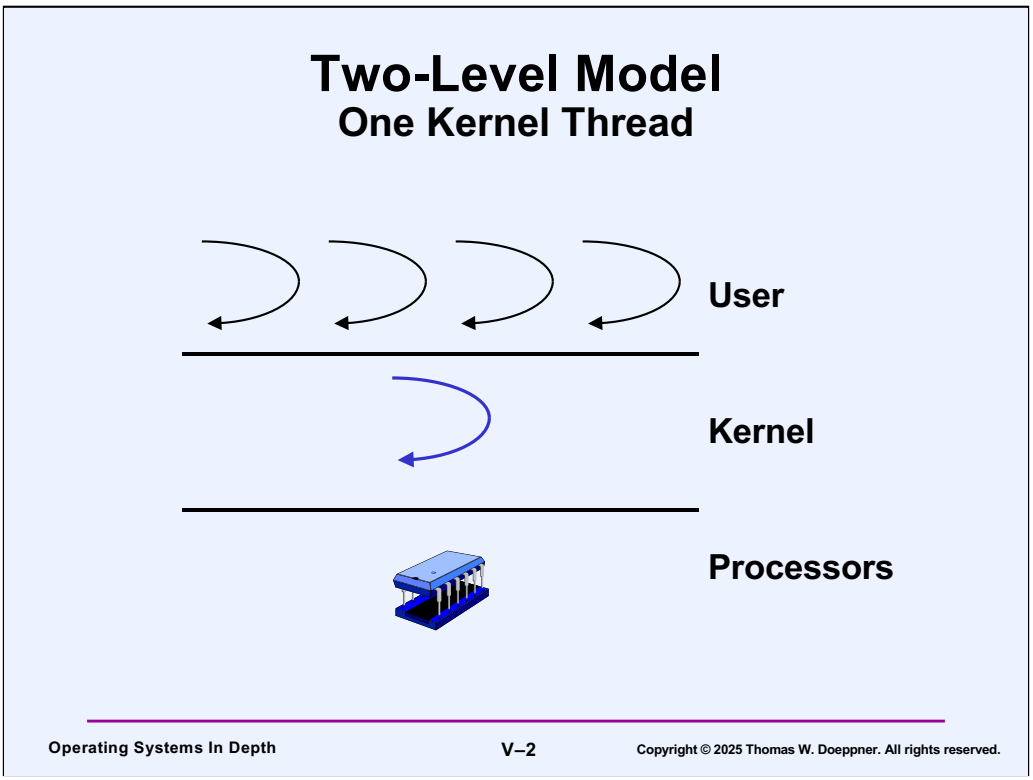
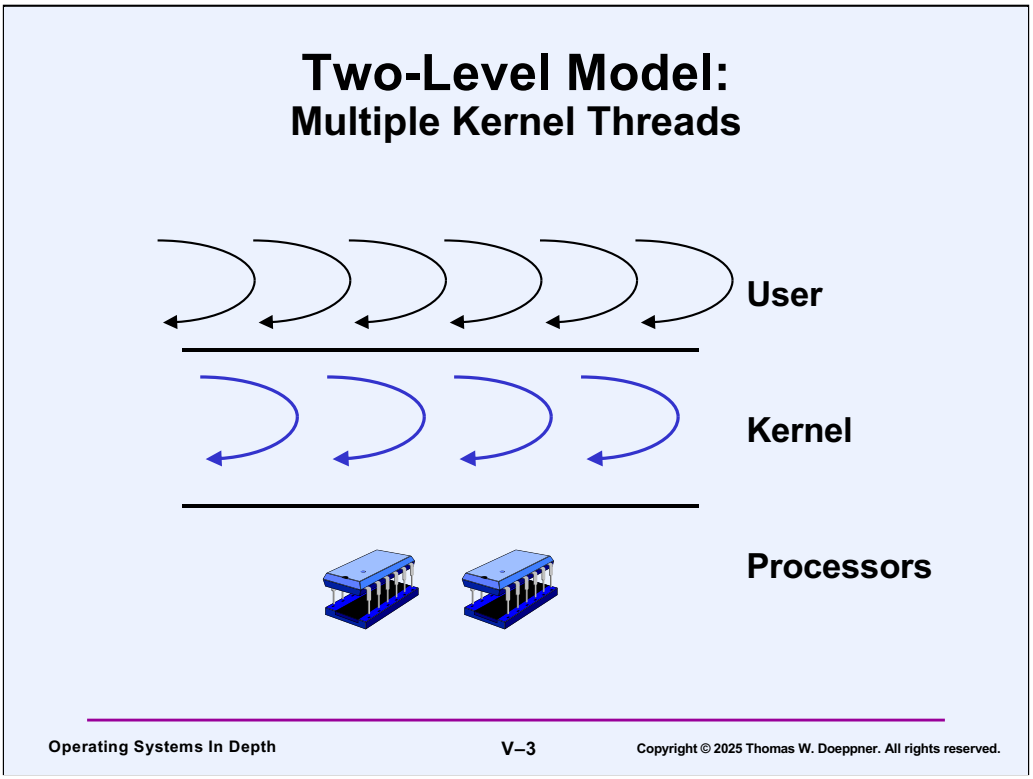
Implementing Threads 4



