Implementing Threads 2

Time Slicing

- **Periodically**
  - **current thread forced to do a thread yield**

```
void ClockInterrupt(int sig) {
  thread_yield();
}
```

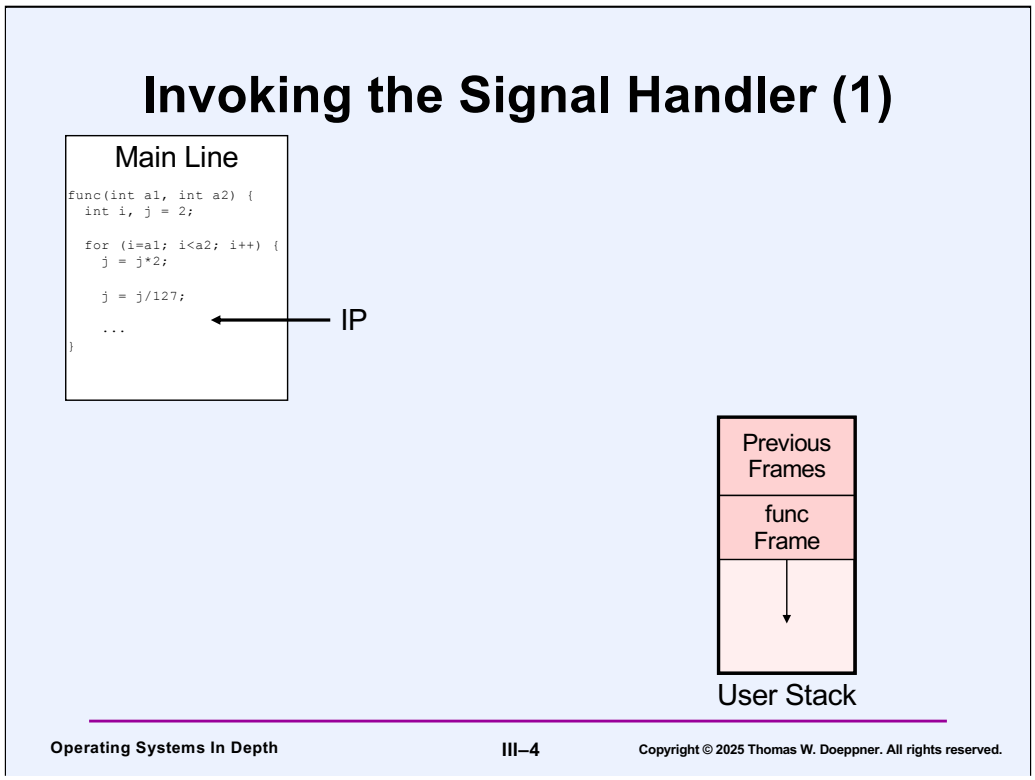
- **Implement ClockInterrupt with VTALRM signal**

Note that the use of time slicing complicates our implementation of mutexes and other synchronization constructs. We discuss this in in more detail soon.

