Interrupts, Etc.

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).

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);
   …
}
```


The call to MaskInterrupts masks clock interrutps and X interrupts (as well as probably most all other interrupts).

The call to UnMaskInterrupts restores the previous interrupt mask.

Quiz 1

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) **In addition, threads must lock (blocking) mutexes before masking interrupts and accessing *X***

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

d) **In addition to c, the dev interrupt handler must lock a spinlock before accessing *X***

e) **Something else is needed**

Deferred Work

- Interrupt handlers run with interrupts masked
  – may interfere with handling of other interrupts, particularly if they do a lot of computation
- Solution
  – do minimal work now
  – do rest later without interrupts masked

Deferred Processing

```
void TopLevelInterruptHandler(int dev) {
  InterruptVector[dev](); // call appropriate handler
  if (PreviousContext == ThreadContext) {
    UnMaskInterrupts();
    while(!Empty(WorkQueue)) {
      Work = DeQueue(WorkQueue);
      Work();
    }
  }
}

void NetworkInterruptHandler() {
  // deal with interrupt
  …
  EnQueue(WorkQueue, MoreWork);
}
```

Windows Interrupt Priority Levels

hardware	31	High
	30	Power fail
	29	Inter-processor
	28	Clock
	. . .	
	4	Device 2
	3	Device 1
software	2	DPC
	1	APC
	0	Thread

The slide shows the interrupt priority levels used in Windows (which calls them *interrupt request levels*). At any particular moment, the processor is running at a particular interrupt level, and all interrupts at equal and lower levels are masked.

Deferred Procedure Calls

```
void InterruptHandler( ) {
  // deal with interrupt
  …
  QueueDPC(MoreWork, arg);
  /* enqueues MoreWork on
     the DPC queue and
     requests a DPC
     interrupt
  */
}
```

```
void DPCHandler( … ) {
  while(!Empty(DPCQueue)) {
    Work = DeQueue(DPCQueue);
    Work();
  }
}
```


Deferred Procedure Calls (DPCs) are a concept used by Windows.

Software Interrupt Threads

```
void InterruptHandler() {
   // deal with interrupt
   …
   EnQueue(WorkQueue,
         MoreWork);
   SetEvent(Work);
}
```

```
void SoftwareInterruptThread() {
   while(TRUE) {
      WaitEvent(Work)
      while(!Empty(WorkQueue)) {
         Work = DeQueue(
            WorkQueue);
         Work();
      }
   }
}
```


Software interrupts are a means for handling deferred work on Unix systems. Linux uses one software interrupt thread per processor.

Preemption: User-Level Only

```
void ClockHandler() {
  // deal with clock interrupt
  …
  if (TimeSliceOver())
    ShouldReschedule = 1;
}
```

```
void TopLevelInterruptHandler(int dev) {
  InterruptVector[dev]();
  if (PreviousMode == UserMode) {
    // the clock interrupted user-mode code
      if (ShouldReschedule)
        Reschedule();
  }
  …
}

void TopLevelTrapHandler(…) {
  SpecificTrapHandler();
  …
  if (ShouldReschedule) {
    /* the time slice expired while the thread
       was in kernel mode */
    Reschedule();
  }
}
```


Note the distinction between interrupts and traps. Interrupts are actions caused by entities other than the current thread. Traps are caused by the current thread – an example is system calls.

Preemption: Full

```
void ClockInterruptHandler( ) {
  // deal with clock interrupt
  …
  if (TimeSliceOver)
    QueueDPC(Reschedule);
}
```


What's important is that the call to Reschedule occurs after all other interrupt handling has been dealt with.

Directed Processing

- Signals: Unix
  - perform given action in context of a particular thread in user mode
- APC: Windows asynchronous procedure calls
  - roughly same thing, but also may be done in kernel mode

Asynchronous Procedure Calls

- Two uses
  - kernel APC: release of kernel resources
  - user APC: notifying a thread of an external event

Asynchronous Procedure Calls (APCs) are another Windows concept.

