Physical Memory, I/O, Segmentation

Outline

- x86 Physical Memory Map
- I/O
- Example hardware for address spaces: x86 segments

x86 Physical Memory Map

- The physical address space mostly looks like ordinary RAM
- Except some low-memory addresses actually refer to other things
- Writes to VGA memory appear on the screen
- Reset or power-on jumps to ROM at 0x000ffff0 (so must be ROM at top of BIOS)

```
+------------------+  <- 0xFFFFFFFF (4GB)
|      32-bit      |
|  memory mapped   |
|     devices      |
|                  |
/\/\/\/\/\/\/\/\/\/\

/\/\/\/\/\/\/\/\/\/\
|                  |
|      Unused      |
|                  |
+------------------+  <- depends on amount of RAM
|                  |
|                  |
| Extended Memory  |
|                  |
|                  |
+------------------+  <- 0x00100000 (1MB)
|     BIOS ROM     |
+------------------+  <- 0x000F0000 (960KB)
|  16-bit devices, |
|  expansion ROMs  |
+------------------+  <- 0x000C0000 (768KB)
|   VGA Display    |
+------------------+  <- 0x000A0000 (640KB)
|                  |
|    Low Memory    |
|                  |
+------------------+  <- 0x00000000
```

I/O

- Original PC architecture: use dedicated *I/O space*
  - Works same as memory accesses but set I/O signal
  - Only 1024 I/O addresses
  - Accessed with special instructions (IN, OUT)
  - Example: write a byte to line printer:

```
#define DATA_PORT    0x378
#define STATUS_PORT  0x379
#define   BUSY 0x80
#define CONTROL_PORT 0x37A
#define   STROBE 0x01
void
lpt_putc(int c)
{
  /* wait for printer to consume previous byte */
  while((inb(STATUS_PORT) & BUSY) == 0)
    ;

  /* put the byte on the parallel lines */
  outb(DATA_PORT, c);

  /* tell the printer to look at the data */
  outb(CONTROL_PORT, STROBE);
  outb(CONTROL_PORT, 0);
}
```

- Memory-Mapped I/O
    - Use normal physical memory addresses
        - Gets around limited size of I/O address space
        - No need for special instructions
        - System controller routes to appropriate device
    - Works like ``magic'' memory:
        - *Addressed* and *accessed* like memory, but ...
        - ... does not *behave* like memory!
        - Reads and writes can have ``side effects''
        - Read results can change due to external events

Example hardware for address spaces: x86 segments

The operating system can switch the x86 to protected mode, which supports virtual and physical addresses, and allows the O/S to set up address spaces so that user processes can't change them. Translation in protected mode is as follows:

- selector:offset (virtual / logical addr)
  ==SEGMENTATION==>
- linear address
  ==PAGING ==>
- physical address

Next lecture covers paging; now we focus on segmentation.

Protected-mode segmentation works as follows (see handout):

- segment register holds segment selector
- selector: 13 bits of index, local vs global flag, 2-bit RPL
- selector indexes into global descriptor table (GDT)
- segment descriptor holds 32-bit base, limit, type, protection
- la = va + base ; assert(va < limit);
- choice of seg register usually implicit in instruction
    - ESP uses SS, EIP uses CS, others (mostly) use DS
    - some instructions can take far addresses:
        - `ljmp $selector, $offset`
- GDT lives in memory, CPU's GDTR register points to base of GDT
- LGDT instruction loads GDTR
- you turn on protected mode by setting PE bit in CR0 register
- What about protection?
    - instructions can only r/w/x memory reachable through seg regs
    - not before base, not after limit
    - can my program change a segment register? yes, but... to one of the permitted (accessible) descriptors in the GDT
    - can my program re-load GDTR? no!
    - how does h/w know if user or kernel?
    - Current privilege level (CPL) is in the low 2 bits of CS
    - CPL=0 is privileged O/S, CPL=3 is user
    - why can't app modify the descriptors in the GDT? it's in memory...
    - what about system calls? how do they transfer to kernel?
    - app cannot **just** lower the CPL