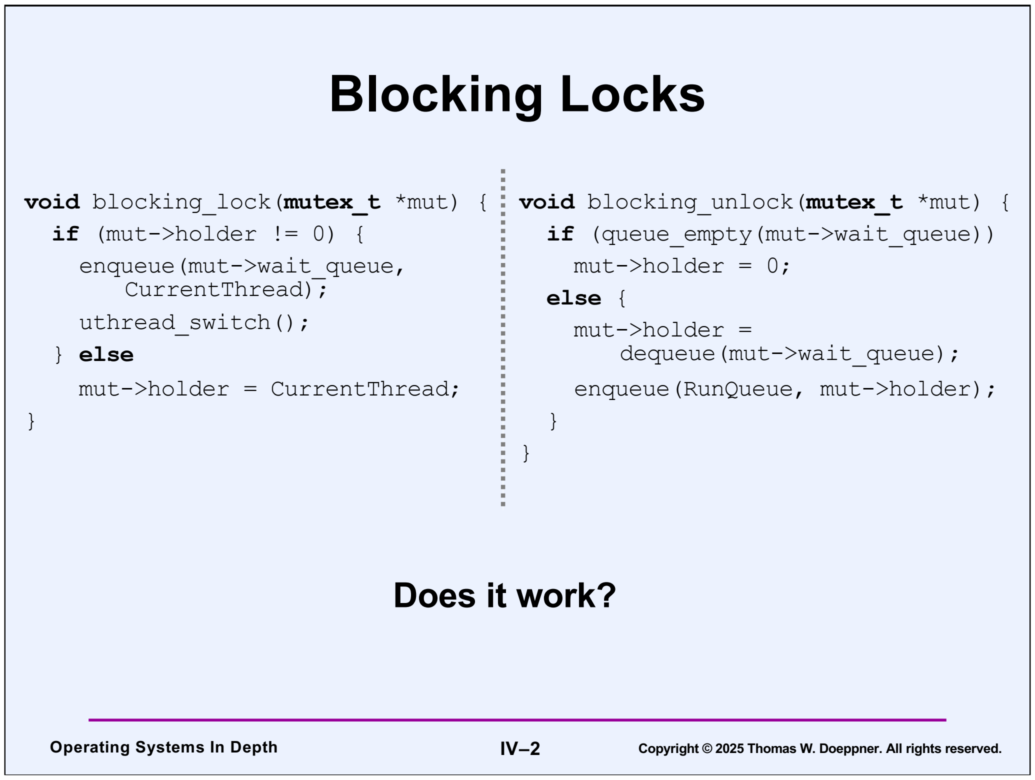
Implementing Threads 3



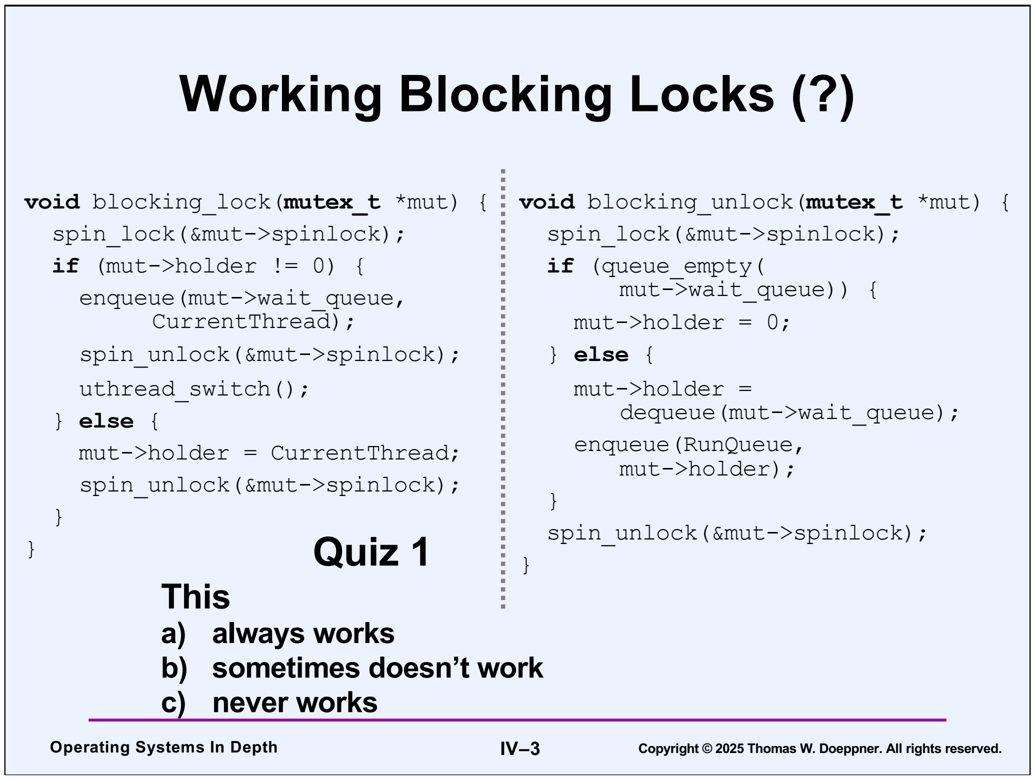
Blocking Locks

```
void blocking_lock(mutex_t *mut) {
  if (mut->holder != 0) {
    enqueue(mut->wait_queue,
        CurrentThread);
    uthread_switch();
  } else
    mut->holder = CurrentThread;
}
```

```
void blocking_unlock(mutex_t *mut) {
  if (queue_empty(mut->wait_queue))
    mut->holder = 0;
  else {
    mut->holder =
        dequeue(mut->wait_queue);
    enqueue(RunQueue, mut->holder);
  }
}
```

Does it work?

There's a problem here: once the thread that's calling **blocking_lock** has put itself on the wait queue, but before it calls **uthread_switch**, it might be released and made runnable (if not running) by a thread on another processor that's calling **blocking_unlock.** Thus it could be running on two processors at once, resulting in total confusion.



Working Blocking Locks (?)

```
void blocking_lock(mutex_t *mut) {
  spin_lock(&mut->spinlock);
  if (mut->holder != 0) {
    enqueue(mut->wait_queue,
          CurrentThread);
    spin_unlock(&mut->spinlock);
    uthread_switch();
  } else {
    mut->holder = CurrentThread;
    spin_unlock(&mut->spinlock);
  }
}
```

```
void blocking_unlock(mutex_t *mut) {
  spin_lock(&mut->spinlock);
  if (queue_empty(
       mut->wait_queue)) {
    mut->holder = 0;
  } else {
    mut->holder =
       dequeue(mut->wait_queue);
    enqueue(RunQueue,
       mut->holder);
  }
  spin_unlock(&mut->spinlock);
}
```

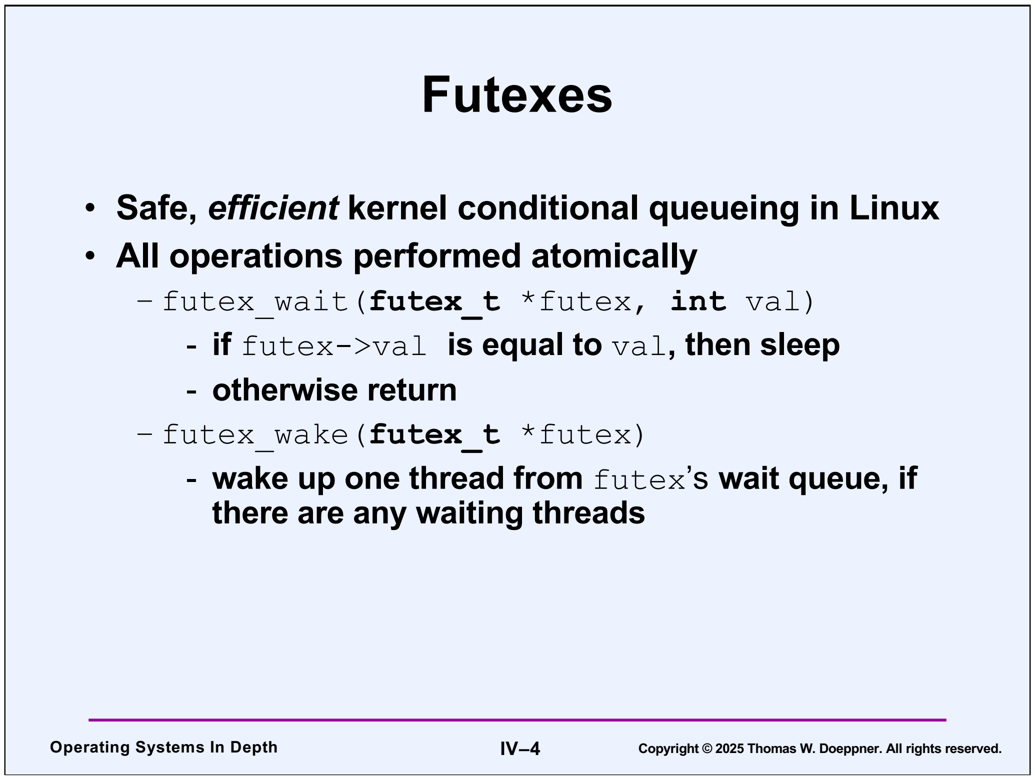
Quiz 1

This

a) always works

b) sometimes doesn't work

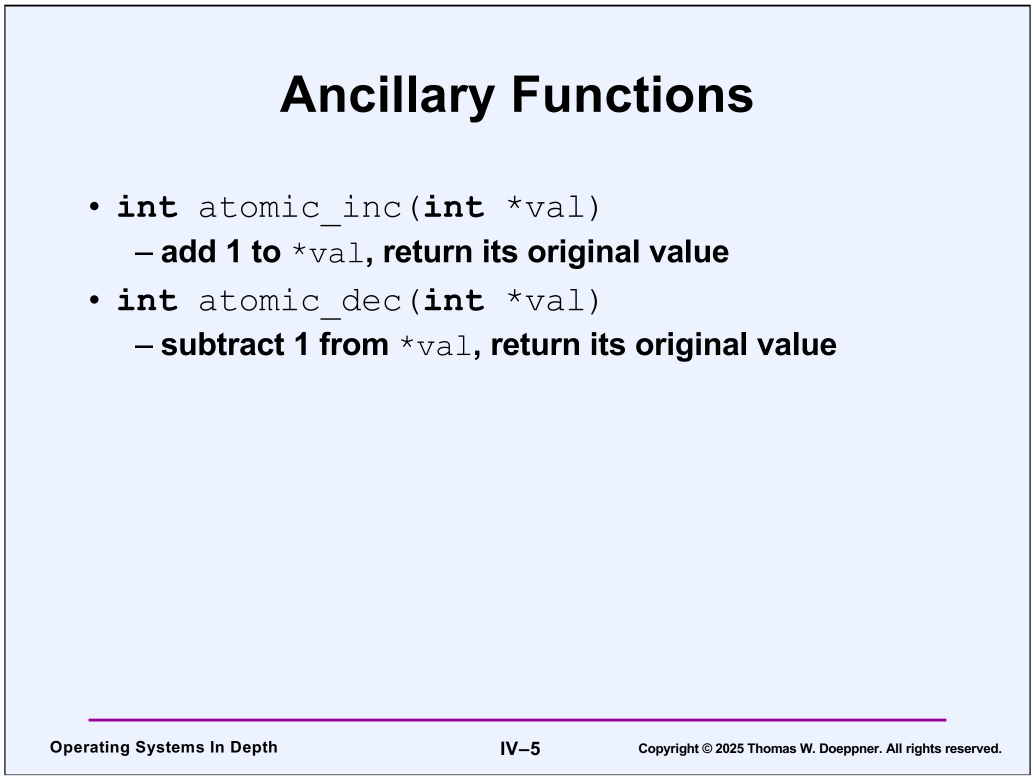
c) never works



Futexes

- Safe, *efficient* kernel conditional queueing in Linux
- All operations performed atomically
  - `futex_wait(`**`futex_t`**` *futex, `**`int`**` val)`
    - if `futex->val` is equal to `val`, then sleep
    - otherwise return
  - `futex_wake(`**`futex_t`**` *futex)`
    - wake up one thread from `futex`'s wait queue, if there are any waiting threads