Kernel APC

- Release of kernel resources
  - interrupt handler has information that must be copied to user process
  - can't be done unless in context of process
    - otherwise address space not mapped in
  - interrupt handler requests kernel APC to have user thread, running in kernel mode, copy information to user space, and then free data in the kernel

User APC

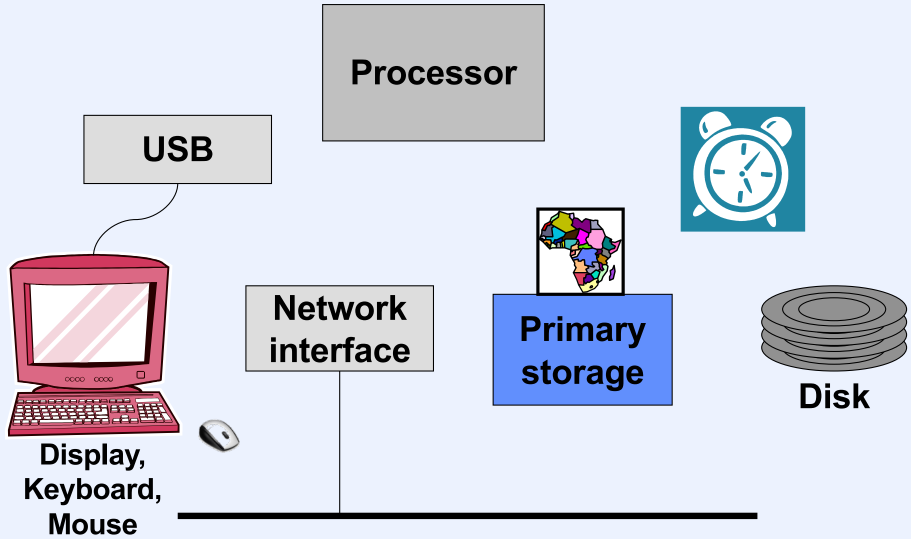
- Notifying thread of external event
  - example: asynchronous I/O
    - thread supplies *completion routine* when starting asynchronous I/O request
    - called in thread's context when I/O completes
      - similar to a Unix signal
      - called only when thread is in *alertable wait state*
