I/O (2)

Not a Quiz

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

Computer Terminal

A “tty”

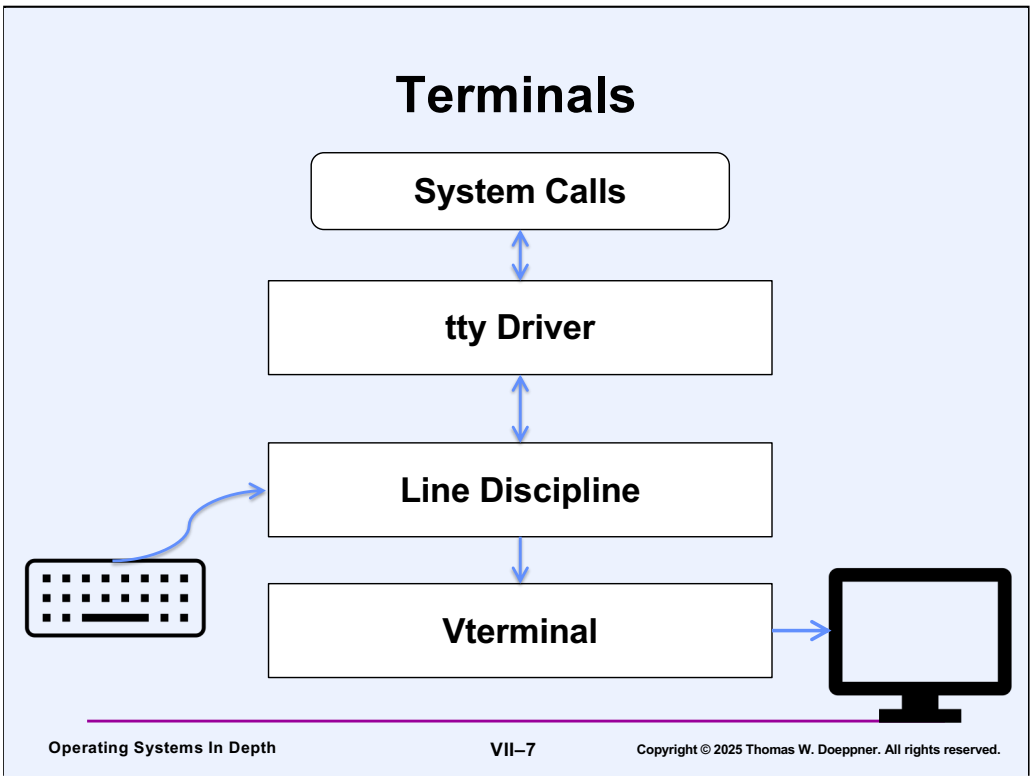
The photo shows a Teletype terminal being used on an early Unix system by Ken Thompson, one of the two co-developers of Unix. (Dennis Ritchie, the other co-developer of Unix, is standing.) The photo is from http://histoire.info.free.fr/images/pdp11-unix.jpeg, but it is probably owned by Lucent Technologies. “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

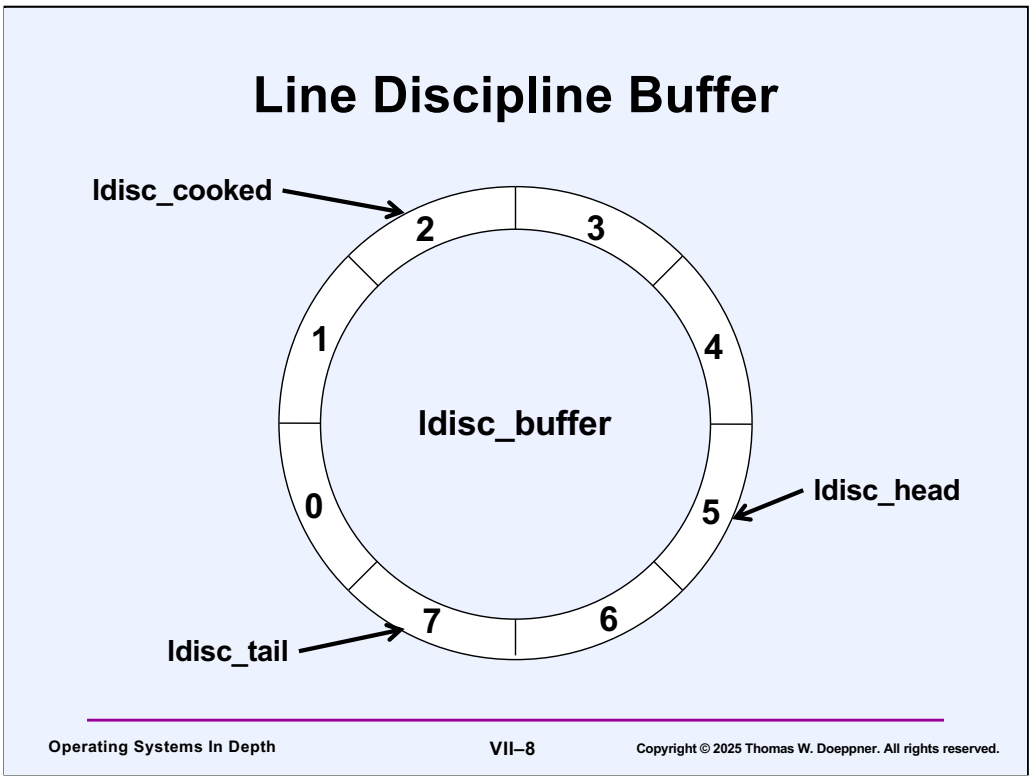


Physical terminals are rarely used these days, but their terminology and concepts persist. In the paragraphs below, the file names refer to source code for the Weenix OS that we provide to you.

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,** which 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 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 may be input 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 for you).



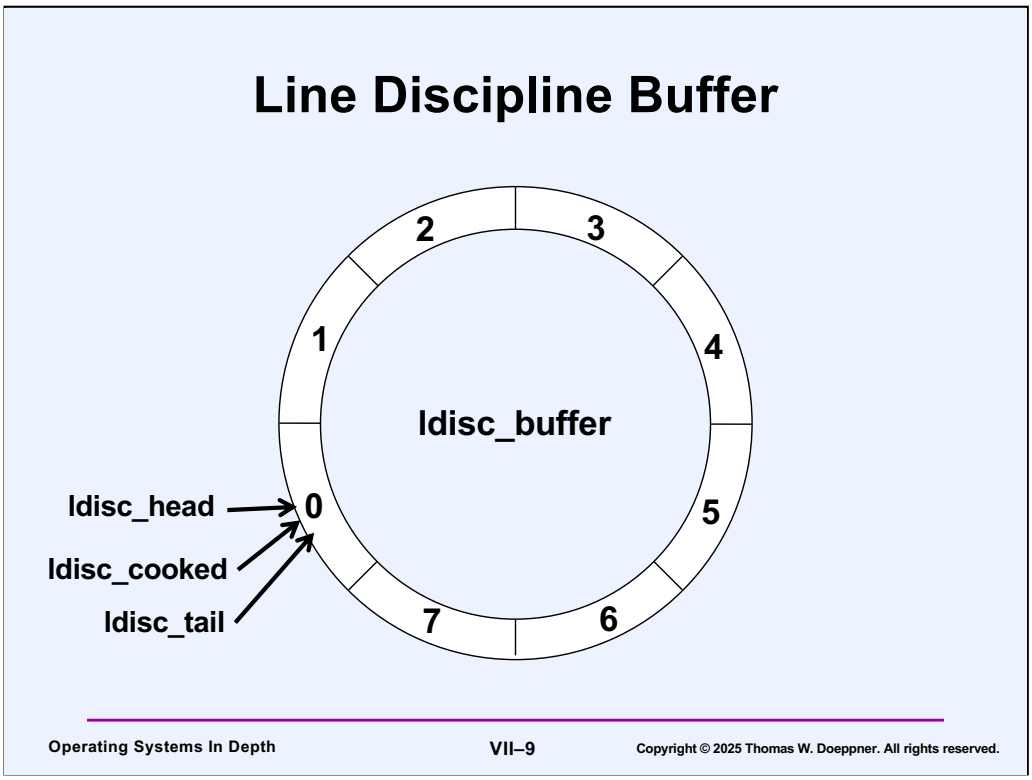
Each terminal has a buffer, **ldisc_buffer**, in its **ldisc_t** (the default size is not 8, but 128). **ldisc_tail** is the index of the unread cooked character that's been in the buffer the longest. **ldisc_head** is the index of the first empty slot in the buffer. **ldisc_cooked** is the index of the most recent cooked character. Thus the characters from **ldisc_tail** through **ldisc_cooked** are the cooked characters -- they may be consumed via a **read** system call. The characters from just after **ldisc_cooked** to just before **ldisc_head** are the raw characters and thus may be edited (but not yet consumed).

