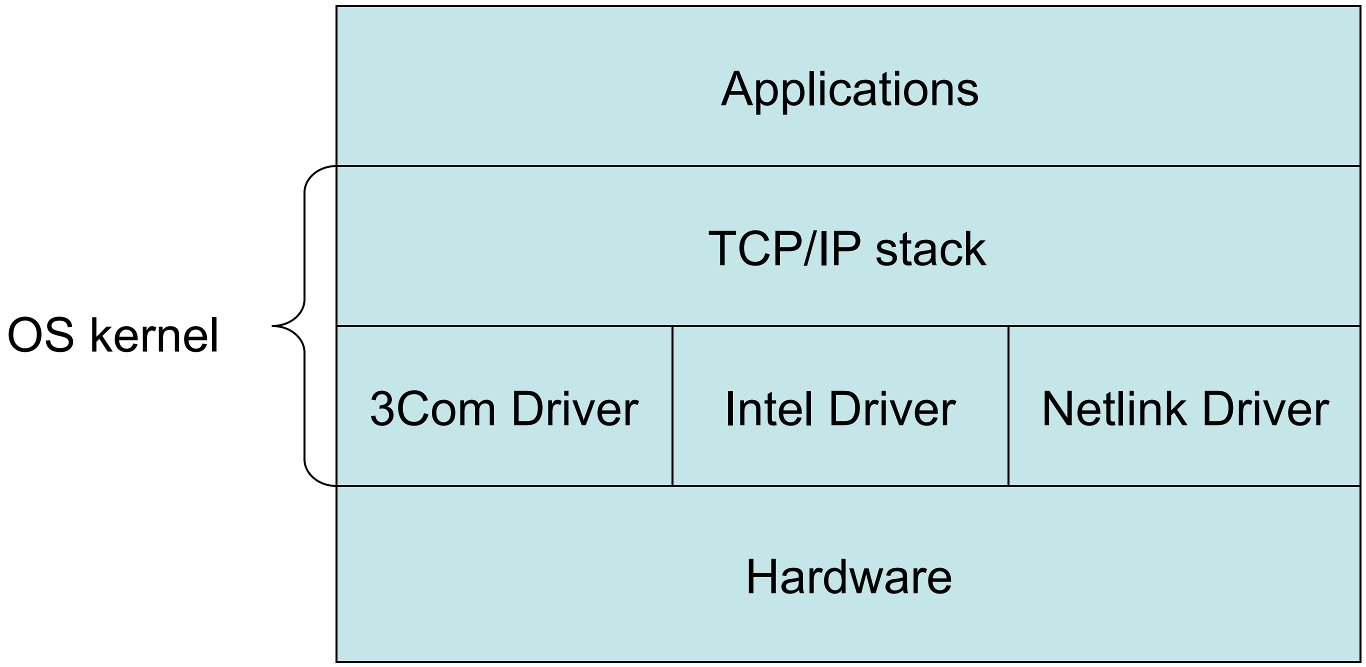
Device Drivers

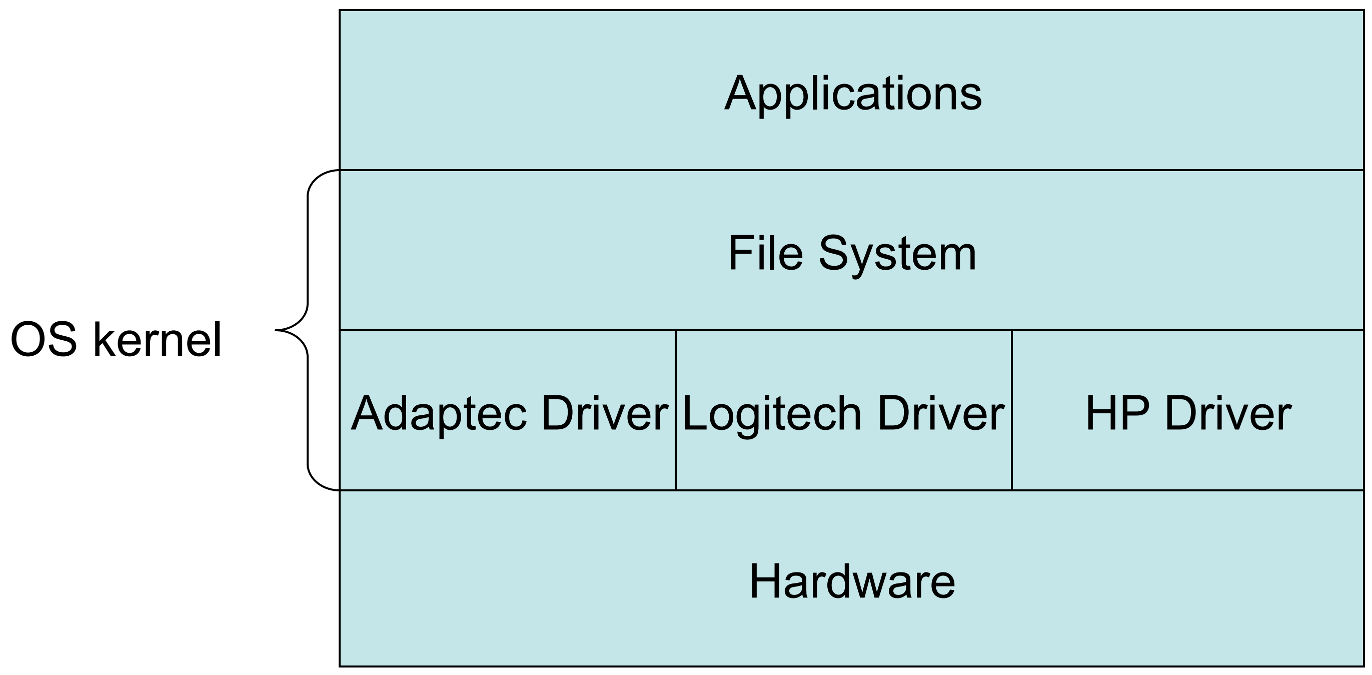
Device Drivers

- Code that manages the details of interacting with a particular piece of hardware
- Interacting with hardware includes handling I/O and interrupts
- Linux kernel (2.6.x):
  - 2.3 million lines of code in device drivers
  - < 1 million lines of code for everything else!

Example: Network Drivers



Example: Disk Controller Drivers



Handling I/O

- There are three major ways to perform I/O:
  - Memory mapped: a range of physical memory addresses correspond to the device (e.g., the video display)
  - Programmed I/O: Special instructions to move data to/from external devices
  - Direct Memory Access (DMA): a device transfers memory to/from regular memory (note the difference with memory mapped I/O)

Handling I/O

- DMA is very efficient but complex to set up and manage
- Many high-speed devices (e.g., network, disk controller) use DMA for efficiency
- Slower devices (e.g., keyboard, serial port) often use memory-mapped or programmed I/O for simplicity (and historical compatability)

Programmed I/O

- x86 implements programmed I/O via ports (not to be confused with TOS' IPC ports)
- I/O ports have their own address space (0..2^{16}-1)
- Two x86 instructions to access I/O ports:
  - `in port, location`: reads from I/O port `port`
  - `out data, port`: writes `data` to I/O port `port`
- External hardware is connected to certain port addresses
- TOS provides C functions that read and write bytes from I/O ports. These functions are in `tos/kernel/inout.c`

inportb()

```
/*
 * Reads a byte from the I/O port designated by port
 */
unsigned char inportb (unsigned short port)
{
    unsigned char _v;

    asm ("inb %w1,%0" : "=a" (_v) : "Nd" (port));
    return _v;
}
```

‘a’ : %EAX register

‘d’ : %EDX register

‘N’ : constant between 0 and 255

outportb()