        - an option in certain blocking system calls

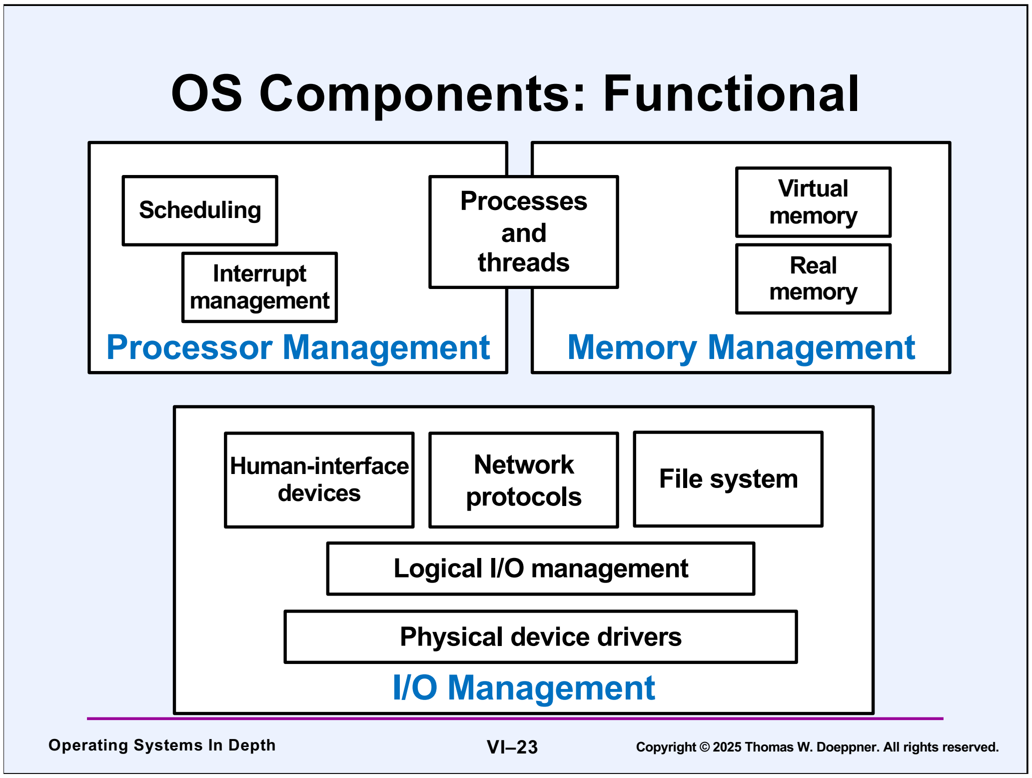
APC Implementation

- Per-thread list of pending APCs
  - on notification, thread executes them
- User APC
  - thread in alertable state is woken up and executes pending APCs when it returns to user mode
- Kernel APC
  - running thread interrupted by APC interrupt (lowest-priority interrupt)
  - waiting thread is “unwaited”
  - execute pending kernel APCs

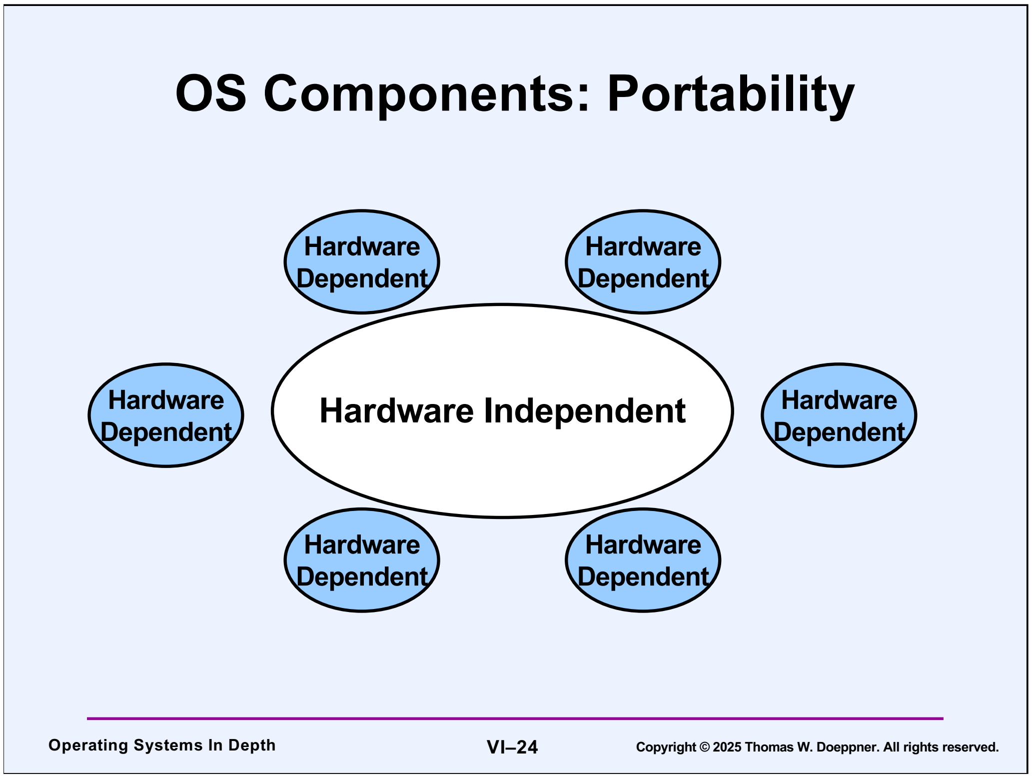
I/O

Simple Configuration



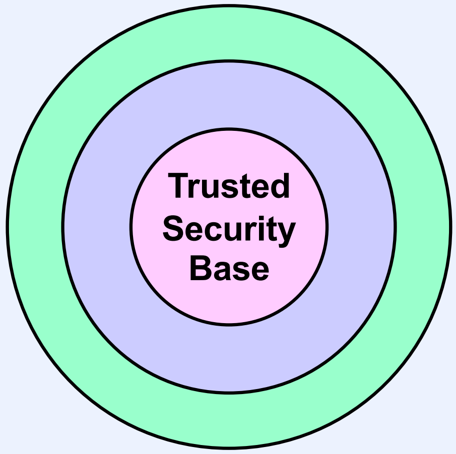


This diagram is not intended to be all-inclusive, but suggestive of what the major OS components are.