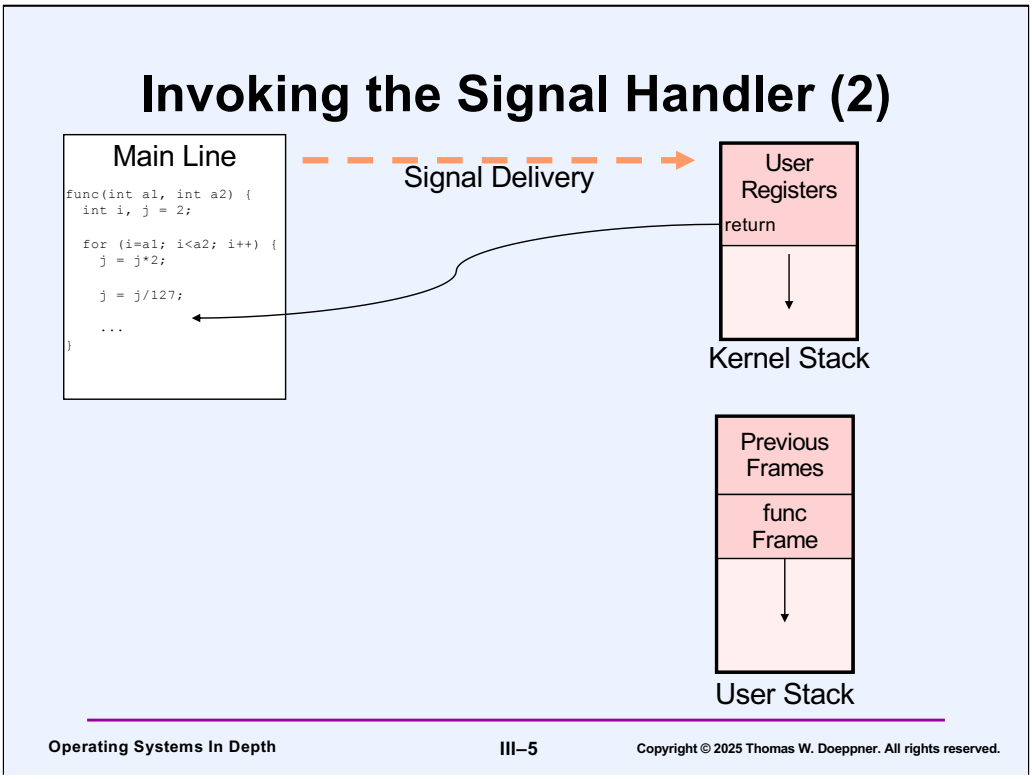
Invoking the Signal Handler

- **Basic idea is to set up the user stack so that the handler is called as a subroutine and so that when it returns, normal execution of the thread may continue**
- **Complications:**
  - **saving and restoring registers**
  - **signal mask**

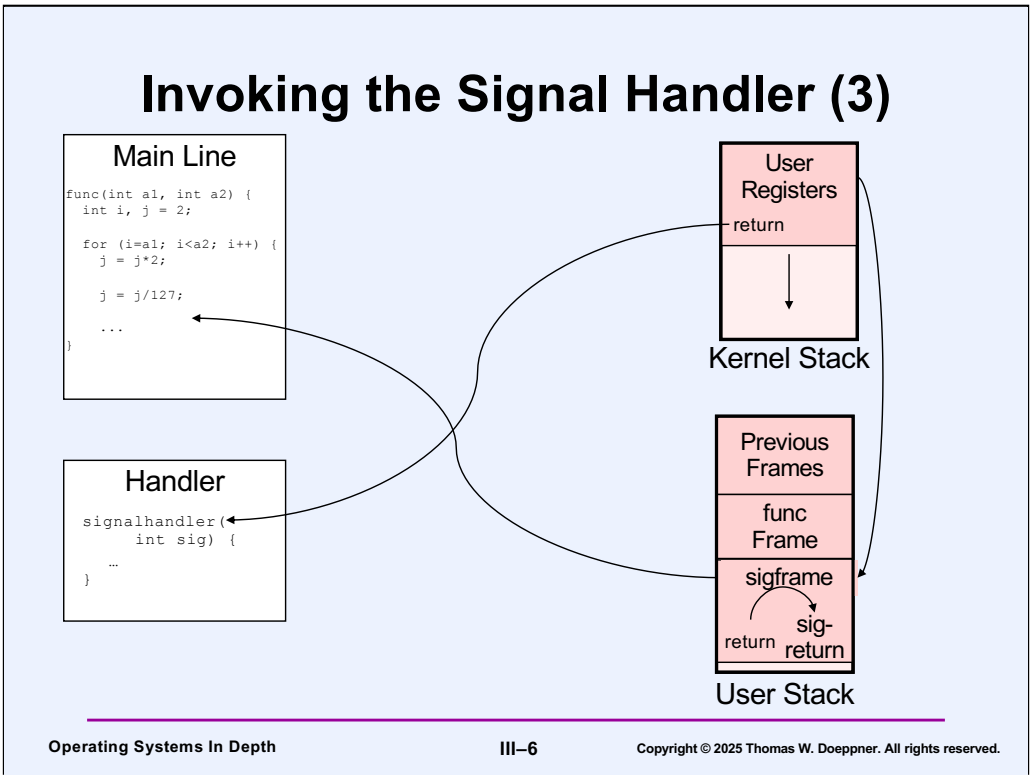
If a signal is to be dealt with via a signal handler, the kernel arranges so that the handler is invoked much like a subroutine, and that when it returns, the thread resumes its normal execution. To accomplish this, the state of the thread must be saved before invocation of the handler and resumed on return. A major component of the thread's state information is the contents of the registers: a typical subroutine does not save and restore **all** of the registers, since some are allowed to be modified (e.g., the register used to pass back a return value); since a signal handler can be invoked at any point, something must be done to save and restore all registers. We must also deal with masking signals while in the handler: invoking a signal handler causes a specified set of signals (including the one that was just delivered) to be masked off; the thread's original signal mask must be restored on return from the handler.