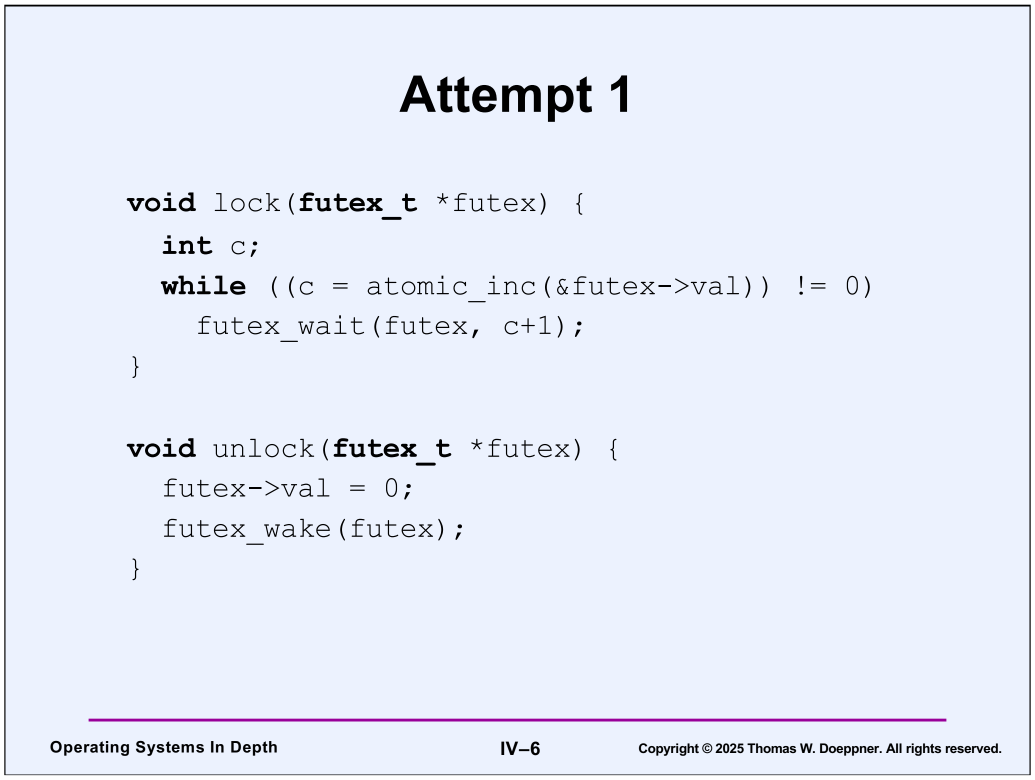
For details on futexes, avoid the Linux man pages, but look at http://people.redhat.com/drepper/futex.pdf, from which this material was obtained. Note that there's actually just one **futex** system call; whether it's a **wait** or a **wakeup** is specified by an argument.



Ancillary Functions

- `int atomic_inc(int *val)`
  - add 1 to `*val`, return its original value
- `int atomic_dec(int *val)`
  - subtract 1 from `*val`, return its original value

These functions are available on most architectures, particularly on the x86. Note that their effect must be *atomic*: everything happens at once.

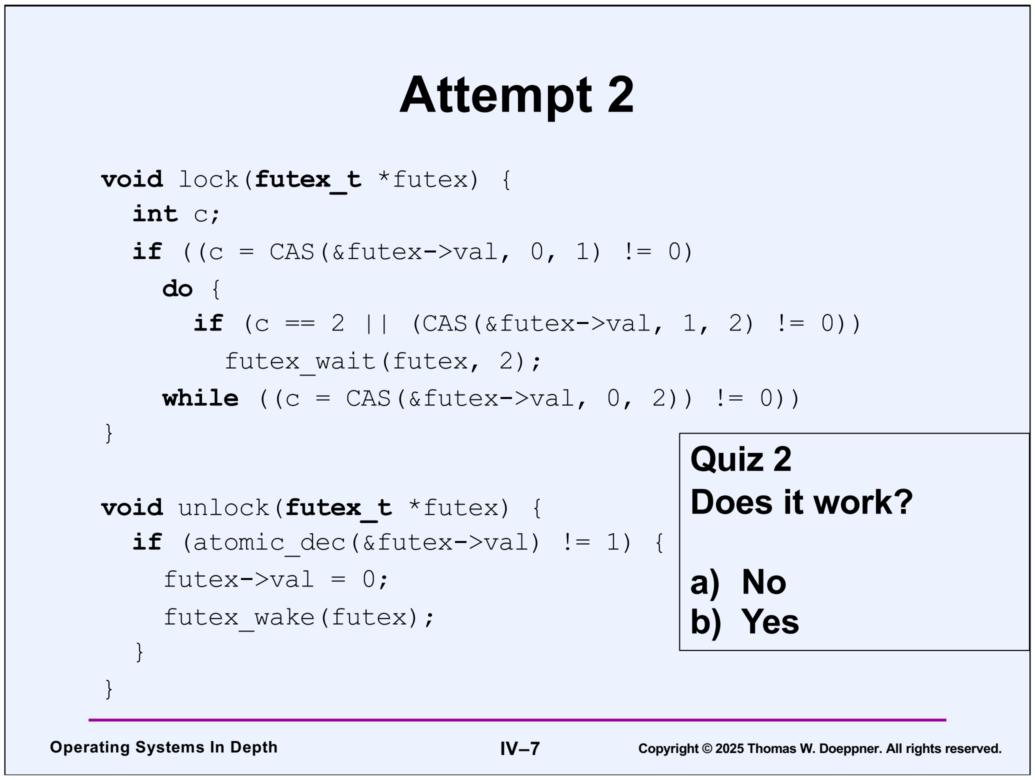


Attempt 1

```
void lock(futex_t *futex) {
  int c;
  while ((c = atomic_inc(&futex->val)) != 0)
    futex_wait(futex, c+1);
}

void unlock(futex_t *futex) {
  futex->val = 0;
  futex_wake(futex);
}
```

In this code, the value of the **futex** is 0 if the mutex is unlocked, and is greater than zero if it's locked. The **futex_wait** call puts the caller to sleep if the value of the **futex** is what is expected: the result of adding one to its previous value. If it's something different, then some other thread has modified the **futex**, and the caller returns to check if it is still locked.

This code has a potential problem if multiple threads are locking the mutex and all are in the while loop. After one thread sets the futex's value, another thread modifies it before the first thread has entered **futex_wait**. Then the first thread modifies the futex's value before the second thread calls **futex_wait**. This can continue on indefinitely to the point that the futex's value overflows and wraps around to zero, causing the mutex to appear unlocked.



Attempt 2

```
void lock(futex_t *futex) {
  int c;
  if ((c = CAS(&futex->val, 0, 1) != 0)
    do {
      if (c == 2 || (CAS(&futex->val, 1, 2) != 0))
        futex_wait(futex, 2);
    while ((c = CAS(&futex->val, 0, 2)) != 0))
}

void unlock(futex_t *futex) {
  if (atomic_dec(&futex->val) != 1) {
    futex->val = 0;
    futex_wake(futex);
  }
}
```

Quiz 2
Does it work?

a) No
b) Yes

In this code, a futex value of 0 means the mutex is unlocked. A value of 1 means the mutex is locked and no threads are waiting for it to be unlocked. A value of 2 means that the mutex is locked, possibly with threads waiting for it. The futex should take on no other values.

Blocking Locks in MThreads

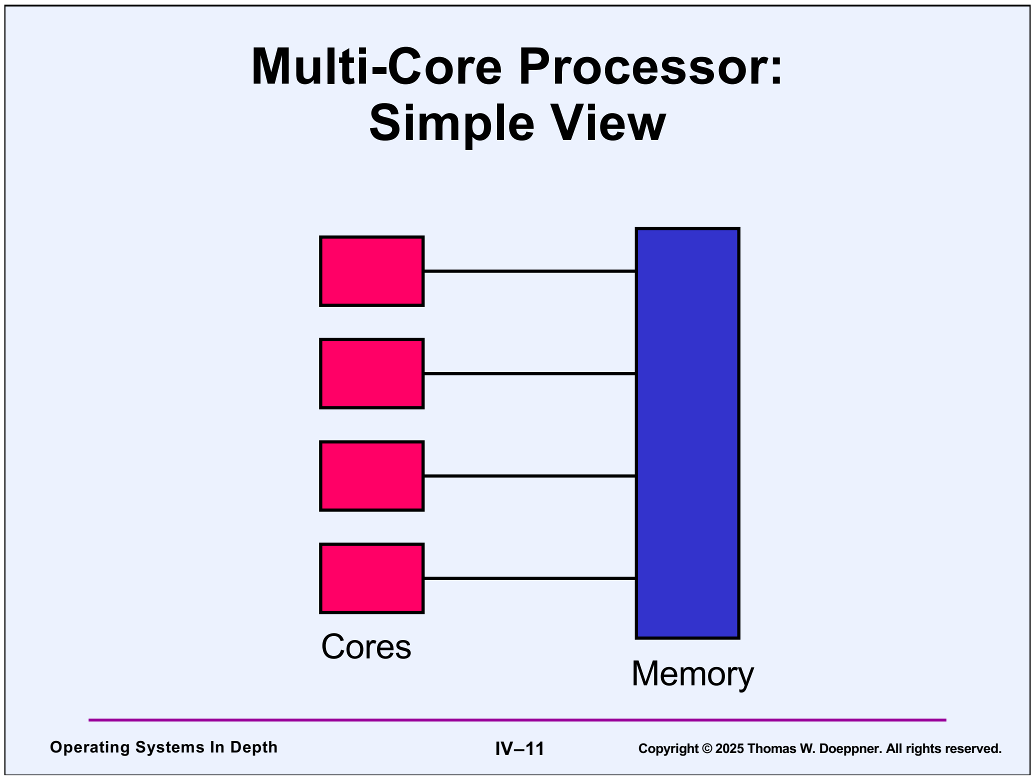
- We could use futexes, but don’t
- *uthread_switch* gets an additional argument
  - a POSIX mutex (representing a spin lock)
  - unlock it after getting out of the context of the calling thread

Actual Code

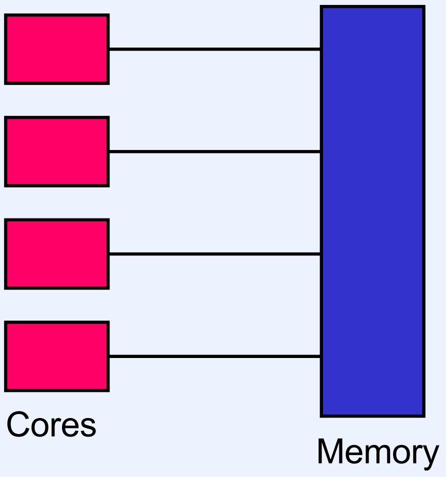
```
uthread_mtx_lock(uthread_mtx_t *mtx) {
    uthread_nopreempt_on();
    pthread_mutex_lock(&mtx->m_pmut);
    if (mtx->m_owner == NULL) {
        mtx->m_owner = ut_curthr;
        pthread_mutex_unlock(&mtx->m_pmut);
        uthread_nopreempt_off();
    } else {
        ut_curthr->ut_state = UT_WAIT;
        uthread_switch(&mtx->m_waiters, 0, &mtx->m_pmut);
        uthread_nopreempt_off();
    }
}
```

