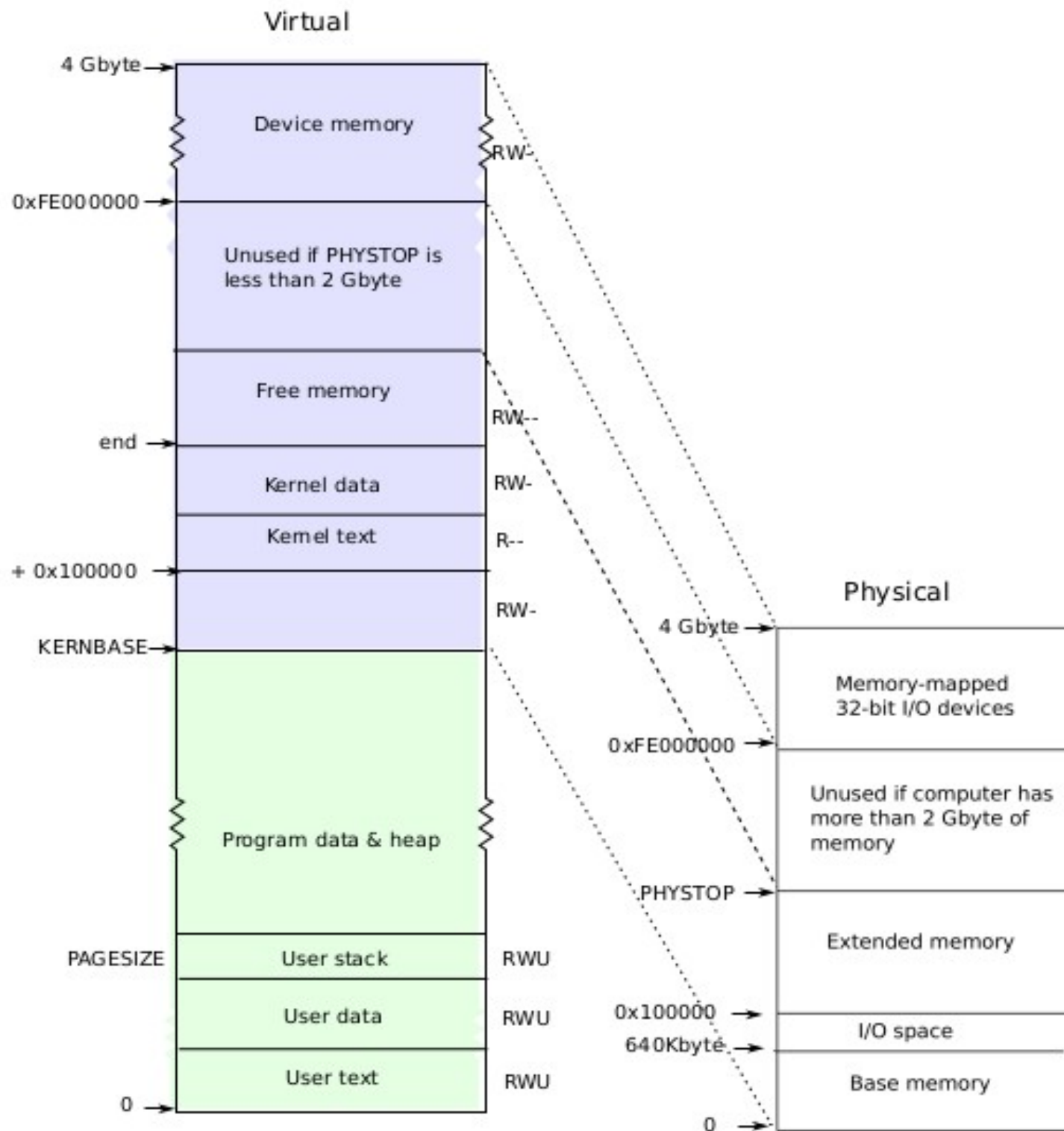


Processes in xv6 code

Process Table

```
struct {  
    struct spinlock lock;  
    struct proc proc[NPROC];  
} ptable;
```