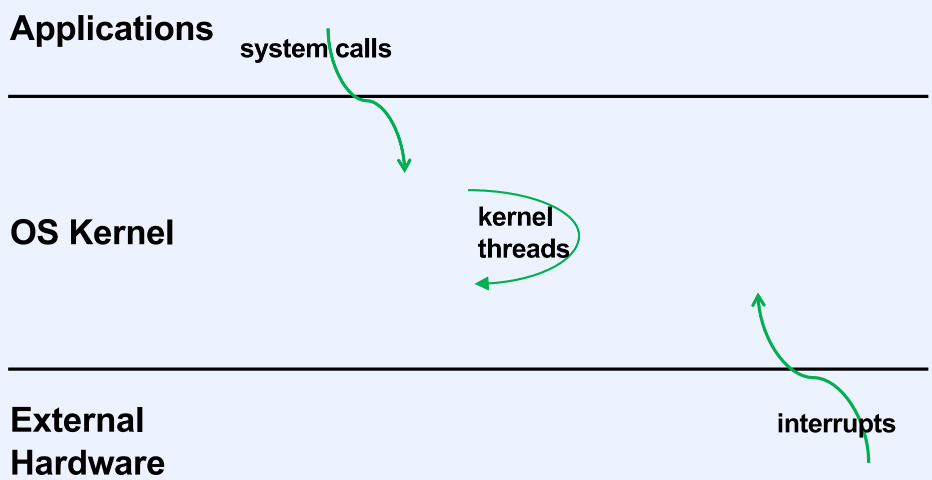


Examples of hardware-dependent portions of an OS are device drivers, interrupt management, power management, etc.

OS Components: Importance

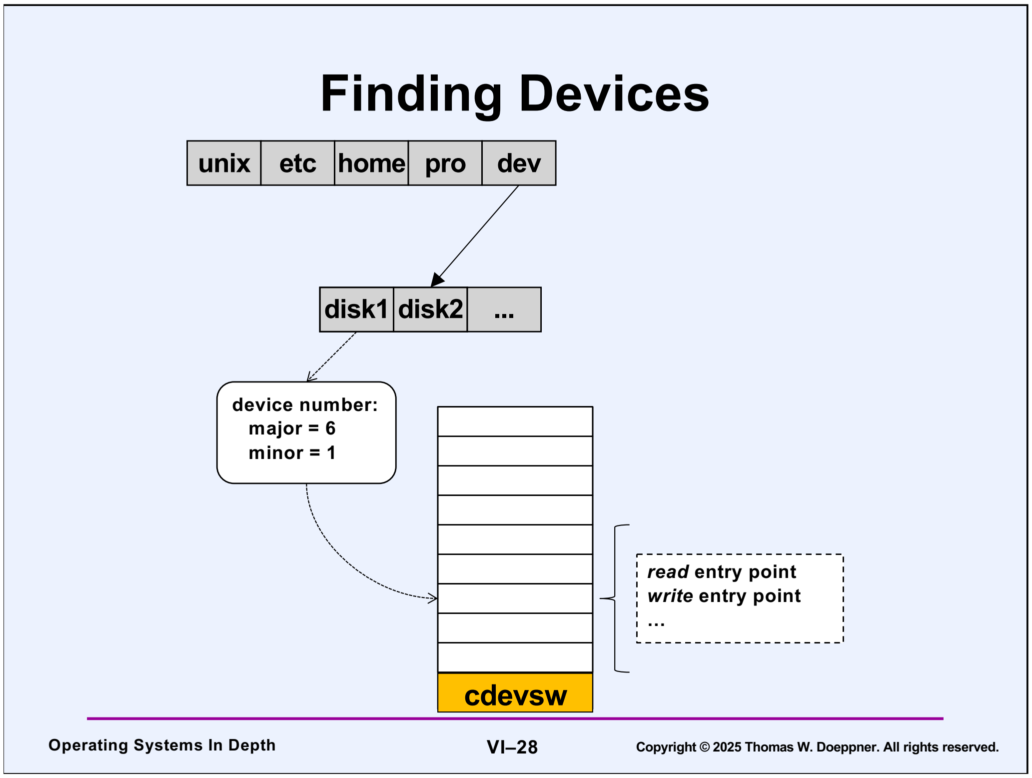


OS Components: Flow of Control



To Be Discussed

- What is the functionality of the components?
- What are the key data structures?
- How is the system broken up into modules?
- To what extent is the system extensible?
- What parts run in the OS kernel in privileged mode? What parts run as library code in user applications? What parts run as separate applications?
- In which execution contexts do the various activities take place?



This is the architecture for representing devices in traditional Unix systems. A somewhat different approach is used in Weenix, which is discussed in an upcoming lecture.