MP Memory Issues

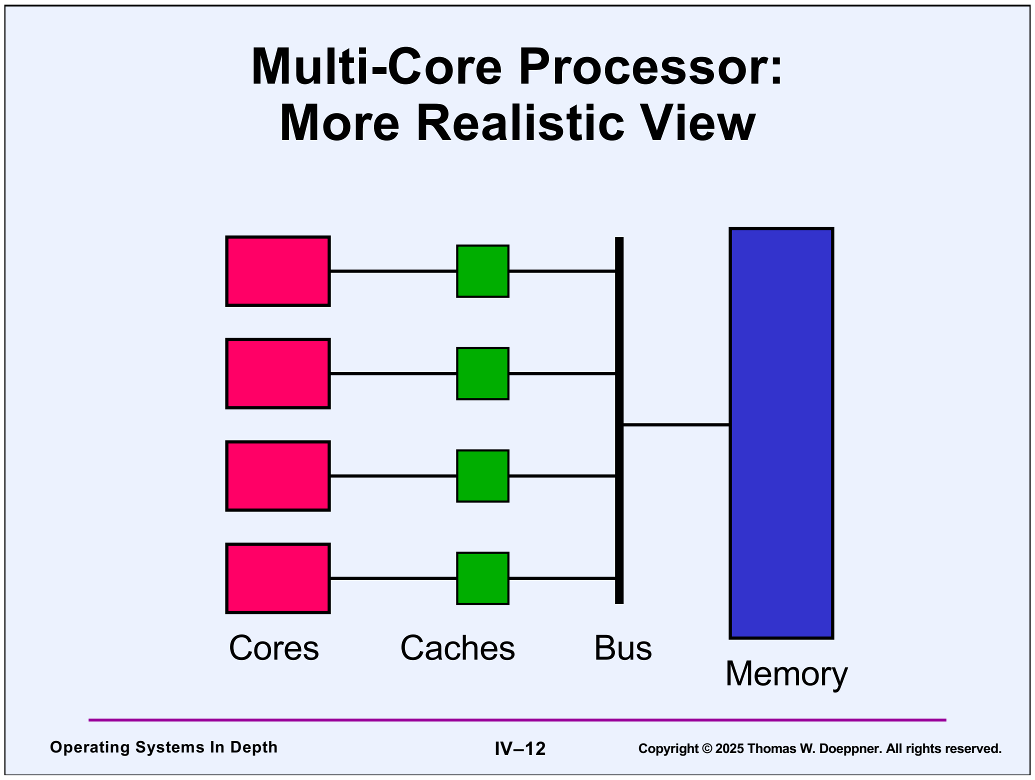
- Naive view is that all processors in MP system see same memory contents at all times
  - they don't



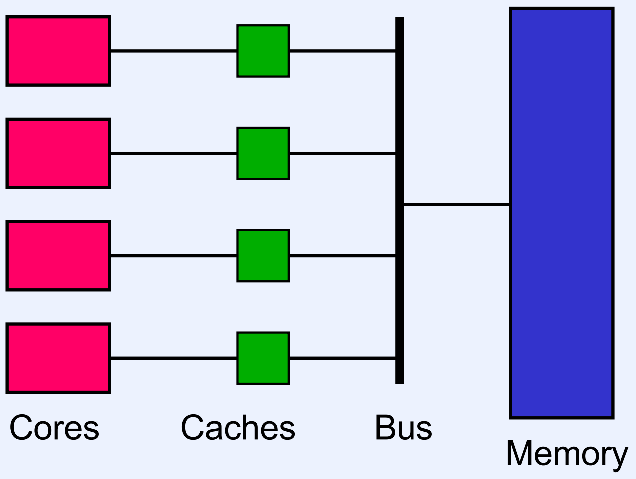
Multi-Core Processor: Simple View



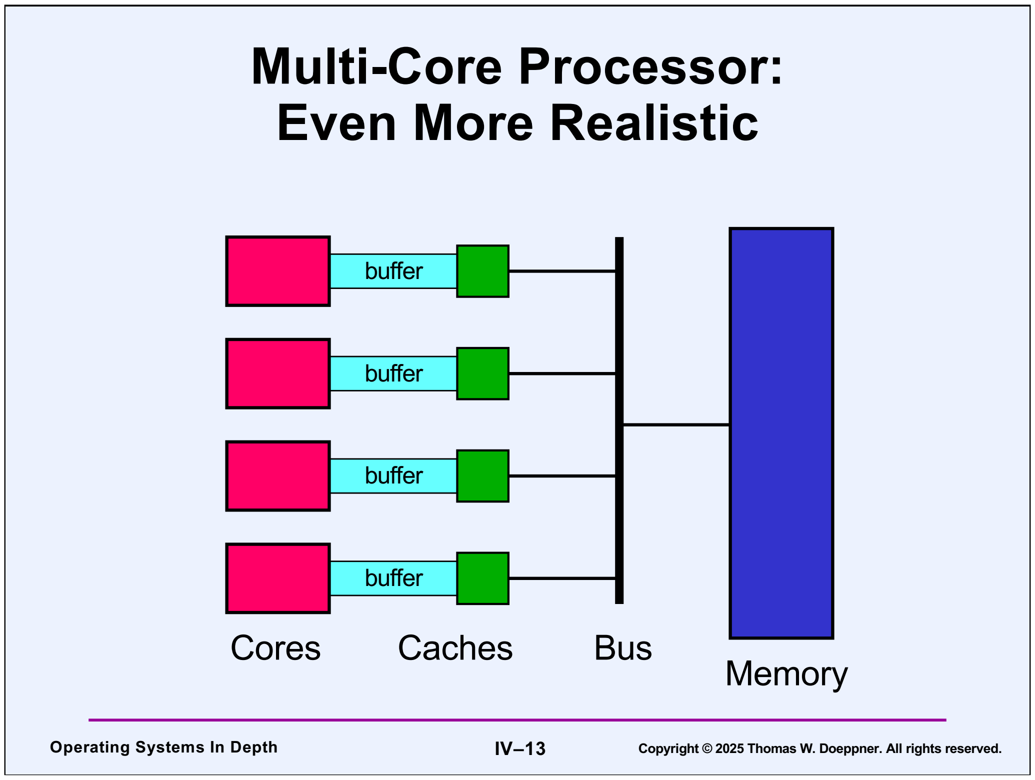
This slide illustrates a simplistic view of the architecture of a multi-core processor: a number of processors are all directly connected to the same memory (which they share). If one core (or processor) stores into a storage location and immediately thereafter another core loads from the same storage location, the second core loads exactly what the first core stored.



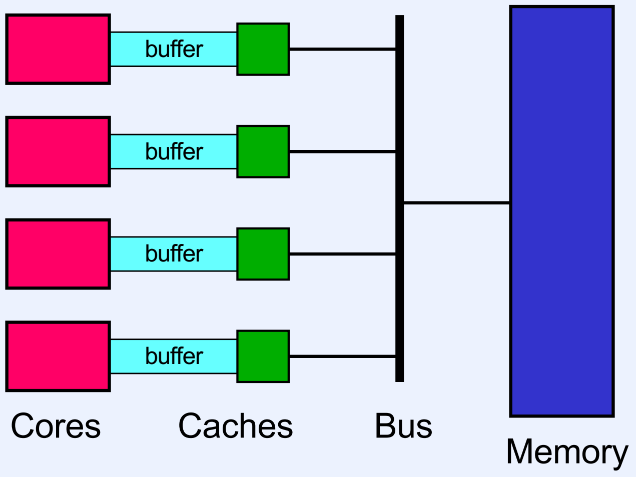
Multi-Core Processor: More Realistic View



Real multi-core processors employ a hierarchy of caches between the cores and memory, will the caches sharing the bus with the memory controller. (The Intel I-5 architecture, used on SunLab machines, has three levels of caches: two L1 caches (one each for data and instructions) per core, one L2 cache per core, and an L3 cache shared by all four cores. For the purposes of this discussion, it suffices to think of just one cache for each core.) An elaborate cache-coherency protocol is used so that the caches are consistent: no two caches have different versions of the same data item and if some a particular cache contains a data item that different (and newer) than what's in memory, other caches when they attempt to load that item will get the newest version.

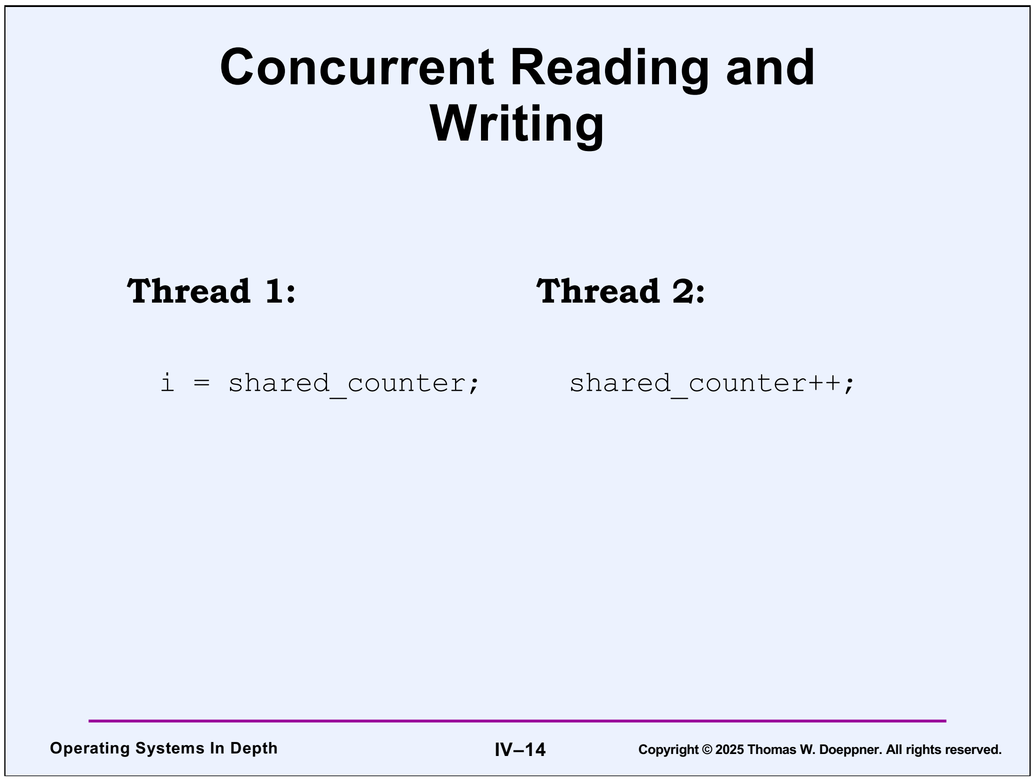


Multi-Core Processor: Even More Realistic



This slide shows an even more realistic model, pretty much the same as what we saw is actually used in recent Intel processors. Between each core and its caches is a store buffer. Stores by a core go into the buffer. Sometime later the effect of the store reaches the cache. In the meantime, the core is issuing further instructions. Loads by the core are handled from the buffer if the data is still there; otherwise, they are taken from the caches, or perhaps from memory.

In all instances of this model the effect of a store, as seen by other cores, is delayed. In some instances of this model the order of stores made by one core might be perceived differently by other cores. Architectures with the former property are said to have **delayed stores**; architectures with the latter are said to have **reordered stores** (an architecture could well have both properties).