- One single global array of processes
- Protected by ptable.lock

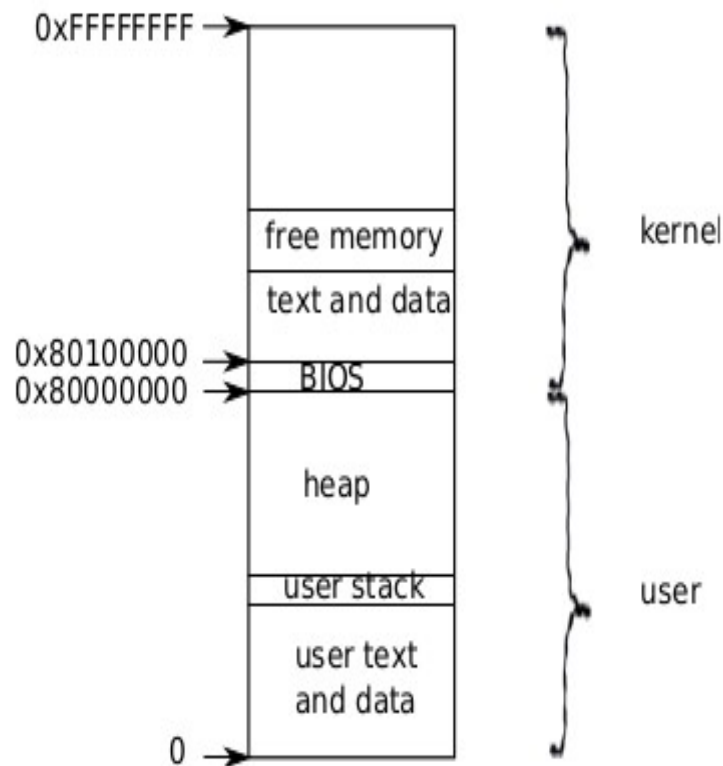


**Layout of
process's
VA space**

**xv6
schema!**

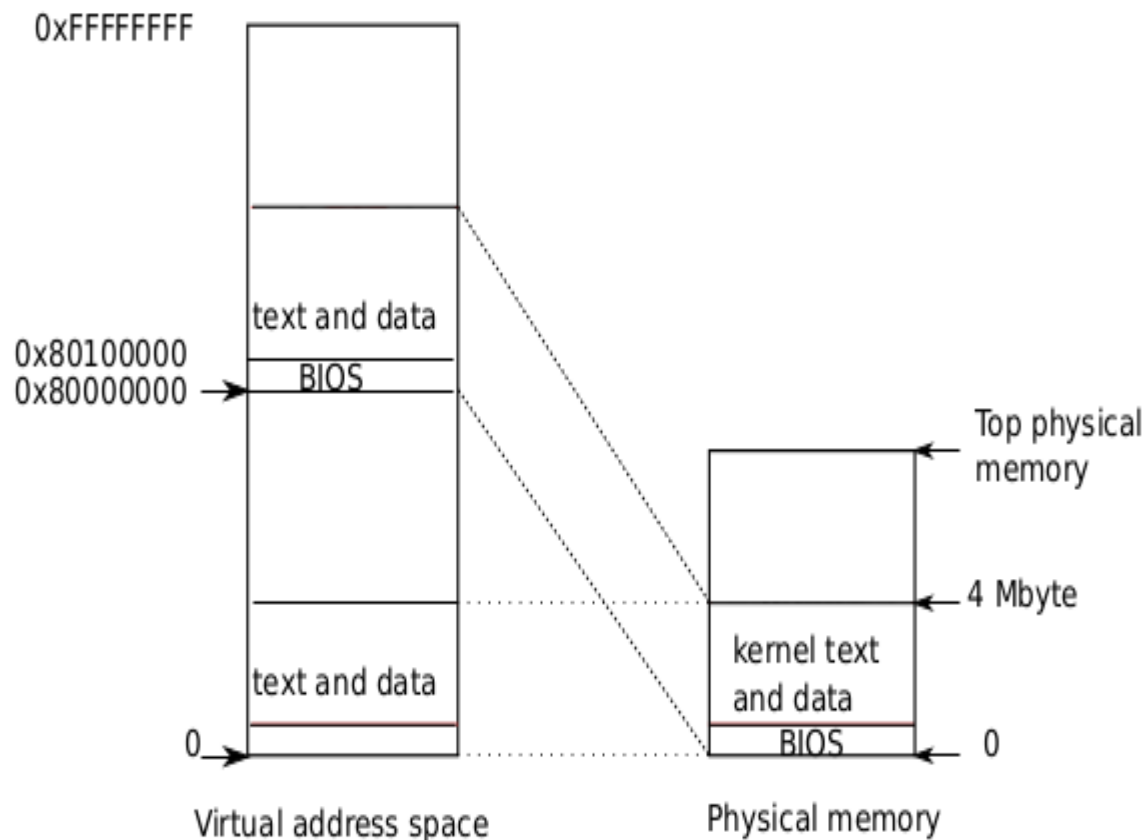
**different
from Linux**

Logical layout of memory for a process



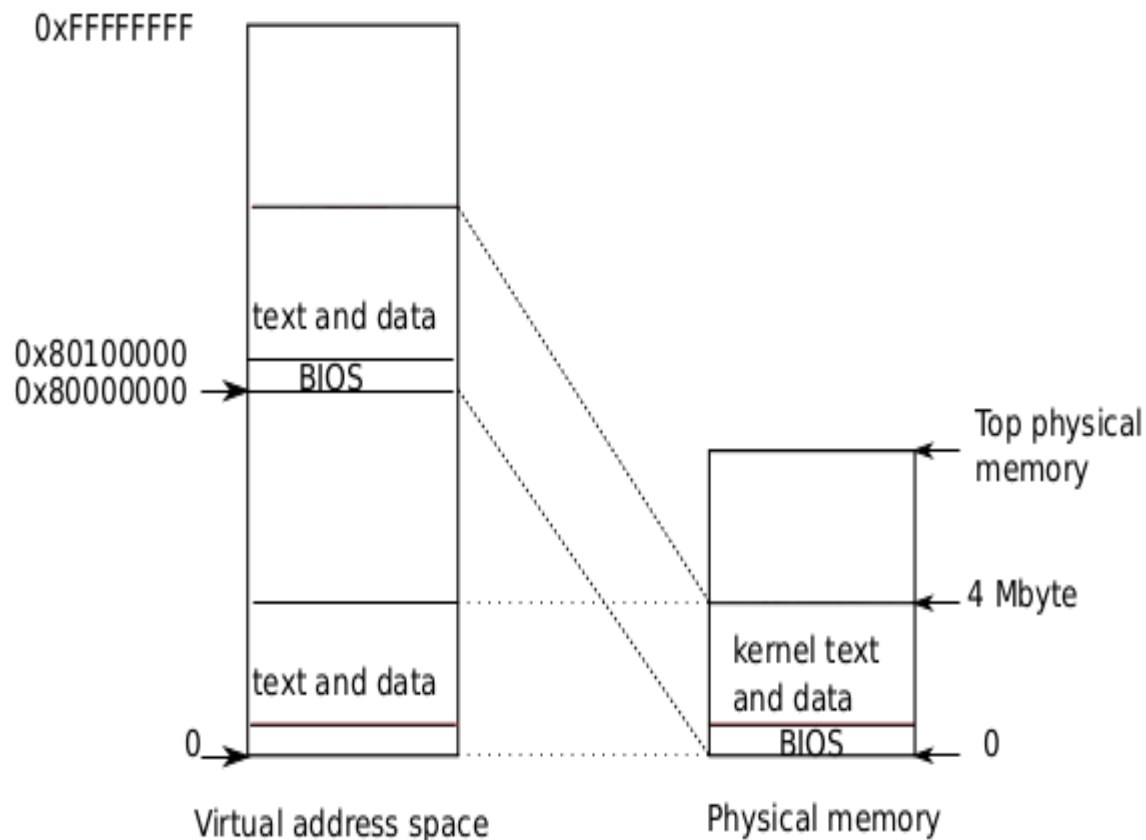
- Address 0: code
- Then globals
- Then stack
- Then heap
- Each process's address space maps kernel's text, data also --> so that system calls run with these mappings
- Kernel code can directly access user memory now

Kernel mappings in user address space actual location of kernel



- Kernel is loaded at 0x100000 physical address
- PA 0 to 0x100000 is BIOS and devices
- Process's page table will map
VA 0x80000000 to PA 0x00000 and
VA 0x80100000 to 0x100000

