Reminder: goal of “processes”

- Bring a program to life

- Give each process a private memory area
  - For code, data, stack
  - Illusion of its own dedicated machine

- Isolation + sharing:
  - Prevent each process from reading/writing outside its address space
  - But allow sharing when needed
HW & OS collaboration

♦ OS role
  ❖ (De)allocate physical memory of processes
    ▪ Create, grow, shrink remove
  ❖ Configure HW (in our case: memory)
  ❖ Multiplex HW (= allow for multiprogramming)
  ❖ Keep track of processes (executing or not)

♦ HW performs address translation & protection
  ❖ Translate user addresses to physical addresses
  ❖ Detect & prevent accesses outside address space
  ❖ Allow cross-space transfers (system calls, interrupts, …)

♦ Note that
  ❖ OS needs its own address space
  ❖ OS should be able to easily read/write user memory

HW & OS collaboration

♦ HW support not necessarily corresponds well to what OS wants
  ❖ For example: virtual memory management in
    Linux/PPC (with radix tree) vs. AIX/PPC (with hash)

♦ Two main approaches to x86 memory protection
  ❖ Segments & page tables

♦ Paging has won
  ❖ Most OSes are designed for paging

♦ Thankfully ☺, segments stopped being relevant for us; xv6
  ❖ <= rev4 implemented protection with segments
  ❖ rev5 changed that
  ❖ in rev6 students still needed to learn some segmentation
  ❖ we’re now at rev7 and no longer dealing with that
Case study: Unix v6

- Early Unix OS for DEC PDP11
  - By Ken Thompson & Dennis Ritchie, 1975
- From Bell labs;
  - 1st version to be widely used outside Bell
- Ancestor of all Unix flavors (Linux, *BSD, Solaris,…)
  - Much smaller
- Written in C
  - Recognizable (shell, multiuser, files, directories, …)
  - Today’s Unix flavors have inherited many of the conceptual ideas, even though they added lots of stuff (e.g., graphics) and improved performance

Case study: Unix v6

- 1976 Commentary by Lions: a classic….
  - “Lions’ Commentary on Unix 6th ed.”
- Despite its age
  - still considered excellent commentary on simple but high quality code
- For many years, was the only Unix kernel documentation publicly available
  - v6 allowed classroom use of source code; v7 & onwards didn’t
- Commonly held to be
  - one of the most copied books in computer science.
- Reprinted in 1996
  - still sold ($32.02 @ Amazon as of Mar 12, 2014)
**Case study: Unix v6**

- We could have used it for OSE

- But
  - PDP11
  - Old K&R C style
  - Missing some key issues in modern OSes (such as paging, multicore)

- Luckily...

---

**Case study: xv6**

- xv6 is an MIT reimplementation of Unix v6
  - Runs on x86 (if you insist)
    - We'll run it on top of QEMU
  - Even smaller than v6
  - Preserves basic structure (processes, files, pipes, etc.)
  - Runs on multicores
  - Got paging in 2011 😊

- To “get it”, you'll need to read every line in the source code
  - But it's really isn't that hard
  - The xv6 commentary book (see course website) will really help
Case study: xv6

- **First half of course**
  - Each lecture studies source code of one xv6 part
  - Should help in HW assignments (or vise versa)

- **About half-way the term**
  - You’ll understand the source code for one well-designed OS for an Intel-based machine

- **2nd half**
  - Covers OS concepts invented after Unix v6
  - We’ll typically discuss research papers targeting these concepts
  - No tutorials from that point…

---

xv6 – why?