Discovering Devices

- You plug in a new device to your computer …
  - OS must notice
    - must find a device driver
      - what kind of device is it?
      - where is the driver?
    - must assign a name
      - how chosen?
    - multiple similar devices
      - how does application choose?

Computer Terminal

A “tty”

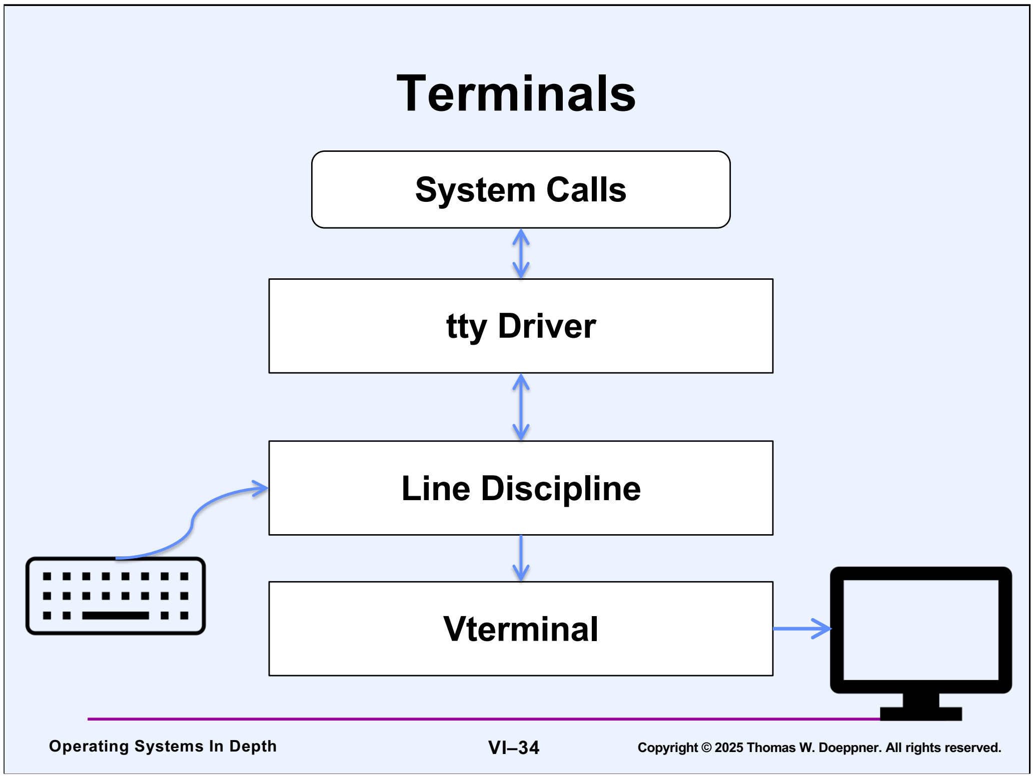
The photo shows a Teletype terminal being used on an early Unix system by Ken Thompson, one of the two co-developers of Unix. (Dennis Ritchie, the other co-developer of Unix, is standing.) The photo is from http://histoire.info.free.fr/images/pdp11-unix.jpeg, but it is probably owned by Lucent Technologies. “Teletype” is the word from which tty is derived. (According to Wikipedia (August 25, 2010) “Teletype” was a trademark of the Teletype Corporation, but the company no longer exists.)

A Typewriter

The image is from http://filmdoctor.co.uk/2013/03/26/the-brit-list-2013/.

Terminals

- Long obsolete, but still relevant
- Issues
  1) characters are generated by the application faster than they can be sent to the terminal
  2) characters arrive from the keyboard even though there isn't a waiting read request from an application
  3) input characters may need to be processed in some way before they reach the application



Physical terminals are rarely used these days. What we discuss here is how they are implemented in weenix.

The tty driver (kernel/drivers/tty/tty.c) provides an interface to the rest of the kernel. Thus, read and write system calls dealing with terminal I/O go to it. Each terminal is represented by a **tty_t** object, which includes an **ldisc_t** object that contains a circular buffer to hold incoming characters (from the keyboard).