Another approach, the two-level model, is to represent the two contexts as separate types of threads: user threads and kernel threads. Kernel threads become "virtual activities" upon which user threads are scheduled. Thus two schedulers are used: kernel threads are multiplexed on activities by a kernel scheduler; user threads are multiplexed on kernel threads by a user-level scheduler. An extreme case of this model is to use only a single kernel thread per process (perhaps because this is all the operating system supports). There are two obvious disadvantages of this approach, both resulting from the restriction of a single kernel thread per process: only one activity can be used at a time (thus a single process cannot take advantage of a multiprocessor) and if the kernel thread is blocked (e.g., as part of an I/O operation), no user thread can run.

This is the approach used in the **uthreads assignment**.



A more elaborate use of the two-level model is to allow multiple kernel threads per process. This deals with both the disadvantages described above and is the basis of the Solaris implementation of threading. It has some performance issues; in addition, the notion of multiplexing user threads onto kernel threads is very different from the notion of multiplexing threads onto activities—there is no direct control over when a chore is actually run by an activity. From an application's perspective, it is sometimes desired to have direct control over which chores are currently being run.

This is the model we employ for **mthreads**.

Quiz 1

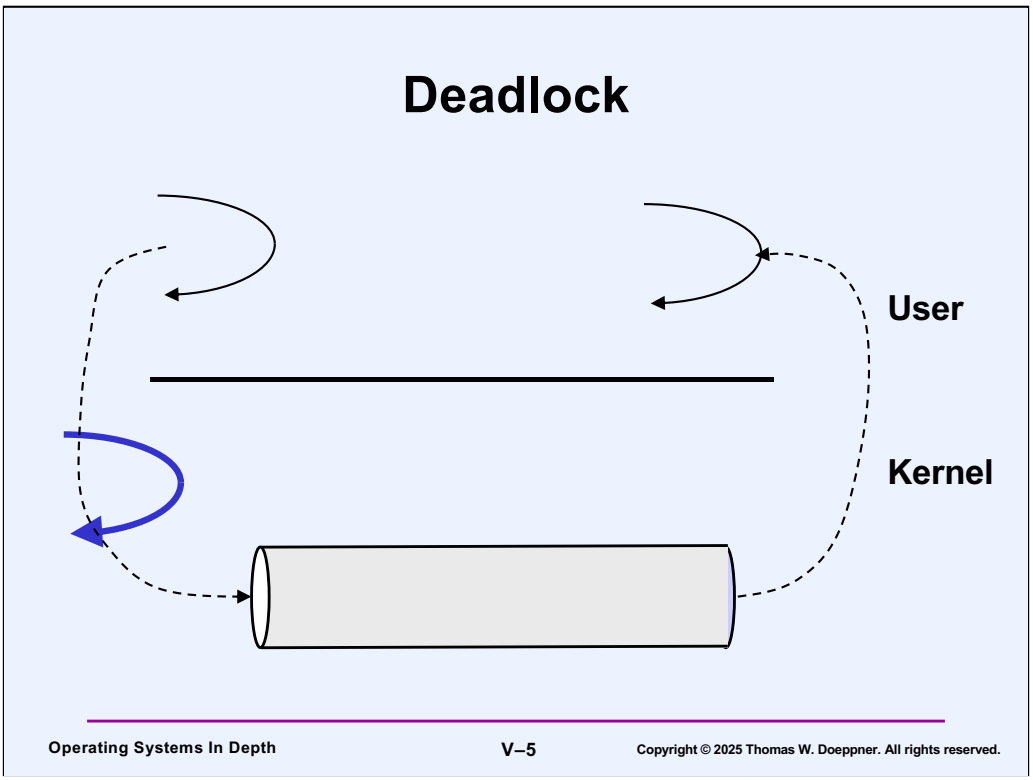
One kernel thread for each user thread is clearly a sufficient number of kernel threads in the two-level model. Is it necessary for maximum concurrency?

a) **there are no situations in which that number of threads is necessary, as long as there are at least as many kernel threads as processors**

b) **there must always be that number of kernel threads for the two-level model to work well**

c) **there are situations in which that number is necessary, but they occur rarely**

For the two-level model to "work well", we want to achieve the maximum concurrent execution that's possible. Thus if we have n threads that are ready to run, and n processors, then all n threads should be able to run.

