Agenda

- Architectural overview
  - Features & limitations
- Hardware dependencies/features
Agenda

- Code review:
  - Headers and Process structures
  - Bootstrap procedure
  - Scheduling & Context switching
  - Sleep & Wakeup
  - Trap / Syscall mechanism
§ Architectural Overview
xv6 is a monolithic, preemptively-multitasked, multiprocessor-capable, 32-bit, UNIX-like operating system
some limitations:

- max addressable memory: 2GB
- few supported devices (e.g., no network)
- no support for kernel-level threading
limited syscall API:

<table>
<thead>
<tr>
<th>System call</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>fork()</td>
<td>Create process</td>
</tr>
<tr>
<td>exit()</td>
<td>Terminate current process</td>
</tr>
<tr>
<td>wait()</td>
<td>Wait for a child process to exit</td>
</tr>
<tr>
<td>kill(pid)</td>
<td>Terminate process pid</td>
</tr>
<tr>
<td>getpid()</td>
<td>Return current process's id</td>
</tr>
<tr>
<td>sleep(n)</td>
<td>Sleep for n seconds</td>
</tr>
<tr>
<td>exec(filename, *argv)</td>
<td>Load a file and execute it</td>
</tr>
<tr>
<td>sbrk(n)</td>
<td>Grow process's memory by n bytes</td>
</tr>
<tr>
<td>open(filename, flags)</td>
<td>Open a file; flags indicate read/write</td>
</tr>
<tr>
<td>read(fd, buf, n)</td>
<td>Read n bytes from an open file into buf</td>
</tr>
<tr>
<td>write(fd, buf, n)</td>
<td>Write n bytes to an open file</td>
</tr>
<tr>
<td>close(fd)</td>
<td>Release open file fd</td>
</tr>
<tr>
<td>dup(fd)</td>
<td>Duplicate fd</td>
</tr>
<tr>
<td>pipe(p)</td>
<td>Create a pipe and return fd's in p</td>
</tr>
<tr>
<td>chdir(dirname)</td>
<td>Change the current directory</td>
</tr>
<tr>
<td>mkdir(dirname)</td>
<td>Create a new directory</td>
</tr>
<tr>
<td>mknod(name, major, minor)</td>
<td>Create a device file</td>
</tr>
<tr>
<td>fstat(fd)</td>
<td>Return info about an open file</td>
</tr>
<tr>
<td>link(f1, f2)</td>
<td>Create another name (f2) for the file f1</td>
</tr>
<tr>
<td>unlink(filename)</td>
<td>Remove a file</td>
</tr>
</tbody>
</table>
very limited set of user-level programs:

- shell, cat, echo, grep, kill, ln, ls, mkdir, rm, wc
- no compiler/debugger/editor
- development (kernel/user) takes place on another platform!
§Hardware Dependencies / Features
xv6 runs on an x86 (Intel) processor, and relies on many of its hardware features e.g., privilege levels (kernel/user mode), interrupt vector & procedure, segmentation & paging (VM)
Recall: 2-bit current privilege level (CPL) flag
- CPL=3 ➞ “user” mode
- CPL=0 ➞ “supervisor/kernel” mode
  - guards special instructions & hardware
  - also restricts access to interrupt & VM structures
CPL is actually part of the `%cs` register, which specifies the *code segment* address. `%cs` and `%eip` (x86 PC) identify an instruction to execute *and its privilege level*
but CPL cannot be modified directly!

- lower (raise priority) via `int` instruction
- raise (lower priority) via `iret` instruction
**int** instruction (and h.w. interrupt) result in *interrupt descriptor table* (IDT) lookup

- fetches target `%cs` and `%eip` (aka “gate”) for corresponding handler
- restricts entry points into kernel
- install base address of IDT with `lidt`
xv6 also relies on x86 segmentation and paging to implement virtual memory
PROTECTED-MODE MEMORY MANAGEMENT

The segment, the segment type, and the location of the first byte of the segment in the linear address space (called the base address of the segment). The offset part of the logical address is added to the base address for the segment to locate a byte within the segment. The base address plus the offset thus forms a linear address in the processor's linear address space.

If paging is not used, the linear address space of the processor is mapped directly into the physical address space of the processor. The physical address space is defined as the range of addresses that the processor can generate on its address bus.

Because multitasking computing systems commonly define a linear address space much larger than it is economically feasible to contain all at once in physical memory, some method of "virtualizing" the linear address space is needed. This virtualization of the linear address space is handled through the processor's paging mechanism.

Paging supports a "virtual memory" environment where a large linear address space is simulated with a small amount of physical memory (RAM and ROM) and some disk.

Figure 3-1. Segmentation and Paging

Global Descriptor Table (GDT)

Segment Selector

Segment Base Address

Linear Address

Space

Segment Descriptor

Lin. Addr.

Page Table

Entry

Page Directory

Entry

Page

Dir

Table

Offset

Physical Address Space

Segmentation

Paging
Access checks can be used to protect not only against referencing an address outside the limit of a segment, but also against performing disallowed operations in certain segments. For example, since code segments are designated as read-only segments, hardware can be used to prevent writes into code segments. The access rights information created for segments can also be used to set up protection rings or levels. Protection levels can be used to protect operating-system procedures from unauthorized access by application programs.

3.2.4 Segmentation in IA-32e Mode

In IA-32e mode of Intel 64 architecture, the effects of segmentation depend on whether the processor is running in compatibility mode or 64-bit mode. In compatibility mode, segmentation functions just as it does using legacy 16-bit or 32-bit protected mode semantics.

Segment descriptors
PROTECTED-MODE MEMORY MANAGEMENT

FFFF_FFF0H. RAM (DRAM) is placed at the bottom of the address space because the initial base address for the DS data segment after reset initialization is 0.

3.2.2 Protected Flat Model

The protected flat model is similar to the basic flat model, except the segment limits are set to include only the range of addresses for which physical memory actually exists (see Figure 3-3). A general-protection exception (#GP) is then generated on any attempt to access nonexistent memory. This model provides a minimum level of hardware protection against some kinds of program bugs.

“Flat” model
IA-32 paging (4KB pages)
Page table and page directory entries are identical except for the D bit.

Physical Page Number

- **P**: Present
- **W**: Writable
- **U**: User
- **WT**: Write-through, **0**: Write-back
- **CD**: Cache Disabled
- **A**: Accessed
- **D**: Dirty (0 in page directory)
- **AVL**: Available for system use
§Demo & Code Review