To control access to individual devices, we place code that communicates with the devices into the **O.S.** (this portion of the O.S. is also called the **kernel**) and in special routines called **device drivers**.

```
ch = getchar();
```

1. call `getchar`
2. `syscall`
3. `int`
4. `ret`
What must the device driver code look like?

Assume simplistic devices that generate or consume one ASCII character.

A poor idea adds a new instruction to the instruction set

input ..... implies the use of a register that starts with the device # and is overwritten with the byte of input

Look up x86 instructions insb outsb insb outsb w l ...

Why is this a poor idea?
We want communication (the transfer of that single byte) **without special instr.**

To make it work, use **memory mapping**

Within the kernel's **data segment**: Each device is assigned a word at boot time.

- Keyboard Data
- Display Data

Assume simplistic system with only keyboard & ASCII display
1. user types 'X'

2. 'X' \rightarrow \text{Keyboard Data}
1. 'Q' appears in memory

2. character appears on display
The **keyboard driver code**:

\[ \text{movl Keyboard-Data, } \%\text{eax} \]
\[ \text{ret from syscall} \]

↑ Assume this is where the ASCII char is supposed to be after syscall completes.

The **display driver code**:

\[ \text{movl } \%\text{eax, Display-Data} \]
\[ \text{ret from syscall} \]

What happens if user has not typed a char on the keyboard?
We want **blocking input**.
(no `recvfromsyscall` until there
is a char)

We need a **status bit**

\[
\begin{align*}
1 & \quad \text{ready} \\
0 & \quad \text{not ready (busy)}
\end{align*}
\]

Place this bit into its own memory mapped word & make it the \text{msb}, so code can test for \( \geq 0 \) or \( < 0 \).

---

Keyboard-Data
Keyboard-Status

Display-Data
Display-Status
1. user types 'X'

<table>
<thead>
<tr>
<th>Keyboard_Data</th>
<th>Keyboard_Status</th>
</tr>
</thead>
<tbody>
<tr>
<td>'X'</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td></td>
</tr>
</tbody>
</table>
Now, driver code uses a **spin wait loop** (to implement blocking I/O)

```assembly
kb_spin:  testl Keyboard_Status, Keyboard_Status
          jz kb-spin
          movl Keyboard_Data, %eax
          ret from syscall

disp-spin: testl Display_Status, Display_Status
           jz disp-spin
           movl %eax, Display_Data
           ret from syscall
```
Byte transfers are OK,
But, what about faster devices that like to transfer more than a byte?

the solution: DMA
Direct Memory Access