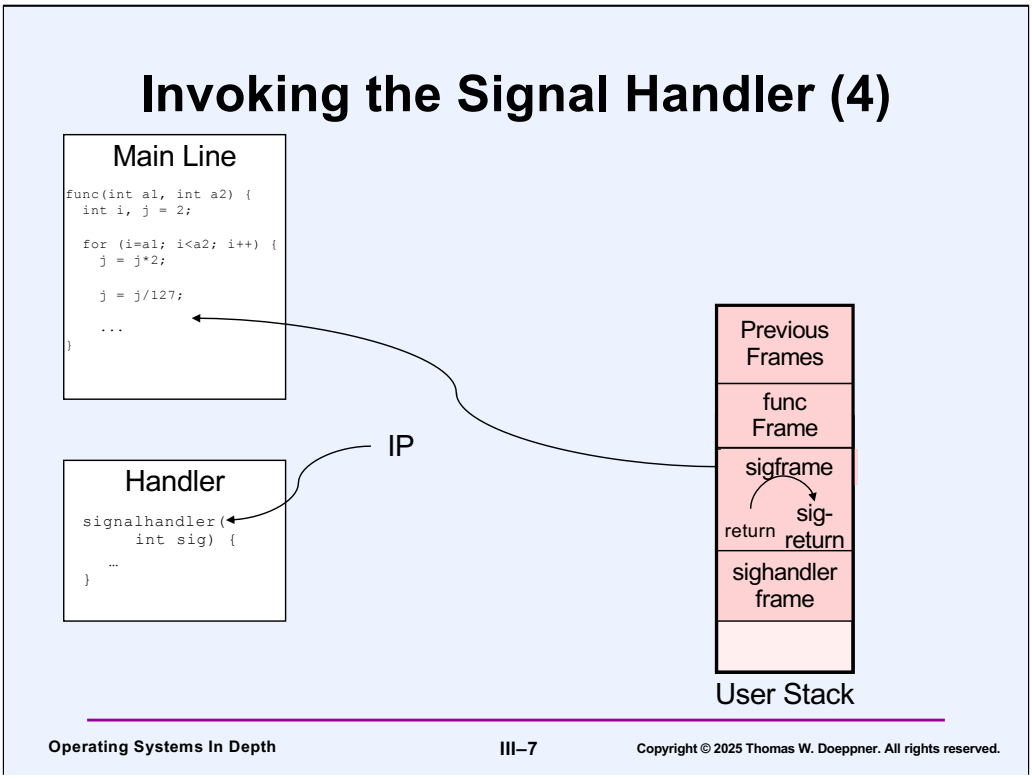
In this sequence of slides, we show how a signal handler is invoked in Linux. In this slide, our program is executing within its "main line."



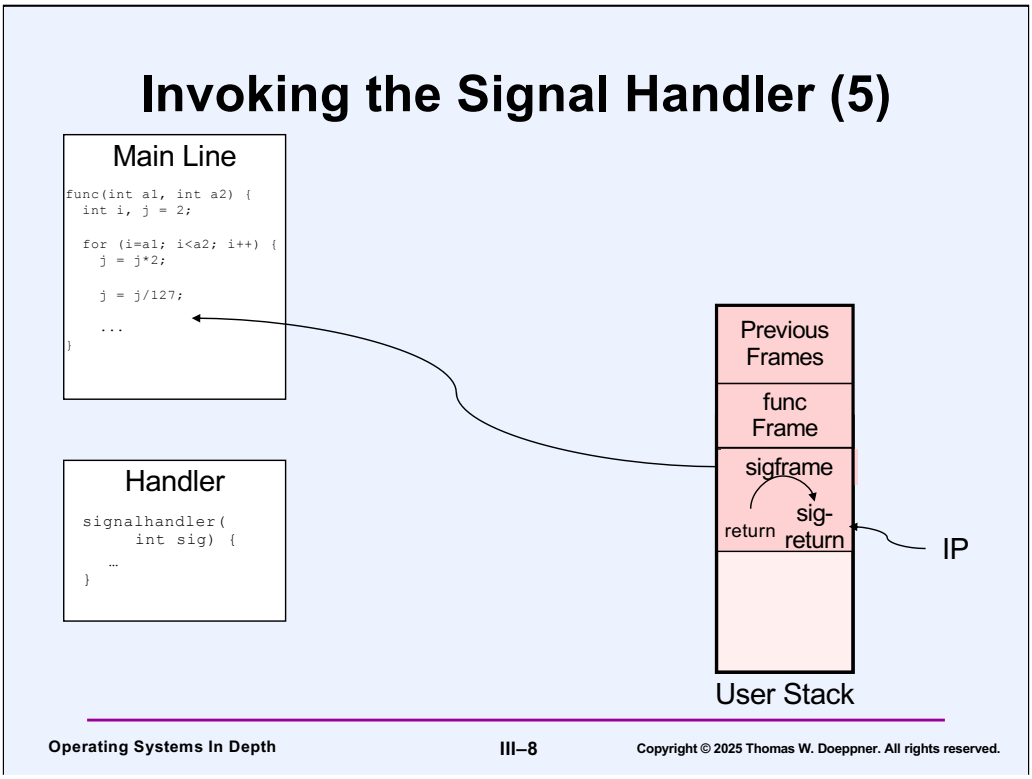
A signal has occurred. This causes the thread to be interrupted, and the kernel is entered. The thread's registers (including its instruction pointer) are saved on the thread's kernel stack.