When a character comes in from the keyboard (i.e., it's typed), it's put into **ldisc_buffer[ldisc_head]**, and then **ldisc_head** is incremented by one.

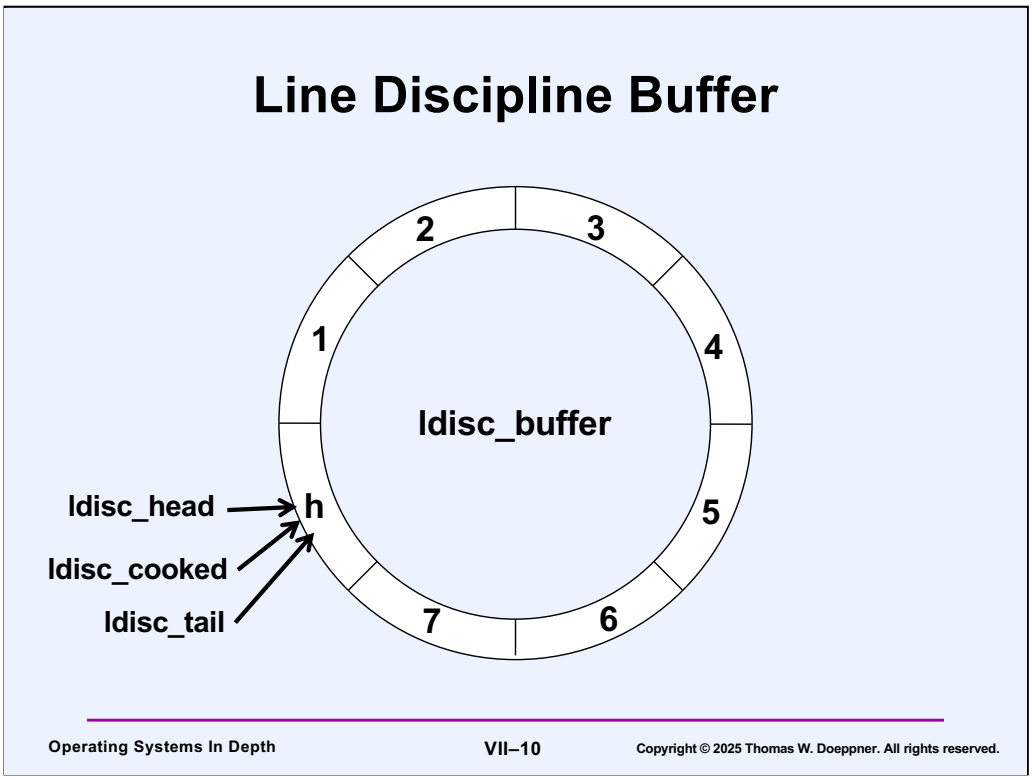
If the buffer is full, then **ldisc_head** is equal to **ldisc_tail** (the next input character would overwrite the oldest unread character). However, the two are also equal if the buffer is empty. An additional member, **ldisc_full** is one if and only if the buffer is full.

If a character arrives when the buffer is full – it is ignored – there's no space for it.

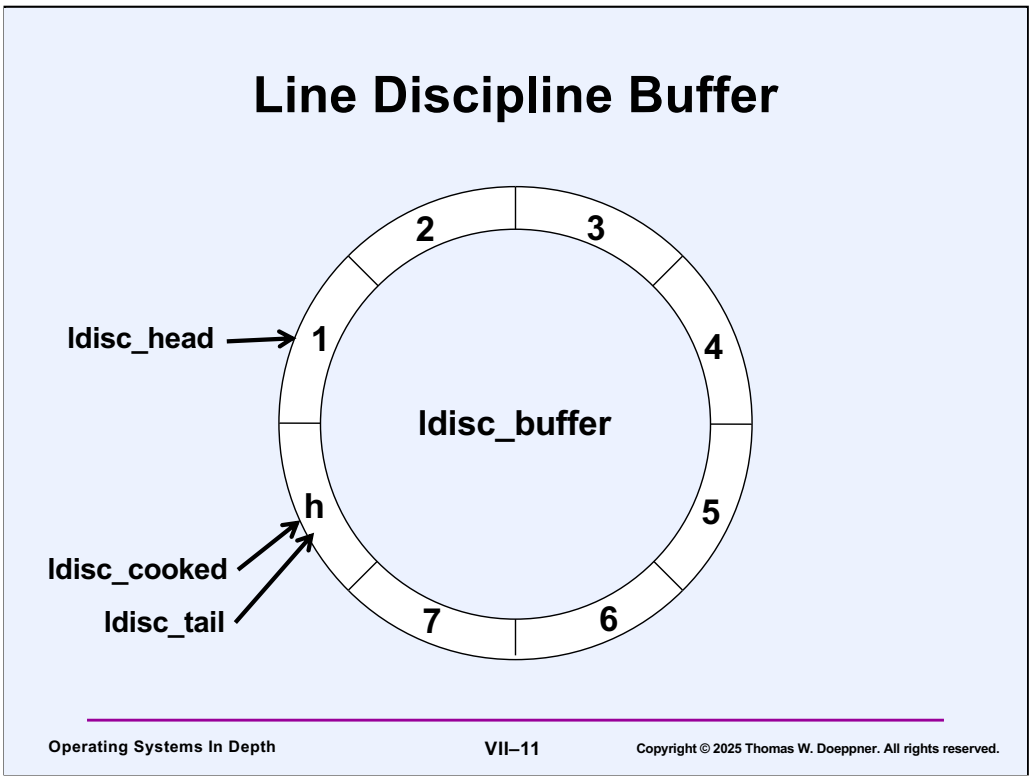
Characters are consumed (via the read system call) starting at **ldisc_buffer[ldisc_tail].** After each character is consumed. **ldisc_tail** is incremented by one. The buffer contains no cooked characters if **ldisc_tail** is equal to **ldisc_cooked.**



