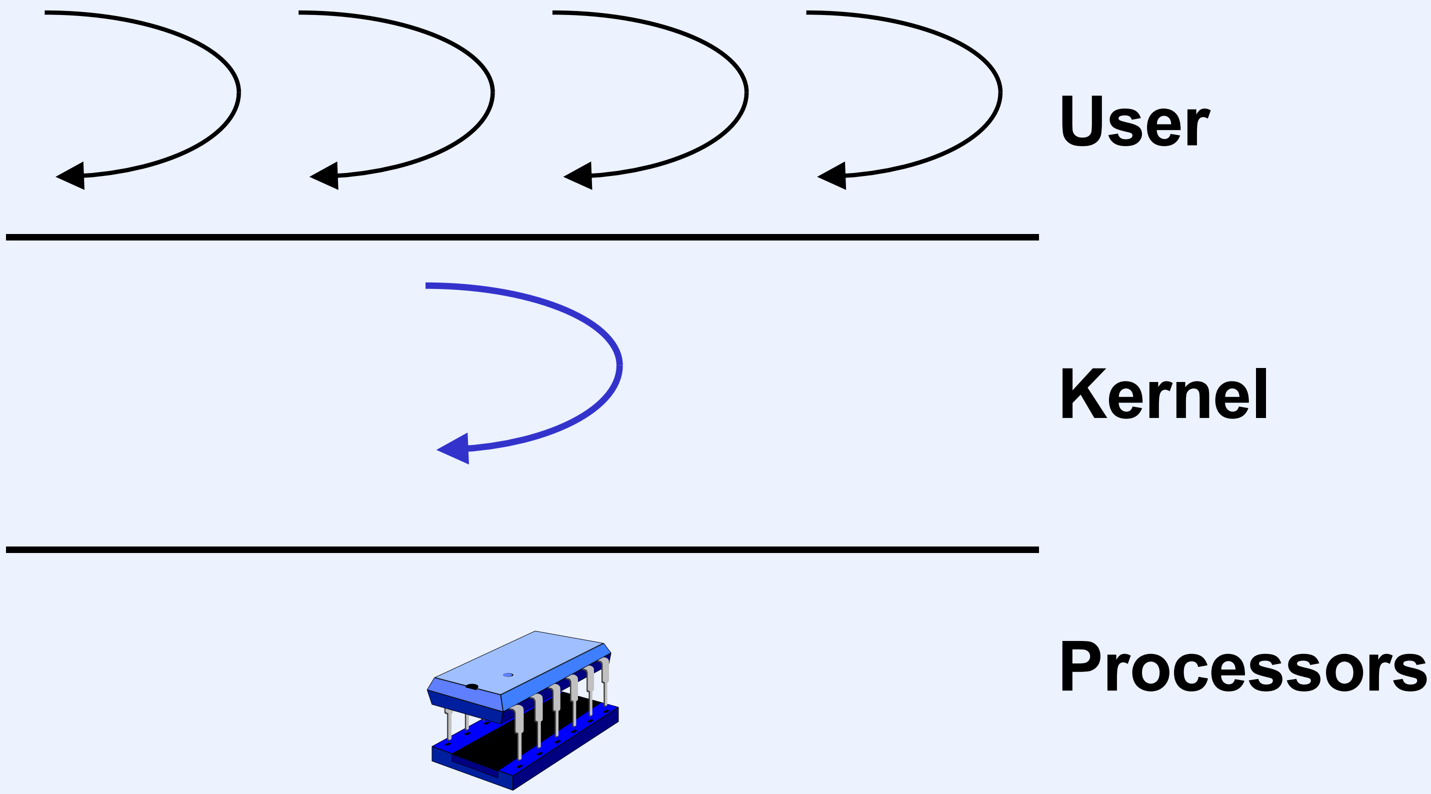
