Midterm

- You have to bring a hard copy of xv6-code-listing to the examination hall
- You can bring any additional reference (books, notes, etc.)
- Electronic gadgets are not allowed

Boot loader

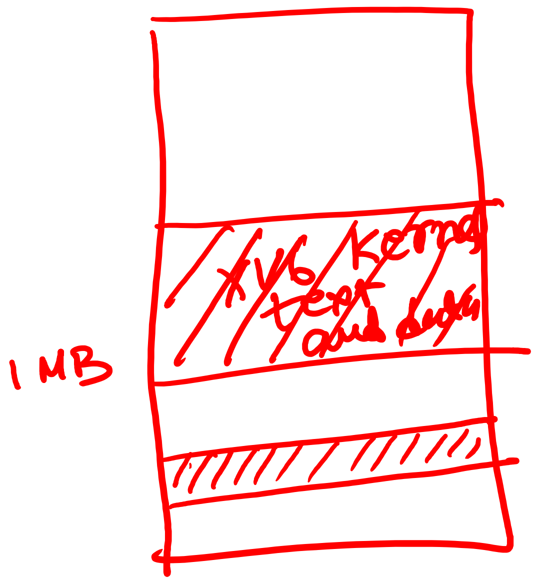
- When the PC starts, it loads the boot loader into a predefined location in physical memory

- The boot loader loads the kernel text and data at the physical address 0x100000 (1 MB)

- The kernel code is compiled to be loaded at location 0x80100000
  - Why does boot loader not load the kernel at 0x80100000?

Boot loader

xv6



entry:1144

- After loading the kernel the boot loader jumps to entry:1144
- Notice that entry:1144 is the kernel code and it is compiled for addresses greater than 2 GB
- The entry:1144 loads a page table that maps kernel at 0x80100000
    - How do we load a page table when the current EIP is a physical address?

entry:1144

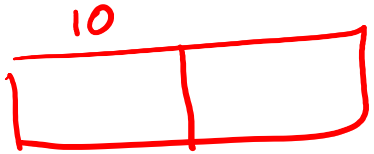
entry:1144

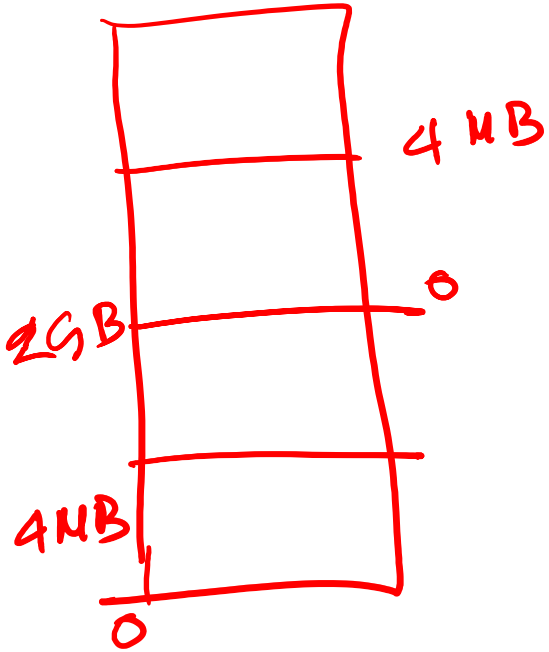
- entrypgdir:1411

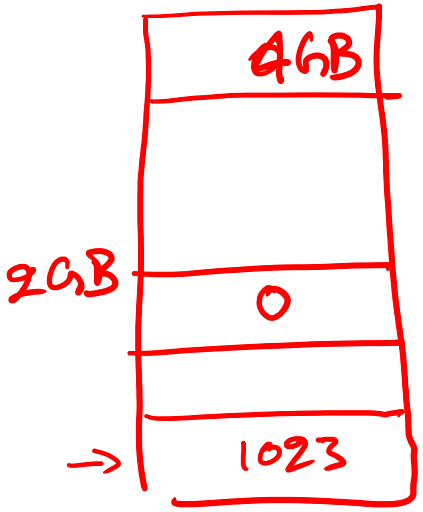
0x100000 ✓

0x80100000 ✓

0x100018 → mov %eax, %cr3







EIP = (1MB - 4MB)

entrypgdir:1411

- Maps 4 MB physical page (0 – 4MB), at virtual addresses 0 and 0x80000000 (2GB)

