Sistemas Operativos

xv6

Marcelo Arroyo

Dpto. de Computación - FCEFQyN - Universidad Nacional de Río Cuarto

2014

xv6

Build system

1. The build system (see `Makefile`) construct two files:
   - `xv6.img`: Disk image (for booting) containing boot sector and kernel code and data.
   - `fs.img`: Filesystem disk image containing some user data and program files and free blocks (see `mkfs.c`).

Running xv6 with PC software emulators

- Bochs: `make bochs` (see `.bochsrc`)
- qemu: `make qemu`

  `qemu -serial mon:stdio -hdb fs.img xv6.img -smp 2 -m 512`

  (xv6.img as IDE disk 0, fs.img as IDE disk 1, 512 Mb RAM and two cpus)

Running xv6

Booting (see `bootasm.S` and `bootmain.c`)

1. On power-on the IBM-PC runs on 8086 mode (16 bits, no protection).
2. PC BIOS load the first block (512 bytes sector) from first IDE disk at physical address `0x7c00` and jumps to there.
3. Boot loader set initial segments and set CPU in *protected mode*.
4. Enabling A20 line, the cpu can handle addresses of 20 bits.
5. Load the kernel (starting at block 1) at address 0x100000 (EXTMEM).
6. Jumps to `_start` (physical address `0x10000c`) (see `entry.S`)

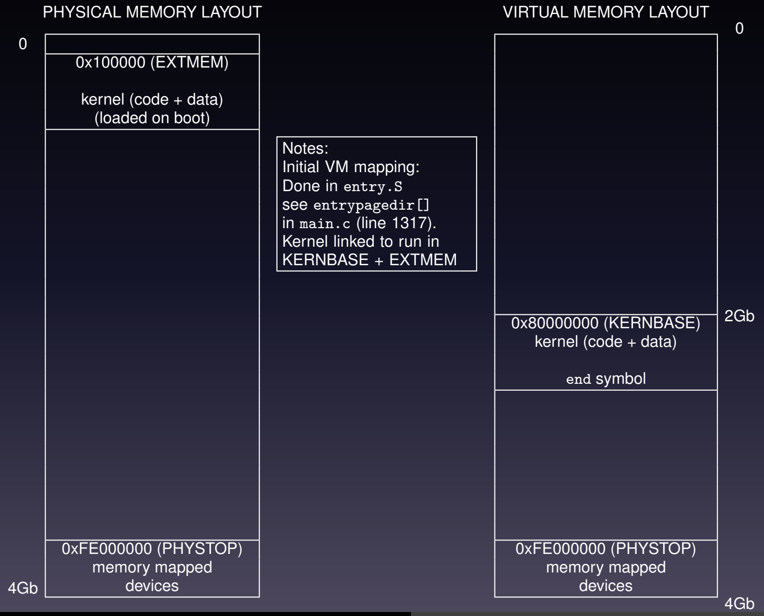
Configurating initial memory layout

1. Kernel is linked at high addresses, starting at `0x80100000` (above 2Gb).
2. So, `entry` routine have to prepare an initial virtual memory mapping:

   $$\begin{array}{lcl} [0x0, 4Mb) & \rightarrow & [0, 4Mb) \\ [0x80000000, 0x80000000 + 4Mb) & \rightarrow & [0, 4Mb) \end{array}$$

   The first mapping for `entry` code (running at low virtual addresses).
   The second mapping for running the rest of kernel code and data (high virtual addresses).
   This mapping is on page table `entrypgdir` (line 1317 in `main.c`).
3. Note this is a mapping using *supertables* (4Mb size pages).
4. `entry` enables pagging, set an initial stack pointer (`esp`) and calls `main()`.

xv6 memory layout in `main()`



xv6: Initialización (`main()`)

1. Set kernel page table (`kvmalloc()`)
2. Set *local apic* (timer) of each CPU (`lapicinit()`)
3. Set per-cpu segment descriptor tables (`seginit()`)
4. Initialization of interrupt controllers (`picinit()`,`ioapicinit()`)
5. Initialization of console and serial (uart) device drivers
6. Setting process table
7. Set interrupt vector (`trapinit()`)
8. Initializacion filesystem subsystem and IDE disk driver
9. Starting others CPUs (`startothers()`) and setting scheduler stacks.
10. Set kernel memory allocator (`kinit()`)
11. Load and set first user process `init` (`userinit()`)
12. Each cpu executing `scheduler()` (on each stack)

xv6: Setting segments

Per-CPU state: GDT settings (`seginit()`)

Description	FLAGS	Base Address	Size	DPL
Kernel code	R-X	0	0xffffffff	0
Kernel data	-W-	0	0xffffffff	0
User code	R-X	0	0xffffffff	3
User data	-W-	0	0xffffffff	3
(cpu, proc)	-W-	$\&c \rightarrow cpu$	8	0