- **Q: why study an aging OS?**
  - Instead of, say, Linux, or Windows, or FreeBSD, or Solaris, …

- **A1: it’s big enough**
  - To illustrate basic OS design & implementation
  - Yet is far smaller => much easier to understand

- **A2: it’s similar enough**
  - To those other modern OSes
  - Once you’ve explored xv6, you’ll find your way inside kernels such as Linux

- **A3: as noted, it’ll help you**
  - To build your own (J)OS
OS *engineering*

- **JOS**
  - Occupies a very different point in the design & implementation space from xv6
- **Types of OSes**
  - Micro kernel
    - QNX, L4, Minix
  - Monolithic kernel
    - xv6, Unix family (Linux, FreeBSD, NetBSD, OpenBSD, Solaris/SunOS, AIX, HPUX, IRIX, Darwin), Windows family
    - Actually, most are nowadays hybrid
  - Exokernel
    - JOS

The 1st process

- **In previous tutorial**
  - Chapter #1 in xv6 commentary (see course webpage)
- **This lecture:**
  - Chapter #2
From the perspective of the boot sequence

IMPLEMENTING VIRTUAL MEMORY IN XV6

xv6 address space

◆ xv6 enforces memory address space isolation
  ❖ No process can write on another’s space or the kernel’s
◆ xv6 does memory “virtualization”
  ❖ every process’s memory starts at 0 & (appears) contiguous
  ❖ C (as well as the linker) expect contiguity
◆ xv6 does simple page-table tricks
  ❖ Mapping the same memory (kernel’s) in several address spaces
  ❖ Mapping the same memory (kernel’s) more than once in one address space
    ▪ Kernel can access user pages using user virtual addresses (0…)
    ▪ Kernel can also access user pages through kernel’s “physical view” of memory
  ❖ Guarding a user stack with an unmapped page
xv6 address space

- **x86 defines three kinds of memory addresses**
  - Virtual (used by program), which is transformed to
  - Linear (accounting for segments), which is transformed to
  - Physical (actual DRAM address)
- **xv6 nearly doesn’t use segments**
  - All their bases set to 0 (and their limits to the max)
  - virtual address = linear address
  - Henceforth we’ll use only the term “virtual”

Reminder: paging in a nutshell

- **Virtual address (we assume 32bit space & 4KB pages):**
  - Given a process to run, set cr3 = pgdir (“page directory”)
  - Accessing pgdir[pdx], we find this PDE (page directory entry)
  - Flags (bits): PTE_P (present), PTE_W (write) , PTE_U (user)
  - 20bits are enough, as we count pages of the size $2^{12}$ (=4KB)
  - The “page table” pgtab = pgdir[pdx] & 0x ffff f000
    - An actual physical address
  - Accessing pgtab[ptx], we find this PTE (page table entry)
  - Target physical address =
    - (pgtab[ptx] & 0x ffff f000) | (virt_adrs & 0x 0000 0fff)
xv6 virtual memory

- Each process has its own unique pgdir (= address space)
- When the process is about to run
  - cr3 is assigned with the corresponding pgdir
- Every process has at most KERNBASE (=2GB) memory
  - (Actually less, since we assume PHYSTOP = 224 MB)
- Kernel maps for itself the entire physical memory as follows:
  - VA: \( KERNBASE \ldots KERNBASE+PHYSTOP \)
    mapped to PA: \( 0 \ldots PHYSTOP \)
- Such mapping exists in every v-space of every process
  - PTEs corresponding to addresses higher than KERNBASE have the PTE_U bit off, so processes can’t access them
- Benefit:
  - Kernel can use each process v-space to access physical memory
  - There exists a simple mapping from kernel v-space to all physical
    \[ PA = VA - KERNBASE \]
**xv6 virtual memory**

- **Assume**
  - Process $P$ size if 12KB (3 pages)
  - $P$ sbrk-s (dynamically allocates) another page

- **Assume the free page xv6 decides to give $P$ is in PA:**
  - 0x 2010 0000

- **Thus, to ensure contiguity, the 4th PTE of $P$...**
  - (Which covers VAs: 0x 0000 3000 – 0x 0000 3fff)
  - (4096 = $16^3 = 0x 0000 1000$)

- **...should be mapped to physical page**
  - 0x201000 (= the upper 20 bits of PA 0x 2010 0000)

- **So 2 different PTEs now refer to PA = 0x 2010 0000**
  - kernel: PTE of VA = KERNBASE + 0x 2010 0000, and
  - process: PTE of VA = 0x 0000 3000