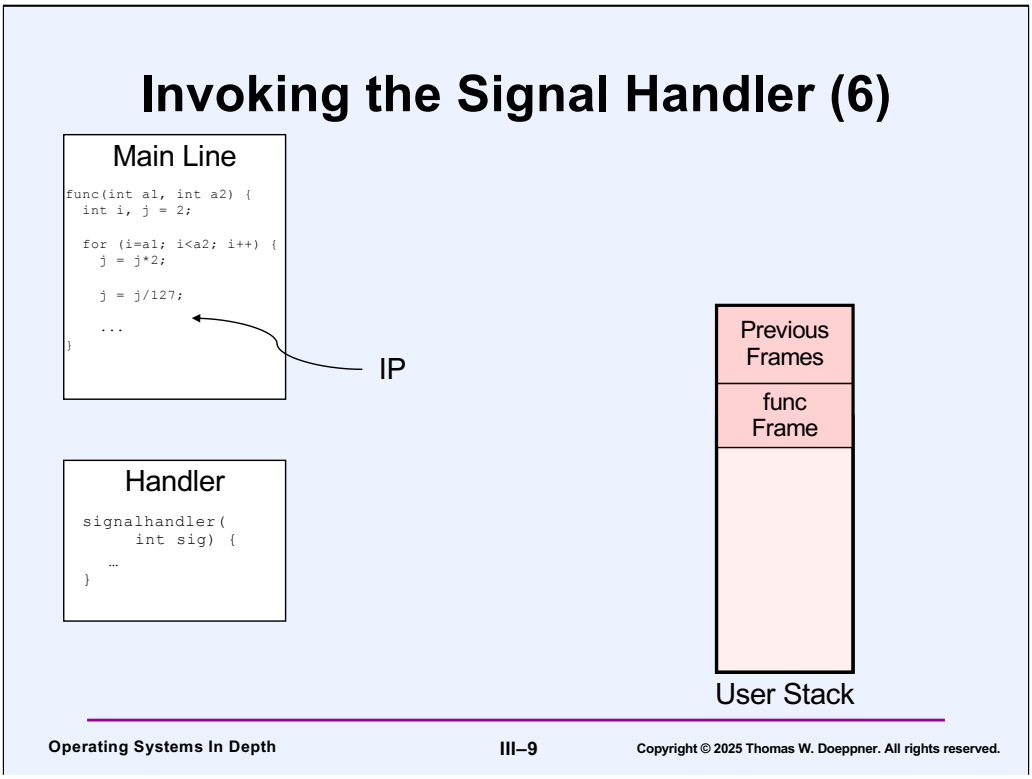
The kernel pushes a **struct sigframe** onto the thread's user-mode stack, containing a copy of the thread's user-mode registers (copied from the kernel stack — it includes the thread's user-mode instruction pointer, which points to the instruction following the point of interruption), the signal mask, and a new return address that points to executable code (also pushed onto the stack). This executable code, which is executed on return from the signal handler, invokes the **sigreturn** system call with the signal mask and the user-mode registers as its arguments. The effect of the call is to return to the point of the interrupt with the signal mask restored.



Now the system returns our thread to user mode, and it resumes execution in the appropriate signal-handling routine.



When the signal handler returns, it returns to the executable code that was pushed onto the stack (within **sigframe**) by the kernel. This code, as already mentioned, invokes the **sigreturn** system call, which passes a pointer to the **sigframe** structure back to the kernel.



The **sigreturn** handler in the kernel restores the user-mode registers and signal mask from the **sigframe** structure and returns back to the point of interrupt.

Quiz 1

The description of invoking the signal handler:

a) works fine

b) is susceptible to buffer-overflow attacks

c) is rendered useless because of an approach for making such attacks more difficult (by turning off execute permission on some memory)

d) doesn't work, period

Time Slicing

- **Periodically**
  - **current thread forced to do a thread yield**

```
void ClockInterrupt(int sig) {
  // SIGVTALRM is now masked
  sigprocmask(SIG_UNBLOCK, &VTALRMmask, 0);
  // SIGVTALRM is now unmasked
  uthread_yield();
  // thread resumes here
}
```

- **Implement ClockInterrupt with VTALRM signal**

When the handler is entered for SIGVTALRM, SIGVTALRM is masked. Since the thread yields and does not return from the handler right away, we need to explicitly unmask the signal, as shown in the slide.