Concurrent Reading and Writing

Thread 1:

```
i = shared_counter;
```

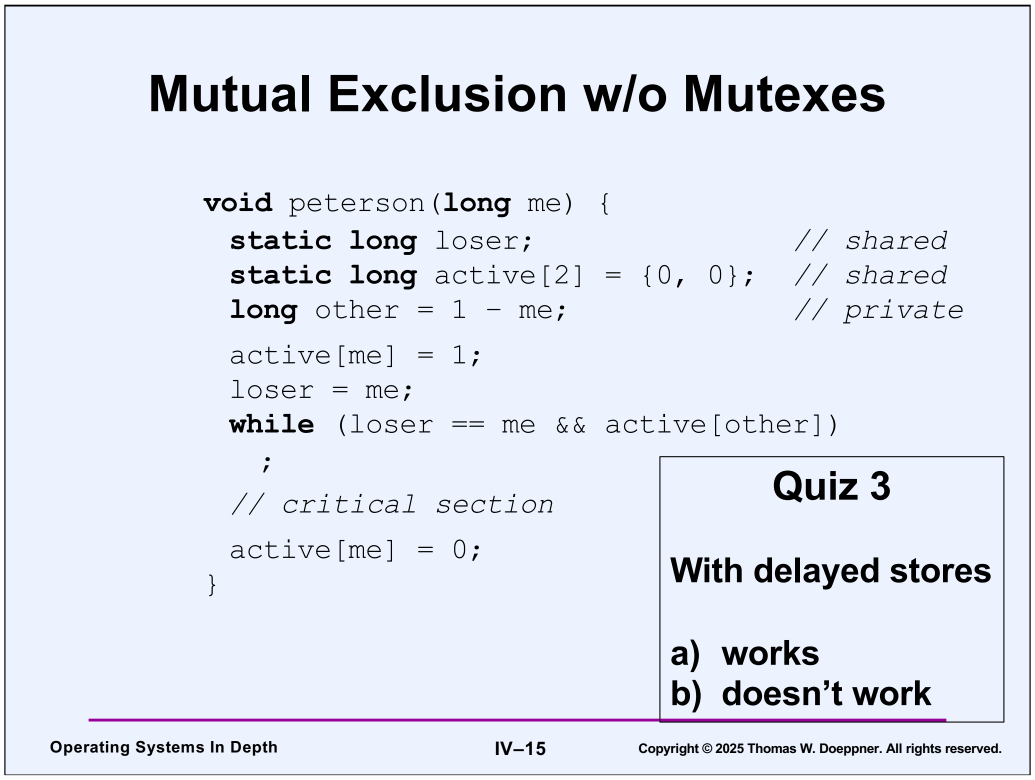
Thread 2:

```
shared_counter++;
```

In this example, one thread running on one processor is loading from an integer in storage; another thread running on another processor is loading from and then storing into an integer in storage. Can this be done safely without explicit synchronization?

On most architectures, the answer is yes. If the integer in question is aligned on a natural (e.g., eight-byte) boundary, then the hardware (perhaps the cache) insures that loads and stores of the integer are atomic.

However, one cannot assume that this is the case on all architectures. Thus a portable program must use explicit synchronization (e.g., a mutex) in this situation.



Mutual Exclusion w/o Mutexes

```
void peterson(long me) {
  static long loser;                    // shared
  static long active[2] = {0, 0};  // shared
  long other = 1 - me;              // private
  active[me] = 1;
  loser = me;
  while (loser == me && active[other])
    ;
  // critical section
  active[me] = 0;
}
```

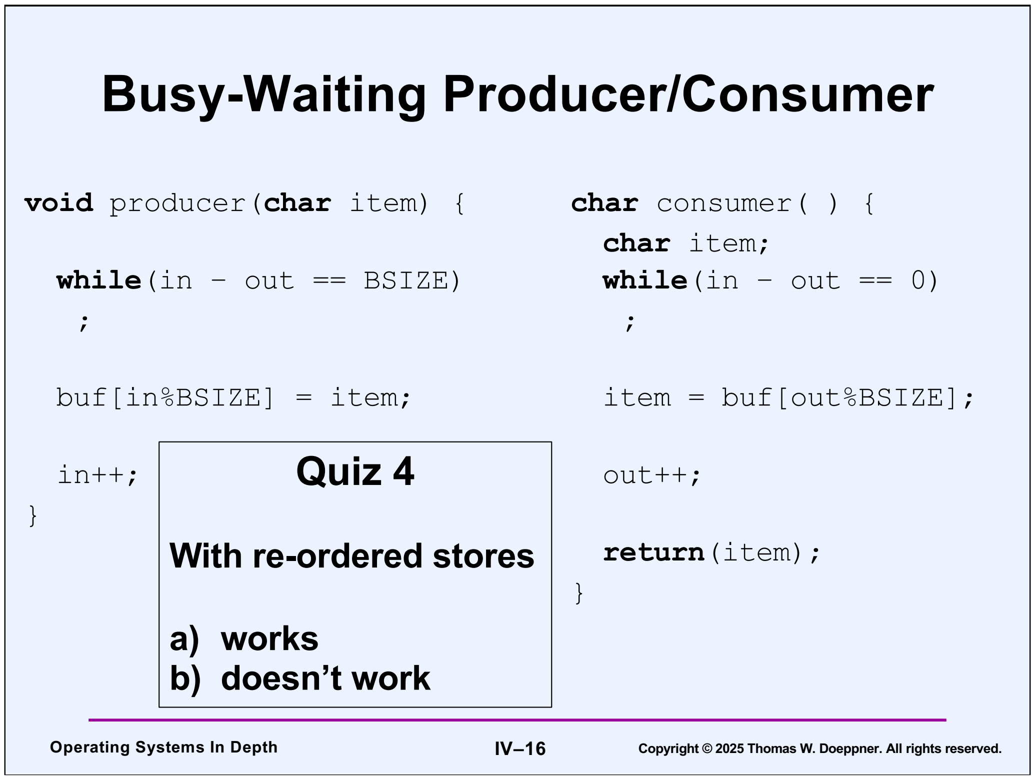
Quiz 3

With delayed stores

a) works
b) doesn't work

Shown on the slide is Peterson's algorithm for handling mutual exclusion for two threads without explicit synchronization. (The **me** argument for one thread is 0 and for the other is 1.) This program works given the first two shared-memory models. Does it work with delayed-store architectures?

The algorithm is from "Myths About the Mutual Exclusion Problem," by G. L. Peterson, Information Processing Letters 12(3) 1981: 115–116.



Busy-Waiting Producer/Consumer

```
void producer(char item) {

  while(in - out == BSIZE)
    ;

  buf[in%BSIZE] = item;

  in++;
}
```

```
char consumer( ) {
  char item;
  while(in - out == 0)
    ;

  item = buf[out%BSIZE];

  out++;

  return(item);
}
```

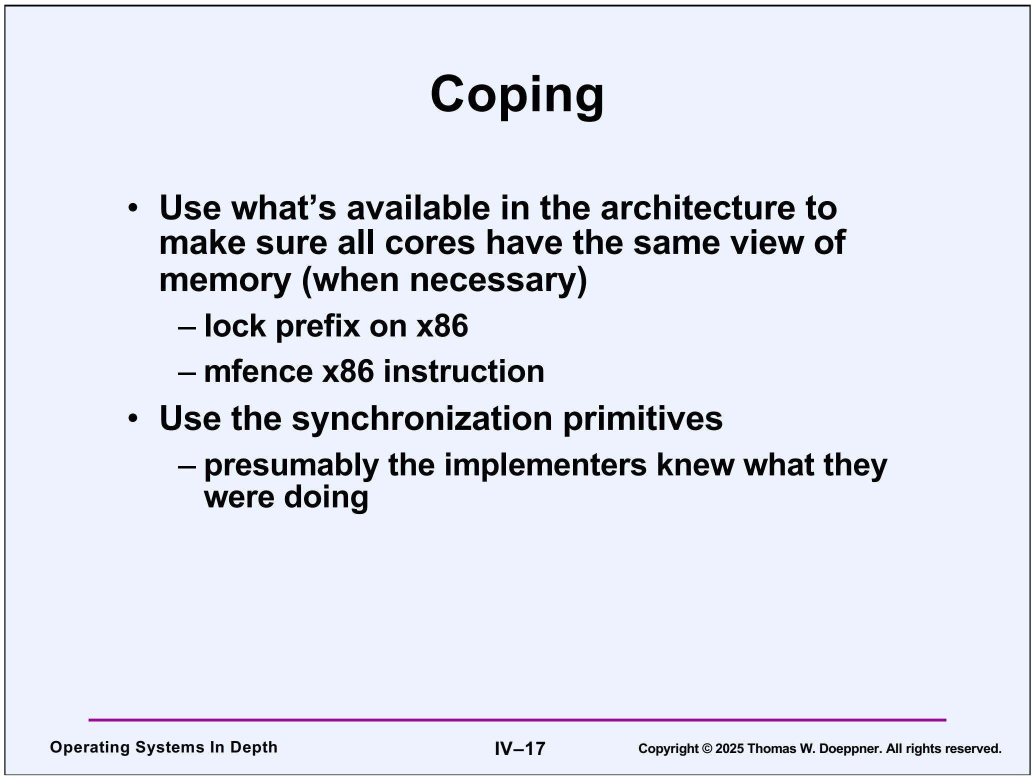
Quiz 4

With re-ordered stores

a) works
b) doesn’t work

This example is a solution, employing “busy waiting,” to the producer-consumer problem for one consumer and one producer. It works for the first two shared-memory models, and even for delayed-store architectures. But does it work on reordered-store architectures?

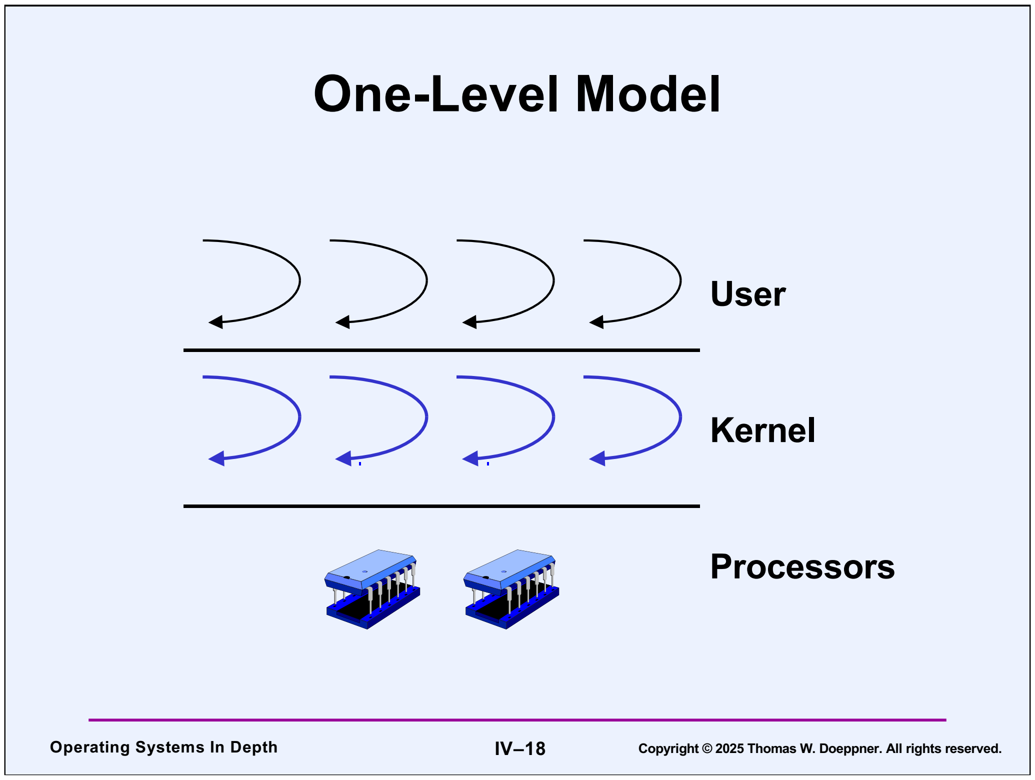
This solution to the producer-consumer problem is from “Proving the Correctness of Multiprocess Programs,” by L. Lamport, IEEE Transactions on Software Engineering, SE-3(2) 1977: 125-143.