Kernel mappings in user address space actual location of kernel



- Kernel is not loaded at the PA 0x80000000 because some systems may not have that much memory
- 0x80000000 is called **KERNBASE** in xv6

Imp Concepts

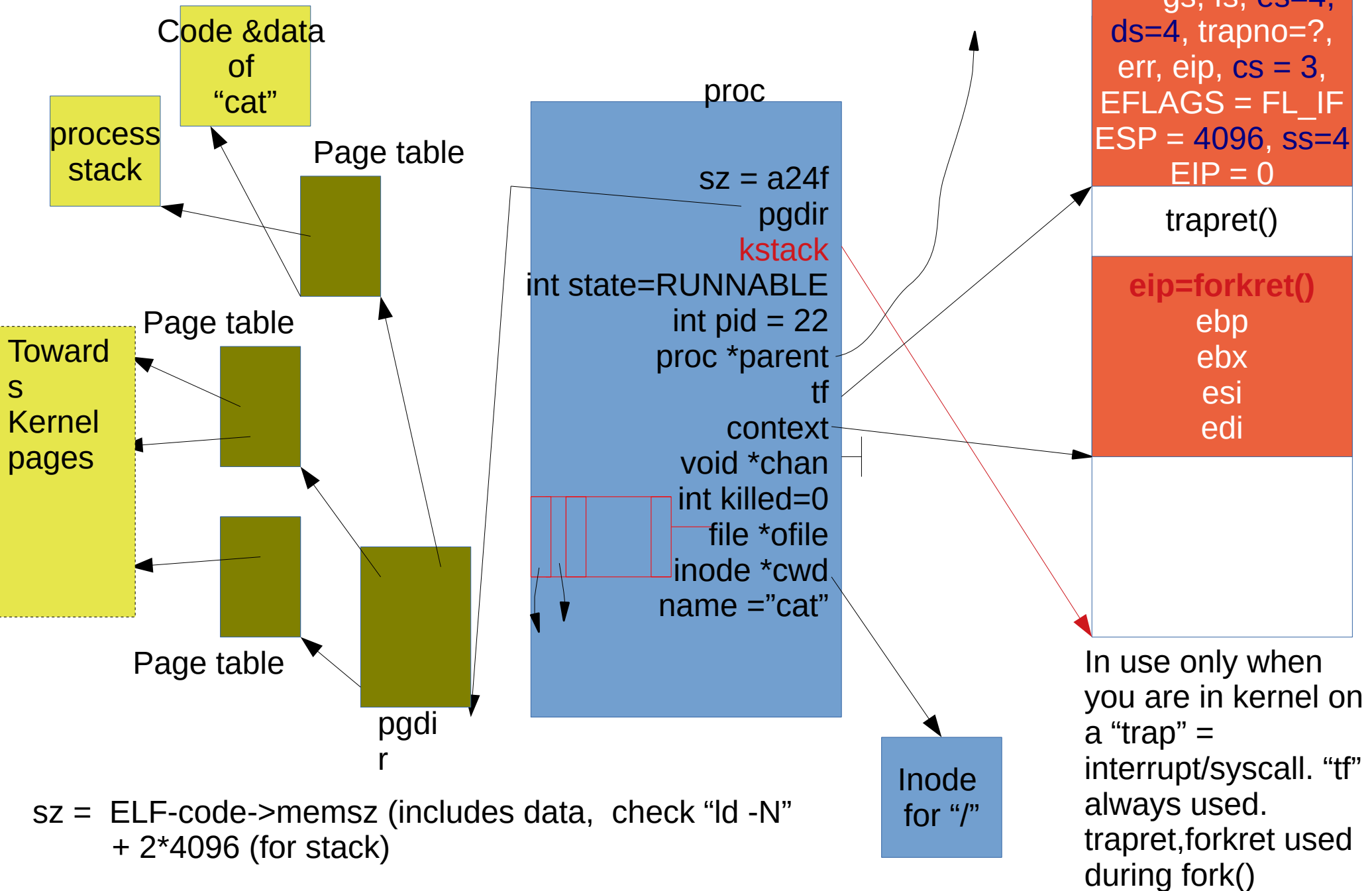
- **A process has two stacks**
 - user stack: used when user code is running
 - kernel stack: used when kernel is running on behalf of a process
- **Note: there is a third stack also!**
 - The kernel stack used by the scheduler itself
 - Not a per process stack

Struct proc

// Per-process state

```
struct proc {  
    uint sz;                // Size of process memory (bytes)  
    pde_t* pgdir;           // Page table  
    char *kstack;           // Bottom of kernel stack for this process  
    enum procstate state;    // Process state. allocated, ready to run, running, wait-  
ing for I/O, or exiting.  
    int pid;                // Process ID  
    struct proc *parent;     // Parent process  
    struct trapframe *tf;    // Trap frame for current syscall  
    struct context *context; // swtch() here to run process. Process's context  
    void *chan;              // If non-zero, sleeping on chan. More when we discuss sleep, wakeup  
    int killed;              // If non-zero, have been killed  
    struct file *ofile[NOFILE]; // Open files, used by open(), read(),...  
    struct inode *cwd;        // Current directory, changed with "chdir()"   
    char name[16];           // Process name (for debugging)  
};
```


struct proc diagram: Very imp!