Incoming characters are given to the line discipline code (kernel/drivers/tty/ldisc.c) for processing. They are inserted into the buffer. Until a linefeed (return) is received, incoming characters can be edited using backspace. Once the linefeed has been received, no more editing can be done, and the line of characters are transferred to a list that may be consumed by a thread doing a read system call. Characters that can still be edited (because a subsequent linefeed hasn't been received) are called **raw characters**. Characters that can no longer be edited are called **cooked characters**.

When characters are outputted, either because they are echoed as they are received from the keyboard, or are written via a write system call, they are sent to the vterminal code (kernel/drivers/tty/vterminal.c), where processing is done to have them displayed on a display device (this code is provided to you).

Quiz 2

In which context are characters transformed from raw into cooked?

a) **In the interrupt context (i.e., on a "borrowed" stack)**

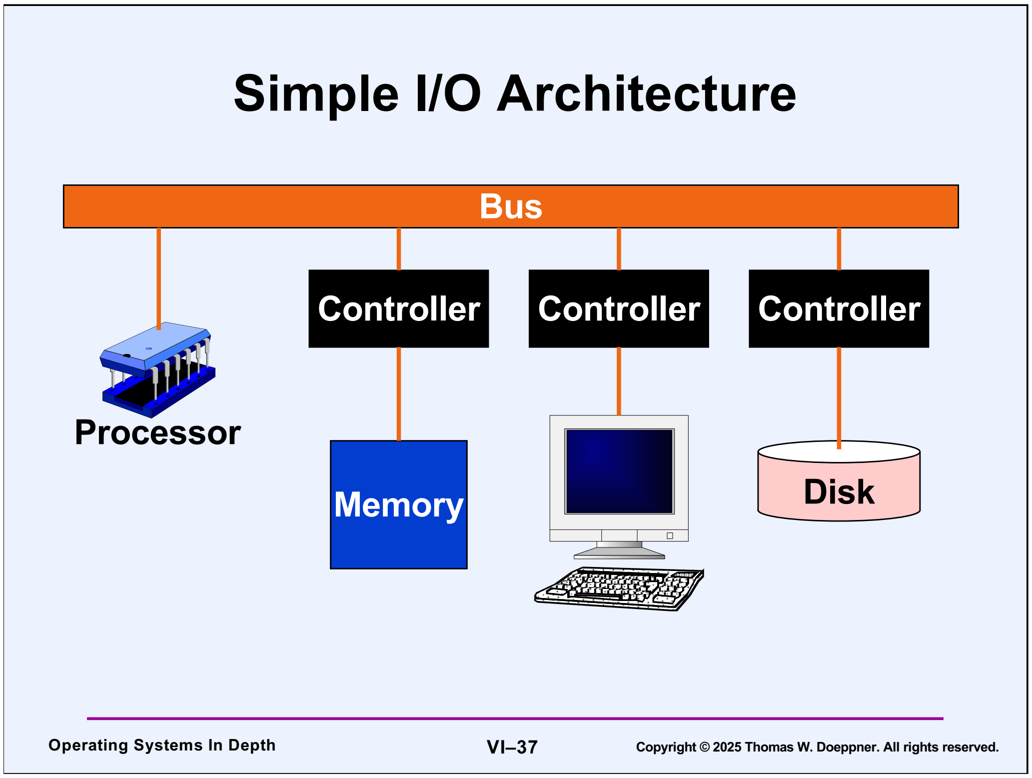
b) **In the context of the thread performing the *read* system call**

c) **Some other context**

Input/Output

- Architectural concerns
  - memory-mapped I/O
    - programmed I/O (PIO)
    - direct memory access (DMA)
  - I/O processors (channels)
- Software concerns
  - device drivers
  - concurrency of I/O and computation

In this section we address the area of input and output (I/O). We discuss two basic I/O architectures and talk about the fundamental I/O-related portion of an operating system—the **device driver**.



A very simple I/O architecture is the **memory-mapped** architecture. Each device is controlled by a controller and each controller contains a set of registers for monitoring and controlling its operation. In the memory-mapped approach, these registers appear to the processor as if they occupied physical memory locations. In reality, each of the controllers is connected to a **bus**. When the processor wants to access or modify a particular location, it broadcasts the address on the bus. Each controller listens for a fixed set of addresses and, if it finds that one of its addresses has been broadcast, then it pays attention to what the processor would like to have done, e.g., read the data at a particular location or modify the data at a particular location. The memory controller is a special case. It passes the bus requests to the actual primary memory. The other controllers respond to far fewer addresses, and the effect of reading and writing is to access and modify the various controller registers.