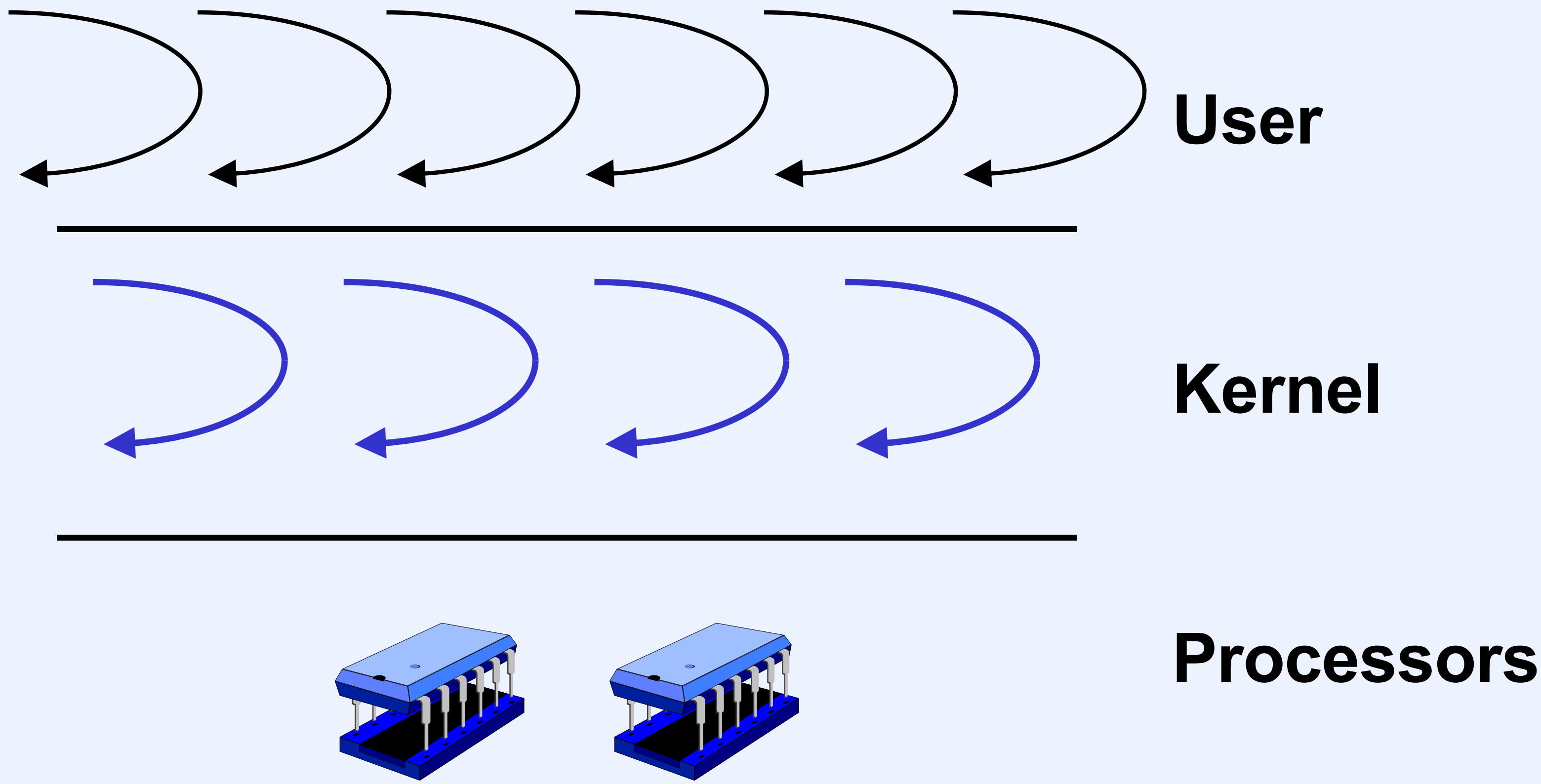
Implementing Threads 4