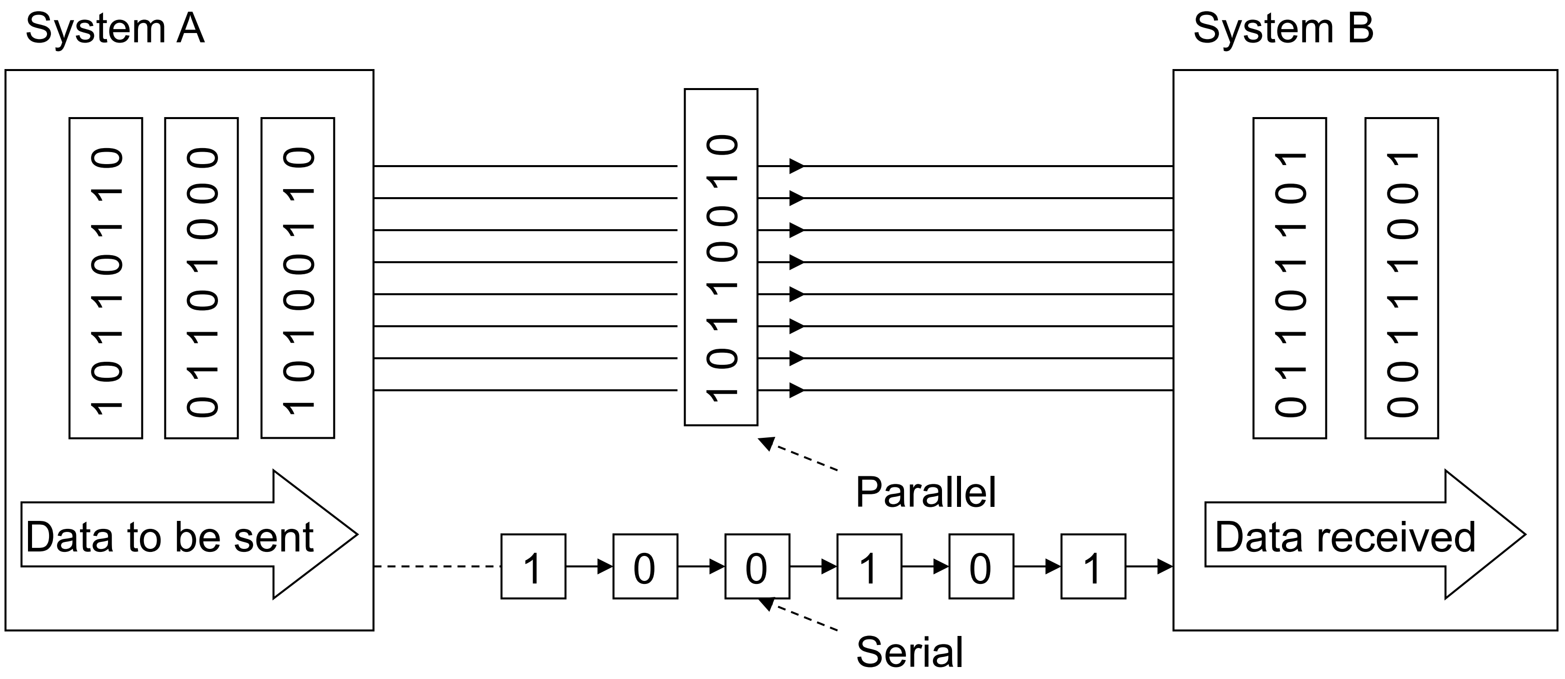
```
/*
 * Writes the byte value to I/O port port
 */
void outportb (unsigned short port, unsigned char value)
{
    asm ("outb %b0,%w1" : : "a" (value), "Nd" (port));
}
```

Serial Port Device Driver

- Our next goal: add a device driver for the serial port to TOS
- Like the timer service, we develop a new process that interacts directly with the serial port and communicates with other processes via IPC
- The eventual goal is to write an application that uses the serial port to control a model train
- First: details about the serial port

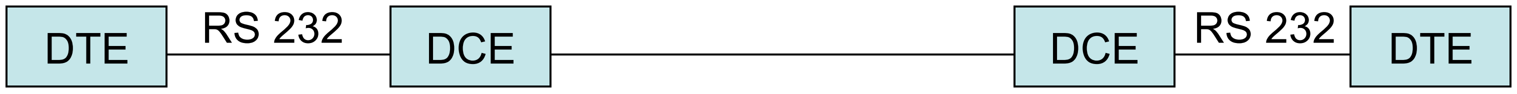
Data Communication

- Goal: physically transmit data between two systems
- Data is as a sequence of bytes
- Two ways to transmit a sequence of bytes:
  – Parallel: send one byte at a time
  – Serial: send one bit at a time



RS 232 Overview

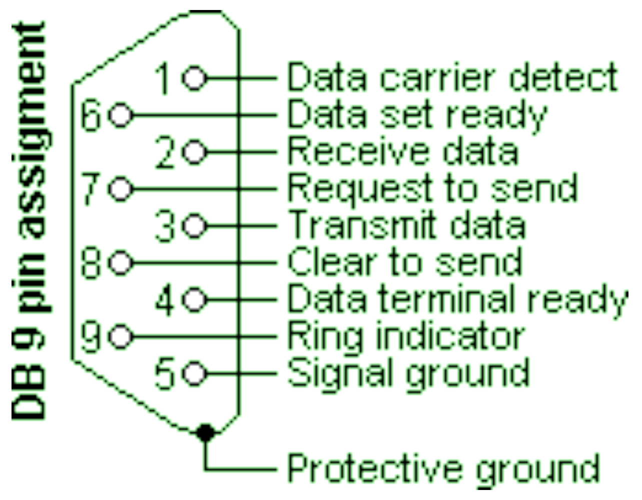
- RS 232 is a standard for transmitting data over a serial line
- First introduced in 1960 by the Electronic Industries Association (EIA)
- Defines a serial line connection between a DCE and a DTE
  - DCE (Data Communications Equipment), e.g. modem, printer
  - DTE (Data Terminal Equipment), e.g. computer
- RS 232 does not specify how data is transmitted between two DCEs
- Maximum transfer rate is 115,200 BPS