There are two categories of devices, **programmed I/O** (PIO) devices and **direct memory access** (DMA) devices. In the former, I/O is performed by reading or writing data in the controller registers a byte or word at a time. In the latter, the controller itself performs the I/O: the processor puts a description of the desired I/O operation into the controller's registers, then the controller takes over and transfers data between a device and primary memory.

PIO Registers

GoR	GoW	IER	IEW					Control register
RdyR	RdyW							Status register
								Read register
								Write register

Legend:

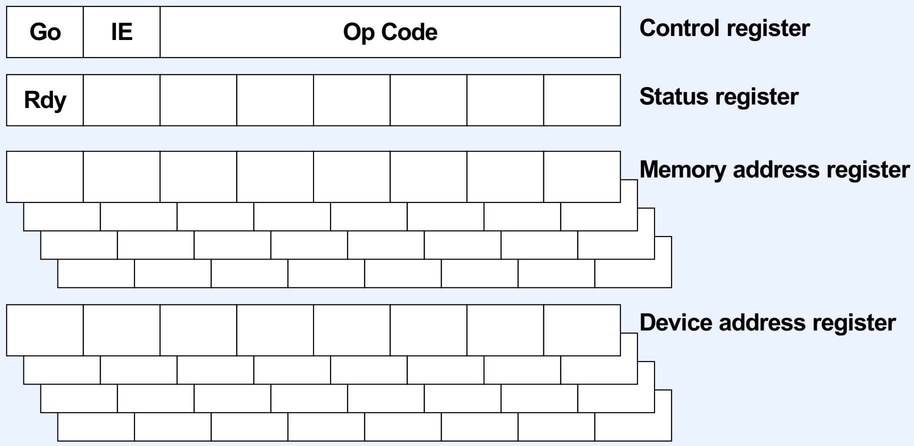
GoR	Go read (start a read operation)
GoW	Go write (start a write operation)
IER	Enable read-completion interrupts
IEW	Enable write-completion interrupts
RdyR	Ready to read
RdyW	Ready to write

Programmed I/O

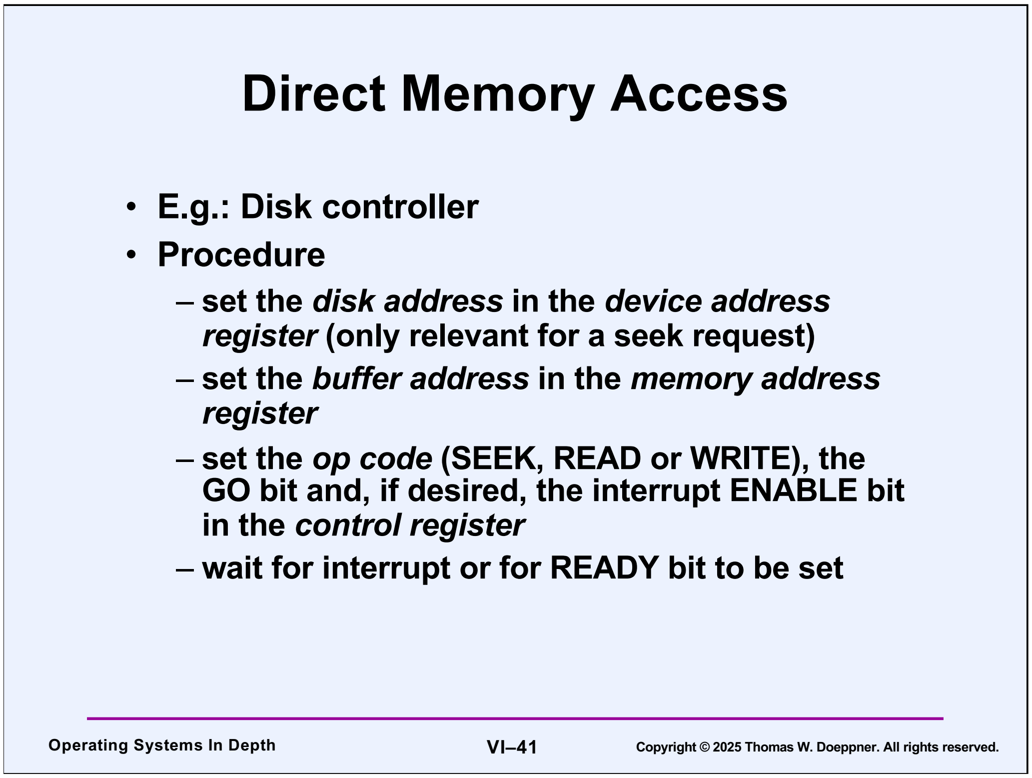
- E.g.: Terminal controller
- Procedure (write)
  - write a byte into the *write register*
  - set the WGO bit in the *control register*
  - wait for WREADY bit (in *status register*) to be set (if interrupts have been enabled, an interrupt occurs when this happens)

The sequence of operations necessary for performing PIO is outlined in the picture. One may choose to perform I/O with interrupts **disabled**, you must check to see if I/O has completed by testing the ready bit. If you perform I/O with interrupts **enabled**, then an interrupt occurs when the operation is complete. The primary disadvantage of the former technique is that the ready bit is typically checked many times before it is discovered to be set.

