

Memory Management Basics

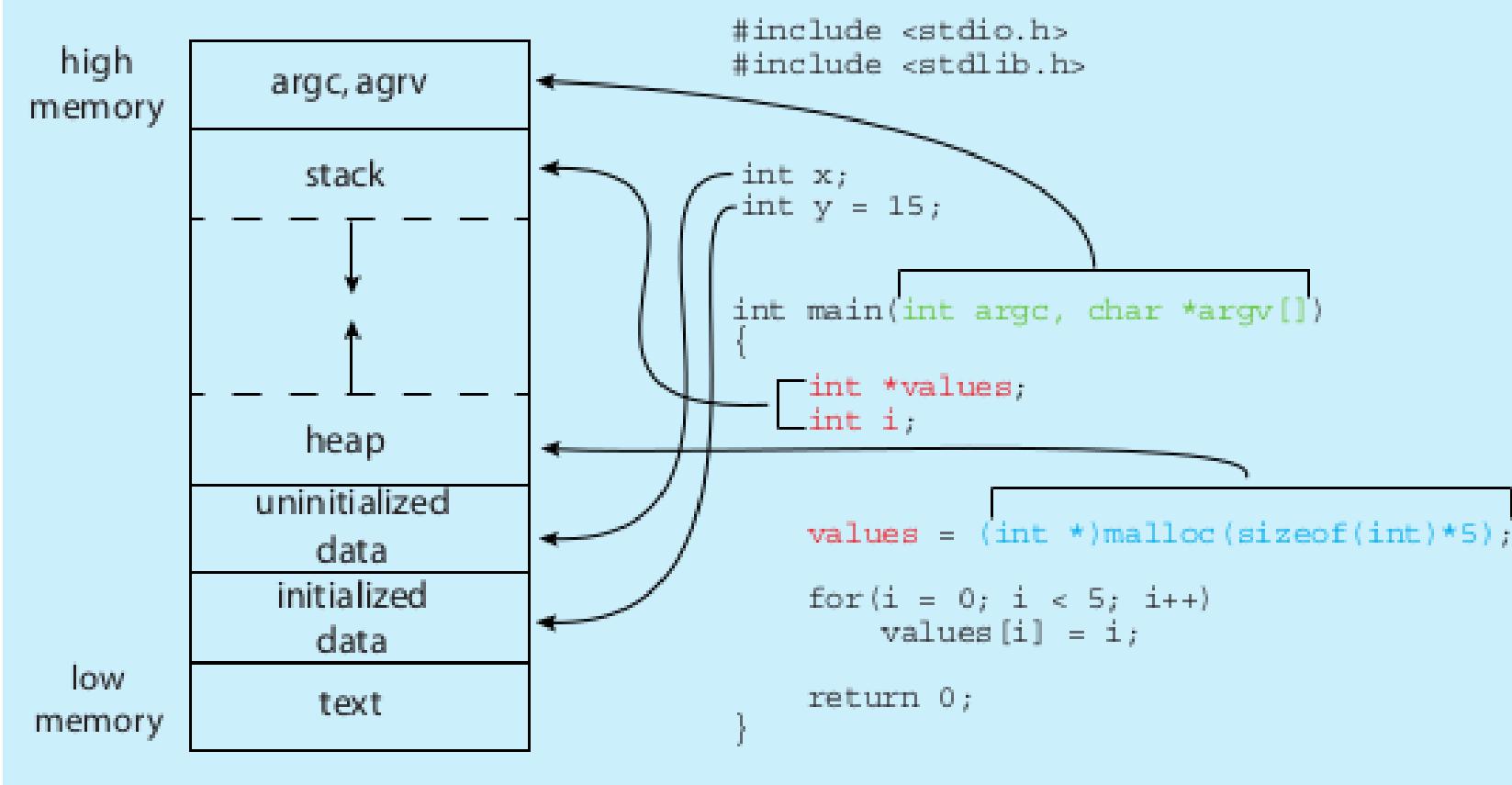
Summary

- Understanding how the processor architecture drives the memory management features of OS and system programs (compilers, linkers)
- Understanding how different hardware designs lead to different memory management schemes by operating systems

Addresses issued by CPU

- During the entire ‘on’ time of the CPU
 - Addresses are “issued” by the CPU on address bus
 - One address to fetch instruction from location specified by PC
 - Zero or more addresses depending on instruction
 - e.g. mov \$0x300, r1 # move contents of address 0x300 to r1 --> one extra address issued on address bus

Memory layout of a C program



\$ size /bin/ls

| text | data | bss | dec | hex | filename |
|--------|------|------|--------|-------|----------|
| 128069 | 4688 | 4824 | 137581 | 2196d | /bin/ls |

Desired from a multi-tasking system

- Multiple processes in RAM at the same time (multi-programming)
- Processes should not be able to see/touch each other's code, data (globals), stack, heap, etc.
- Further advanced requirements
 - Process could reside anywhere in RAM
 - Process need not be continuous in RAM
 - Parts of process could be moved anywhere in RAM

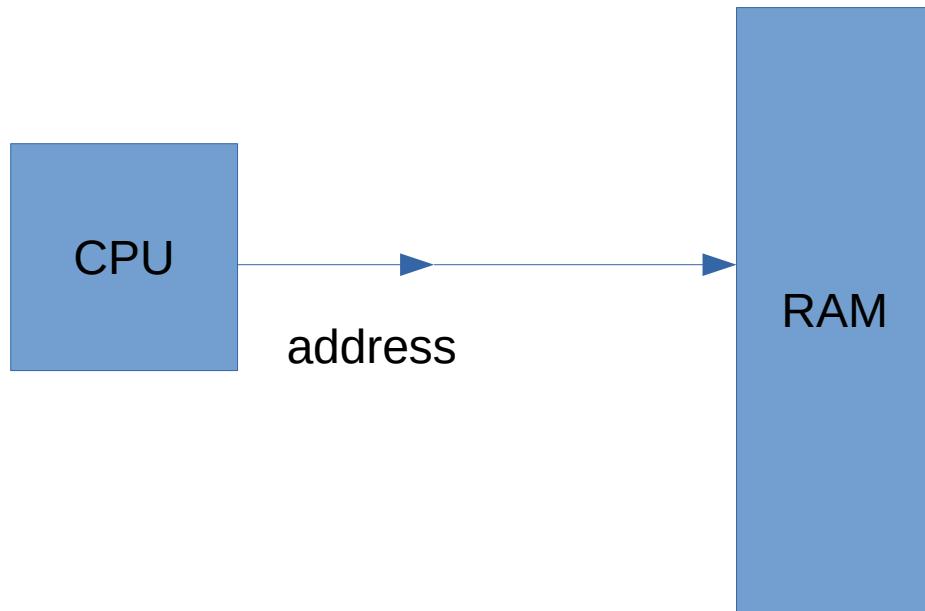
Different ‘times’

- Different actions related to memory management for a program are taken at different times. So let's know the different ‘times’
- Compile time
 - When compiler is compiling your C code
- Load time
 - When you execute “./myprogram” and it's getting loaded in RAM by loader i.e. `exec()`
- Run time
 - When the process is alive, and getting scheduled by the OS

Different types of Address binding

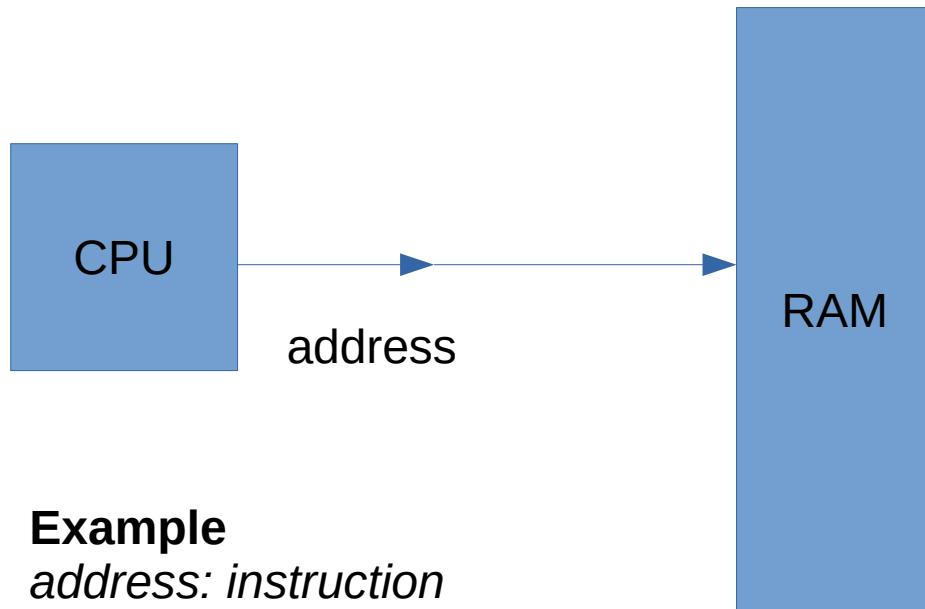
- Compile time address binding
 - Address of code/variables is fixed by compiler
 - Very rigid scheme
 - Location of process in RAM can not be changed ! Non-relocatable code.
 - Load time address binding
 - Address of code/variables is fixed by loader
 - Location of process in RAM is decided at load time, but can't be changed later
 - Flexible scheme, relocatable code
 - Run time address binding
 - Address of code/variables is fixed at the time of executing the code
 - Very flexible scheme , highly relocatable code
 - Location of process in RAM is decided at load time, but CAN be changed later also
- Which binding is actually used, is mandated by processor features + OS

Simplest case



- Suppose the address issued by CPU reaches the RAM controller directly

Simplest case



Example

address: instruction

1000: mov \$0x300, r1

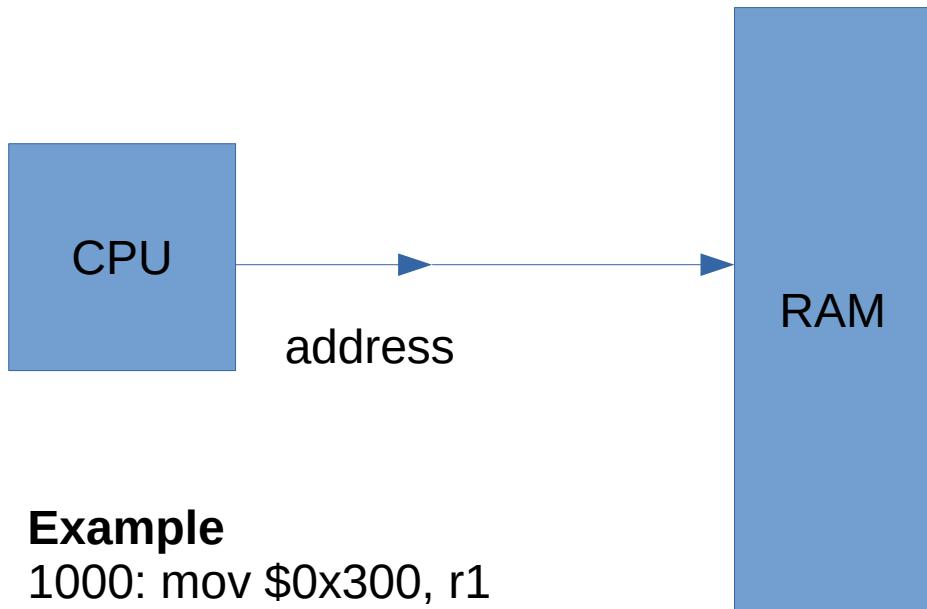
1004: add r1, -3

1008: jnz 1000

Sequence of addressed sent by CPU: 1000, 0x300, 1004, 1008, 1000, 0x300, ...

- How does this impact the compiler and OS ?
- When a process is running the addresses issued by it, will reach the RAM directly
- So exact addresses of globals, addresses in “jmp” and “call” must be part the machine instructions generated by compiler
 - How will the compiler know the addresses, at “compile time” ?

Simplest case

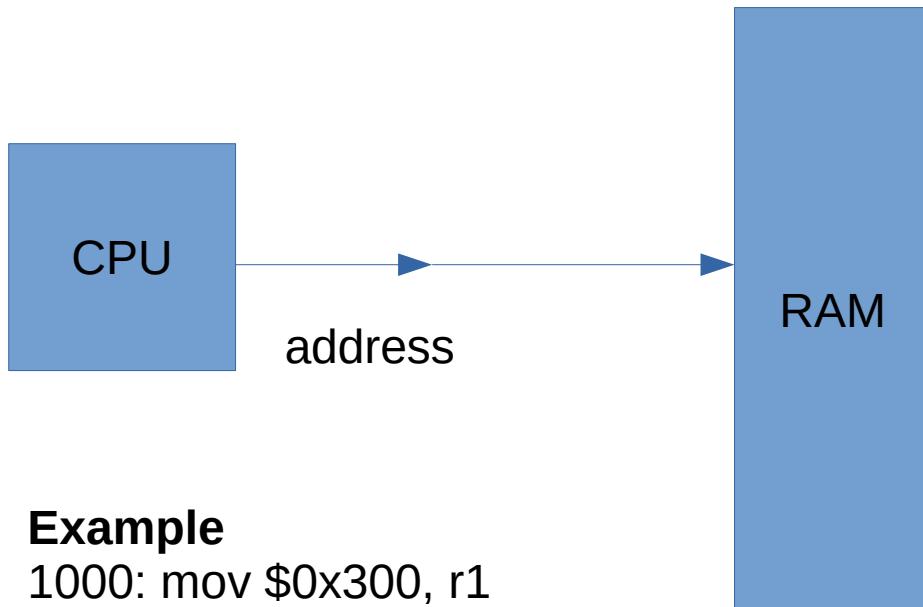


Example

```
1000: mov $0x300, r1  
1004: add r1, -3  
1008: jnz 1000
```

- Solution: compiler assumes some fixed addresses for globals, code, etc.
- OS loads the program exactly at the same addresses specified in the executable file.
Non-relocatable code.
- Now program can execute properly.

Simplest case



Example

```
1000: mov $0x300, r1  
1004: add r1, -3  
1008: jnz 1000
```

- Problem with this solution
 - Programs once loaded in RAM must stay there, can't be moved
 - What about 2 programs?
 - Compilers being “programs”, will make same assumptions and are likely to generate same/overlapping addresses for two different programs
 - Hence only one program can be in memory at a time !
 - No need to check for any memory boundary violations – all memory belongs to one process
- Example: DOS

Base/Relocation + Limit scheme

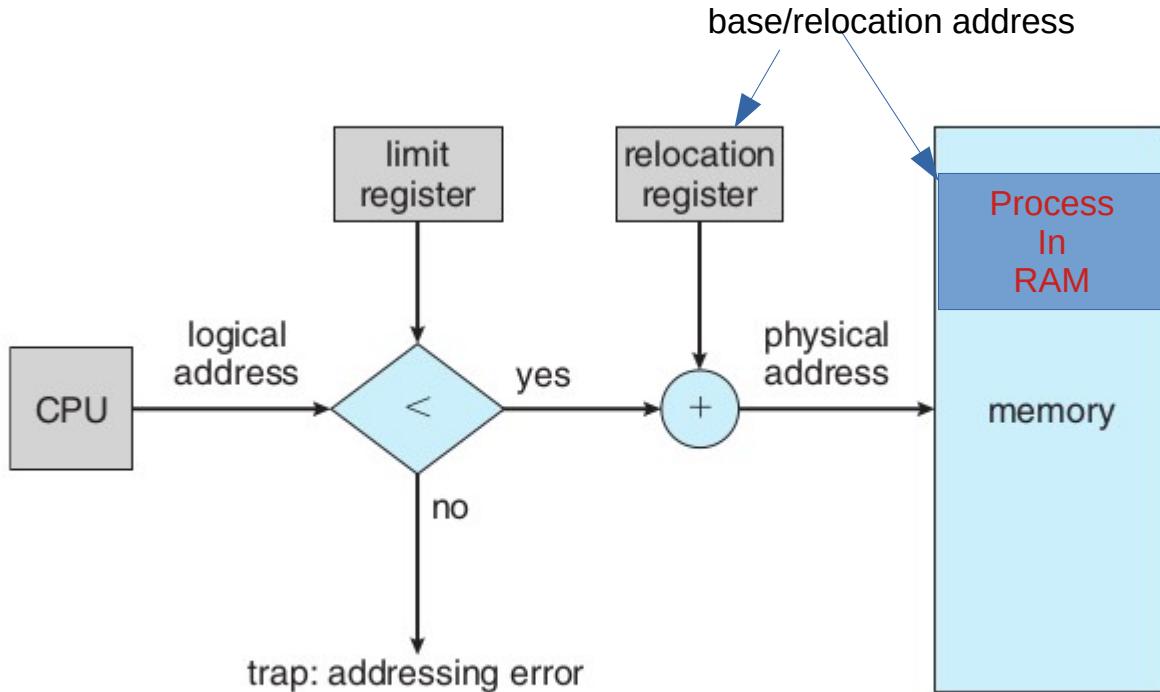


Figure 9.6 Hardware support for relocation and limit registers.

- Base and Limit are two registers inside CPU's Memory Management Unit
- 'base' is added to the address generated by CPU
- The result is compared with base+limit and if less passed to memory, else hardware interrupt is raised

Memory Management Unit (MMU)

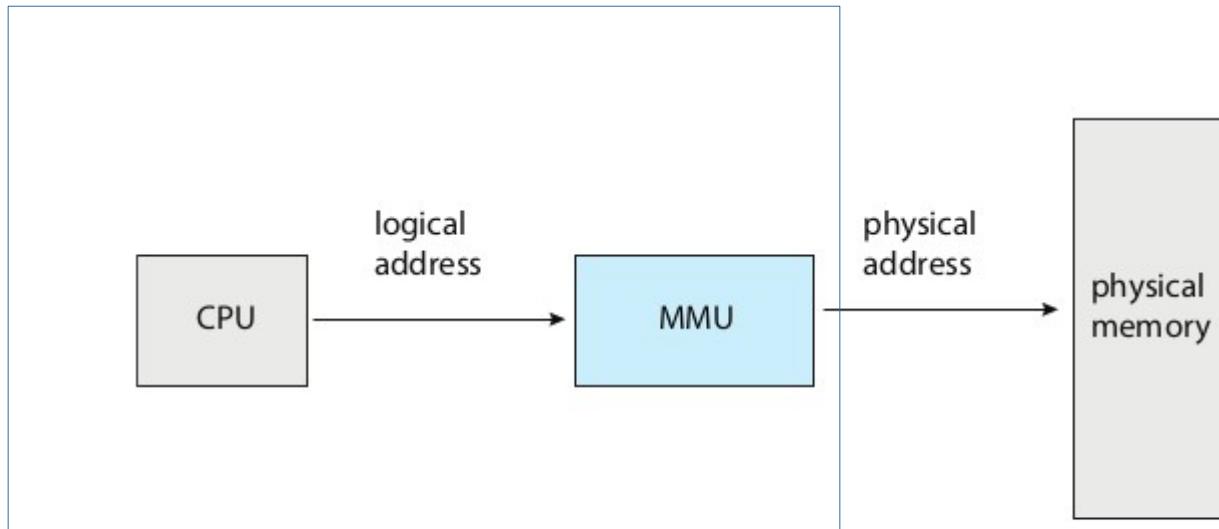
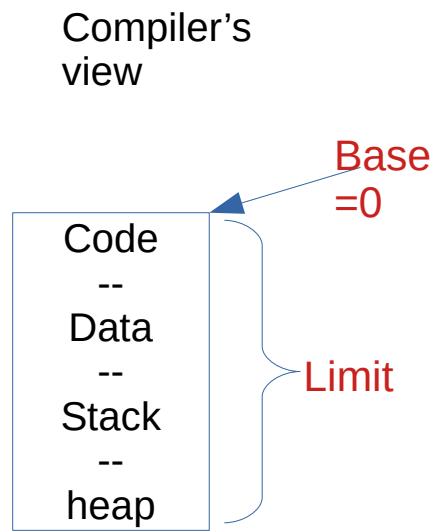
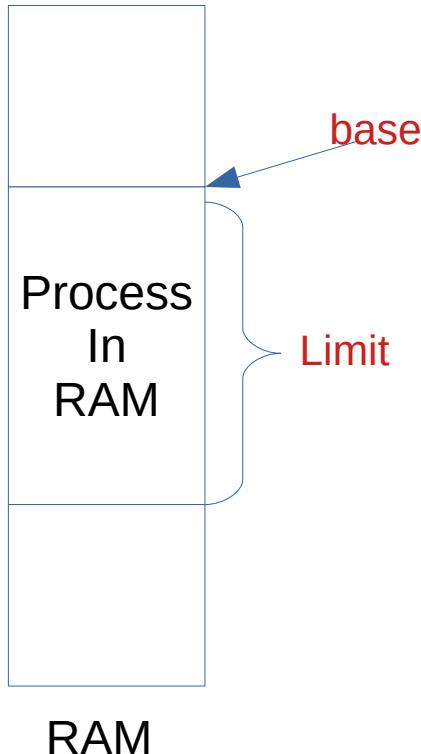


Figure 9.4 Memory management unit (MMU).

- Is part of the CPU chip, acts on every memory address issue by execution unit of the CPU
- In the scheme just discussed, the base, limit calculation parts are part of MMU

Base/Relocation + Limit scheme



- Compiler's work
 - Assume that the process is one continuous chunk in memory, with a size limit
 - Assume that the process starts at address zero (!) and calculate addresses for globals, code, etc. And accordingly generate machine code

Base/Relocation + Limit scheme

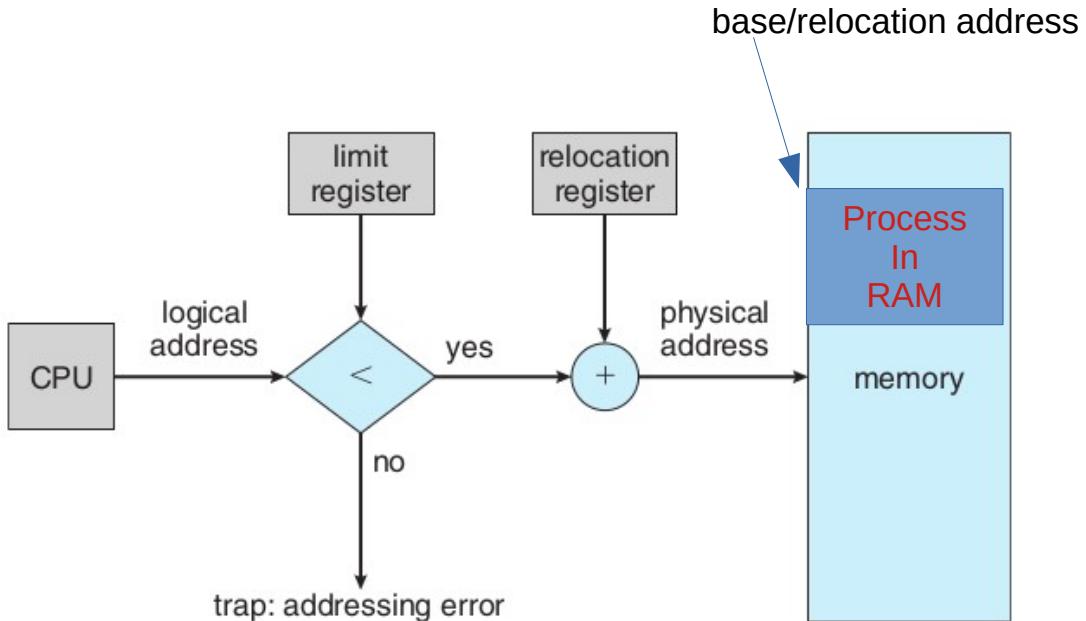


Figure 9.6 Hardware support for relocation and limit registers.

- OS's work

- While loading the process in memory – must load as one continuous segment
- Fill in the ‘base’ register with the actual address of the process in RAM.
- Setup the limit to be the size of the process as set by compiler in the executable file. *Remember the base+limit in OS's own data structures.*

Base/Relocation + Limit scheme

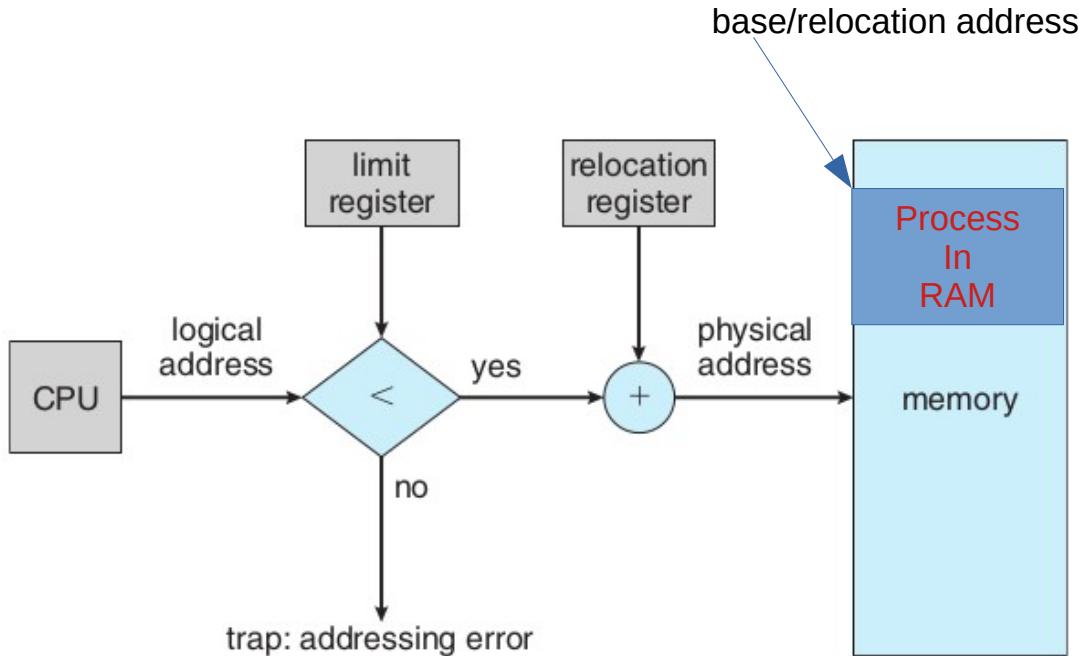
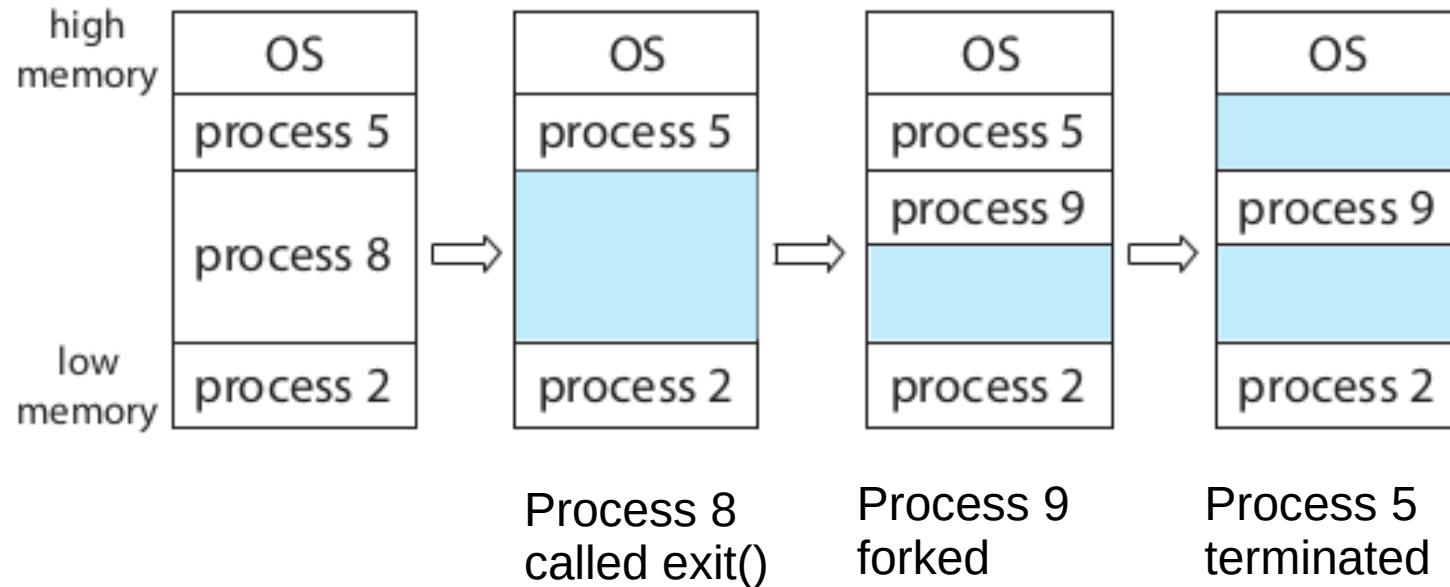


Figure 9.6 Hardware support for relocation and limit registers.

- Combined effect
 - “**Relocatable code**” – the process can go anywhere in RAM at the time of loading
 - Some memory violations can be detected – a memory access beyond base+limit will raise interrupt, thus running OS in turn, which may take action against the process

Example scenario of memory in base+limit scheme



It should be possible to have relocatable code
even with “simplest case”

By doing extra work during “loading”.

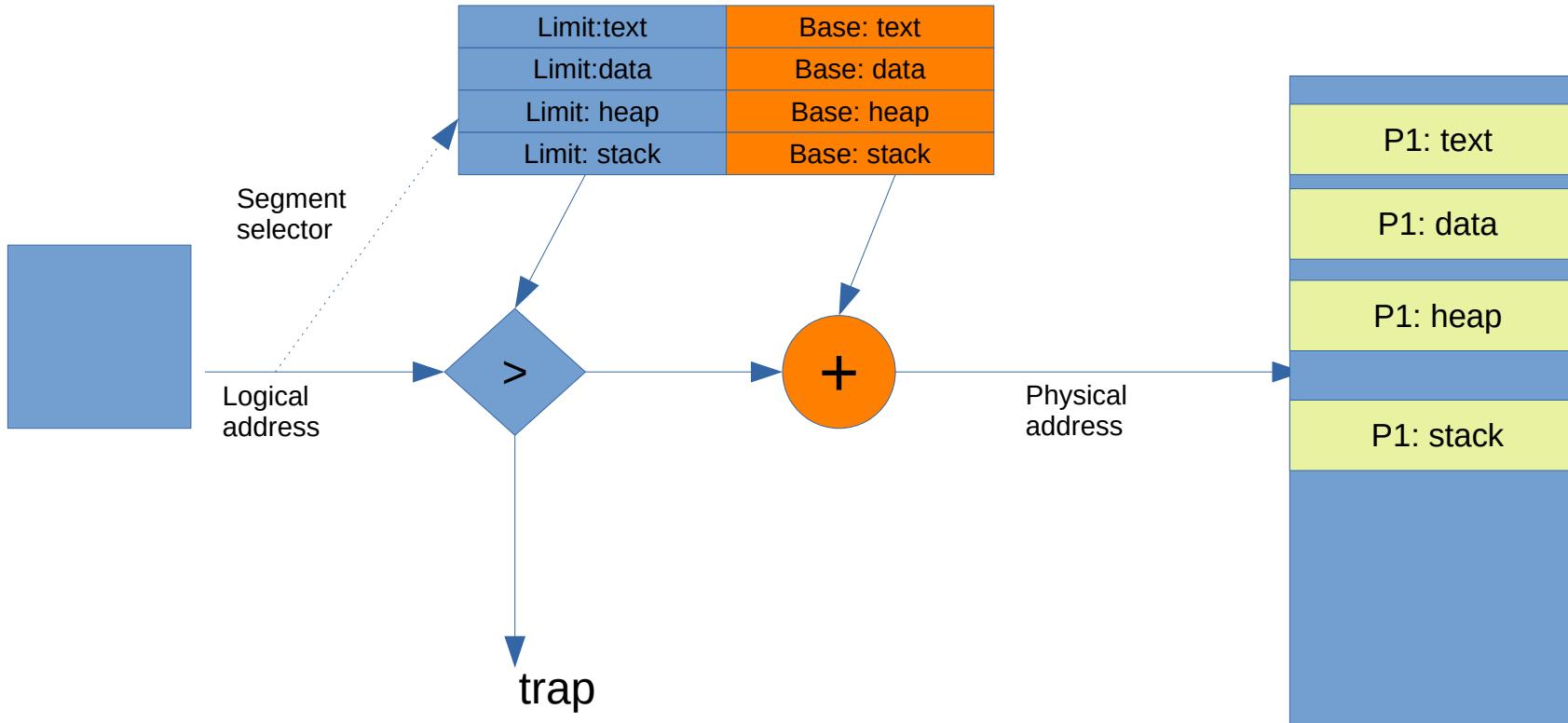
How?

Next scheme: Segmentation

Multiple base +limit pairs

- Multiple sets of base + limit registers
- Whenever an address is issued by execution unit of CPU, it will also include reference to some base register
 - And hence limit register paired to that base register will be used for error checking
- Compiler: can assume a separate chunk of memory for code, data, stack, heap, etc. And accordingly calculate addresses . Each “segment” starting at address 0.
- OS: will load the different ‘sections’ in different memory regions and accordingly set different ‘base’ registers
-

Next scheme: Multiple base +limit pairs



Next scheme: Multiple base +limit pairs, with further indirection

- Base + limit pairs can also be stored in some memory location (not in registers). Question: how will the cpu know where it's in memory?
 - One CPU register to point to the location of table in memory
- Segment registers still in use, they give an index in this table
- This is x86 segmentation
 - Flexibility to have lot more “base+limits” in the array/table in memory

Next scheme: Multiple base +limit pairs, with further indirection

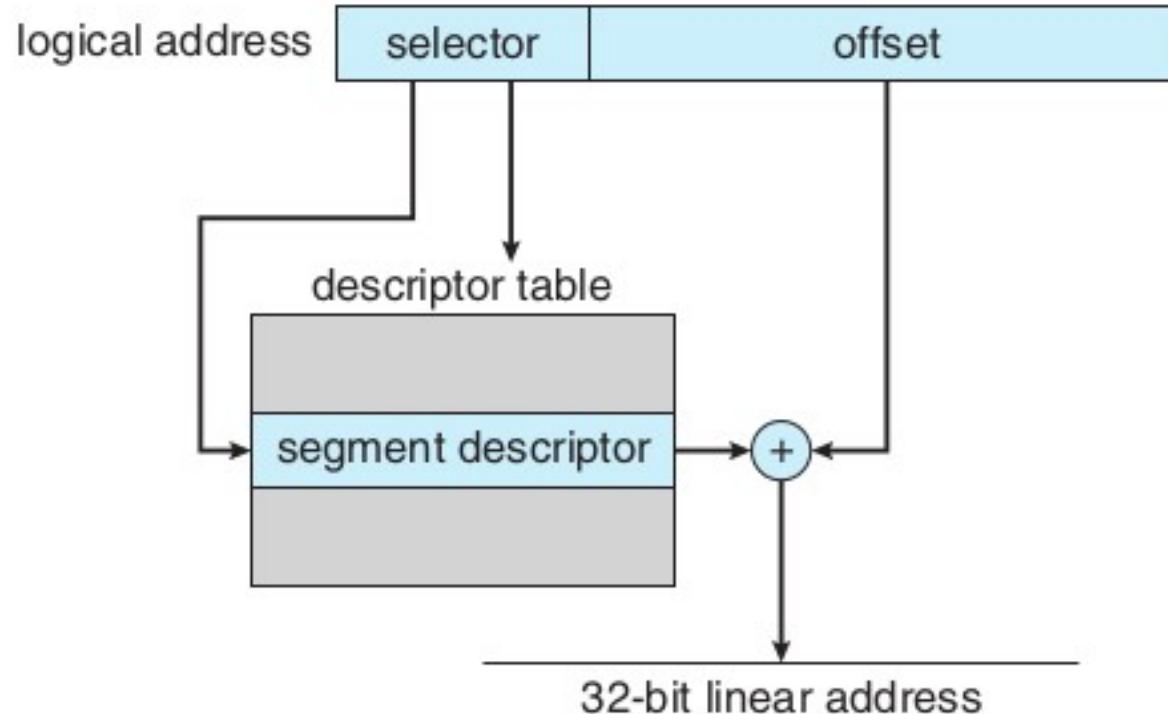


Figure 9.22 IA-32 segmentation.

Problems with segmentation schemes

- OS needs to find a continuous free chunk of memory that fits the size of the “segment”
 - If not available, your exec() can fail due to lack of memory
- Suppose 50k is needed
 - Possible that among 3 free chunks total 100K may be available, but no single chunk of 50k!
 - **External fragmentation**
- Solution to external fragmentation: **compaction** – move the chunks around and make a continuous big chunk available. Time consuming, tricky.

Solving external fragmentation problem

- Process should not be continuous in memory!
- Divide the continuous process image in smaller chunks (let's say 4k each) and locate the chunks anywhere in the physical memory
 - Need a way to map the *logical* memory addresses into *actual physical memory addresses*

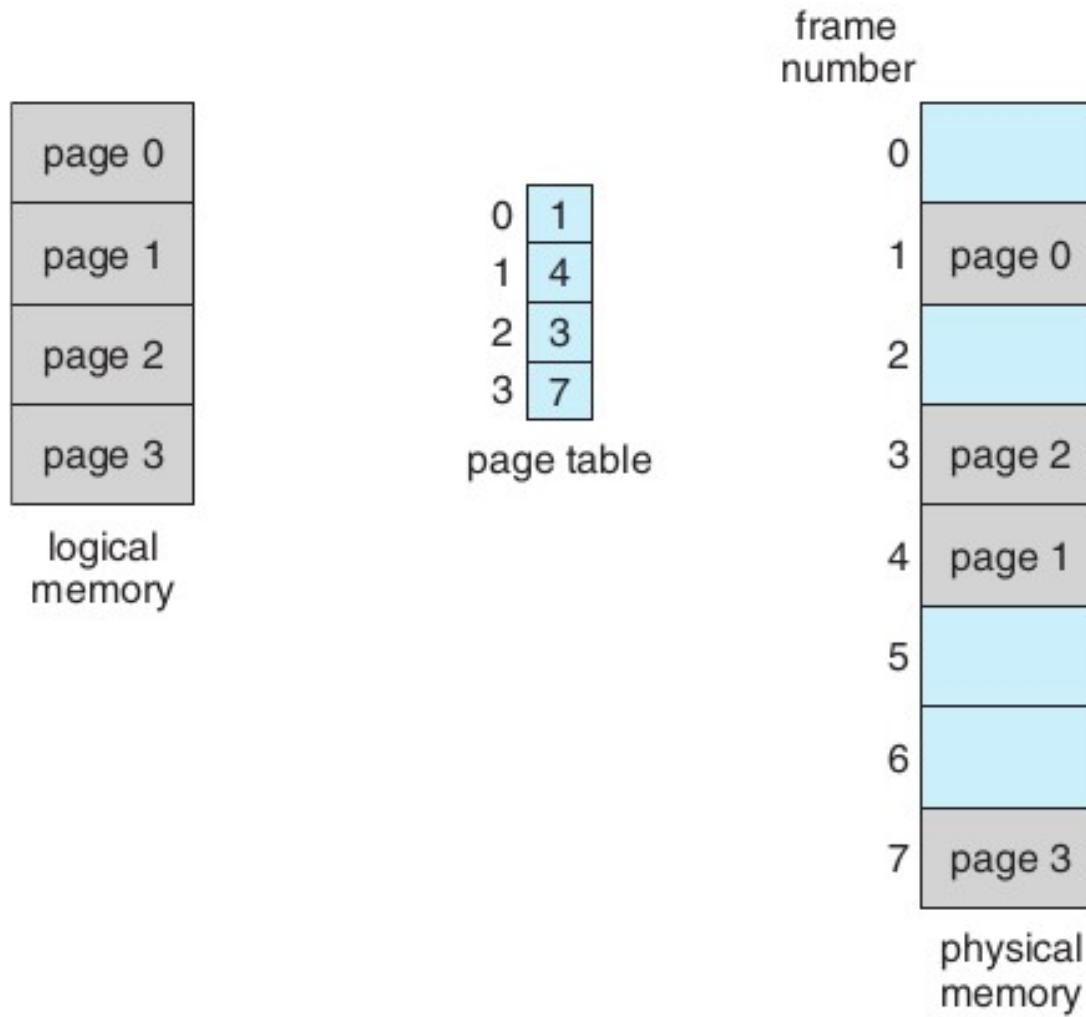


Figure 9.9 Paging model of logical and physical memory.

| | |
|----|---|
| 0 | a |
| 1 | b |
| 2 | c |
| 3 | d |
| 4 | e |
| 5 | f |
| 6 | g |
| 7 | h |
| 8 | i |
| 9 | j |
| 10 | k |
| 11 | l |
| 12 | m |
| 13 | n |
| 14 | o |
| 15 | p |

logical memory

| | |
|---|---|
| 0 | 5 |
| 1 | 6 |
| 2 | 1 |
| 3 | 2 |

page table

| | |
|----|------------------|
| 0 | |
| 4 | i j k l |
| 8 | m n o p |
| 12 | |
| 16 | |
| 20 | a b c d |
| 24 | e f g h |
| 28 | |

physical memory

Figure 9.10 Paging example for a 32-byte memory with 4-byte pages.

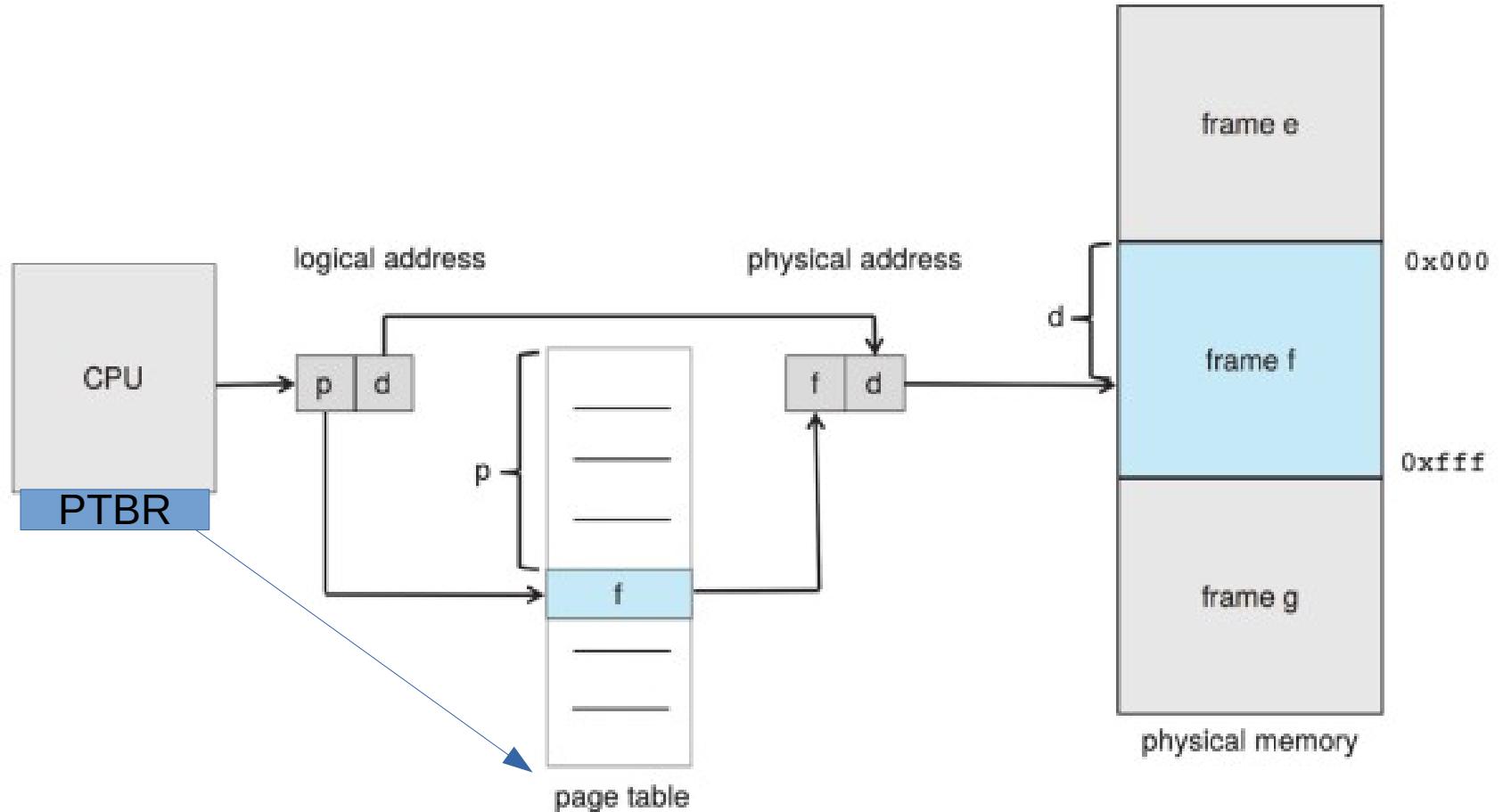


Figure 9.8 Paging hardware.

Paging

- Process is assumed to be composed of equally sized “pages” (e.g. 4k page)
- Actual memory is considered to be divided into page “frames”.
- CPU generated logical address is split into a page number and offset
- A page table base register inside CPU will give location of an in memory table called page table
- Page number used as offset in a table called page table, which gives the physical page frame number
- Frame number + offset are combined to get physical memory address

Paging

- Compiler: assume the process to be one continuous chunk of memory (!) . Generate addresses accordingly
- OS: at exec() time – allocate different frames to process, allocate a page table(!), setup the page table to map page numbers with frame numbers, setup the page table base register, start the process
- Now hardware will take care of all translations of logical addresses to physical addresses

X86 memory management



Figure 9.21 Logical to physical address translation in IA-32.

Segmentation in x86

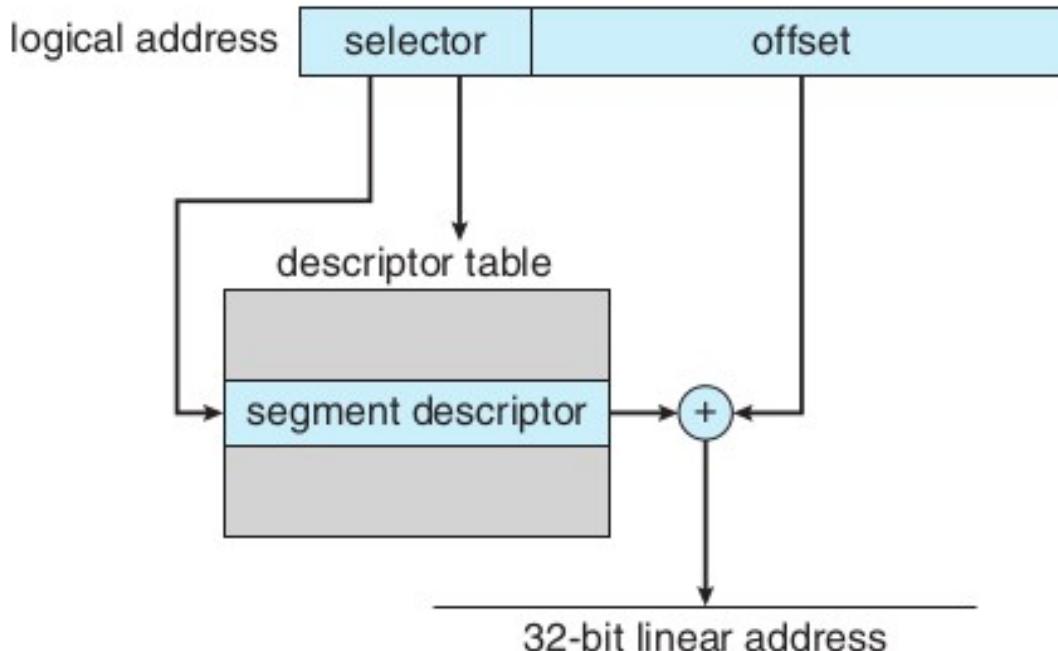


Figure 9.22 IA-32 segmentation.

- The selector is automatically chosen using Code Segment (CS) register, or Data Segment (DS) register depending on which type of memory address is being fetched
- Descriptor table is in memory
- The location of Descriptor table (Global DT- GDT or Local DT – LDT) is given by a GDT-register i.e. GDTR or LDT-register i.e. LDTR
-

Paging in x86

- Depending on a flag setup in CR3 register, either 4 MB or 4 KB pages can be enabled
- Page directory, page table are both in memory

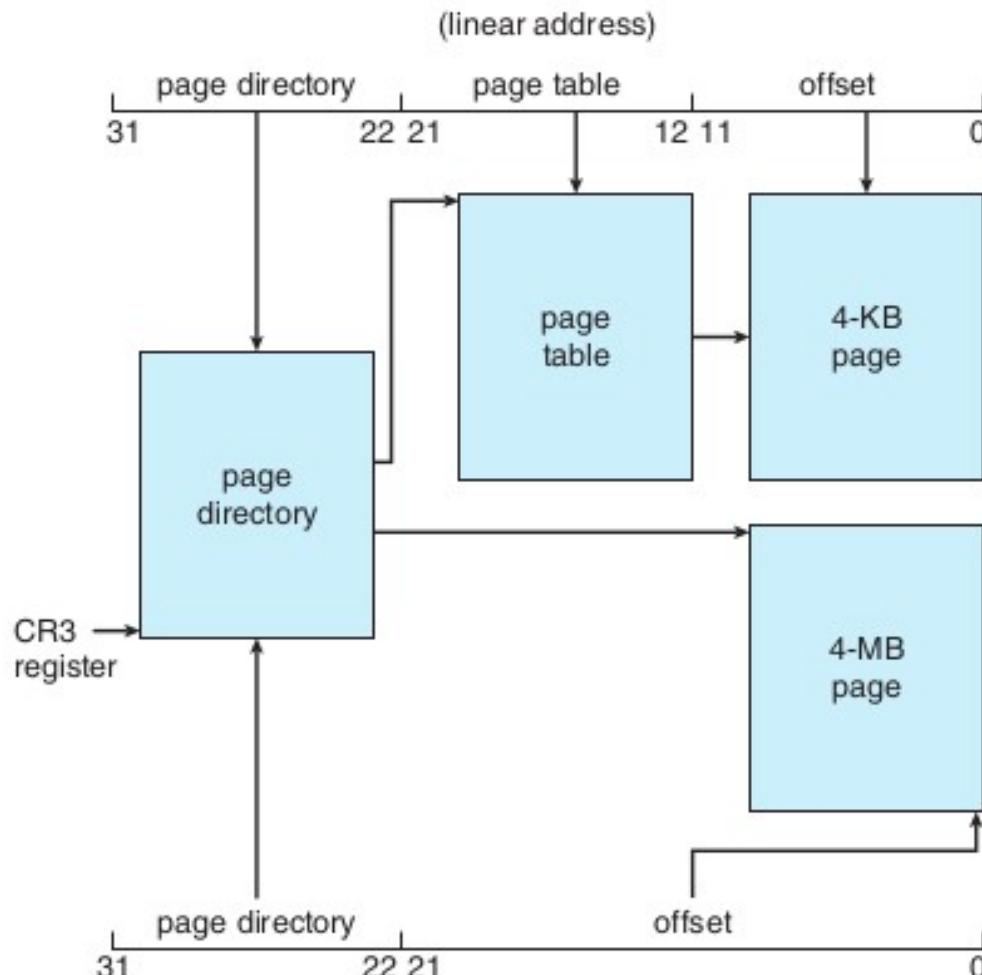


Figure 9.23 Paging in the IA-32 architecture.