Setting Up Time Slicing

```
struct sigaction timesliceact;
timesliceact.sa_handler = ClockInterrupt;
timesliceact.sa_mask = VTALRMmask;
timesliceact.sa_flags = SA_RESTART; // avoid EINTR
struct timeval interval = {0, 1};   // every microsecond
struct itimerval timerval;
timerval.it_value = interval;
timerval.it_interval = interval;
sigaction(SIGVTALRM, &timesliceact, 0);
setitimer(ITIMER_VIRTUAL, &timerval, 0);
    // time slicing is started!
```


Setting the SA_RESTART flag in the **sigaction** structure ensures that interrupted system calls do not fail with the error code EINTR on return from the signal handler, but instead are automatically restarted.

Async-Signal Safety

- A function is asynchronous-signal safe if it may be used in the handler for an asynchronous signal (such as SIGVTALRM)
  - malloc and free
    - no
  - mutex_lock
    - no
  - read and write
    - yes

Achieving Async-Safety

- The problem: an action in the signal handler interferes with an action in the main-line code
  - while in malloc/free, a signal occurs and the handler calls malloc/free
  - while holding the lock on a mutex, a thread is interrupted and the handler attempts to lock the mutex
- The solution: mask signals while in malloc/free and when holding locks

Caution!

- ***uthread_switch* is not async-signal safe**
  - **it's called from *uthread_yield*, which is called from the signal handler for SIGVTALRM**
  - **must mask signals before calling it (and unmask afterwards)**

Since **uthread_switch** manipulates the run queue, it would cause problems if it's invoked by the handler for SIGVTALRM, which was called while a thread was in **uthread_switch** – it would be interrupted with the run queue in an undefined state, and its invocation from the signal handler would then attempt to manipulate the run queue.

Masking/Unmasking Signals

```
sigset_t VTALRMmask;
…
sigemptyset(&VTALRMmask);
sigaddset(&VTALRMmask, SIGVTALRM);
…
sigprocmask(SIG_BLOCK, &VTALRMmask, 0);
…
sigprocmask(SIG_UNBLOCK, &VTALRMmask, 0);
```


Note that **pthread_sigmask** and **sigprocmask** do exactly the same thing.

Doing It Cheaply

```
void uthread_nopreempt_on() {
  uthread_no_preempt = 1;
}

void uthread_nopreempt_off() {
  uthread_no_preempt = 0;
}

void ClockInterrupt(int sig) {
  if (uthread_no_preempt)
    return;
  …
}
```

```
uthread_no_preempt_on();

uthread_switch();

uthread_no_preempt_off();
```


The solution shown in this slide assumes that preemption is to be off only for calls to **uthread_switch**, and thus whenever **uthread_switch** returns, preemption can be enabled.

Note that we are now using the function names used in the uthreads stencil: for example, **uthread_switch** rather than **thread_switch**.

Nested Calls

```
void uthread_nopreempt_on() {
  uthread_no_preempt_count++;
}

void uthread_nopreempt_off() {
  uthread_no_preempt_count--;
}

void ClockInterrupt(int sig) {
  if (uthread_no_preempt_count > 0)
    return;
  …
}
```


In some cases, functions might need to disable preemption before calling functions that, in turn, call **uthread_switch**. This might be because they do some sort of manipulation of the run queue before calling **uthread_switch**, and must have the run queue protected until after **uthread_switch** is called.

Corrected Nested Calls

```
void uthread_nopreempt_on() {
  ut_curthr->uthread_no_preempt_count++;
}

void uthread_nopreempt_off() {
  ut_curthr->uthread_no_preempt_count--;
}

void ClockInterrupt(int sig) {
  if (ut_curthr->uthread_no_preempt_count > 0)
    return;
  …
}
```


The preemption count (**uthread_no_preempt_count**) is really a property of the individual thread and thus should be part of **uthread_t**. Note that in the uthreads stencil, the current thread is referred to by the global variable **ut_curthr**.

Limitations of User Threads

- **Threads are implemented strictly at user level**
  - **the OS kernel is unaware of their existence**
- **What happens if a user thread makes a blocking system call, e.g., *read*?**

If a user-level-implemented thread makes a system call that blocks (i.e., causes the thread to go to sleep), then there's no opportunity for a thread switch to some other user-level thread. The user-level threads are multiplexed on a single kernel (OS) thread, and if that thread blocks, then, effectively, all user-level threads block.

In many situations there are ways around this problem (in particular, one can determine for terminal- and socket-oriented I/O whether an operation will block, then arrange to wait on a condition variable until I/O is possible without blocking). However, we won't be doing this in either the uthreads or the mthreads assignments.