1167 : char stack [STACKSIZE]

1158 : %esp = & stack[STACKSIZE]

entry:1144

- entry loads entrypgdir in the cr3 register
  - At this point the paging hardware is active
  - The virtual pages in the range 0:0x400000 and 0x80000000:0x80400000 refer to the same physical pages
  - The entire xv6 kernel and data can fit into 4 MB
  - Let us call 0:0x400000 range as low addresses
  - Let us call 0x8000000:0x80400000 range as high addresses

- Even after loading the page table the current EIP is still a low address

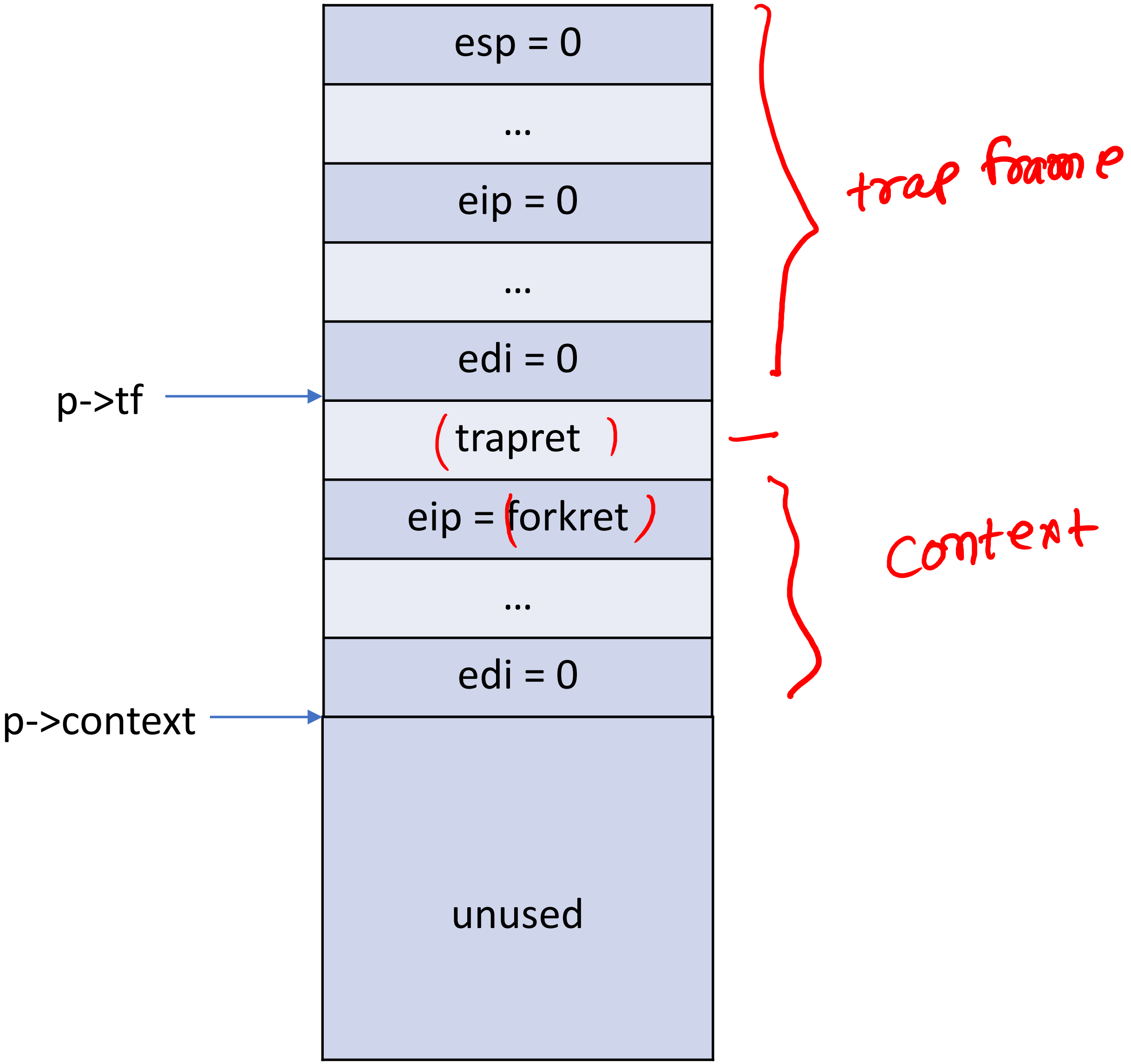
entry:1144

- entry uses global variable "stack" as a kernel stack
- entry loads the address (stack+KSTACKSIZE), which is a high address, in register %esp
