reg2) { /* 68000 */
      if (reg1 == M[address]) {
          M[address] = reg2;
          return success;
      } else {
          return failure;
      }
  }
• load-linked&store conditional(&address) {
      /* R4000, alpha */
      loop:
          ll r1, M[address];
          movi r2, 1;          /* Can do arbitrary comp */
          sc r2, M[address];
          beqz r2, loop;
  }
```


Implementing Locks with test&set: Spin Lock

- Another flawed, but simple solution:

```
int value = 0; // Free
Acquire() {
  while (test&set(value)); // while busy
}
Release() {
  value = 0;
}
```

- Simple explanation:
  - If lock is free, test&set reads 0 and sets value=1, so lock is now busy. It returns 0 so while exits.
  - If lock is busy, test&set reads 1 and sets value=1 (no change). It returns 1, so while loop continues
  - When we set value = 0, someone else can get lock
- Better: test&test&set
- Busy-Waiting: thread consumes cycles while waiting

Better Locks using test&set

- Can we build test&set locks without busy-waiting?
  - Can't entirely, but can minimize!
  - Idea: only busy-wait to atomically check lock value

```
int guard = 0;
int value = FREE;
```

```
Acquire() {
  // Short busy-wait time
  while (test&set(guard));
  if (value == BUSY) {
    put thread on wait queue;
    go to sleep() & guard = 0;
  } else {
    value = BUSY;
    guard = 0;
  }
}
```

```
Release() {
  // Short busy-wait time
  while (test&set(guard));
  if anyone on wait queue {
    take thread off wait queue
    Place on ready queue;
  } else {
    value = FREE;
  }
  guard = 0;
```

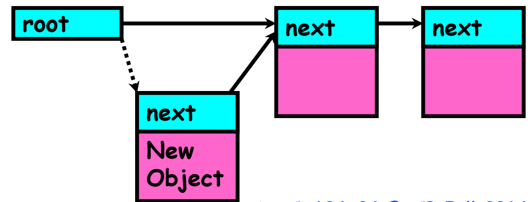
- Note: sleep has to be sure to reset the guard variable
  - Why can't we do it just before or just after the sleep?

Using of Compare&Swap for queues

```
• compare&swap (&address, reg1, reg2) { /* 68000 */
    if (reg1 == M[address]) {
      M[address] = reg2;
      return success;
    } else {
      return failure;
    }
  }
```

Here is an atomic add to linked-list function:

```
addToQueue(&object) {
    do {                    // repeat until no conflict
      ld r1, M[root]        // Get ptr to current head
      st r1, M[object]      // Save link in new object
    } until (compare&swap(&root,r1,object));
}
```



Portable Atomic operations in Linux

- Linux provides `atomic_t` for declaring an atomic integer
  - Also, `atomic64_t` for declaring atomic 64-bit variant
  - Not necessarily same as a regular integer!
    - » Originally on SPARC, `atomic_t` ⇒ only 24 of 32 bits usable
- Example usage:

```
atomic_t v;         /* define v */
atomic_t u = ATOMIC_INIT(0); /* define and init u=0 */

atomic_set(&v, 4);  /* v=4 (atomically) */
atomic_add(2, &v);  /* v = v + 2 (atomically) */
atomic_inc(&v);     /* v = v + 1 (atomically) */

int final = atomic_read(&v);  /* final == 7 */
```

- Some operations (see Love, Ch 10, Table 10.1/10.2):

```
atomic_inc()/atomic_dec()   /* Atomically inc/dec */
atomic_add()/atomic_sub()   /* Atomically add/sub */
int atomic_dec_and_test()   /* Sub 1.  True if 0 */
int atomic_inc_return()     /* Add 1, return result */
```


Portable bit operations in Linux

- Atomic bitwise operations operate on regular Ints.
  - For example, to set nr^{th} bit, starting from addr:

```
void set_bit(int nr, void *addr);
```

  - Atomicity ensures that bit transitions are always seen atomically – regardless of competing concurrency
    - » When bit is set and cleared – actually reflected as stores
    - » When two different bits set – end up with two bits set, rather than one set operation erasing result of another
- Some operations (see Love, Ch 10, Table 10.3):

```
void set_bit()             /* Atomically set bit */
void clear_bit()           /* Atomically clear bit */
void change__bit()         /* Atomically toggle bit */
int test_and_set_bit()     /* set bit, return previous*/
int test_and_clear_bit()   /* clear bit, return prev */
int test_and_change_bit()/* toggle bit, return prev */
int test_bit()             /* Return value of bit*/
```

Portable Locking constructs in Linux

- Linux provides lots of synchronization constructs
  - We will highlight them throughout the term
- Example: Spin Lock support: Not recursive!

```
#include <linux/spinlock.h>
DEFINE_SPINLOCK(my_lock);

spin_lock(&my_lock);
/* Critical section … */
spin_unlock(&my_lock);
```

- Disable interrupts and grab lock (while saving and restoring state in case interrupts already disabled):

```
DEFINE_SPINLOCK(my_lock);
unsigned long flags;

spin_lock_irqsave(&my_lock,flags);
/* Critical section … */
spin_unlock_irqrestore(&my_lock);
```


Locks (Mutexes) in POSIX Threads

- To create a mutex:

```
#include <pthread.h>
pthread_mutex_t amutex = PTHREAD_MUTEX_INITIALIZER;
pthread_mutex_init(&amutex, NULL);
```

- To use it:

```
int pthread_mutex_lock(amutex);
int pthread_mutex_unlock(amutex);
```

- To deallocate a mutex

```
int pthread_mutex_destroy(pthread_mutex_t *mutex);
```


Summary

- pTreads: POSIX interface for threading
  - Operations to create, destroy, and synchronize threads
  - Shared-memory model
- Important concept: Atomic Operations
  - An operation that runs to completion or not at all
  - These are the primitives on which to construct various synchronization primitives
- Talked about hardware atomicity primitives:
  - Disabling of Interrupts, test&set, swap, comp&swap, load-linked/store conditional
- Showed several constructions of Locks
  - Must be very careful not to waste/tie up machine resources
    - » Shouldn't disable interrupts for long
    - » Shouldn't spin wait for long
  - Key idea: Separate lock variable, use hardware mechanisms to protect modifications of that variable
- Started talking abut higher level constructs that are harder to "screw up"