Quiz 2

```
void uthread_switch( ) {
   volatile int first = 1;
   getcontext(&CurrentThread->ctx);
   if (first) {
      first = 0;
      CurrentThread = dequeue(RunQueue);
      setcontext(&CurrentThread->ctx);
   }
   return;
}
```

Given the discussion so far, will RunQueue ever be empty (in a program that has no deadlocks)?

a) yes

b) no

Note that we don't (yet) have a means for terminating threads.

Multiple Processors

```
void uthread_switch( ) {
   volatile int first = 1;
   getcontext(&CurrentThread->ctx);
   if (first) {
      first = 0;
      CurrentThread = dequeue(RunQueue);
      setcontext(&CurrentThread->ctx);
   }
   return;
}
```

- How do we employ multiple processors?
  - code merely switches the caller's processor to another thread
- What if the RunQueue is empty?
  - it could be empty, particularly if we have multiple processors

Solution Sketch

- Introduce “idle threads”, one for each processor
- Thread calling *uthread_switch* switches to idle thread for its current processor
- Idle thread then switches to first thread on *RunQueue*, if any
- If *RunQueue* is empty, idle thread repeatedly checks *RunQueue* until it’s not empty, then switches to first thread

Solution Details

```
1  void uthread_switch() {
2    volatile int first = 1;
3    getcontext(&CurrentThread[processor_ID]->context);
4    if (!first)
5      return;
6    first = 0;
7    setcontext(&IdleThread[processor_ID]->context);
8  }

9  void IdleThread_switch() {
10   getcontext(&IdleThread[processor_ID]->context);
11   while (1) {
12     if (queue_empty(RunQueue))
13         continue;
14     CurrentThread[processor_ID] = dequeue(RunQueue);
15     setcontext(&CurrentThread[processor_ID]->context);
16   }
17 }
```


In this slide, **getcontext** saves the thread's register context in the area pointed to by its argument. **setcontext** loads the registers with the context that's located in the area pointed to by its argument. **processor_ID** is private to each idle thread, i.e., each idle thread has a separate copy of it; it contains the ID of the processor represented by that thread.

This code is deceptively simple-looking. Each idle thread starts off by calling **IdleThread_switch**. It saves its context at line 10, and then goes on to repeatedly check the **RunQueue**. When one finds a thread on the **RunQueue**, it switches to that thread's context in line 15.

Threads calling **uthread_switch** first save their registers by calling **getcontext** at line 3. The variable **first** (declared **volatile** to force gcc not to put it in a register, but to leave it in memory) is set to 1 so that, on the (first) return from **getcontext**, its value is still 1 and the thread continues executing sequentially. At line 7 the call to **setcontext** switches to the registers of the processor's idle thread. That context was saved in line 10, and thus control continues at line 11, in the context of idle thread, as explained in the preceding paragraph.

When an idle thread switches to a normal thread (by calling **setcontext** at line 15), the normal thread resumes execution starting from where it had last saved its context, at line 3. Thus the thread continues at line 4. At this point, **first**, its local variable, now has a value of zero, and thus the thread returns from its original call to **uthread_switch**.

MThreads

- Idle threads are pthreads
  - called LWPs (lightweight processes), following standard usage
  - each LWP represents a processor (core)
    - a "virtual processor"
  - *lwp_switch* rather than *IdleThread_switch*
  - rather than "busy-wait" for a uthread to run, it calls *lwp_park* to wait for one (using a POSIX condition variable)

For the uthreads assignment, we had multiple uthreads being multiplexed on just one thread, the only thread in the process. But for the mthreads assignment, we have multiple uthreads being multiplexed on a number of idle threads (now call LWPs), with each LWP representing a processor.

MP Mutual Exclusion

- Two sorts
  - spin locks
    - threads wait by repeatedly testing the lock
  - blocking locks
    - threads wait by sleeping, depending on some other thread to wake them up

Note that our earlier implementation of mutexes assumed there was just one processor.

Hardware Support for Spin Locks

- **Compare and swap instruction**

```
int CAS(int *ptr, int old, int new) {
   int tmp = *ptr;
   if (*ptr == old)
      *ptr = new;
   return tmp;
}
```


The slide uses C code to show what the **CAS** (compare and swap) instruction does. Note that the **CAS** instruction is implemented (in hardware) as an atomic instruction: its effect is effectively instantaneous: nothing else can affect memory during the execution of the instruction. In the x86 and x86-64 architectures, the effect of **CAS** is achieved with the **cmpxchng** (compare and exchange) instruction with the lock prefix.