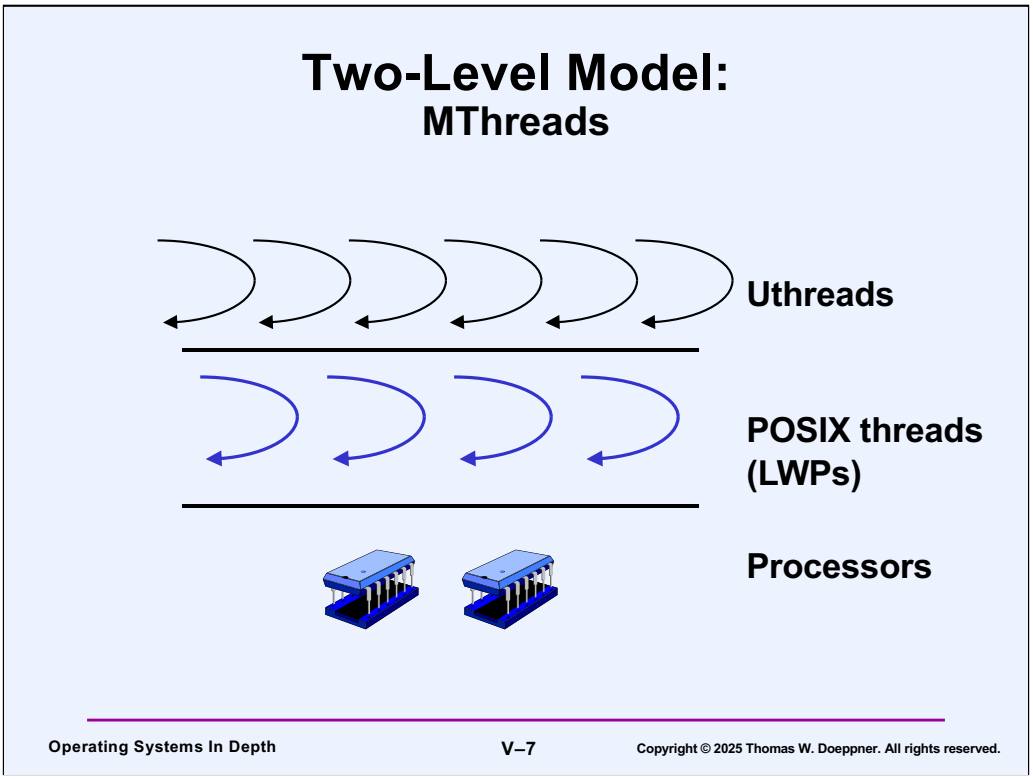


One negative aspect of the two-level model is that its use might induce deadlock. For example, suppose we have two user threads and one kernel thread. One thread is writing into a pipe (using the write system call). However, at the moment the pipe is full. The the call to write blocks. The other user thread is ready to do a read system call on the pipe, thus making it not full and unblocking the first thread, but since there's only one kernel thread and it's blocked (since it's running the first user thread), the second user thread can't read from the pipe and thus we're stuck: deadlocked.

The solution would be to introduce an additional kernel thread if such a situation happens. This was done in the Solaris implementation of the two-level thread model: if all kernel threads in a process are blocked, a new one was automatically created.

MThreads

- Two-level threads implementation of Uthreads
  - kernel-supported threads are POSIX threads
  - user threads based on your implementation of Uthreads
- Effectively a multiprocessor implementation
  - use POSIX mutexes rather than spin locks
  - use POSIX condition variables rather than the idle loop



LWP stands for lightweight process and was the term used by Sun Microsystems in their implementation of this model.

Synchronizing LWPs

```
uthread_switch(...) {
   uthread_mtx_lock(&runq_mtx)
   volatile int first = 1;
   getcontext(&ut_curthr->ut_ctx);
   if (!first) {
      ...
   }
   setcontext(&curlwp->lwp_ctx);
}
lwp_switch() {
   ...
   ut_curthr = top_priority_thread(&runq);
   uthread_mtx_unlock(&runq_mtx);
   setcontext(&ut_curthr->ut_ctx);
   ...
}
```