- entry loads the address of the main in %eax register, which is a high address
- entry does an indirect jump through %eax register to jump to main
- After this step all the addresses are high address

Creating the first process

- userinit : 2502
- allocproc: 2455
  - allocate a process control block (PCB)
    - size, kernel stack, page directory, context, pid, trap frame, state, open files, etc.
  - allocate kernel stack
  - allocate space on kernel stack for trap frame and process context

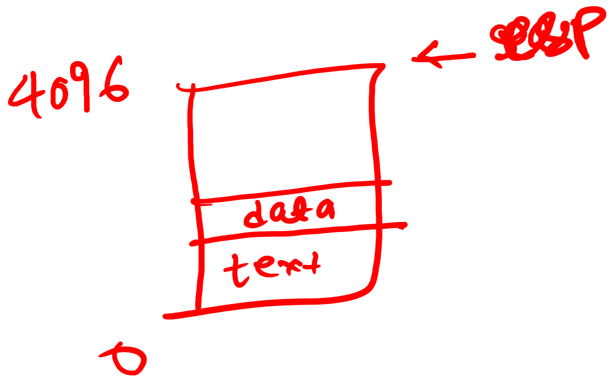
kernel stack in allocproc

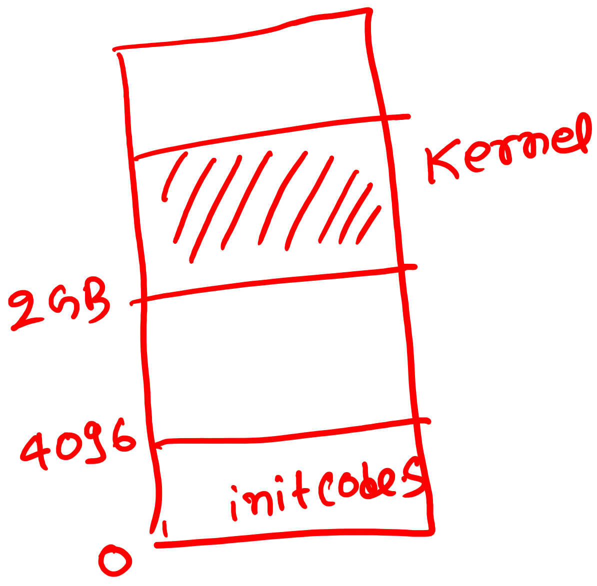


eip
ebp
ebx
esi
edi

userinit : 2502

- create page directory and map kernel pages
- load “initcode.S” at the first virtual page in the user address space
- set up the trapframe

