Terminals

Terminal Hardware

Terminals are of three types:

- interface serially with the machine (through RS-232), 25-pin connectors
- memory-mapped terminals
- network terminals

Serial-Line Terminals

UARTs (Universal Asynchronous Receiver Transmitters)—chip to do character-to-serial and serial-to-character for bits passed over the serial line.

Generally the device driver will output a character at a time, waiting for the interface card to interrupt when another character can be sent.

Speed between 50-9600 bps (bits/sec). Even at highest rate takes 1 ms to deliver a character.

Look at Fig 5-34

“tty” is an abbreviation for Teletype. Now represents any terminal.

Now used a lot for terminal emulation windows.

Memory-Mapped Terminals

See Fig 5-38

Interface with special memory called *video RAM*. Also a *video controller*. Beam of electrons is periodically (60 times/sec) scanned across the screen to continually refresh the screen. Not scanning frequently enough causes flicker.

PC, workstation uses a character-mapped display for the console.

Network Terminals

See Fig 5-45 X Windows where the client has a lot of computing power and memory.

Terminal Software

The device drivers for the terminal interface are referred to collectively as the *terminal driver.*

The terminal driver provides a *user interface* through which users interact with processes accessing the terminal device.

Processes interact with the driver through the *process interface.*

The user interface concerns itself with such aspects as:

- echoing characters typed on the terminal
- processing special characters such as backspace and line-kill
- stopping and restarting output (e.g., $\hat{\mathrm{Q}}$ and $\hat{\mathrm{S}}$)
- generating the appropriate CR/LF sequence to move the cursor to the start of the next line

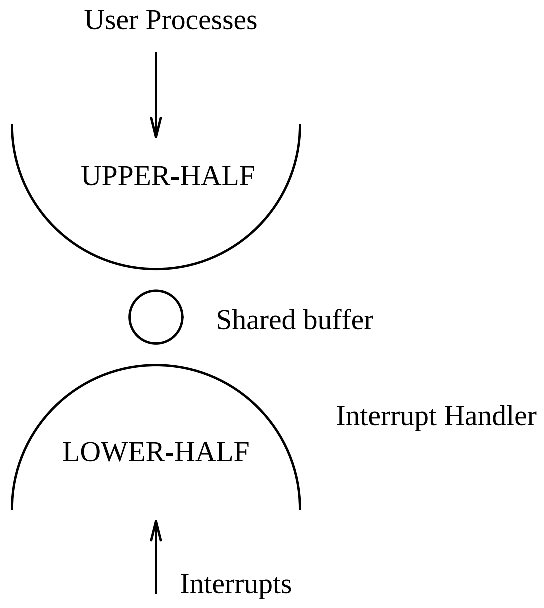
Device Driver Organization

Drivers are typically organized into an *upper-half* and *lower-half*.

The upper-half driver implements the device-independent routines such as *read* and *write*. They do not manipulate the devices directly.

The lower-half interacts with the device registers and handles interrupt.

The upper-half and lower half synchronize through a shared data structure. Typically, the upper-half enqueues requests for service, and the lower-half driver services requests in the queue.



Generic picture for serial line (terminal or network driver)

The terminal device driver maintains two queues:

- an input buffer holds characters received from the terminal by the interrupt handler, but not yet read by any upper half routine.
- an output queue for characters received from the upper-half, but not yet transferred to the device.

The two buffers are needed so that the interface between processes and the device can be asynchronous.

Synchronization of the Upper and Lower Halves

Both buffers are instances of the bounded buffer problem (producer/consumer):

- for the input buffer, the lower-half produces characters and the upper-half consumes them
- for the output buffer, the upper-half produces characters, and the lower-half consumes them

Earlier, we solved the bounded buffer using two semaphores. The producer waits for space to become available, while the consumer for items to placed in the buffer.

However, the lower-half is invoked as an interrupt handler; thus it cannot wait.

Solution: we can still use semaphores, but change the lower-half so that it never waits for space.

Output Driver

- instead of having the lower half wait for characters from the upper-half, have the upper half wait for space in the buffer.
- thus, the lower half never waits for anything.
- When the upper-half deposits a character in the output queue, it "kicks" the driver to start transmission (if idle). The transmitter continues until the queue has emptied.

Input Driver

- the upper half waits for characters to be placed in the buffer
- the lower half never waits: if the buffer is "full", the new character is dropped (or oldest character is over written by the newly arrived character
- when character arrives from terminal, the device "kicks" the lower-half by posting an interrupt

```
ttygetc(device)
  wait for char to arrive
  pick up next char from buffer
  return(ch)

lower half: (on interrupt from keyboard device)
  if (no space available in buffer)
    return;
  place char in buffer
  signal(char-available)


ttyputc(device, char)
  wait(space in buffer)
  put char in buffer
  start device

lower half (on interrupt when display is ready)
  if more chars
    send char to device
    signal(space in buffer)
  else
    disable device
```

Input Processing

The characteristics of the user interface are controlled by the *control* (*ioctl*) operation. For instance, control calls allow:

- a process to turn echoing on or off
- a process to change the *mode* of processing to:
  - *RAW* — (noncanonical mode in POSIX) driver passes the characters without interpretation directly to the process. Used by *emacs*.
  - *CBREAK* — driver handles echoes characters, handles start and stop characters
  - *COOKED* — (canonical mode in POSIX) driver also handles backspace and line-kill characters. Gives input a line at a time.

Normal Unix I/O is cooked.

echo buffer, should input be echoed (should for full-duplex). For example echoing is turned off when entering a password.

Watermark Processing

Could apply to sending to a network device.

One problem that arises with programs that print a lot of data is that they quickly consume all the buffer space and block. The following sequence then occurs:

- process blocks, waiting for space in the buffer
- current character transfer completes, *lower half output handler* signals the semaphore, and the waiting process is moved to the ready list
- Waiting process is scheduled, which promptly deposits another character in the queue and blocks again.
- the entire sequence repeats for each character

Because context-switching is expensive, the above phenomenon is undesirable. The technique known as *watermark processing* reduces this cost.

Idea: maintain a *high watermark* that indicates at what point signaling becomes inefficient. When the high watermark is reached, the signal operation is delayed until the *low watermark* is reached, at which time all the delayed signals are completed.

For example, the output interrupt handler enters a delayed mode when the buffer become full. As the tty driver drains characters from the queue, the handler delays signaling the upper half until 20 characters have been transmitted, at which time it it issues 20 signals.