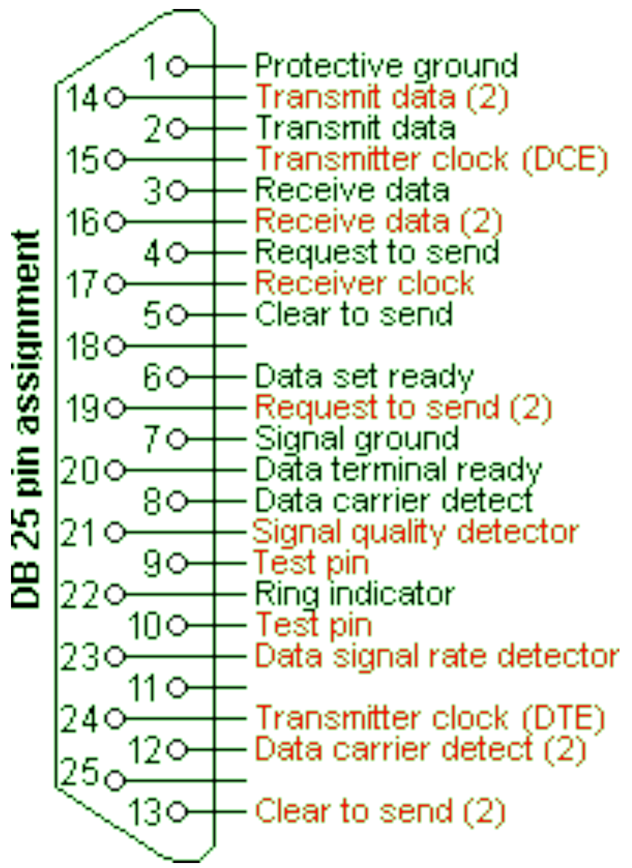


RS 232 Continued

- Standard covers details such as electronic representation of signals:
  - TRUE: -3V to -15V; FALSE: 3V to 15V
- Flow control:
  - Software: sending special flow control characters XON and XOFF
  - Hardware: flow control via extra cables (RTS/CTS; see next slides)
- The RS 232 protocol is implemented by a chip called a UART (Universal Asynchronous Receiver/Transmitter)