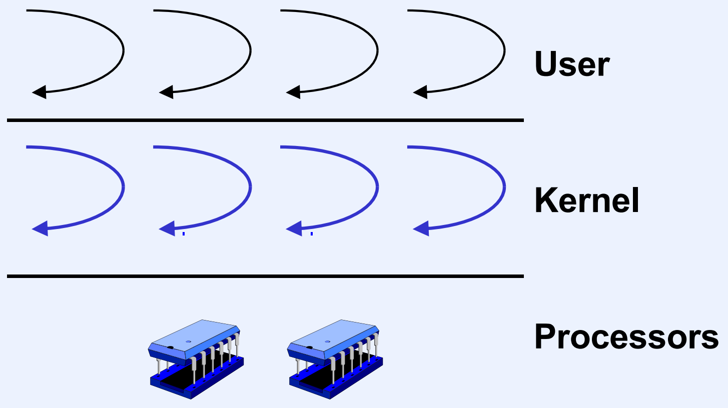
Coping

- Use what’s available in the architecture to make sure all cores have the same view of memory (when necessary)
  - lock prefix on x86
  - mfence x86 instruction
- Use the synchronization primitives
  - presumably the implementers knew what they were doing

The point of the previous several slides is that one cannot rely on expected properties of shared memory to eliminate explicit synchronization. Shared memory can behave in some very unexpected ways. However, it is the responsibility of the implementers of the various synchronization primitives to make certain not only that they behave correctly, but also that they synchronize memory with respect to other threads.



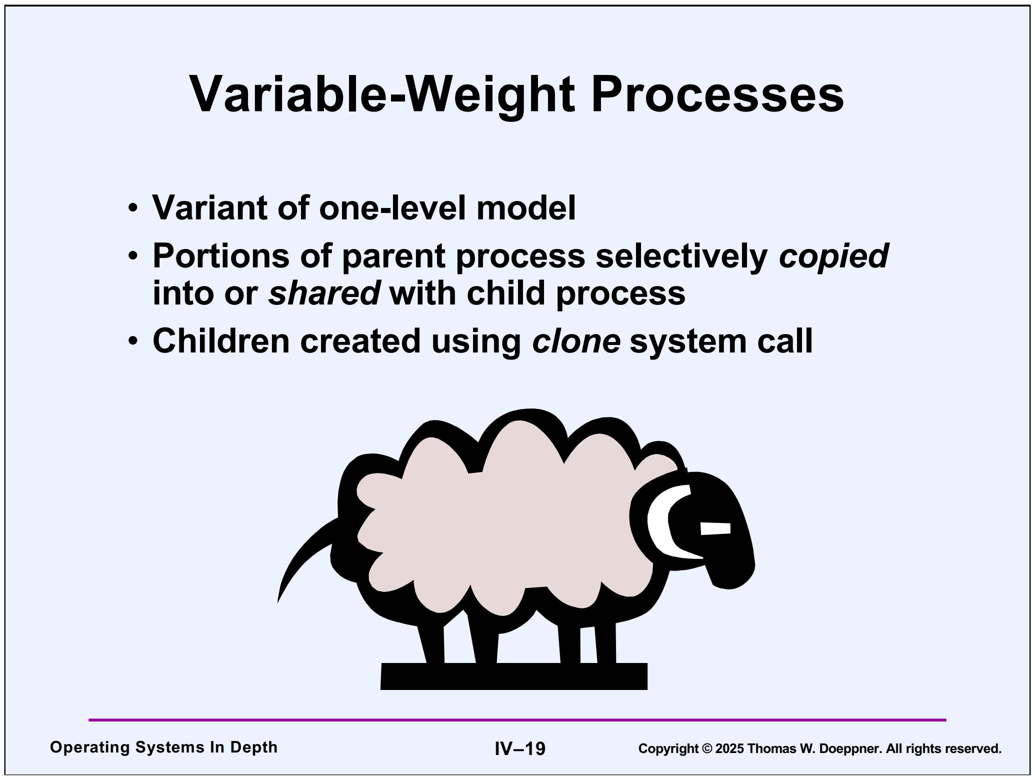
One-Level Model



We review some of the threads implementation concepts we've seen by looking at how threads are implemented (at a high level) in some real systems (some of which no longer exist).

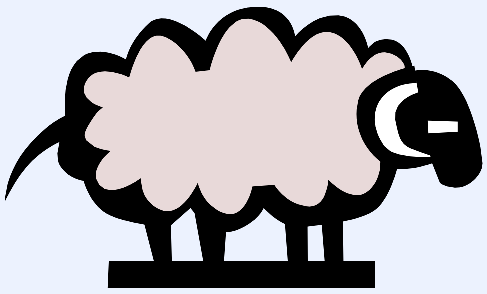
In most systems there are actually two components of the execution context: the user context and the kernel context. The former is for use when an activity is executing user code; the latter is for use when the activity is executing kernel code (on behalf of the chore). How these contexts are manipulated is one of the more crucial aspects of a threads implementation.

The conceptually simplest approach is what is known as the one-level model: each thread consists of both contexts. Thus a thread is scheduled to an activity and the activity can switch back and forth between the two types of contexts. A single scheduler in the kernel can handle all the multiplexing duties. The threading implementation in Windows is (mostly) done this way.



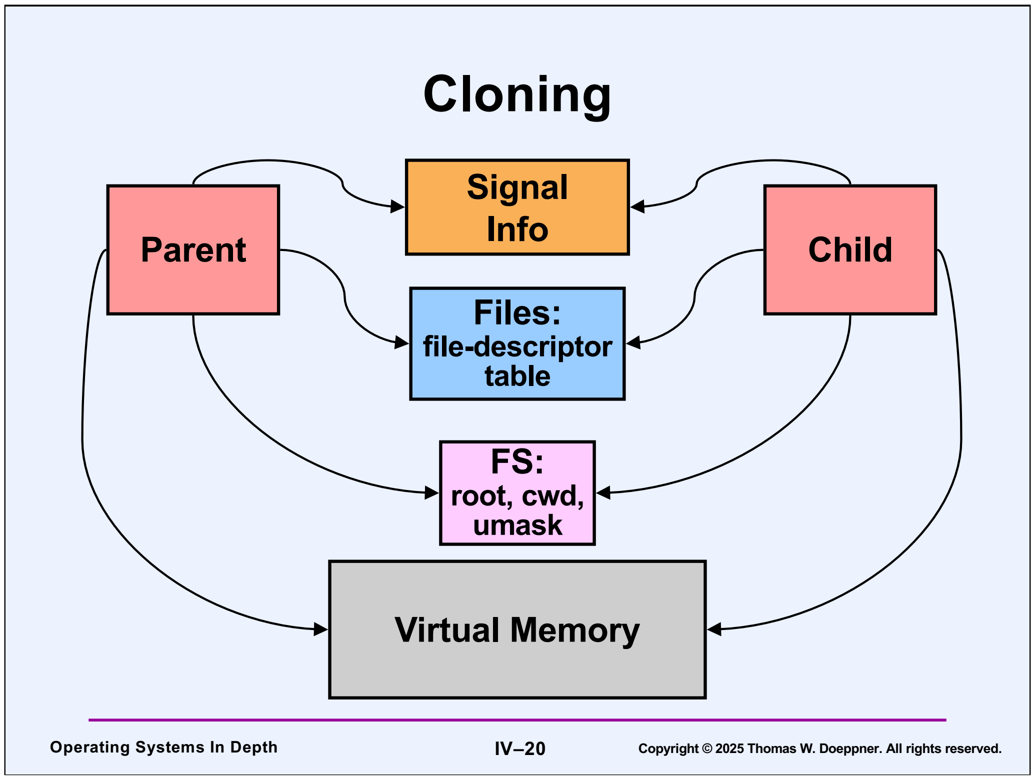
Variable-Weight Processes

- Variant of one-level model
- Portions of parent process selectively *copied* into or *shared* with child process
- Children created using *clone* system call

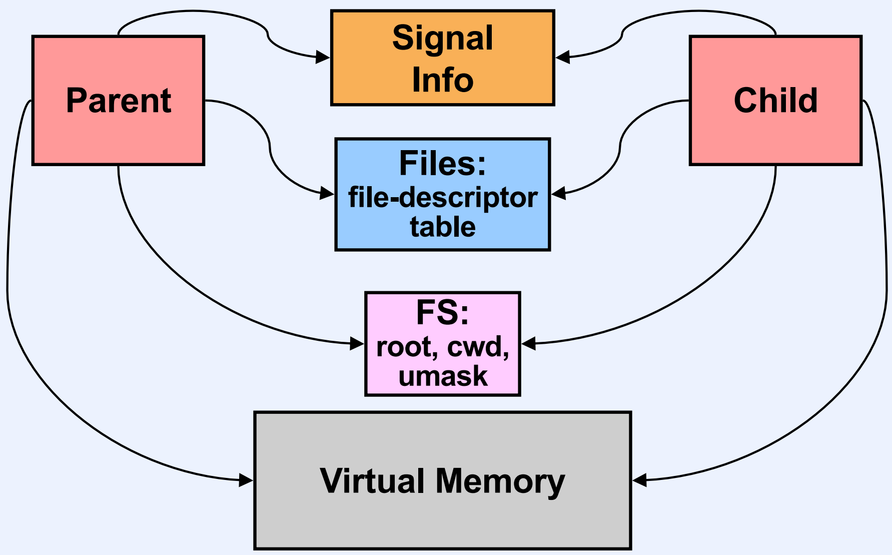


Unlike most other Unix systems, which make a distinction between processes and threads, allowing multithreaded processes, Linux maintains the one-thread-per-process approach. However, so that we can have multiple threads sharing an address space, Linux supports the **clone** system call, a variant of **fork**, via which a new process can be created that shares resources (in particular, its address space) with the parent. The result is a variant of the one-level model.

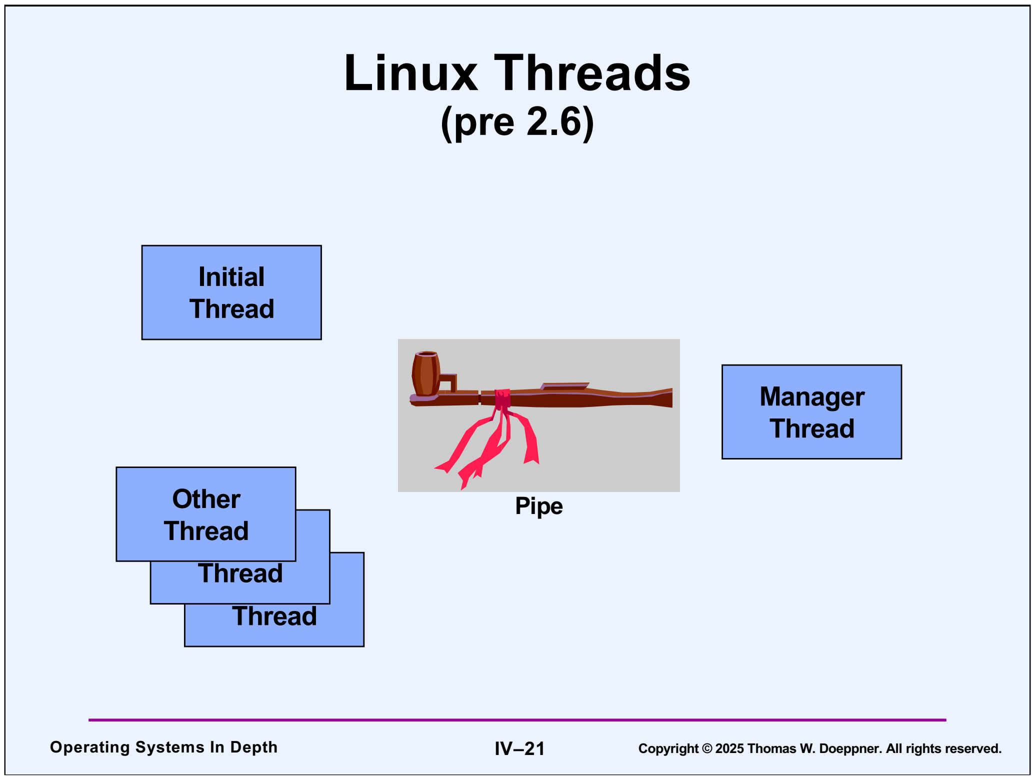
This approach is not unique to Linux. It was used in SGI's IRIX and was first discussed in early '89, when it was known as **variable-weight processes**. (See "Variable-Weight Processes with Flexible Shared Resources," by Z. Aral, J. Bloom, T. Doeppner, I. Gertner, A. Langerman, G. Schaffer, **Proceedings of Winter 1989 USENIX Association Meeting**.)