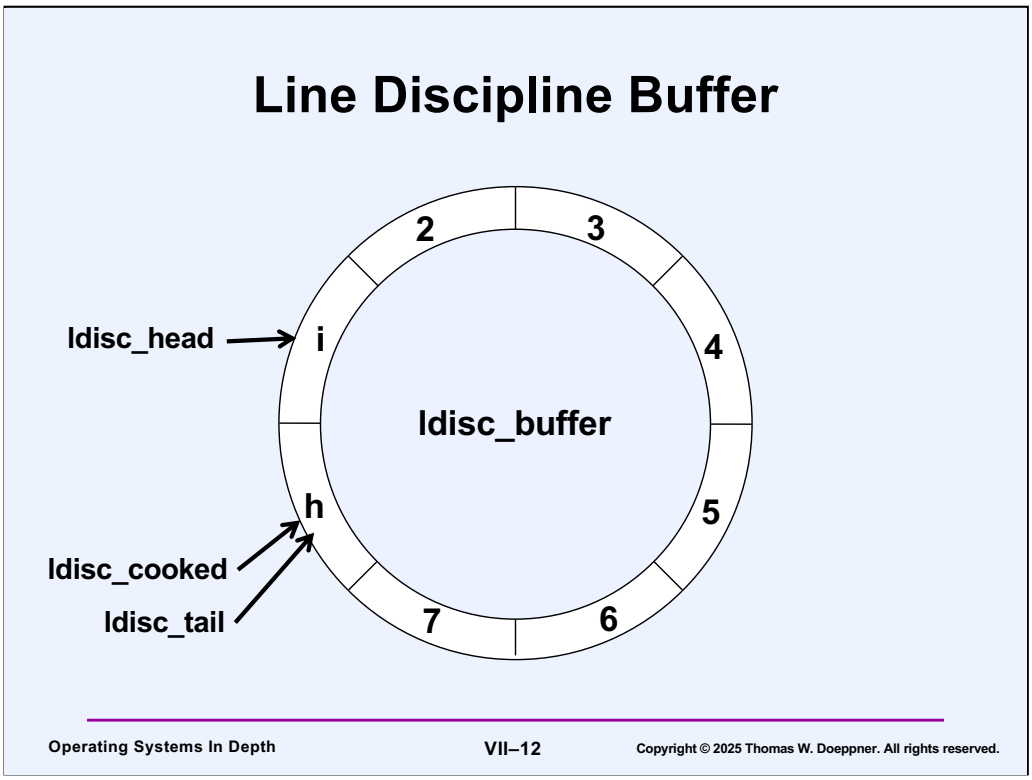
Here we have an empty buffer.



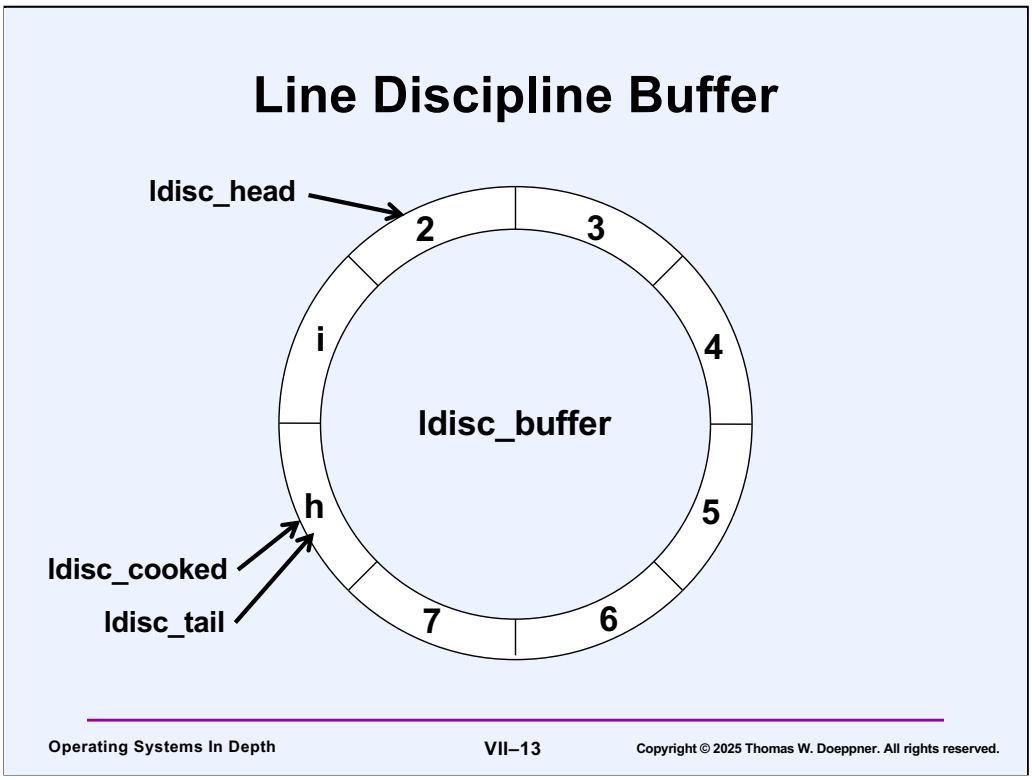
An “h” is received from the keyboard.



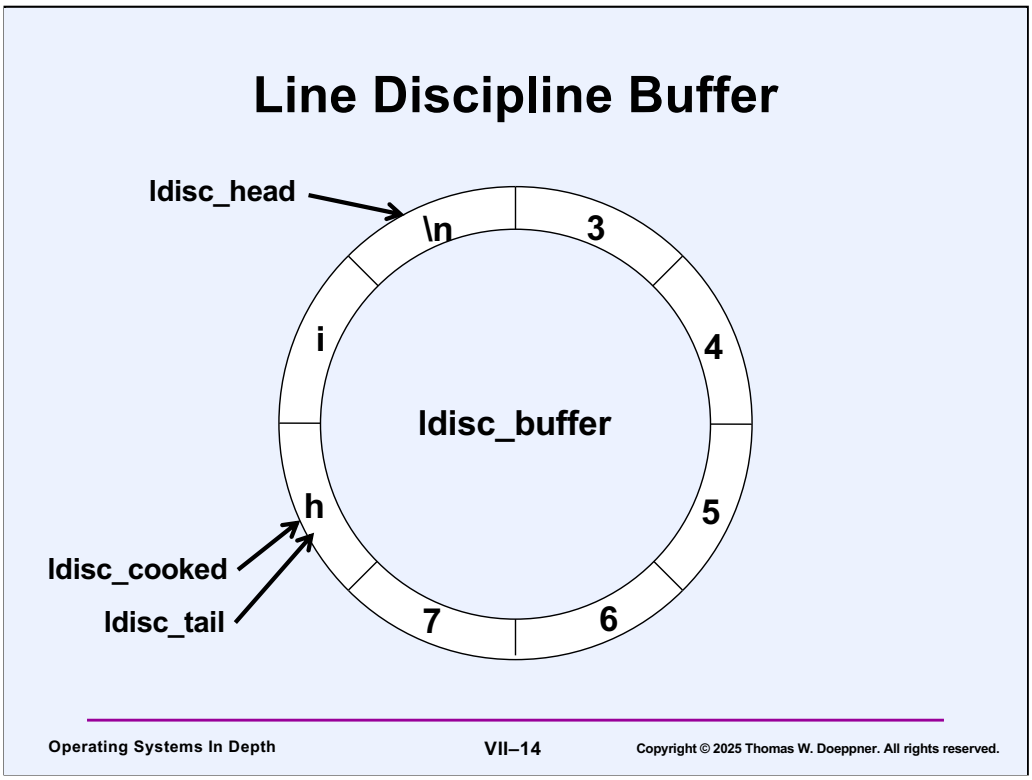
ldisc_head is incremented to point to the next slot.