XV6 bootloader

Abhijit A. M.
abhijit.comp@coep.ac.in

Credits:
xv6 book by Cox, Kaashoek, Morris
Notes by Prof. Sorav Bansal

A word of caution

- We begin reading xv6 code
- But it's not possible to read this code in a “linear fashion”
 - The dependency between knowing OS concepts and reading/writing a kernel that is written using all concepts

What we have seen

- **Compilation process, calling conventions**
- **Basics of Memory Management by OS**
- **Basics of x86 architecture**
 - Registers, segments, memory management unit, addressing, some basic machine instructions,
- **ELF files**
 - Objdump, program headers
 - Symbol tables

Boot-process

- **Bootloader itself**
 - Is loaded by the BIOS at a fixed location in memory and BIOS makes it run
 - Our job, as OS programmers, is to write the bootloader code
- **Bootloader does**
 - Pick up code of OS from a ‘known’ location and loads it in memory
 - Makes the OS run
- **Xv6 bootloader: bootasm.S bootmain.c (see Makefile)**

bootloader

- BIOS Runs (automatically)
- Loads boot sector into RAM at 0x7c00
- Starts executing that code
 - Make sure that your bootloader is loaded at 0x7c00
 - Makefile has

```
bootblock: bootblock.S bootmain.c  
$(CC) $(CFLAGS) -fno-pic -nostdinc -I. -c bootasm.S .....  
...  
$(LD) $(LDFLAGS) -N -e start -Ttext 0x7C00 -o bootblock.o  
bootasm.o bootmain.o
```

Results in:

00007c00 <start>: in bootblock.asm

Processor starts in real mode

- Processor starts in real mode – works like 16 bit 8088
- eight 16-bit general-purpose registers,
- Segment registers %cs, %ds, %es, and %ss --> additional bits necessary to generate 20-bit memory addresses from 16-bit registers.

$\text{addr} = \text{seg} \ll 4 + \text{addr}$



**Effective memory translation in the beginning
At _start in bootasm.S:**

%cs=0 %ip=7c00.

So effective address = 0*16+ip = ip

bootloader

- First instruction is ‘cli’
 - disable interrupts
- So that until your code loads all hardware interrupt handlers, no interrupt will occur

Zeroing registers

```
# Zero data segment registers DS, ES, and  
ss.  
  
xorw %ax,%ax      # Set %ax to zero  
  
movw %ax,%ds      # -> Data Segment  
  
movw %ax,%es      # -> Extra Segment  
  
movw %ax,%ss      # -> Stack Segment
```

- **zero ax and ds, es, ss**
- **BIOS did not put in anything perhaps**

A not so necessary detail Enable 21 bit address

seta20.1:

```
inb    $0x64,%al          # Wait for not busy
testb   $0x2,%al
jnz    seta20.1
movb    $0xd1,%al          # 0xd1 -> port
0x64
```

```
outb   %al,$0x64
```

seta20.2:

```
inb    $0x64,%al          # Wait for not busy
testb   $0x2,%al
jnz    seta20.2
movb    $0xdf,%al          # 0xdf -> port 0x60
outb   %al,$0x60
```

- **Seg:off with 16 bit segments can actually address more than 20 bits of memory. After 0x100000 (=2²⁰), 8086 wrapped addresses to 0.**
- **80286 introduced 21st bit of address. But older software required 20 bits only. BIOS disabled 21st bit. Some OS needed 21st Bit. So enable it.**
- **Write to Port 0x64 and 0x60 -> keyboard controller**
 - to enable 21st bit out of address translation
 - **Why?** Before the A20, i.e. 21st bit was introduced, it belonged to keyboard controller
 - **For more details see https://en.wikipedia.org/wiki/A20_line**

https://en.wikipedia.org/wiki/A20_line

After this

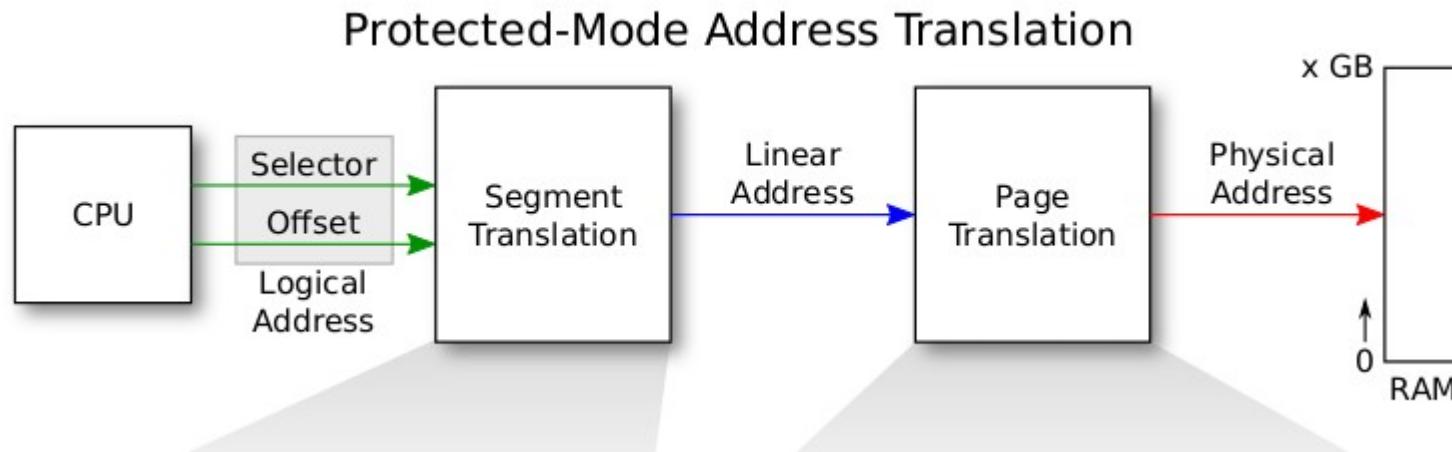
**Some instructions are run
to enter protected mode**

And further code runs in protected mode

Real mode Vs protected mode

- **Real mode 16 bit registers**
- **Protected mode**
 - Enables segmentation + Paging both
 - No longer seg*16+offset calculations
 - Segment registers is index into segment descriptor table. But segment:offset pairs continue
 - `mov %esp, $32 # SS will be used with esp`
 - More in next few slides
 - Other segment registers need to be explicitly mentioned in instructions
 - `Mov FS:$200, 30`
 - **32 bit registers**
 - can address upto 2^{32} memory
 - Can do arithmetic in 32 bits

X86 address : protected mode address translation



**Both Segmentation and Paging are used in x86
X86 allows optionally one-level or two-level paging**

**Segmentation is a must to setup, paging is optional (needs to be enabled)
Hence different OS can use segmentation+paging in different ways**

X86 segmentation

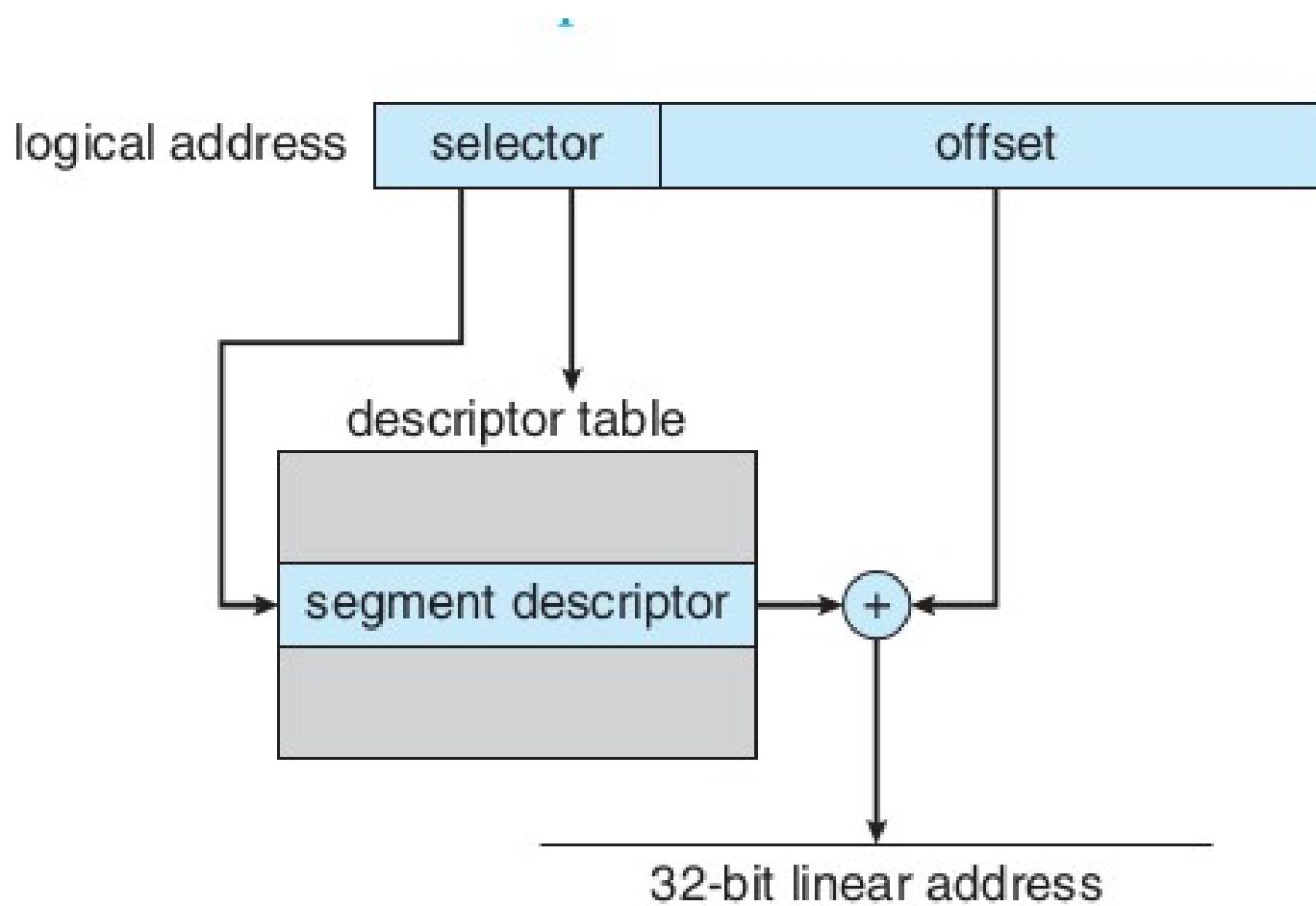


Figure 8.22 IA-32 segmentation.

Paging concept, hierarchical paging

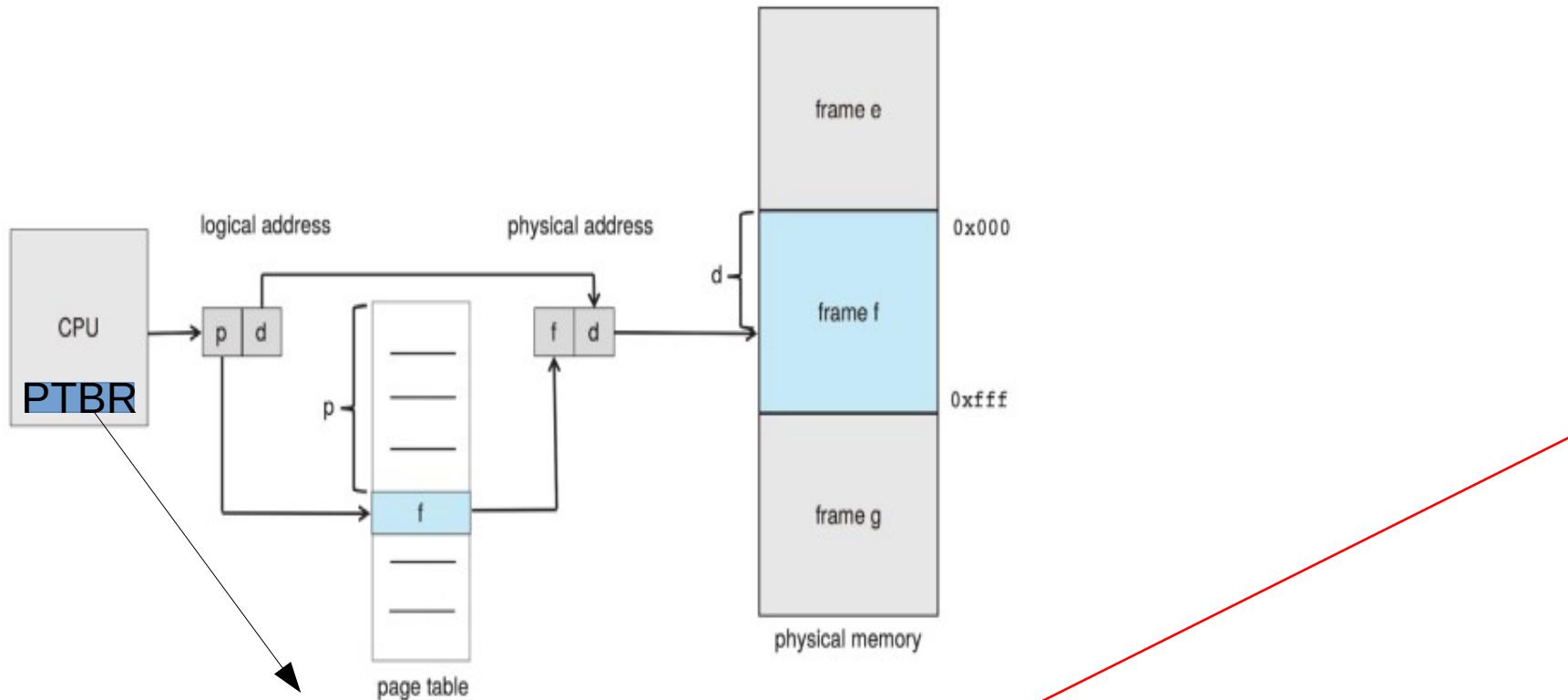


Figure 9.8 Paging hardware.

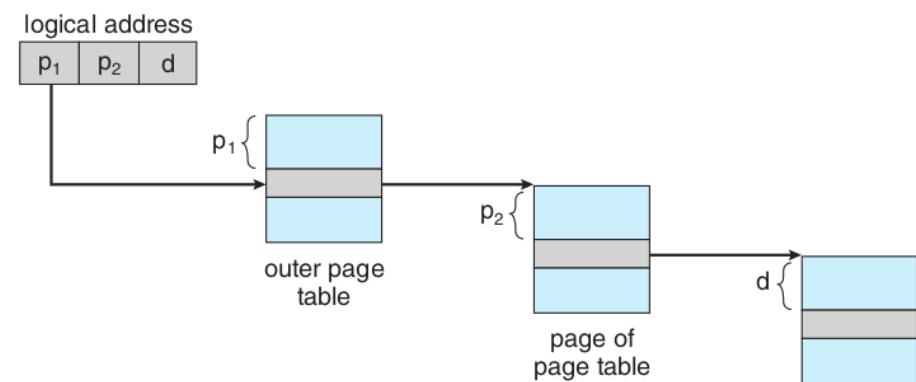


Figure 9.16 Address translation for a two-level 32-bit paging architecture.

X86 paging

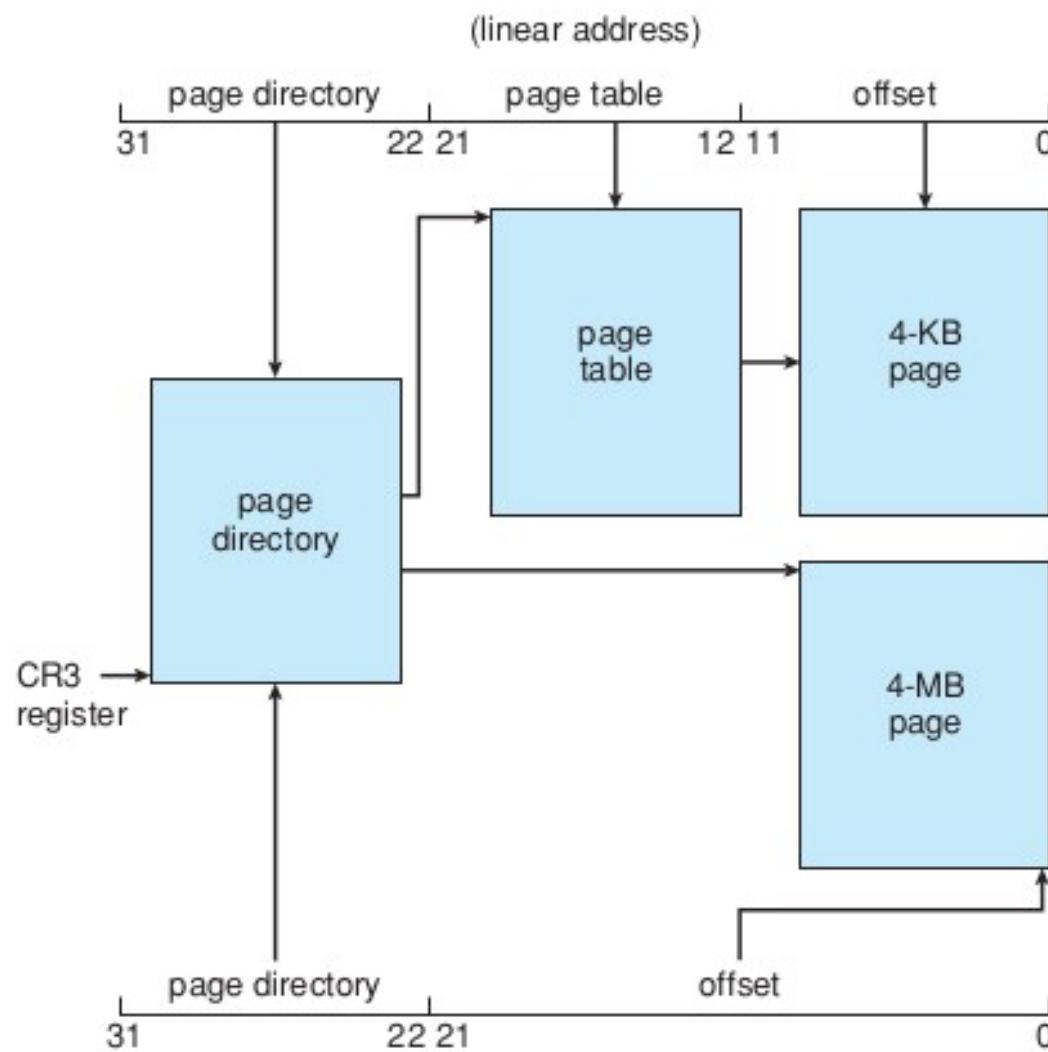


Figure 8.23 Paging in the IA-32 architecture.

Page Directory Entry (PDE) Page Table Entry (PTE)

31

| Page table physical page number | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|---------------------------------|-------------|--------|--------|---|--------|--------|--------|--------|--------|--------|--------|--------|---|
| | A V L | G S | P 0 | A | C D | W T | U D | W T | U D | W T | U D | W T | P |

PDE

P Present

W Writable

U User

WT 1=Write-through, 0=Write-back

CD Cache disabled

A Accessed

D Dirty

PS Page size (0=4KB, 1=4MB)

PAT Page table attribute index

G Global page

AVL Available for system use

31

| Physical page number | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|----------------------|-------------|--------|--------|--------|--------|--------|--------|--------|--------|--------|--------|--------|---|
| | A V L | G T | P A | D A | A D | C D | W T | U D | W T | U D | W T | U D | P |

PTE

CR3

CR3



PWT Page-level writes transparent

PCT Page-level cache disable

CR4

CR4



VME Virtual-8086 mode extensions

PVI Protected-mode virtual interrupts

TSD Time stamp disable

DE Debugging extensions

PSE Page size extensions

PAE Physical-address extension

MCE Machine check enable

PGE Page-global enable

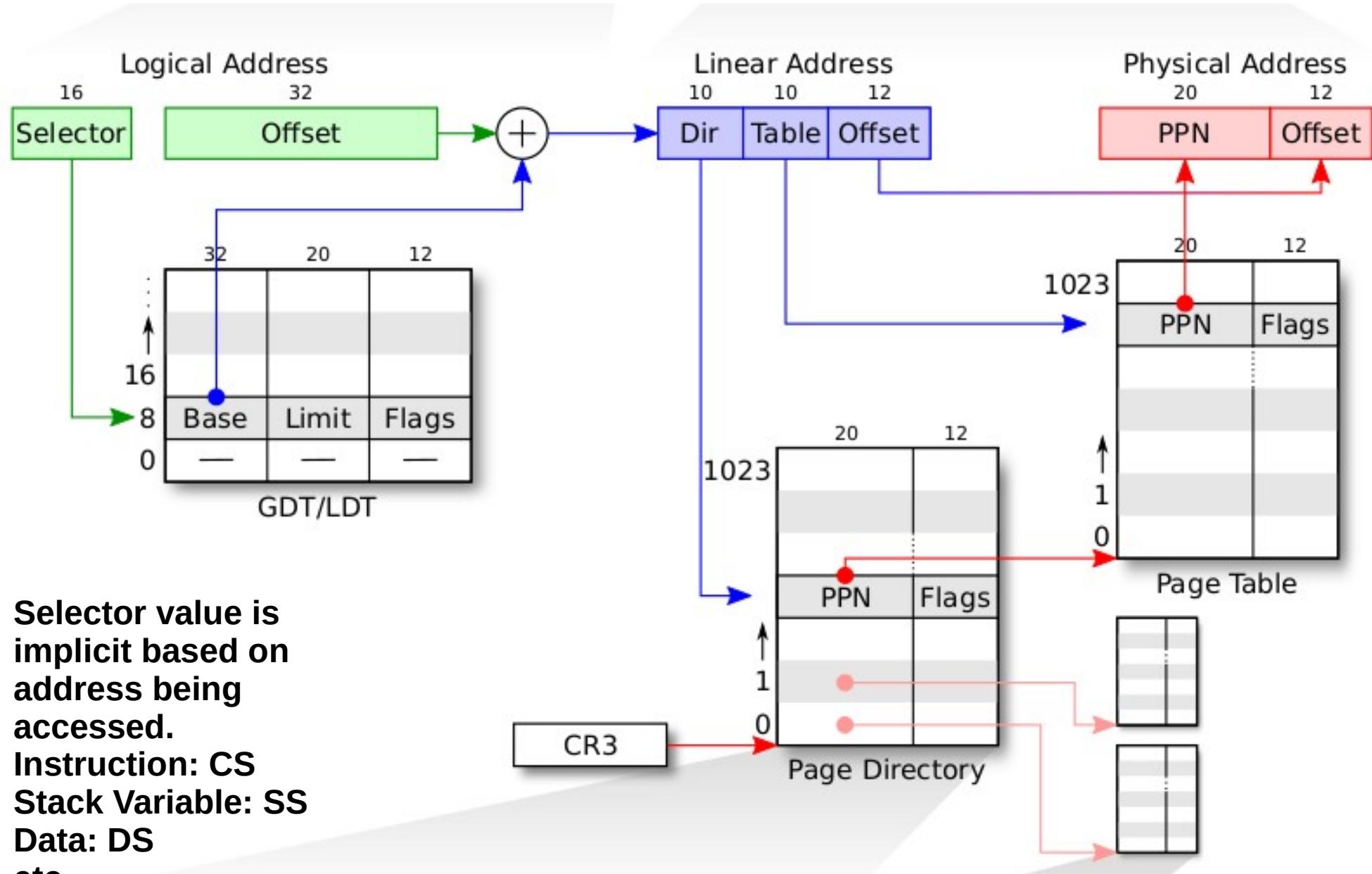
PCE Performance counter enable

OSFXSR OS FXSAVE/FXRSTOR support

OSXMM- OS unmasked exception support

EXCPT

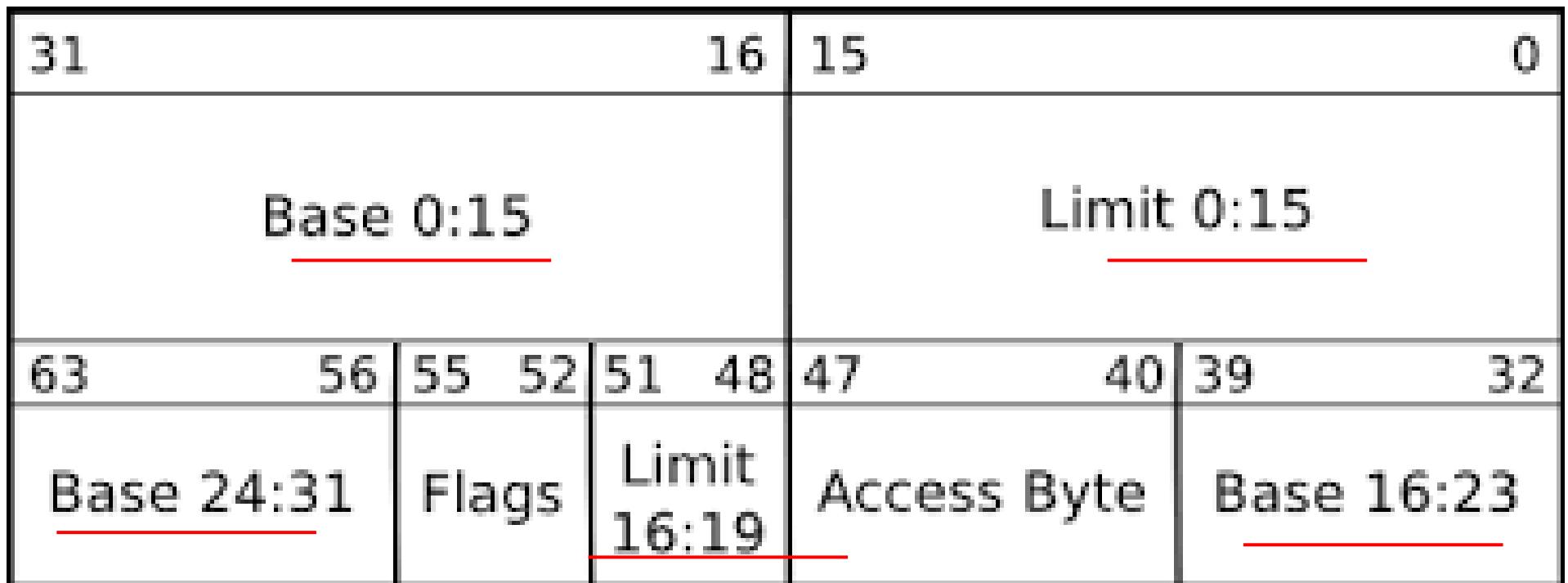
Segmentation + Paging



Segmentation + Paging setup of xv6

- xv6 configures the segmentation hardware by setting Base = 0, Limit = 4 GB
 - translate logical to linear addresses without change, so that they are always equal.
 - Segmentation is practically off
- Once paging is enabled, the only interesting address mapping in the system will be linear to physical.
 - In xv6 paging is NOT enabled while loading kernel
 - After kernel is loaded 4 MB pages are used for a while
 - Later the kernel switches to 4 kB pages!

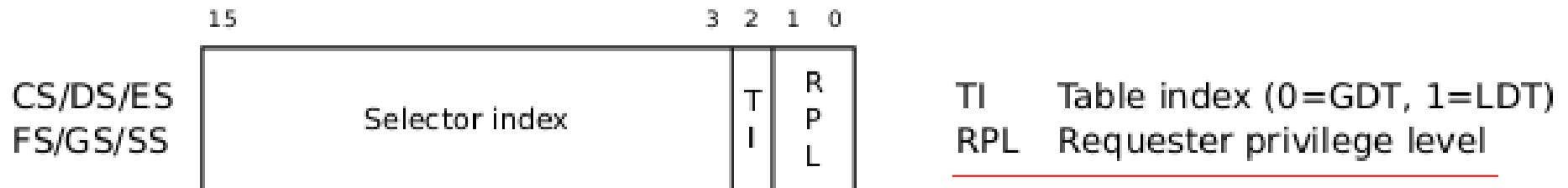
GDT Entry



asm.h

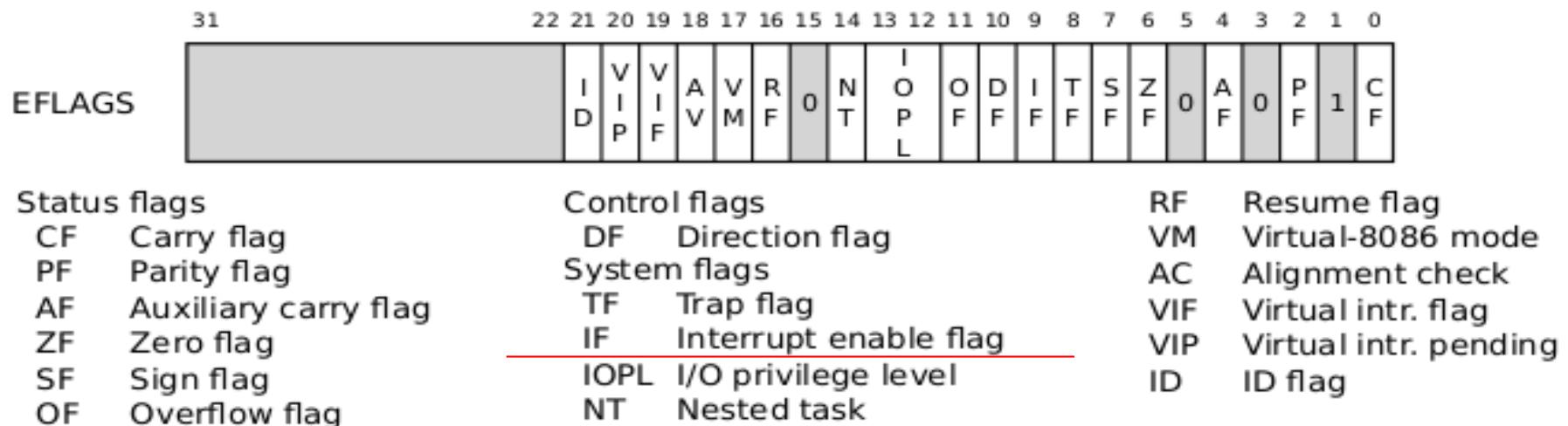
```
#define SEG_ASM(type,base,lim) \
    .word (((lim) >> 12) & 0xffff), ((base) & 0xffff); \
    .byte (((base) >> 16) & 0xff), (0x90 | (type)), \
          (0xC0 | (((lim) >> 28) & 0xf)), (((base) >> 24) & 0xff)
```

Segment selector



Note in 16 bit mode, segment selector is 16 bit, here it's 13 bit + 3 bits

EFLAGS register



Igdt

Igdt gdtdesc

...

Bootstrap GDT

.p2align 2 # force 4 byte alignment

gdt:

SEG_NULLASM # null seg

SEG_ASM(STA_X|STA_R, 0x0,
0xffffffff) # code seg

SEG_ASM(STA_W, 0x0, 0xffffffff)

data seg

gdtdesc:

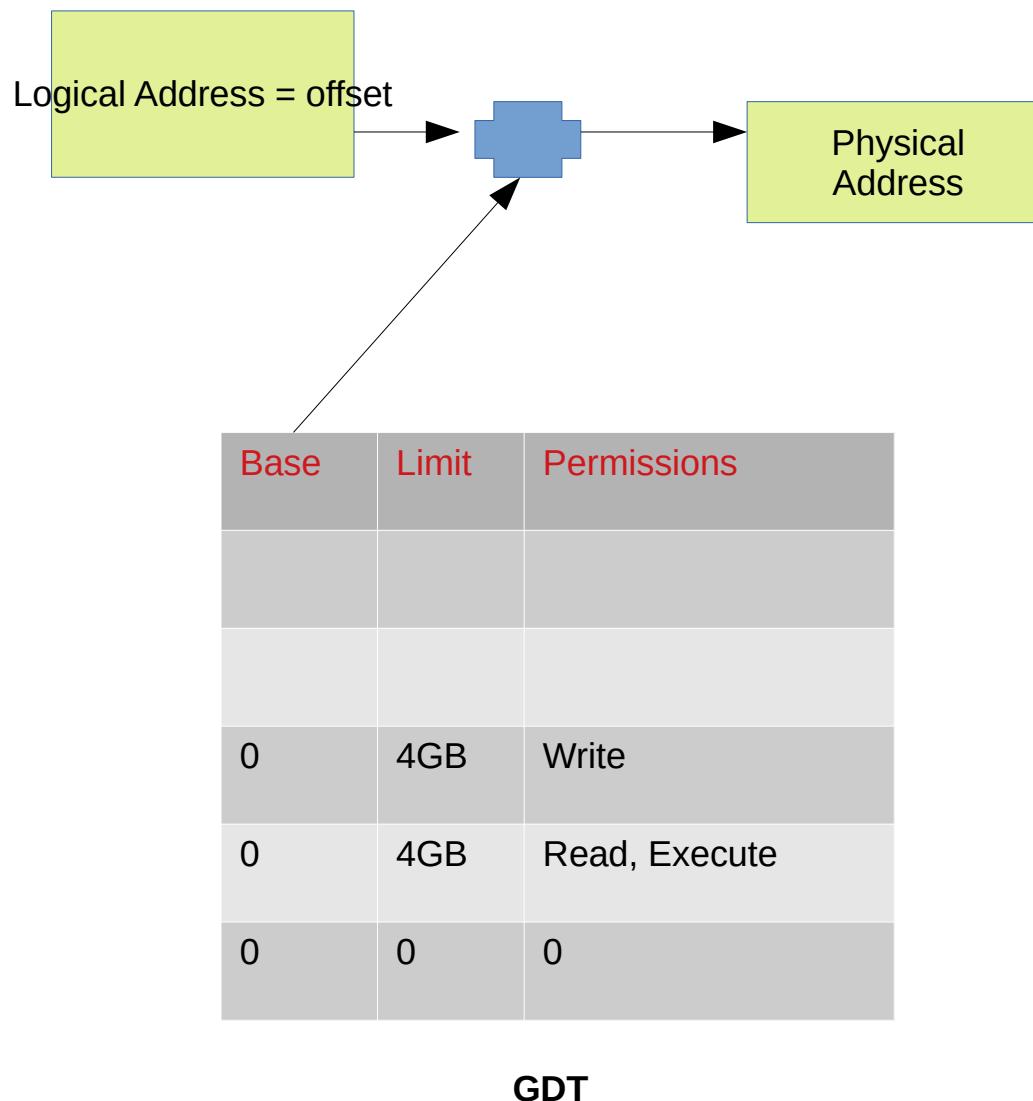
.word (gdtdesc - gdt - 1)

sizeof(gdt) - 1

.long gdt

- load the processor's (GDT) register with the value gdtdesc which points to the table gdt.
- **table gdt :** The table has a null entry, one entry for executable code, and one entry to data.
- all segments have a base address of zero and the maximum possible limit
- The code segment descriptor has a flag set that indicates that the code should run in 32-bit mode
- With this setup, when the boot loader enters protected mode, logical addresses map one-to-one to physical addresses.
- At gdtdesc we have this data
2, <4 byte addr of gdt>
Total 6 bytes
- GDTR is : <address 4 byte>, <table limit 2 byte>
- So Igdt gdtdesc loads these two values in GDTR

bootasm.S after “`lgdt gdtdesc`”
till jump to “`entry`”



Still
Logical Address =
Physical address!

But with GDT in picture
and
Protected Mode
operation

During this time,

**Loading kernel from
ELF into physical
memory**

**Addresses in “kernel”
file translate to same
physical address!**

Prepare to enable protected mode

- Prepare to enable protected mode by setting the 1 bit (CR0_PE) in register %cr0

```
movl %cr0, %eax  
orl $CR0_PE, %eax  
movl %eax, %cr0
```

CR0

| CR0 | 31 30 29 28 | 19 18 17 16 15 | 6 5 4 3 2 1 0 | | |
|-----|---------------------|----------------|-------------------|----|-------------------|
| | P G C D N W | A M W P | N E T S E M P P E | | |
| PE | Protection enabled | ET | Extension type | NW | Not write-through |
| MP | Monitor coprocessor | NE | Numeric error | CD | Cache disable |
| EM | Emulation | WP | Write protect | PG | Paging |
| TS | Task switched | AM | Alignment mask | | |

PG: Paging enabled or not

WP: Write protection on/off

PE: Protection Enabled --> protected mode.

Complete transition to 32 bit mode

`Ijmp $(SEG_KCODE<<3), $start32`

Complete the transition to 32-bit protected mode by using a long jmp

to reload %cs (=1) and %eip (=start32).

Note that ‘start32’ is the address of next instruction after Ijmp.

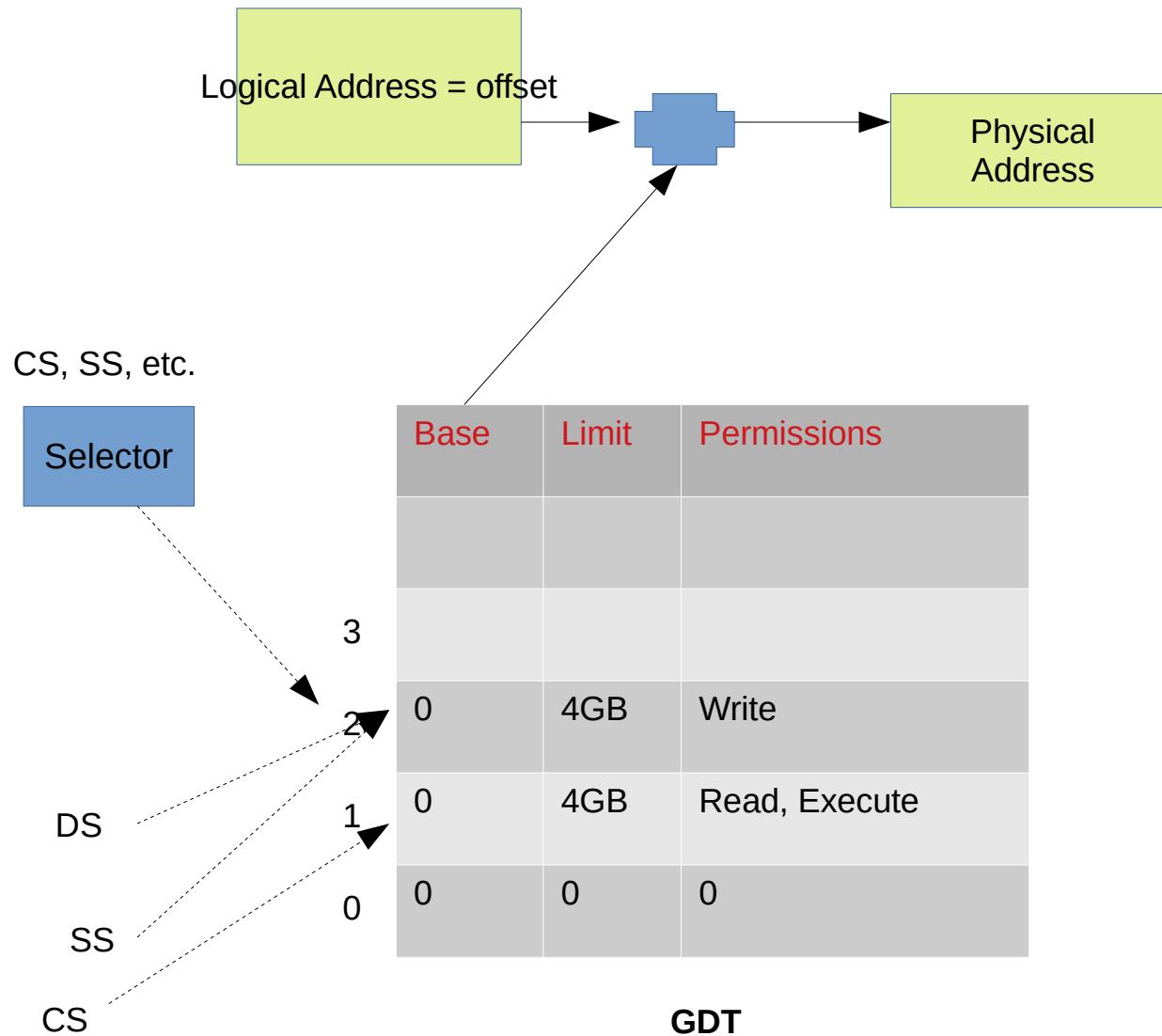
Note: The segment descriptors are set up with no translation (that is 0-4GB setting), so that the mapping is still the identity mapping.

Jumping to “C” code

```
movw $(SEG_KDATA<<3), %ax # Our data  
segment selector  
  
    movw %ax, %ds      # -> DS: Data  
Segment  
  
    movw %ax, %es      # -> ES: Extra  
Segment  
  
    movw %ax, %ss      # -> SS: Stack  
Segment  
  
    movw $0, %ax        # Zero segments  
not ready for use  
  
    movw %ax, %fs      # -> FS  
  
    movw %ax, %gs      # -> GS  
  
  
# Set up the stack pointer and call into C.  
movl $start, %esp  
call bootmain
```

- **Setup Data, extra, stack segment with SEG_KDATA (=2), FS & GS (=0)**
- **Copy “\$start” i.e. 7c00 to stack-ptr**
 - It will grow from 7c00 to 0000
- **Call bootmain() a C function**
 - In bootmain.c

Setup now



bootmain(): already in memory, as part of ‘bootblock’

- **bootmain.c , expects to find a copy of the kernel executable on the disk starting at the second sector (sector = 1).**
 - Why?
- **The kernel is an ELF format binary**
- **Bootmain loads the first 4096 bytes of the ELF binary. It places the in-memory copy at address 0x10000**
- **readseg() is a function that runs OUT instructions in particular IO ports, to issue commands to read from Disk**

```
void bootmain(void)
{
    struct elfhdr *elf;
    struct proghdr *ph, *eph;
    void (*entry)(void);
    uchar* pa;

    elf = (struct elfhdr*)0x10000; // scratch space

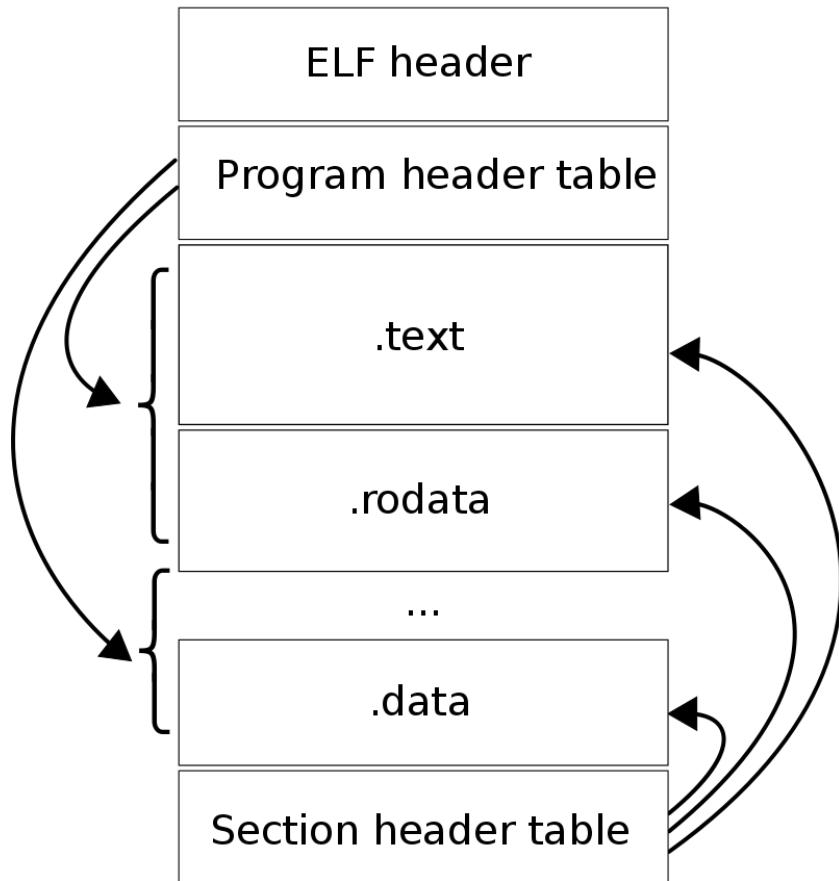
    // Read 1st page off disk
    readseg((uchar*)elf, 4096, 0);
```

bootmain()

- Check if it's really ELF or not // Is this an ELF executable?
- Next load kernel code from ELF file “kernel” into memory

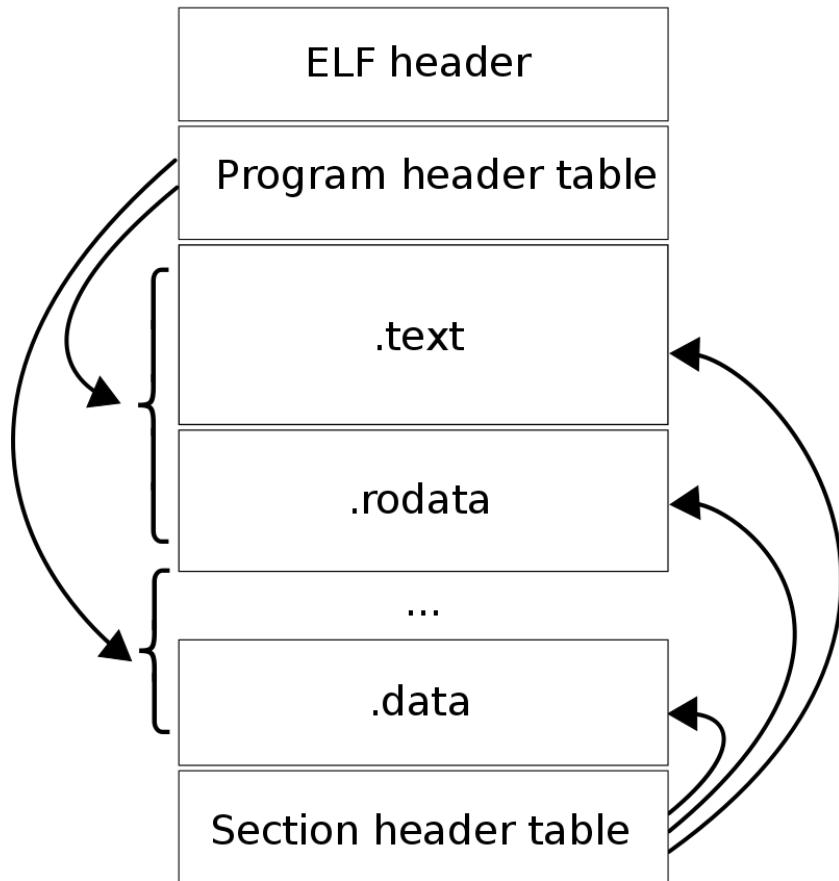
```
if(elf->magic != ELF_MAGIC)
    return; // let bootasm.S handle error
```

ELF



```
struct elfhdr {  
    uint magic; // must equal  
    ELF_MAGIC  
    uchar elf[12];  
    ushort type;  
    ushort machine;  
    uint version;  
    uint entry;  
    uint phoff; // where is program  
    header table  
    uint shoff;  
    uint flags;  
    ushort ehsiz;  
    ushort phentsiz;  
    ushort phnum; // no. Of program  
    header entries  
    ushort shentsiz;  
    ushort shnum;  
    ushort shstrndx;  
};
```

ELF



// Program header

struct proghdr {

 uint type; // Loadable segment ,
 Dynamic linking information ,
 Interpreter information , Thread-
 Local Storage template , etc.

 uint off; //Offset of the segment
 in the file image.

 uint vaddr; //Virtual address of
 the segment in memory.

 uint paddr; // physical address to
 load this program, if PA is relevant

 uint filesz; //Size in bytes of the
 segment in the file image.

 uint memsz; //Size in bytes of the
 segment in memory. May be 0.

 uint flags;

 uint align;

};

Run ‘objdump -x -a kernel | head -15’ & see this

```
kernel: file format elf32-i386  
kernel  
architecture: i386, flags 0x000000112:  
EXEC_P, HAS_SYMS, D_PAGED  
start address 0x0010000c
```

Program Header:

```
LOAD off 0x00001000 vaddr 0x80100000 paddr 0x00100000 align 2**12  
  filesz 0x0000a516 memsz 0x000154a8 flags rwx  
STACK off 0x00000000 vaddr 0x00000000 paddr 0x00000000 align 2**4  
  filesz 0x00000000 memsz 0x00000000 flags rwx
```

Code to be
loaded at
KERNBASE +
KERNLINK

Diff
between
memsz &
filesz, will
be filled
with zeroes
in memory

Stack :
everything
zeroes

Load code from ELF to memory

```
// Load each program segment (ignores ph flags).
ph = (struct proghdr*)((uchar*)elf + elf->phoff);
eph = ph + elf->phnum;
// Abhijit: number of program headers
for(; ph < eph; ph++){
    // Abhijit: iterate over each program header
    pa = (uchar*)ph->paddr;
    // Abhijit: the physical address to load program
    /* Abhijit: read ph->filesz bytes, into 'pa',
       from ph->off in kernel/disk */
    readseg(pa, ph->filesz, ph->off);
    if(ph->memsz > ph->filesz)
        stosb(pa + ph->filesz, 0, ph->memsz - ph-
->filesz); // Zero the remainder section*/
}
```

Jump to Entry

```
// Call the entry point from the ELF header.  
// Does not return!  
/* Abhijit:  
 * elf->entry was set by Linker using kernel.ld  
 * This is address 0x80100000 specified in  
kernel.ld  
 * See kernel.asm for kernel assembly code).  
 */  
entry = (void(*)(void))(elf->entry);  
entry();
```