Cloning



As implemented in Linux, a process may be created with the *clone* system call (in addition to using the *fork* system call). One can specify, for each of the resources shown in the slide, whether a copy is made for the child or the child shares the resource with the parent. Only two cases are generally used: everything is copied (equivalent to fork) or everything is shared (creating what we ordinarily call a thread, though the “thread” has a separate process ID).



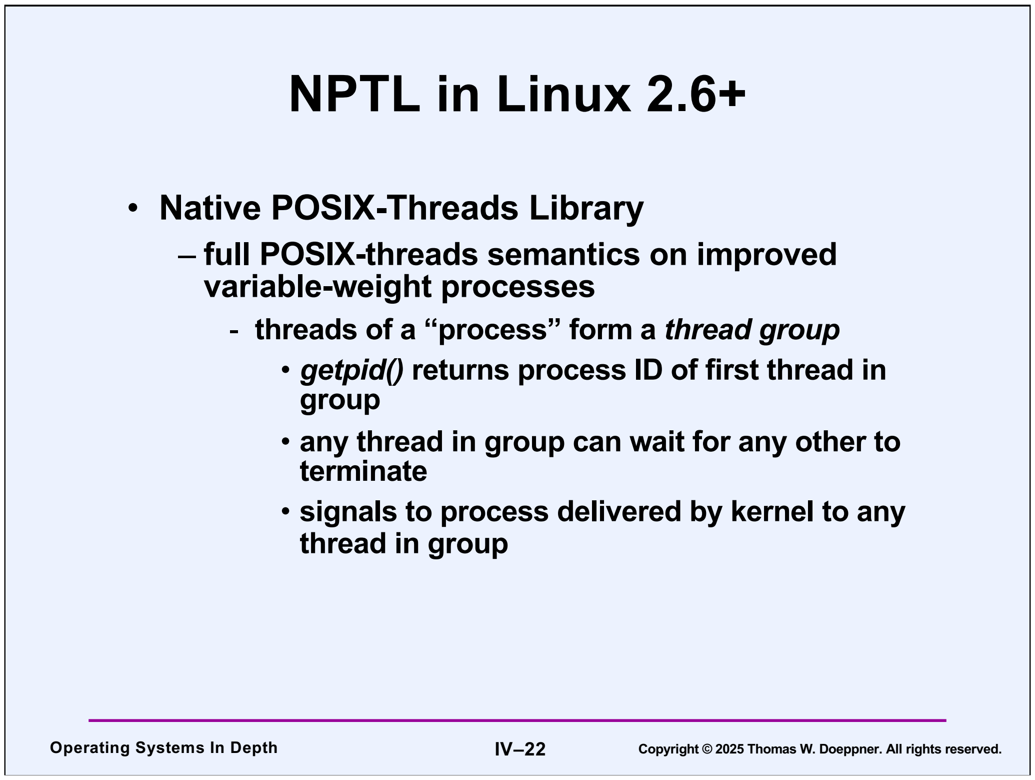
Linux Threads
(pre 2.6)

Building a POSIX-threads implementation on top of Linux's variable-weight processes requires some work. What's discussed here is the approach used prior to Linux 2.6.

Each thread is, of course, a process; all threads of the same computation share the same address space, open files, and signal handlers. One might expect that the implementation of *pthread_create* would be a simple call to clone. This, unfortunately, wouldn't allow an easy implementation of operations such as **pthread_join**: a Unix process may wait only for its children to terminate; a POSIX thread can join with any other joinable thread. Furthermore, if a Unix process terminates, its children are inherited by the init process (process number 1). So that **pthread_join** can be implemented without undue complexity, a special manager thread (actually a process) is the parent/creator of all threads other than the initial thread. This manager thread handles thread (process) termination via the wait4 system call and thus provides a means for implementing **pthread_join**. So, when any thread invokes *pthread_create* or **pthread_join**, it sends a request to the manager via a pipe and waits for a response. The manager handles the request and wakes up the caller when appropriate.

The state of a mutex is represented by a bit. If there are no competitors for locking a mutex, a thread simply sets the bit with a compare-and-swap instruction (allowing atomic testing and setting of the mutex's state bit). If a thread must wait for a mutex to be unlocked, it blocks using a **sigsuspend** system call, after queuing itself to a queue headed by the mutex. A thread unlocking a mutex wakes up the first waiting thread by sending it a Unix signal (via the kill system call). The wait queue for condition variables is implemented in a similar fashion.

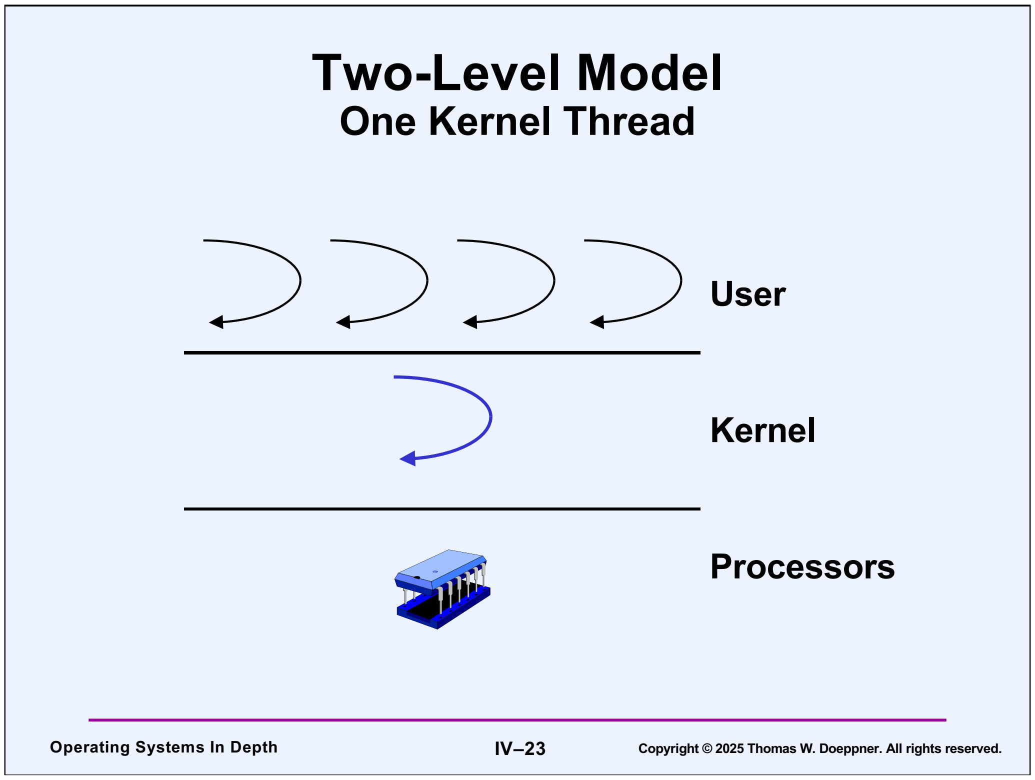
On multiprocessors, for mutexes that are neither recursive nor error-checking, waiting is implemented with an adaptive strategy: under the assumption that mutexes are typically not held for a long period of time, a thread attempting to lock a locked mutex “spins” on it for up to a short period of time, i.e., it repeatedly tests the state of the mutex in hopes that it will be unlocked. If the mutex does not become available after the maximum number of tests, then the thread finally blocks by queuing itself and calling **sigsuspend**.



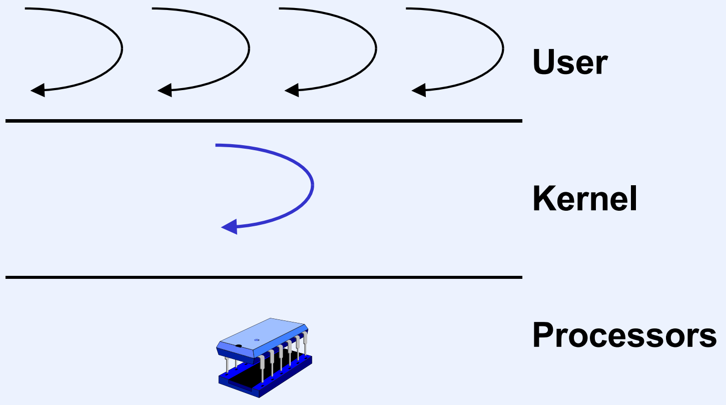
NPTL in Linux 2.6+

- Native POSIX-Threads Library
  - full POSIX-threads semantics on improved variable-weight processes
    - threads of a "process" form a *thread group*
      - *getpid()* returns process ID of first thread in group
      - any thread in group can wait for any other to terminate
      - signals to process delivered by kernel to any thread in group

NPTL, the "Native POSIX Threads Library" that comes with Linux systems starting with 2.6, provides a big improvement over the previous version of threads on Linux, which is referred to as "Linux Threads." There's no need for a manager thread anymore, signal-handling semantics are now as they should be in POSIX, and synchronization constructs are implemented much more efficiently than on Linux Threads.

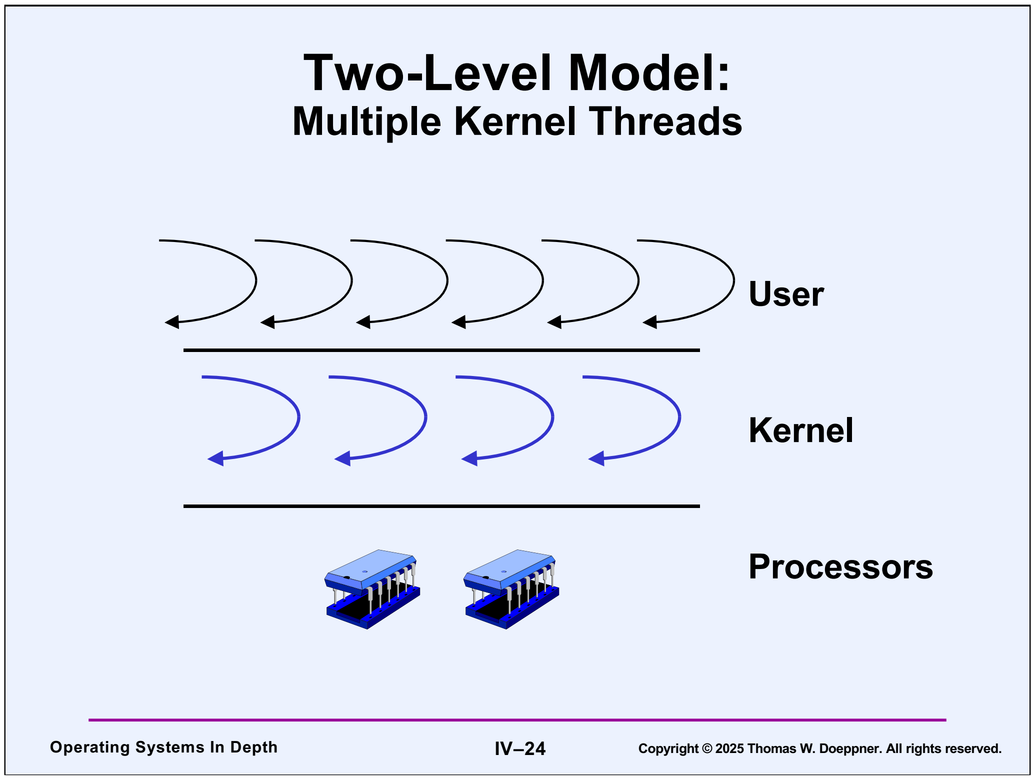


Two-Level Model
One Kernel Thread

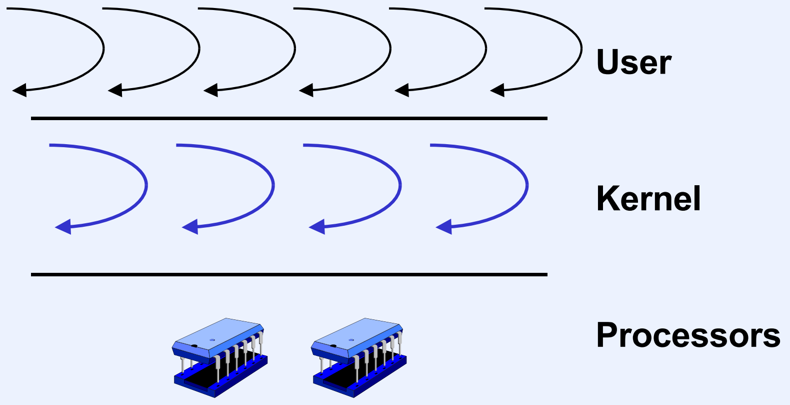


Another approach, the two-level model, is to represent the two contexts as separate types of threads: user threads and kernel threads. Kernel threads become "virtual activities" upon which user threads are scheduled. Thus two schedulers are used: kernel threads are multiplexed on activities by a kernel 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).