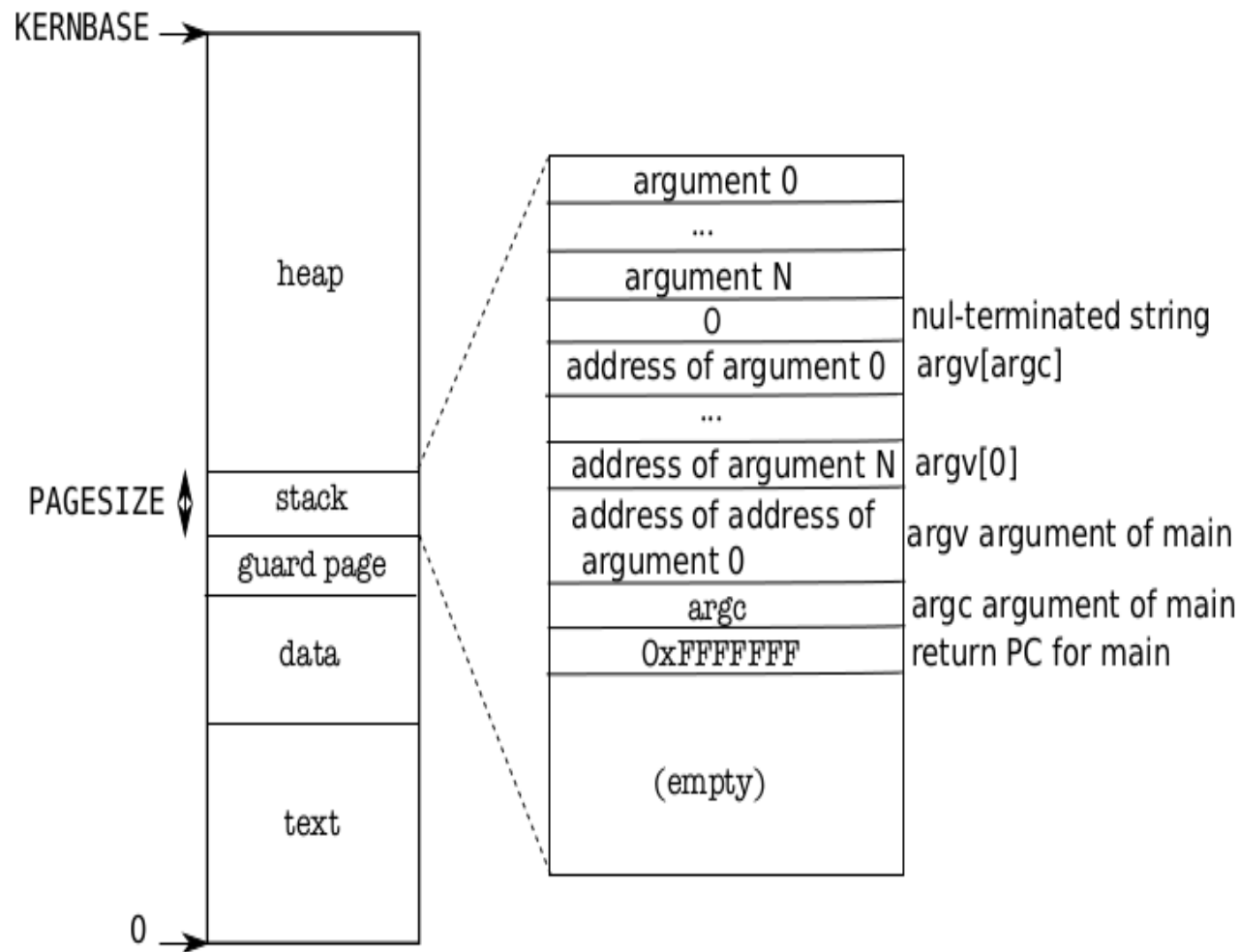
Memory Layout of a user process

Memory Layout of a user process

After exec()

Note the argc, argv on stack

The “guard page” is just a mapping in page table. No frame allocated. It's marked as invalid. So if stack grows (due to many function calls), then OS will detect it with an exception