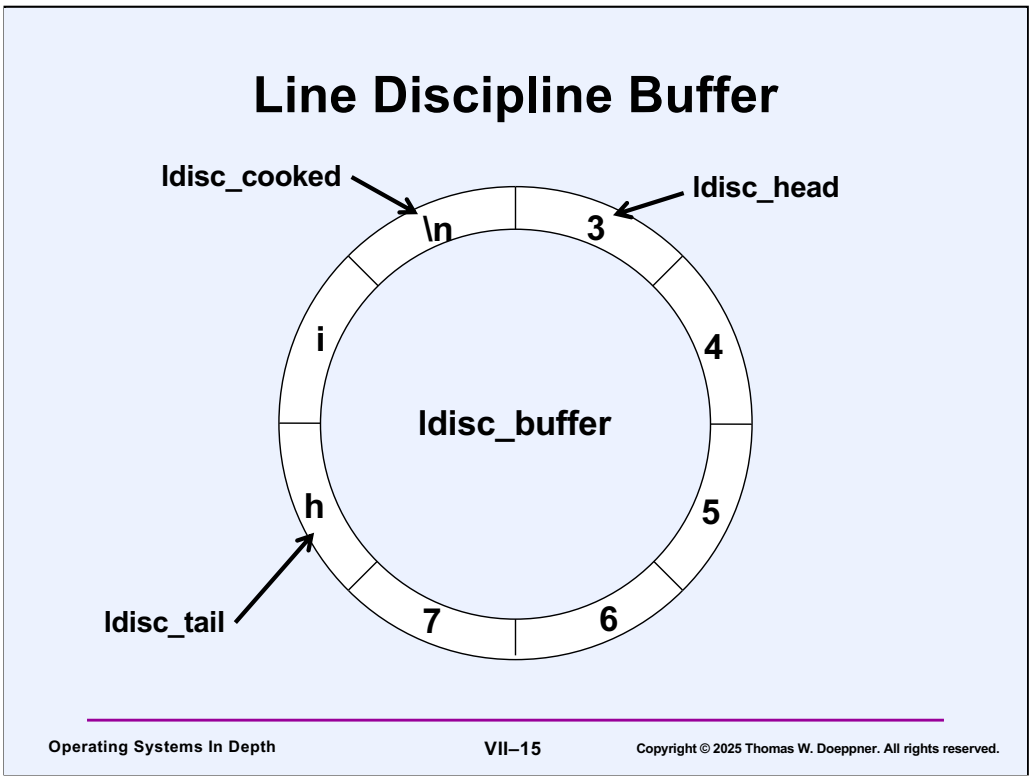
An ”i” is received.



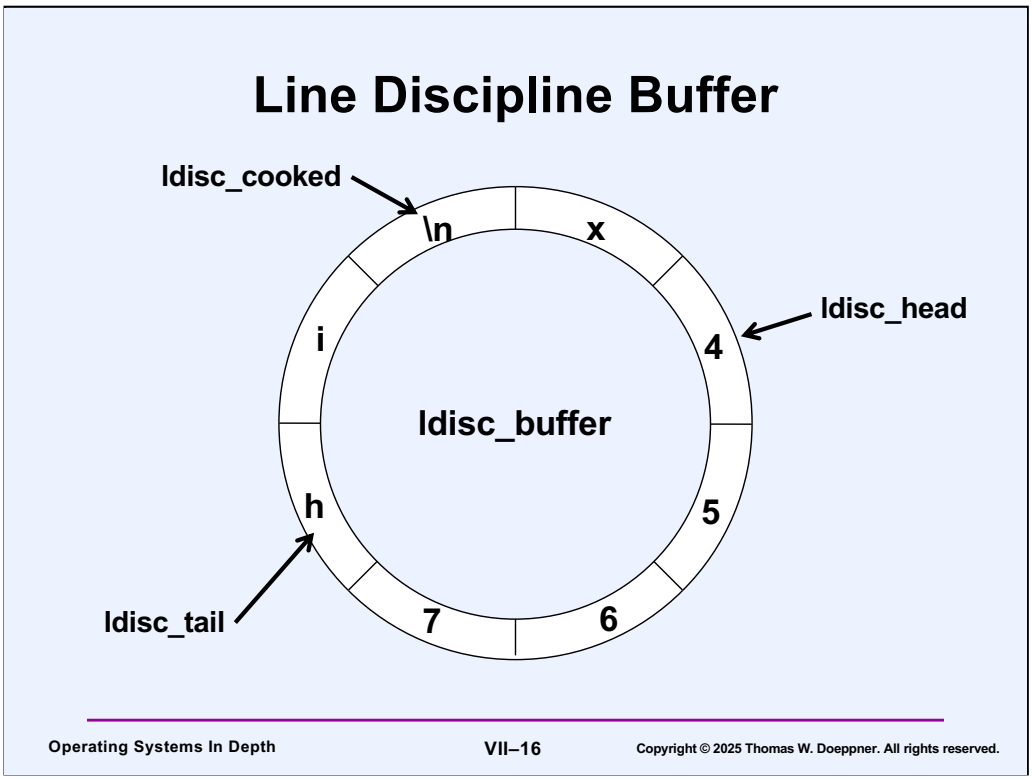
ldisc_head is incremented to point to the next slot.



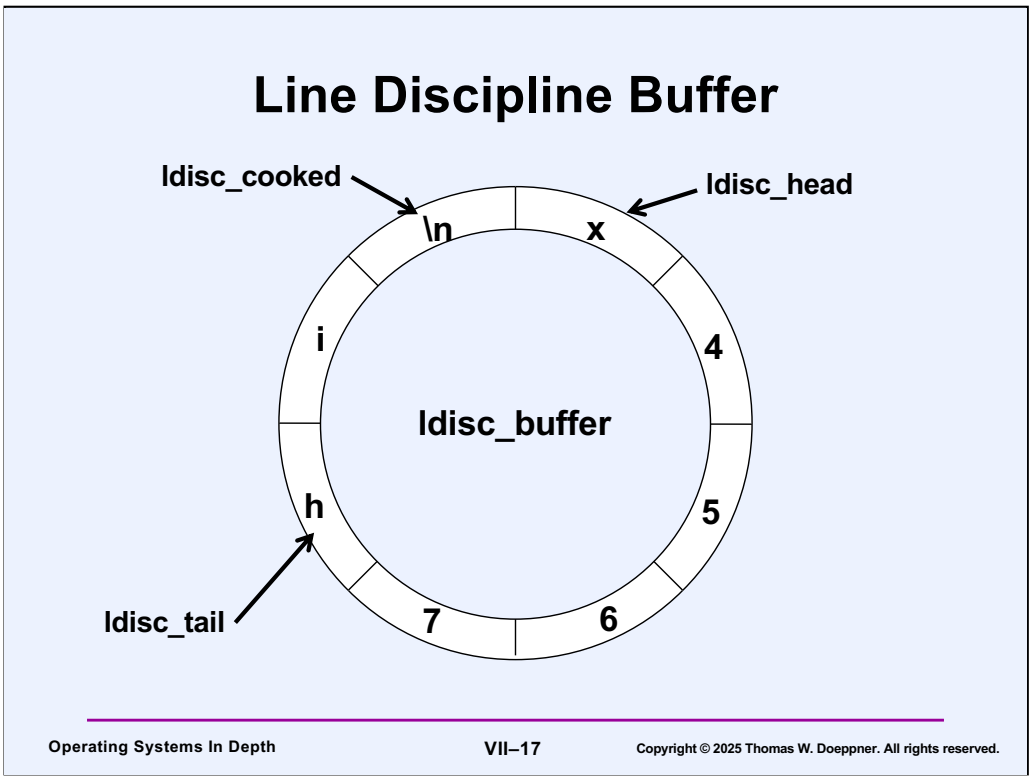
A newline character is received.



ldisc_head is incremented to point to the next slot; ldisc_cooked is set point to the newline – it now shows the end of the range of cooked characters. The characters for ldisc_tail through ldisc_cooked are now cooked characters and may be consumed via a **read** system call.