Serial Pinouts for D9 and D25 Connectors





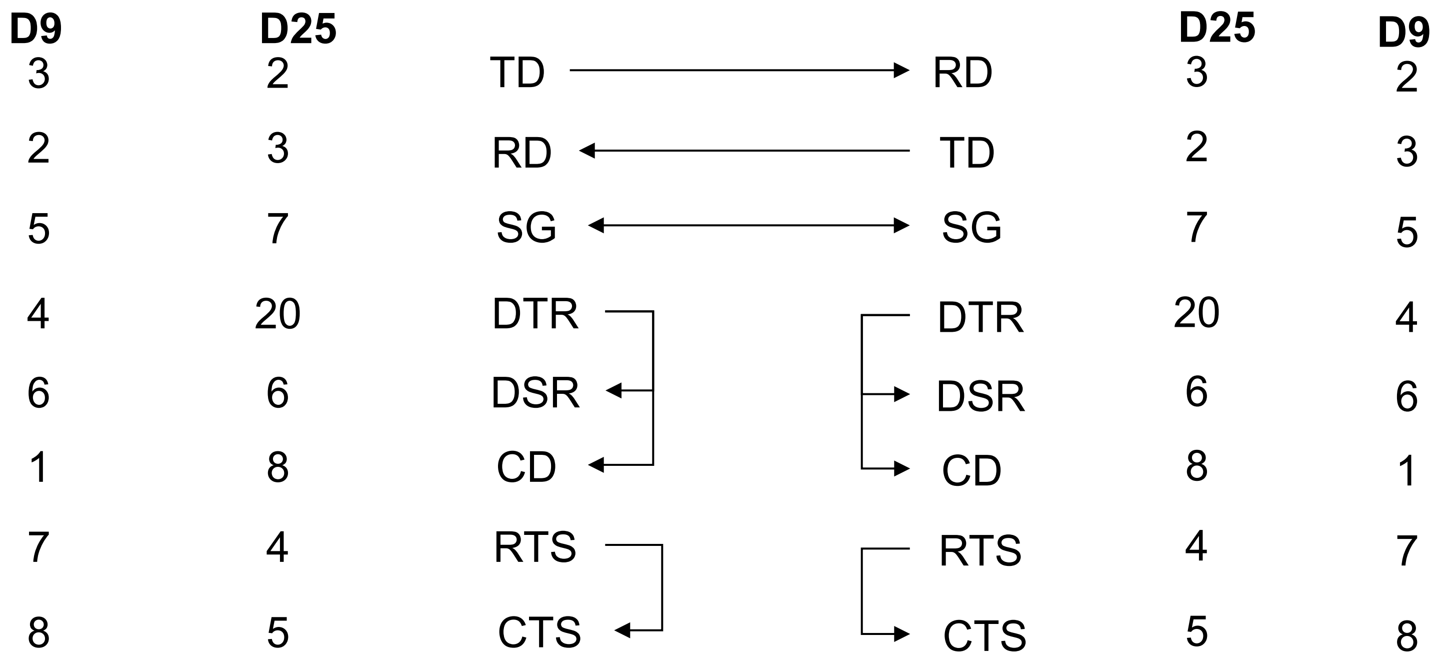
Data and Control Signals

25	9	Ab	Name	Purpose	DTE-DCE
2	3	TD	Transmit Data	Carries data from DTE to DCE	→
3	2	RD	Receive Data	Carries data from DCE to DTE	←
4	7	RTS	Request To Send	Asserted by DTE when it wants to send	→
5	8	CTS	Clear To Send	DCE is ready to accept data from DTE	←
6	6	DSR	Data Set Ready	Asserted by DCE to show its presence	←
7	5	SG	Signal Ground	Common ground	
8	1	DCD	Data Carrier Detect	DCE is successfully connected to a remote DCE	←
20	4	DTR	Data Terminal Ready	Asserted by DTE to show its presence	→
22	9	RI	Ring Indicator	DCE has detected an incoming phone call	←

RS-232 Wiring

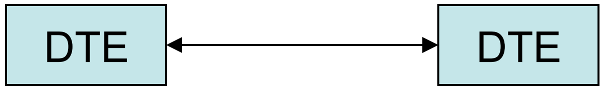
- Notice that transmit/receive are on specific pins
- A DCE (e.g., a modem) thus receives data on the transmit line and sends data on the receive line
- What if we want to directly connect two DTEs (i.e., connect two computers via the serial ports)?

Null Modem Wiring



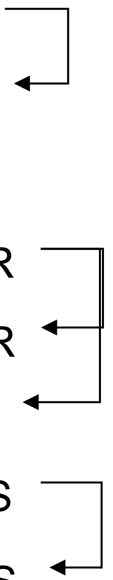
D9	D25			D25	D9
3	2	TD	RD	3	2
2	3	RD	TD	2	3
5	7	SG	SG	7	5
4	20	DTR	DTR	20	4
6	6	DSR	DSR	6	6
1	8	CD	CD	8	1
7	4	RTS	RTS	4	7
8	5	CTS	CTS	5	8

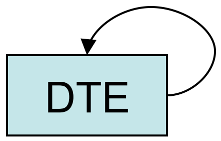
- Full-duplex connection between two DTEs. No flow control necessary since both DTEs send at the same speed.



Loop Back Plug

D9	**D25**	
3	2	TD
2	3	RD
5	7	SG
4	20	DTR
6	6	DSR
1	8	CD
7	4	RTS
8	5	CTS





The loop back plug has the receive and transmit lines connected together, so that anything transmitted out of the serial port is immediately received by the same port. Useful for debugging purposes. TOS contains a simulation of a loop back plug in `tos/tools/serial/loopback.pyw`

RS232 vs USB

- RS232:
  - 1:1 connection between one DTE and one DCE/DTE
  - Low transmission speed (max. 115,200 BPS)
  - Unbalanced signaling wrt to ground (-15V to +15V)
  - Data transfer is unidirectional on each line
  - Does not provide power
- USB (Universal Serial Bus):
  - 1:N connection between one computing device and N peripherals
  - Two power lines and 2 data lines. No physical control lines. Control and configuration done exclusively in software
  - USB2: 480 Mbps, USB3: 5 Gbps
  - Balanced signaling (0V to +5V)
  - Data transfer is bidirectional. Ownership of the data lines is part of the protocol
  - Provides power