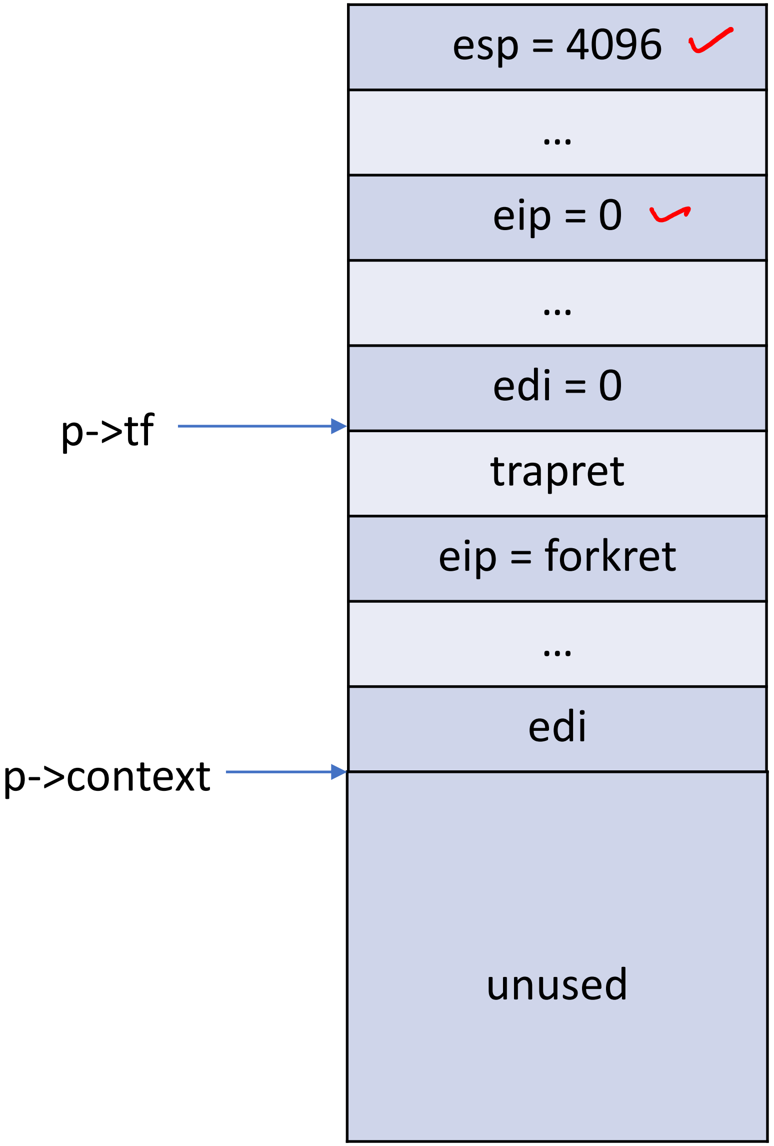




initcode.S:8408

- Doesn't take any argument
- Does exec system call to load "init" executable
- Entire code, data, and stack of initcode.S fit into 4096 bytes