An “x” is received from the keyboard and **ldisc_head** is incremented.



A backspace is now received from the keyboard. **ldisc_head** is decremented by one; thus the next character received will replace the “x”.

Quiz 1

In which context are characters transformed from raw into cooked?

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

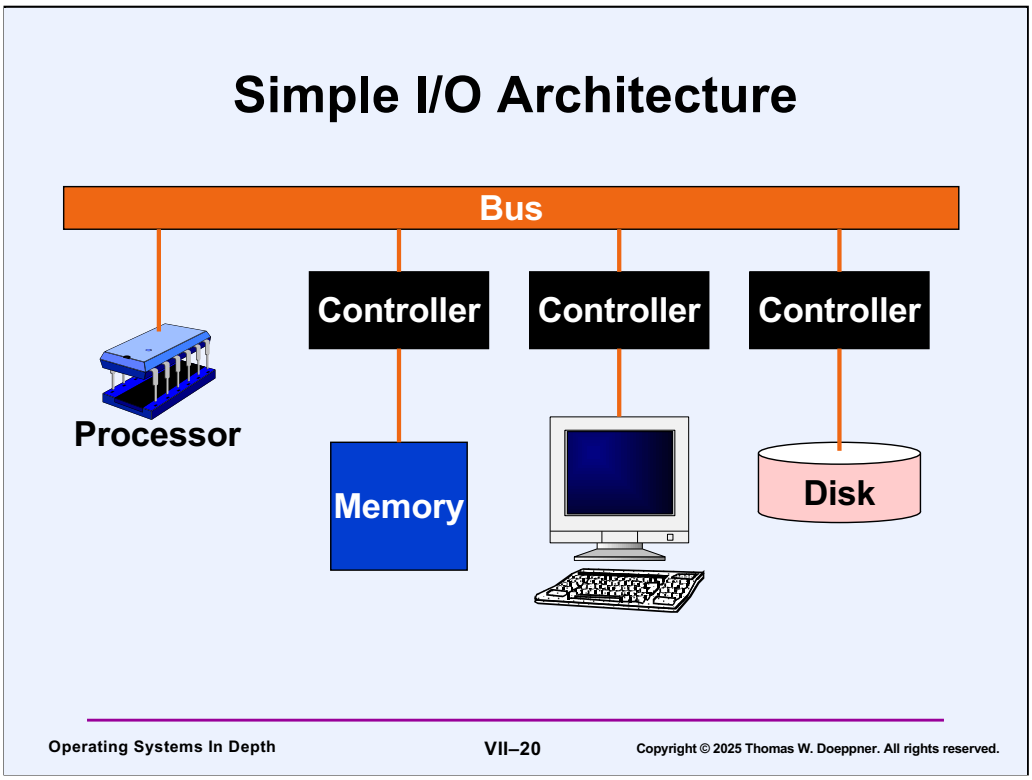
b) **In the interrupt context (i.e., on a "borrowed" stack)**

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

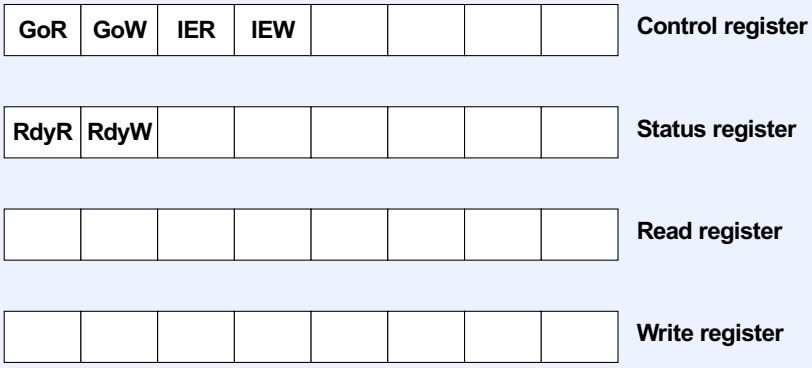
In this section we address the area of input and output (I/O). We discuss two basic I/O architectures and talk about the fundamental I/O-related portion of an operating system—the *device driver*.



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



Legend:

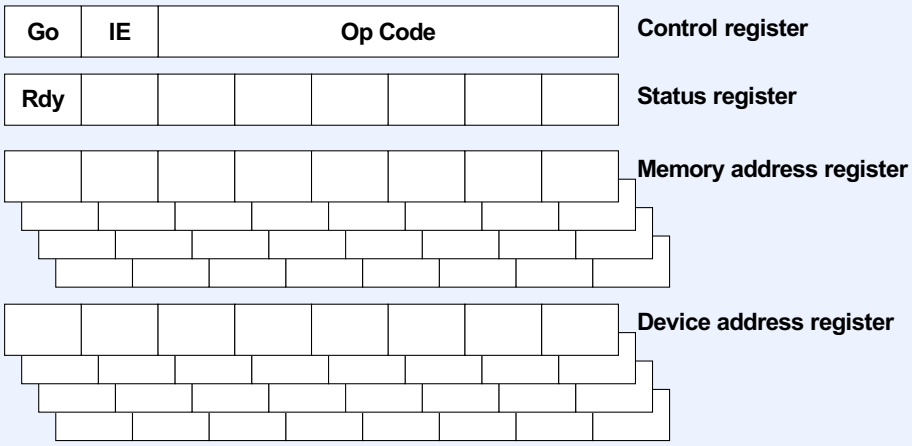
GoR	Go read (start a read operation)
GoW	Go write (start a write operation)
IER	Enable read-completion interrupts
IEW	Enable write-completion interrupts
RdyR	Ready to read
RdyW	Ready to write