Serial Port in Bochs

- Bochs has various ways to handle the emulated serial port -- for TOS we need to have data sent/received on the serial port sent to a network socket
- This requires a recent version of Bochs -- if you are using a version we provided, you are set.
- Must be enabled with the following lines in `.bochsrc`:

```
com1: enabled=1, mode=socket-client, dev=localhost:8888
com2: enabled=1, mode=socket-client, dev=localhost:8899
```

Serial Port Interface

- X86 communicates with the UART via programmed I/O
- COM1 can be accessed via I/O ports `0x3F8` to `0x3FF`
  - 8 ports for various purposes (see next slide)
- The base address `0x3F8` is defined as `COM1_PORT` in `tos/include/kernel.h`

Initializing the UART

```
void init_uart()
{
    int divisor;

    divisor = 115200 / 1200;
    /* LineControl disabled to set baud rate */
    outportb (COM1_PORT + 3, 0x80);
    /* lower byte of baud rate */
    outportb (COM1_PORT, divisor & 255);
    /* upper byte of baud rate */
    outportb (COM1_PORT + 1, (divisor >> 8) & 255);
    /* LineControl 2 stop bits */
    outportb (COM1_PORT + 3, 2);
    /* Interrupt enable*/
    outportb (COM1_PORT + 1, 1);
    /* Modem control */
    outportb (COM1_PORT + 4, 0x0b);
    inportb (COM1_PORT);
}
```

Sending to the UART

- Reading from port `0x3fd` retrieves the *Line Status Register*
- Bit 5 in this register tells us if the UART send buffer is full or empty
- Before sending, we must first wait until the send buffer is empty:

```
while (!(inportb(COM1_PORT+5) & (1<<5)));
```

- To write a byte, it is simply sent to port `0x3f8`:

```
outportb(COM1_PORT, byte_to_be_written);
```

Receiving from the UART

- Whenever data is received, interrupt `COM1_IRQ` (0x64) is raised
- The interrupt signals the arrival of a byte ready to be read from the I/O port:

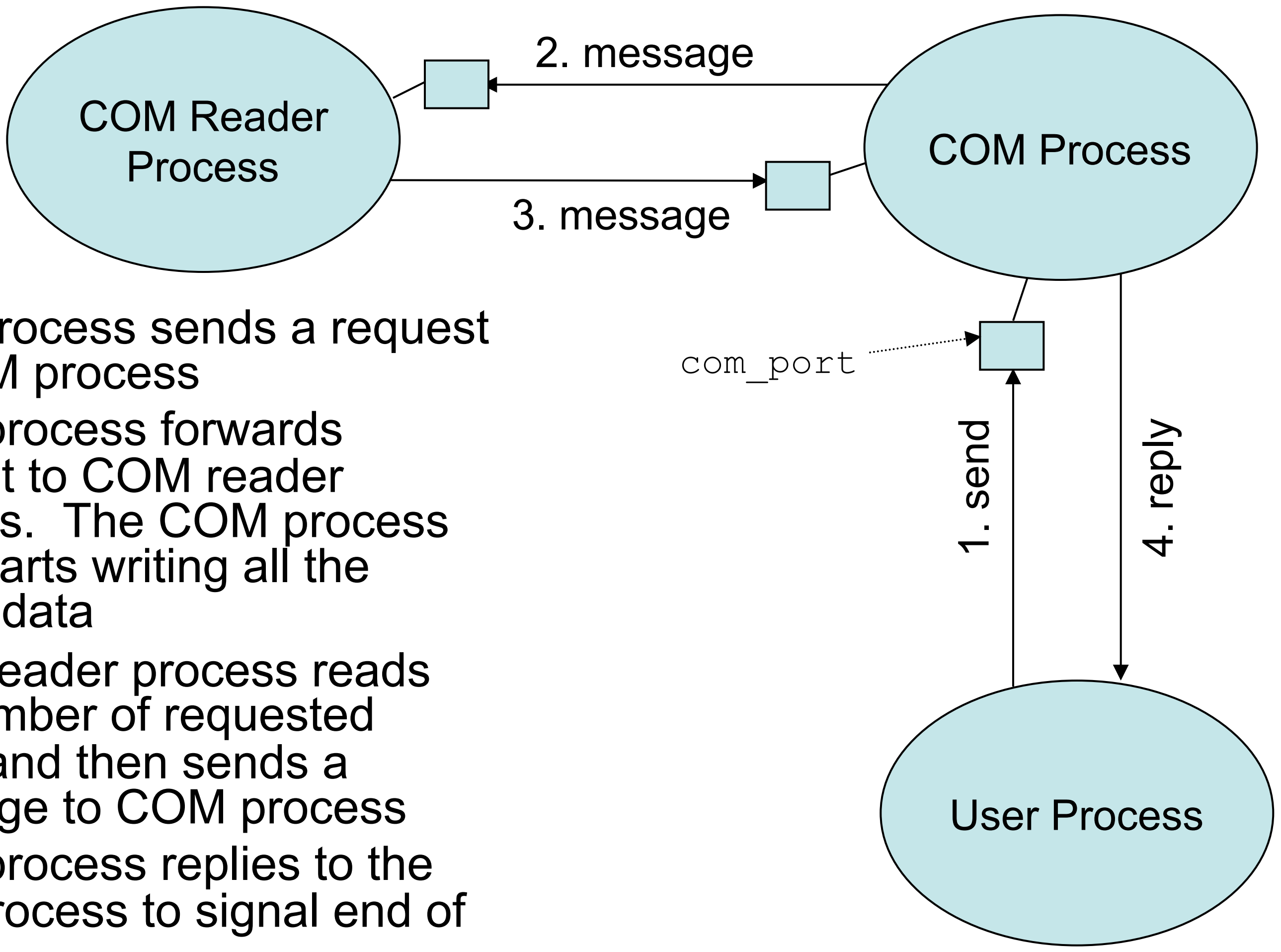
```
byte_to_be_read = inportb(COM1_PORT);
```