To understand
further
code

Remember: 4
MB pages
are possible

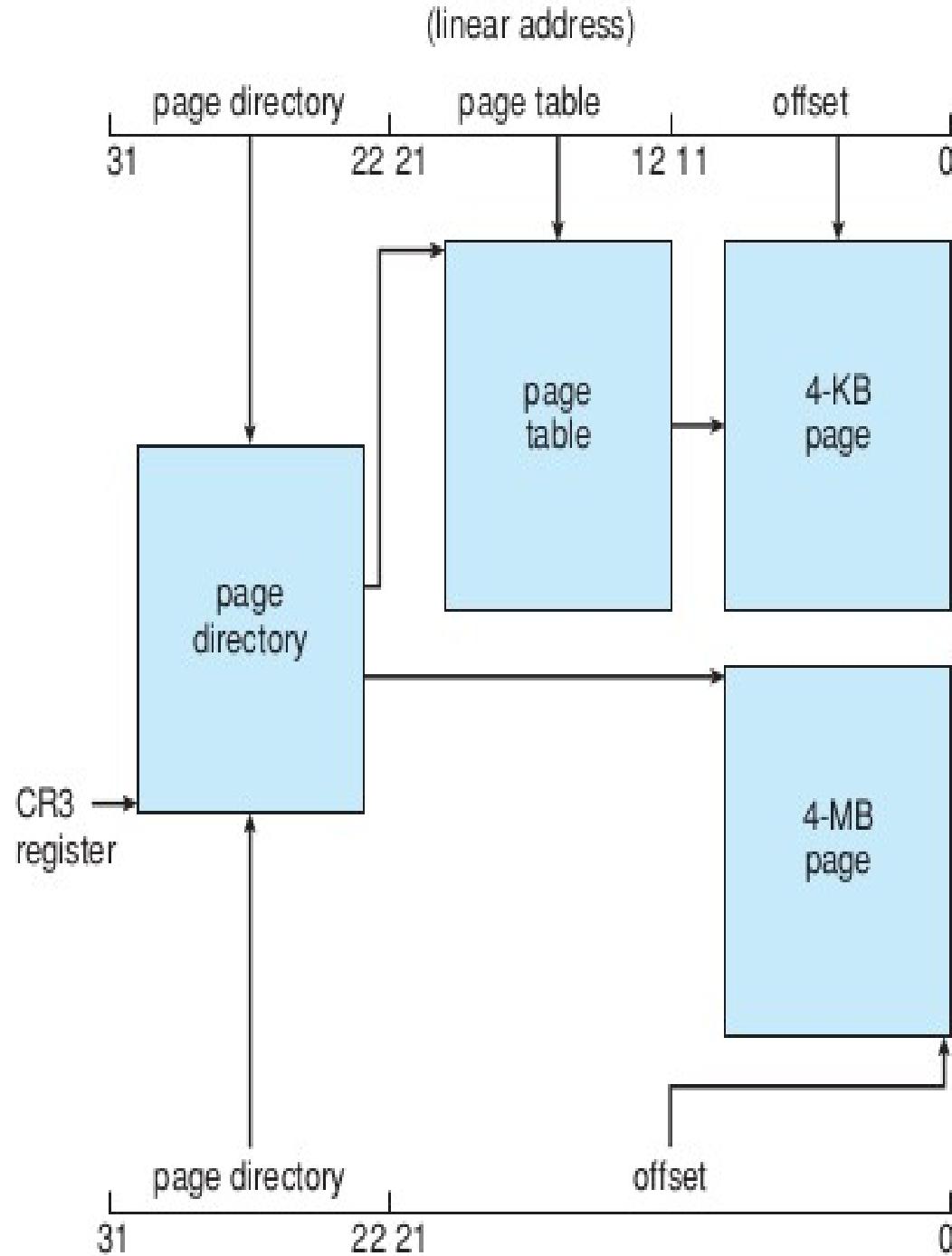
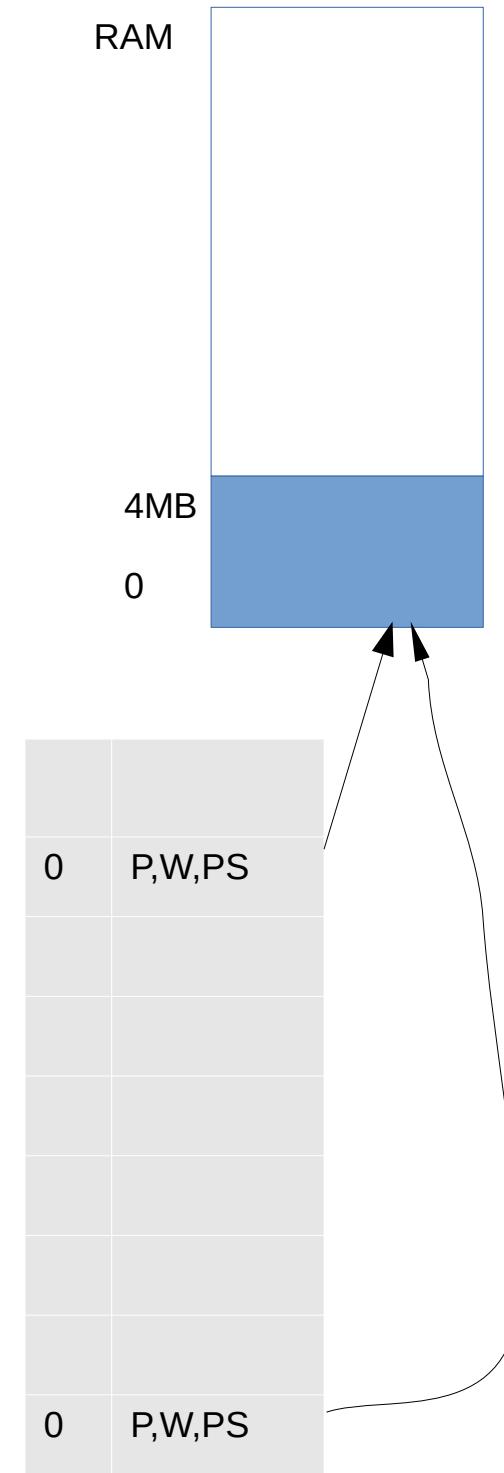
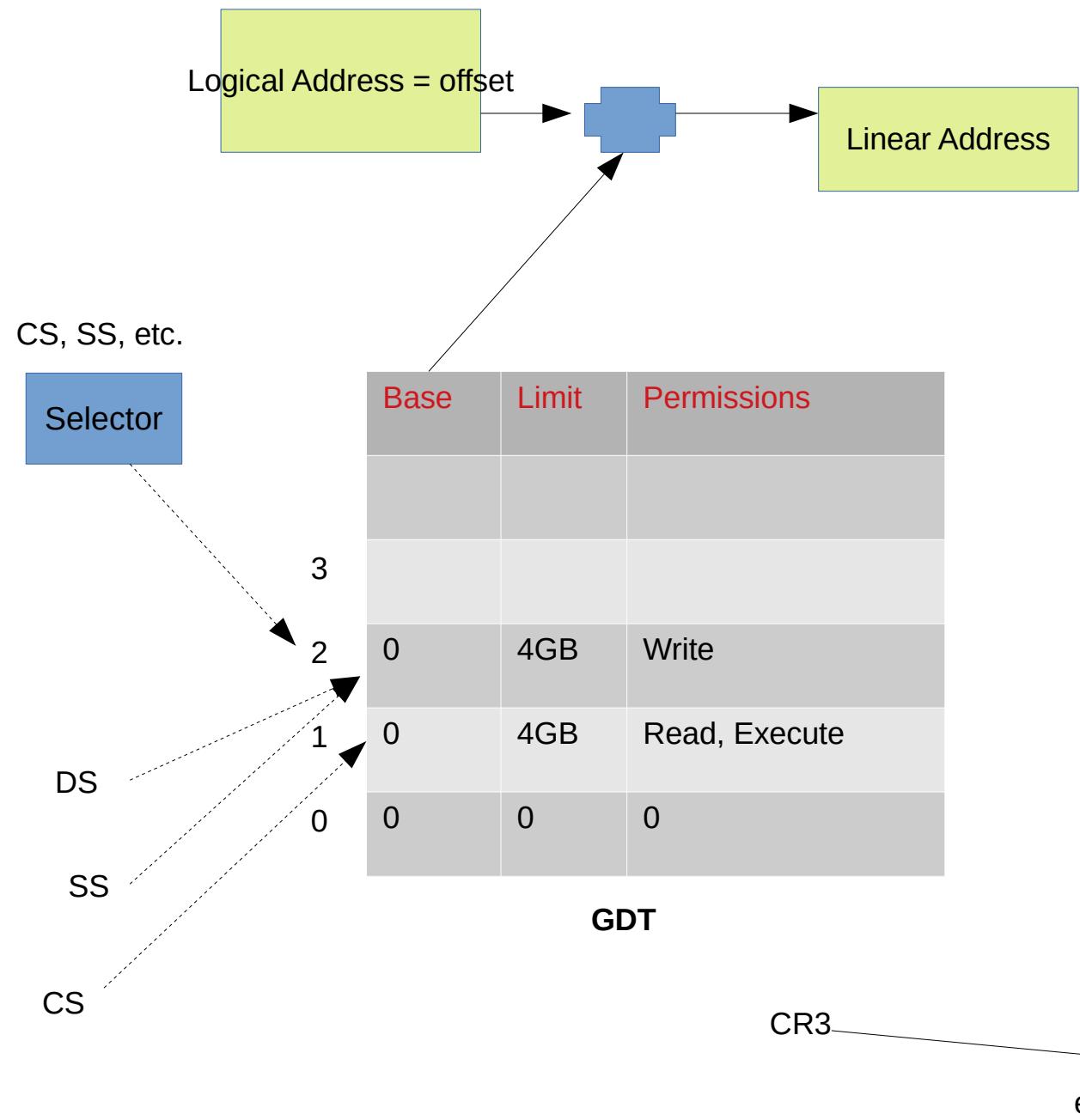
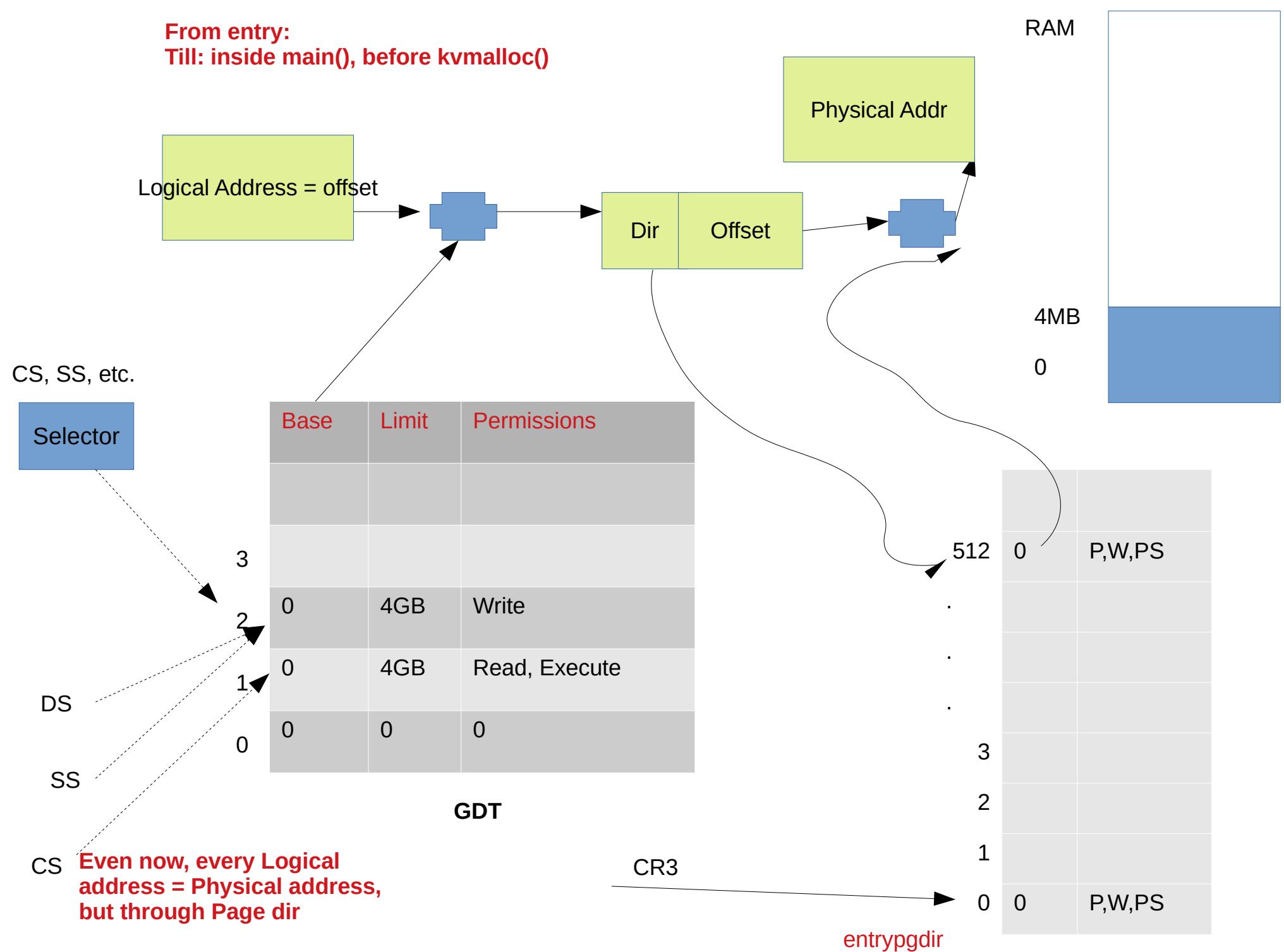


Figure 8.23 Paging in the IA-32 architecture.

**From entry:
Till: inside main(), before kvmalloc()**



From entry:
Till: inside main(), before kvmalloc()



entrypgdir in main.c, is used by entry()

```
_attribute_((__aligned__(PGSIZE)))
pde_t entrypgdir[NPDENTRIES] = {
    // Map VA's [0, 4MB) to PA's [0, 4MB)
    [0] = (0) | PTE_P | PTE_W | PTE_PS,
    // Map VA's [KERNBASE, KERNBASE+4MB) to PA's [0, 4MB). This is entry 512
    [KERNBASE>>PDXSHIFT] = (0) | PTE_P | PTE_W | PTE_PS,
};
```

This is entry page directory during entry(), beginning of kernel
Mapping 0:0x400000 (i.e. 0: 4MB) to physical addresses 0:0x400000. is required as long as entry is executing at low addresses, but will eventually be removed.
This mapping restricts the kernel instructions and data to 4 Mbytes.

| | |
|------------------|---|
| #define PTE_P | 0x001 // Present |
| #define PTE_W | 0x002 // Writeable |
| #define PTE_U | 0x004 // User |
| #define PTE_PS | 0x080 // Page Size |
| #define PDXSHIFT | 22 // offset of PDX in a linear address |

entry() in entry.S

entry:

```
movl %cr4, %eax  
orl $(CR4_PSE), %eax  
movl %eax, %cr4  
movl $(V2P_WO(entrypgdir)),  
%eax  
movl %eax, %cr3  
movl %cr0, %eax  
orl $(CR0_PG|CR0_WP), %eax  
movl %eax, %cr0  
movl $(stack + KSTACKSIZE),  
%esp  
mov $main, %eax  
jmp *%eax
```

- # Turn on page size extension for 4Mbyte pages
- # Set page directory. 4 MB pages (temporarily only. More later)
- # Turn on paging.
- # Set up the stack pointer.
- # Jump to main(), and switch to executing at high addresses. The indirect call is needed because the assembler produces a PC-relative instruction for a direct jump.
-

More about entry()

```
movl $(V2P_WO(entrypgdir)), %eax  
movl %eax, %cr3
```

-> Here we use physical address using V2P_WO because paging is not turned on yet

- **V2P is simple: subtract 0x80000000 i.e. KERNBASE from address**

More about entry()

```
movl %cr0, %eax  
orl $(CR0_PG|  
CR0_WP), %eax  
movl %eax, %cr0
```

- But we have already set 0'th entry in pgdir to address 0
- So it still works!

This turns on paging

After this also, entry() is running and processor is executing code at lower addresses

entry()

```
movl $(stack +  
KSTACKSIZE), %esp
```

```
mov $main, %eax  
jmp *%eax
```

```
.comm stack,  
KSTACKSIZE
```

```
# Abhijit: allocate here 'stack' of size =  
KSTACKSIZE
```

- # Set up the stack pointer.
- # Abhijit: +KSTACKSIZE is done as stack grows downwards
- # Jump to main(), and switch to executing at high addresses. The indirect call is needed because the assembler produces a PC-relative instruction for a direct jump.

bootmasm.S bootmain.c: Steps

- 1) Starts in “real” mode, 16 bit mode. Does some misc legacy work.**
- 2) Runs instructions to do MMU set-up for protected-mode & only segmentation (0-4GB, identity mapping), changes to protected mode.**
- 3) Reads kernel ELF file and loads it in RAM, as per instructions in ELF file**
- 4) Sets up paging (4 MB pages)**
- 5) Runs main() of kernel**

Code from bootasm.S bootmain.c is over!
Kernel is loaded.
Now kernel is going to prepare itself

Processes

Abhijit A M

abhijit.comp@coep.ac.in

Process related data structures in kernel code

- Kernel needs to maintain following types of data structures for managing processes
 - List of all processes
 - Memory management details for each, files opened by each etc.
 - Scheduling information about the process
 - Status of the process
 - List of processes “waiting” for different events to occur,
 - Etc.

Process Control Block



Figure 3.3 Process control block (PCB).

- A record representing a process in operating system's data structures
- OS maintains a “list” of PCBs, one for each process
- Called “`struct task_struct`” in Linux kernel code and “`struct proc`” in xv6 code

Fields in PCB



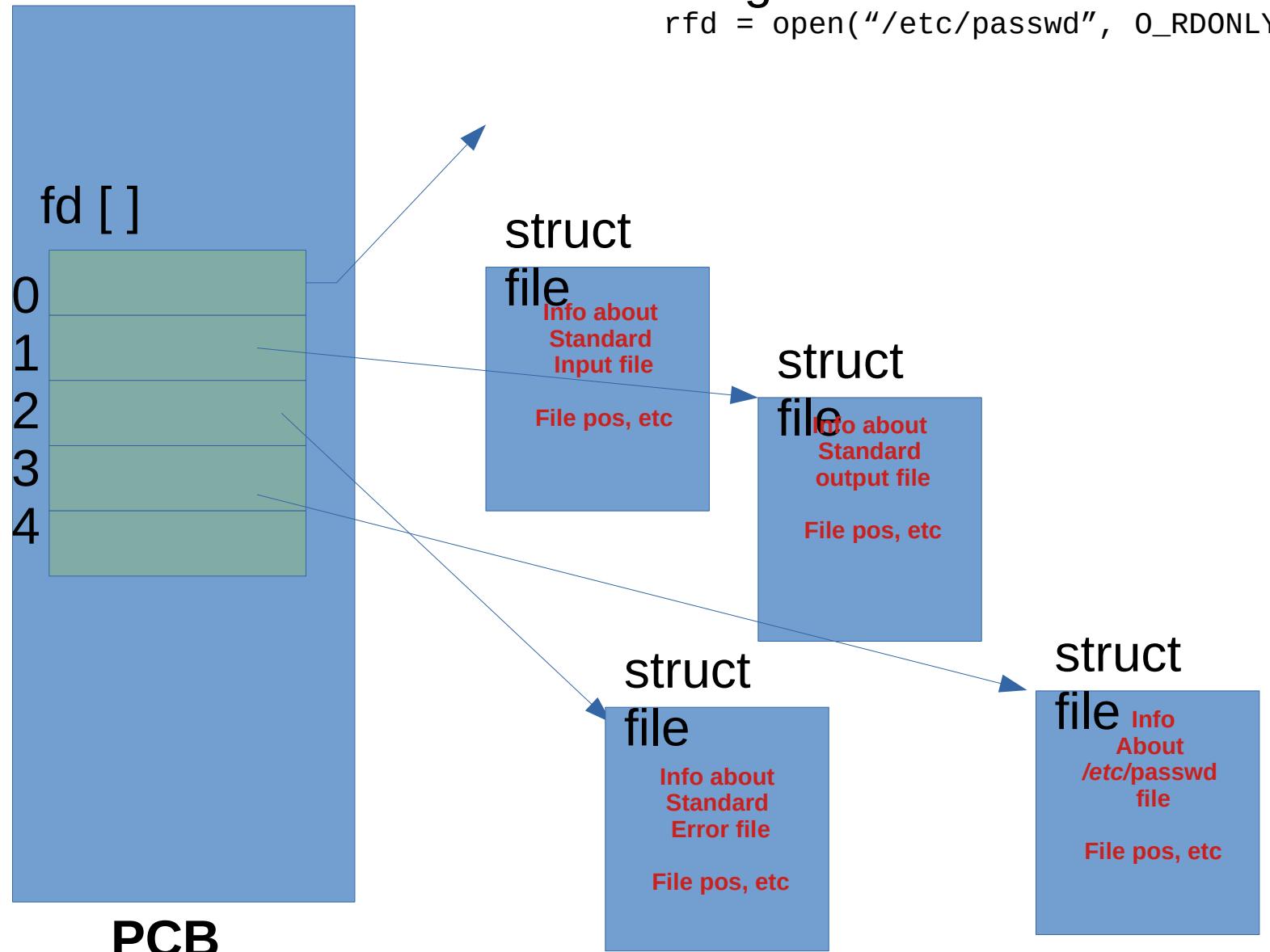
Figure 3.3 Process control block (PCB).

- Process ID (PID)
- Process State
- Program counter
- Registers
- Memory limits of the process
- Accounting information
- I/O status
- Scheduling information
- array of file descriptors (list of open files)
- ...etc

List of open files

Program did

```
rfd = open("/etc/passwd", O_RDONLY)
```



List of open files

- The PCB contains an array of pointers, called file descriptor array (fd[]), pointers to structures representing files
- When open() system call is made
 - A new file structure is created and relevant information is stored in it
 - Smallest available of fd [] pointers is made to point to this new struct file
 - The index of this fd [] pointer is returned by open
- When subsequent calls are made to read(fd,) or write(fd, ...), etc.
 - The kernel gets the “fd” as an index in the fd[] array and is able to locate the file structure for that file

```
// XV6 Code : Per-process state
enum procstate { UNUSED, EMBRYO, SLEEPING,
RUNNABLE, RUNNING, ZOMBIE };

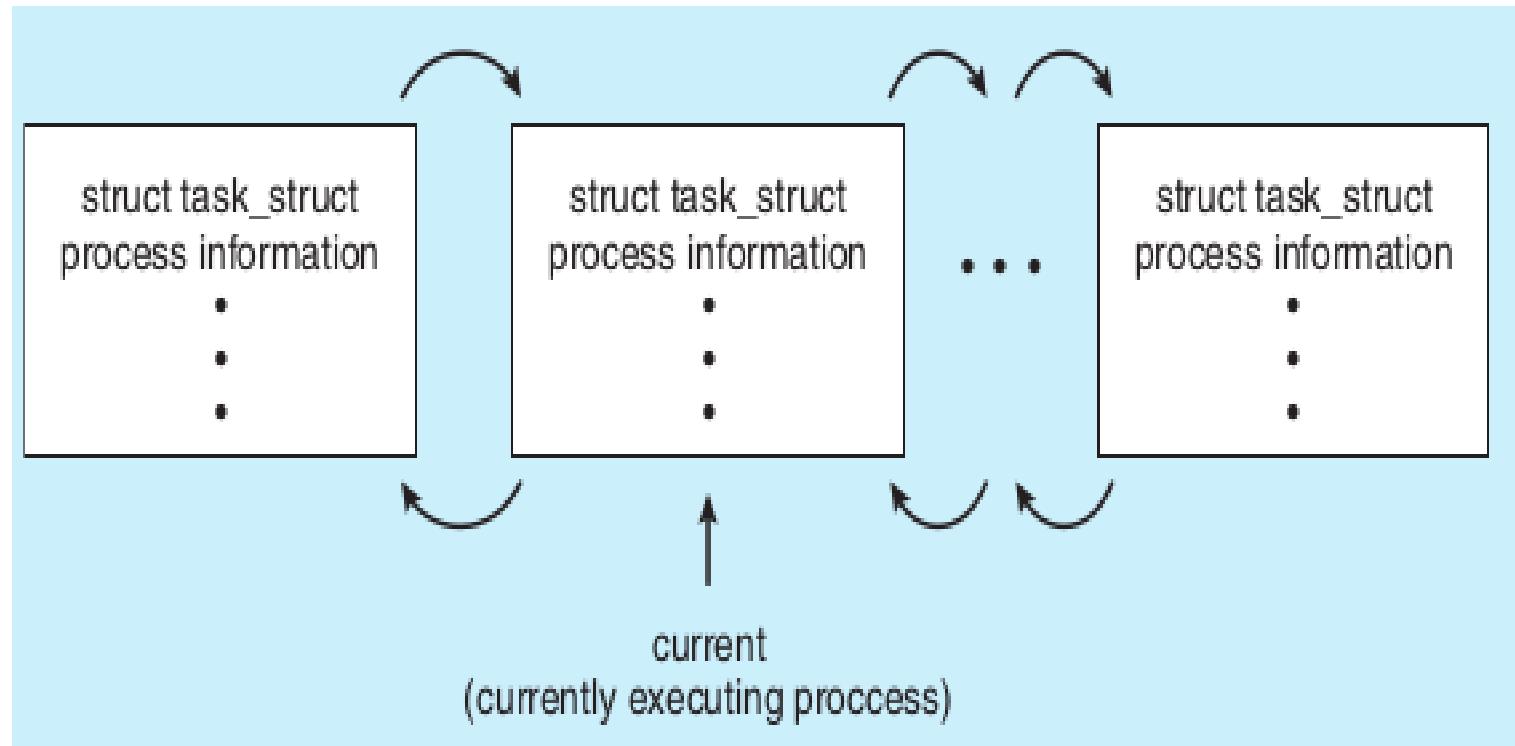
struct proc {
    uint sz;                      // Size of process memory (bytes)
    pde_t* pgdir;                 // Page table
    char *kstack;                 // Bottom of kernel stack for this
process
    enum procstate state;         // Process state
    int pid;                      // Process ID
    struct proc *parent;          // Parent process
    struct trapframe *tf;         // Trap frame for current syscall
    struct context *context;      // swtch() here to run process
    void *chan;                   // If non-zero, sleeping on chan
    int killed;                   // If non-zero, have been killed
    struct file *ofile[NFILE];    // Open files
    struct inode *cwd;            // Current directory
    char name[16];                // Process name (debugging)
};

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

```
struct file {
    enum { FD_NONE,
FD_PIPE, FD_INODE } type;
    int ref; // reference count
    char readable;
    char writable;
    struct pipe *pipe;
    struct inode *ip;
    uint off;
};
```

Process Queues/Lists inside OS

- Different types of queues/lists can be maintained by OS for the processes
 - A queue of processes which need to be scheduled
 - A queue of processes which have requested input/output to a device and hence need to be put on hold/wait
 - List of processes currently running on multiple CPUs
 - Etc.



// Linux data structure

```
struct task_struct {
    long state; /* state of the process */
    struct sched_entity se; /* scheduling information */
    struct task_struct *parent; /* this process's parent */
    struct list_head children; /* this process's children */
    struct files_struct *files; /* list of open files */
    struct mm_struct *mm; /* address space */
```

```
struct list_head {
    struct list_head *next, *prev;
};
```

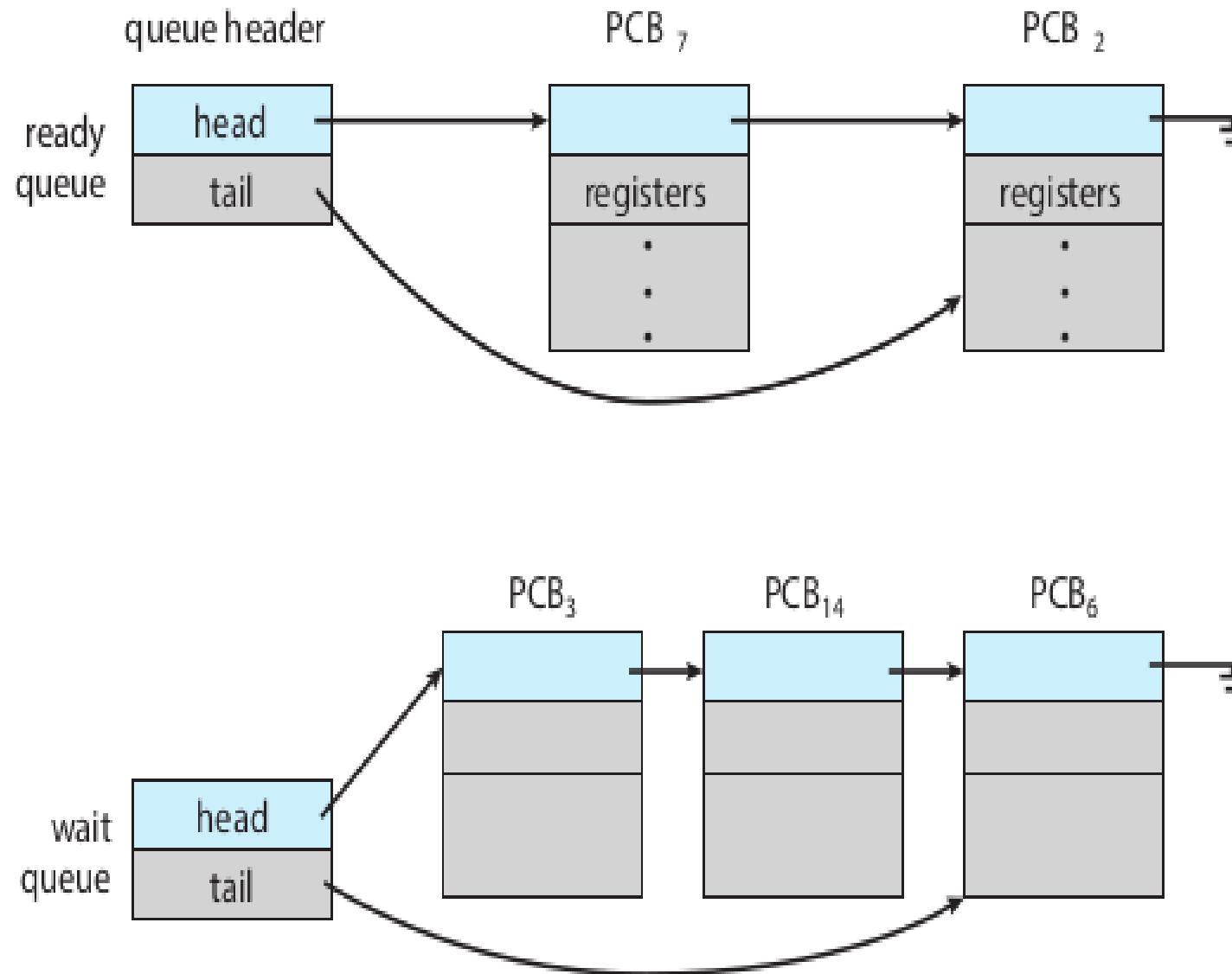


Figure 3.4 The ready queue and wait queues.

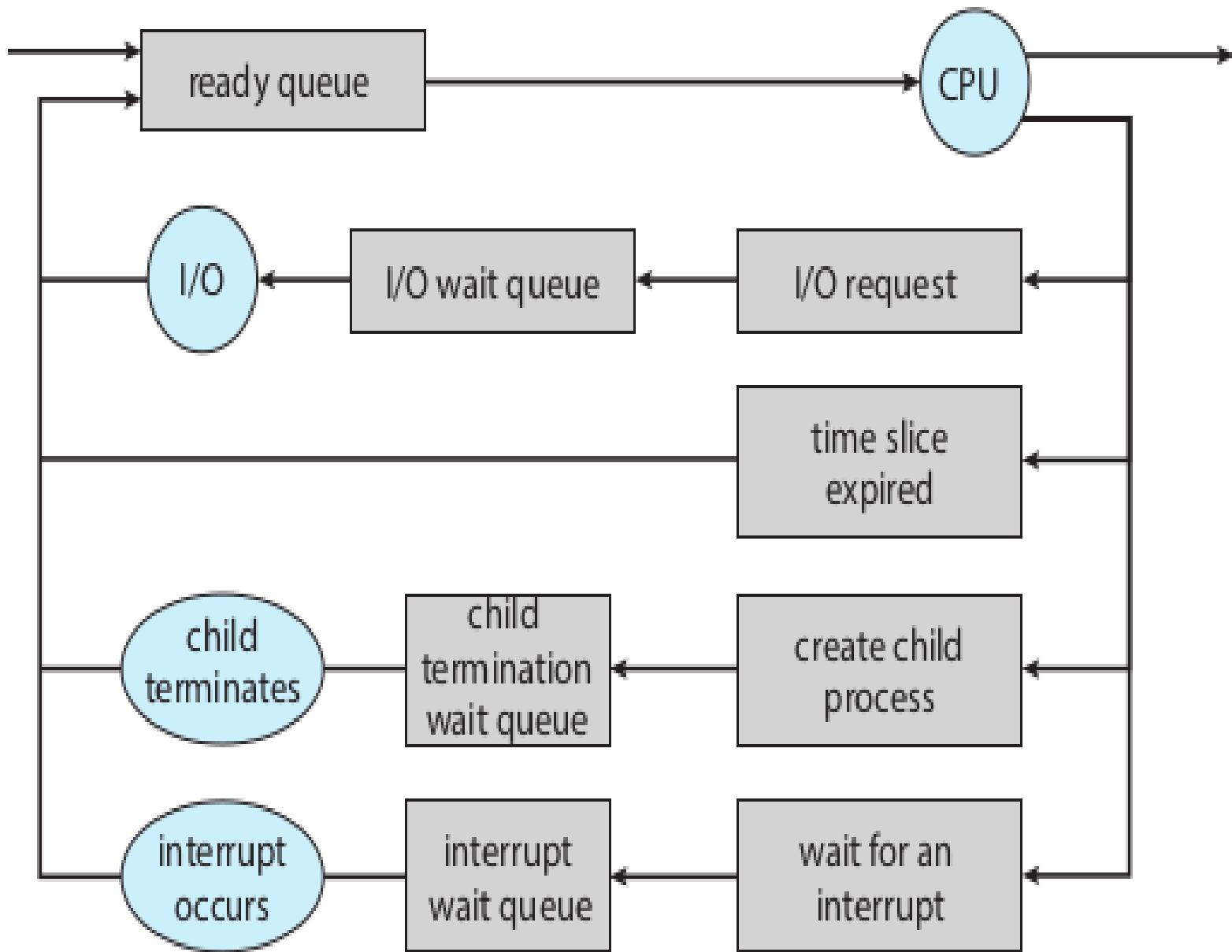


Figure 3.5 Queueing-diagram representation of process scheduling.

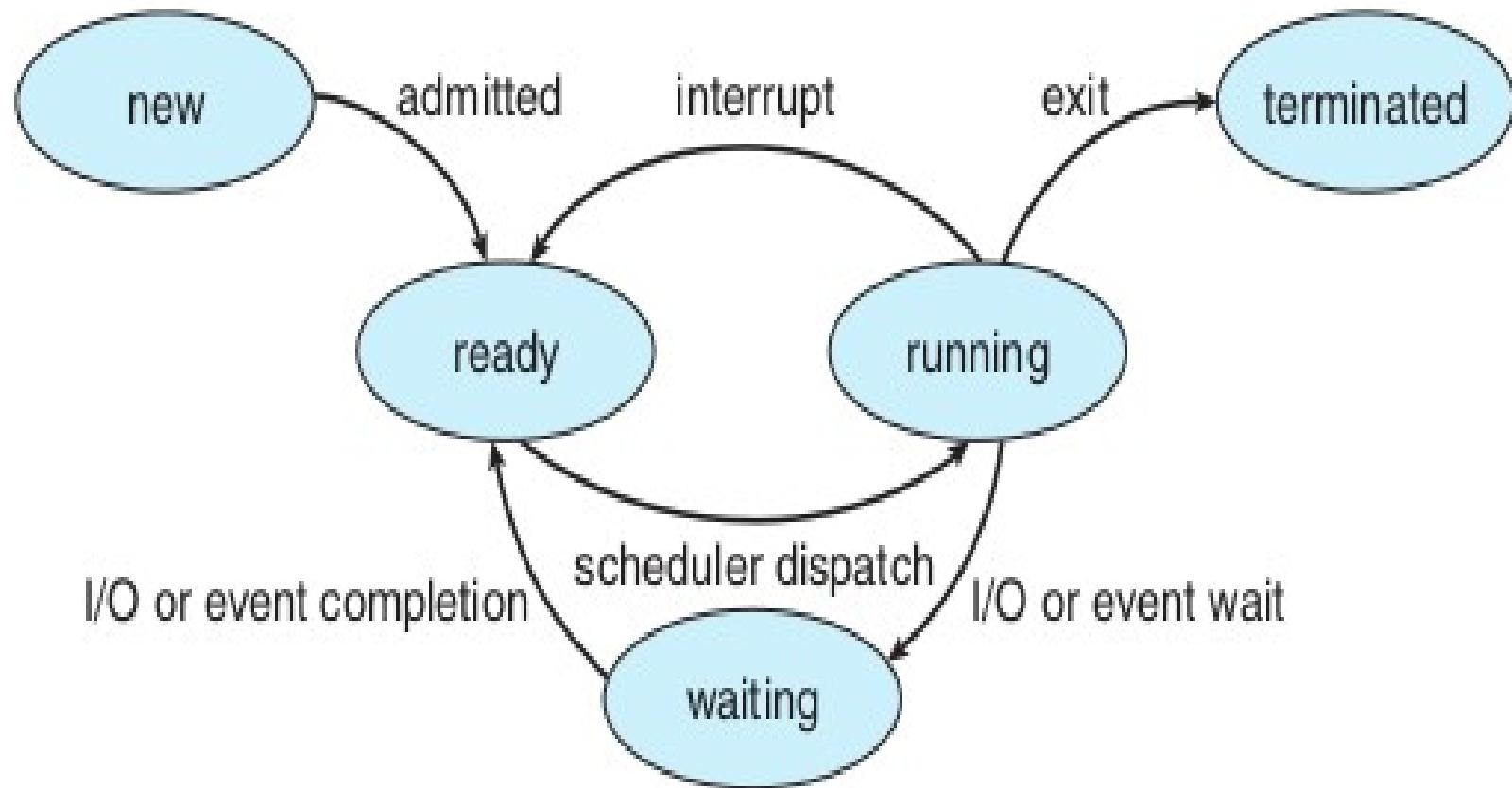


Figure 3.2 Diagram of process state.

Conceptual diagram

“Giving up” CPU by a process or blocking

```
int main() {  
    i = j + k;  
    scanf("%d", &k);  
}  
  
int scanf(char *x, ...) {  
    ...  
    read(0, ..., ...);  
}  
  
int read(int fd, char *buf, int len) {  
    ...  
    __asm__ { "int 0x80..." }  
    ...  
}
```

OS Syscall

```
sys_read(int fd, char *buf, int len) {  
    file f = current->fdarray[fd];  
    int offset = f->position;  
    ...  
    disk_read(..., offset, ...);  
    // Do what now?  
    //asynchronous read  
    //Interrupt will occur when the disk read is complete  
    // Move the process from ready queue to a wait queue and call scheduler!  
    // This is called "blocking"  
    Return the data read ;  
}  
disk_read(..., offset, ... ) {  
    __asm__("outb PORT ..");  
    return;  
}
```

“Giving up” CPU by a process or blocking

The relevant code in xv6 is in

Sleep()

The wakeup code is in wakeup() and wakeup1()

To be seen later

Context Switch

- Context
 - Execution context of a process
 - CPU registers, process state, memory management information, all configurations of the CPU that are specific to execution of a process/kernel
- Context Switch
 - Change the context from one process/OS to OS/another process
 - Need to save the old context and load new context
 - Where to save? --> PCB of the process

Context Switch

- Is an overhead
- No useful work happening while doing a context switch
- Time can vary from hardware to hardware
- Special instructions may be available to save a set of registers in one go

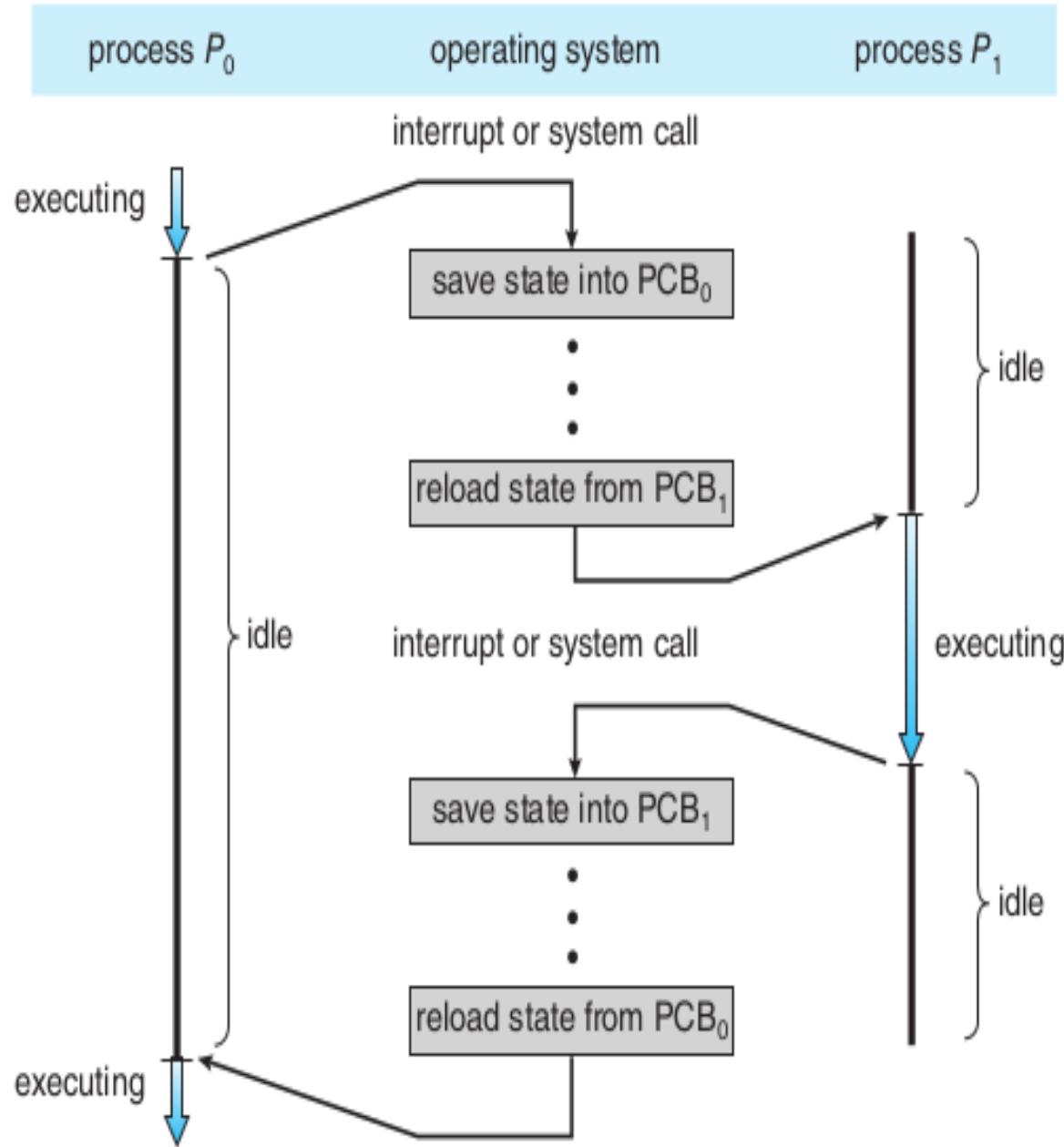


Figure 3.6 Diagram showing context switch from process to process.

Peculiarity of context switch

- When a process is running, the function calls work in LIFO fashion
 - Made possible due to calling convention
- When an interrupt occurs
 - It can occur anytime
 - Context switch can happen in the middle of execution of any function
- After context switch
 - One process takes place of another
 - This “switch” is obviously not going to happen using calling convention, as no “call” is happening
 - Code for context switch must be in assembly!

NEXT: XV6 code overview

1. Understanding how traps are handled
2. How timer interrupt goes to scheduler
 3. How scheduling takes place
4. How a “blocking” system call (e.g. read()) “blocks”

Processes in xv6 code

Process Table

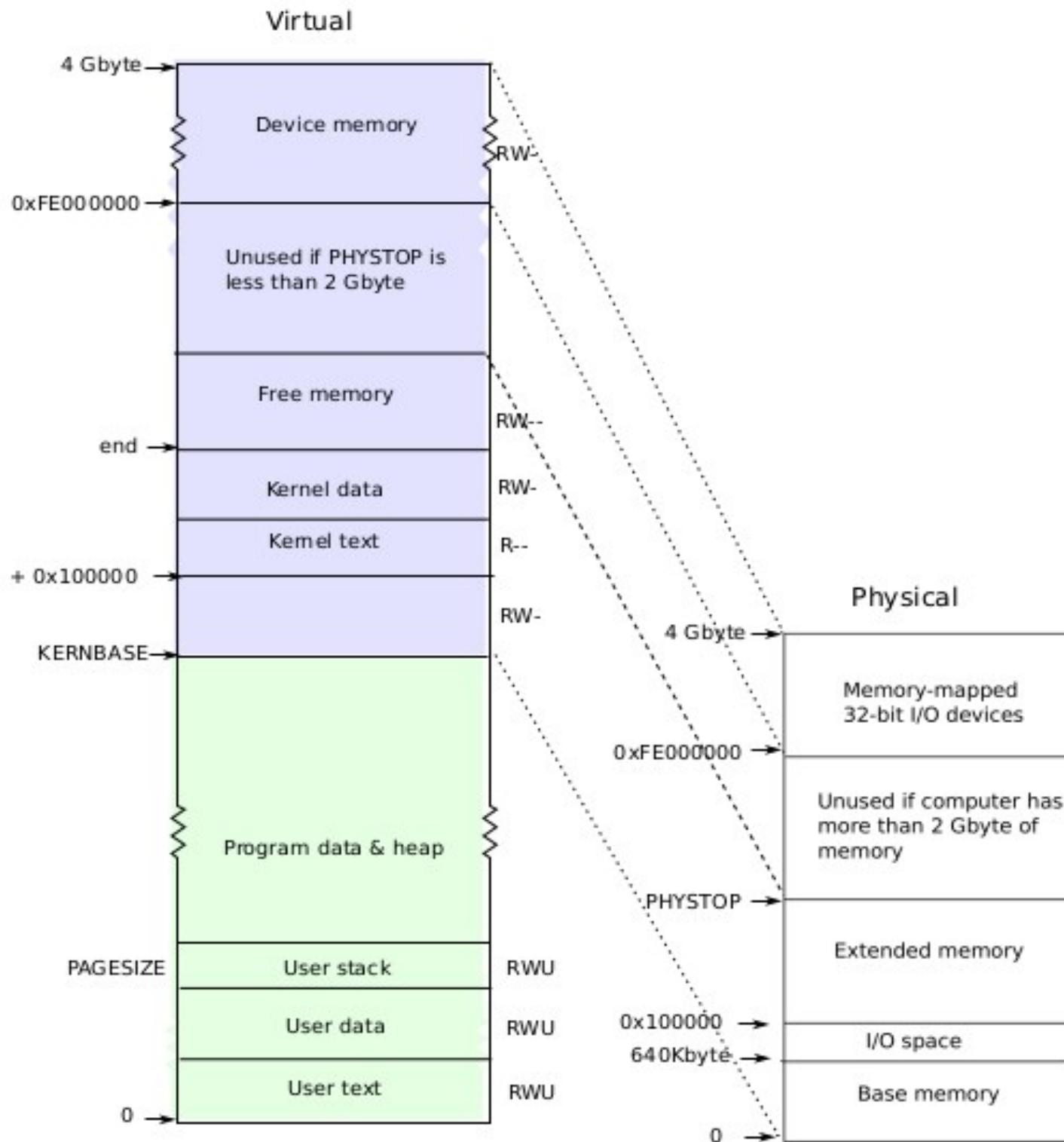
```
struct {  
    struct spinlock lock;  
    struct proc proc[NPROC];  
} ptable;
```

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

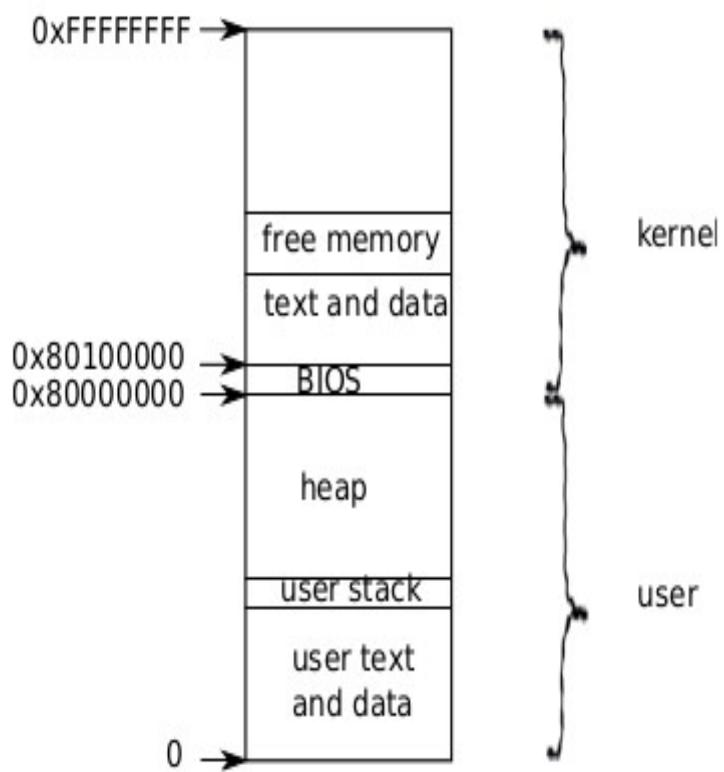
Layout of process's VA space

xv6
schema!

different
from Linux

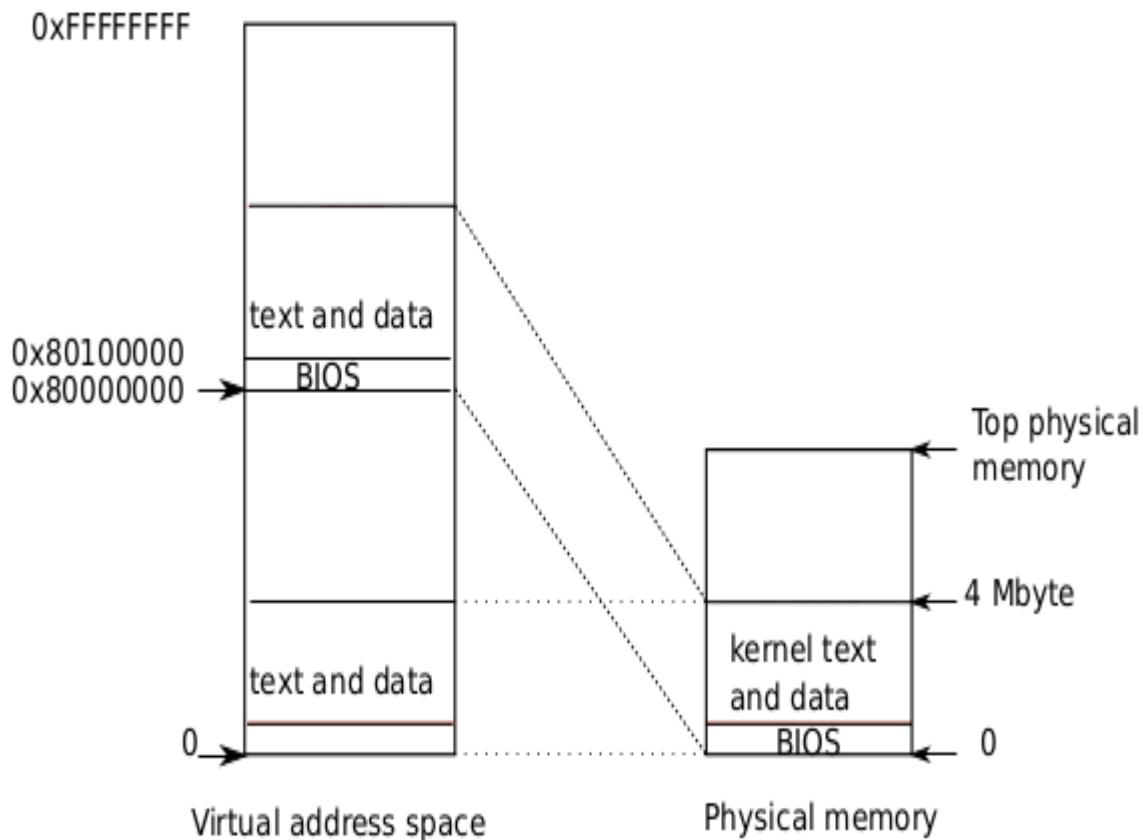


Logical layout of memory for a process



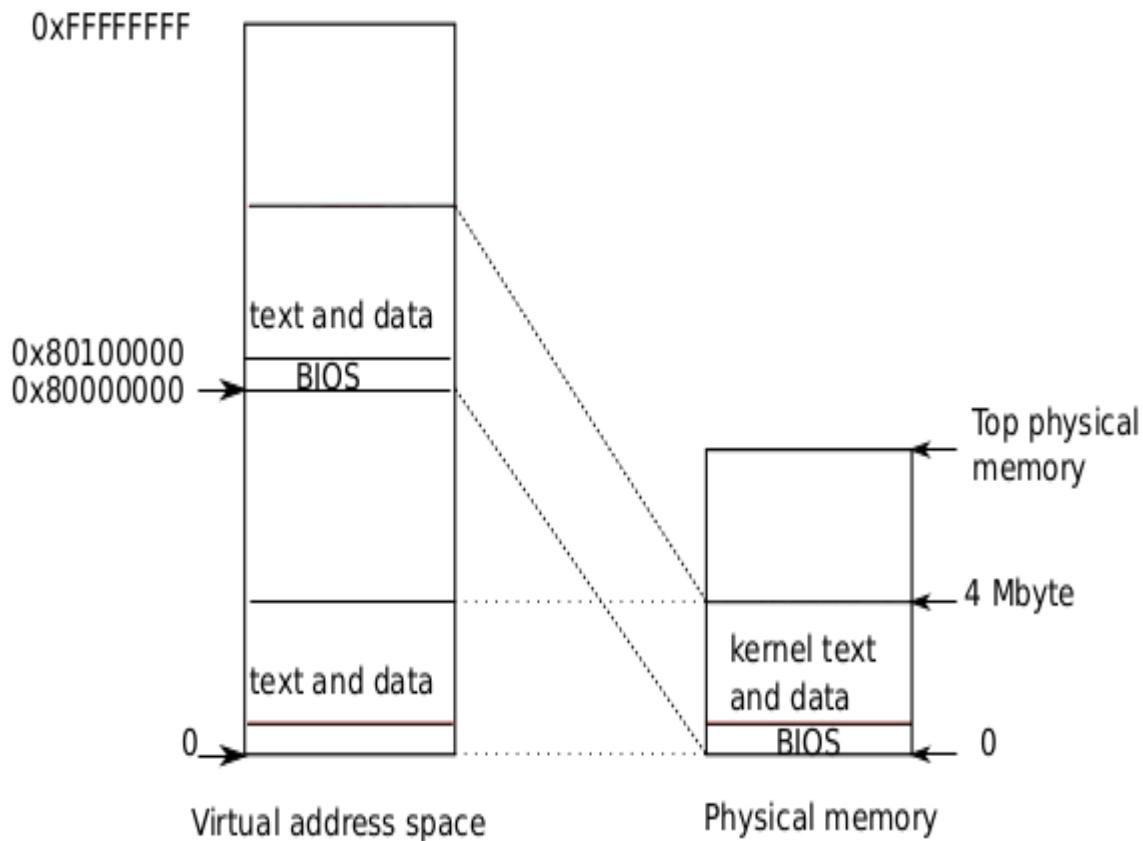
- **Address 0: code**
- **Then globals**
- **Then stack**
- **Then heap**
- **Each process's address space maps kernel's text, data also --> so that system calls run with these mappings**
- **Kernel code can directly access user memory now**

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

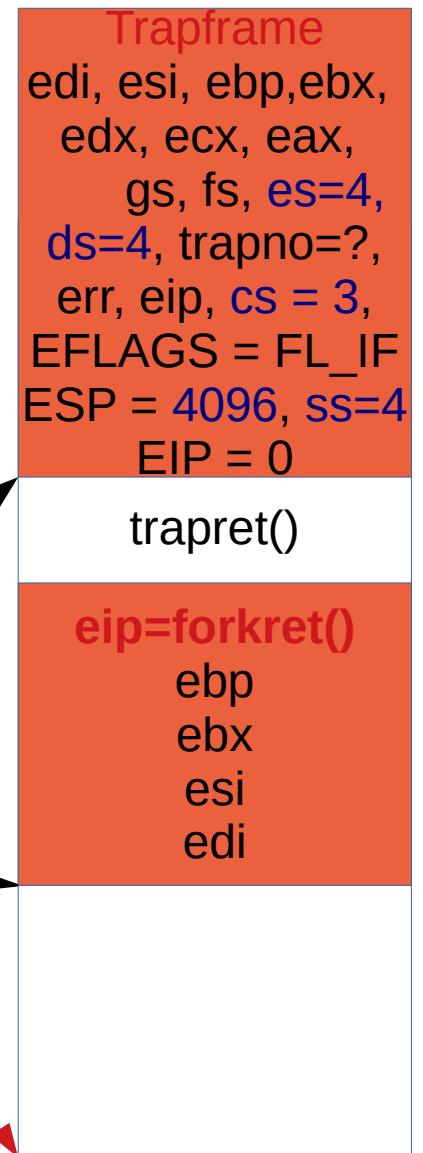
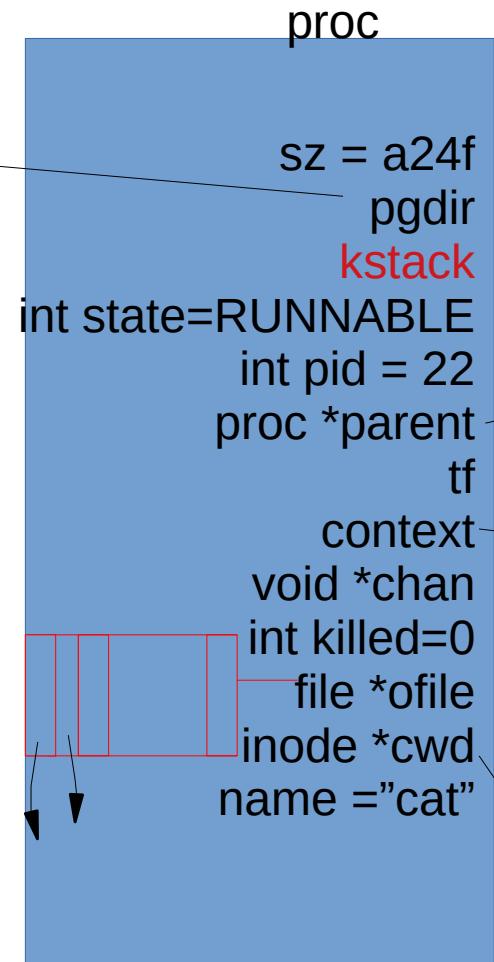
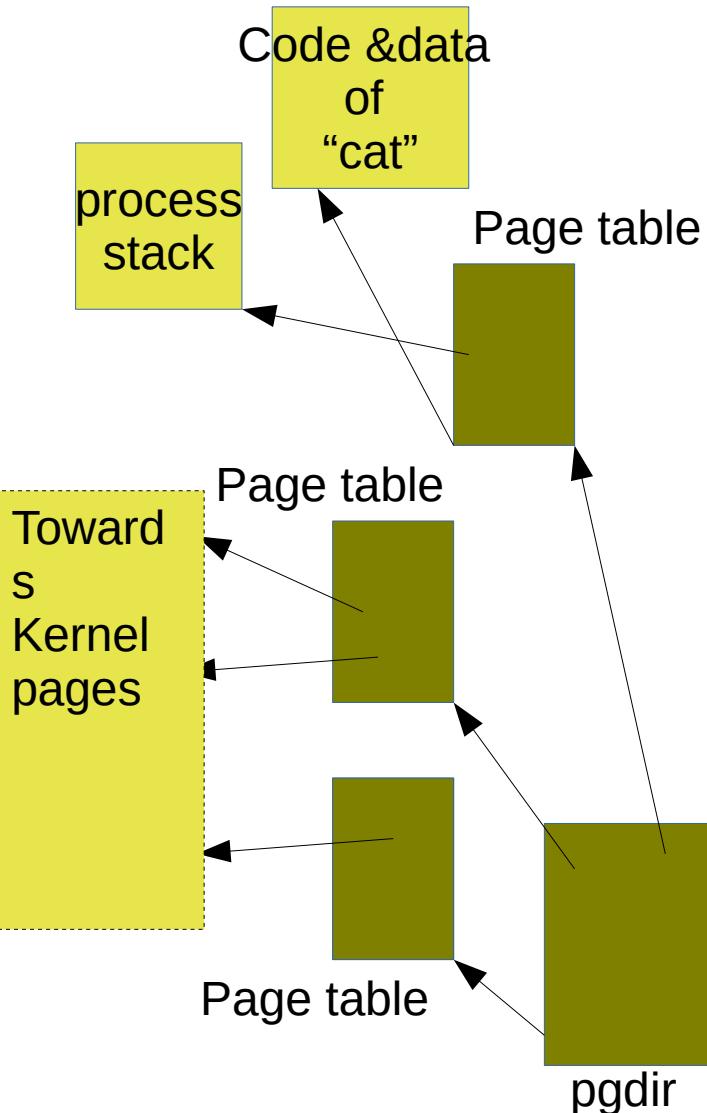
Imp Concepts

- **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, waiting 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)
};
```