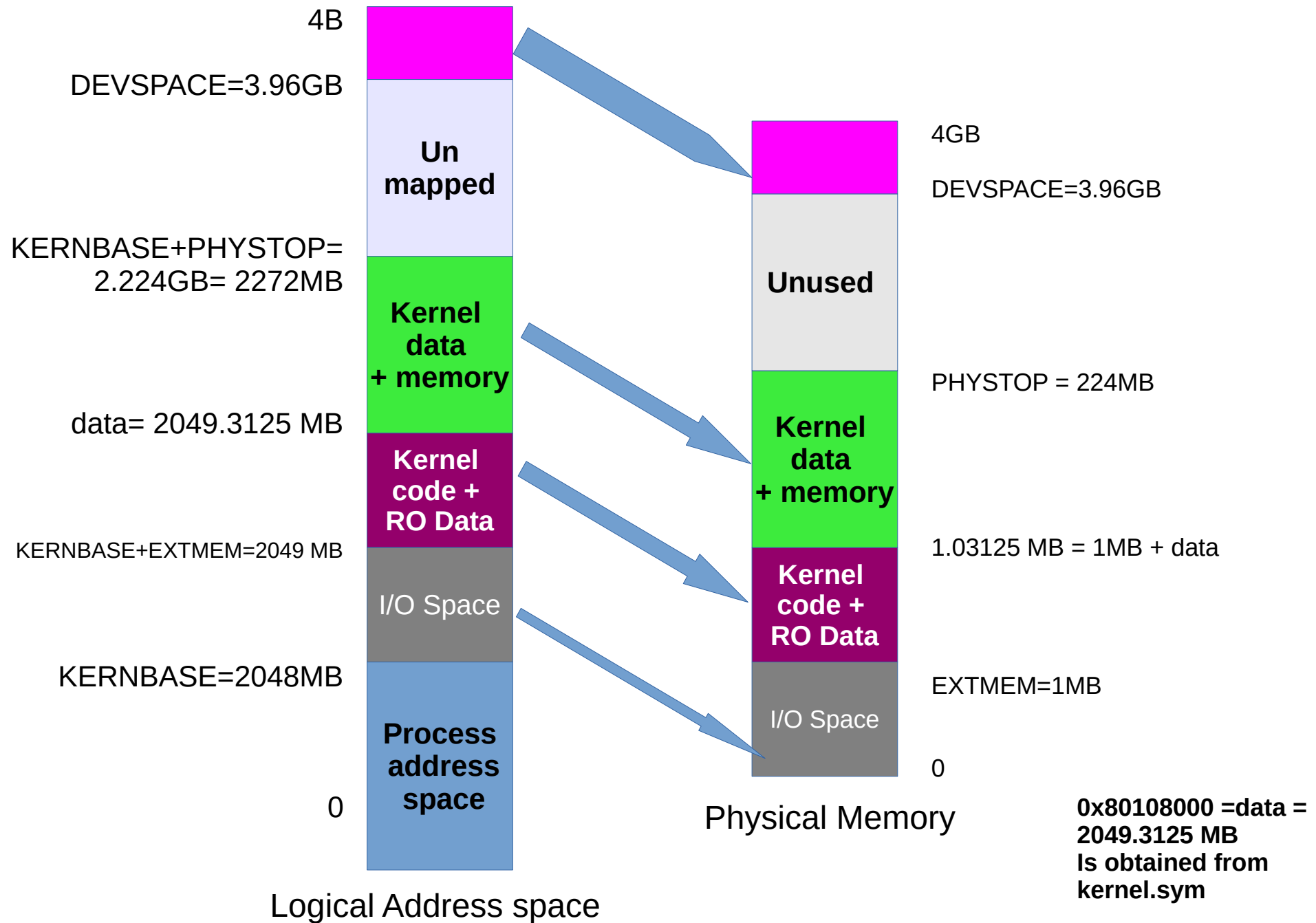


Handling Traps

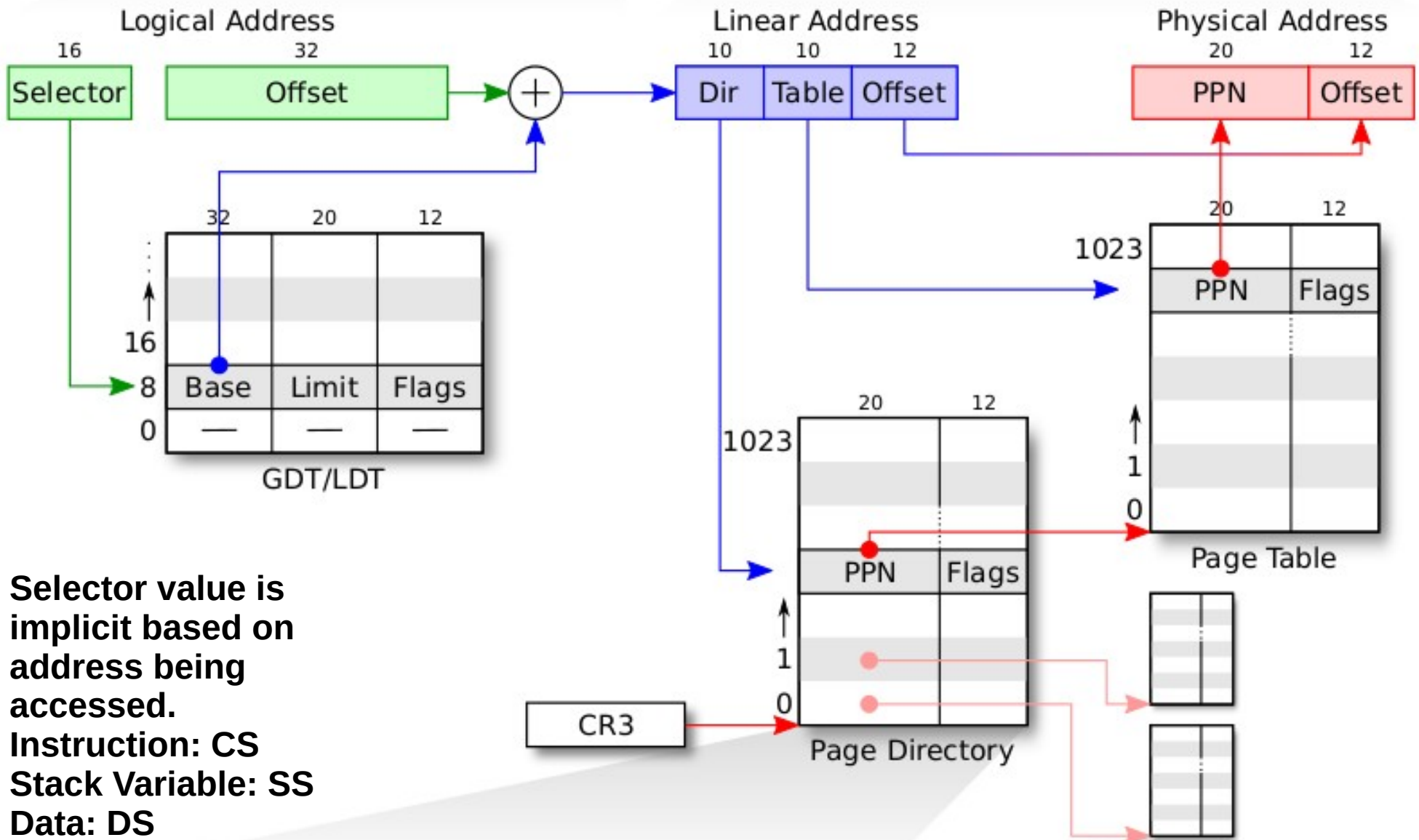
Some basic steps

- **Xv6.img is created by “make”**
 - Contains bootsector, kernel code, kernel data
- **QEMU boots using xv6.img**
 - First it runs bootloader
 - Bootloader loads kernel of xv6 (from xv6.img)
 - Kernel starts running
- **Kernel running..**
 - **Kernel calls main() of kernel (NOT a C application!) & Initializes:**
 - memory management data structures
 - process data structures
 - file handling data structures
 - Multi-processors
 - Multi-processor data structures
 - Interrupt Descriptor Table
 - ...
 - **Then creates init() ...**
 - **Init() fork-execs shell**