Two-Level Model
One Kernel Thread



Two-Level Model:
Multiple Kernel Threads



Quiz 1

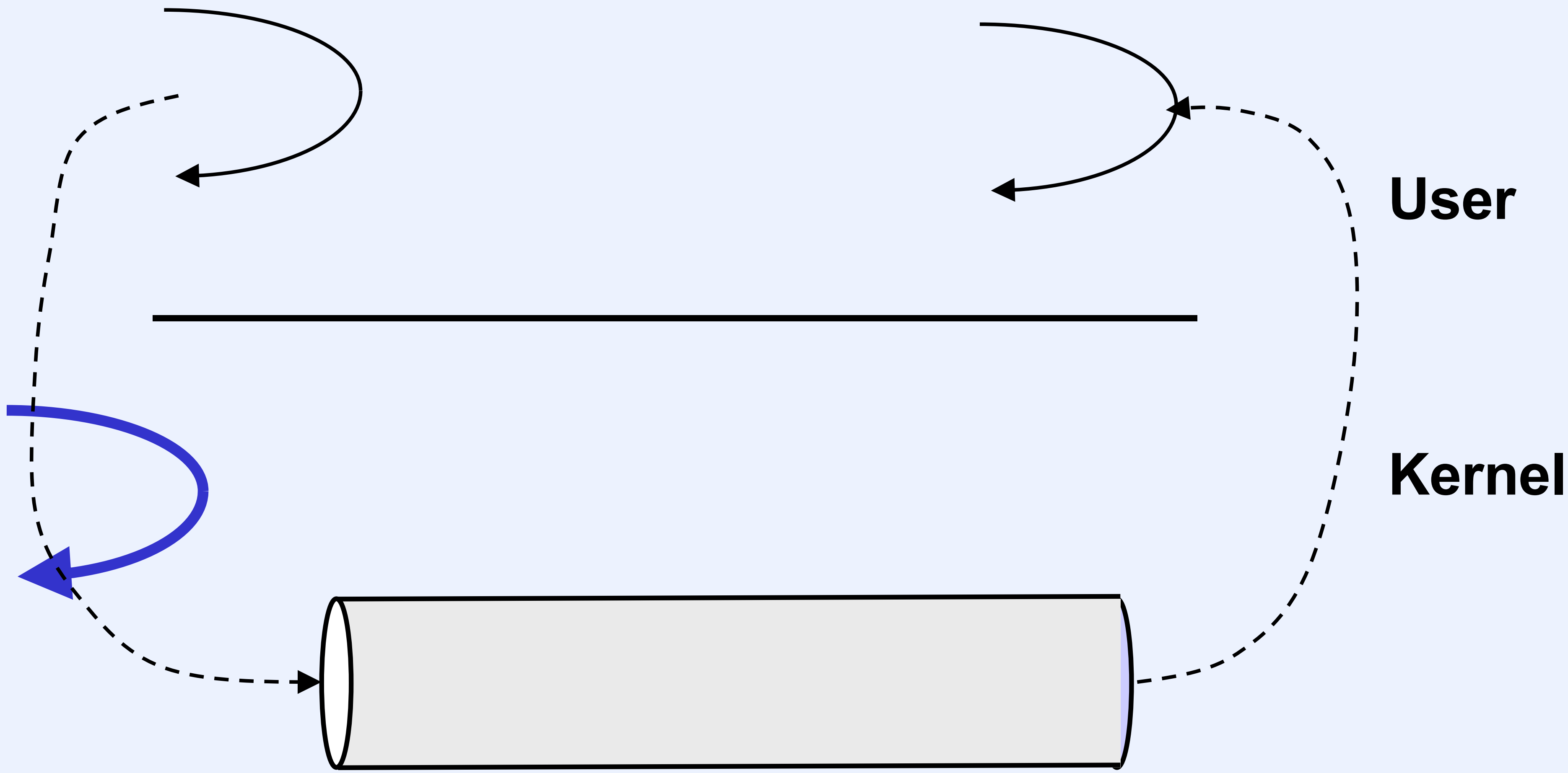
One kernel thread for each user thread is clearly a sufficient number of kernel threads in the two-level model. Is it necessary for maximum concurrency?