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

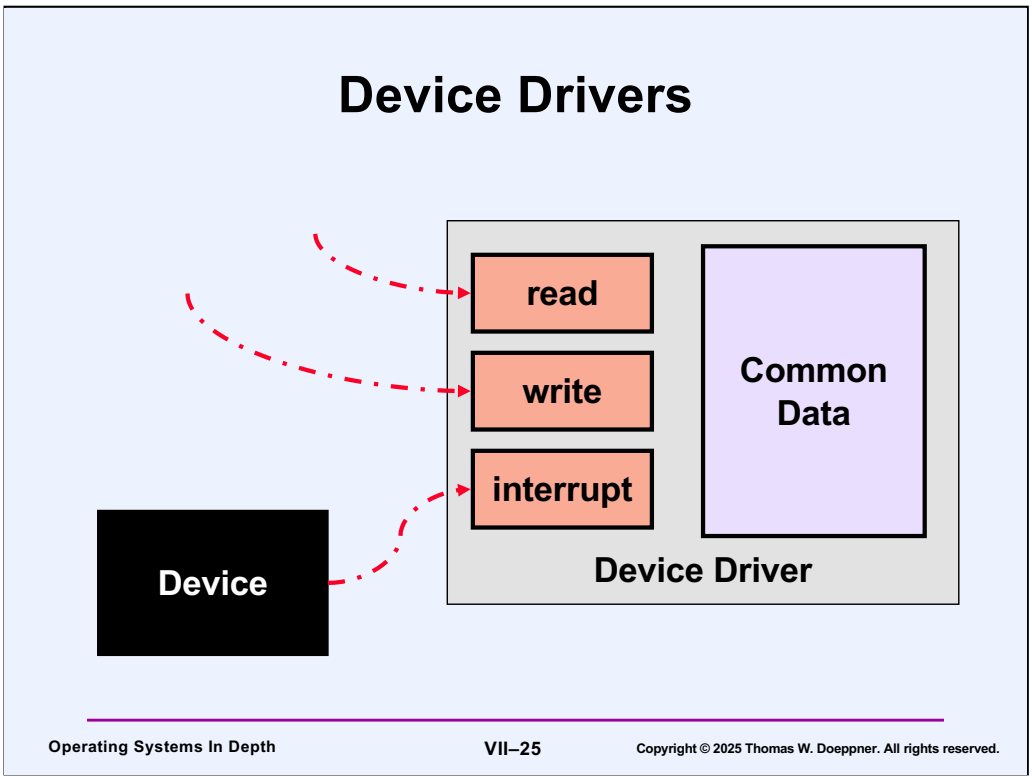


Legend:		
	Go	Start an operation
	Op Code	Operation code (identifies the operation)
	IE	Enable interrupts
	Rdy	Controller is ready

Direct Memory Access

- **E.g.: Disk controller**
- **Procedure**
  - **set the *disk address* in the *device address register* (only relevant for a seek request)**
  - **set the *buffer address* in the *memory address register***
  - **set the *op code* (SEEK, READ or WRITE), the GO bit and, if desired, the interrupt ENABLE bit in the *control register***
  - **wait for interrupt or for READY bit to be set**

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.

PDP-8

The photo is from http://www.pdp8.net/pdp8i/pdp8i.shtml.

PDP-8 Boot Code

```
07756 6032 KCC
07757 6031 KSF
07760 5357 JMP .-1
07761 6036 KRB
07762 7106 CLL RTL
07763 7006 RTL
07764 7510 SPA
07765 5357 JMP 7757
07766 7006 RTL
07767 6031 KSF
07770 5367 JMP .-1
07771 6034 KRS
07772 7420 SNL
07773 3776 DCA I 7776
07774 3376 DCA 7776
07775 5356 JMP 7756
07776 0000 AND 0
07777 5301 JMP 7701
```


The code in the slide was “toggled in” to the computer by hand. It would load a program provided on a paper tape, then run that program.

VAX-11/780

VAX-11/780 Boot

- Separate “console computer”
  - LSI-11
  - read boot code from floppy disk
  - load OS from root directory of first file system on primary disk

Configuring the OS (1)

- Early Unix
  - OS statically linked to contain all needed device drivers
    - all device-specific info included with drivers
  - disk drivers contained partitioning description

Configuring the OS (2)

- Later Unix
  - OS statically linked to contain all needed device drivers
    - at boot time, OS would probe to see which devices were present and discover device-specific info
  - partition table in first sector of each disk

IBM PC

The photo is from wikipedia.

Issues

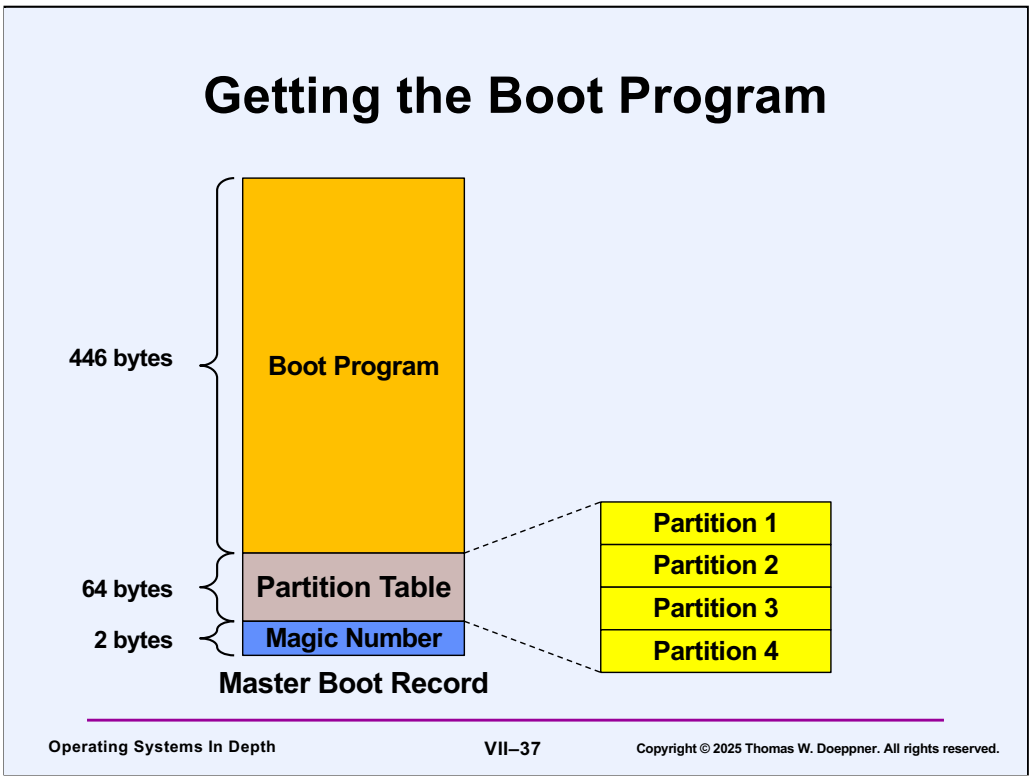
- Open architecture
  - large market for peripherals, most requiring special drivers
  - how to access boot device?
  - how does OS get drivers for new devices?

The Answer: BIOS

- Basic Input-Output System
  - code stored in *non-volatile RAM*
    - CMOS, flash, whatever …
  - configuration data also in NV RAM
    - including set of boot-device names
  - three primary functions
    - power-on self test (POST)
    - load and transfer control to boot program
    - provide drivers for all devices
  - main BIOS on motherboard
  - additional BIOSes on other boards

POST

- On power-on, CPU executes BIOS code
  - located in last 64k of first megabyte of address space
  - initializes hardware
  - counts memory locations



The master boot record (MBR) is stored in the first sector of each disk. This diagram is based on http://www.ibm.com/developerworks/library/l-linuxboot/index.html (which no longer exists). BIOS loads the MBR into memory, then passes control to the boot program. It scans the partition table to find one that's marked as the boot partition, then loads its first sector into memory and passes control to it. This code then loads the initial kernel image from the partition into memory and passes control to it. All disk I/O is performed by calling back to BIOS code.

BIOS is invoked by the x86's "software interrupt" procedure: the caller uses the x86 **int** (interrupt) instruction, which simulates the occurrence of an interrupt, passing control through an interrupt vector to the appropriate BIOS code. For the case of getting to device drivers, **int 13** is called.

Linux Booting (1)

- Two stages of booting provided by one of:
  - lilo (Linux Loader)
    - uses sector numbers of kernel image
  - grub (Grand Unified Boot Manager)
    - understands various file systems
  - both allow dual (or greater) booting
    - select which system to boot from menu
      - perhaps choice of Linux or Windows

See http://thestarman.pcministry.com/asm/mbr/index.html for details of some of this.