Naive Spin Lock

```
void spin_lock(int *spin) {
   while(CAS(spin, 0, 1))
      ;
}

void spin_unlock(int *spin) {
   *spin = 0;
}
```


The problem with this code is that each waiting thread repeatedly executes CAS, which, as explained with the previous slide, prevents all other access to memory for a brief period. Thus such repeated calls to CAS slows the system.

Better Spin Lock

```
void spin_lock(int *spin) {
   while (1) {
      if (*spin== 0) {
         // the mutex was at least momentarily unlocked
         if (!CAS(spin, 0, 1)
            break;  // we have locked the mutex
         // some other thread beat us to it, so try again
      }
   }
}
```

Spin Locks in MThreads

- Since LWPs represent virtual processors, we don't want busy waiting
- Use POSIX mutexes instead

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 3

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 4
Does it work?

a) Yes
b) No

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

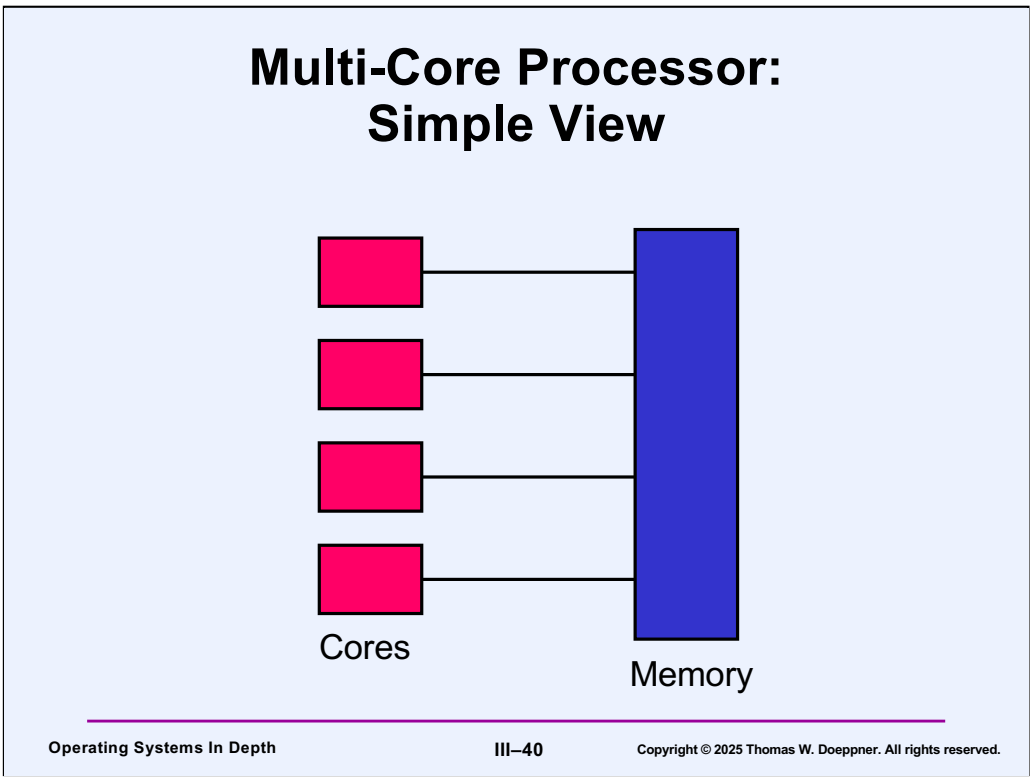
- We could use futexes, but don't
- *uthread_switch* gets an additional argument
  - a POSIX mutex (representing a spin lock)
  - unlock it after getting out of the context of the calling thread