In this example, a simplified version of uthread_switch and lwp_switch from the mthreads assignment, a uthread calls uthread_switch, so that its LWP may run some other uthread. The uthread's context is saved in the getcontext call in uthread_switch, then the setcontext call causes the lwp to resume its execution within lwp_switch. It then finds the top-priority thread in the runq, and switches to its context.

A problem is that other LWPs might also be accessing the runq. Thus we need to synchronize access to it. We might try using uthreads mutexes, but if the mutex is already locked, the thread would put itself on the mutex's wait queue, then call uthread_switch to give the LWP to another uthread. But then it tries to lock the mutex, which, we know, is already locked, so it again tries to call uthread_mutex, ...

Synchronizing LWPs (2)

```
uthread_switch(...) {
   spin_lock(&runq_mtx)
   volatile int first = 1;
   getcontext(&ut_curthr->ut_ctx);
   if (!first) {
      ...
   }
   setcontext(&curlwp->lwp_ctx);
}
lwp_switch() {
   ...
   ut_curthr = top_priority_thread(&runq);
   spin_unlock(&runq_mtx);
   setcontext(&ut_curthr->ut_ctx);
   ...
}
```


Rather than using uthread mutexes, we might use spinlocks. These don't involve calls to uthread_switch, and thus they would work here.

But since our LWPs are effectively virtual processors, we prefer not to use spin locks, but POSIX mutexes.

Synchronizing LWPs (3)

```
uthread_switch(...) {
   pthread_mutex_lock(&runq_mtx)
   volatile int first = 1;
   getcontext(&ut_curthr->ut_ctx);
   if (!first) {
      ...
   }
   setcontext(&curlwp->lwp_ctx);
}
lwp_switch() {
   ...
   ut_curthr = top_priority_thread(&runq);
   pthread_mutex_unlock(&runq_mtx);
   setcontext(&ut_curthr->ut_ctx);
   ...
}
```


Here we are using a POSIX mutex, which is what is used instead of spinlocks in the mthreads assignment.