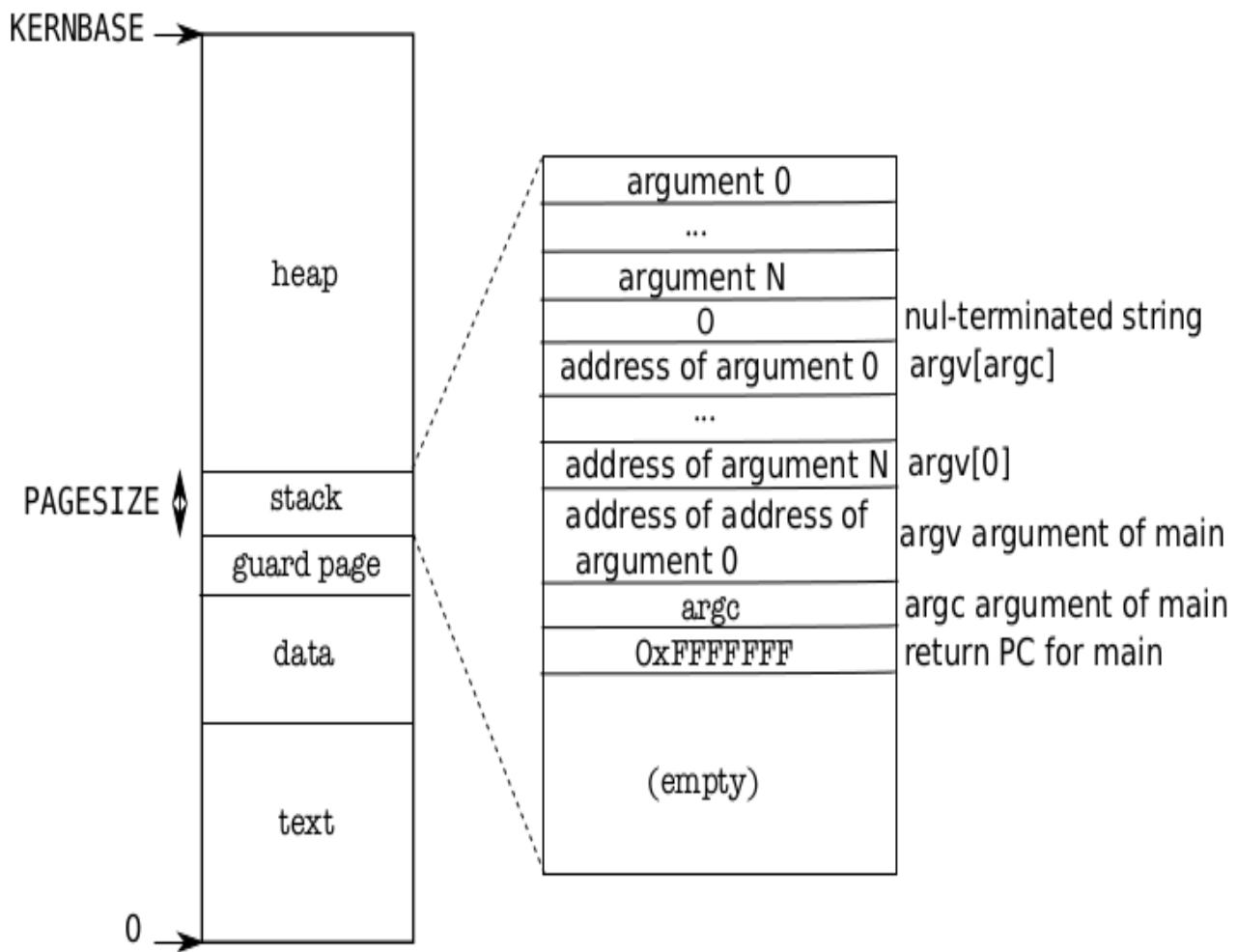
struct proc diagram: Very imp!



$sz = \text{ELF-code-} \rightarrow \text{memsz}$ (includes data, check "ld -N"
+ 2×4096 (for stack)

In use only when you are in kernel on a "trap" = interrupt/syscall. "tf" always used. trapret,forkret used during fork()

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

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
 - exception
- 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.
- **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**

Interrupt Descriptor Table (IDT) entries

```
// 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(meaning 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.
- Check that CPL in %cs is <= DPL, where DPL is the privilege level in the descriptor.
- 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 switchuvm()
- 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

```
# 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
```

alltraps:

- 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
 - `UatINTR()`
- 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,

Inter Process Communication

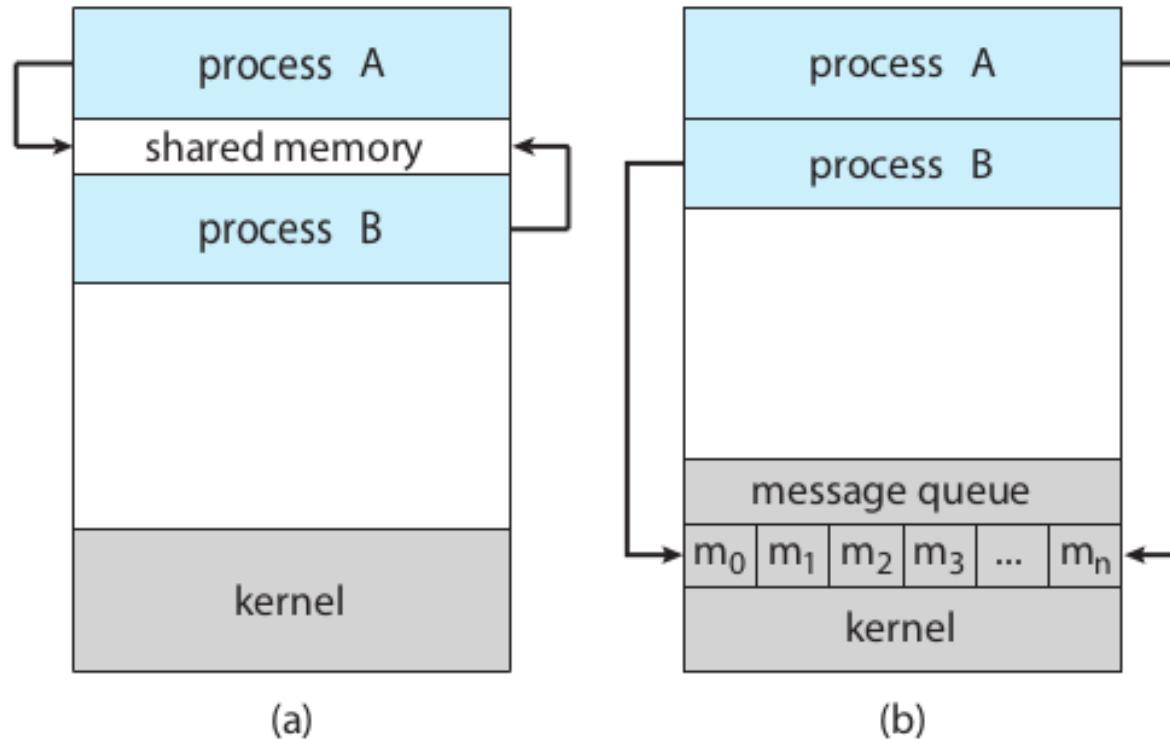
Revision of process related concepts

- **PCB, struct proc**
- **Process lifecycle – different states**
- **Memory layout**
- **Memory management**
- **Interrupts handling, system call handling, code from xv6**
- **Scheduler, code of scheduler in xv6**

IPC: Inter Process Communication

- Processes within a system may be independent or cooperating
- Cooperating process can affect or be affected by other processes, including sharing data
- Reasons for cooperating processes:
 - Information sharing, e.g. copy paste
 - Computation speedup, e.g. matrix multiplication
 - Modularity, e.g. chrome – separate process for display, separate for fetching data
 - Convenience,
- Cooperating processes need interprocess communication (IPC)
- Two models of IPC
 - Shared memory
 - Message passing

Shared Memory Vs Message Passing



Each requires OS to provide system calls for

- Creating the IPC mechanism
- To read/write using the IPC mechanism
- Delete the IPC mechanism

Note: processes communicating with each other with the help of OS!

Figure 3.11 Communications models. (a) Shared memory. (b) Message passing.

Example of co-operating processes: Producer Consumer Problem

- **Paradigm for cooperating processes, producer process produces information that is consumed by a consumer process**
 - unbounded-buffer places no practical limit on the size of the buffer
 - bounded-buffer assumes that there is a fixed buffer size

Example of co-operating processes: Producer Consumer Problem

- Shared data

```
#define BUFFER_SIZE 10
typedef struct {
    . . .
} item;
```

```
item buffer[BUFFER_SIZE];
int in = 0;
int out = 0;
```

- Can only use BUFFER_SIZE-1 elements

Example of co-operating processes: Producer Consumer Problem

- **Code of Producer**

```
while (true) {  
    /* Produce an item */  
  
    while (((in = (in + 1) % BUFFER SIZE count) == out)  
        ; /* do nothing -- no free buffers */  
    buffer[in] = item;  
    in = (in + 1) % BUFFER SIZE;  
}
```

Example of co-operating processes: Producer Consumer Problem

- **Code of Consumer**

```
while (true) {  
    while (in == out)  
        ; // do nothing -- nothing to consume  
  
    // remove an item from the buffer  
    item = buffer[out];  
    out = (out + 1) % BUFFER SIZE;  
    return item;  
}
```

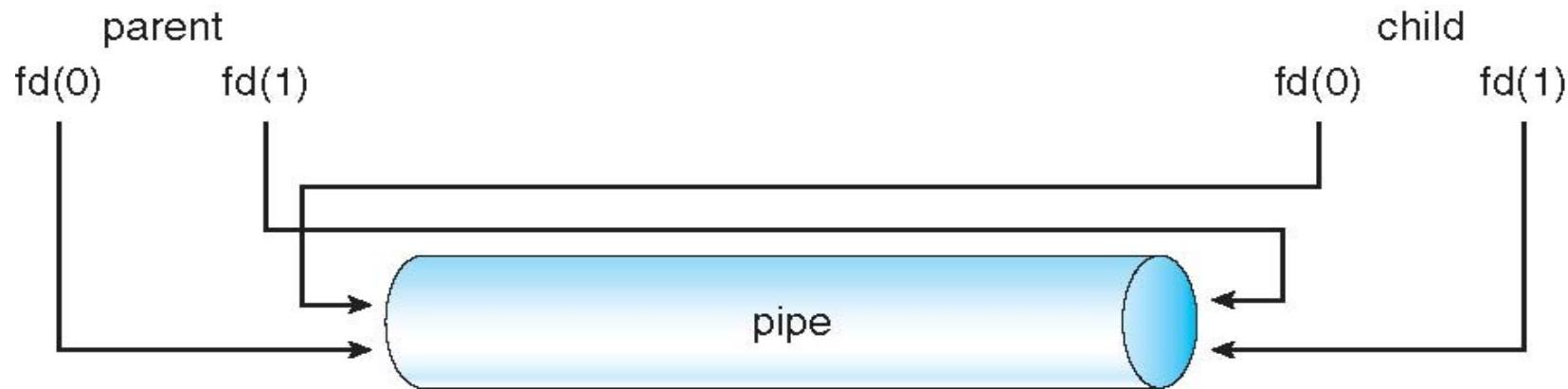
Pipes

Pipes for IPC

- **Two types**
 - Unnamed Pipes or ordinary pipes
 - Named Pipe

Ordinary pipes

- Ordinary Pipes allow communication in standard producer-consumer style
- Producer writes to one end (the write-end of the pipe)
- Consumer reads from the other end (the read-end of the pipe)
- Ordinary pipes are therefore unidirectional
- Requires a parent-child (or sibling, etc) kind of relationship between communicating processes



Named pipes

- Also called FIFO
- Processes can create a “file” that acts as pipe. Multiple processes can share the file to read/write as a FIFO
- Named Pipes are more powerful than ordinary pipes
- Communication is bidirectional
- No parent-child relationship is necessary between the communicating processes
- Several processes can use the named pipe for communication
- Provided on both UNIX and Windows systems
- Is not deleted automatically by OS

Named pipes

- **int mkfifo(const char *pathname, mode_t mode);**
- **Example**

Shared Memory

System V shared memory

- **Process first creates shared memory segment**
`segment id = shmget(IPC_PRIVATE, size, S_IRUSR | S_IWUSR);`
- **Process wanting access to that shared memory must attach to it**
`shared_memory = (char *) shmat(id, NULL, 0);`
- **Now the process could write to the shared memory**
`sprintf(shared_memory, "Writing to shared memory");`
- **When done, a process can detach the shared memory from its address space**
`shmdt(shared_memory);`

Example of Shared memory

POSIX Shared Memory

- **What is POSIX?**
 - Portable Operating System Interface (**POSIX**)
 - family of standards
 - specified by the IEEE Computer Society
 - for maintaining compatibility between operating systems.
 - API (system calls), shells, utility commands for compatibility among UNIXes and variants

POSIX Shared Memory

- **shm_open**
- **ftruncate**
- **Mmap**
- **See the example in Textbook**

Message passing

Message Passing

- **Message system – processes communicate with each other using send(), receive() like syscalls given by OS**
- **IPC facility provides two operations:**
 - send(message) – message size fixed or variable
 - Receive(message)
- **If P and Q wish to communicate, they need to:**
 - establish a communication link between them
 - exchange messages via send/receive
- **Communication link can be implemented in a variety of ways**

Message Passing using “Naming”

- **Pass a message by “naming” the receiver**
 - **A) Direct communication with receiver**
 - **Receiver is identified by sender directly using it's name**
 - **B) Indirect communication with receiver**
 - **Receiver is identified by sender in-directly using it's 'location of receipt'**

Message passing using direct communication

- **Processes must name each other explicitly:**
 - send (P, message) – send a message to process P
 - receive(Q, message) – receive a message from process Q
- **Properties of communication link**
 - Links are established automatically
 - A link is associated with exactly one pair of communicating processes
 - Between each pair there exists exactly one link
 - The link may be unidirectional, but is usually bi-directional

Message passing using IN-direct communication

- **Messages are directed and received from mailboxes (also referred to as ports)**
 - Each mailbox has a unique id
 - Processes can communicate only if they share a mailbox
- **Properties of communication link**
 - Link established only if processes share a common mailbox
 - A link may be associated with many processes
 - Each pair of processes may share several communication links
 - Link may be unidirectional or bi-directional

Message passing using IN-direct communication

- **Operations**
 - create a new mailbox
 - send and receive messages through mailbox
 - destroy a mailbox
- **Primitives are defined as:**
 - `send(A, message)` – send a message to mailbox A
 - `receive(A, message)` – receive a message from mailbox A

Message passing using IN-direct communication

- **Mailbox sharing**
 - P1, P2, and P3 share mailbox A
 - P1, sends; P2 and P3 receive
 - Who gets the message?
- **Solutions**
 - Allow a link to be associated with at most two processes
 - Allow only one process at a time to execute a receive operation
 - Allow the system to select arbitrarily the receiver. Sender is notified who the receiver was.

Message Passing implementation: Synchronization issues

- Message passing may be either blocking or non-blocking
- Blocking is considered synchronous
 - Blocking send has the sender block until the message is received
 - Blocking receive has the receiver block until a message is available
- Non-blocking is considered asynchronous
 - Non-blocking send has the sender send the message and continue
 - Non-blocking receive has the receiver receive a valid message or null

Producer consumer using blocking send and receive

Producer

```
message next_produced;  
while (true) {  
/* produce an item in  
next_produced */  
send(next_produced);  
}
```

Consumer

```
message  
next_consumed;  
while (true) {  
receive(next_consumed);  
}
```

Message Passing implementation: choice of Buffering

- Queue of messages attached to the link; implemented in one of three ways
 1. Zero capacity – 0 messages
 - Sender must wait for receiver (rendezvous)
 2. Bounded capacity – finite length of n messages
 - Sender must wait if link full
 3. Unbounded capacity – infinite length
 - Sender never waits

Scheduler

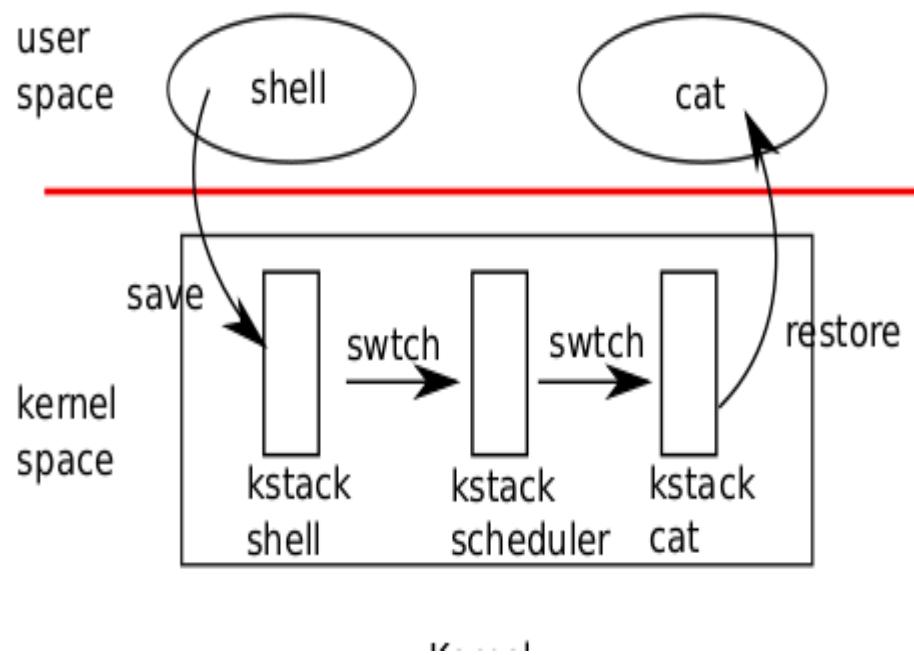
Scheduler – in most simple terms

- **Selects a process to execute and passes control to it !**
 - The process is chosen out of “READY” state processes
 - Saving of context of “earlier” process and loading of context of “next” process needs to happen
- **Questions**
 - What are the different scenarios in which a scheduler called ?
 - What are the intricacies of “passing control”
 - What is “context” ?

Steps in scheduling scheduling

- Suppose you want to switch from P1 to P2 on a timer interrupt
- P1 was doing
 - F() { i++; j++; }
- P2 was doing
 - G() { x--; y++; }
- P1 will experience a timer interrupt, switch to kernel (scheduler) and scheduler will schedule P2

4 stacks need to change!



- **User stack of process -> kernel stack of process**
 - Switch to kernel stack
 - The normal sequence on any interrupt !
- **Kernel stack of process -> kernel stack of scheduler**
 - Why?
- **Kernel stack of scheduler -> kernel stack of new process . Why?**
- **Kernel stack of new process -> user stack of new process**

scheduler()

- **Disable interrupts**
- **Find a RUNNABLE process. Simple round-robin!**
- **c->proc = p**
- **switchuvm(p) : Save TSS of scheduler's stack and make CR3 to point to new process pagedir**
- **p->state = RUNNING**
- **swtch(&(c->scheduler), p->context)**

swtch

swtch:

```
movl 4(%esp), %eax
```

```
movl 8(%esp), %edx
```

```
# Save old callee-saved registers
```

```
pushl %ebp
```

```
pushl %ebx
```

```
pushl %esi
```

```
pushl %edi
```

```
# Switch stacks
```

```
movl %esp, (%eax)
```

```
movl %edx, %esp
```

```
# Load new callee-saved registers
```

```
popl %edi
```

```
popl %esi
```

```
popl %ebx
```

```
popl %ebp
```

```
ret
```

scheduler()

- **swtch(&(c->scheduler), p->context)**
- **Note that when scheduler() was called, when P1 was running**
- **After call to swtch() shown above**
 - The call does NOT return!
 - The new process P2 given by ‘p’ starts running !
 - Let’s review swtch() again

swtch(old, new)

- The magic function in swtch.S
- Saves callee-save registers of old context
- Switches esp to new-context's stack
- Pop callee-save registers from new context

ret

- where? in the case of first process – returns to forkret() because stack was setup like that !
- in case of other processes, return where?
 - Return address given on kernel stack. But what's that?
 - The EIP in p->context
 - When was EIP set in p->context ?

scheduler()

- **Called from?**
 - `mpmain()`
 - **No where else!**
- **`sched()` is another scheduler function !**
 - **Who calls `sched()` ?**
 - `exit()` - a process exiting calls `sched ()`
 - `yield()` - a process interrupted by timer calls `yield()`
 - `sleep()` - a process going to wait calls `sleep()`

sched()

```
void
sched(void)
{
    int intena;
    struct proc *p = myproc();
    if(!holding(&ptable.lock))
        panic("sched ptable.lock");
    if(mycpu()->ncli != 1)
        panic("sched locks");
    if(p->state == RUNNING)
        panic("sched running");
    if(readeflags()&FL_IF)
        panic("sched interruptible");
    intena = mycpu()->intena;
    swtch(&p->context, mycpu()->scheduler);
/*A*/ mycpu()->intena = intena;
}
```

- get current process
- Error checking code (ignore as of now)
- get interrupt enabled status on current CPU (ignore as of now)
- call to swtch
 - Note the arguments' order
 - p->context first, mycpu()->scheduler second
- swtch() is a function call
 - pushes address of /*A*/ on stack of current process p
 - switches stack to mycpu()->scheduler. Then pops EIP from that stack and jumps there.
 - when was mycpu()->scheduler set? Ans: during scheduler()

sched() and scheduler()

```
sched() {  
    ...  
    swtch(&p->context, mycpu()->scheduler); /* X */  
}  
}
```

```
scheduler(void) {  
    ...  
    swtch(&(c->scheduler), p->context); /* Y */  
}
```

- **scheduler() saves context in c->scheduler, sched() saves context in p->context**
- **after swtch() call in sched(), the control jumps to Y in scheduler**
 - **Switch from process stack to scheduler's stack**
- **after swtch() call in scheduler(), the control jumps to X in sched()**
 - **Switch from scheduler's stack to new process's stack**
- **Set of co-operating functions**

sched() and scheduler() as co-routines

- **In sched()**

```
swtch(&p->context, mycpu()->scheduler);
```

- **In scheduler()**

```
swtch(&(c->scheduler), p->context);
```

- **These two keep switching between processes**
- **These two functions work together to achieve scheduling**
- **Using asynchronous jumps**
- **Hence they are co-routines**

To summarize

- On a timer interrupt during P1
 - trap() is called. Stack has changed from P1's user stack to P1's kernel stack
 - trap()->yield()
 - yield()->sched()
 - sched() -> swtch(&p->context, c->scheduler())
 - Stack changes to scheduler's kernel stack.
 - Switches to location “Y” in scheduler().
- Now the loop in scheduler()
 - calls switchkvm()
 - Then continues to find next process (P2) to run
 - Then calls switchuvvm(p): changing the page table to the P2's page tables
 - then calls swtch(&c->scheduler, p2->context)
 - Stack changes to P2's kernel stack.
 - P2 runs the last instruction it was was in ! Where was it?
 - mycpu()->intena = intena; in sched()
 - Then returns to the one who called sched() i.e. exit/sleep, etc
 - Finally returns from it's own “TRAP” handler and returns to P2's user stack and user code

Memory Management – Continued

More on Linking, Loading, Paging

Review of last class

- MMU : Hardware features for MM
- OS: Sets up MMU for a process, then schedules process
- Compiler : Generates object code for a particular OS + MMU architecture
- MMU: Detects memory violations and raises interrupt --> Effectively passing control to OS

More on Linking and Loading

- **Static Linking:** All object code combined at link time and a big object code file is created
- **Static Loading:** All the code is loaded in memory at the time of `exec()`
- **Problem:** Big executable files, need to load functions even if they do not execute
- **Solution:** Dynamic Linking and Dynamic Loading

Dynamic Linking

- **Linker is normally invoked as a part of compilation process**
 - **Links**
 - function code to function calls
 - references to global variables with “extern” declarations
- **Dynamic Linker**
 - **Does not combine function code with the object code file**
 - **Instead introduces a “stub” code that is indirect reference to actual code**
 - **At the time of “loading” (or executing!) the program in memory, the “link-loader” (part of OS!) will pick up the relevant code from the library machine code file (e.g. libc.so.6)**

Dynamic Linking on Linux

```
#include <stdio.h>

int main() {
    int a, b;
    scanf("%d%d", &a, &b);
    printf("%d %d\n", a, b);
    return 0;
}
```

PLT: Procedure Linkage Table

used to call external procedures/functions whose address is to be resolved by the dynamic linker at run time.

Output of objdump -x -D

Disassembly of section .text:

```
0000000000001189 <main>:
```

```
11d4:    callq  1080 <printf@plt>
```

Disassembly of section .plt.got:

```
0000000000001080 <printf@plt>:
```

```
1080:    endbr64
```

```
1084:    bnd jmpq *0x2f3d(%rip)      # 3fc8
<printf@GLIBC_2.2.5>
```

```
108b:    nopl  0x0(%rax,%rax,1)
```

Dynamic Loading

- **Loader**
 - Loads the program in memory
 - Part of exec() code
 - Needs to understand the format of the executable file (e.g. the ELF format)
- **Dynamic Loading**
 - Load a part from the ELF file only if needed during execution
 - Delayed loading
 - Needs a more sophisticated memory management by operating system – to be seen during this series of lectures

Dynamic Linking, Loading

- Dynamic linking necessarily demands an advanced type of loader that understands dynamic linking
 - Hence called ‘link-loader’
 - Static or dynamic loading is still a choice
- Question: which of the MMU options will allow for which type of linking, loading ?

Continuous memory management

What is Continuous memory management?

- Entire process is hosted as one continuous chunk in RAM
- Memory is typically divided into two partitions
 - One for OS and other for processes
 - OS most typically located in “high memory” addresses, because interrupt vectors map to that location (Linux, Windows) !

Hardware support needed: base + limit (or relocation + limit)

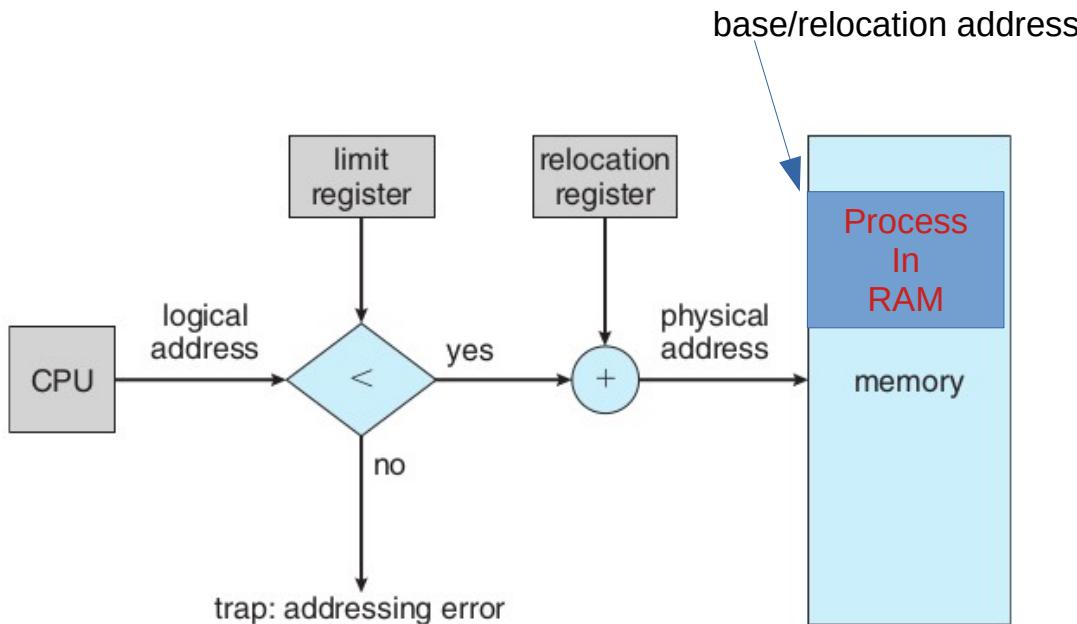


Figure 9.6 Hardware support for relocation and limit registers.

Problems faced by OS

- Find a continuous chunk for the process being forked
- Different processes are of different sizes
 - Allocate a size parameter in the PCB
- After a process is over – free the memory occupied by it
- Maintain a list of free areas, and occupied areas
 - Can be done using an array, or linked list

Variable partition scheme

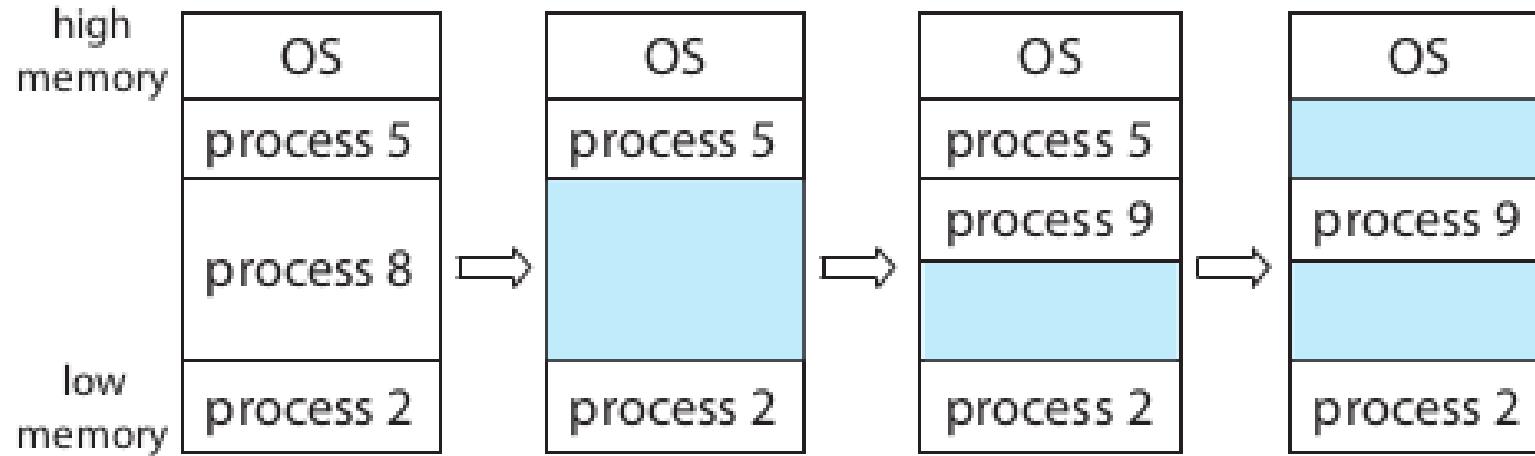


Figure 9.7 Variable partition.

Problem: how to find a “hole” to fit in new process

- Suppose there are 3 free memory regions of sizes 30k, 40k, 20k
- The newly created process (during `fork()` + `exec()`) needs 15k
- Which region to allocate to it ?

Strategies for finding a free chunk

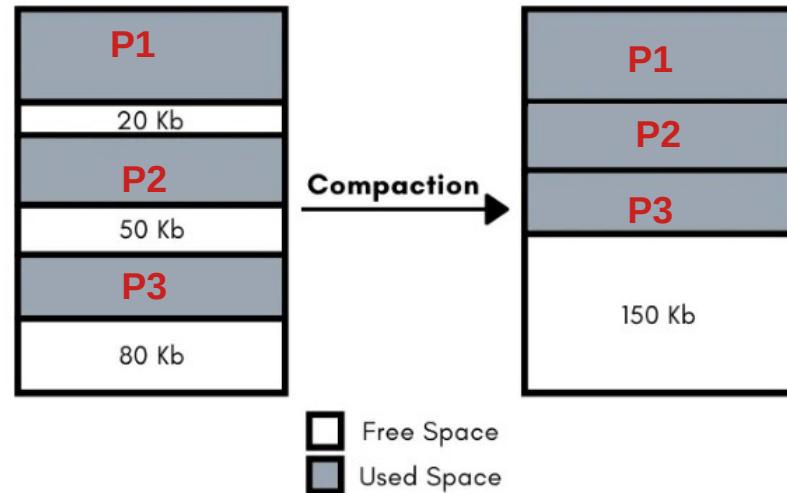
- 6k, 17k, 16k, 40k holes . Need 15k.
- **Best fit:** Find the smallest hole, larger than process. **Ans: 16k**
- **Worst fit:** Find the largest hole. **Ans: 40k**
- **First fit:** Find the “first” hole larger than the process. **Ans: 17k**

Problem : External fragmentation

- Free chunks: 30k, 40k, 20k
- The newly created process (during `fork()` + `exec()`) needs 50k
- Total free memory: $30+40+20 = 90\text{k}$
 - But can't allocate 50k !

Solution to external fragmentation

- Compaction !
- OS moves the process chunks in memory to make available continuous memory region
 - Then it must update the memory management information in PCB (e.g. base of the process) of each process
- Time consuming
- Possible only if the relocation+limit scheme of MMU is available



Another solution to external fragmentation: Fixed size partitions

- Fixed partition scheme
- Memory is divided by OS into chunks of equal size: e.g., say, 50k
 - If total 1M memory, then 20 such chunks
- Allocate one or more chunks to a process, such that the total size is \geq the size of the process
 - E.g. if request is 50k, allocate 1 chunk
 - If request is 40k, still allocate 1 chunk
 - If request is 60k, then allocate 2 chunks
- Leads to internal fragmentation
 - space wasted in the case of 40k or 60k requests above



Fixed partition scheme

- OS needs to keep track of
 - Which partition is free and which is used by which process
 - Free partitions can simply be tracked using a bitmap or a list of numbers
 - Each process's PCB will contain list of partitions allocated to it

Solution to internal fragmentation

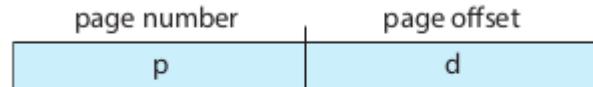
- Reduce the size of the fixed sized partition
- How small then ?
 - Smaller partitions mean more overhead for the operating system in allocating deallocating

Paging

An extended version of fixed size partitions

- Partition = page
 - Process = logically continuous sequence of bytes, divided in ‘page’ sizes
 - Memory divided into equally sized page ‘frames’
- Important distinction
 - Process need not be continuous in RAM
 - Different page sized chunks of process can go in any page frame
 - Page table to map pages into frames

Logical address seen as



Paging hardware

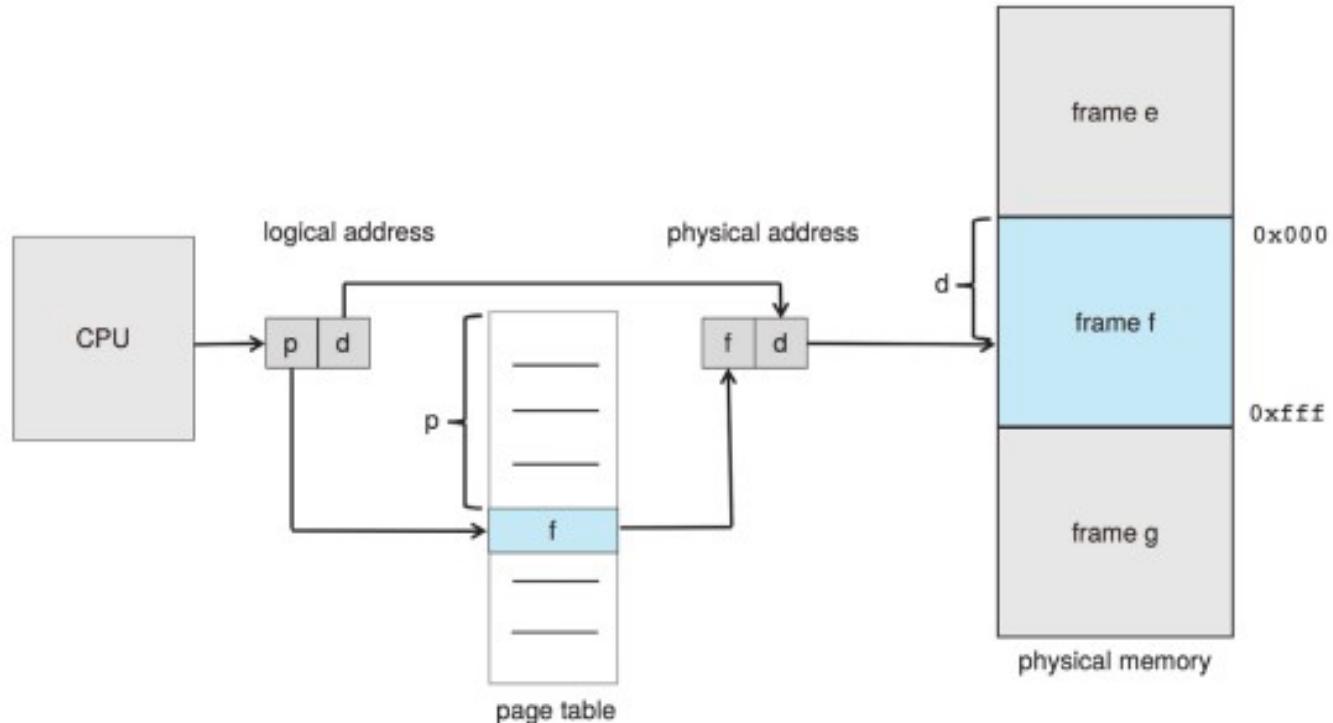


Figure 9.8 Paging hardware.

MMU's job

To translate a logical address generated by the CPU to a physical address:

1. Extract the page number p and use it as an index into the page table.

(Page table location is stored in a hardware register

Also stored in PCB of the process, so that it can be used to load the hardware register on a context switch)

2. Extract the corresponding frame number f from the page table.

3. Replace the page number p in the logical address with the frame number f .

Job of OS

- Allocate a page table for the process, at time of fork()/exec()
 - Allocate frames to process
 - Fill in page table entries
- In PCB of each process, maintain
 - Page table location (address)
 - List of pages frames allocated to this process
- During context switch of the process, load the PTBR using the PCB

Job of OS

- **Maintain a list of all page frames**
 - Allocated frames
 - Free Frames (called frame table)
 - Can be done using simple linked list
 - Innovative data structures can also be used to maintain free and allocated frames list (e.g. xv6 code)
 -

free-frame list

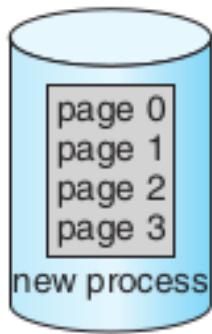
14

13

18

20

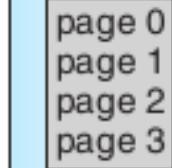
15



(a)

free-frame list

15



new-process page table

(b)

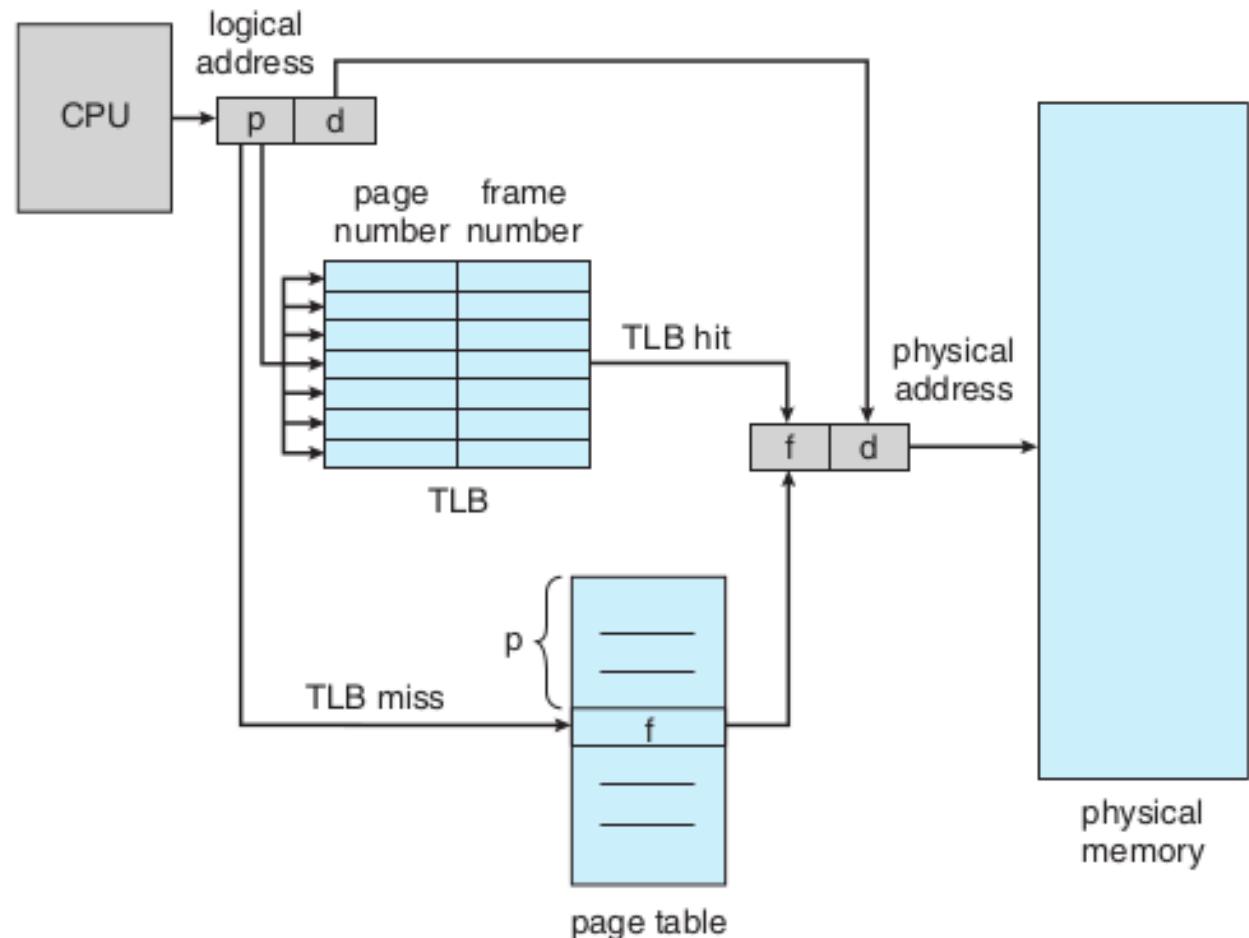
Figure 9.11 Free frames (a) before allocation and (b) after allocation.

Disadvantage of Paging

- Each memory access results in two memory accesses!
 - One for page table, and one for the actual memory location !
 - Done as part of execution of instruction in hardware (not by OS!)
 - Slow down by 50%

Speeding up paging

- Translation Lookaside Buffer (TLB)
- Part of CPU hardware
- A cache of Page table entries
- Searched in parallel for a page number



Speedup due to TLB

- Hit ratio
- Effective memory access time
 - = Hit ratio * 1 memory access time + miss ratio * 2 memory access time
- Example: memory access time 10ns, hit ratio = 0.8, then
 - effective access time = $0.80 \times 10 + 0.20 \times 20$
 - = 12 nanoseconds

Memory protection with paging

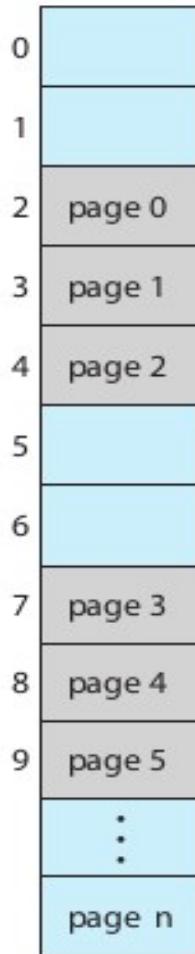
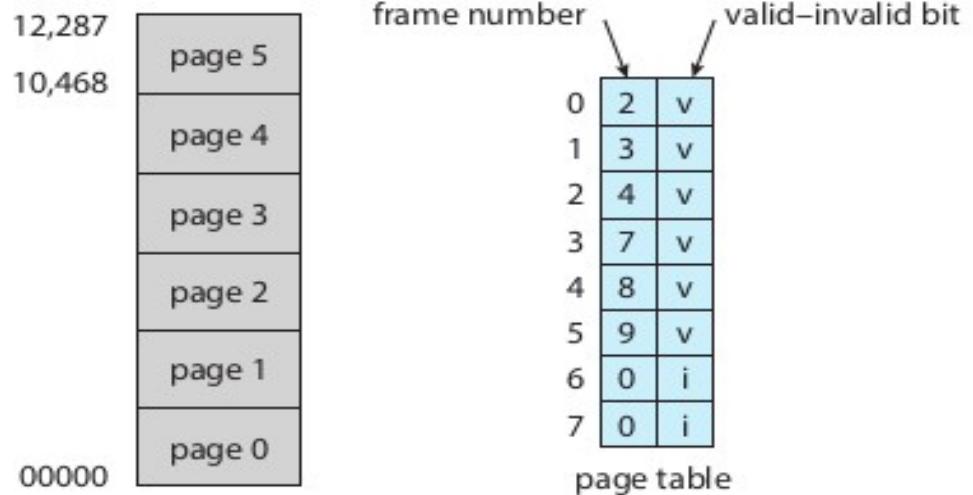


Figure 9.13 Valid (v) or invalid (i) bit in a page table.

X86 PDE and PTE

31

| Page table physical page number | A V L | G S T | P A D | 0 A D | A C D | C W T | W U T | U W P | 1 0 0 |
|---------------------------------|-------------|-------------|-------------|-------------|-------------|-------------|-------------|-------------|-------------|
|---------------------------------|-------------|-------------|-------------|-------------|-------------|-------------|-------------|-------------|-------------|

PDE

31

| Physical page number | A V L | G A T | P A D | 0 D A | A C D | C W T | W U P | U W P | 1 0 0 |
|----------------------|-------------|-------------|-------------|-------------|-------------|-------------|-------------|-------------|-------------|
|----------------------|-------------|-------------|-------------|-------------|-------------|-------------|-------------|-------------|-------------|

PTE

P Present

W Writable

U User

WT 1=Write-through, 0=Write-back

CD Cache disabled

A Accessed

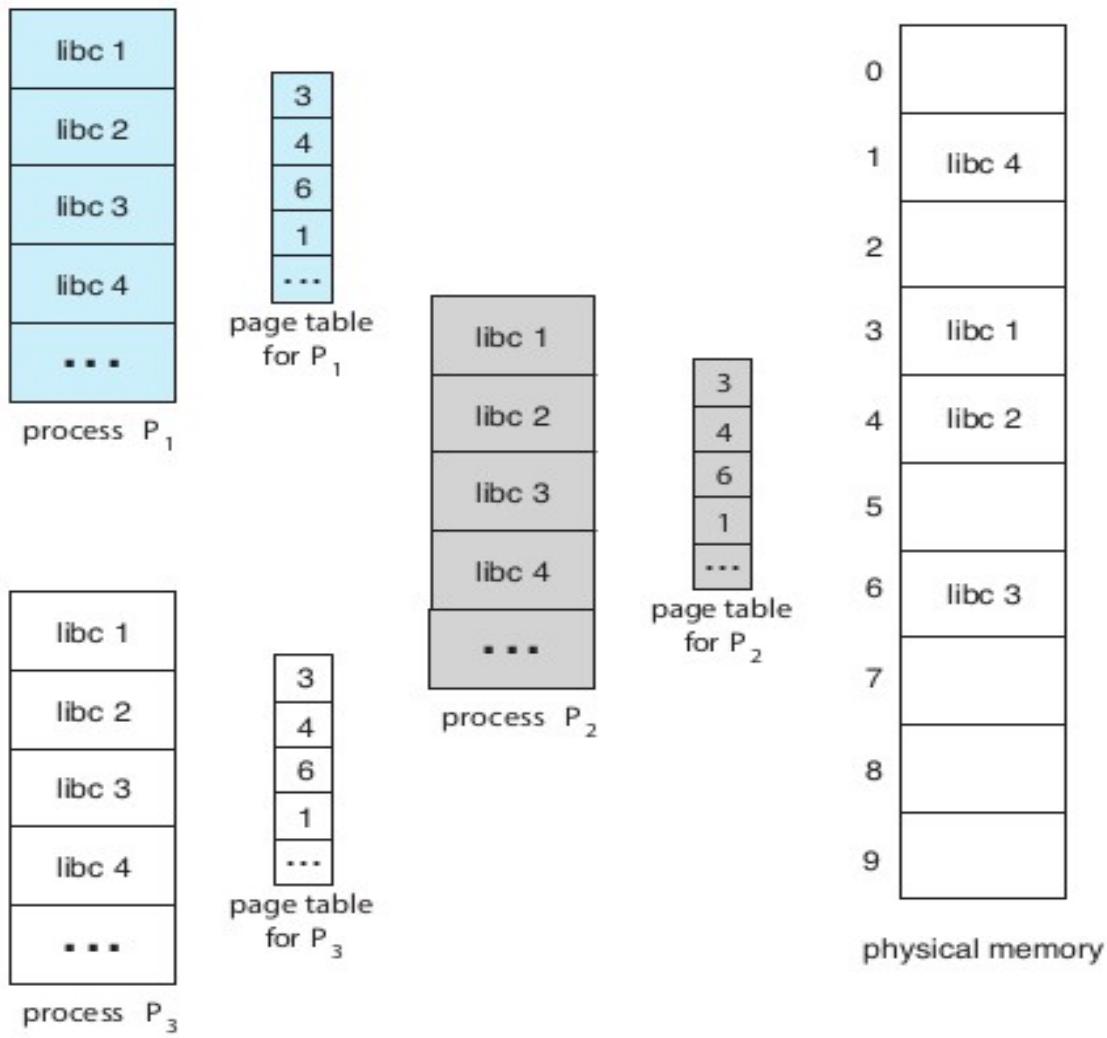
D Dirty

PS Page size (0=4KB, 1=4MB)

PAT Page table attribute index

G Global page

AVL Available for system use



Shared pages (e.g. library) with paging

Figure 9.14 Sharing of standard C library in a paging environment.

Paging: problem of large PT

- 64 bit address
- Suppose 20 bit offset
 - That means $2^{20} = 1 \text{ MB}$ pages
 - 44 bit page number: 2^{44} that is trillion sized page table!
 - Can't have that big continuous page table!

Paging: problem of large PT

- 32 bit address
- Suppose 12 bit offset
 - That means $2^{12} = 4 \text{ KB pages}$
 - 20 bit page number: 2^{20} that is a million entries
 - Can't always have that big continuous page table as well, for each process!

Hierarchical paging

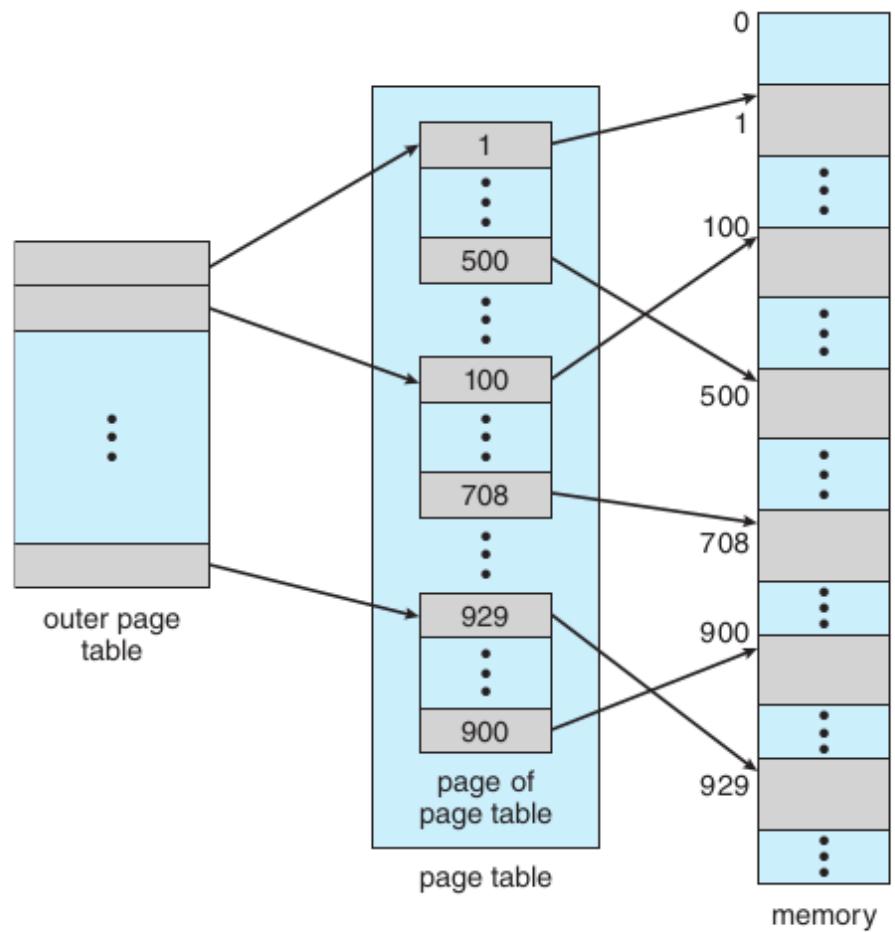
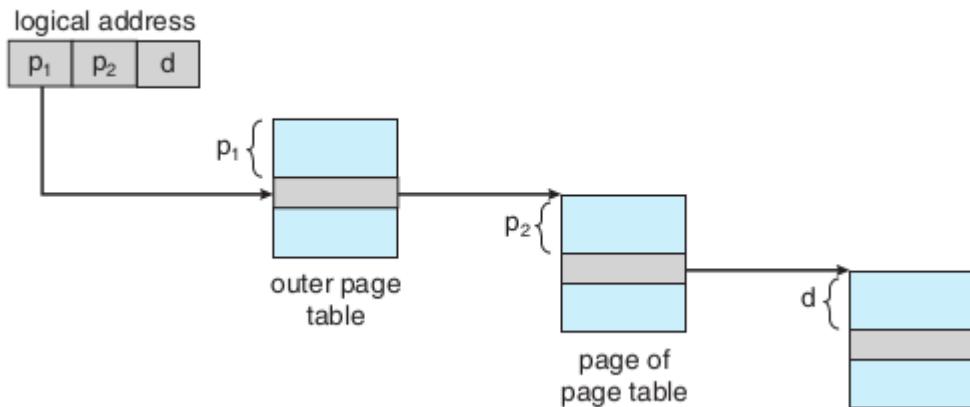


Figure 9.15 A two-level page-table scheme.

| outer page | inner page | offset |
|------------|------------|--------|
| p_1 | 10 | 12 |

More hierarchy

| 2nd outer page | outer page | inner page | offset |
|----------------|------------|------------|--------|
| p_1 | p_2 | p_3 | d |
| 32 | 10 | 10 | 12 |

Problems with hierarchical paging

- More number of memory accesses with each level !
 - Too slow !
- OS data structures also needed in that proportion

Hashed page table

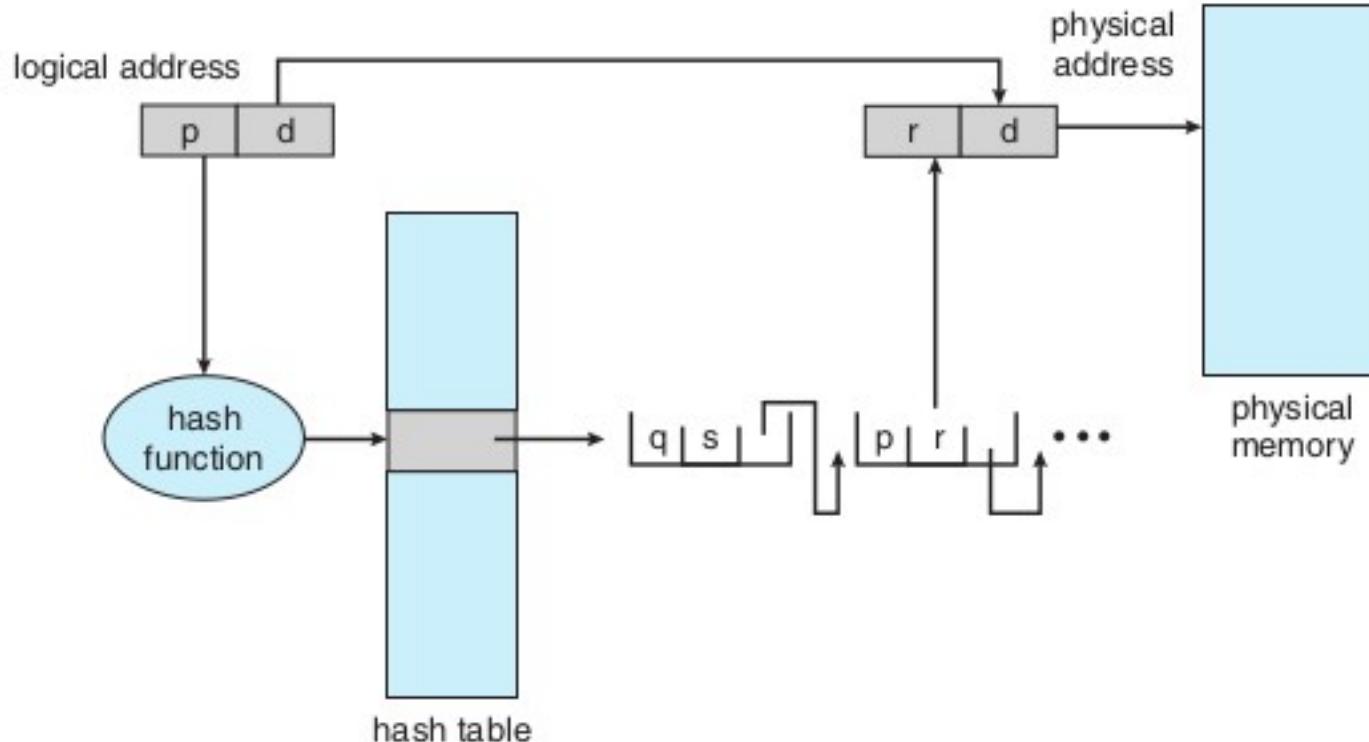


Figure 9.17 Hashed page table.

Inverted page table

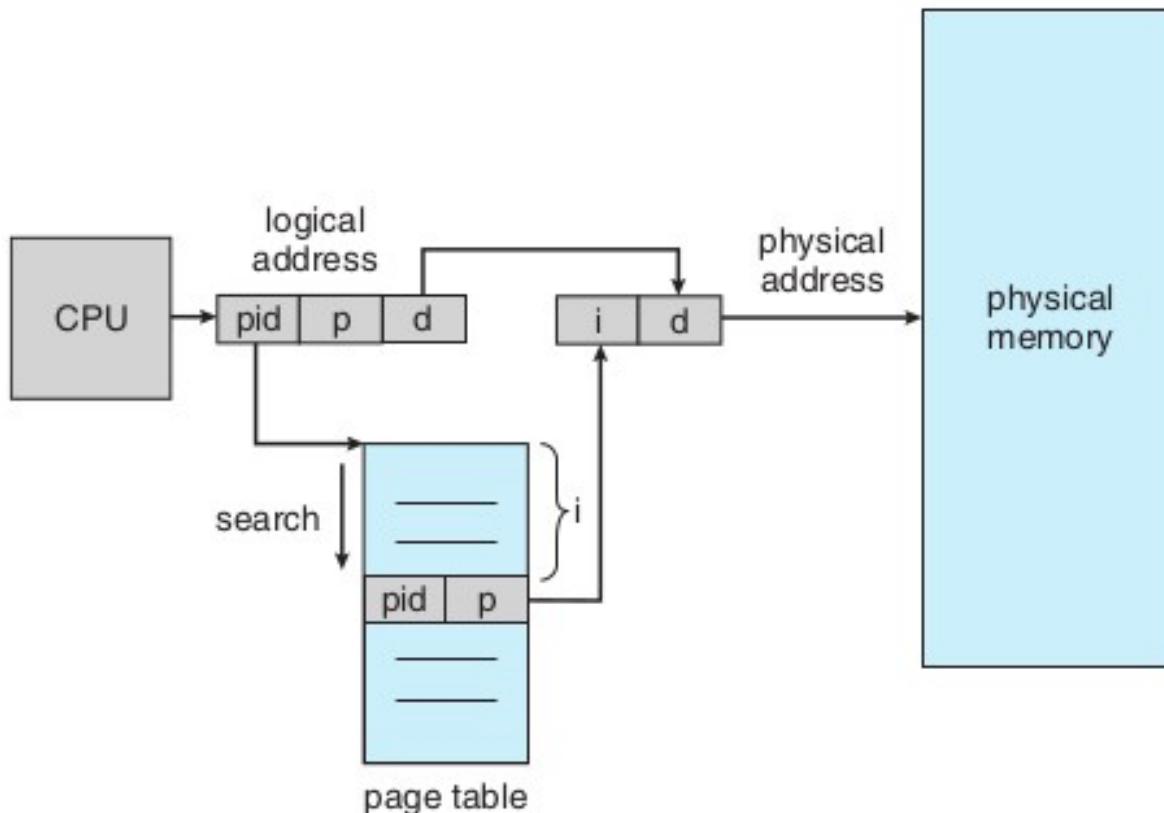


Figure 9.18 Inverted page table.

Normal page table – one per process --> Too much memory consumed

Inverted page table : global table – only one
Needs to store PID in the table entry

Examples of systems using inverted page tables include the 64-bit Ultra SPARC and Power PC

virtual address consists of a triple:
<process-id, page-number,
offset>

Case Study: Oracle SPARC Solaris

- 64 bit SPARC processor , 64 bit Solaris OS
- Uses Hashed page tables
 - one for the kernel and one for all user processes.
 - Each hash-table entry : base + span (#pages)
 - Reduces number of entries required

Case Study: Oracle SPARC Solaris

- Caching levels: TLB (on CPU), TSB(in Memory), Page Tables (in Memory)
 - CPU implements a TLB that holds translation table entries (TTE s) for fast hardware lookups.
 - A cache of these TTEs resides in a in-memory translation storage buffer (TSB), which includes an entry per recently accessed page
 - When a virtual address reference occurs, the hardware searches the TLB for a translation.
 - If none is found, the hardware walks through the in memory TSB looking for the TTE that corresponds to the virtual address that caused the lookup

Case Study: Oracle SPARC Solaris

- If a match is found in the TSB , the CPU copies the TSB entry into the TLB , and the memory translation completes.
- If no match is found in the TSB , the kernel is interrupted to search the hash table.
- The kernel then creates a TTE from the appropriate hash table and stores it in the TSB for automatic loading into the TLB by the CPU memory-management unit.
- Finally, the interrupt handler returns control to the MMU , which completes the address translation and retrieves the requested byte or word from main memory.

Swapping

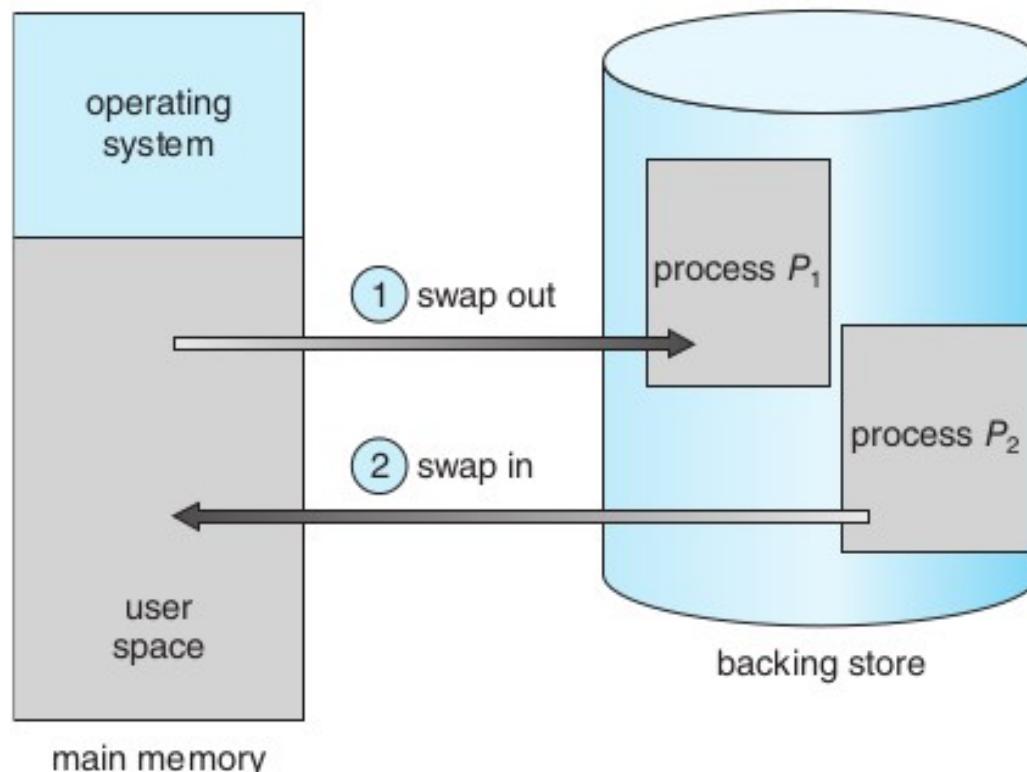


Figure 9.19 Standard swapping of two processes using a disk as a backing store.

Swapping

- Standard swapping
 - Entire process swapped in or swapped out
 - With continuous memory management
- Swapping with paging
 - Some pages are “paged out” and some “paged in”
 - Term “paging” refers to paging with swapping now

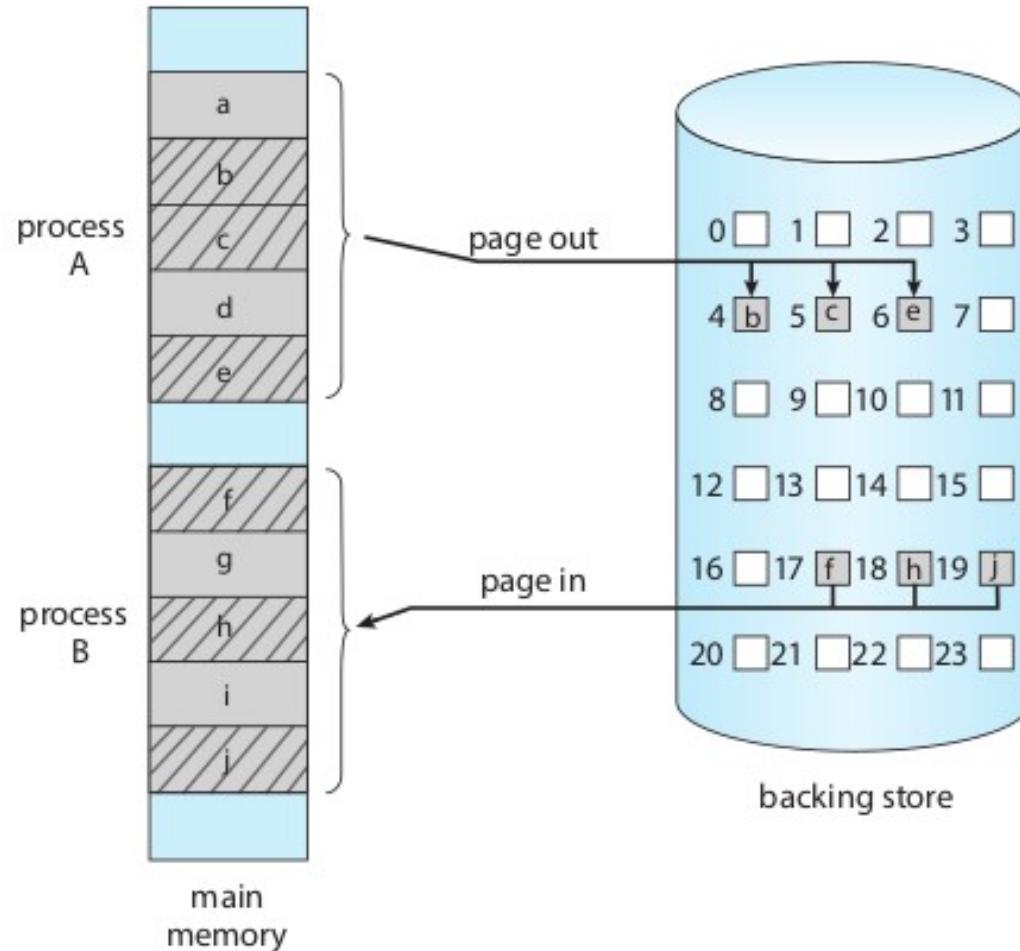


Figure 9.20 Swapping with paging.

Words of caution about ‘paging’

- Not as simple as it sounds when it comes to implementation
 - Writing OS code for this is challenging

Virtual Memory

Introduction

- Virtual memory != Virtual address

Virtual address is address issued by CPU's execution unit, later converted by MMU to physical address

Virtual memory is a memory management technique employed by OS (with hardware support, of course)

Unused parts of program

```
int a[4096][4096]                                All parts of array a[] not accessed
int f(int m[][4096]) {
    int i, j;                                     Function f() may not be called
    for(i = 0; i < 1024; i++)
        m[0][i] = 200;
}
int main() {
    int i, j;
    for(i = 0; i < 1024; i++)
        a[1][i] = 200;
    if(random() == 10)
        f(a);
}
```

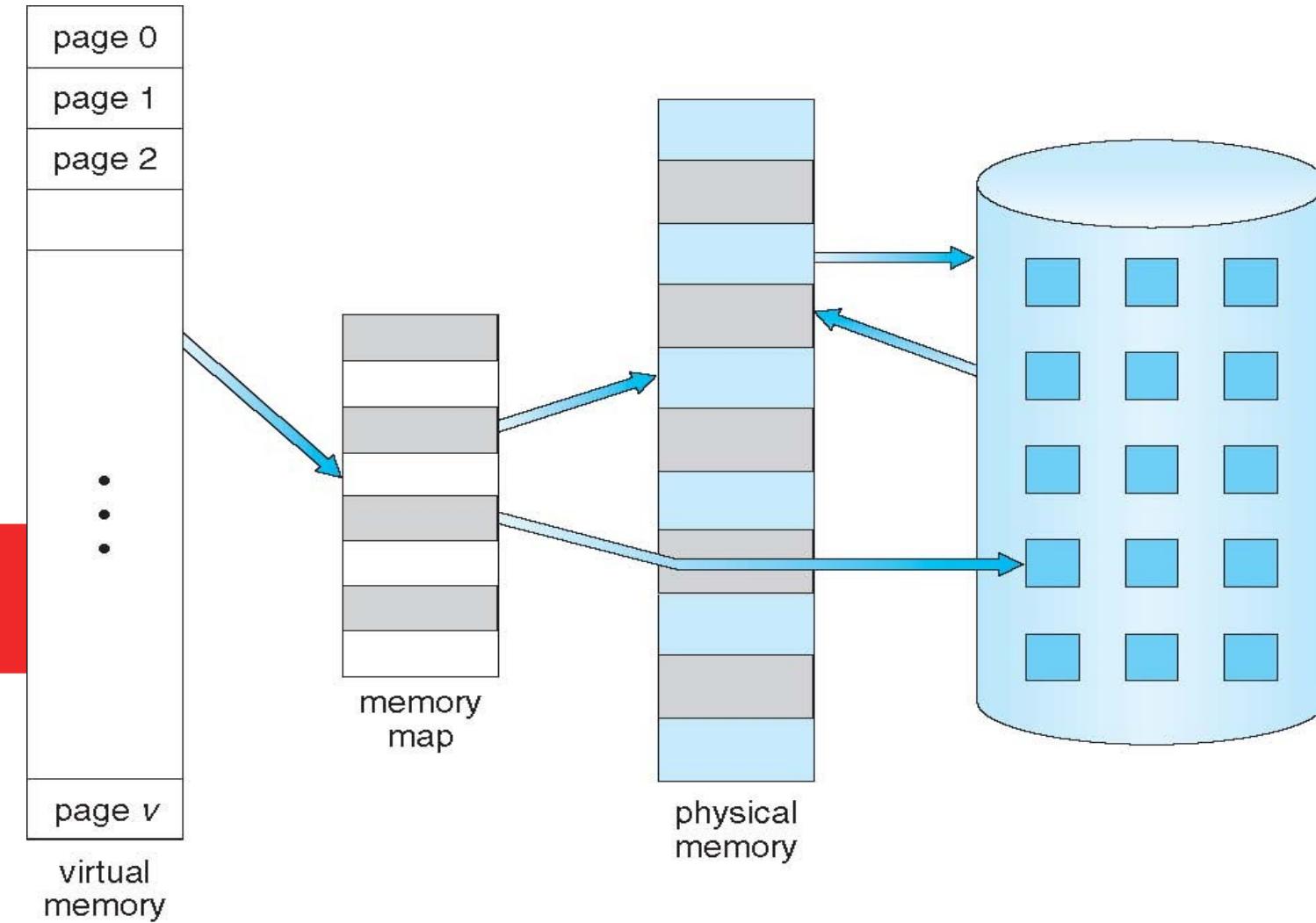
Some problems with schemes discussed so far

- Code needs to be in memory to execute, But entire program rarely used
 - Error code, unusual routines, large data structures are rarely used
- So, entire program code, data not needed at same time
- So, consider ability to execute partially-loaded program
 - One Program no longer constrained by limits of physical memory
 - One Program and collection of programs could be larger than physical memory

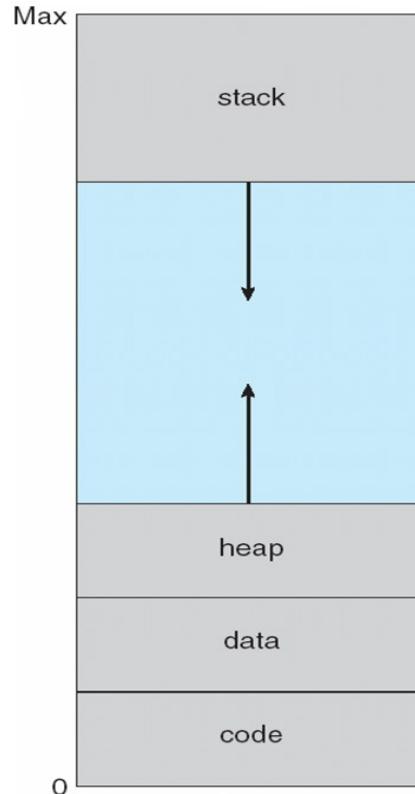
What is virtual memory?

- Virtual memory – separation of user logical memory from physical memory
 - Only part of the program needs to be in memory for execution
 - Logical address space can therefore be much larger than physical address space
 - Allows address spaces to be shared by several processes
 - Allows for more efficient process creation
 - More programs running concurrently
 - Less I/O needed to load or swap processes
- Virtual memory can be implemented via:
 - Demand paging
 - Demand segmentation

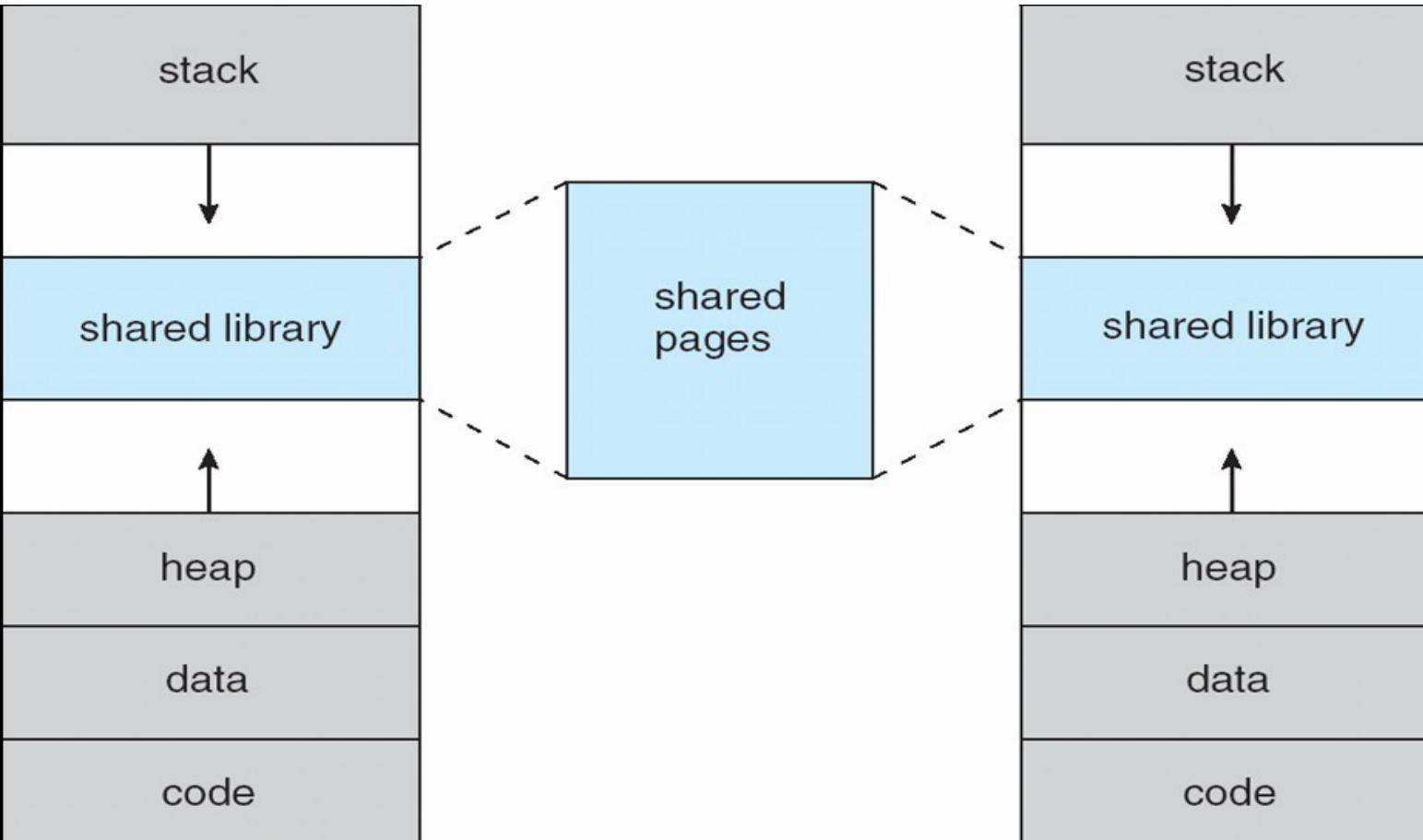
Virtual Memory
Larger
Than
Physical
Memory



Virtual Address space



Enables **sparse** address spaces with holes left for growth, dynamically linked libraries, etc



Shared pages Using Virtual Memory

System libraries shared via mapping into virtual address space

Shared memory by mapping same page-frames into page tables of involved processes

Pages can be shared during `fork()`, speeding process creation (more later)



Demand Paging

Demand Paging

- Load a “page” to memory when it’s needed (on demand)

Less I/O needed, no unnecessary I/O

Less memory needed

Faster response

More users

-

Demand Paging

- Options:
 - Load entire process at load time : achieves little
 - Load some pages at load time: good
 - Load no pages at load time: pure demand paging

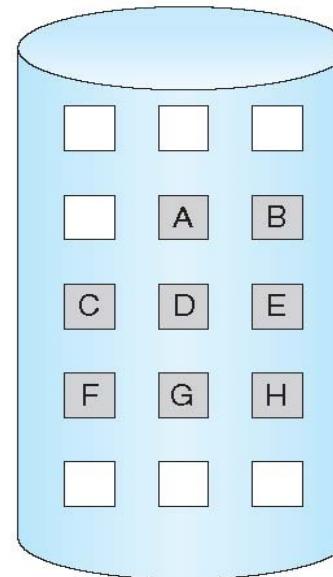
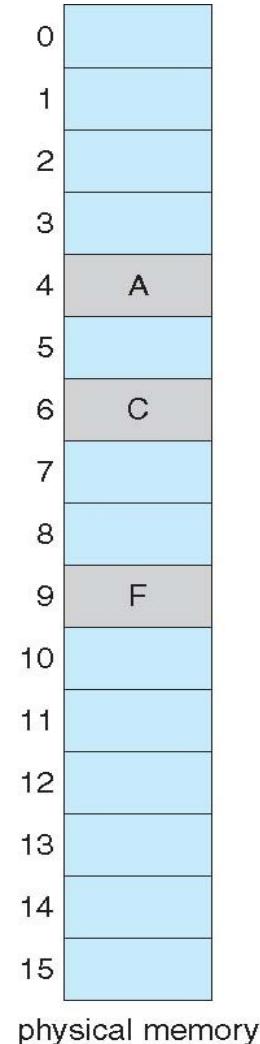
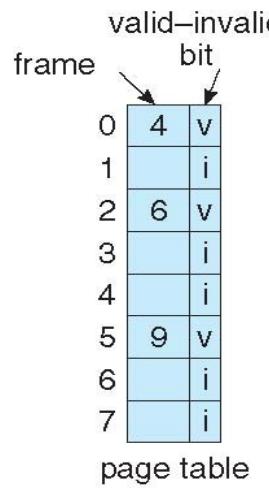
New meaning for valid/invalid bits in page table

| Frame # | valid-invalid bit |
|---------|-------------------|
| | v |
| | v |
| | v |
| | v |
| | i |
| | i |
| | i |

- With each page table entry a valid-invalid bit is associated
 - v: in-memory – memory resident
 - i : not-in-memory or illegal
- During address translation, if valid-invalid bit in page table entry is I : **raises trap called page fault**

| | |
|---|---|
| 0 | A |
| 1 | B |
| 2 | C |
| 3 | D |
| 4 | E |
| 5 | F |
| 6 | G |
| 7 | H |

logical
memory



**Page
Table
With
Some pages
Not
In memory**



Page fault

Page fault

- Page fault is a hardware interrupt
- It occurs when the page table entry corresponding to current memory access is “i”
- All actions that a kernel takes on a hardware interrupt are taken!

Change of stack to kernel stack

Saving the context of process

Switching to kernel code

On a Page fault

- 1) Operating system looks at another data structure (table), most likely in PCB itself, to decide:
 - If it's Invalid reference -> abort the process (segfault)
 - Just not in memory -> Need to get the page in memory
- 2) Get empty frame (this may be complicated!)
- 3) Swap page into frame via scheduled disk/IO operation
- 4) Reset tables to indicate page now in memory.
- 5) Set validation bit = v
- 6) Restart the instruction that caused the page fault

Additional problems

- Extreme case – start process with *no* pages in memory
 - OS sets instruction pointer to first instruction of process, non-memory-resident -> page fault
 - And for every other process pages on first access

Pure demand paging

- Actually, a given instruction could access multiple pages -> multiple page faults
 - Pain decreased because of **locality of reference**
- Hardware support needed for demand paging
 - Page table with valid / invalid bit
 - Secondary memory (swap device with **swap space**)
 - Instruction restart

Instruction restart

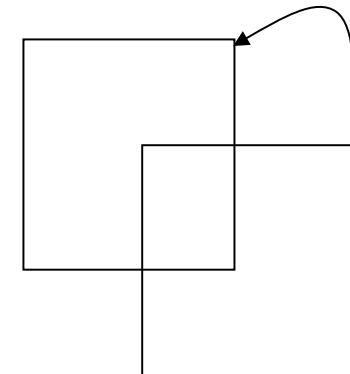
- A critical Problem
- Consider an instruction that could access several different locations

```
movarray 0x100, 0x200, 20
```

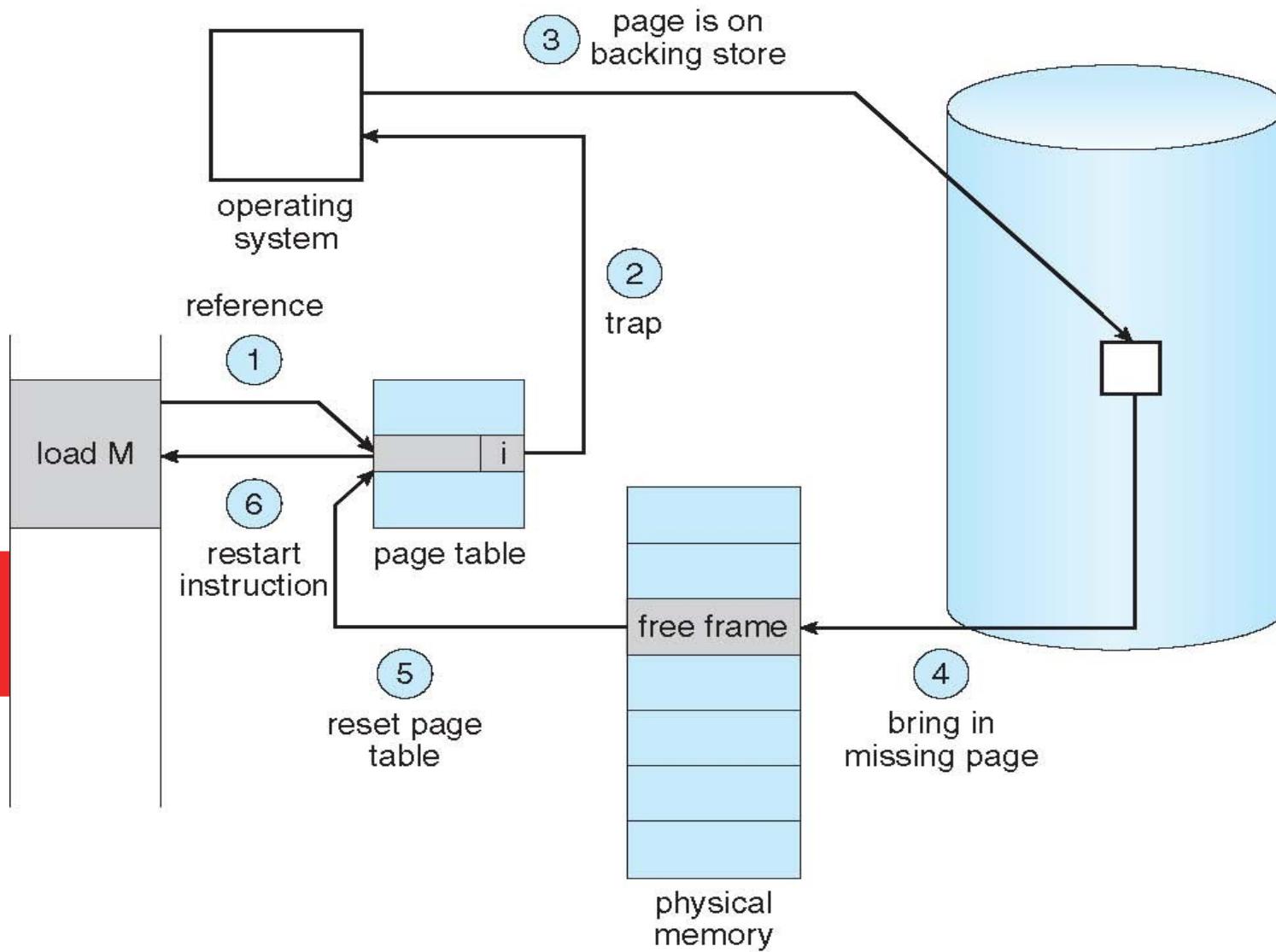
copy 20 bytes from address 0x100 to
address 0x200

```
movarray 0x100, 0x110, 20
```

what to do in this case?



Handling A Page Fault



Page fault handling

- 1) Trap to the operating system
- 2) Default trap handling():

Save the process registers and process state

Determine that the interrupt was a page fault. Run page fault handler.

- 3) Page fault handler(): Check that the page reference was legal and determine the location of the page on the disk. If illegal, terminate process.
- 4) Find a free frame. Issue a read from the disk to a free frame:

Process waits in a queue for disk read. Meanwhile many processes may get scheduled.

Disk DMA hardware transfers data to the free frame and raises interrupt in end

Page fault handling

- 6) (as said on last slide) While waiting, allocate the CPU to some other process
- 7) (as said on last slide) Receive an interrupt from the disk I/O subsystem (I/O completed)
- 8) Default interrupt handling():
 - Save the registers and process state for the other user
 - Determine that the interrupt was from the disk
- 9) Disk interrupt handler():
 - Figure out that the interrupt was for our waiting process
 - Make the process runnable
- 10) Wait for the CPU to be allocated to this process again
 - Kernel restores the page table of the process, marks entry as "v"
 - Restore the user registers, process state, and new page table, and then resume the interrupted instruction

Performance of demand paging

Page Fault Rate $0 \leq p \leq 1$

if $p = 0$ no page faults

if $p = 1$, every reference is a fault

Effective (memory) Access Time (EAT)

$$EAT = (1 - p) * \text{memory access time} +$$

$p * (\text{page fault overhead} // \text{Kernel code execution time}$

$+ \text{swap page out} // \text{time to write an occupied frame to disk}$

$+ \text{swap page in} // \text{time to read data from disk into free frame}$

$+ \text{restart overhead}) // \text{time to reset process context, restart it}$

Performance of demand paging

Memory access time = 200 nanoseconds

Average page-fault service time = 8 milliseconds

$$\text{EAT} = (1 - p) \times 200 + p \text{ (8 milliseconds)}$$

$$= (1 - p \times 200 + p \times 8,000,000)$$

$$= 200 + p \times 7,999,800$$

If one access out of 1,000 causes a page fault, then

$$\text{EAT} = 8.2 \text{ microseconds.}$$

This is a slowdown by a factor of 40!!

If want performance degradation < 10 percent

$$220 > 200 + 7,999,800 \times p$$

$$20 > 7,999,800 \times p$$

$$p < .0000025$$

< one page fault in every 400,000 memory accesses

An optimization: Copy on write

The problem with `fork()` and `exec()`. Consider the case of a shell