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

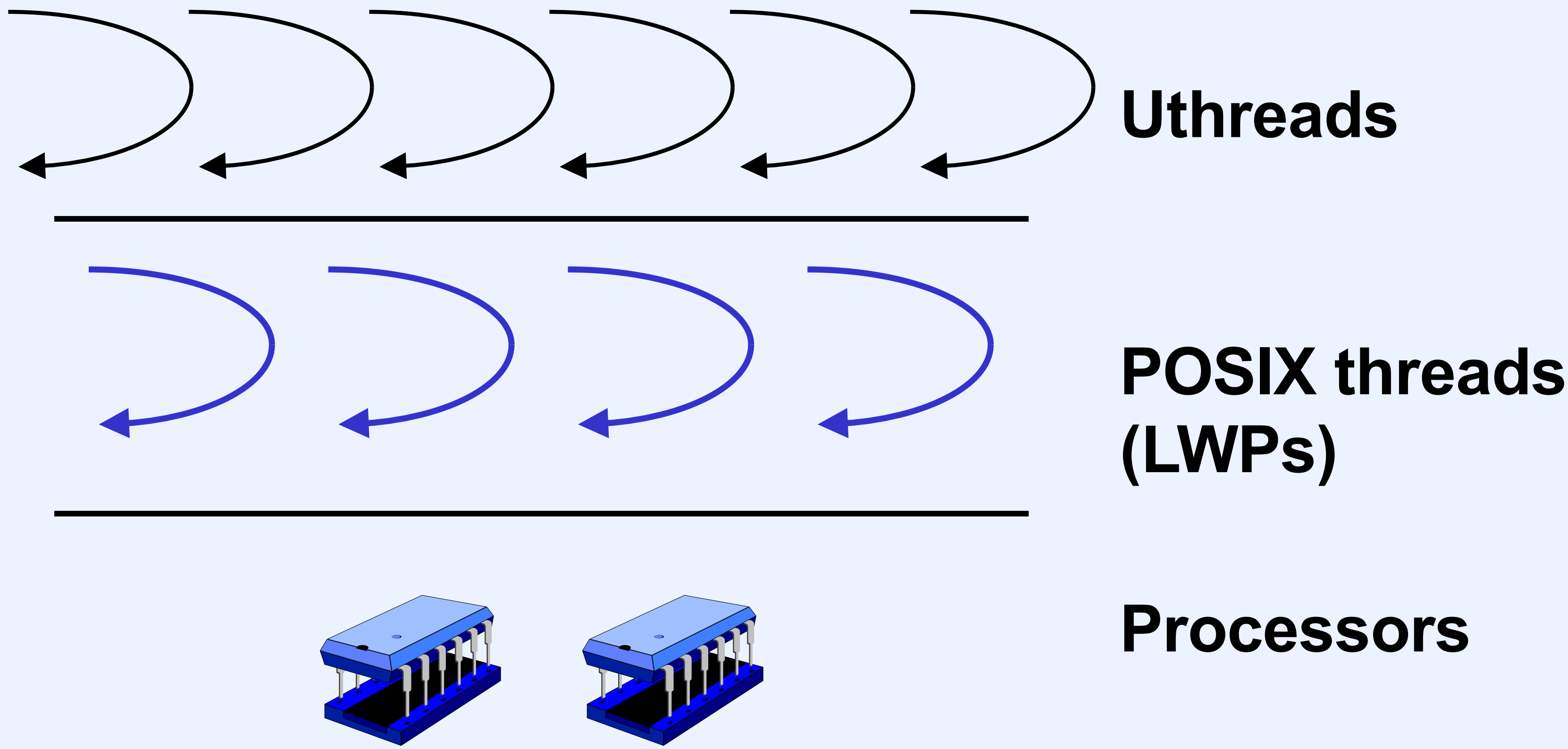
Deadlock



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

Two-Level Model:
MThreads



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);
   ...
}
```

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);
   ...
}
```


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);
   ...
}
```


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);

}
```