POSIX Mutexes and MThreads

- POSIX mutexes used to synchronize activity among LWPs
- Problem case
  - uthread (running on LWP) locks mutex
  - clock interrupt occurs, uthread yields LWP to another uthread
  - that uthread (running on same LWP) locks same mutex
  - deadlock: LWP attempting to lock mutex it currently has locked
- Solution
  - mask interrupts while thread has mutex locked

Example

```
void uthread_wake(uthread_t *uthr) {
    pthread_mutex_lock(&runq_mtx);
    ...
    // wake up thread, put it on runq
    ...
    pthread_mutex_unlock(&runq_mtx);
}
```


This is another example of the synchronization required when accessing the runq. Here we have a portion of the function that wakes up a thread.

This code is executed by a uthread, say t1, that is running on a particular LWP. It locks the runq mutex to make sure that no other LWP is currently using the runq.

What might happen is that t1 is forced to yield to another thread, t2, while it is in the code that's modifying the runq (and thus the mutex is locked). But t2 also calls uthread_wake, and attempts to lock runq_mtx. But since its LWP already has the mutex locked, we have a deadlock.

Example: Fixed

```
void uthread_wake(uthread_t *uthr) {
    uthread_noprempt_on();
    pthread_mutex_lock(&runq_mtx);

    ...

    // wake up thread, put it on runq

    ...

    pthread_mutex_unlock(&runq_mtx);
    uthread_nopreempt_off();
}
```


Thus we must turn off preemption while holding a POSIX mutex.

Thread-Local Storage in Mthreads

- `__thread thread_t *ut_curthr;`
  - **reference to the current uthread**
- `__thread lwp_t *curlwp`
  - **reference to the current LWP (POSIX thread)**

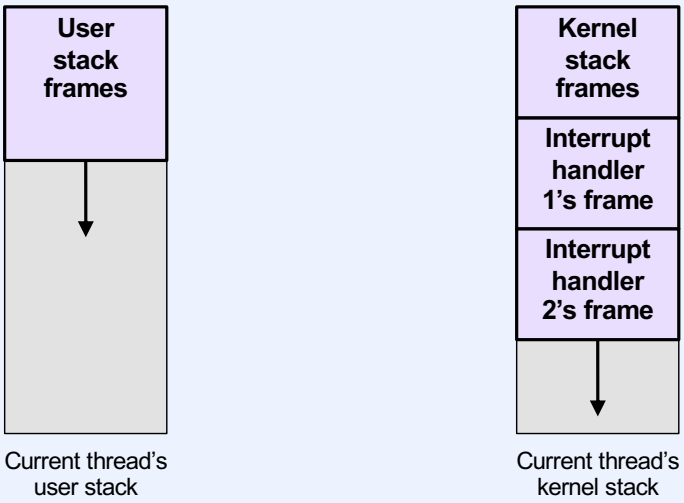
- **Thread-Local Storage accesses are not async-signal safe!**
- **Must turn off preemption while using TLS**
  - **otherwise thead could be preempted and later resumed on another LWP**
  - **TLS pointer refers to the wrong item!**