Actual Code

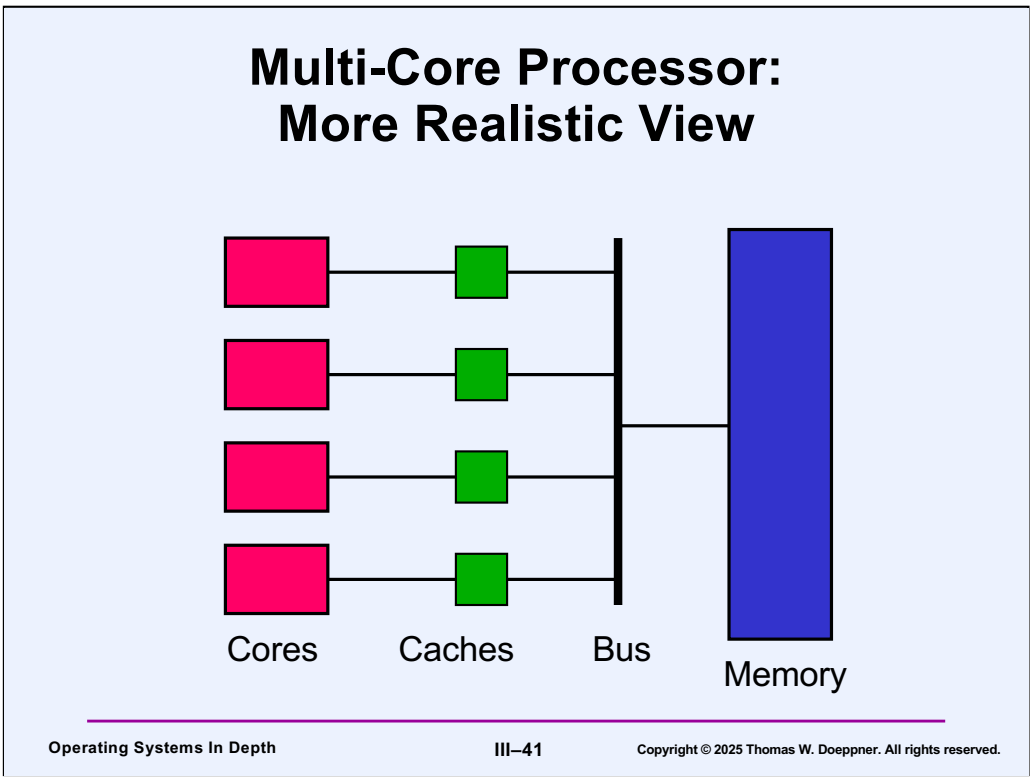
```
uthread_mtx_lock(uthread_mtx_t *mtx) {
    uthread_nopreempt_on();
    pthread_mutex_lock(&mtx->m_pmut);
    if (mtx->m_owner == NULL) {
        mtx->m_owner = ut_curthr;
        pthread_mutex_unlock(&mtx->m_pmut);
        uthread_nopreempt_off();
    } else {
        ut_curthr->ut_state = UT_WAIT;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

MP Memory Issues

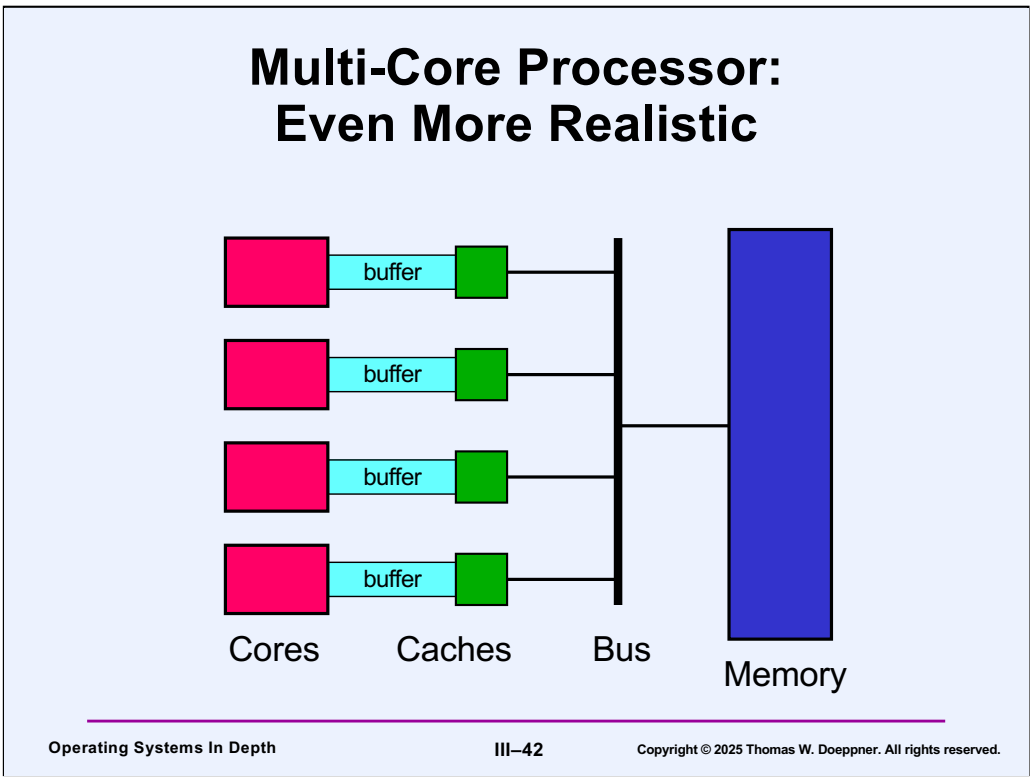
- Naive view is that all processors in MP system see same memory contents at all times
  - they don't



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.



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.



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;
}
```


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);
}
```


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.