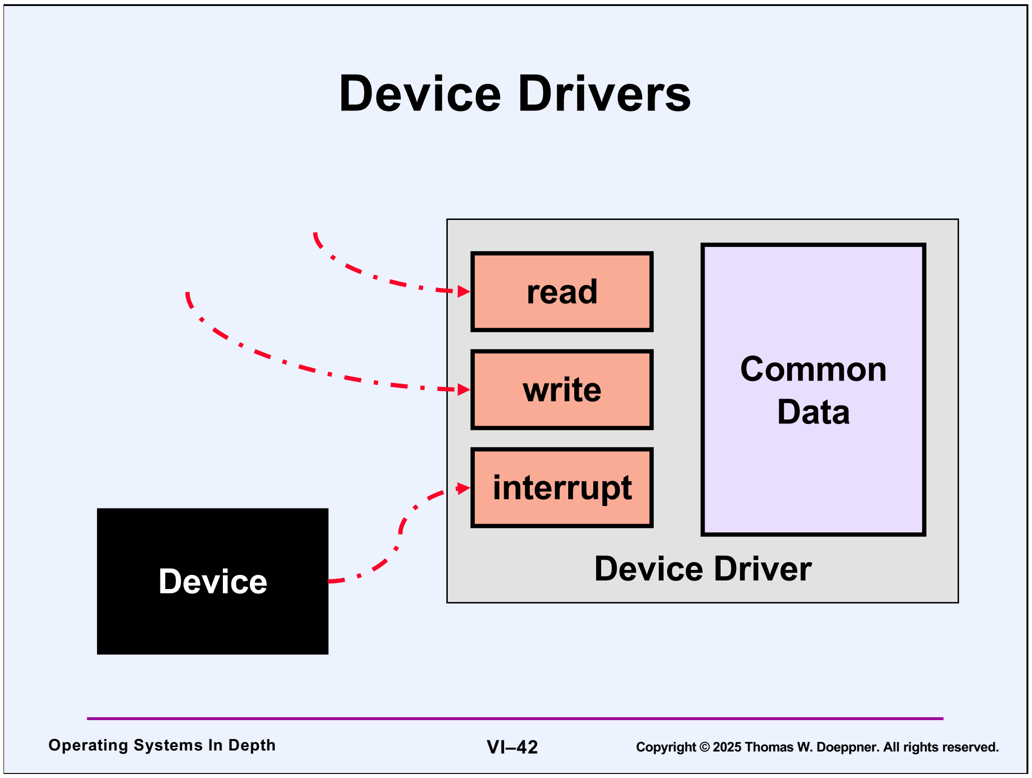
DMA Registers



Legend:		
	Go	Start an operation
	Op Code	Operation code (identifies the operation)
	IE	Enable interrupts
	Rdy	Controller is ready



For I/O to a DMA device, one must put a description of the desired operation into the controller registers. A disk request on the simulator typically requires two operations: one must first perform a *seek* to establish the location on disk from or to which the transfer will take place. The second step is the actual transfer, which specifies that location in primary memory to or from which the transfer will take place.



A device driver is a software module responsible for a particular device or class of devices. It resides in the lowest layers of an operating system and provides an interface to other layers that is device-independent. That is, the device driver is the only piece of software that is concerned about the details of particular devices. The higher layers of the operating system need only pass on read and write requests, leaving the details to the driver. The driver is also responsible for dealing with interrupts that come from its devices.