- The kernel can use both

---

**Code: entry page table**

- **In virtual space, kernel starts in**
  - **line:** 0208 (memlayout.h)
  - KERNLINK = KERNBASE + EXTMEM
    - = 2GB + 1MB = 0x 8010 0000

- **Why KERNBASE (= 2GB) so high in v space?**
  - b/c kernel v space mapped in each process v space
  - leave enough room to allow process v space to grow

- **Boot loader loads xv6 kernel into physical address:**
  - 0x 0010 0000 (= 1MB)
    - Why not at physical address 0x 8010 0000 (where, in terms of VA, the kernel expects to find its instructions & data)?
      - Because it might not exist
    - Why not at 0x0?
      - First 1MB for legacy devices
First page directory

- **PTE structure & bits (mmu.h)**
  - line: 0805

- **entrypgdir (main.c)**
  - The first page directory
  - line: 1311
  - Used by...

- **(See question in next page)**

- **entry (entry.S)**
  - Boot-loader loads xvs6 from disk & starts executing here
  - line: 1040 ... 1061 (V2P_WO at 0220 – sheet 2)
Creating an address space

- **main**
  - line: 1217
- 2nd line of main: `kvmalloc` (we'll get back to the 1st)
  - in vm.c (implementation of virtual memory)
  - line: 1757; switch to page table that maps all memory
  - kmap array (1728) + `setupkvm` (1737)
    => `mappages` (1679)
    => `walpgdir` (1654)
Physical memory allocator

- **Maintain a free-list of all free 4KB-pages in system**
  - From end of kernel to PHYSTOP
- **Bootstrap problem**
  - Entire physical memory must be mapped in order for the allocator to initialize the free list
  - But creating a page table with those mappings involves allocating a hierarchy of page-table pages
  - xv6 solves this problem by using a separate page allocator during entry, which allocates memory just after the end of the kernel's data segment
  - This allocator does not support freeing & is limited by 4 MB mapping in the `entrypgdir`, but our kernel is small enough for it to be sufficient to allocate the first kernel page table
Physical memory allocator

- The 2 free lists (for init phase, and for normal runs)
  - File: kalloc.c = the kernel’s allocator
  - Representation: `struct run` (2764) and `kmem` (2772)
    - ‘next’ saved in chained pages
  - `kinit1` (2780), `kinit2` (2788)
    - Called from `main` (rows: 1219, 1238)
  - `freerange` (2801), `kfree` (2815)
  - Allocation of a page: `kalloc` (2838)

---

Homework

- Read pages 30 – 32 in xv6 commentary
### User part of an address space

<table>
<thead>
<tr>
<th>Offset</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>Stack</td>
</tr>
<tr>
<td></td>
<td>Guard page</td>
</tr>
<tr>
<td></td>
<td>Data</td>
</tr>
<tr>
<td></td>
<td>Text</td>
</tr>
<tr>
<td></td>
<td>Heap</td>
</tr>
</tbody>
</table>

- **KERNBASE**: Base of the kernel address space.
- **PAGESIZE**: Size of a page.
- **(unmapped)**: Unmapped region of the address space.

### Arguments

- **Argument 0**: Nul-terminated string
- **Address of Argument 0**: `argv[0]`
- **Address of Argument N**: `argv[N]`
- **Address of Address of Argument 0**: `arg0`
- **Address of Address of Argument N**: `argc`
- **Return PC for Main**: `0xffffffff`

**END**
Process creation

- **struct proc (lines: 2053-2067)**
  - Why do we need a kernel stack for every process?
  - Why can’t the kernel use the user stack?
- **main (line 1237) => userinit (line 2202) => allocproc (line 2155)**
  - userinit creates the first process (init) using allocproc
  - allocproc creates all processes (used by fork)
- **allocproc (lines: 2155-2194)**
  - find empty proc entry
  - allocate pid and set state
  - create the kernel stack of the new process as in the next slide
- **userinit (lines: 2202-2226)**
  - inituvn (1786), initcode.S (7500)

---

![Diagram of process creation](image)
Running a process

* main => userinit => mpmmain (1267) => scheduler (2408)
  => switchuvm (1764)