```
scanf("%s", cmd);
if(strcmp(cmd, "exit") == 0)
    return 0;
pid = fork(); // A->B
if(pid == 0) {
    ret = execl(cmd, cmd, NULL);
    if(ret == -1) {
        perror("execution failed");
        exit(errno);
    }
} else {
    wait(0);
}
```

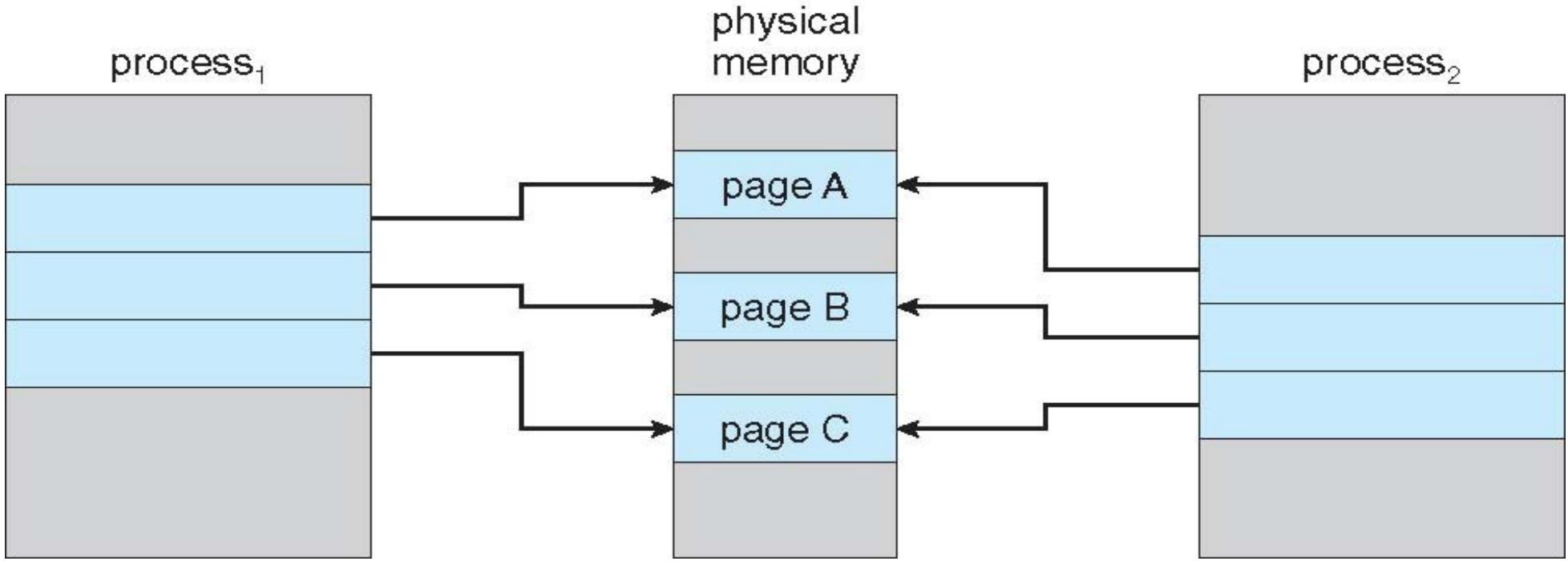
- During `fork()`
 - Pages of parent were duplicated
 - Equal amount of page frames were allocated
 - Page table for child differed from parent (as it has another set of frames)
- In `exec()`
 - The page frames of child were taken away and new frames were allocated
 - Child's page table was rebuilt!
 - Waste of time during `fork()` if the `exec()` was to be called immediately

An optimization: Copy on write

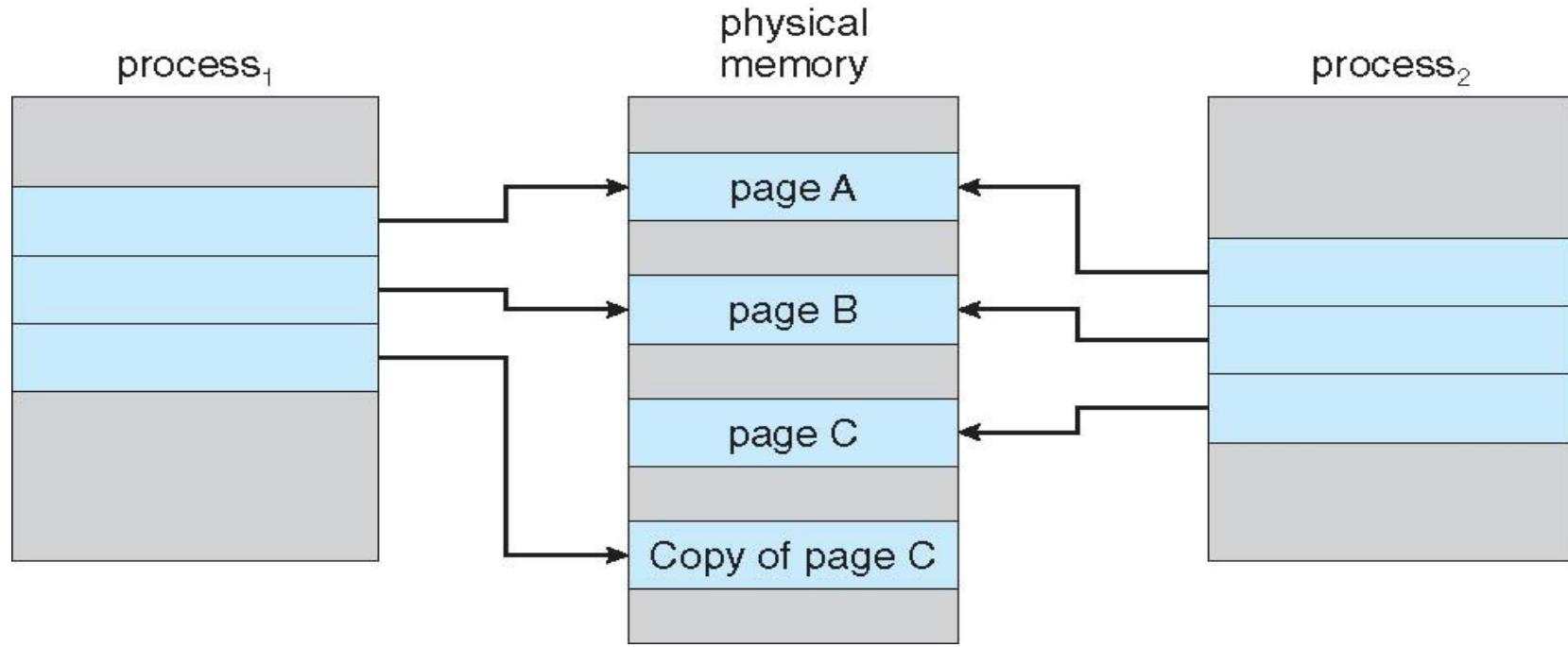
- Copy-on-Write (COW) allows both parent and child processes to initially share the same pages in memory
 - If either process modifies a shared page, only then is the page copied
- COW allows more efficient process creation as only modified pages are copied
- vfork() variation on fork() system call has parent suspend and child using copy-on-write address space of parent

Designed to have child call exec()

Very efficient



Before Process 1 Modifies Page C



After Process 1 Modifies Page C

Challenges and improvements in implementation

- Choice of backing store
 - For stack, heap pages: on swap partition
 - For code, shared library? : swap partition or the actual executable file on the file-system?
 - If the choice is file-system for code, then the page-fault handler needs to call functions related to file-system
- Is the page table itself pageable?
 - If no, good
 - If Yes, then there can be page faults in accessing the page tables themselves! More complicated!
- Is the kernel code pageable?
 - If no, good
 - If yes, life is very complicated ! Page fault in running kernel code, interrupt handlers, system calls, etc.



Page replacement

Review

- Concept of virtual memory, demand paging.
- Page fault
- Performance degradation due to page fault: Need to reduce #page faults to a minimum
- Page fault handling process, broad steps: (1) Trap (2) Locate on disk (3) find free frame (4) schedule disk I/O (5) update page table (6) resume
- More on (3) today

List of free frames

- Kernel needs to maintain a list of free frames
- At the time of loading the kernel, the list is created
- Frames are used for allocating memory to a process
 - But may also be used for managing kernel's own data structures also
- More processes --> more demand for frames

What if no free frame found on page fault?

- Page frames in use depends on “Degree of multiprogramming”
 - More multiprogramming -> overallocation of frames
 - Also in demand from the kernel, I/O buffers, etc
 - How much to allocate to each process? How many processes to allow?
- Page replacement – find some page(frame) in memory, but not really in use, page it out
 - Questions : terminate process? Page out process? replace the page?
 - For performance, need an algorithm which will result in minimum number of page faults
- Bad choices may result in same page being brought into memory several times

Need for Page Replacement

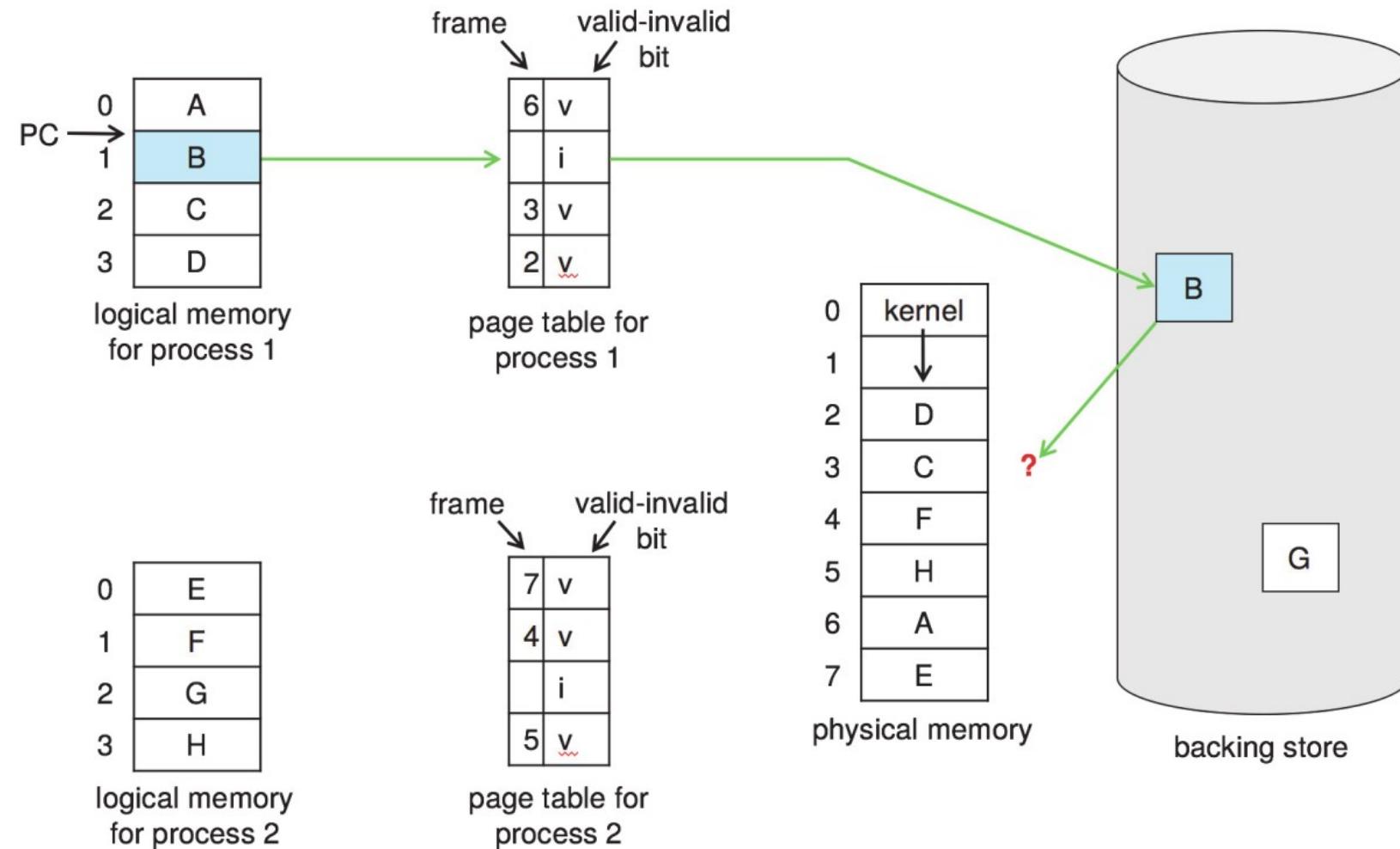


Figure 10.9 Need for page replacement.

Page replacement

- Strategies for performance

Prevent over-allocation of memory by modifying page-fault service routine to include page replacement

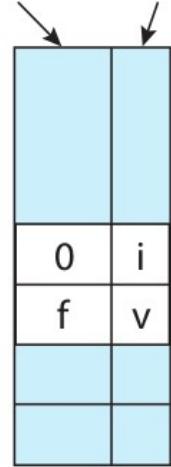
Use modify (dirty) bit in page table. To reduce overhead of page transfers – only modified pages are written to disk. If page is not modified, just reuse it (a copy is already there in backing store)

Basic Page replacement

- 1) Find the location of the desired page on disk
- 2) Find a free frame:
 - If there is a free frame, use it
 - If there is no free frame, use a page replacement algorithm to select a victim frame & write victim frame to disk if dirty
- 3) Bring the desired page into the free frame; update the page table of process and global frame table/list
- 4) Continue the process by restarting the instruction that caused the trap
 - Note now potentially 2 page transfers for page fault – increasing EAT

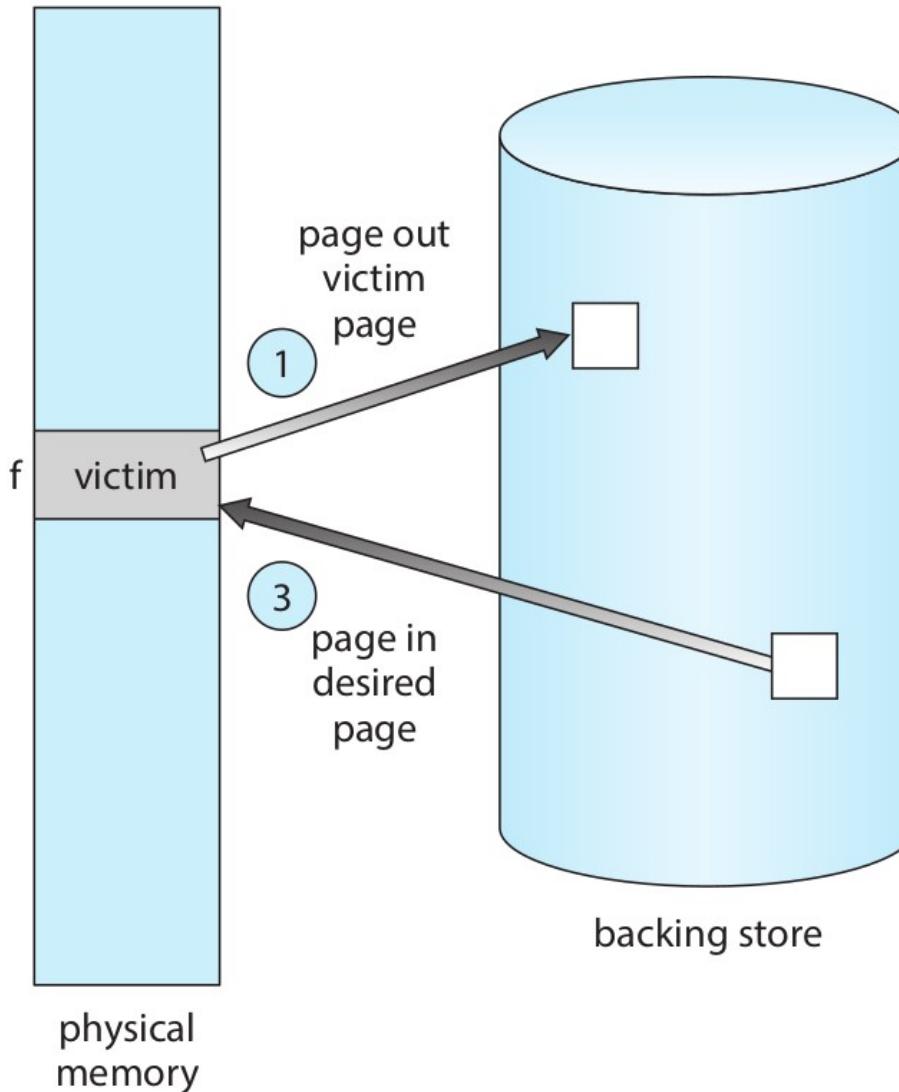
Page Replacement

frame valid-invalid bit



2 change to invalid

4 reset page table for new page

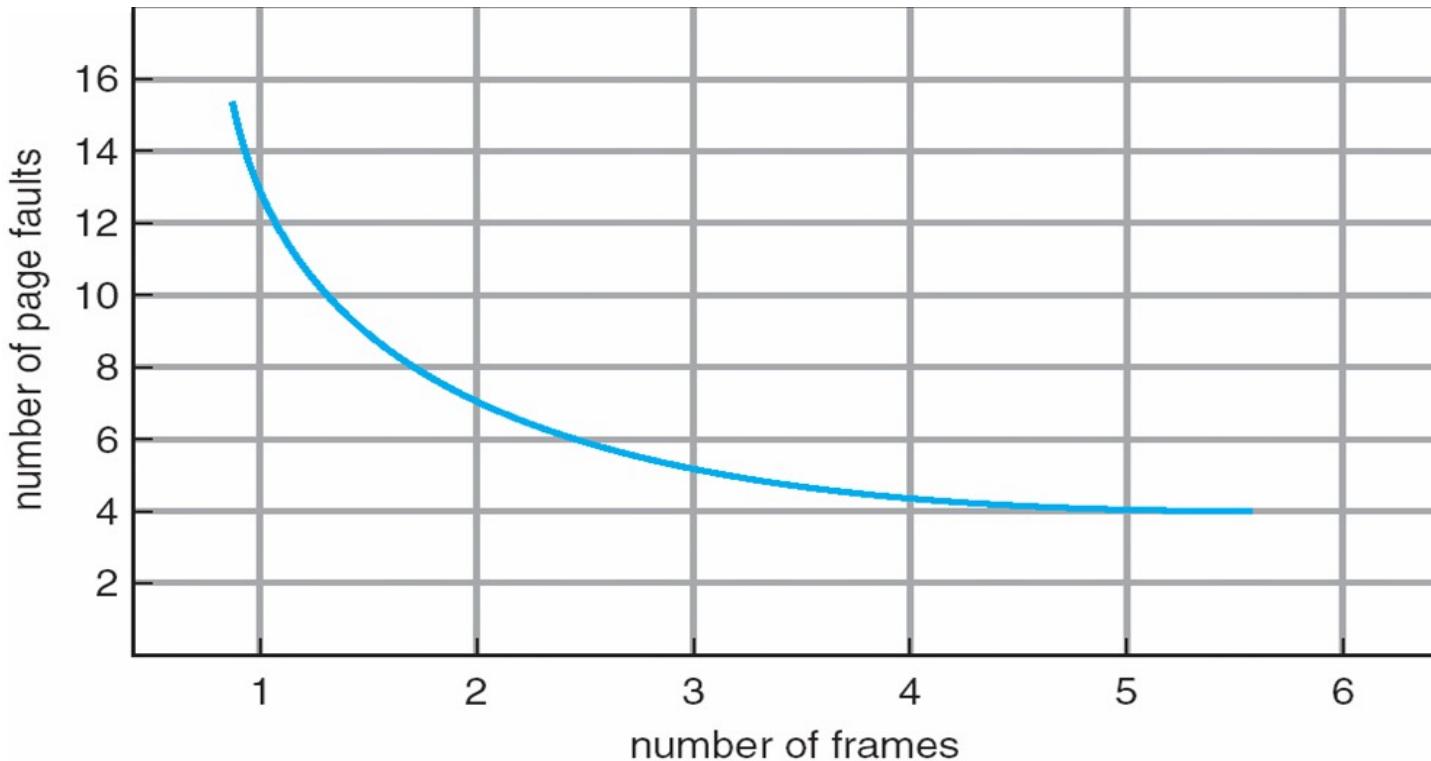


Two problems to solve

- Frame-allocation algorithm determines
 - How many frames to give each process
 - Which frames to replace
- Page-replacement algorithm
 - Want lowest page-fault rate on both first access and re-access

Evaluating algorithm: Reference string

- Evaluate algorithm by running it on a particular string of memory references (reference string) and computing the number of page faults on that string
 - String is just page numbers, not full addresses
 - Repeated access to the same page does not cause a page fault
- In all our examples, the reference string is
7,0,1,2,0,3,0,4,2,3,0,3,0,3,2,1,2,0,1,7,0,1



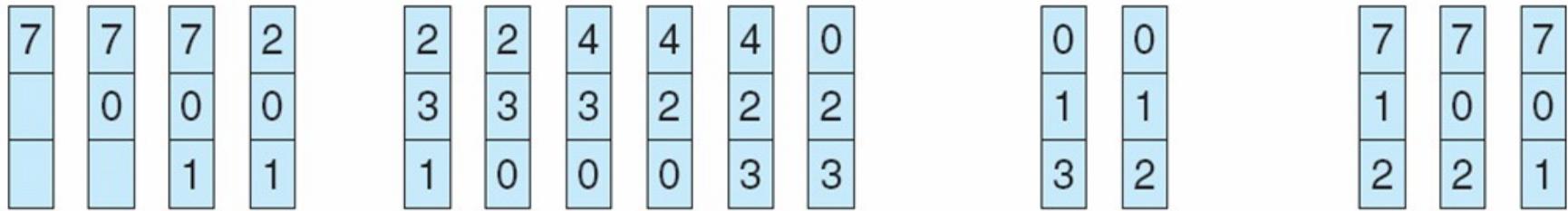
An
Expectation

More page
Frames
Means less
faults

FIFO Algorithm

reference string

7 0 1 2 0 3 0 4 2 3 0 3 2 1 2 0 1 7 0 1



page frames

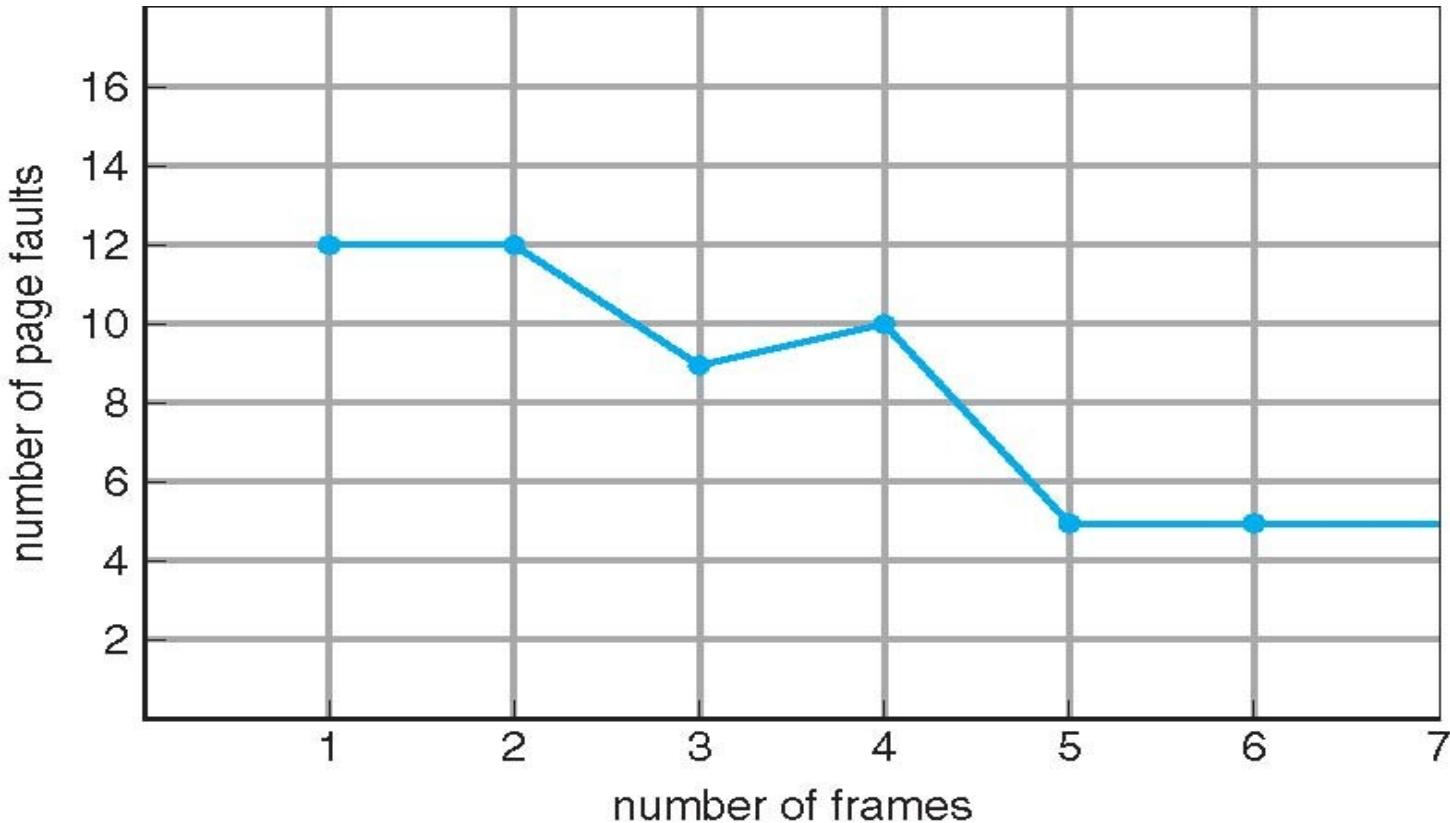
FIFO Algorithm

| | | | | | |
|---|---|---|---|---|---|
| 1 | 7 | 2 | 4 | 0 | 7 |
| 2 | 0 | 3 | 2 | 1 | 0 |
| 3 | 1 | 0 | 3 | 2 | 1 |

15 page faults

- Reference string:
7,0,1,2,0,3,0,4,2,3,0,3,0,3,2,1,2,0,1,
7,0,1
- 3 frames (3 pages can be in memory at a time per process)
- **Belady's Anomaly**
- Adding more frames can cause more page faults!
- Can vary by reference string:
consider 1,2,3,4,1,2,5,1,2,3,4,5

FIFO Algorithm: Balady's anamoly



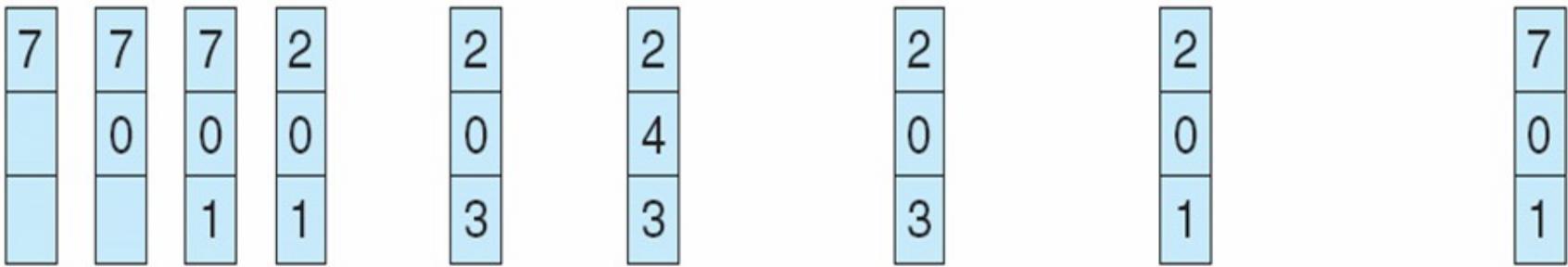
Optimal Algorithm

- Replace page that will not be used for longest period of time
 - 9 is optimal #replacements for the example on the next slide
- How do you know this?
 - Can't read the future
- Used for measuring how well your algorithm performs

Optimal page replacement

reference string

7 0 1 2 0 3 0 4 2 3 0 3 2 1 2 0 1 7 0 1

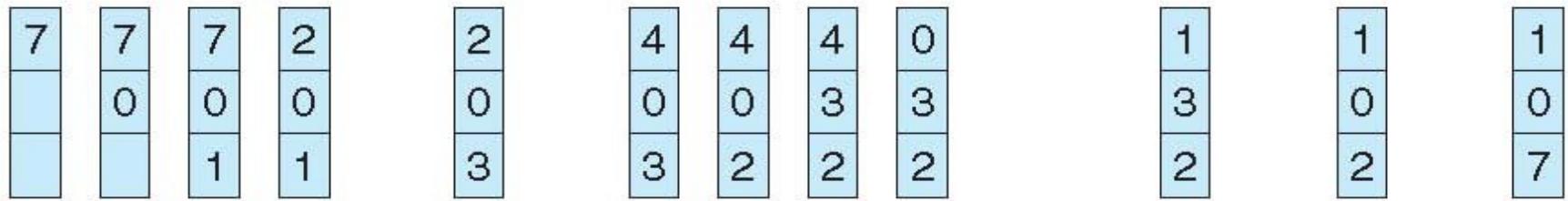


page frames

Least Recently Used: an approximation of optimal

reference string

7 0 1 2 0 3 0 4 2 3 0 3 2 1 2 0 1 7 0 1



page frames

Use past knowledge rather than future

Replace page that has not been used in the most amount of time

Associate time of last use with each page

12 faults – better than FIFO but worse than OPT

Generally good algorithm and frequently used

But how to implement?

LRU: Counter implementation

- Counter implementation
 - Every page entry has a counter; every time page is referenced through this entry, copy the clock into the counter
 - When a page needs to be changed, look at the counters to find smallest value
 - Search through table needed

LRU: Stack implementation

- Keep a stack of page numbers in a double link form:
- Page referenced: move it to the top
 - requires 6 pointers to be changed and
 - each update more expensive
 - But no need of a search for replacement

Use Of A Stack to Record The

Most Recent Page References

reference string

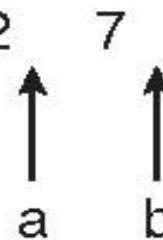
4 7 0 7 1 0 1 2 1 2 7 1 2

| |
|---|
| 2 |
| 1 |
| 0 |
| 7 |
| 4 |

stack
before
a

| |
|---|
| 7 |
| 2 |
| 1 |
| 0 |
| 4 |

stack
after
b



Stack algorithms

- An algorithm for which it can be shown that the set of pages in memory for n frames is always a subset of the set of pages that would be in memory with $n + 1$ frames
- Do not suffer from Balady's anomaly
- For example: Optimal, LRU

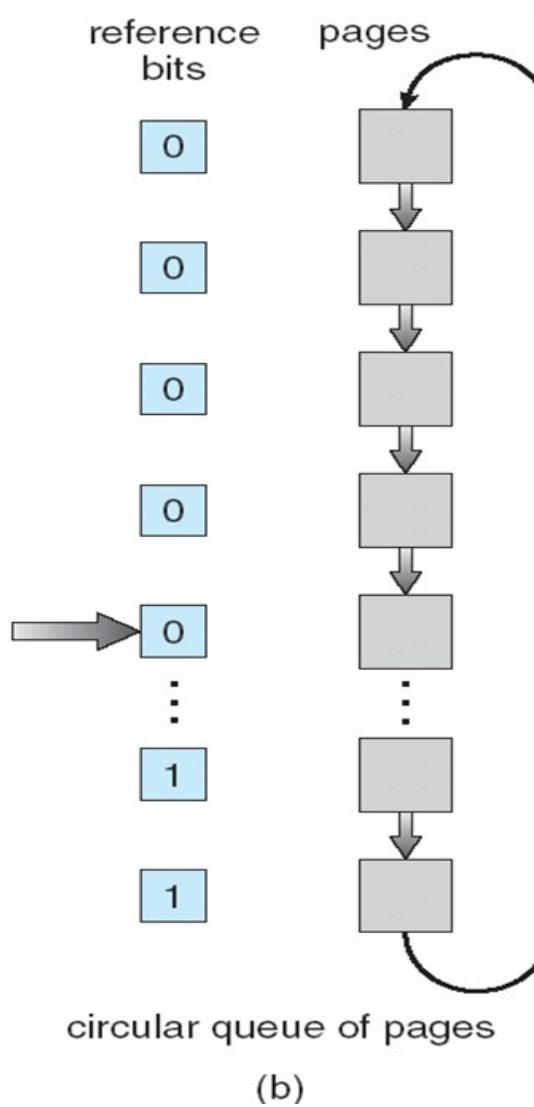
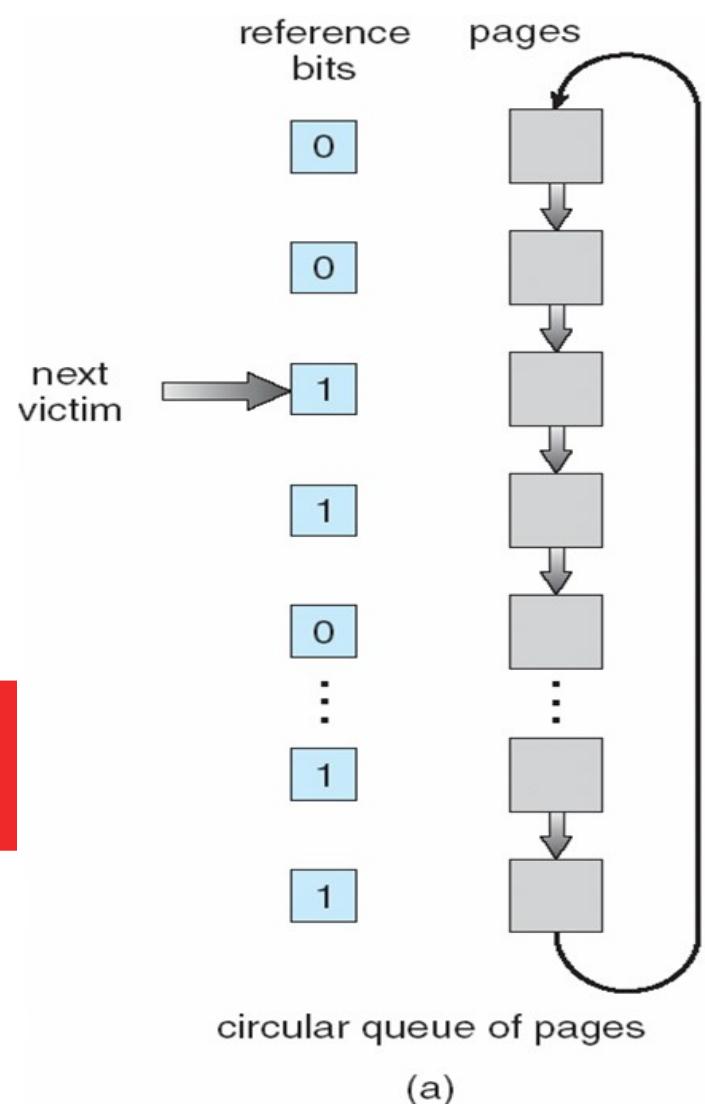
LRU: Approximation algorithms

- LRU needs special hardware and still slow
- **Reference bit**
 - With each page associate a bit, initially = 0
 - When page is referenced bit set to 1
 - Replace any with reference bit = 0 (if one exists)
 - We do not know the order, however

LRU: Approximation algorithms

- **Second-chance algorithm**
 - FIFO + hardware-provided reference bit. If bit is 0 select, if bit is 1, then set it to 0 and move to next one.
- **An implementation of second-chance: Clock replacement**
 - If page to be replaced has
 - Reference bit = 0 -> replace it
 - reference bit = 1 then:
 - set reference bit 0, leave page in memory
 - replace next page, subject to same rules

Second- Chance (clock) Page- Replacement Algorithm



Counting Algorithms

- Keep a counter of the number of references that have been made to each page
 - Not common
- **LFU Algorithm:** replaces page with smallest count
- **MFU Algorithm:** based on the argument that the page with the smallest count was probably just brought in and has yet to be used

Page buffering algorithms

- Keep a pool of free frames, always
 - Then frame available when needed, not found at fault time
 - Read page into free frame and select victim to evict and add to free pool
 - When convenient, evict victim
- Possibly, keep list of modified pages
 - When backing store otherwise idle, write pages there and set to non-dirty
- Possibly, keep free frame contents intact and note what is in them
 - If referenced again before reused, no need to load contents again from disk
 - Generally useful to reduce penalty if wrong victim frame selected

Major and Minor page faults

- Most modern OS refer to these two types
- Major fault
 - Fault + page not in memory
- Minor fault
 - Fault, but page is in memory
 - For example shared memory pages; second instance of fork(), page already on free-frame list,
- On Linux run
 - `$ ps -eo min_flt,maj_flt,cmd`

Special rules for special applications

- All of earlier algorithms have OS guessing about future page access
- But some applications have better knowledge – e.g. databases
- Memory intensive applications can cause double buffering
 - OS keeps copy of page in memory as I/O buffer
 - Application keeps page in memory for its own work
- Operating system can give direct access to the disk, getting out of the way of the applications
 - Raw disk mode
 - Bypasses buffering, locking, etc

Allocation of frames

- Each process needs *minimum* number of frames
- Example: IBM 370 – 6 pages to handle SS MOVE instruction:
 - instruction is 6 bytes, might span 2 pages
 - 2 pages to handle *from*
 - 2 pages to handle *to*
- Maximum of course is total frames in the system
- Two major allocation schemes
 - fixed allocation
 - priority allocation
- Many variations

Fixed allocation of frames

- Equal allocation – For example, if there are 100 frames (after allocating frames for the OS) and 5 processes, give each process 20 frames
 - Keep some as free frame buffer pool
- Proportional allocation – Allocate according to the size of process

Dynamic as degree of multiprogramming, process sizes change

s_i =size of process p_i

$s_1 = 10$

$S = \sum s_i$

$s_2 = 127$

m =total number of frames

$$a_1 = \frac{10}{137} \times 64 \approx 5$$

a_i =allocation for p_i $= \frac{s_i}{S} \times m$

$$a_2 = \frac{127}{137} \times 64 \approx 59$$

Priority Allocation of frames

- Use a proportional allocation scheme using priorities rather than size
- If process P_i generates a page fault,
 - select for replacement one of its frames
 - select for replacement a frame from a process with lower priority number

Global Vs Local allocation

- **Global replacement** – process selects a replacement frame from the set of all frames; one process can take a frame from another
 - But then process execution time can vary greatly
 - But greater throughput so more common
- **Local replacement** – each process selects from only its own set of allocated frames
 - More consistent per-process performance
 - But possibly underutilized memory



Virtual Memory – Remaining topics

Agenda

- Problem of Thrashing and possible solutions
- Mmap(), Memory mapped files
- Kernel Memory Management
- Other Considerations

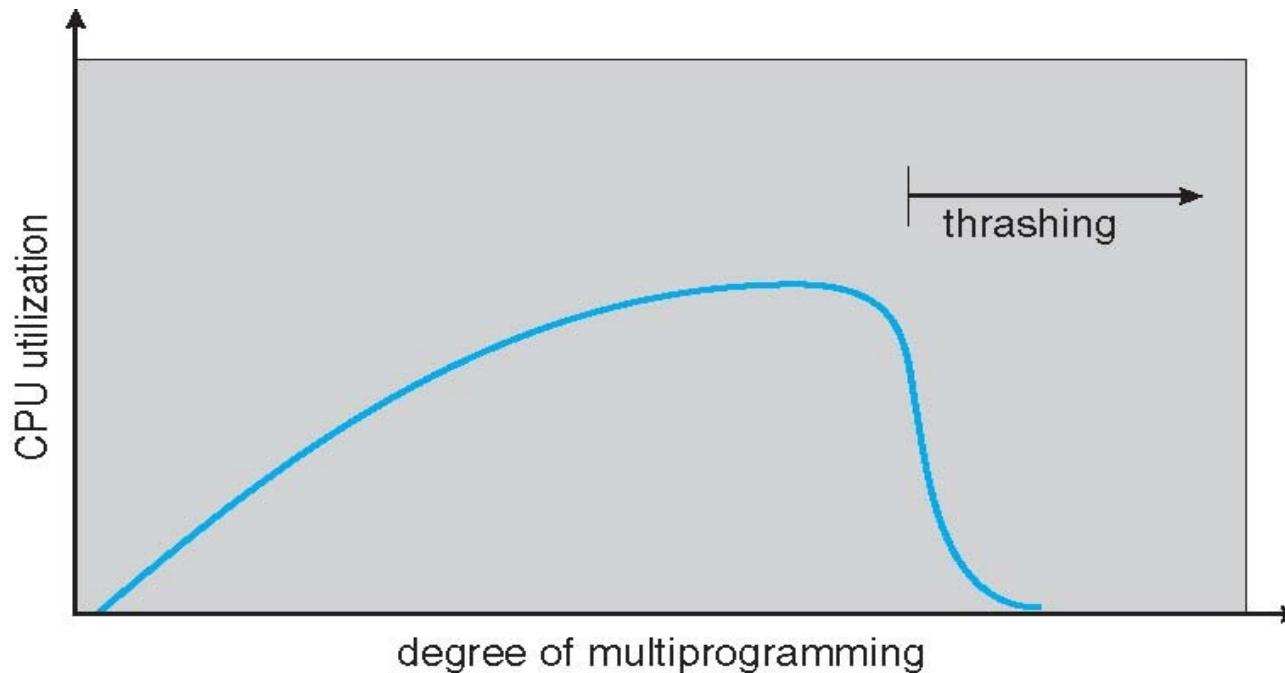


Thrashing

Thrashing

- If a process does not have “enough” pages, the page-fault rate is very high
 - Page fault to get page
 - Replace existing frame
 - But quickly need replaced frame back
- This leads to:
 - Low CPU utilization
 - Operating system thinking that it needs to increase the degree of multiprogramming
 - Another process added to the system
- Thrashing : a process is busy swapping pages in and out

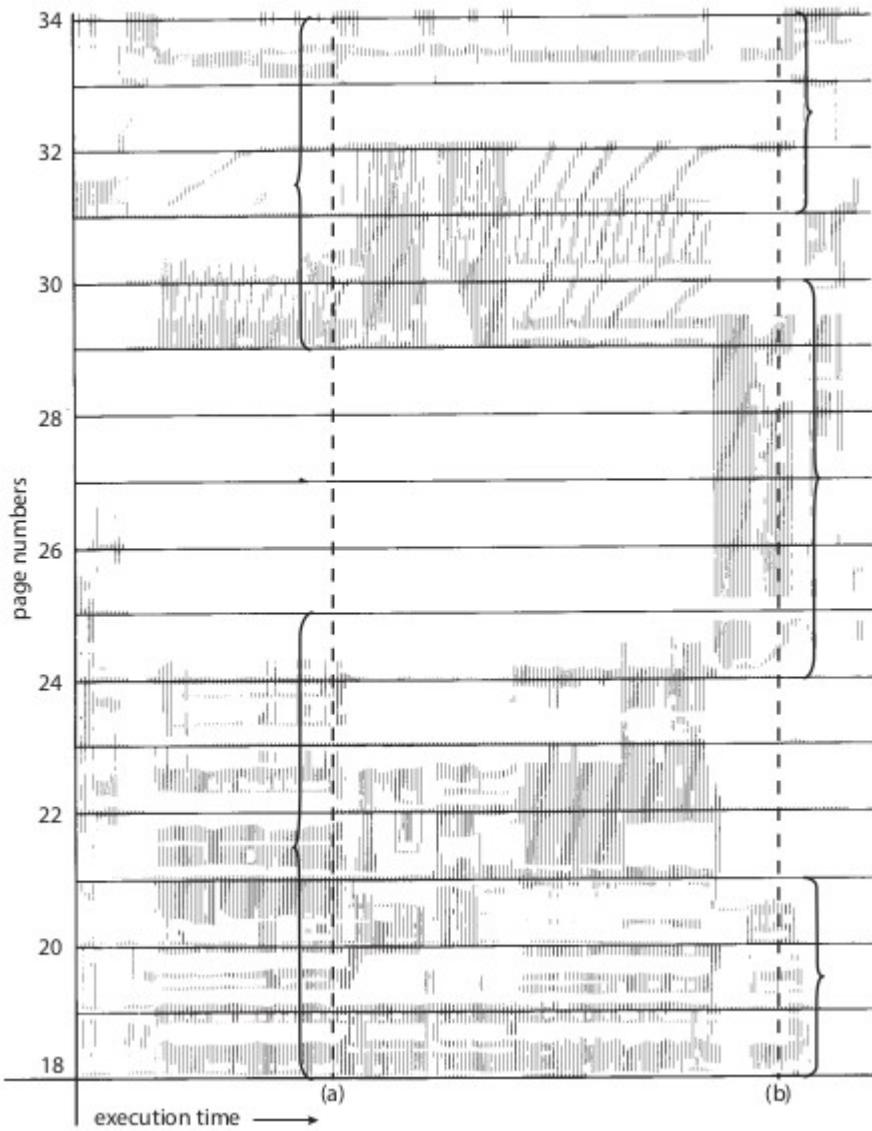
Thrashing



Demand paging and thrashing

- Why does demand paging work?
 - Locality model
 - Process migrates from one locality to another
 - Localities may overlap
- Why does thrashing occur?
 - size of locality > total memory size
 - Limit effects by using local or priority page replacement

Locality In A Memory-Reference Pattern



Working set model

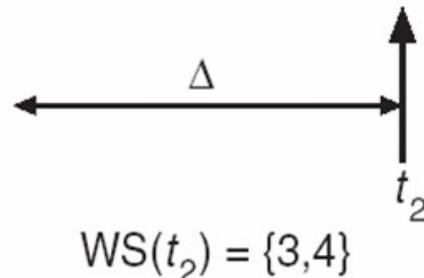
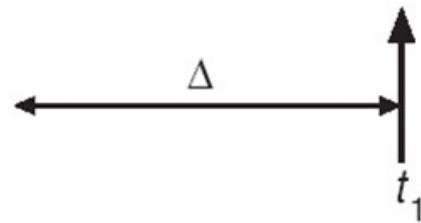
- $\Delta \equiv$ working-set window \equiv a fixed number of page references
 - Example: 10,000 instructions
- Working Set Size, WSS_i (working set of Process P_i) =
 - total number of pages referenced in the most recent Δ (varies in time)
 - if Δ too small will not encompass entire locality
 - if Δ too large will encompass several localities
 - if $\Delta = \infty \Rightarrow$ will encompass entire program

Working set model

- $D = \sum WSS_i$ \equiv total demand frames
 - Approximation of locality
 - if $D > m$ (total available frames) \Rightarrow Thrashing
 - Policy if $D > m$, then suspend or swap out one of the processes

page reference table

... 2 6 1 5 7 7 7 7 5 1 6 2 3 4 1 2 3 4 4 4 3 4 3 4 4 4 1 3 2 3 4 4 4 3 4 4 4 ...



Keeping Track of the Working Set

- Approximate with interval timer + a reference bit
- Example: $\Delta = 10,000$

Timer interrupts after every 5000 time units

Keep in memory 2 bits for each page

Whenever a timer interrupt occurs, copy (to memory) and sets the values of all reference bits to 0

If one of the bits in memory = 1 \Rightarrow page in working set

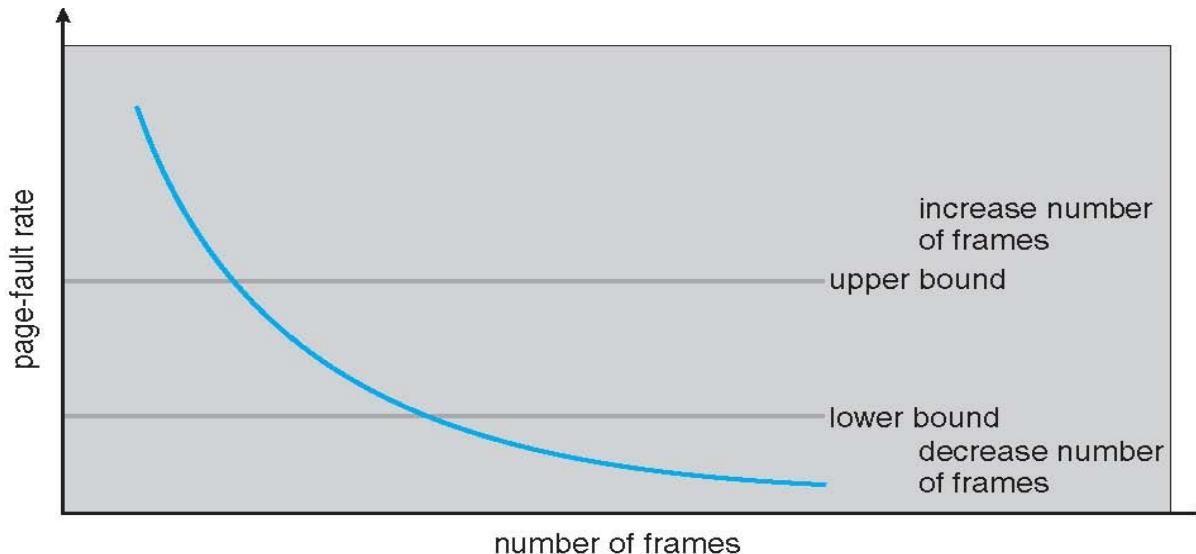
- Why is this not completely accurate?
- Improvement = 10 bits and interrupt every 1000 time units

Page fault frequency

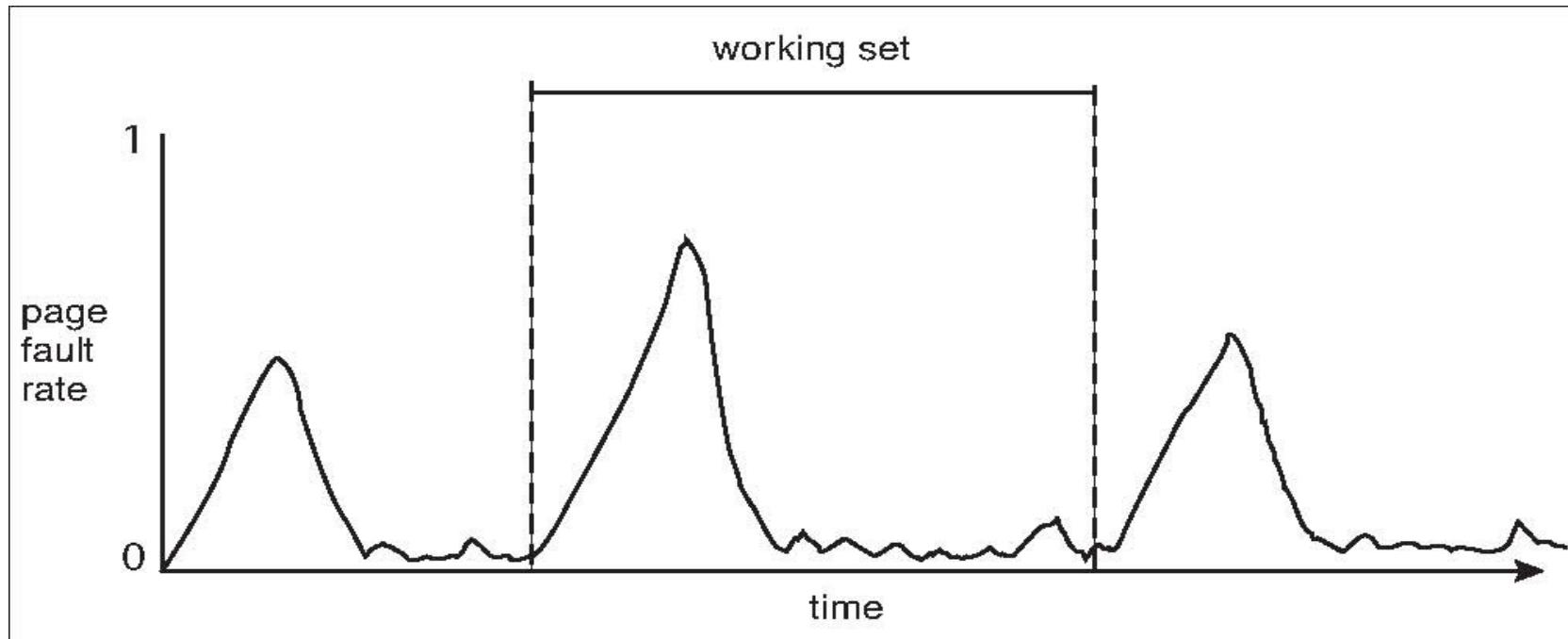
- More direct approach than WSS
- Establish “acceptable” page-fault frequency rate and use local replacement policy

If actual rate too low, process loses frame

If actual rate too high, process gains frame



Working Sets and Page Fault Rates



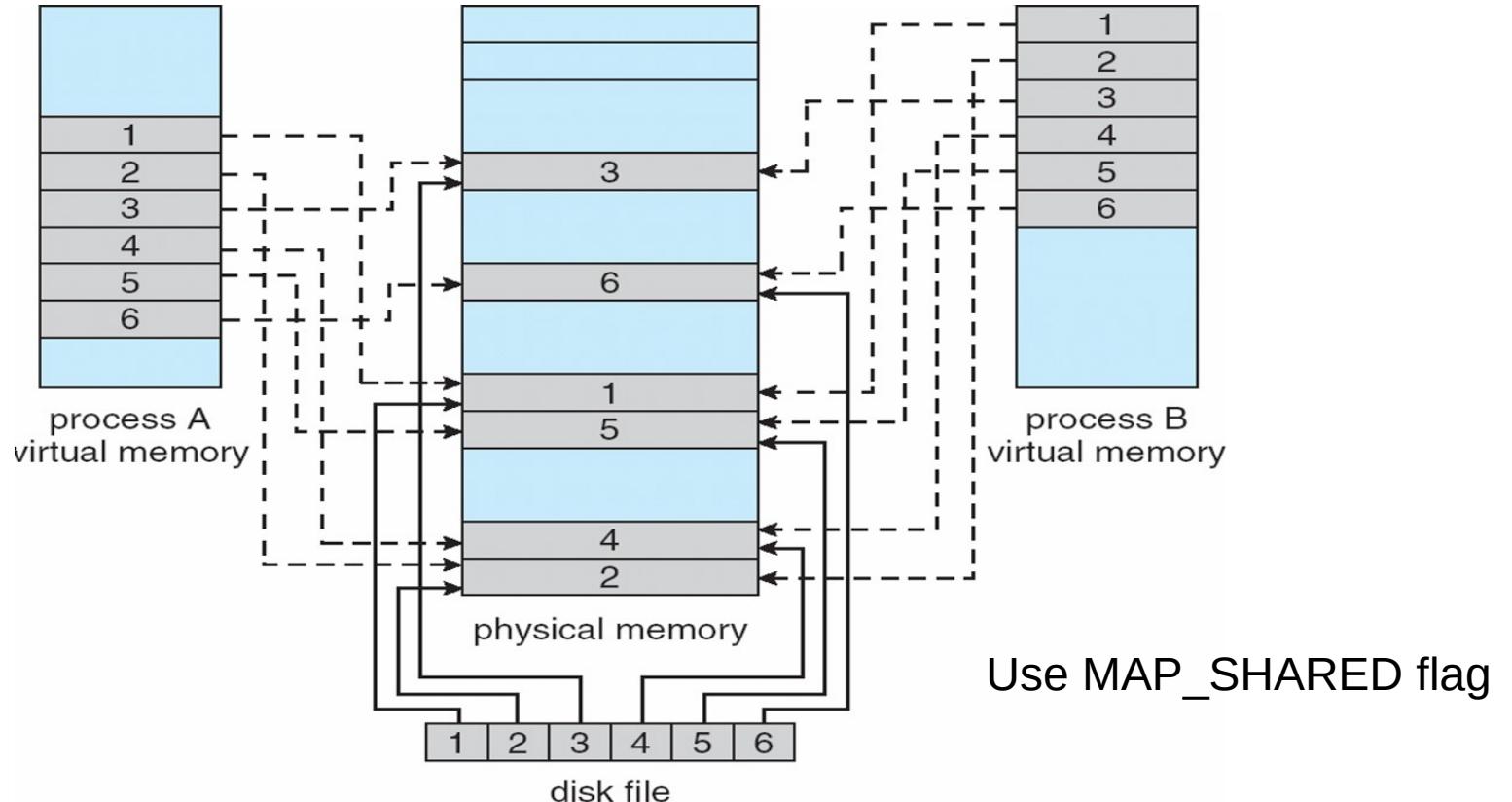


Memory Mapped Files

Memory-Mapped Files

- First, let's see a demo of using mmap()

Memory-Mapped Files



Memory-Mapped Files

- Memory-mapped file I/O allows file I/O to be treated as routine memory access by mapping a disk block to a page in memory
- A file is initially read using demand paging

A page-sized portion of the file is read from the file system into a physical page

Subsequent reads/writes to/from the file are treated as ordinary memory accesses

- Simplifies and speeds file access by driving file I/O through memory rather than read() and write() system calls
- Also allows several processes to map the same file allowing the pages in memory to be shared
- But when does written data make it to disk?

Periodically and / or at file close() time

For example, when the pager scans for dirty pages

Memory-Mapped Files

- Some OSes uses memory mapped files for standard I/O
- Process can explicitly request memory mapping a file via `mmap()` system call
 - Now file mapped into process address space
- For standard I/O (`open()`, `read()`, `write()`, `close()`), `mmap` anyway
 - But map file into kernel address space
 - Process still does `read()` and `write()`
 - Copies data to and from kernel space and user space
 - Uses efficient memory management subsystem
 - Avoids needing separate subsystem
- COW can be used for read/write non-shared pages
- Memory mapped files can be used for shared memory (although again via separate system calls)



Allocating Kernel Memory

Allocating kernel memory

- Treated differently from user memory
- Often allocated from a free-memory pool

Kernel requests memory for structures of varying sizes

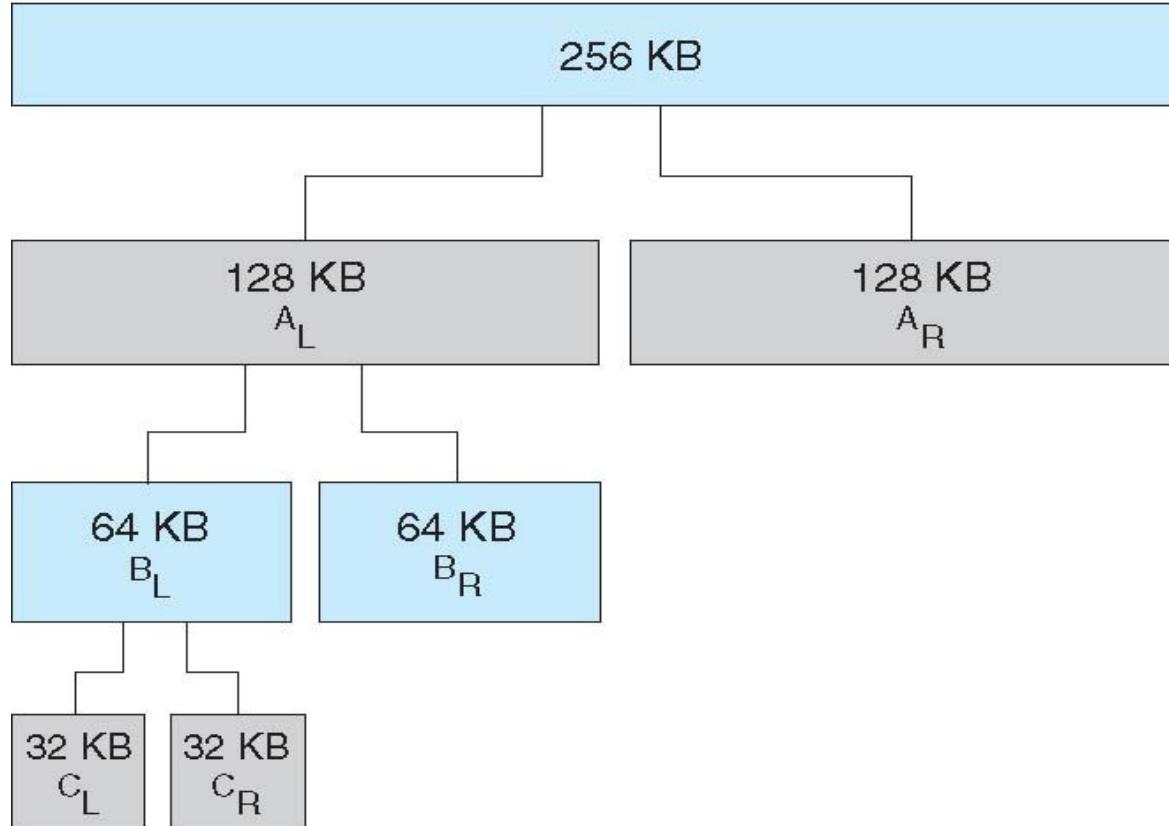
Some kernel memory needs to be contiguous

I.e. for device I/O

-

Buddy Allocator

physically contiguous pages



Buddy Allocator

- Allocates memory from fixed-size segment consisting of physically-contiguous pages
- Memory allocated using power-of-2 allocator

Satisfies requests in units sized as power of 2

Request rounded up to next highest power of 2

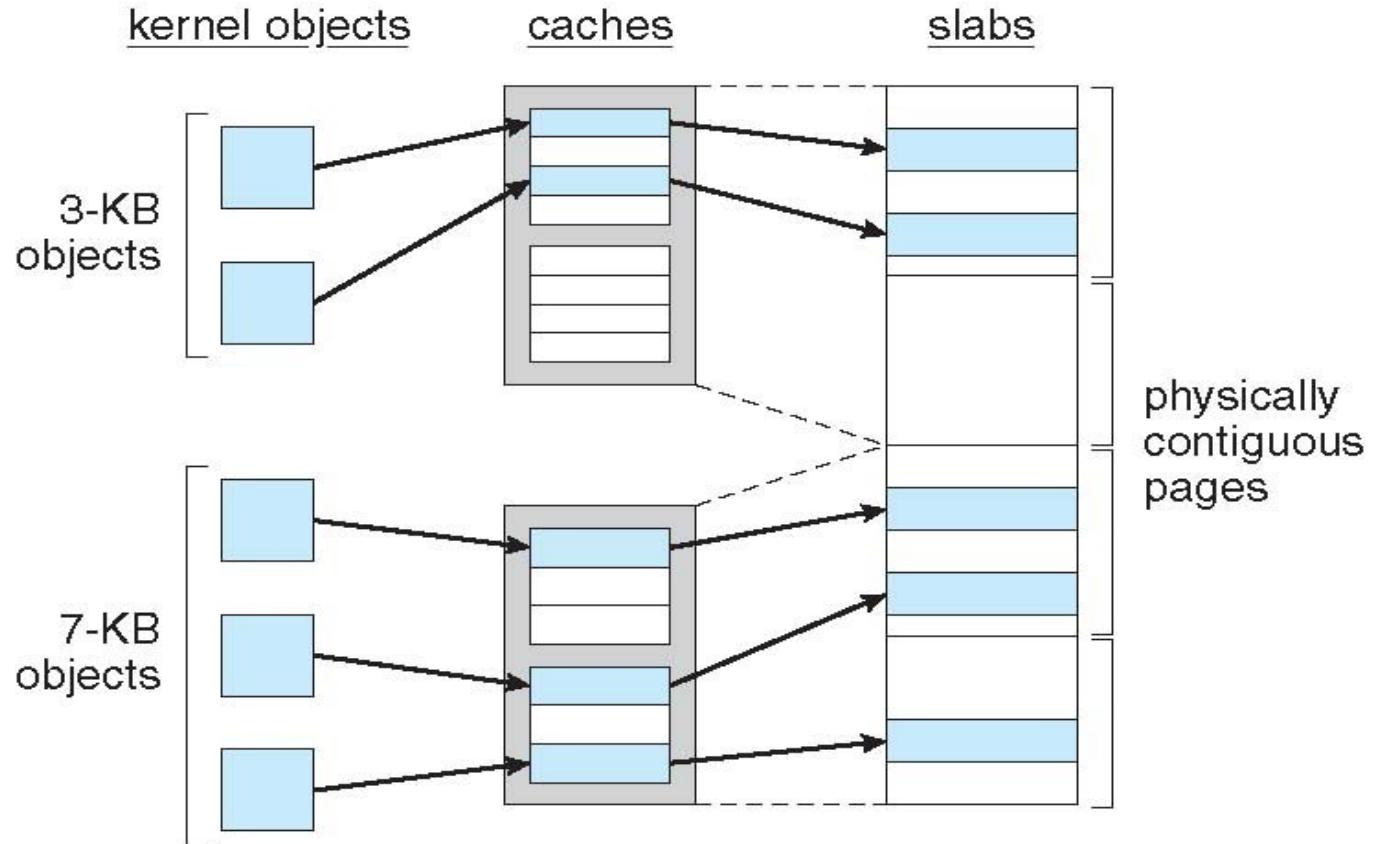
When smaller allocation needed than is available, current chunk split into two buddies of next-lower power of 2

Continue until appropriate sized chunk available

Buddy Allocator

- Continue until appropriate sized chunk available
- For example, assume 256KB chunk available, kernel requests 21KB
 - Split into AL and Ar of 128KB each
 - One further divided into BL and BR of 64KB
 - One further into CL and CR of 32KB each – one used to satisfy request
- Advantage – quickly coalesce unused chunks into larger chunk
- Disadvantage - fragmentation

Slab Allocator



Slab Allocator

- Alternate strategy
- **Slab** is one or more physically contiguous pages
- **Cache** consists of one or more slabs
- Single cache for each unique kernel data structure
 - Each cache filled with **objects** – instantiations of the data structure
- When cache created, filled with objects marked as **free**
- When structures stored, objects marked as **used**
- If slab is full of used objects, next object allocated from empty slab
 - If no empty slabs, new slab allocated
- Benefits include no fragmentation, fast memory request satisfaction



Other considerations

Other Considerations -- Prepaging

- To reduce the large number of page faults that occurs at process startup
- Prepage all or some of the pages a process will need, before they are referenced
- But if prepaged pages are unused, I/O and memory was wasted
- Assume s pages are prepaged and α of the pages is used
 - Is cost of $s * \alpha$ save pages faults > or < than the cost of prepaging
 - $s * (1 - \alpha)$ unnecessary pages?
 - α near zero --> prepaging loses

Page Size

- Sometimes OS designers have a choice
Especially if running on custom-built CPU
- Page size selection must take into consideration:
 - Fragmentation
 - Page table size

Resolution

I/O overhead

Number of page faults

Locality

TLB size and effectiveness

- Always power of 2, usually in the range 2^{12} (4,096 bytes) to 2^{22} (4,194,304 bytes)
- On average, growing over time

TLB Reach

- TLB Reach - The amount of memory accessible from the TLB
- $\text{TLB Reach} = (\text{TLB Size}) \times (\text{Page Size})$
- Ideally, the working set of each process is stored in the TLB
Otherwise there is a high degree of page faults
- Increase the Page Size
This may lead to an increase in fragmentation as not all applications require a large page size
- Provide Multiple Page Sizes
This allows applications that require larger page sizes the opportunity to use them without an increase in fragmentation

Program Structure

- Program structure
- `Int[128, 128] data;`
- Each row is stored in one page
- Program 1