kernel stack after userinit

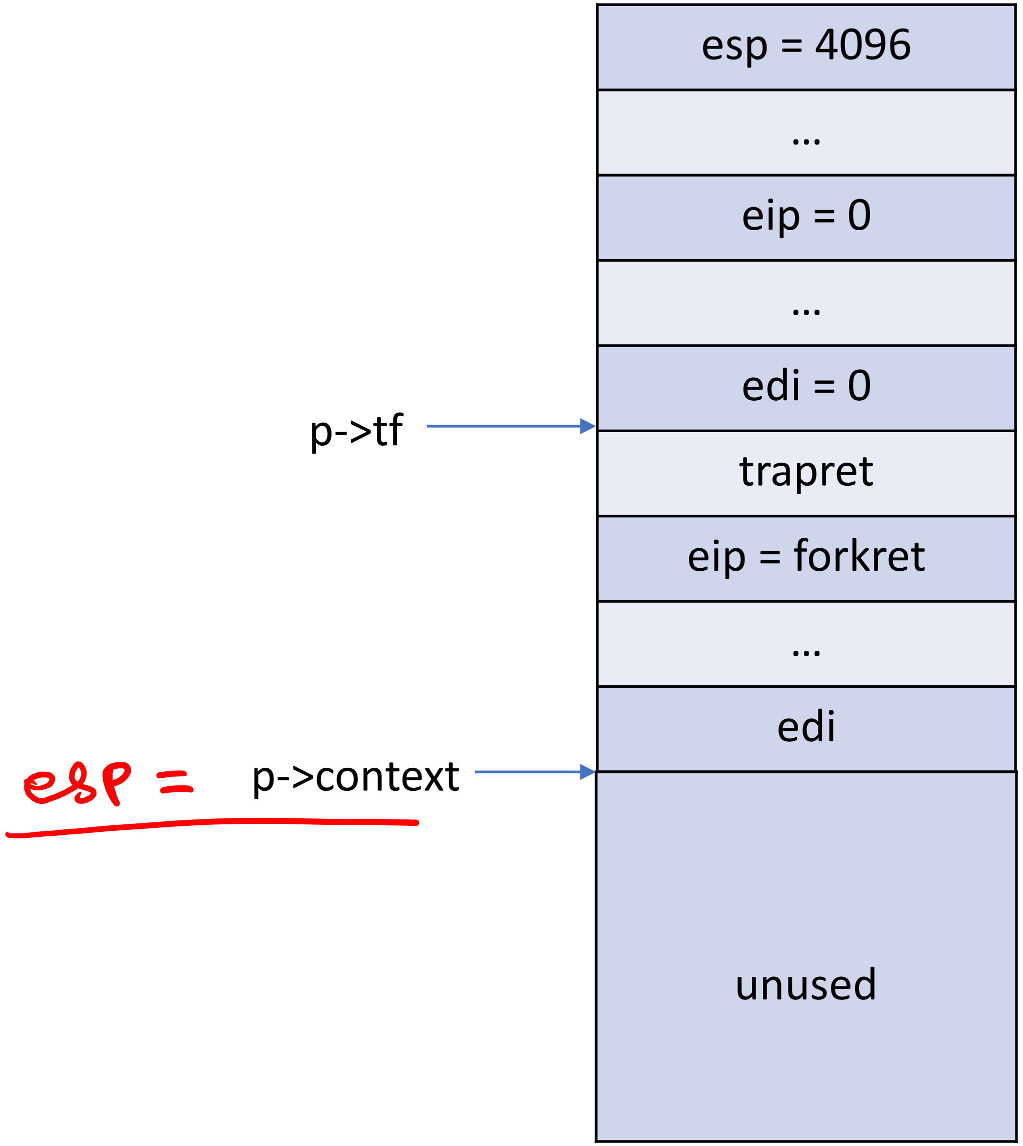


Schedule

- After userinit at some point main calls mpmain that eventually calls the scheduler:2708

- scheduler:2708 finds a runnable process, switch the page table and call swtch:2958 to load the new process context