This is the approach we use in uthreads.

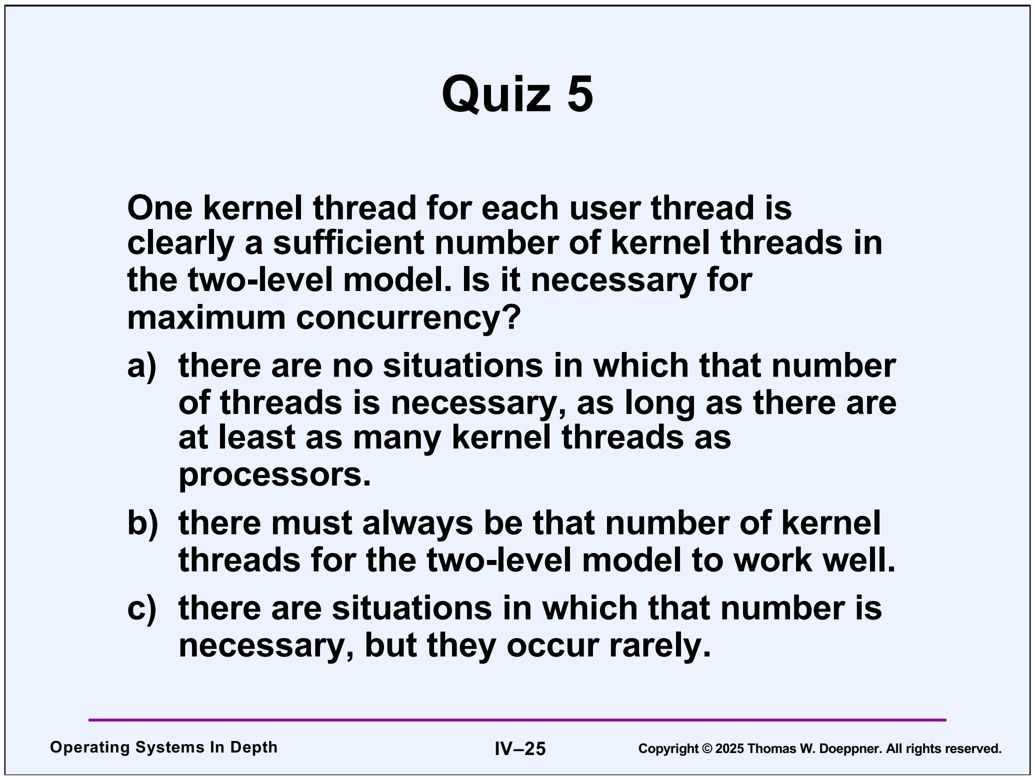


Two-Level Model:
Multiple Kernel Threads



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 5

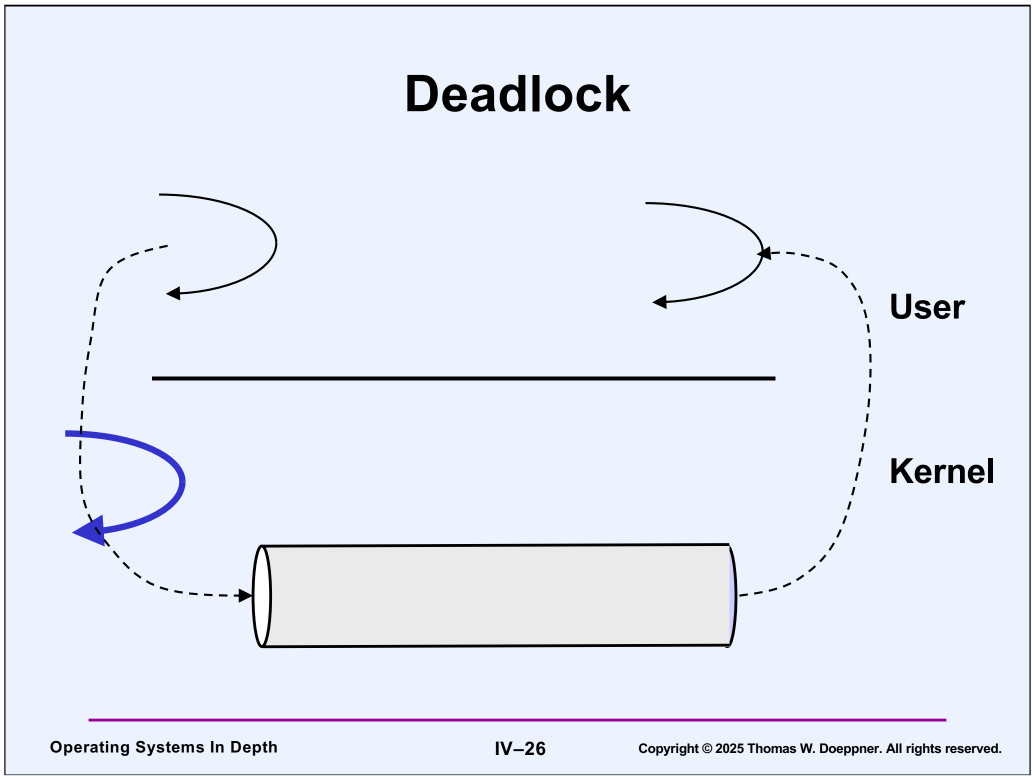
One kernel thread for each user thread is clearly a sufficient number of kernel threads in the two-level model. Is it necessary for maximum concurrency?

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.



Deadlock

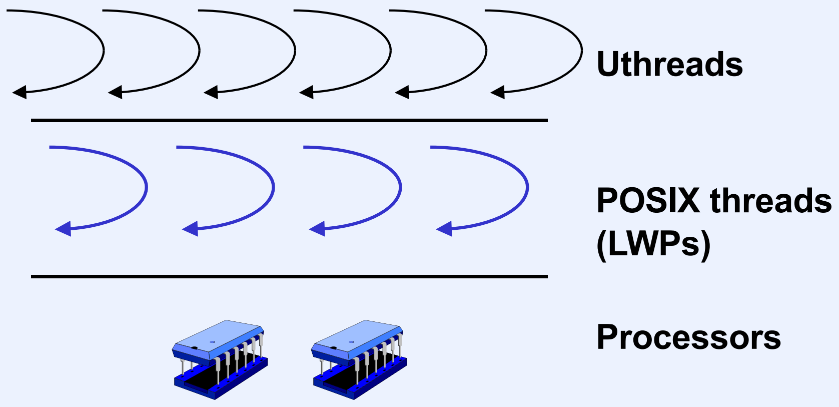
One negative aspect of the two-level model is that its use might induce deadlock. For example, suppose we have two user threads and one kernel thread. One thread is writing into a pipe (using the **write** system call). However, at the moment the pipe is full. The the call to write blocks. The other 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.

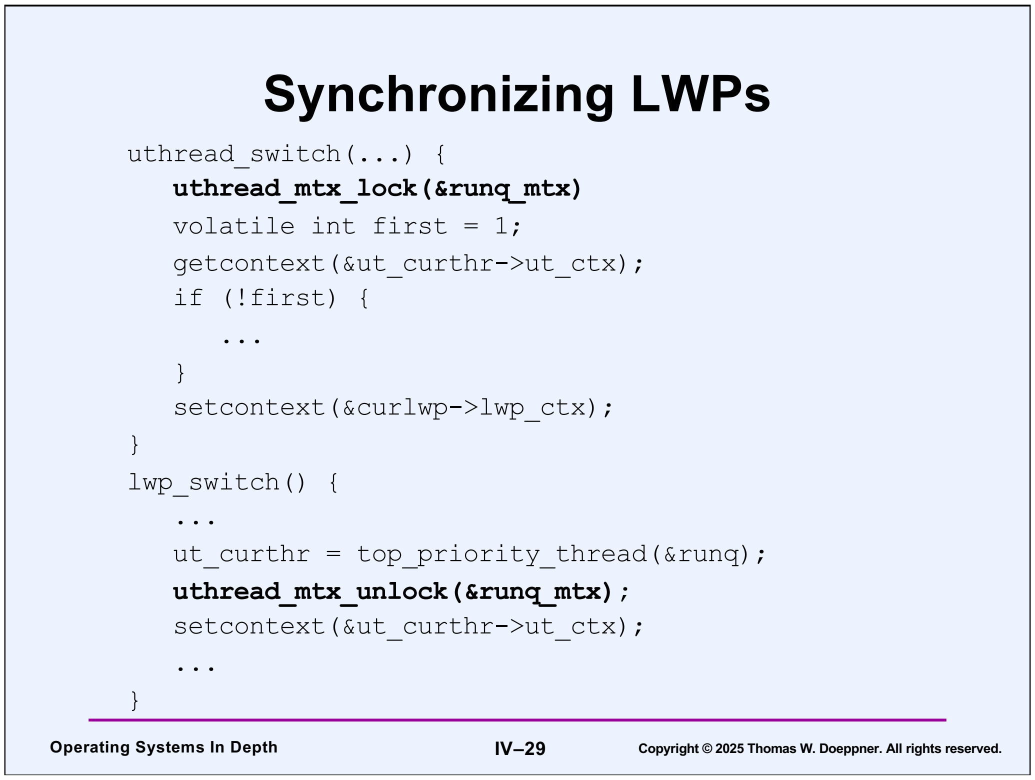
The solution is to introduce an additional kernel thread if such a situation happens. This was done in the Solaris implementation of the two-level thread model: if all kernel threads in a process are blocked, a new one was automatically created.