- Code and data segments maps full memory.
- *Descriptor Privilege Level* (0=kernel, 3=user).
  So, on each interrupt processor whill change mode.
- `gs` points (index) to *(cpu,proc)* segment secriptor.
  It maps per-cpu variables current `proc` and `cpu`.

xv6: Kernel main data structures

CPU state

- Global array `struct cpu cpus[NCPU]` defined in `mp.c` (line 6213)
- Each element (`struct cpu`) contains:

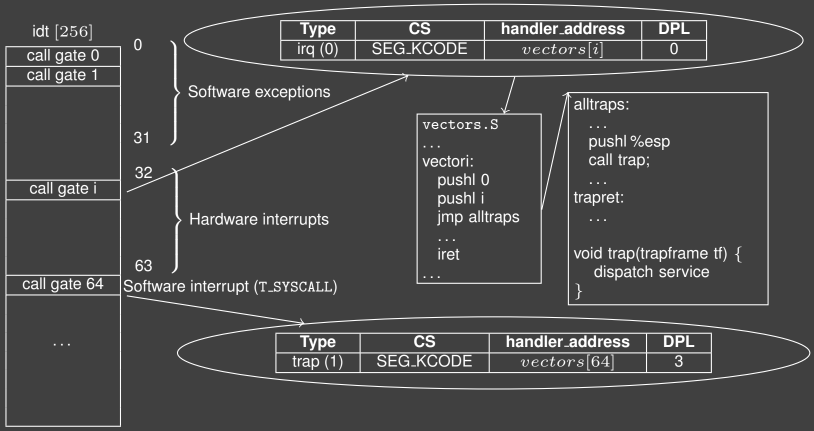
  `id`: cpu identifier
  `scheduler`: scheduler saved context (in scheduler stack)
  `ts`: Task State register (info on where jump on interrupt)
  `gdt`: Segment Global Descriptor Table
  ...
  `per-cpu local variables`:

    `cpu`: pointer to current cpu state
    `proc`: pointer to current proc in this cpu (`%gs:4`)

  Note: These two variables are mapped by `gs` segment selector:
  `*cpu=%gs:0=&cpus[cpunum()]`,
  `*proc=%gs:0=cpus[cpunum()].proc`

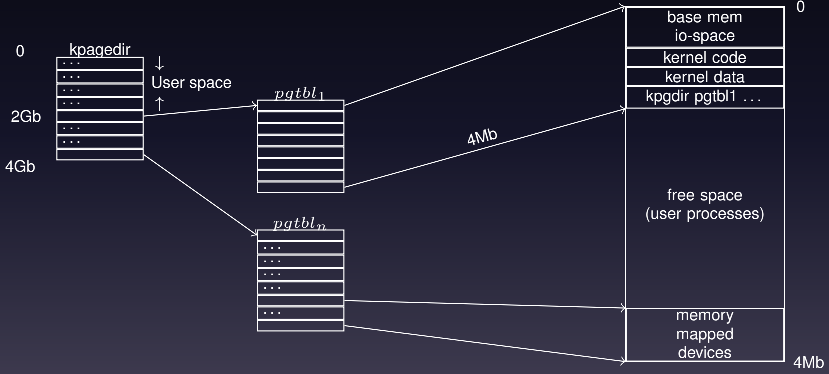
xv6: Kernel main data structures

Interrupt Descriptor Table (interrupts vectors)

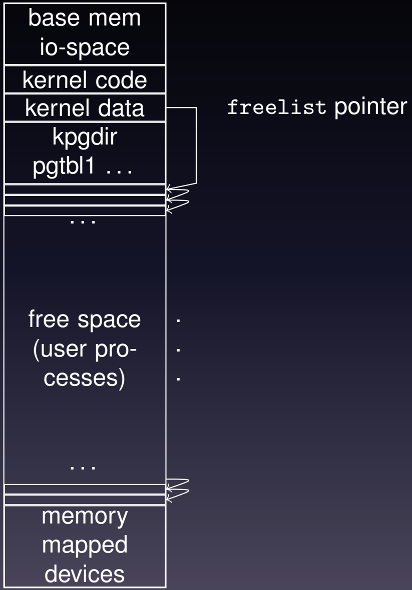


xv6: vm mapping at main: line 1219

This layout is generated by `kvmalloc()`. Most of work is done by `setupkvm()` (line 1734).



xv6: kernel heap



`kinit` (in `kalloc.c`) set the *freelist*.

xv6: Processes

- The `struct proc` (in `proc.h`) represents an user process:
  - `sz`: Amount of memory used.
  - `pgdir`: Pointer to page directory (vm mapping).
  - `kstack`: Pointer to bottom of stack when running in *kernel mode*.
  - `state`: Process state (`RUNNING, SLEEPING, ...`)
  - `pid`: Process identifier.
  - `parent`: Pointer to parent process.
  - `tf`: Pointer to `trapframe` (interrupt state on `kstack`).
  - `context`: Pointer to some saved cpu registers (sed by `switch`).
  - `chan`: Pointer to some data (representing the waiting channel).
  - `killed`: Non-zero if process has been killed.
  - `ofile`: Array of pointers to opened files.
  - `cmd`: Pointer to `inode` of current directory.
  - `name`: Program name.

xv6: Processes (cont.)