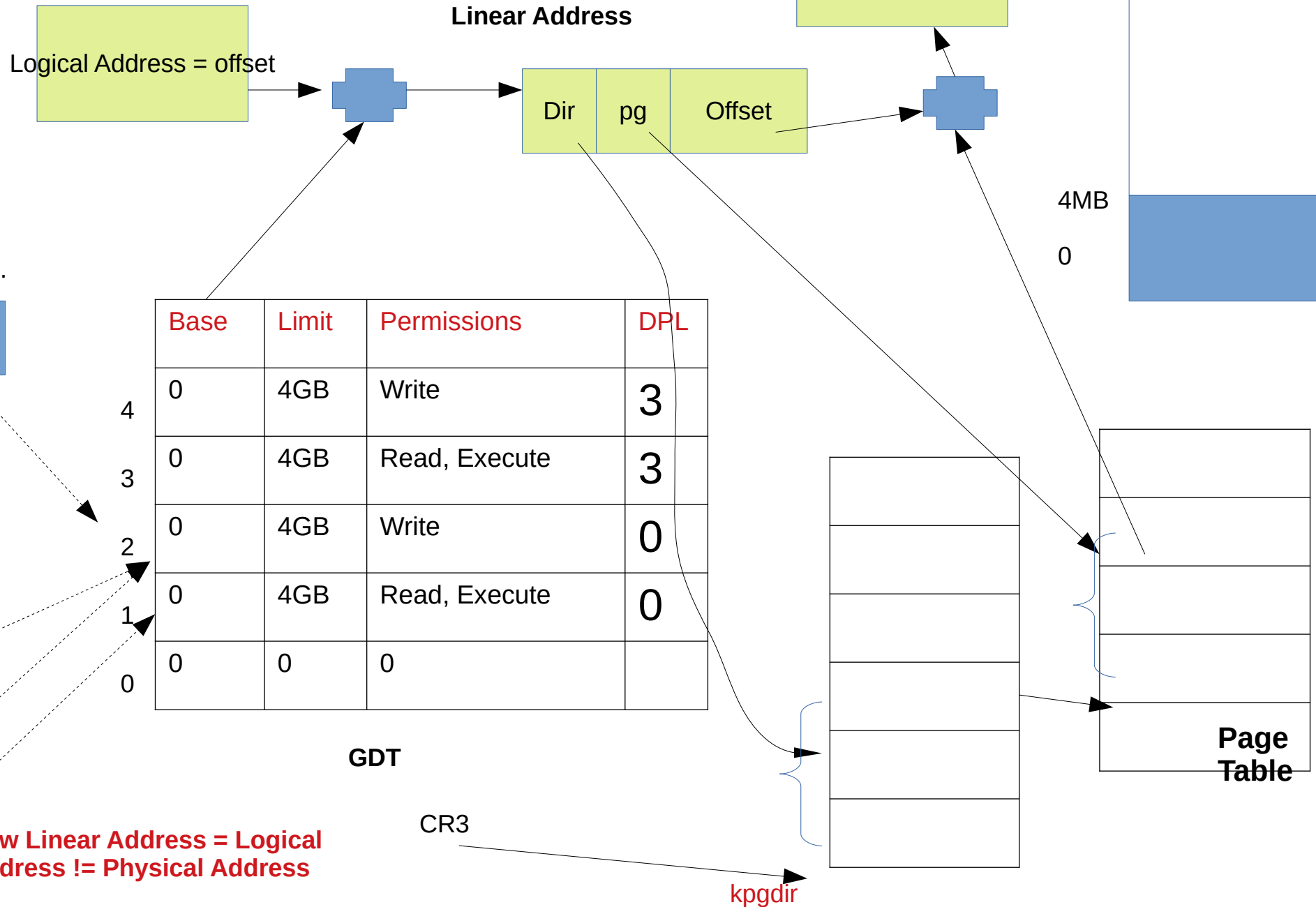
kmap[] mappings done in kvmalloc(). This shows segmentwise, entries are done in page directory and page table for corresponding VA-> PA mappings



Segmentation + Paging



After seginit() in main().
On the processor where we started booting



Handling traps

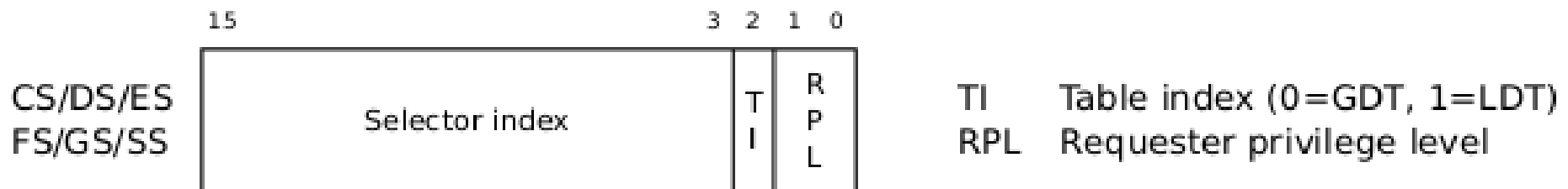
- **Transition from user mode to kernel mode**
 - On a system call
 - On a hardware interrupt
 - User program doing illegal work (exception)
- **Actions needed, particularly w.r.t. to hardware interrupts**
 - Change to kernel mode & switch to kernel stack
 - Kernel to work with devices, if needed
 - Kernel to understand interface of device

Handling traps

- **Actions needed on a trap**
 - Save the processor's registers (context) for future use
 - Set up the system to run kernel code (kernel context) on kernel stack
 - Start kernel in appropriate place (sys call, intr handler, etc)
 - Kernel to get all info related to event (which block I/O done?, which sys call called, which process did exception and what type, get arguments to system call, etc)

Privilege level

- The x86 has 4 protection levels, numbered 0 (most privilege) to 3 (least privilege).
- In practice, most operating systems use only 2 levels: 0 and 3, which are then called kernel mode and user mode, respectively.
- The current privilege level with which the x86 executes instructions is stored in %cs register, in the field CPL.



Privilege level

- **Changes automatically on**
 - “int” instruction**
 - hardware interrupt**
 - exeception**
- **Changes back on**
 - iret**
- **“int” 10 --> makes 10th hardware interrupt. S/w interrupt can be used to create hardware interrupt'**
- **Xv6 uses “int 64” for actual system calls**

Interrupt Descriptor Table (IDT)