- If multiple bytes are received, an interrupt is raised for each byte

TOS Serial Driver

- Now, we know everything we need to write a driver for the serial port in TOS!
- Goal: processes send a message to the “COM service” asking to read/write data
- Problem: reader wants to block waiting for interrupts, writer wants to poll the ready bit for sending
- As with the timer service, we fix this by splitting the COM service into two processes

TOS Serial Driver Architecture



1. User process sends a request to COM process
2. COM process forwards request to COM reader process. The COM process then starts writing all the output data
3. COM reader process reads the number of requested bytes and then sends a message to COM process
4. COM process replies to the user process to signal end of I/O

COM Service Message

```
typedef struct _COM_Message
{
    char* output_buffer;
    char* input_buffer;
    int   len_input_buffer;
} COM_Message;
```

- Defined in `tos/include/kernel.h`
- Members:
  - `output_buffer`: zero-terminated string to be output
  - `input_buffer`: buffer where input will be stored
  - `len_input_buffer`: number of bytes to be read

COM1 I/O Example

```
void com1_example ()
{
    char buffer [12];   /* 12 == strlen ("Hello World!") */
    COM_Message msg;
    int         i;

    msg.output_buffer    = "Hello World!";
    msg.input_buffer     = buffer;
    msg.len_input_buffer = 12;
    send (com_port, &msg);
    for (i = 0; i < 12; i++)
        kprintf (“%c”, buffer[i]);
}
```

Using the loopback device, this program will print “Hello World!”
Global variable `com_port` is initialized in `init_com()` and is owned by the COM process.

COM Process

```
void com_process (PROCESS self, PARAM param)
{
    while (1) {
            - receive message from user process.
            - forward message to COM reader process
            - write all bytes contained in COM_Message.output_buffer
              to COM1
            - wait for message from COM reader process that signals that
              all bytes have been read
            - reply to user process to signal that all I/O has been completed
    }
}
```

COM Reader Process

```
void com_reader_process (PROCESS self, PARAM param)
{
    while (1) {
            - receive message from COM process.
              This message contains the number of bytes to read in
              COM_Message.len_input_buffer
            - read as many bytes requested from COM1 using
              wait_for_interrupt (COM1_IRQ) and
              inportb(COM1_PORT)
            - send message to COM process to signal that all bytes have
              been read
    }
}
```