Linux Booting (2)

assembler code (*startup_32*)

- **Kernel image is compressed**
  - **step 1: set up stack, clear BSS, uncompress kernel, then transfer control to it**

assembler code (different *startup_32*)

- **Process 0 is created**
  - **step 2: set up initial page tables, turn on address translation**

C code (*start_kernel*)

- **Do further initialization**
  - **step 3: initialize rest of kernel, create init process (#1)**
  - **invoke scheduler**

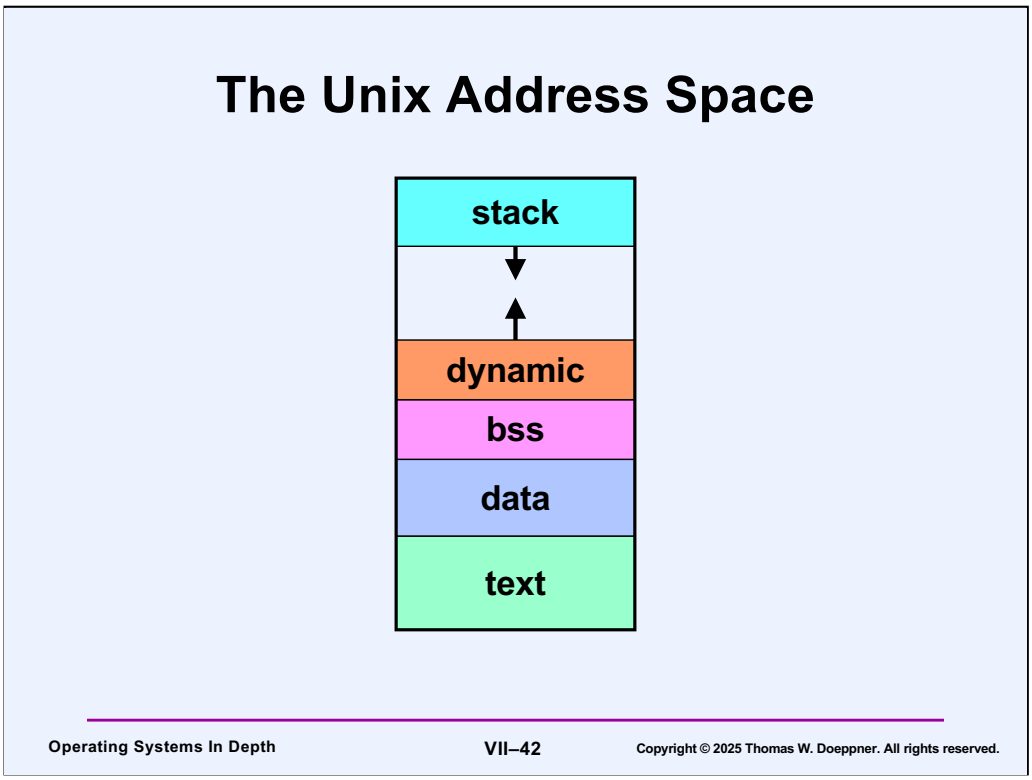
The first *startup_32* routine is in arch/i386/boot/compress/head.S; the second is in arch/i386/kernel/head.S. The *startup_kernel* routine is in init/main.c.

Beyond BIOS

- BIOS
  - designed for 16-bit x86 of mid 1980s
- Open Firmware
  - designed by Sun in the 1990s
  - portable
    - drivers, boot code in Forth
      - compiled into bytecode
  - used on non-Intel systems
- UEFI (Unified Extensible Firmware Interface)
  - improved BIOS originally from Intel
  - also uses bytecode

UNIX Structure

This lecture is covered in Section 4.1 of the textbook.

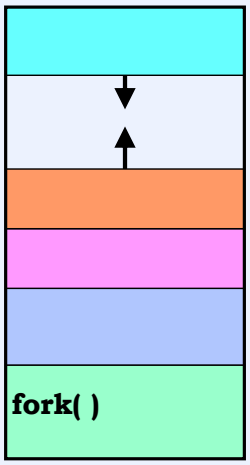


A Unix process's address space appears to be three regions of memory: a read-only **text** region (containing executable code); a read-write region consisting of initialized *data* (simply called data), uninitialized data (**BSS**—a directive from an ancient assembler (for the IBM 704 series of computers), standing for Block Started by Symbol and used to reserve space for uninitialized storage), and a **dynamic area**; and a second read-write region containing the process's user **stack** (a standard Unix process contains only one thread of control).

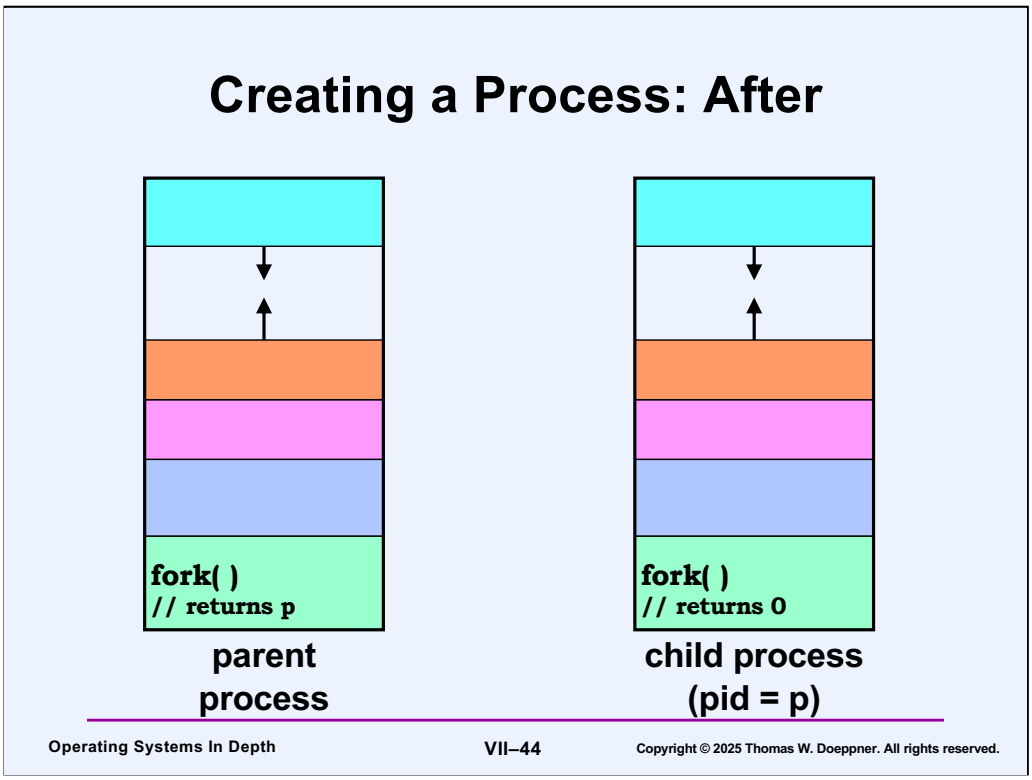
The first area of read-write storage is often collectively called the data region. Its dynamic portion grows in response to **sbrk** system calls. Most programmers do not use this system call directly, but instead use the **malloc** and **free** library routines, which manage the dynamic area and allocate memory when needed by in turn executing **sbrk** system calls.