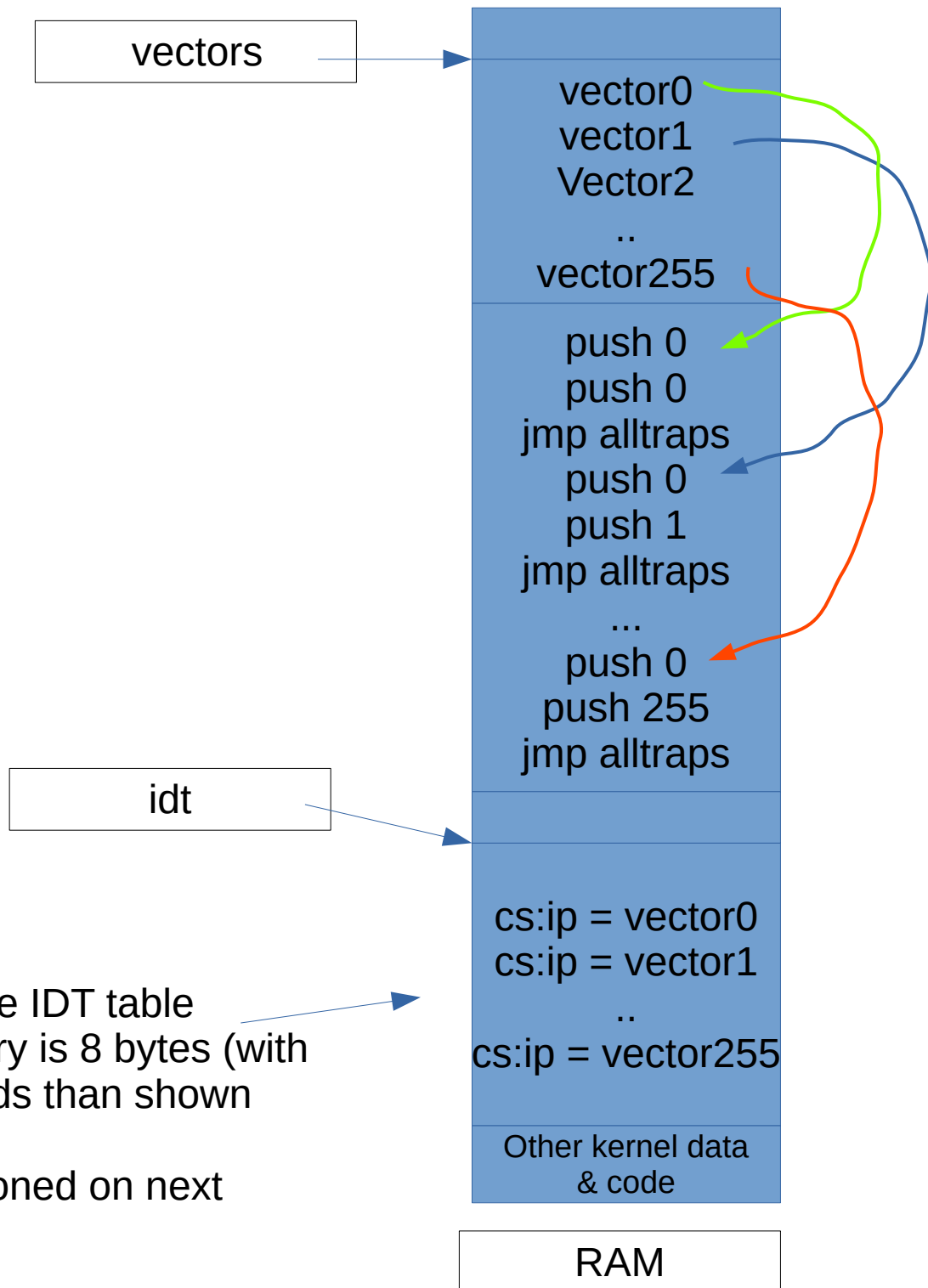
- **IDT defines interrupt handlers**
- **Has 256 entries**
 - each giving the %cs and %eip to be used when handling the corresponding interrupt.
- **Mapping**
 - Interrupts 0-31 are defined for software exceptions, like divide errors or attempts to access invalid memory addresses.
 - Xv6 maps the 32 hardware interrupts to the range 32-63
 - and uses interrupt 64 as the system call interrupt

IDT setup done by tvinit() function

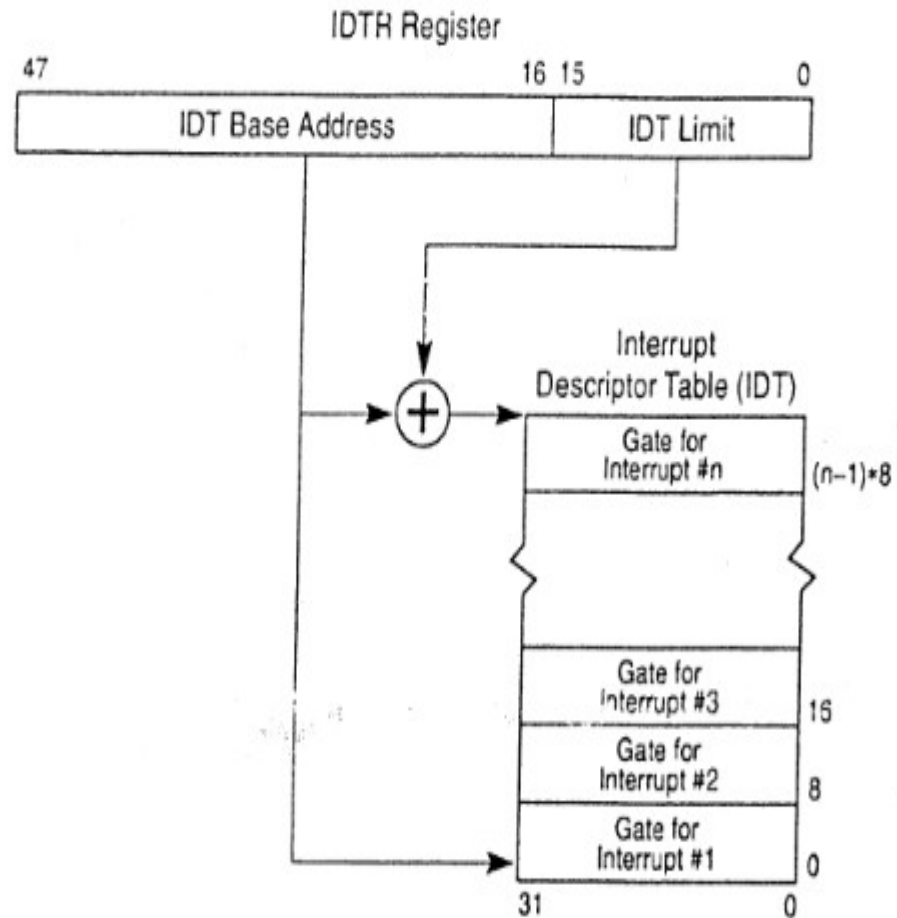
The array of “vectors”
And the code of
“push, ..jmp”
Is part of kernel image
(xv6.img)