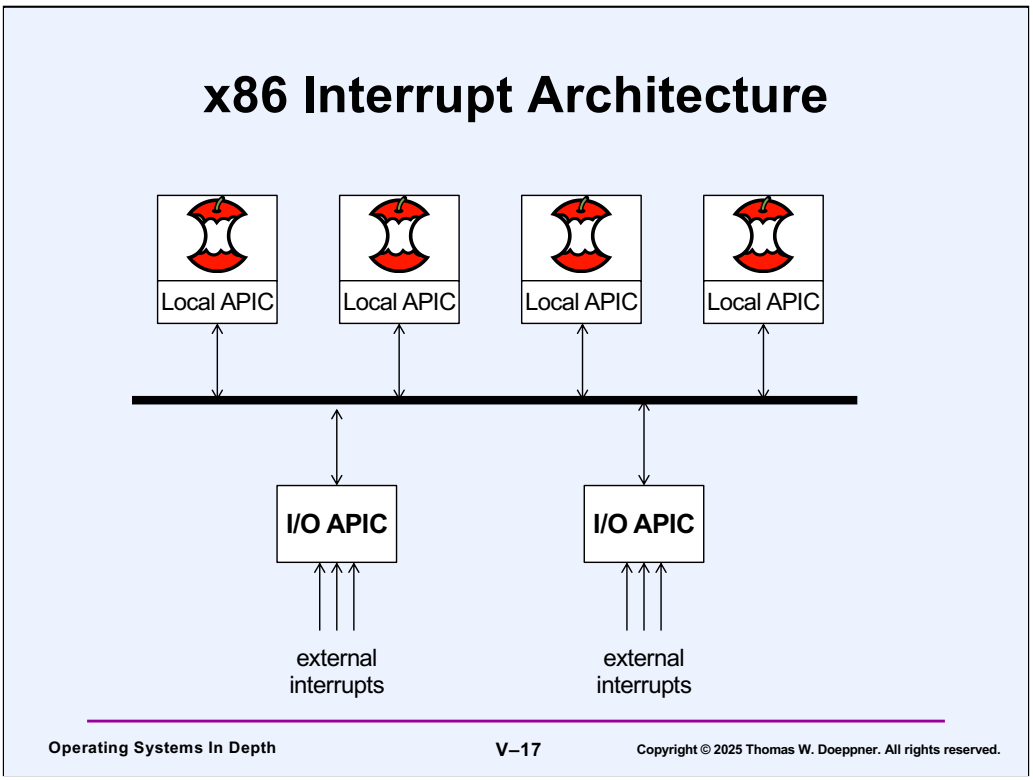
Thread-local storage (TLS) is implemented as part of POSIX threads. We use it in mthreads for references to the current **uthread** and the current **LWP**. Unfortunately, referencing TLS is not async-signal safe. Since you will be using TLS within the signal handler for SIGVTALRM, make sure that you mask SIGVTALRM when using TLS.

The primary issue is that TLS is associated with the POSIX thread and not with the uthread. Thus, while executing a particular uthread, its underlying POSIX thread might change. Suppose that a thread has the address of ut_curthr in a register (what's in the register is not the address of the uthread_t, but the address of the TLS variable (ut_curthr) that contains the address of the uthread_t). A SIGVTALARM now occurs. The thread yields its LWP to another uthread, and eventually resumes execution running on another LWP. The location uthr_cur (of the new LWP) points to the thread's uthread_t, but the address that was put in the register points to the uthr_cur of the other LWP. Thus if the thread now uses the address in this register to get to its uthread_t, it gets to the wrong one – that of some other thread that's now running on the original LWP.

Interrupts, Etc.

Interrupts





This slide shows, roughly, the I/O architecture of multicore x86 systems. For details see "Intel 64 and IA-32 Architectures Software Developer's Manual," Volume 3A, Chapter 10, obtainable from http://www.intel.com/content/www/us/en/processors/architectures-software-developer-manuals.html. "APIC" stands for "advanced programmable interrupt controller." External devices issue interrupts through an I/O APIC. These, based on criteria such as the current interrupt-priority level of the various processor cores, direct an interrupt to one of them. The local APIC determines when to interrupt the local core. It also contains an interval timer that's available to software.

Dealing with Interrupts

- Interrupt comes from external source
- Must execute code to handle it
- Which stack?
  - use current (kernel) stack

    or
  - use separate stack

The choice between the current stack and the separate stack is an architectural decision, i.e., made by the designers of the hardware, not by the designers of the operating system.