Serial Line Interface

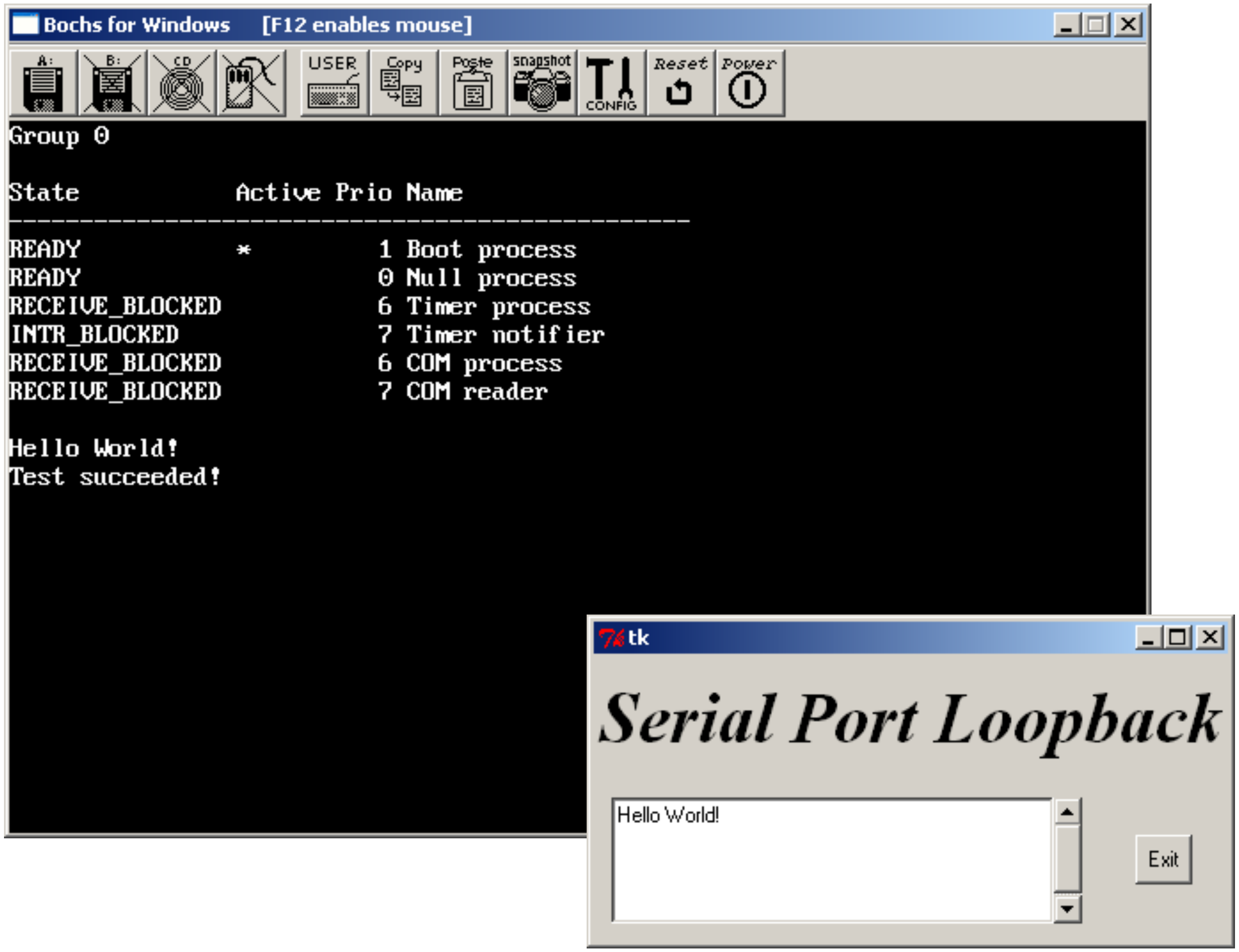
- `void init_com()`
  Initialize the serial line device driver.
  – After initialization the global variable `com_port` should point to the port that is owned by the COM process.
  – The COM process should accept messages of type `COM_Message` (defined in `kernel.h`) as explained on earlier slides.
  – The priority of the COM process should be 6.
  – The priority of the COM reader process should be 7.

Assignment

Assignment 9

- Implement the function located in `tos/kernel/com.c`: `init_com()`
- Note that you will have to implement and register an appropriate ISR (`com1_isr()`).
- Note that you have to run the loopback plug simulator located in `tos/tools/serial/loopback.pyw`
  before running Bochs
- Test case:
  - `test_com_1`

test_com_1


Bochs for Windows [F12 enables mouse]
A: B: CD USER Copy Paste snapshot CONFIG Reset Power
tk
Serial Port Loopback
Exit

```
Group 0

State            Active Prio Name
-------------------------------------------------
READY            *        1 Boot process
READY                     0 Null process
RECEIVE_BLOCKED           6 Timer process
INTR_BLOCKED              7 Timer notifier
RECEIVE_BLOCKED           6 COM process
RECEIVE_BLOCKED           7 COM reader

Hello World!
Test succeeded!
```

```
Hello World!
```