The tvinit() is called during
kernel initialization

This is the IDT table
Each entry is 8 bytes (with
more fields than shown
here)
as mentioned on next
slides



IDTR and IDT



IDT is in RAM

IDTR is in CPU

Interrupt Descriptor Table (IDT) entries (in RAM)

```
// Gate descriptors for interrupts and traps
struct gatedesc {
    uint off_15_0 : 16; // low 16 bits of offset in segment
    uint cs : 16; // code segment selector
    uint args : 5; // # args, 0 for interrupt/trap gates
    uint rsv1 : 3; // reserved(should be zero I guess)
    uint type : 4; // type(STS_{IG32,TG32})
    uint s : 1; // must be 0 (system)
    uint dpl : 2; // descriptor(new) privilege level
    uint p : 1; // Present
    uint off_31_16 : 16; // high bits of offset in segment
};
```

Setting IDT entries

```
void
tvinit(void)
{
    int i;
    for(i = 0; i < 256; i++)
        SETGATE(idt[i], 0, SEG_KCODE<<3, vectors[i], 0);
    SETGATE(idt[T_SYSCALL], 1, SEG_KCODE<<3,
        vectors[T_SYSCALL], DPL_USER);
    /* value 1 in second argument --> don't disable
interrupts
        * DPL_USER means that processes can raise this
interrupt. */
    initlock(&tickslock, "time");
}
```


Setting IDT entries

```
#define SETGATE(gate, istrap, sel, off, d) \
{ \
    (gate).off_15_0 = (uint)(off) & 0xffff; \
    (gate).cs = (sel); \
    (gate).args = 0; \
    (gate).rsv1 = 0; \
    (gate).type = (istrap) ? STS_TG32 : STS_IG32; \
    (gate).s = 0; \
    (gate).dpl = (d); \
    (gate).p = 1; \
    (gate).off_31_16 = (uint)(off) >> 16; \
}
```

Setting IDT entries

Vectors.S

```
# generated by vectors.pl - do not
edit

# handlers

.globl alltraps
.globl vector0
vector0:
    pushl $0
    pushl $0
    jmp alltraps
.globl vector1
vector1:
    pushl $0
    pushl $1
    jmp alltraps
```

trapasm.S

```
#include "mmu.h"

# vectors.S sends all traps
here.

.globl alltraps
alltraps:
    # Build trap frame.
    pushl %ds
    pushl %es
    pushl %fs
    pushl %gs
    Pushal
    ....
```

How will interrupts be handled?

On int instruction/interrupt the CPU does this:

- Fetch the n'th descriptor from the IDT, where n is the argument of int. (IDTR->idt[n])
- Check that CPL in %cs is <= DPL, where DPL is the privilege level in the descriptor.
 - Temporarily save %esp and %ss in CPU-internal registers, but only if the target segment selector's PL < CPL.
 - Switching from user mode to kernel mode. Hence save user code's SS and ESP
- Load %ss and %esp from a task segment descriptor.
 - Stack changes to kernel stack now. TS descriptor is on GDT, index given by TR register. See switchvm()
- Push %ss. // optional
- Push %esp. // optional (also changes ss,esp using TSS)
- Push %eflags.
- Push %cs.
- Push %eip.
- Clear the IF bit in %eflags, but only on an interrupt.
- Set %cs and %eip to the values in the descriptor.

After “int” ‘s job is done

- **IDT was already set**
 - Remember vectors.S
- **So jump to 64th entry in vector's**
vector64:
 - pushl \$0
 - pushl \$64
 - jmp alltraps
- So now stack has ss, esp,eflags, cs, eip, 0 (for error code), 64
- Next run alltraps from trapasm.S

alltraps:

```
# Build trap frame.  
pushl %ds  
pushl %es  
pushl %fs  
pushl %gs  
pushal // push all gen purpose  
regs  
# Set up data segments.  
movw $(SEG_KDATA<<3), %ax  
movw %ax, %ds  
movw %ax, %es  
# Call trap(tf), where tf=%esp  
pushl %esp # first arg to trap()  
call trap  
addl $4, %esp
```

- Now stack contains
- ss, esp, eflags, cs, eip, 0 (for error code), 64, ds, es, fs, gs, eax, ecx, edx, ebx, oesp, ebp, esi, edi
 - This is the struct trapframe !
 - So the kernel stack now contains the trapframe
 - Trapframe is a part of kernel stack

```

void
trap(struct trapframe *tf)
{
    if(tf->trapno ==
    T_SYSCALL){
        if(myproc()->killed)
            exit();
        myproc()->tf = tf;
        syscall();
        if(myproc()->killed)
            exit();
        return;
    }
    switch(tf->trapno){
        .....

```

trap()

- **Argument is trapframe**
- **In alltraps**
 - Before “call trap”, there was “push %esp” and stack had the trapframe
 - Remember calling convention --> when a function is called, the stack contains the arguments in reverse order (here only 1 arg)

trap()

- **Has a switch**
 - `switch(tf->trapno)`
 - Q: who set this trapno?
- **Depending on the type of trap**
 - **Call interrupt handler**
 - **Timer**
 - `wakeup(&ticks)`
 - **IDE: disk interrupt**
 - `Ideintr()`
 - **KBD**
 - `Kbdintr()`
 - **COM1**
 - `Uatrintr()`
 - **If Timer**
 - Call `yield()` -- calls `sched()`
 - **If process was killed (how is that done?)**
 - Call `exit()`!

when trap() returns

- **#Back in alltraps**

call trap

addl \$4, %esp

Return falls through to trapret...

.globl trapret

trapret:

popal

popl %gs

popl %fs

popl %es

popl %ds

addl \$0x8, %esp # trapno and errcode

iret

▪

- **Stack had (trapframe)**
 - **ss, esp,eflags, cs, eip, 0**
(for error code), 64, ds,
es, fs, gs, eax, ecx, edx,
ebx, oesp, ebp, esi, edi,
esp
- **add \$4 %esp**
 - **esp**
- **popal**
 - **eax, ecx, edx, ebx, oesp,**
ebp, esi, edi
- **Then gs, fs, es, ds**
- **add \$0x8, %esp**
 - **0 (for error code), 64**
- **iret**
 - **ss, esp,eflags, cs, eip,**