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

Two-Level Model:
MThreads



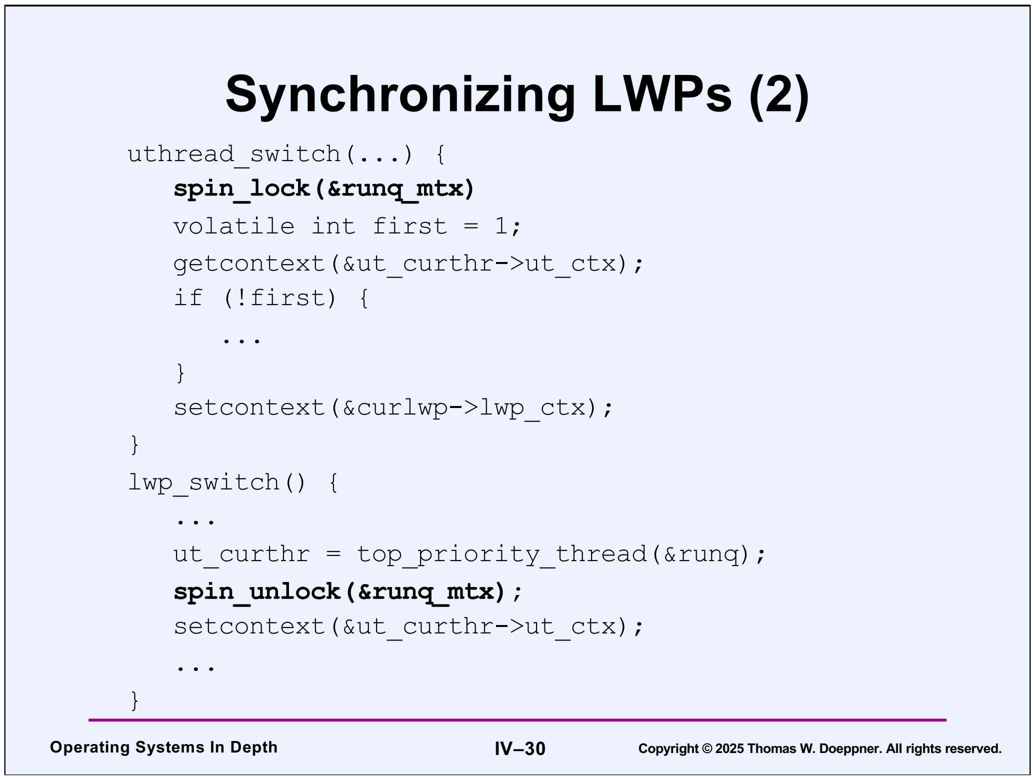


Synchronizing LWPs

```
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);
   ...
}
```

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, …

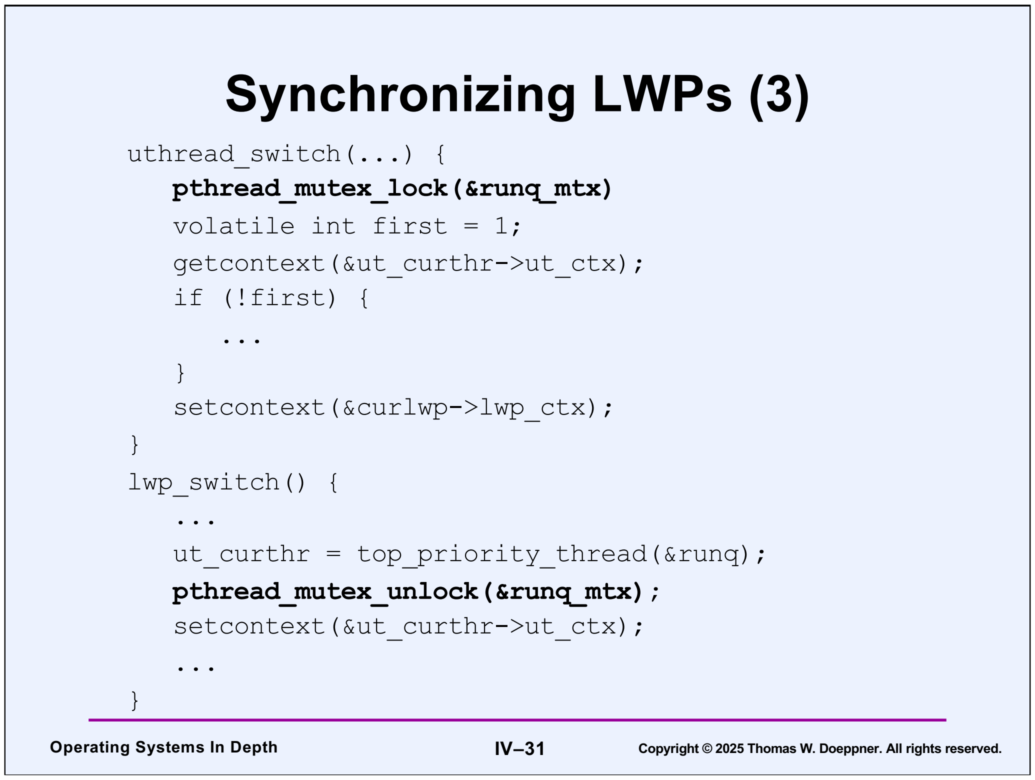


Synchronizing LWPs (2)

```
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.



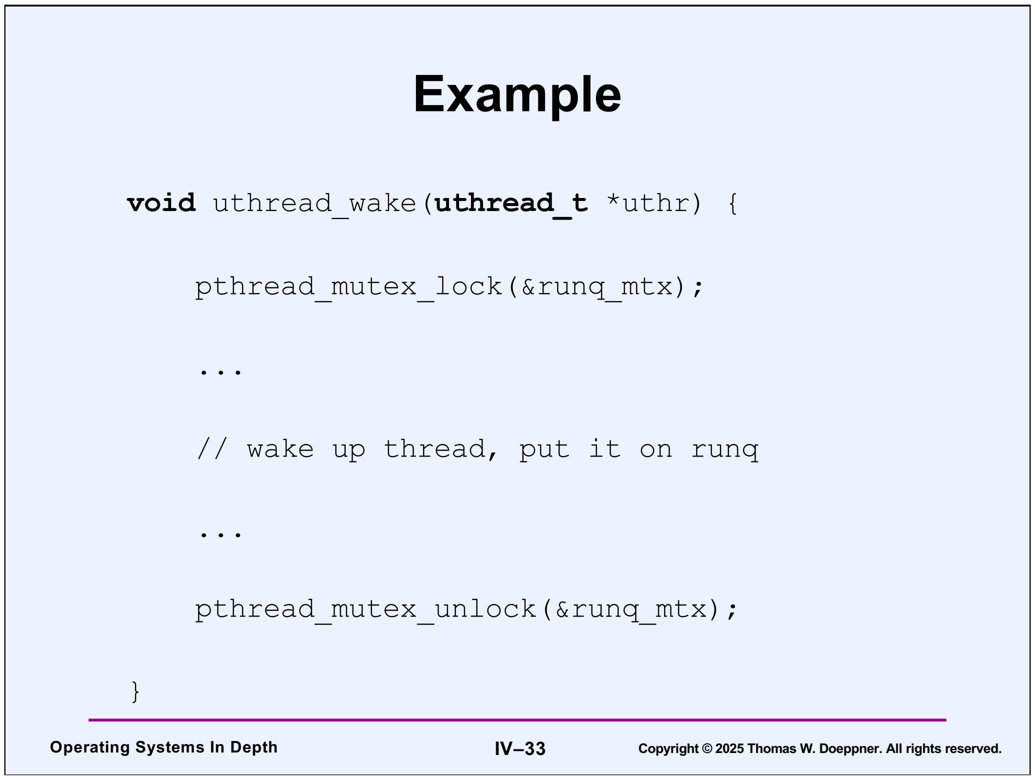
Synchronizing LWPs (3)

```
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



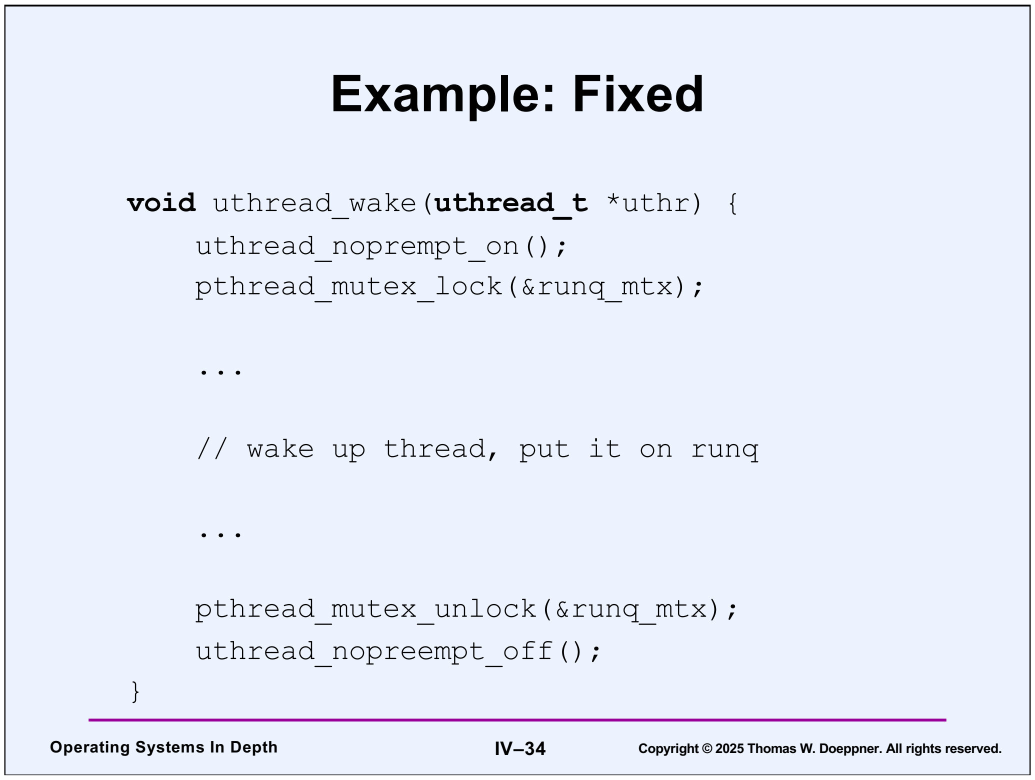
Example

```
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.

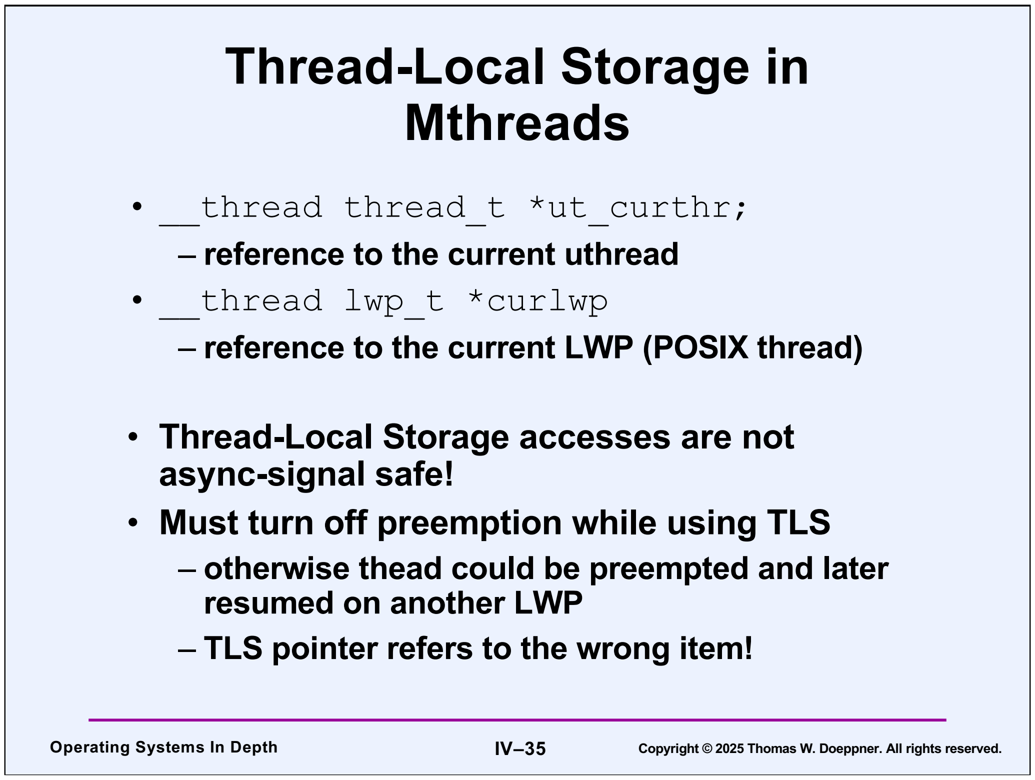
What might happen is that t1 is forced to yield to another thread, t2, while it is in the code that's modifying the runq (and thus the mutex is locked). But t2 also calls uthread_wake, and attempts to lock runq_mtx. But since its LWP already has the mutex locked, we have a deadlock.



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.