```
for (j = 0; j < 128; j++)
    for (i = 0; i < 128; i++)
        data[i, j] = 0;
```

128 x 128 = 16,384 page faults

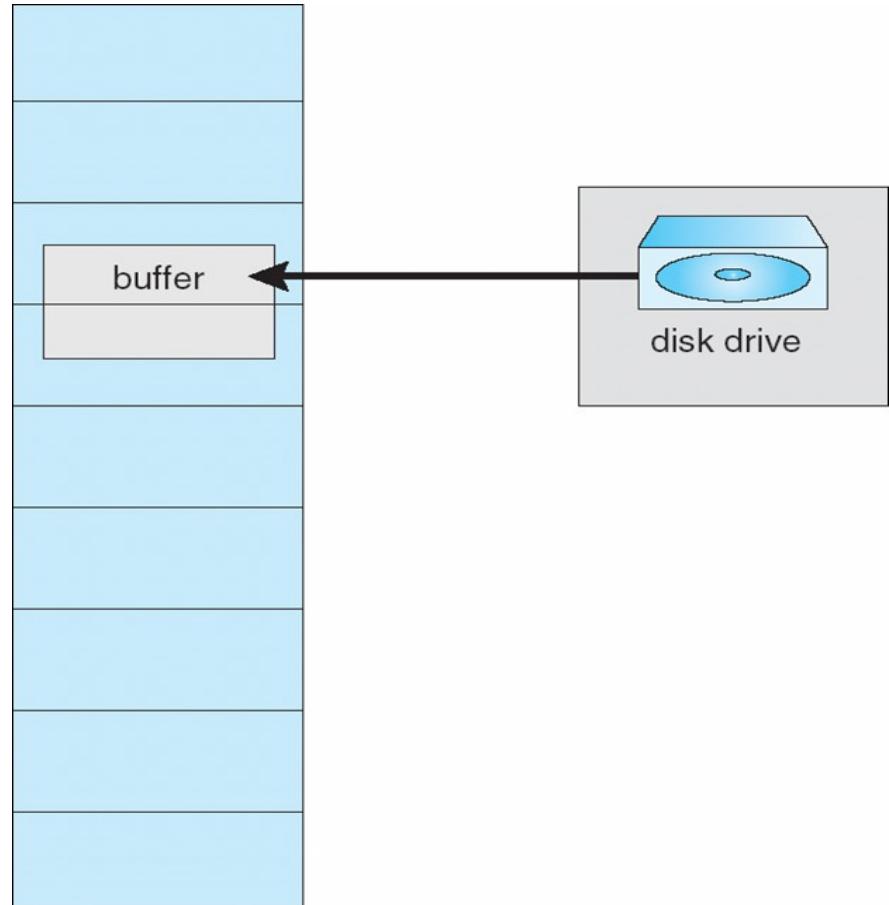
- Program 2

```
for (i = 0; i < 128; i++)
    for (j = 0; j < 128; j++)
        data[i, j] = 0;
```

128 page faults

I/O Interlock

- **I/O Interlock** – Pages must sometimes be locked into memory
- Consider I/O - Pages that are used for copying a file from a device must be locked from being selected for eviction by a page replacement algorithm



Threads, Signals

Abhijit A M
abhijit.comp@coep.ac.in

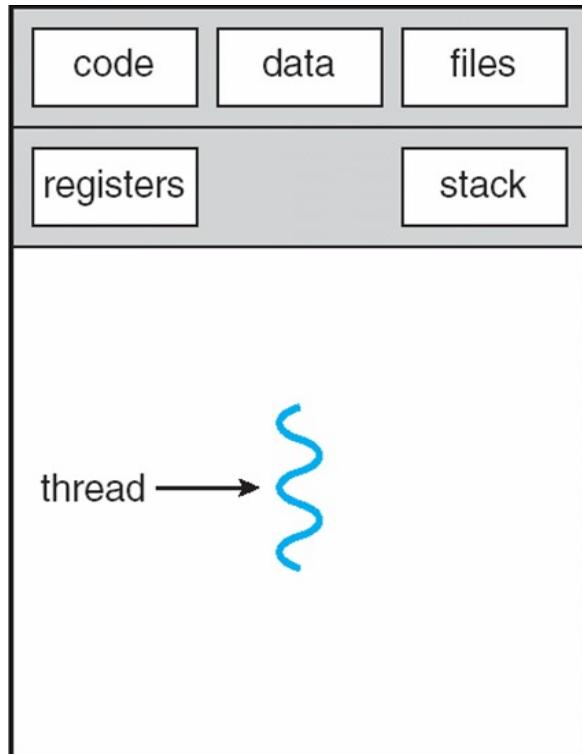
Threads

- **thread — a fundamental unit of CPU utilization**
 - A separate control flow within a program
 - set of instructions that execute “concurrently” with other parts of the code
 - Note the difference: Concurrency: progress at the same time, Parallel: execution at the same time
- **Threads run within application**
 - An application can be divided into multiple parts
 - Each part may be written to execute as a threads
- **Let's see an example**

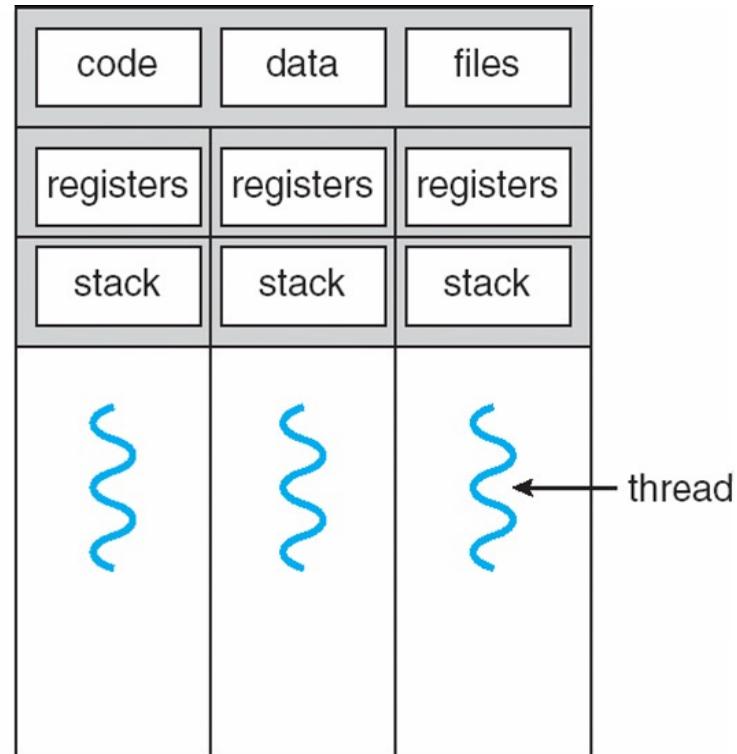
Threads

- **Multiple tasks with the application can be implemented by separate threads**
 - Update display
 - Fetch data
 - Spell checking
 - Answer a network request
- **Process creation is heavy-weight while thread creation is light-weight, due to the very nature of threads**
- **Can simplify code, increase efficiency**
- **Kernels are generally multithreaded**

Single vs Multithreaded process

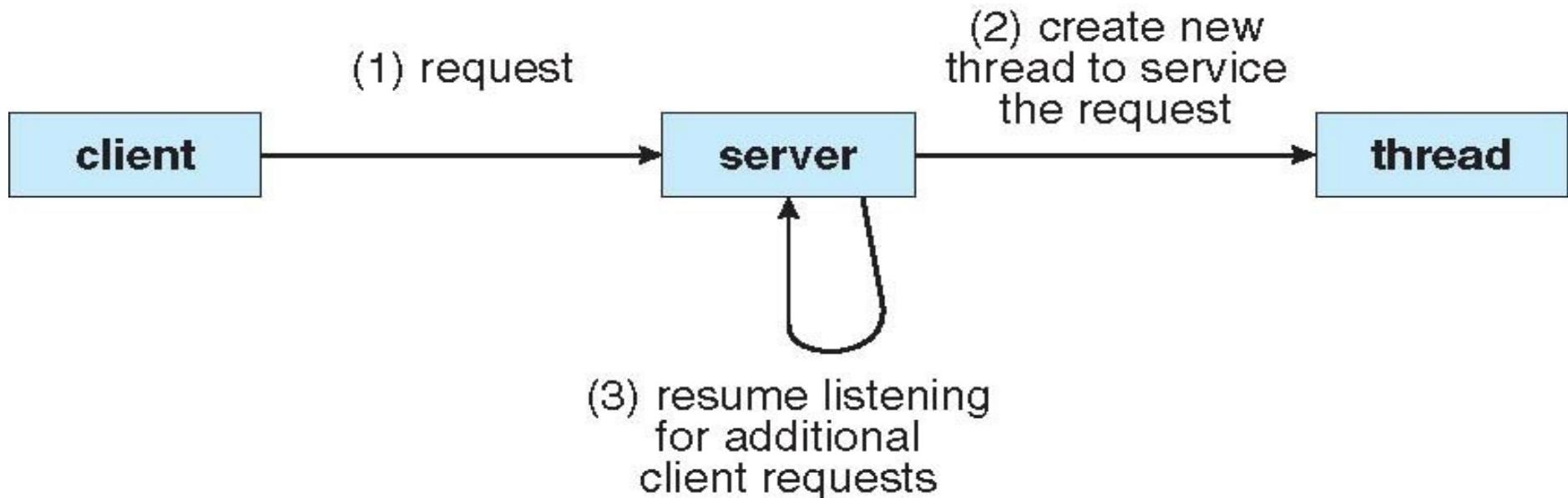


single-threaded process



multithreaded process

A multithreaded server

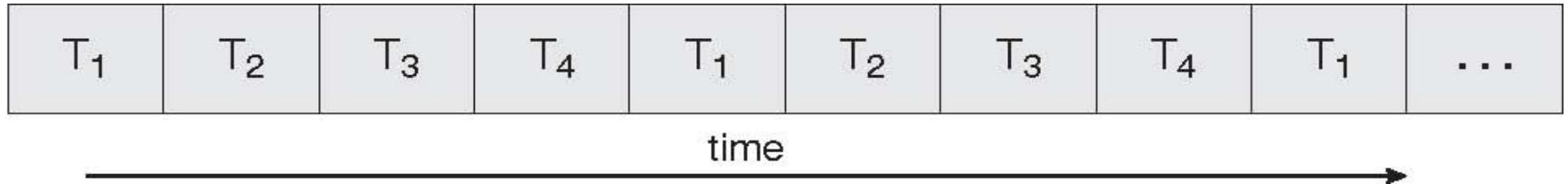


Benefits of threads

- **Responsiveness**
- **Resource Sharing**
- **Economy**
- **Scalability**

Single vs Multicore systems

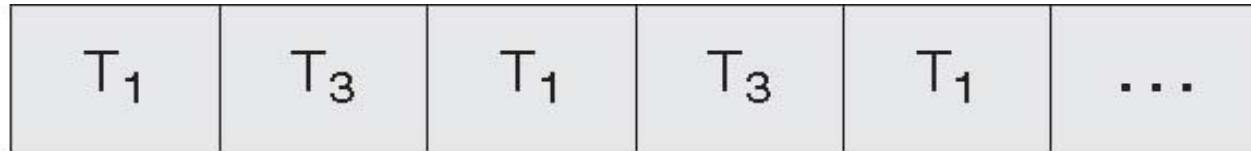
single core



Single core : Concurrency possible

Multicore : parallel execution possible

core 1



core 2



time

Multicore programming

- Multicore systems putting pressure on programmers, challenges include:
 - Dividing activities
 - Balance
 - Data splitting
 - Data dependency
 - Testing and debugging
 -

User vs Kernel Threads

- **User Threads:** Thread management done by user-level threads library
- Three primary thread libraries:
 - POSIX Pthreads
 - Win32 threads
 - Java threads
- **Kernel Threads:** Supported by the Kernel
- **Examples**
 - Windows XP/2000
 - Solaris
 - Linux
 - Tru64 UNIX
 - Mac OS X

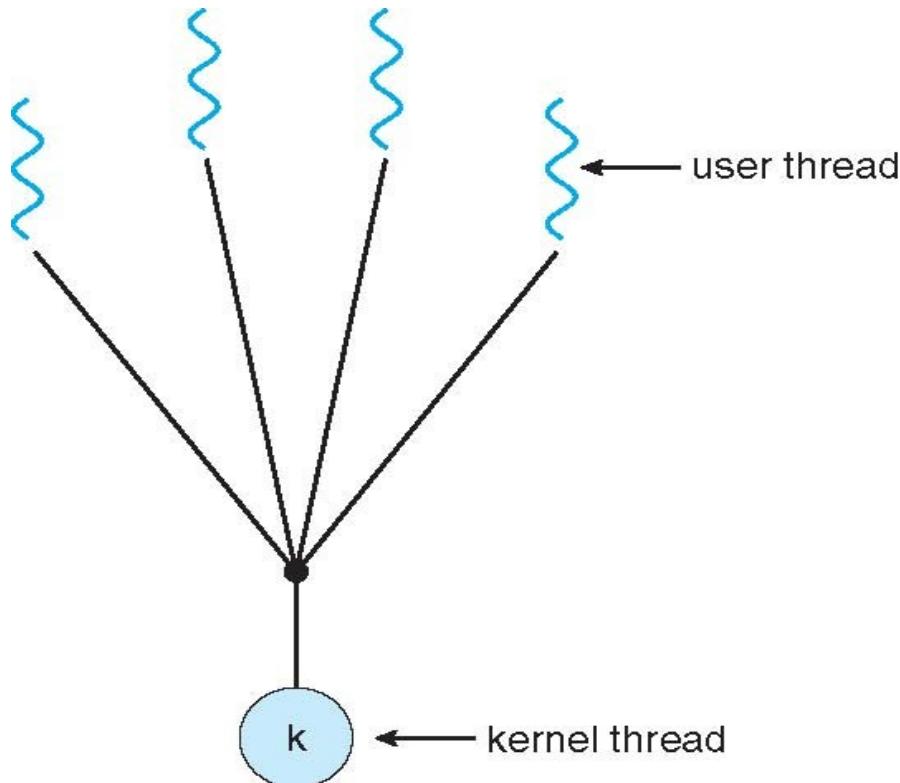
User threads vs Kernel Threads

- **User threads**
 - User level library provides a “typedef” called threads
 - The scheduling of threads needs to be implemented in the user level library
 - Need some type of timer handling functionality at user level execution of CPU
 - OS needs to provide system calls for this
 - Kernel does not know that there are threads!
- **Kernel Threads**
 - Kernel implements concept of threads
 - Still, there may be a user level library, that maps kernel concept of threads to “user concept” since applications link with user level libraries
 - Kernel does scheduling!

Multithreading models

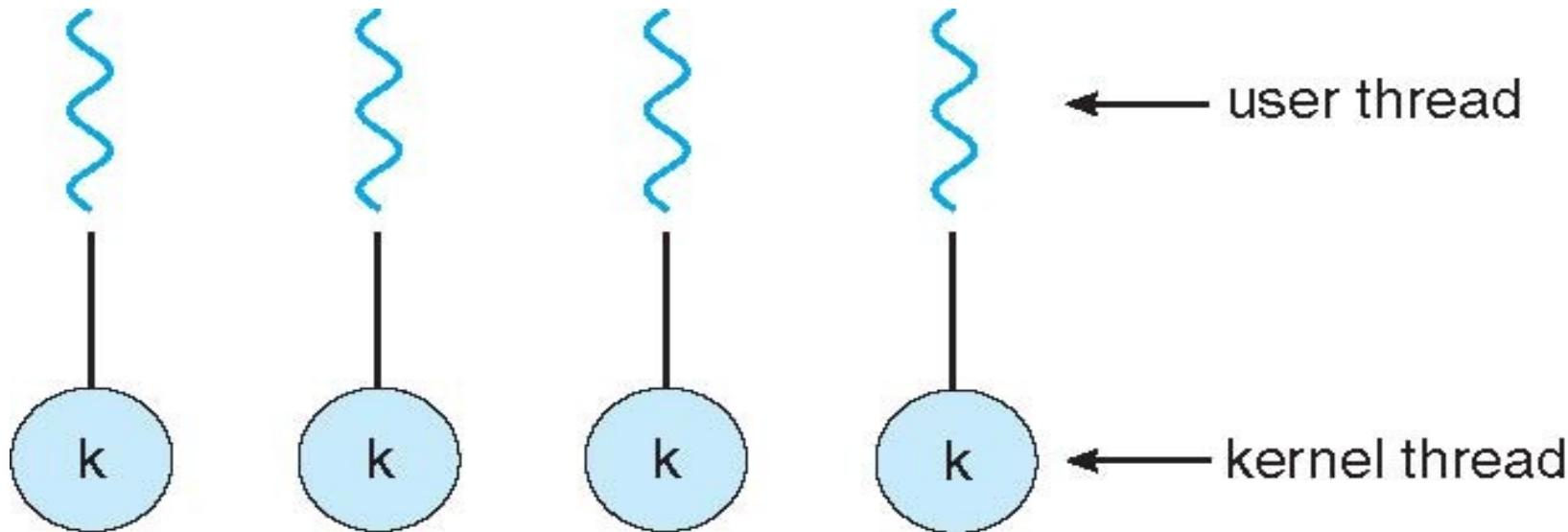
- **How to map user threads to kernel threads?**
 - Many-to-One
 - One-to-One
 - Many-to-Many
- **What if there are no kernel threads?**
 - Then only “one” process. Hence many-one mapping possible, to be done by user level thread library
 - Is One-One possible?

Many-One Model



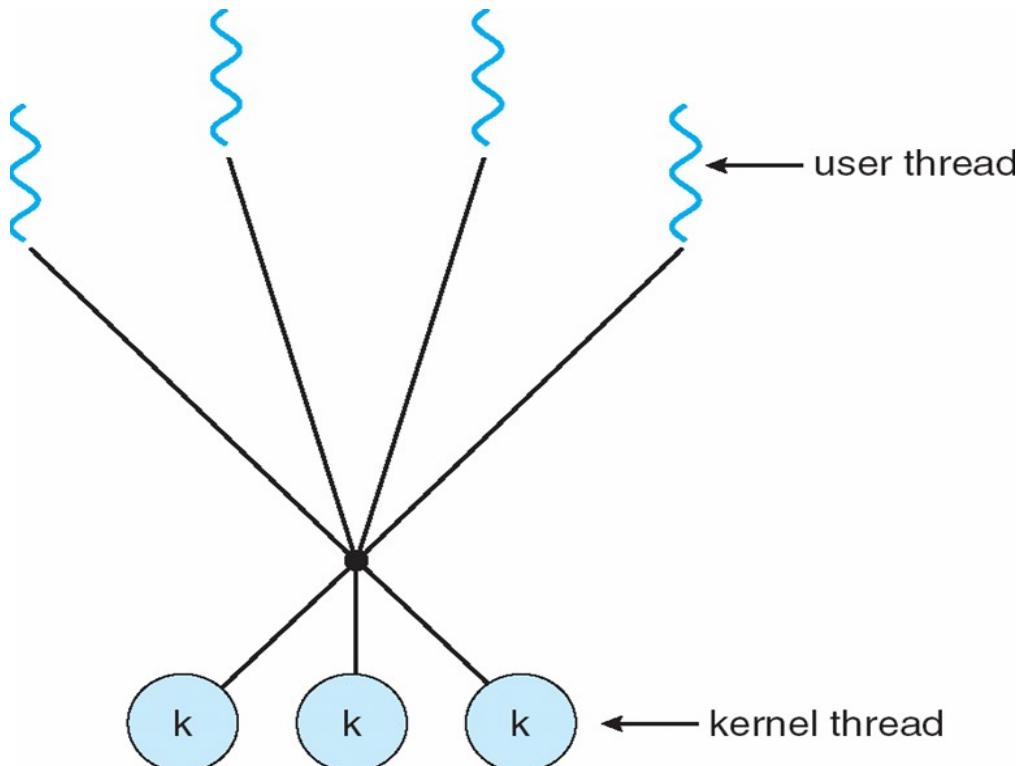
- **Many user-level threads mapped to single kernel thread**
- **Examples:**
 - Solaris Green Threads
 - GNU Portable Threads

One-One Model



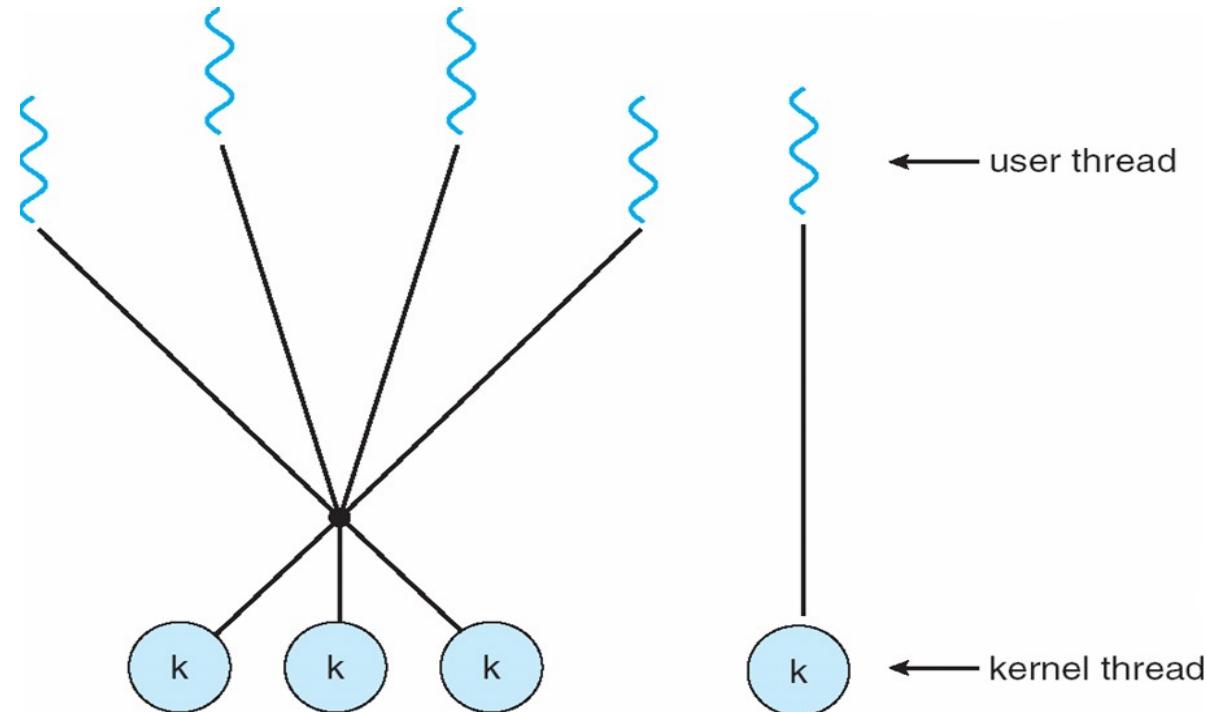
- Each user-level thread maps to kernel thread
- Examples
 - Windows NT/XP/2000
 - Linux
 - Solaris 9 and later

Many-Many Model



- Allows many user level threads to be mapped to many kernel threads
- Allows the operating system to create a sufficient number of kernel threads
- Solaris prior to version 9
- Windows NT/2000 with the ThreadFiber package

Two Level Model



- Similar to M:M, except that it allows a user thread to be bound to kernel thread
- Examples
 - IRIX
 - HP-UX
 - Tru64 UNIX
 - Solaris 8 and earlier

Thread Libraries

Thread libraries

- Thread library provides programmer with API for creating and managing threads
- Two primary ways of implementing
 - Library entirely in user space
 - Kernel-level library supported by the OS

pthreads

- May be provided either as user-level or kernel-level
- A POSIX standard (IEEE 1003.1c) API for thread creation and synchronization
- API specifies behavior of the thread library, implementation is up to development of the library
- Common in UNIX operating systems (Solaris, Linux, Mac OS X)
-

Demo of pthreads code

**Demonstration on Linux – see the code,
compile and execute it.**

Other libraries

- **Windows threading API**
 - `CreateThread(...)`
 - `WaitForSingleObject(...)`
 - `CloseHandle(...)`
- **Java Threads**
 - The `Threads` class
 - The `Runnable` Interface

Issues with threads

- **Semantics of fork() and exec() system calls**
 - Does fork() duplicate only the calling thread or all threads?
- **Thread cancellation of target thread**
 - Terminating a thread before it has finished
 - Two general approaches:
 - Asynchronous cancellation terminates the target thread immediately.
 - Deferred cancellation allows the target thread to periodically check if it should be cancelled.

More on threads

Thread pools

- Some kernels/libraries can provide system calls to :
Create a number of threads in a pool where they await work, assign work/function to a waiting thread
- **Advantages:**
 - Usually slightly faster to service a request with an existing thread than create a new thread
 - Allows the number of threads in the application(s) to be bound to the size of the pool

Thread Local Storage (TLS)

- Thread-specific data, Thread Local Storage (TLS)
 - Not local, but global kind of data for all functions of a thread, more like “static” data
 - Create Facility needed for data private to thread
 - Allows each thread to have its own copy of data
 - Useful when you do not have control over the thread creation process (i.e., when using a thread pool)
 - gcc compiler provides the storage class keyword `thread` for declaring TLS data

```
static __thread int threadID;
```

```
int arr[16];  
int f() {  
    a(); b(); c();  
}  
int g() {  
    x(); y();  
}  
int main() {  
    th_create(...,f,...);  
    th_create(...,g,...);  
}  
//arr is visible to all of them!  
//need data for only f,a,b,c  
//need data for only g,x,y
```

Scheduler activations for threads

Library

```
--  
th_setup(int n) {  
    max_threads = n;  
    curr_threads = 0;  
}  
th_create(..., fn,...) {  
    if(curr_threads < max_threads)  
        create kernel thread;  
    schedule fn on one of the kernel  
    threads;  
}
```

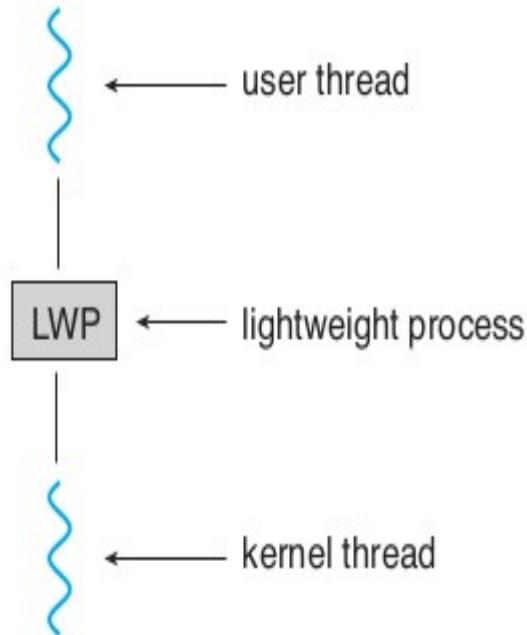
application

```
---  
f() {  
    scanf();  
}  
g() {  
    recv();  
}  
h() {...}; i() {...}  
main()  
    th_setup(2);  
    th_create(...,f,...);  
    th_create(...,g,...);  
    th_create(...,h,...);  
    th_create(...,i,...);  
}
```

Scheduler activations for threads

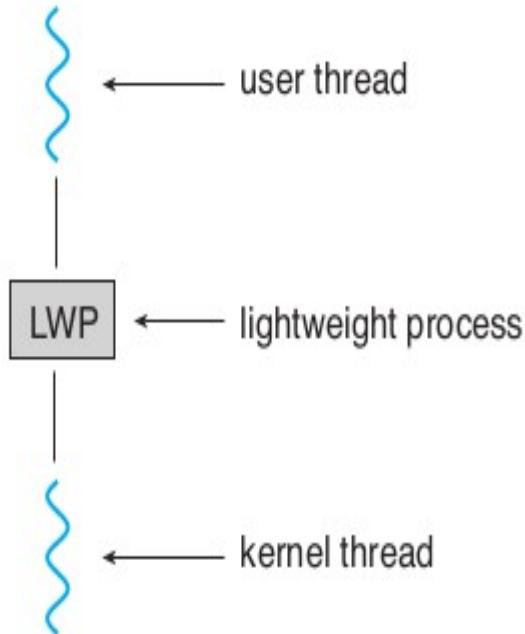
- **Scheduler Activations**
 - Both M:M and Two-level models require communication to maintain the appropriate number of kernel threads allocated to the application
 - Scheduler activations provide upcalls - a communication mechanism from the kernel to the thread library
 - This communication allows an application to maintain the correct number kernel threads

Issues with threads



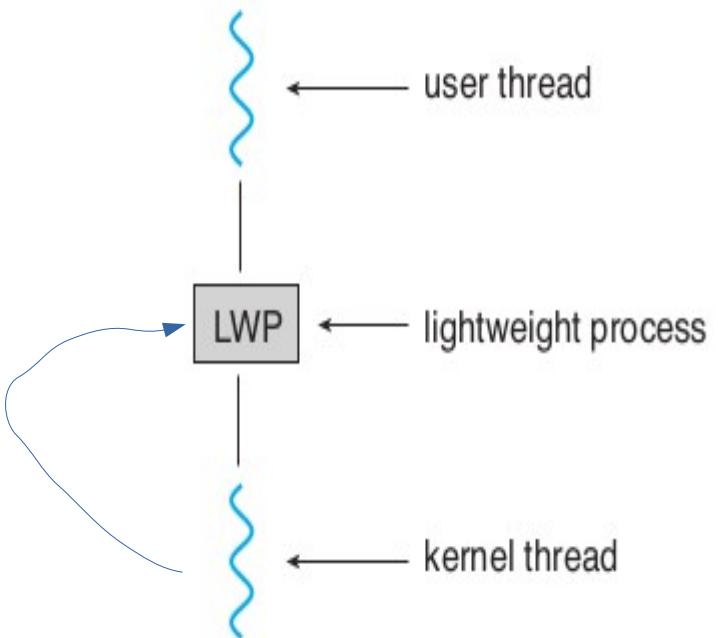
- **Scheduler Activations: LWP approach**
 - An intermediate data structure LWP
 - appears to be a virtual processor on which the application can schedule a user thread to run.
 - Each LWP attached to a kernel thread
 - Typically one LWP per blocking call, e.g. 5 file I/Os in one process, then 5 LWPs needed

Issues with threads



- **Scheduler Activations: LWP approach**
 - Kernel needs to inform application about events like: a thread is about to block, or wait is over
 - This will help application relinquish the LWP or request a new LWP

Issues with threads



- **The actual upcalls**

Linux threads

- Only threads (called task), no processes!
- Process is a thread that shares many particular resources with the parent thread
- `Clone()` system call to create a thread

Linux threads

- **clone() takes options to determine sharing on process create**
- **struct task_struct points to process data structures (shared or unique depending on clone options)**
- **fork() is a wrapper on top of clone()**

| flag | meaning |
|---------------|------------------------------------|
| CLONE_FS | File-system information is shared. |
| CLONE_VM | The same memory space is shared. |
| CLONE_SIGHAND | Signal handlers are shared. |
| CLONE_FILES | The set of open files is shared. |

Issues in implementing threads project

- How to implement a user land library for threads?
- How to handle 1-1, many-one, many-many implementations?
- Identifying the support required from OS and hardware
- Identifying the libraries that will help in implementation

Issues in implementing threads project

- **Understand the clone() system call completely**
 - Try out various possible ways of calling it
 - Passing different options
 - Passing a user-land buffer as stack
- **How to save and restore context?**
 - C: setjmp, longjmp
 - Setcontext, getcontext(), makecontext(), swapcontext() functions
- **Sigaction is more powerful than signal**
 - Learn SIGALRM handling for timer and scheduler, timer_create() & timer_stop() system calls
- **Customized data structure to store threads, and manage thread-lists for scheduling**

Signals

Signals

- Signals are used in UNIX systems to notify a process that a particular event has occurred.
- Signal handling
 - Synchronous and asynchronous
- A signal handler (a function) is used to process signals
 - Signal is generated by particular event
 - Signal is delivered to a process
 - Then, signal is “handled” by the handler

Signals

- **More about signals**
 - Different signals are typically identified as different numbers
 - Operating systems provide system calls like `kill()` and `signal()` to enable processes to deliver and receive signals
 - `Signal()` - is used by a process to specify a “signal handler” – a code that should run on receiving a signal
 - `Kill()` is used by a process to send another process a signal
 - There are restrictions on which process can send which signal to other processes

Demo

- Let's see a demo of signals with respect to processes
- Let's see signal.h
 - /usr/include/signal.h
 - /usr/include/asm-generic/signal.h
 - /usr/include/linux/signal.h
 - /usr/include/sys/signal.h
 - /usr/include/x86_64-linux-gnu/asm/signal.h
 - /usr/include/x86_64-linux-gnu/sys/signal.h
- man 7 signal
- Important signals: SIGKILL, SIGUSR1, SIGSEGV, SIGALRM, SIGCLD, SIGINT, SIGPIPE, ...

Signal handling by OS

```
Process 12323 {  
    signal(19, abcd);  
}
```

```
OS: sys_signal {  
    Note down that process 12323  
    wants to handle signal number  
    19 with function abcd  
}
```