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();
}
```

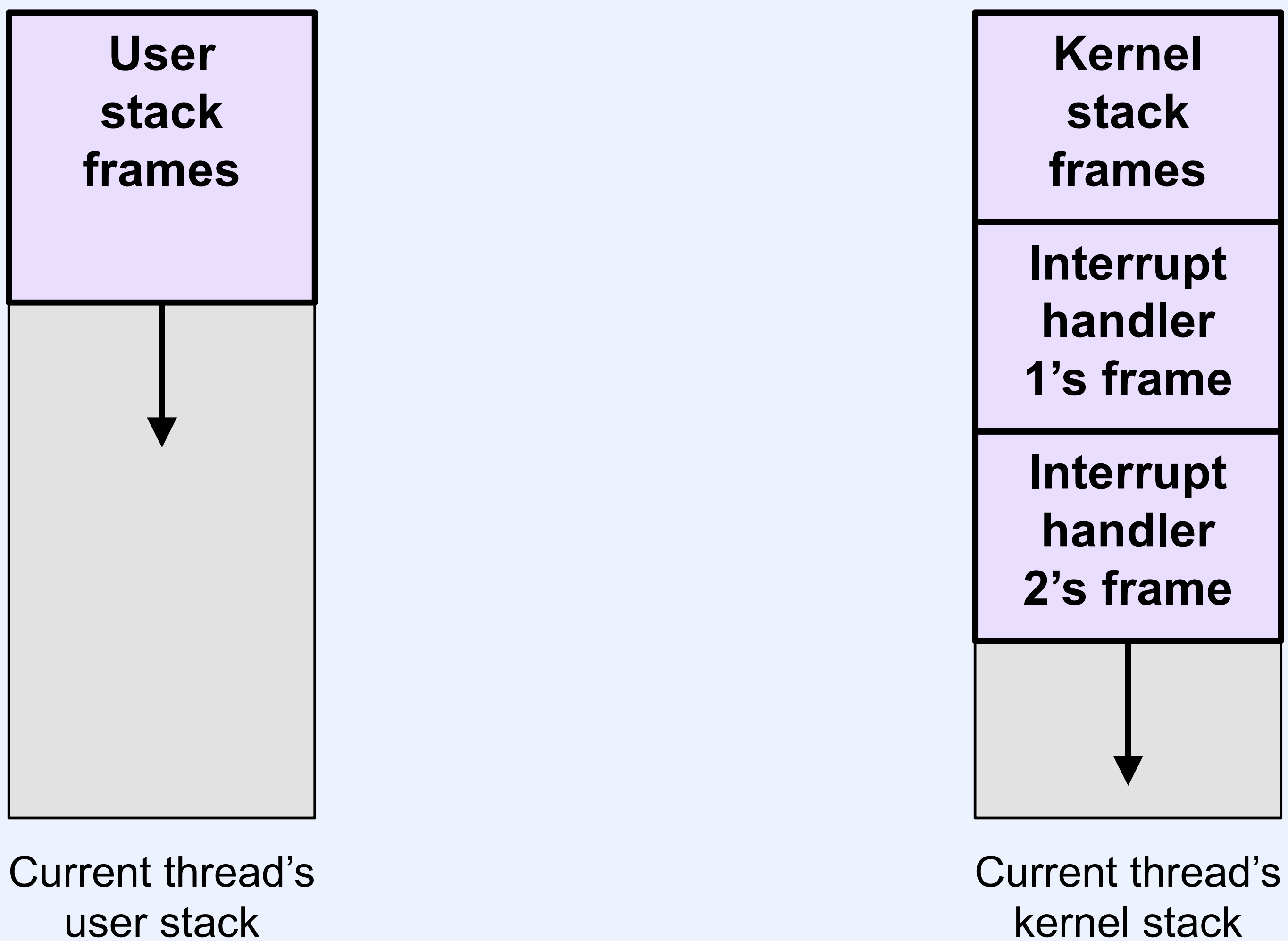

Thread-Local Storage in Mthreads

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

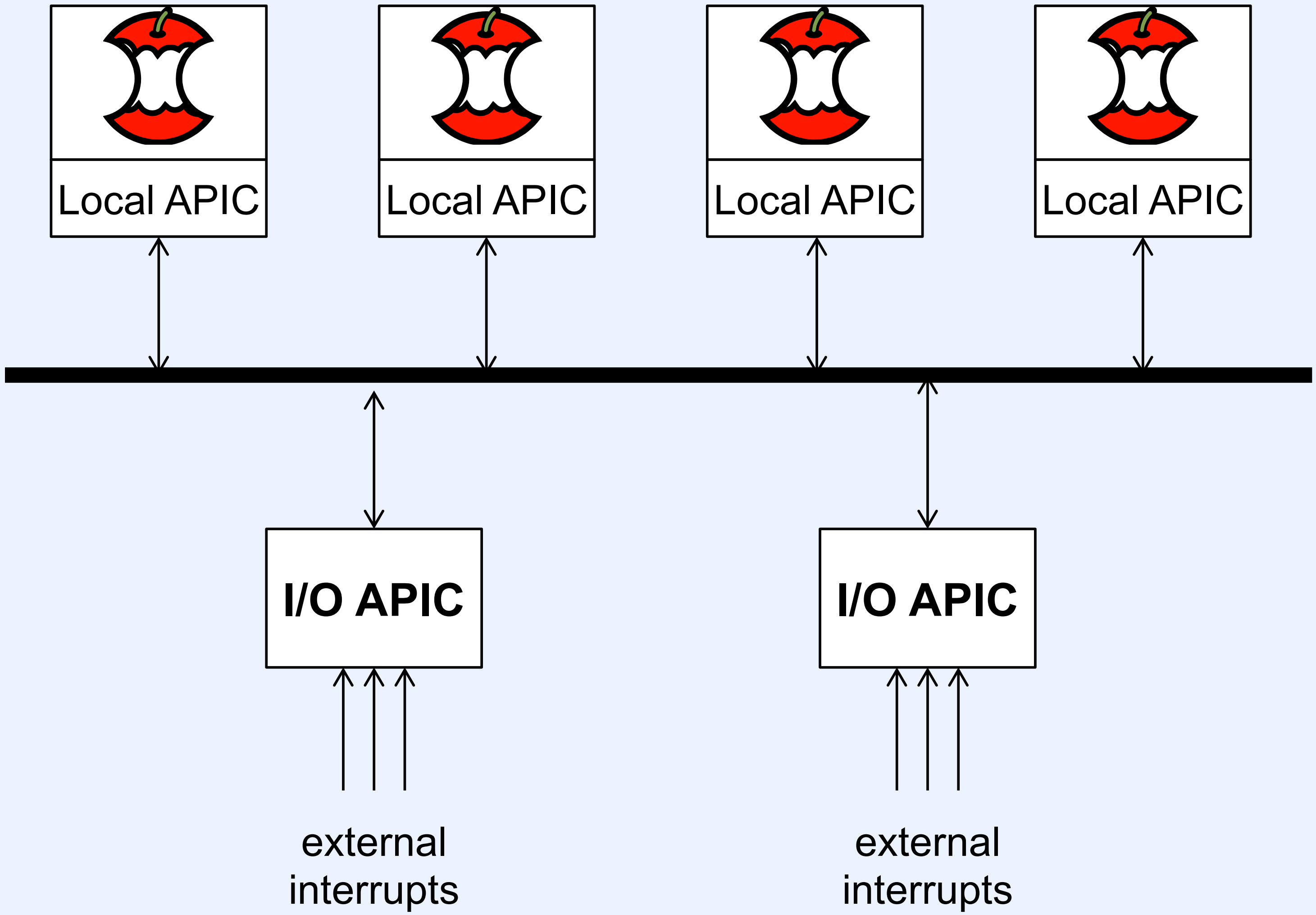
- **Thread-Local Storage accesses are not async-signal safe!**
- **Must turn off preemption while using TLS**
  - **otherwise thead could be preempted and later resumed on another LWP**
  - **TLS pointer refers to the wrong item!**

Interrupts, Etc.

Interrupts


User stack frames
Kernel stack frames
Interrupt handler 1's frame
Interrupt handler 2's frame
Current thread's user stack
Current thread's kernel stack

x86 Interrupt Architecture



Dealing with Interrupts

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

    or
  - use separate stack

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

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

(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)**

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

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

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

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

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
  }
  …
}
```

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);
  …
}
```


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);
}
```


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*