The stack region grows implicitly: whenever an attempt is made to reference beyond the current end of stack, the stack is implicitly grown to the new reference. (There are system-wide and per-process limits on the maximum data and stack sizes of processes.)

Creating a Process: Before



The only way to create a new process is to use the **fork** system call.

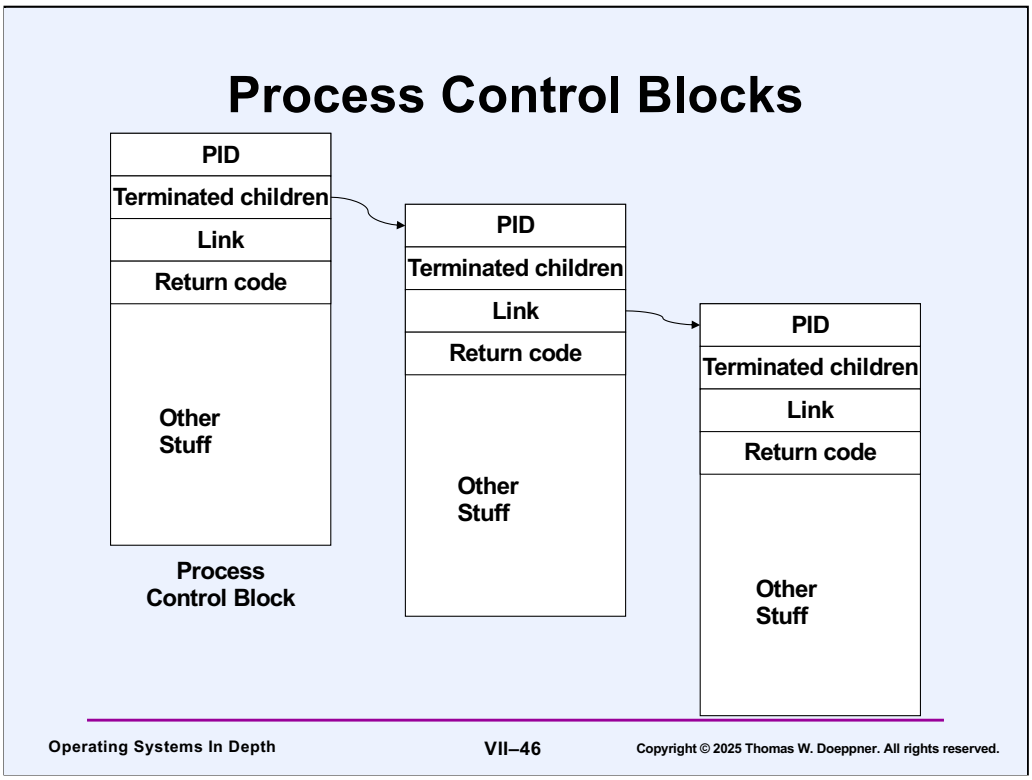


By executing **fork** the parent process creates an almost exact clone of itself which we call the child process. This new process executes the same text as its parent, but contains a copy of the data and a copy of the stack. This copying of the parent to create the child can be very time-consuming. We discuss later how it is optimized.

Fork is a very unusual system call: one thread of control flows into it but two threads of control flow out of it, each in a separate address space. From the parent's point of view, fork does very little: nothing happens to the parent except that fork returns the process ID (PID — an integer) of the new process. The new process starts off life by returning from fork. It always views fork as returning a zero.

Fork and Wait

```
short pid;
if ((pid = fork()) == 0) {
   /* some code is here for the child to execute */
   exit(n);
} else {
   int ReturnCode;
   while(pid != wait(&ReturnCode))
      ;
   /* the child has terminated with ReturnCode as its
      return code */
}
```



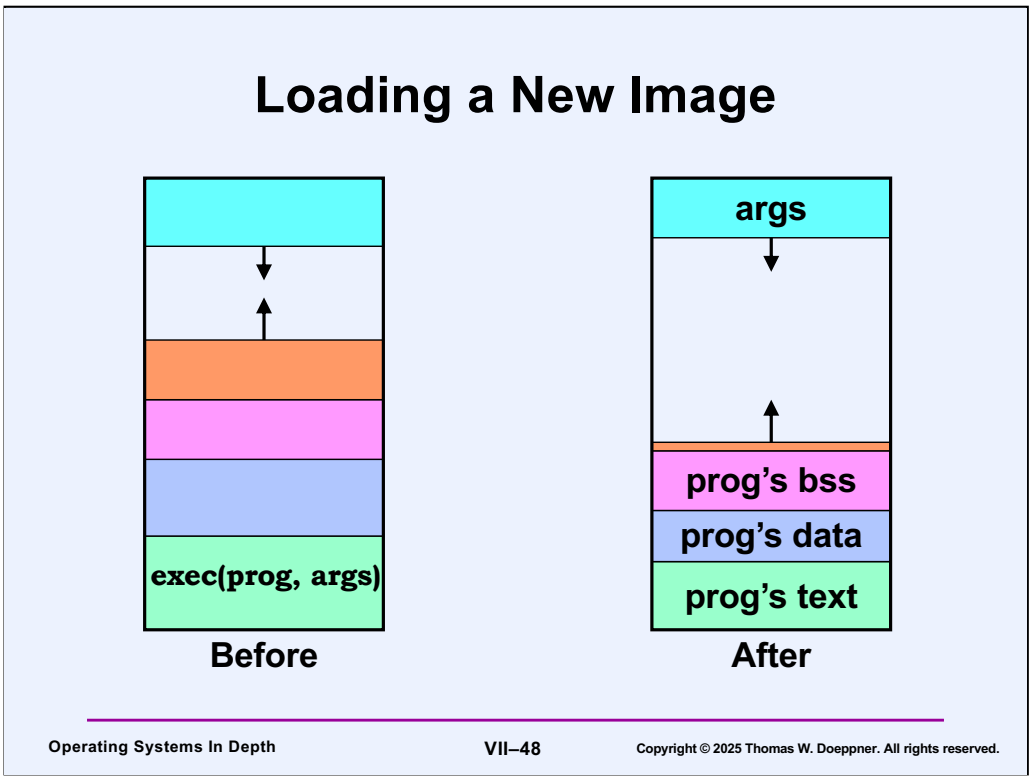
Note that in Linux, the “process control block” is known as the “task_struct”.

Exec

```
int pid;
if ((pid = fork()) == 0) {
   /* we'll soon discuss what might take place before exec
      is called */
   execl("/home/twd/bin/primes", "primes", "300", 0);
   exit(1);
}

/* parent continues here */

while(pid != wait(0))      /* ignore the return code */
   ;
```



Most of the time the purpose of creating a new process is to run a new (i.e., different) program. Once a new process has been created, it can use the **exec** system call to load a new program image into itself, replacing the prior contents of the process's address space. Exec is passed the name of a file containing a fully relocated program image (which might require further linking via a runtime linker). The previous text region of the process is replaced with the text of the program image. The data, BSS and dynamic areas of the process are "thrown away" and replaced with the data and BSS of the program image. The contents of the process's stack are replaced with the arguments that are passed to the main procedure of the program.

Quiz 2

```
int A=0, B=0, C=0, D=0;
A=1;
if (fork() > 0) {
  B=1;
  A=111;
} else {
  C=2;
  if (fork() > 0) {
    D=222;
  } else {
    D=A+B+C;
    // what value is now
    // in D for this process?
  }
}
exit(0);
```

Answer:

a) 0
b) 3
c) 113
d) indeterminate