- When running in user mode, it use the user stack.
- When an interrupt occurs, the cpu switch to the kernel stack (because the setting of the *task segment* of current cpu).

How we can view a process?

As a program running with two alternating threads

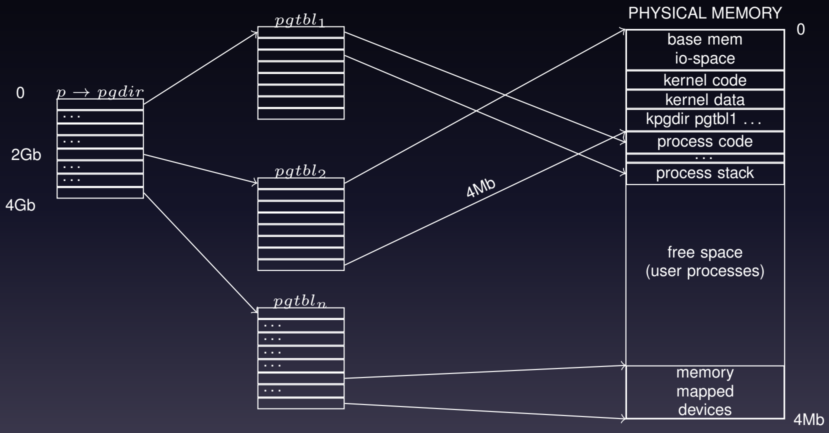
- One running in user mode

  Cpu running in ring 3 (using segments `SEG_UCODE` and `SEG_UDATA`)
  Using user stack

- Other when running in kernel mode

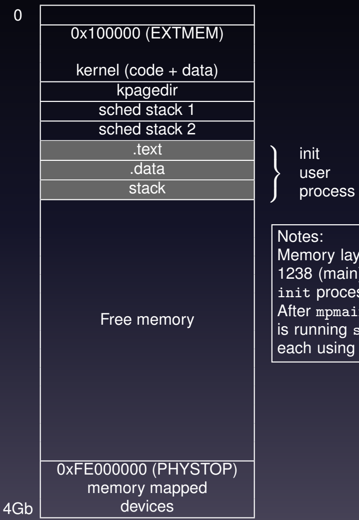
  Cpu running in ring 0 (using segments `SEG_KCODE` and `SEG_UDATA`)
  Using process's kernel stack (`esp points to` $p->kstack$ page area)

xv6: vm mapping for a process



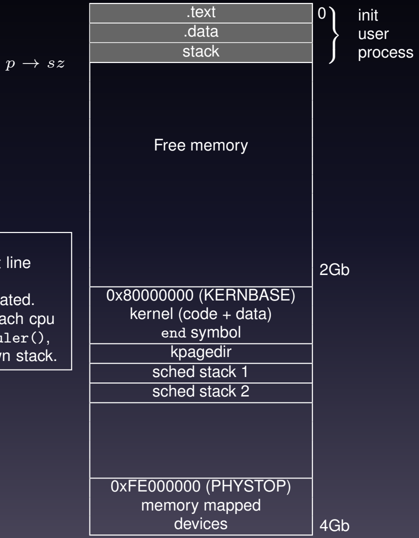
xv6: memory layout after `userinit()`

PHYSICAL MEMORY LAYOUT



Notes:
Memory layout at line 1238 (main).
`init` process created.
After `mpmain()` each cpu is running `scheduler()`, each using its own stack.

VIRTUAL MEMORY LAYOUT



Process creation steps

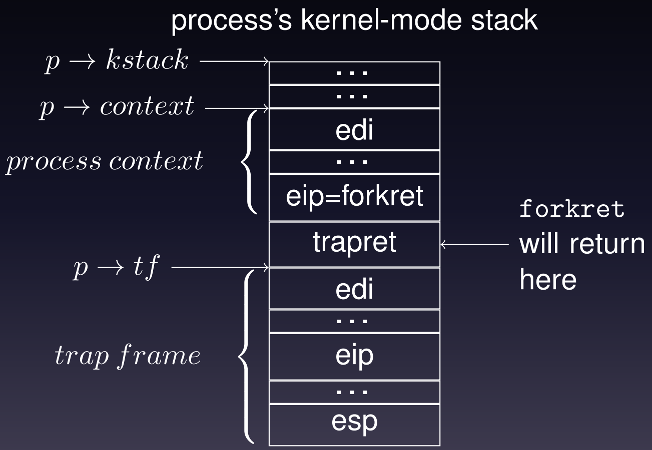
Idea: simulate on process's kernel-mode stack an interrupt state.