swtch:2958



swtch (scheduler, target process

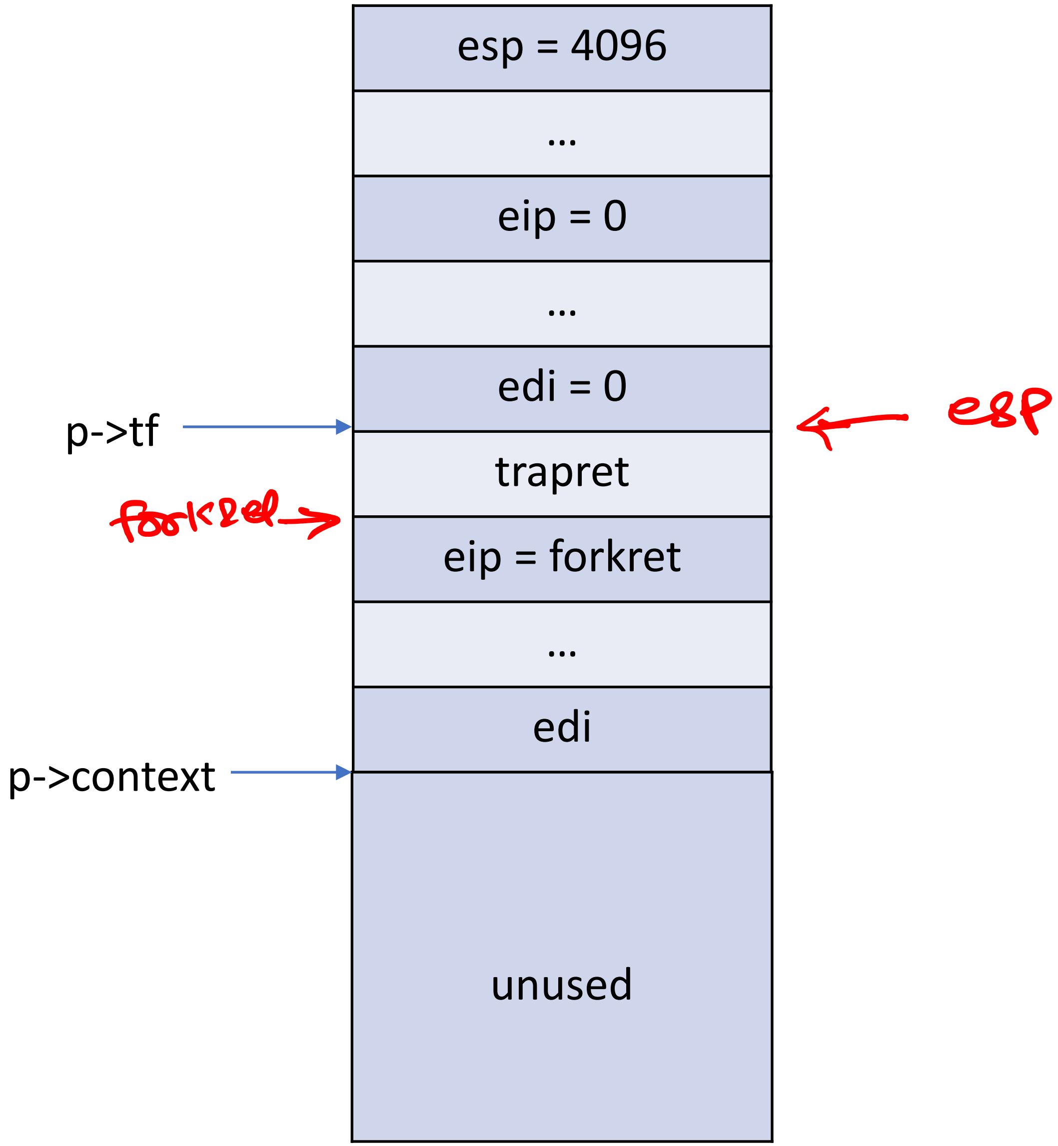
struct context
{
edi;
esi;
ebx;
ebp;
eip;
}

edi, esi, ebx, ebp

forkret:2788

- Release scheduler lock and return

forkret


esp = 4096
...
eip = 0
...
edi = 0
esp
p->tf
trapret
forkret
eip = forkret
...
edi
p->context
unused

trapret:3277

- Pops the trap frame and executes iret

Schedule

- How does scheduler:2708 regain control after calling swtch
  - call to swtch from scheduler saves the scheduler context in cpu->scheduler

  - yield:2777 calls sched:2758 that saves the current process context in “proc->context” and loads the scheduler context from cpu->scheduler