Use the Current (Kernel) Stack

- Borrowed from the current thread
  - requires all threads to have sufficiently large kernel stacks
- Interrupted thread may not concurrently execute
  - would corrupt interrupt handler's stack frames
- Interrupt handler should not block
  - may be waiting on interrupted thread
    - deadlock
  - threads must mask interrupts to protect data structures shared with interrupt handlers

Recall that each thread has two stacks: one for use in user mode, the other for use in kernel mode (when it's executing system calls). The stack that's borrowed by the interrupt handler is the thread's kernel stack.

Hierarchical Interrupt Masking

- Interrupts assigned priorities 1 through n
  - current interrupt priority level *i*
    - interrupts with priorities 1 through *i* masked
  - IPL set to i
    - when interrupt of priority *i* is handled
    - when IPL set explicitly to *i*
- Raising the IPL
  - protects data structures
    - good!
  - delays response to lower priority interrupts
    - bad!

Separate Interrupt Stack

- **Single stack used strictly by interrupt handlers**
  - **hardware saves current register state on interrupt stack**
  - **no interrupt-handling space required for kernel stacks**
    - **all can be smaller**
  - **in principle, interrupted thread may continue to execute**
    - **won't corrupt interrupt handler's frames**

Quiz 2

Is interrupt-masking still required?

a) yes
b) no

This approach was used both by Unix and VMS on the Digital VAX-11 architecture.

(Non-Quiz) Questions

1) Is interrupt masking still required? *Done*
2) May interrupt handler block?
3) Does it work on multiprocessors?

Answers

1) Masking is still required
   - if nothing else, to protect data structures shared by multiple interrupt handlers
   - to prevent deadlock if spin locks are used
2) Interrupt handlers should not block
   - would have to block with raised IPL
   - results in lengthy time period with interrupts masked
     - delayed response
3) Yes, but each processor has its own interrupt stack (if separate interrupt stacks are part of the architecture)

If an interrupt handler were allowed to block, we would need to make certain that all subsequent interrupts that occur (on the interrupt stack) until this one completes must themselves complete before this one is resumed. Otherwise, when the blocked interrupt handler resumes, its new stack frames might overwrite those of the subsequent interrupt handlers. We could obtain this guarantee by making sure that the IPL is never lowered below that of the blocked interrupt handler.

Interrupt Threads

- Give each interrupt instance its own stack
  - handlers effectively execute as separate threads
  - interrupted thread continues to run
  - but interrupts remain masked while interrupt thread is processing interrupt

Interrupt threads were used in the Solaris operating system.

Effect of Interrupts

- Normally don't directly affect current thread
  - thread is interrupted
  - interrupt is dealt with
  - thread is resumed
- I/O-completion interrupts
  - may result in waking up higher-priority thread (that was waiting for the I/O completion)
- Clock interrupts
  - may trigger end of time slice

Synchronization and Interrupts

- Non-preemptive kernels
  - threads running in privileged mode yield the processor only voluntarily
  - involuntary thread switches happen only to threads in user mode
    - end of time slice
    - higher-priority thread is made runnable
  - inter-thread in-kernel synchronization is easy
- Preemptive kernels
  - threads running in privileged mode may be forced to yield the processor
  - inter-thread sync in kernel is not easy

Non-Preemptive Kernel Sync.

```
                    int X = 0;

void AccessXThread() {              void AccessXInterrupt() {
  int oldIPL;                         …
  oldIPL = setIPL(IXLevel);           X = X+1;
  X = X+1;                            …
  setIPL(oldIPL);                   }
}
```


We assume the threads are running on a single-processor system. AccessXThread is called by a thread running in the kernel. Potentially a number of threads could be calling it. "IXLevel" is the interrupt priority level of the "X" device, which causes AccessXInterrupt. Note that the interrupt priority level of clock interrupts would be higher than this.

The function **setIPL** sets the interrupt priority level to its argument and returns the previous interrupt priority level.