Process's kernel-mode stack layout prepared to switch to process code.

- `fork()` (or `userinit()`[1]) calls `allocproc()`.
- `allocproc()` steps:
  1. Find `ptable[]` unused entry.
  2. Allocate kernel-mode stack.
  3. Setup trapframe and process context on stack.
- `fork()` (or `userinit()`) further steps:
  1. `setupkvm()` set paging ($p \rightarrow pgdir$) mapping kernel code+data.
  2. Allocate, map and copy code+data+stack (see `setupuvm()`, `copyuvm()`).

[1] It is called once in `main` to create first process.

Process's kernel-mode stack layout



Context switch: kernel to process

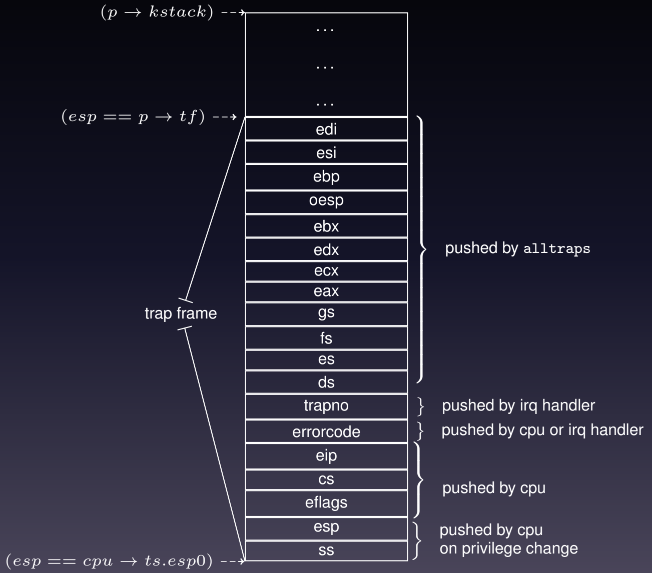
- After calling `mpmain()` (line 1239), xv6 is running `scheduler()` on each cpu (each with its own stack).
- `scheduler()` is in an infinite loop:
  1. It loops until find an `RUNNABLE` process (`p`).
  2. Set `proc = p` (now, the current process in this cpu).
  3. Set `tss` and use process page directory table (`switchuvm()`).
  4. Change `state` of current process (`RUNNING`).
  5. Switch $cpu \rightarrow scheduler$ context to $p \rightarrow context$:
     1. Save (push) cpu registers (*context*) in scheduler stack (remember it in $p \rightarrow scheduler$)
     2. make $esp = p- > context$
     3. load (pop) $p- > context$ (from kernel-mode stack) in cpu registers
  6. Return of `switch()` cause cpu fallback to `trapret` (see stack layout in previous slide).
  7. `trapret` returns from interrupt and now process is running in user mode.

From process to scheduler

When process is interrupted or it does a system call:

1. CPU push `eflags,cs,eip`; set `eip=irq handler code`.
2. Push `errorcode` (by CPU or IRQ handler).
3. IRQ handler push IRQ number (to know what happened).
4. IRQ handler jumps to `alltraps`.
5. `alltraps` save (push) other cpu registers.
6. Change `ds`, `es` to kernel data segment.
7. Change `gs` to (cpu,proc) local cpu variables.
8. Call `trap(esp)`.
9. `trap(tf)` calls to service routine (syscall or device driver).
10. Later, kernel calls `sched()` which calls `switch()`:
    1. change from process context to scheduler context.
    2. Return of `switch()` cause cpu fallback to `scheduler()` (line 2429).

xv6: Interrupt state (`trapframe`)



xv6: first user process

Created by `userinit()`

- Initial user program code is `initcode.S`, linked with kernel code but its symbols are defined as low addresses.
- In does the system call `exec(‘‘init’’,‘‘’’)`.

what does `exec()`?

1. Reuse current process entry in processes table, leaving file descriptors and others members (`pid...`) untouched.
2. Allocates and map memory pages and load `.text,.data` segments from ELF file.
3. Set two pages for stack (the first one is to detect *overflow*).
4. Push command arguments on user stack (for `main()`).
5. Free old memory and switch to new *vm map* ($p \rightarrow pgdir$).