Quiz 2

```
int X = 0;

void AccessXThread() {
  int oldIPL;
  oldIPL = setIPL(IXLevel);
  X = X+1;
  setIPL(oldIPL);
}

void AccessXInterrupt() {
  …
  X = X+1;
  …
}
```

This works

a) only on a single-core system with a non-preemptive kernel
b) also on a single-core system with a preemptive kernel
c) also on multi-core systems

By ”work” we mean that the statements “`X = X+1`” are atomic.

Disk I/O

```
int disk_write(…) {
  …
  startIO(); // start disk operation
  …
  enqueue(disk_waitq, CurrentThread);
  thread_switch();
    // wait for disk operation to
    // complete
  …
}
```

```
void disk_intr(…) {
  thread_t *thread;
  …
  // handle disk interrupt
  …
  thread = dequeue(disk_waitq);
  if (thread != 0) {
    enqueue(RunQueue, thread);
    // wakeup waiting thread
  }
  …
}
```


This code doesn't work!

Improved Disk I/O

```
int disk_write(…) {
  …
  oldIPL = setIPL(diskIPL);
  startIO();      // start disk operation
  …
  enqueue(disk_waitq, CurrentThread);
  thread_switch();
      // wait for disk operation to complete
  setIPL(oldIPL);
  …
}
```


More is needed. (See next slide.)

Modified *thread_switch*

```
void thread_switch() {
  thread_t *OldThread;
  int oldIPL;
  oldIPL = setIPL(HIGH_IPL);
    // protect access to RunQueue by masking all interrupts
  while(queue_empty(RunQueue)) {
    // repeatedly allow interrupts, then check RunQueue
    setIPL(0);  // IPL == 0 means no interrupts are masked
    setIPL(HIGH_IPL);
  }
  // We found a runnable thread
  OldThread = CurrentThread;
  CurrentThread = dequeue(RunQueue);
  swapcontext(OldThread->context, CurrentThread->context);
  setIPL(oldIPL);
}
```


Note that the caller's IPL is saved in a local variable (on the stack) and thus becomes part of the caller's context (and is restored on return from **thread_switch**, in its last statement).

Note that this code doesn't work on multiprocessor systems (it assumes there is just one processor).

Preemptive Kernels on MP

- What's different?
- A thread accesses a shared data structure:
  1. it might be *interrupted* by an interrupt handler (running on its processor) that accesses the same data structure
  2. *another thread* running on another processor might access the same data structure
  3. it might be forced to *give up its processor* to another thread, either because its time slice has expired or it has been preempted by a higher-priority thread
  4. an *interrupt handler* running on *another processor* might access the same data structure

Solution?

```
int X = 0;
SpinLock_t L = UNLOCKED;
```

```
void AccessXThread() {
  SpinLock(&L);
  X = X+1;
  SpinUnlock(&L);
}
```

```
void AccessXInterrupt() {
    …
    SpinLock(&L);
    X = X+1;
    SpinUnlock(&L);
    …
}
```

Solution …

```
int X = 0;
SpinLock_t L = UNLOCKED;
```

```
void AccessXThread() {
    MaskInterrupts();
    SpinLock(&L);
    X = X+1;
    SpinUnlock(&L);
    UnMaskInterrupts();
}
```

```
void AccessXInterrupt() {
    …
    SpinLock(&L);
    X = X+1;
    SpinUnlock(&L);
    …
}
```

Quiz 3

We have a single-core system with a preemptible kernel. We're concerned about data structure *X*, which is accessed by kernel threads as well as by the interrupt handler for dev.

a) It's sufficient for threads to mask dev interrupts while accessing *X*

b) It's sufficient for threads to mask all interrupts while accessing *X*

c) In addition to a, threads must lock (blocking) mutexes before accessing *X*

d) c doesn't work. Instead, threads must lock spinlocks before accessing *X*