```
Process P1 {  
    kill (12323, 19) ;  
}
```

```
OS: sys_kill {
```

Note down in PCB of process 12323 that signal number 19 is pending for you.

When process 12323 is scheduled, at that time the OS will check for pending signals, and invoke the appropriate signal handler for a pending signal.

Threads and Signals

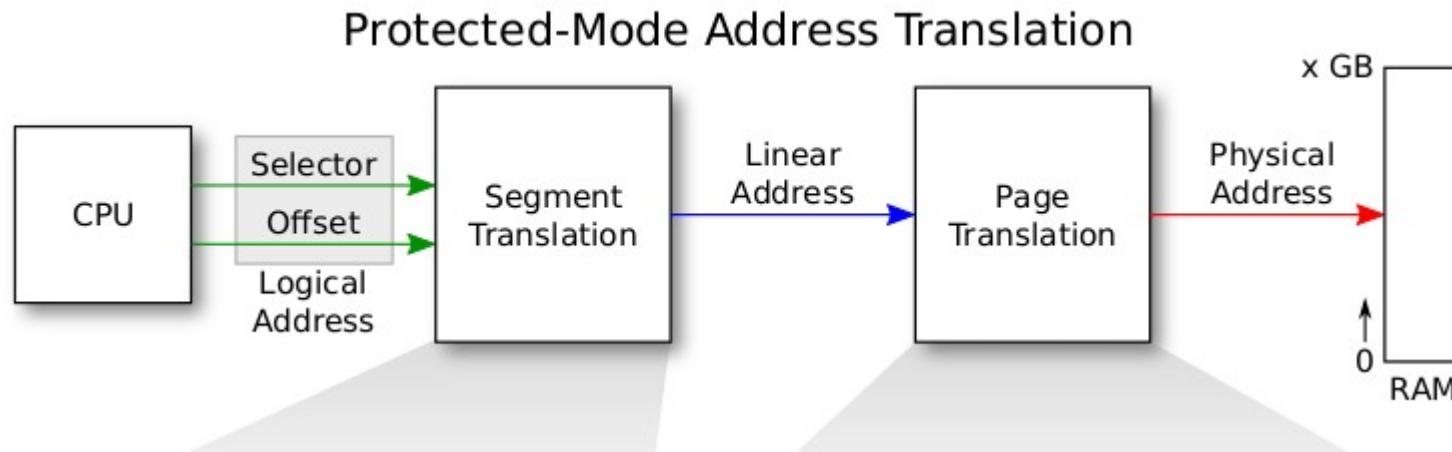
- **Signal handling Options:**
 - Deliver the signal to the thread to which the signal applies
 - Deliver the signal to every thread in the process
 - Deliver the signal to certain threads in the process
 - Assign a specific thread to receive all signals for the process

Some numbers and their ‘meaning’

- These numbers occur very frequently in discussion
- $0x\ 80000000 = 2\ GB = \text{KERNBASE}$
- $0x\ 100000 = 1\ MB = \text{EXTMEM}$
- $0x\ 80100000 = 2GB + 1MB = \text{KERNLINK}$
- $0x\ E000000 = 224\ MB = \text{PHYSTOP}$
- $0x\ FE000000 = 3.96\ GB = 4064\ MB = \text{DEVSPACE}$
 - $4096 - 4064 = 32\ MB$ left on top

X86 Memory Management

X86 address : protected mode address translation



X86 paging

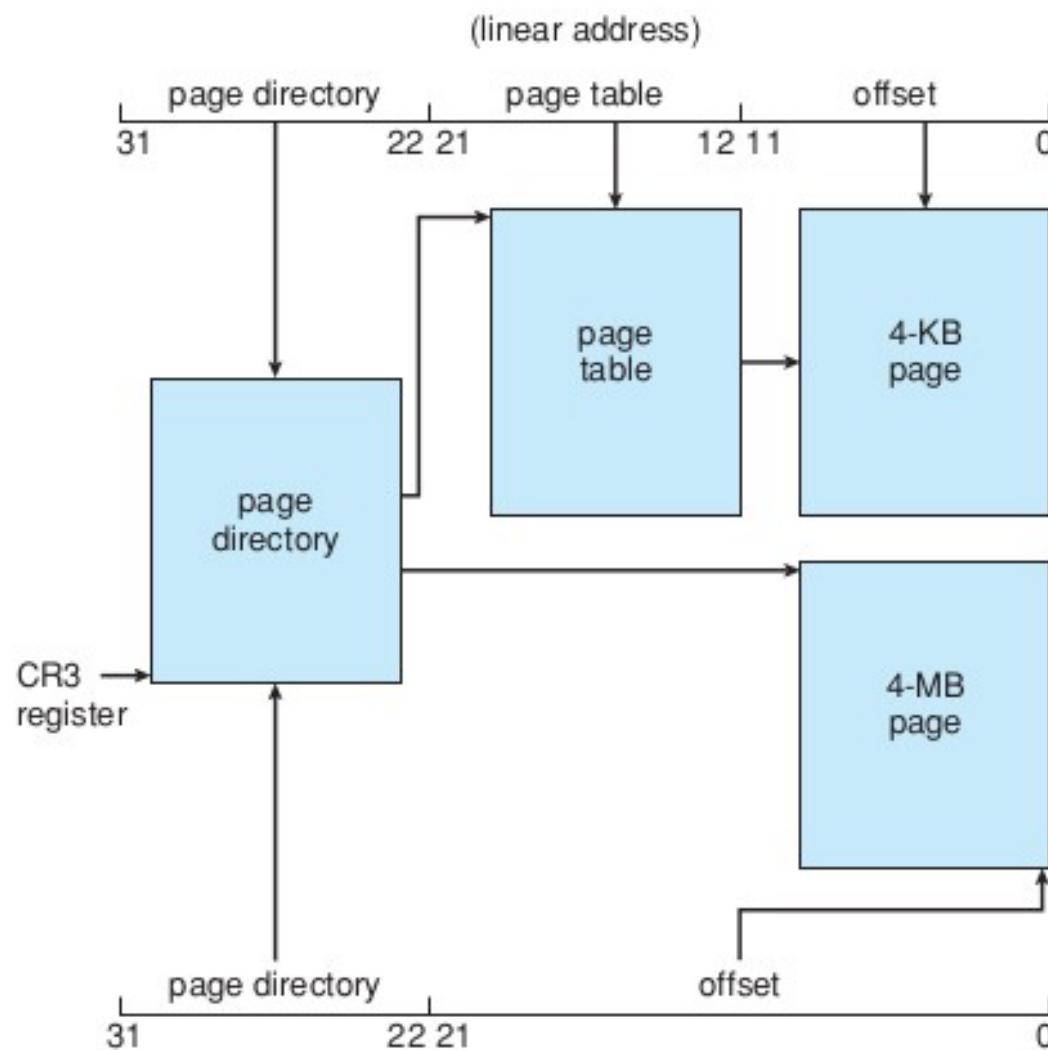
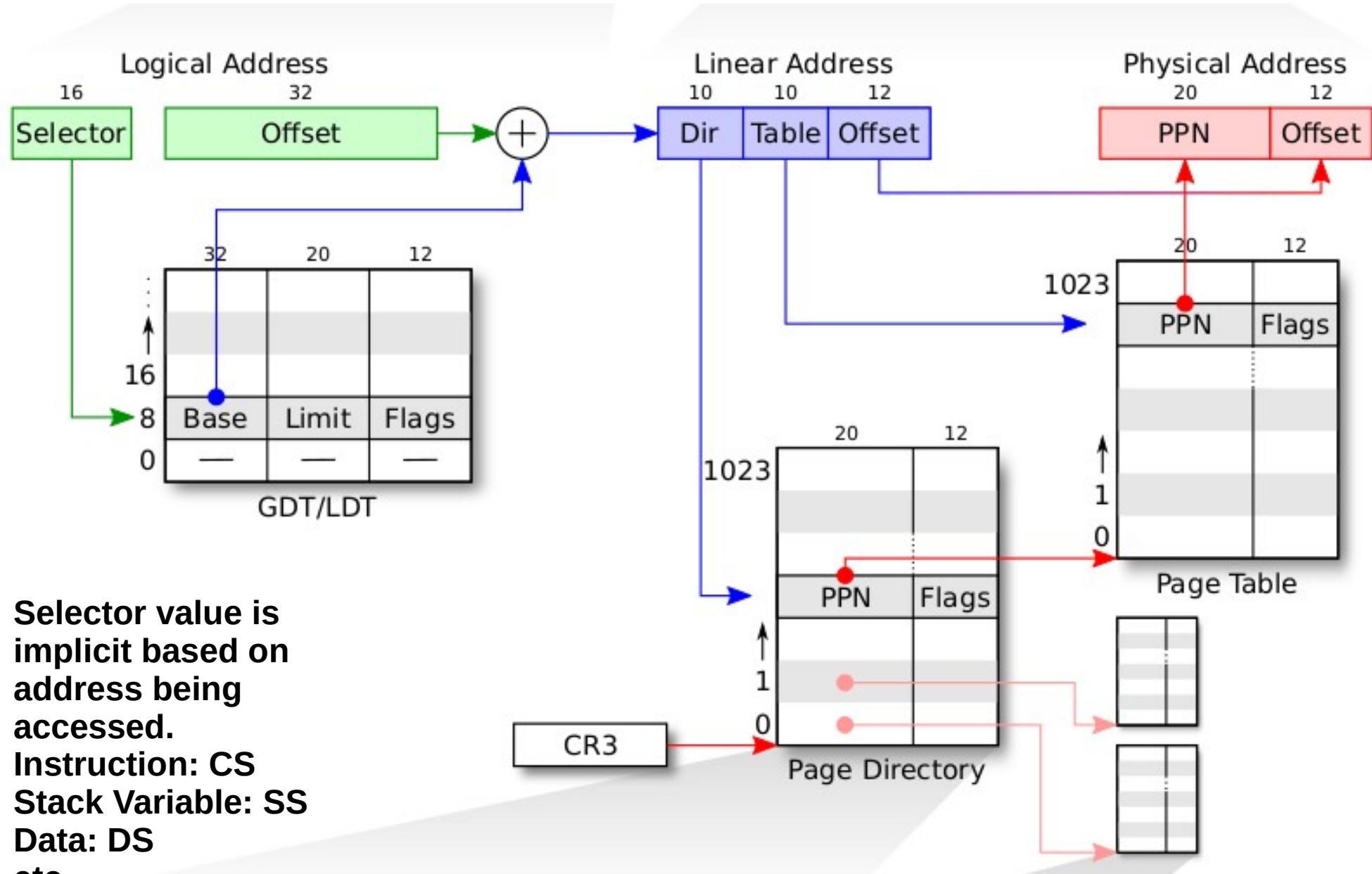


Figure 8.23 Paging in the IA-32 architecture.

Segmentation + Paging



GDT Entry

| | | | |
|---------------------|-------|-------------------------|------------------------------------|
| 31 | 16 | 15 | 0 |
| Base 0:15 _____ | | Limit 0:15 _____ | |
| 63 | 56 | 55 52 | 51 48 47 40 39 32 |
| Base 24:31 _____ | Flags | Limit 16:19 _____ | Access Byte Base 16:23 _____ |

Page Directory Entry (PDE) Page Table Entry (PTE)

31

| Page table physical page number | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|---------------------------------|-------------|--------|--------|---|--------|--------|--------|--------|--------|--------|--------|--------|---|
| | A V L | G S | P 0 | A | C D | W T | U D | W T | U D | W T | U D | W T | P |

PDE

P Present

W Writable

U User

WT 1=Write-through, 0=Write-back

CD Cache disabled

A Accessed

D Dirty

PS Page size (0=4KB, 1=4MB)

PAT Page table attribute index

G Global page

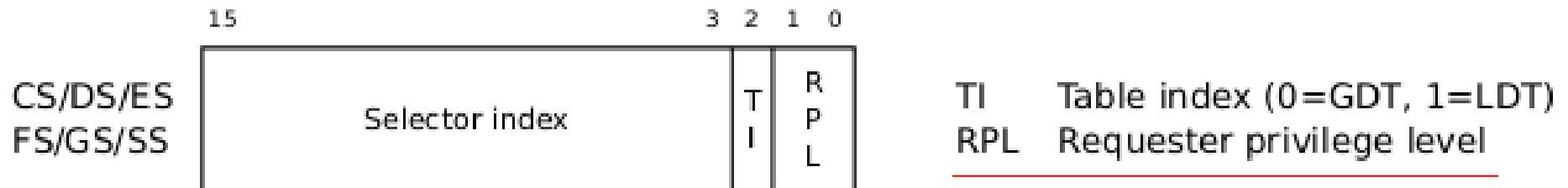
AVL Available for system use

31

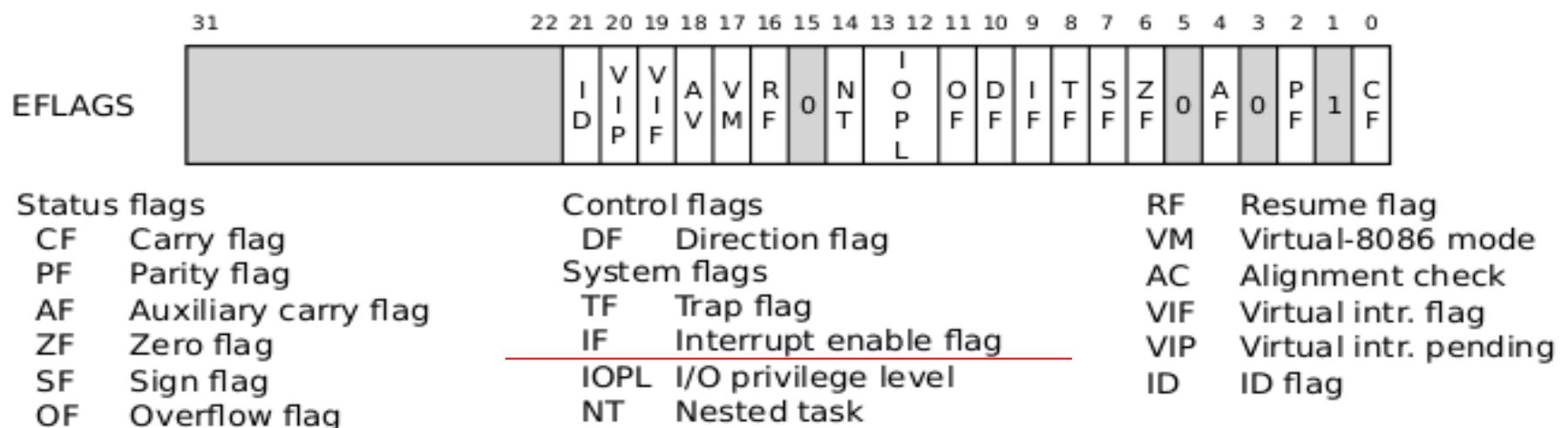
| Physical page number | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|----------------------|-------------|--------|--------|--------|--------|--------|--------|--------|--------|--------|--------|--------|---|
| | A V L | G T | P A | D A | A D | C D | W T | U D | W T | U D | W T | U D | P |

PTE

Segment selector



EFLAGS register



CR0

| CR0 | 31 30 29 28 | 19 18 17 16 15 | 6 5 4 3 2 1 0 |
|-----|---------------------|----------------|------------------------------|
| | P C N G D W | A M W P | N E T E M P P E T S M P E |
| PE | Protection enabled | ET | Extension type |
| MP | Monitor coprocessor | NE | Numeric error |
| EM | Emulation | WP | Write protect |
| TS | Task switched | AM | Alignment mask |

PG: Paging enabled or not

WP: Write protection on/off

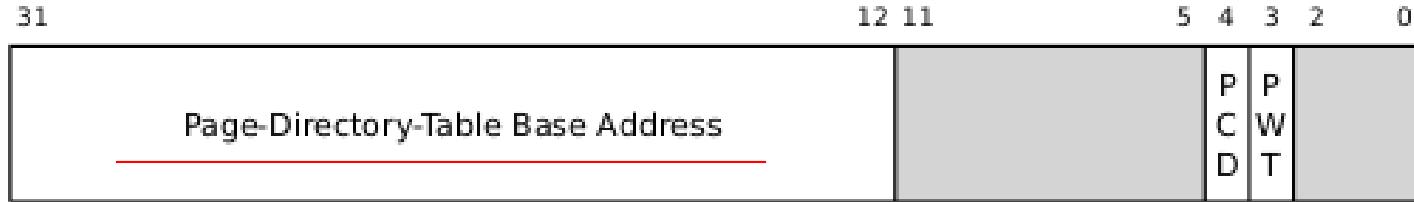
PE: Protection Enabled --> protected mode.

CR2

| CR2 | 31 | 0 |
|-----|----|----------------------------|
| | | Page fault virtual address |

CR3

CR3



PWT Page-level writes transparent

PCT Page-level cache disable

CR4

CR4



VME Virtual-8086 mode extensions

PVI Protected-mode virtual interrupts

TSD Time stamp disable

DE Debugging extensions

PSE Page size extensions

PAE Physical-address extension

MCE Machine check enable

PGE Page-global enable

PCE Performance counter enable

OSFXSR OS FXSAVE/FXRSTOR support

OSXMM- OS unmasked exception support

EXCPT

mmu.h : paging related macros

```
#define PTXSHIFT          12      // offset of PTX in a linear address
#define PDXSHIFT           22      // offset of PDX in a linear address
#define PDX(va)    (((uint)(va) >> PDXSHIFT) & 0x3FF) // page directory index
#define PTX(va)    (((uint)(va) >> PTXSHIFT) & 0x3FF) // page table index
// construct virtual address from indexes and offset
#define PGADDR(d, t, o) ((uint)((d) << PDXSHIFT | (t) << PTXSHIFT | (o)))
// +-----10-----+-----10-----+-----12-----+
// | Page Directory | Page Table | Offset within Page |
// |     Index       |     Index   |                         |
// +-----+-----+-----+
// \--- PDX(va) --/ \--- PTX(va) --/
```

mmu.h : paging related macros

```
// Page directory and page table constants.  
  
#define NPDENTRIES          1024    // # directory  
entries per page directory  
  
#define NPTENTRIES          1024    // # PTEs per  
page table  
  
#define PGSIZE              4096    // bytes mapped  
by a page  
  
#define PGROUNDUP(sz) (( (sz)+PGSIZE-1) &  
~(PGSIZE-1))  
  
#define PGROUNDDOWN(a) (((a)) & ~(PGSIZE-1))
```

mmu.h : paging related macros

```
// Page table/directory entry flags.
```

```
#define PTE_P          0x001 // Present
#define PTE_W          0x002 // Writeable
#define PTE_U          0x004 // User
#define PTE_PS         0x080 // Page Size
```

```
// Address in page table or page directory entry
```

```
#define PTE_ADDR(pte) ((uint)(pte) & ~0xFF) // get  
all but last 12 bits
```

```
#define PTE_FLAGS(pte) ((uint)(pte) & 0xFF) // get  
last 12 bits
```

mmu.h : Segmentation related macros

// various segment selectors.

#define SEG_KCODE 1 // kernel code

#define SEG_KDATA 2 // kernel data+stack

#define SEG_UCODE 3 // user code

#define SEG_UDATA 4 // user data+stack

#define SEG_TSS 5 // this process's task state

mmu.h : Segmentation related macros

// various segment selectors.

```
#define SEG_KCODE 1 // kernel code
```

```
#define SEG_KDATA 2 // kernel data+stack
```

```
#define SEG_UCODE 3 // user code
```

```
#define SEG_UDATA 4 // user data+stack
```

```
#define SEG_TSS 5 // this process's task state
```

```
#define NSEGS 6
```

mmu.h : Segmentation related macros

```
struct segdesc { // 64 bit in size
    uint lim_15_0 : 16; // Low bits of segment limit
    uint base_15_0 : 16; // Low bits of segment base address
    uint base_23_16 : 8; // Middle bits of segment base address
    uint type : 4;      // Segment type (see STS_ constants)
    uint s : 1;         // 0 = system, 1 = application
    uint dpl : 2;       // Descriptor Privilege Level
    uint p : 1;         // Present
    uint lim_19_16 : 4; // High bits of segment limit
    uint avl : 1;       // Unused (available for software use)
    uint rsv1 : 1;      // Reserved
    uint db : 1;        // 0 = 16-bit segment, 1 = 32-bit segment
    uint g : 1;          // Granularity: limit scaled by 4K when set
    uint base_31_24 : 8; // High bits of segment base address
};
```

mmu.h : Segmentation related code

// Application segment type bits

```
#define STA_X      0x8    // Executable segment  
#define STA_W      0x2    // Writeable (non-executable  
segments)  
#define STA_R      0x2    // Readable (executable  
segments)
```

// System segment type bits

```
#define STS_T32A   0x9    // Available 32-bit TSS  
#define STS_IG32    0xE    // 32-bit Interrupt Gate  
#define STS_TG32    0xF    // 32-bit Trap Gate
```

Code from bootasm.S bootmain.c is over!
Kernel is loaded.
Now kernel is going to prepare itself

main() in main.c

- **Initializes “free list” of page frames**
 - In 2 steps. Why?
- **Sets up page table for kernel**
- **Detects configuration of all processors**
- **Starts all processors**
 - Just like the first processor
- **Creates the first process!**
- **Initializes**
 - LAPIC on each processor, IOAPIC
 - Disables PIC
 - “Console” hardware (the standard I/O)
 - Serial Port
 - Interrupt Descriptor Table
 - Buffer Cache
 - Files Table
 - Hard Disk (IDE)

main() in main.c

```
int  
  
main(void) {  
    kinit1(end,  
P2V(4*1024*1024)); // phys  
page allocator  
  
    kvmalloc();      // kernel page  
table  
  
    void  
  
    kinit1(void *vstart, void  
*vend) {  
        initlock(&kmem.lock,  
"kmem");  
  
        kmem.use_lock = 0;  
  
        freerange(vstart, vend);  
    }  
}
```

main() in main.c

void

freerange(void *vstart, void

*vend)

{

char *p;

p =

(char*)PGROUNDUP((uint)vsta
rt);

for(; p + PGSIZE <=
(char*)vend; p += PGSIZE)

kfree(p);

}

```
kfree(char *v) {  
    struct run *r;  
    if((uint)v % PGSIZE || v <  
end || V2P(v) >= PHYSTOP)  
        panic("kfree");  
    // Fill with junk to catch  
dangling refs.  
    memset(v, 1, PGSIZE);  
    if(kmem.use_lock)  
        acquire(&kmem.lock);  
    r = (struct run*)v;  
    r->next = kmem.freelist;  
    kmem.freelist = r;  
    if(kmem.use_lock)  
        release(&kmem.lock); }
```

lock
uselock
run *freelist

run *next

Allocated frame

run *next

Allocated frame

Allocated frame

run *next

run *next

Allocated frame

RAM –
divided
into frames

Free List in XV6 Obtained after main() -> kinit1()

Pages obtained Between
end = 801154a8 = 2049 MB to **P2V(4MB) = 2052 MB**
Remember
Right now Logical = Physical address.

Actually like
this in memory

lock
uselock
run *freelist

kmem

Seen
independently

run *

run *

run *

Back to main()

int

main(void) {

kinit1(end,
P2V(4*1024*1024)); //
phys page allocator

kvmalloc(); //
kernel page table

// Allocate one page
table for the machine
for the kernel address
// space for scheduler
processes.

void

kvmalloc(void)
{

kpgdir = setupkvm();
switchkvm();

}

Back to main()

```
int  
main(void) {
```

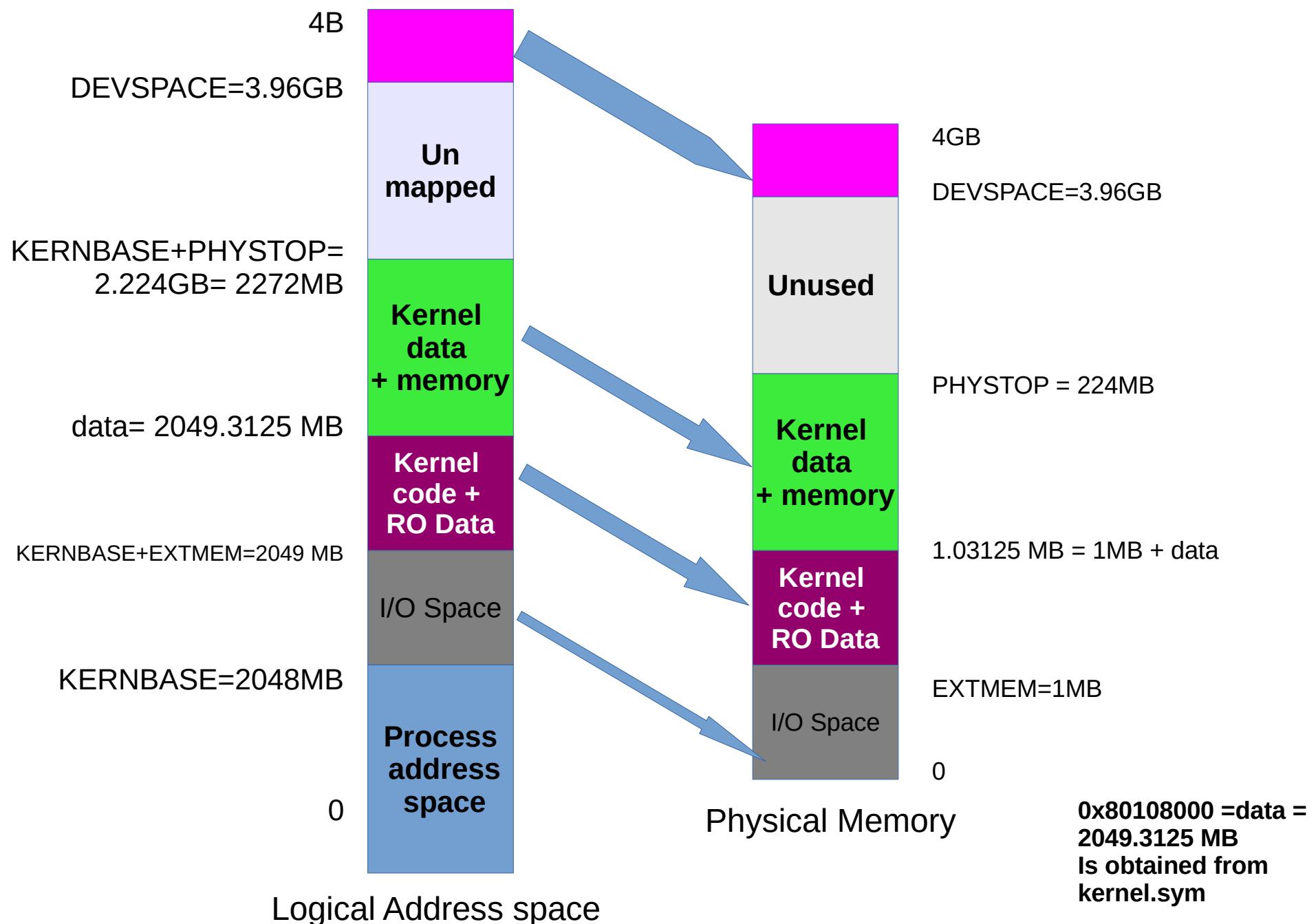
```
    kinit1(end,  
P2V(4*1024*1024)); //  
phys page allocator  
    kvmalloc();    //  
kernel page table
```

```
// Allocate one page  
table for the machine  
for the kernel address  
  
// space for scheduler  
processes.  
  
void  
kvmalloc(void)  
{  
    kpgdir =  
setupkvm(); // global  
var kpgdir  
  
switchkvm();  
}
```

```
pde_t*
setupkvm(void)
{
    pde_t *pgdir;
    struct kmap *k;
    if((pgdir = (pde_t*)kalloc())
== 0)
        return 0;
    memset(pgdir, 0, PGSIZE);
    if (P2V(PHYSTOP) >
(void*)DEVSPACE)
        panic("PHYSTOP too
high");
    for(k = kmap; k <
&kmap[NELEM(kmap)];
k++)
        if(mappages(pgdir, k-
>virt, k->phys_end - k-
>phys_start,
            (uint)k-
>phys_start, k->perm) <
0) {
            freevm(pgdir);
            return 0;
        }
    return pgdir;
}
```

```
static struct kmap {
    void *virt;
    uint phys_start;
    uint phys_end;
    int perm;
} kmap[] = {
{ (void*)KERNBASE, 0,          EXTMEM,  PTE_W}, // I/O space
{ (void*)KERNLINK, V2P(KERNLINK), V2P(data), 0}, // kern text+rodata
{ (void*)data,   V2P(data),   PHYSTOP, PTE_W}, // kern data+memory
{ (void*)DEVSPACE, DEVSPACE, 0,      PTE_W}, // more devices
};
```

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



Reminder: PDE and PTE entries

31

| | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|---------------------------------|-------------|--------|--------|--------|--------|--------|--------|--------|---|---|---|---|---|
| Page table physical page number | A V L | G S | P 0 | 0 A | C D | W T | U U | W P | | | | | |

PDE

P Present

W Writable

U User

WT 1=Write-through, 0=Write-back

CD Cache disabled

A Accessed

D Dirty

PS Page size (0=4KB, 1=4MB)

PAT Page table attribute index

G Global page

AVL Available for system use

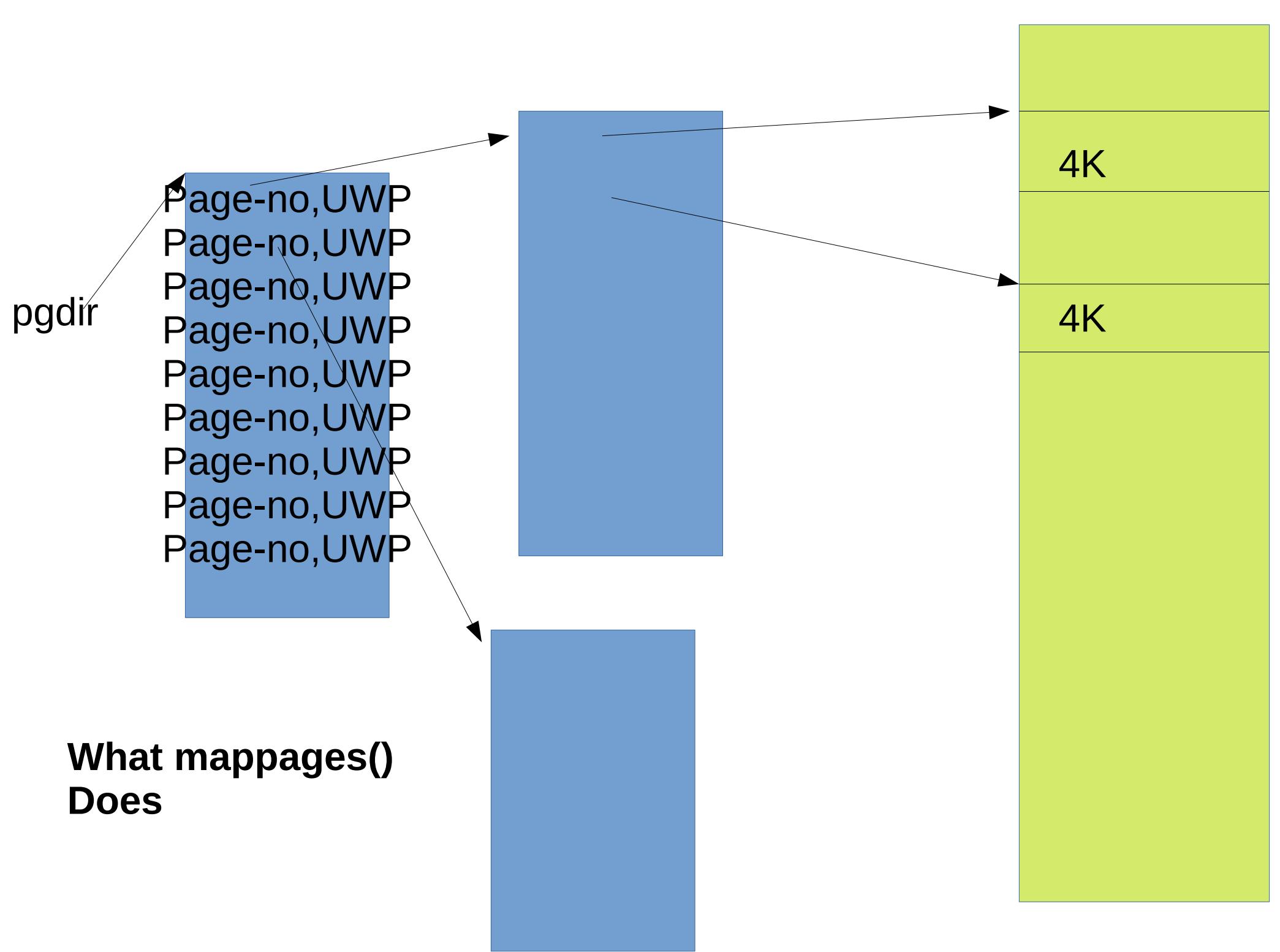
31

| | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|----------------------|-------------|-------------|-------------|--------|--------|--------|--------|--------|---|---|---|---|---|
| Physical page number | A V L | G A T | P D T | 0 A | C D | W T | U U | W P | | | | | |

PTE

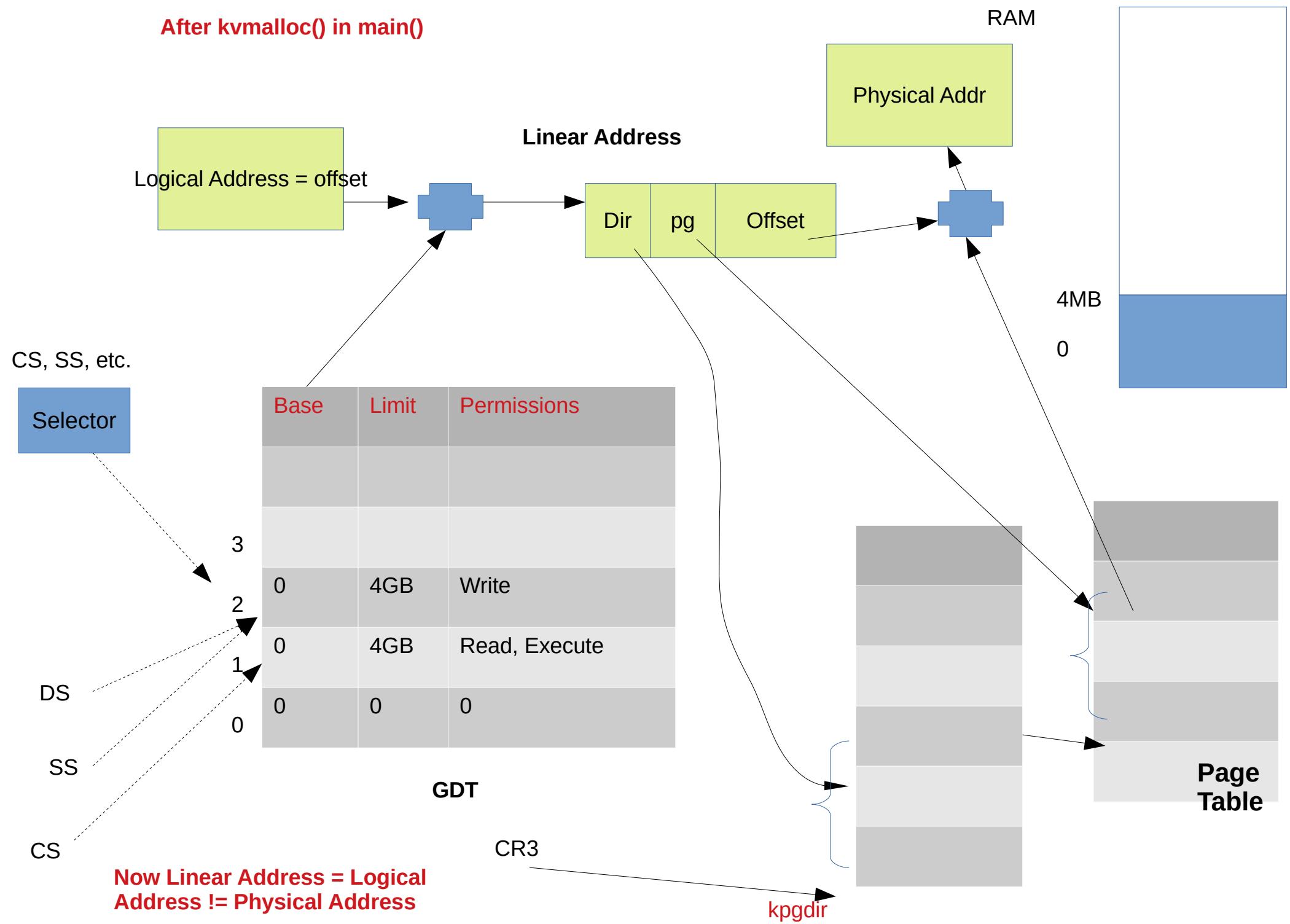
Before mappages()





**What mappages()
Does**

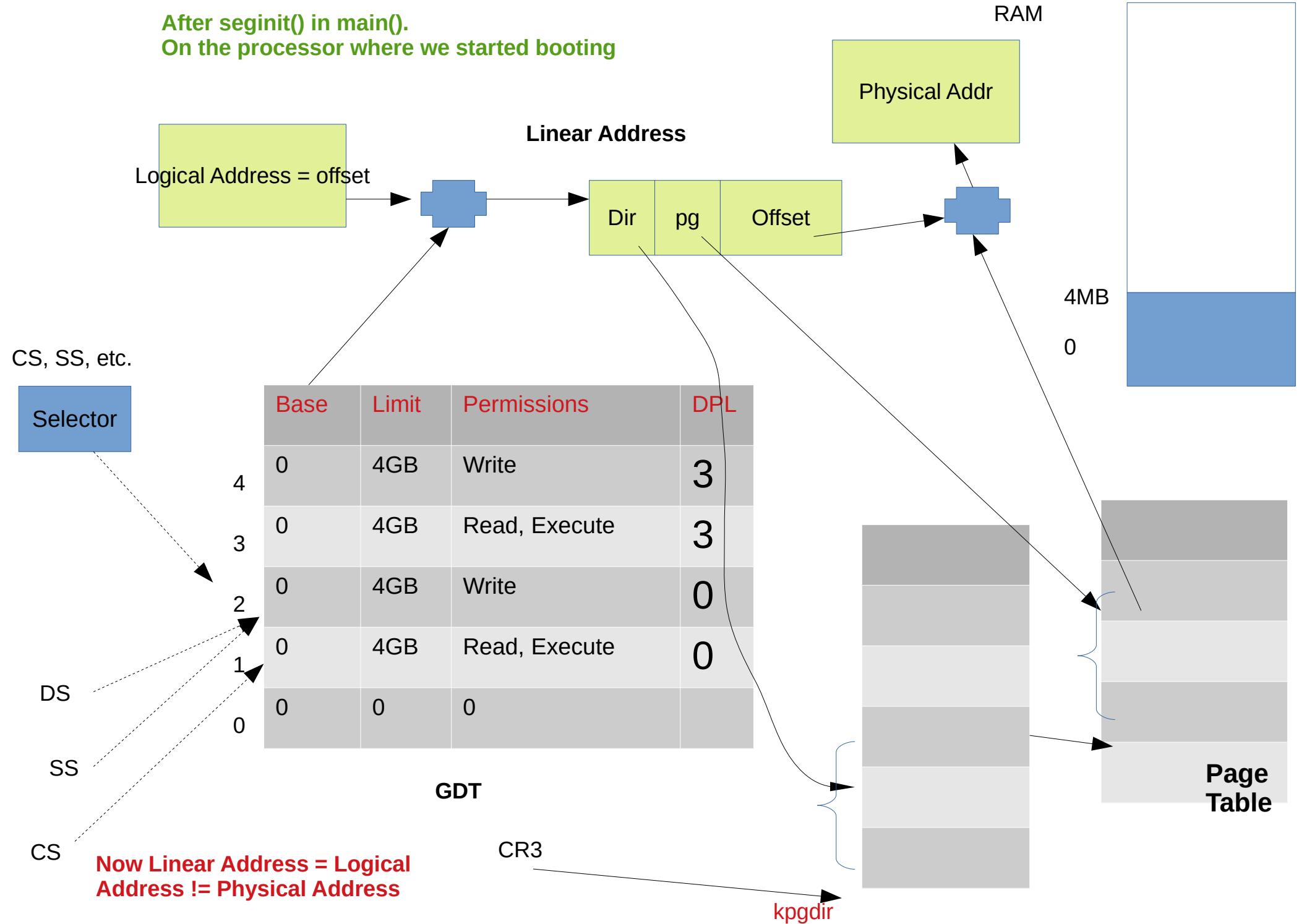
After kvmalloc() in main()



main() -> seginit()

- **Re-initialize GDT**
- **Once and forever now**
- **Just set 4 entries**
 - All spanning 4 GB
 - Differing only in permissions and privilege level

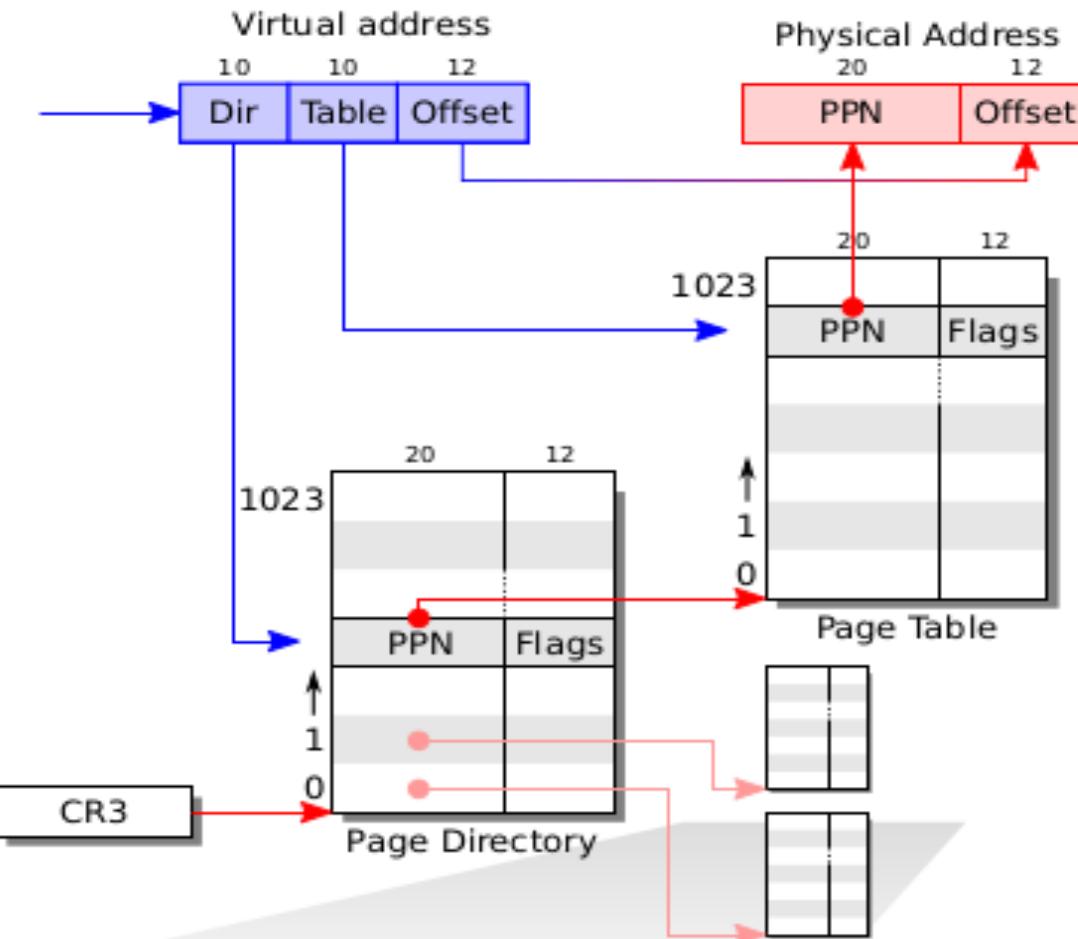
After seginit() in main().
On the processor where we started booting



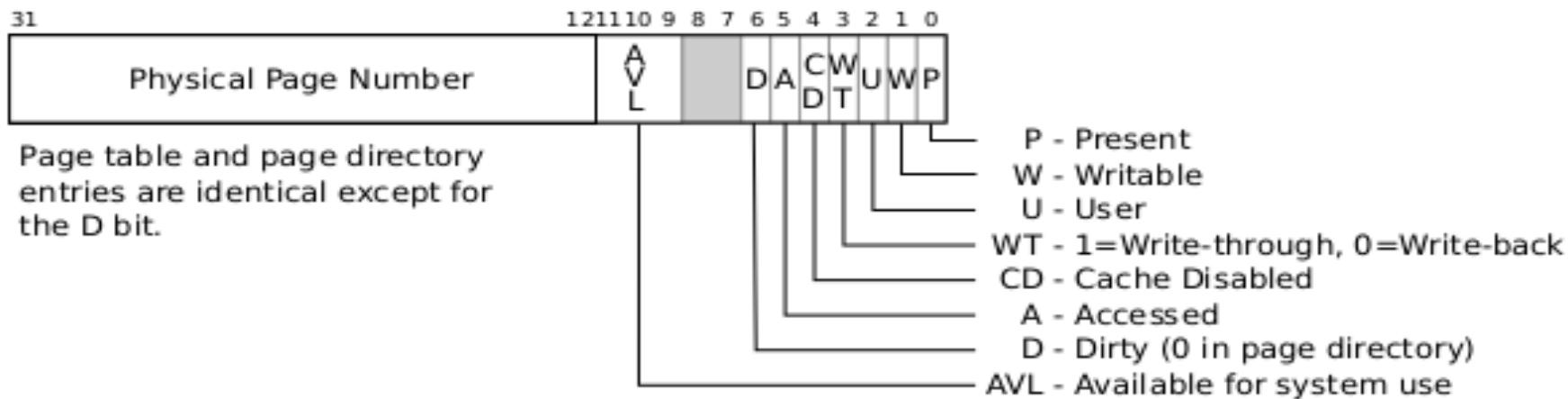
After seginit()

- While running kernel code, necessary to switch CS, DS, SS to index 1,2,2 in GDT
- While running user code, necessary to switch CS, DS, SS to index 3,4,4 in GDT
- This happens automatically as part of “trap” handling (covered separately)

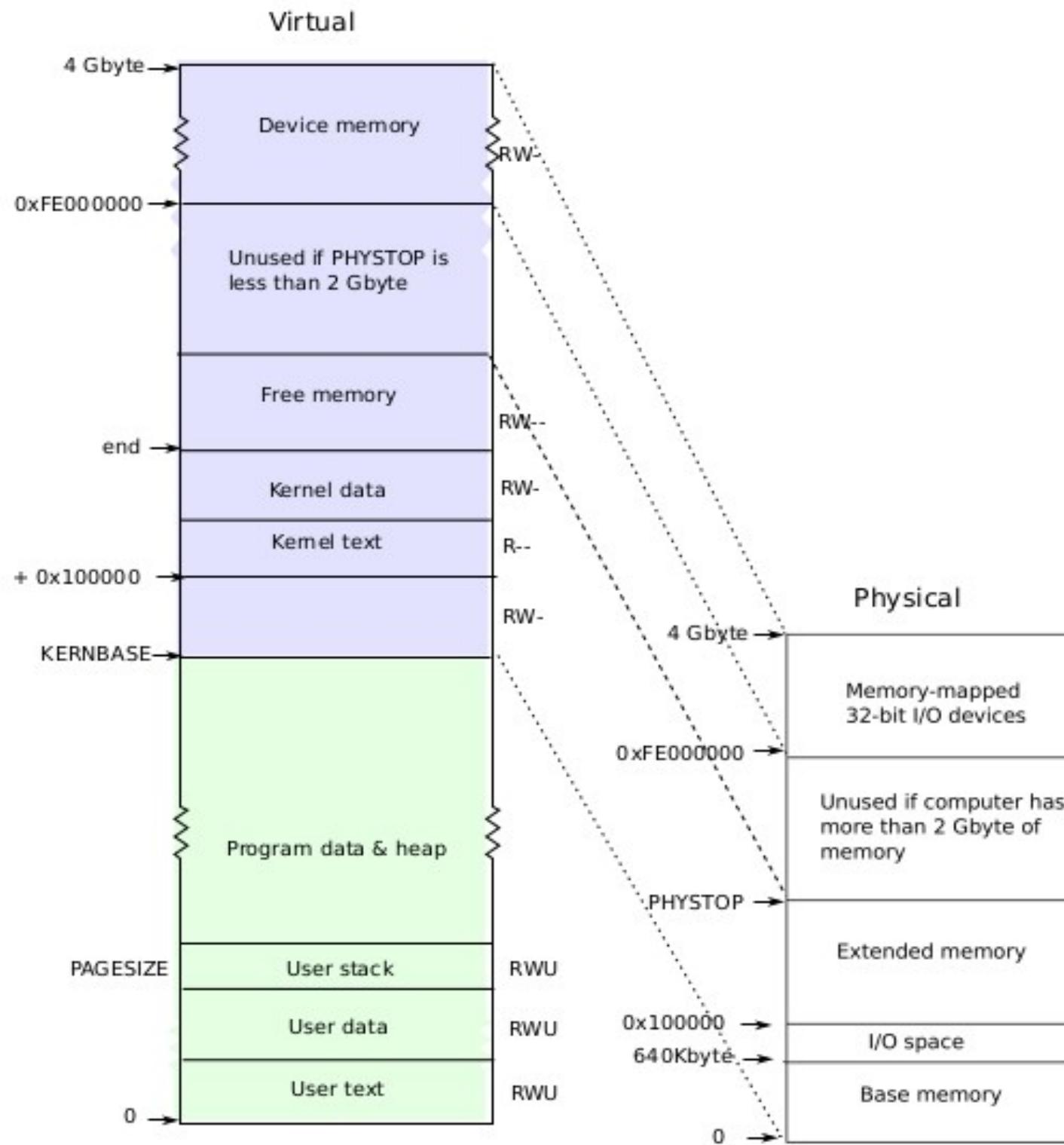
Memory Management



X86 page table hardware



Layout of process's VA space



Memory Layout of a user process

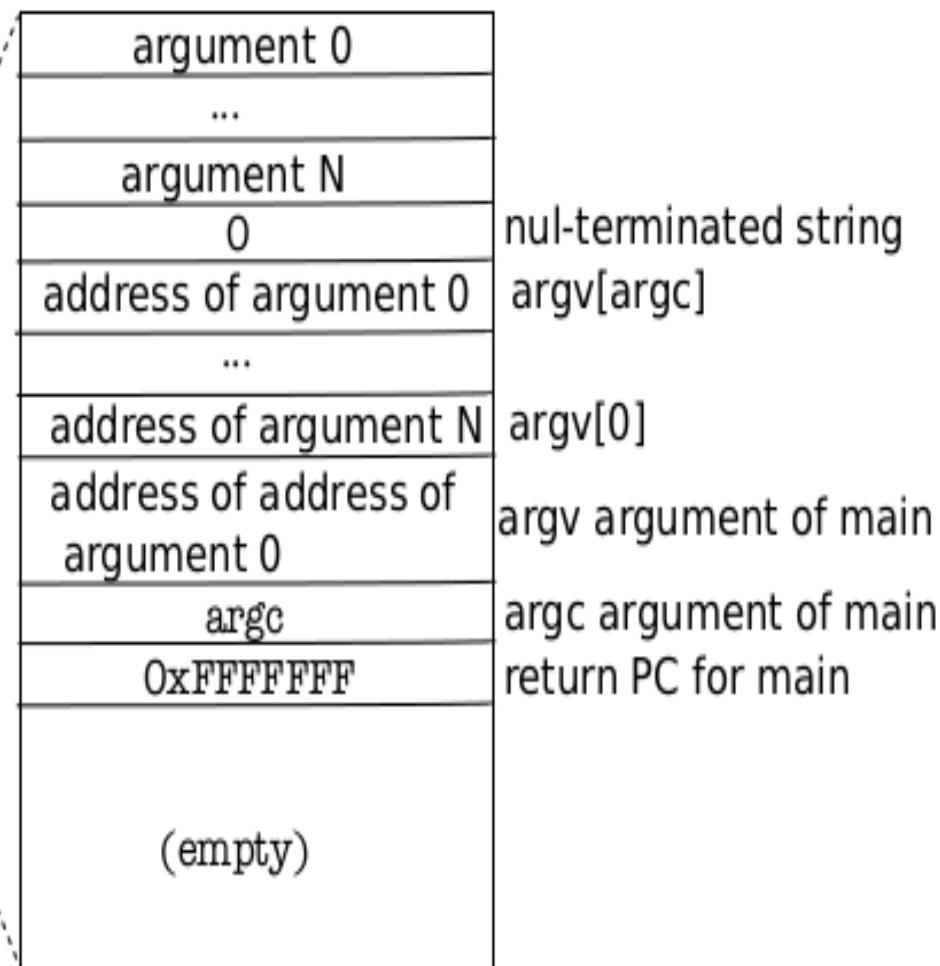
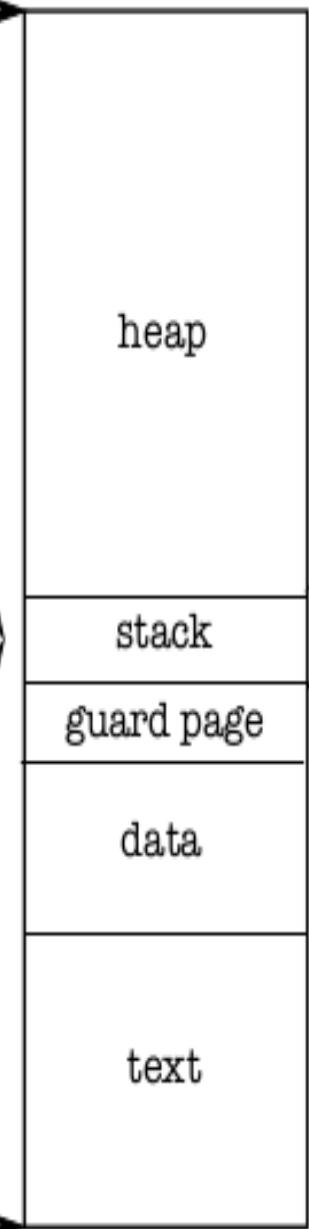
After exec()

Note the argc, argv on stack

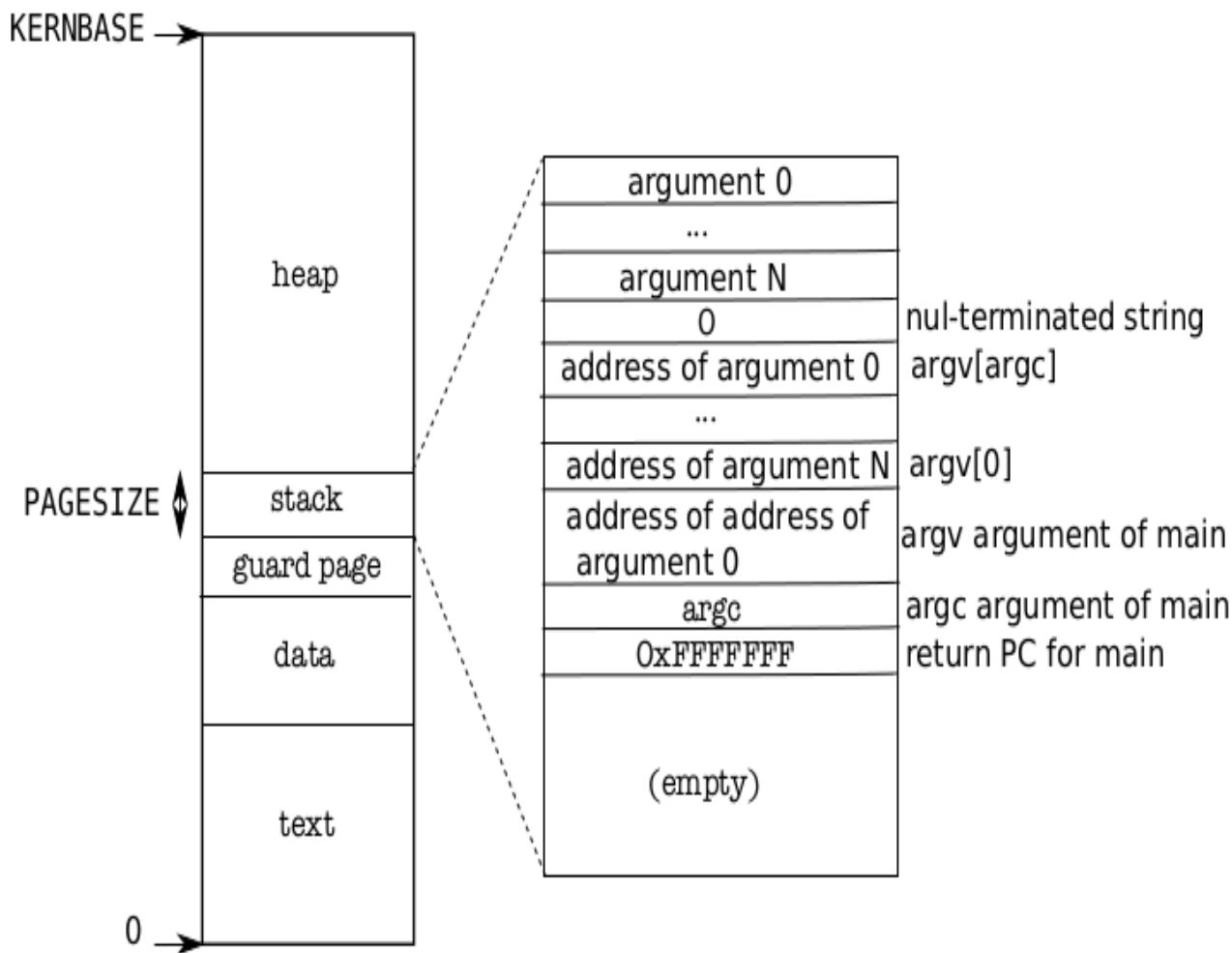
KERNBASE

PAGESIZE

0

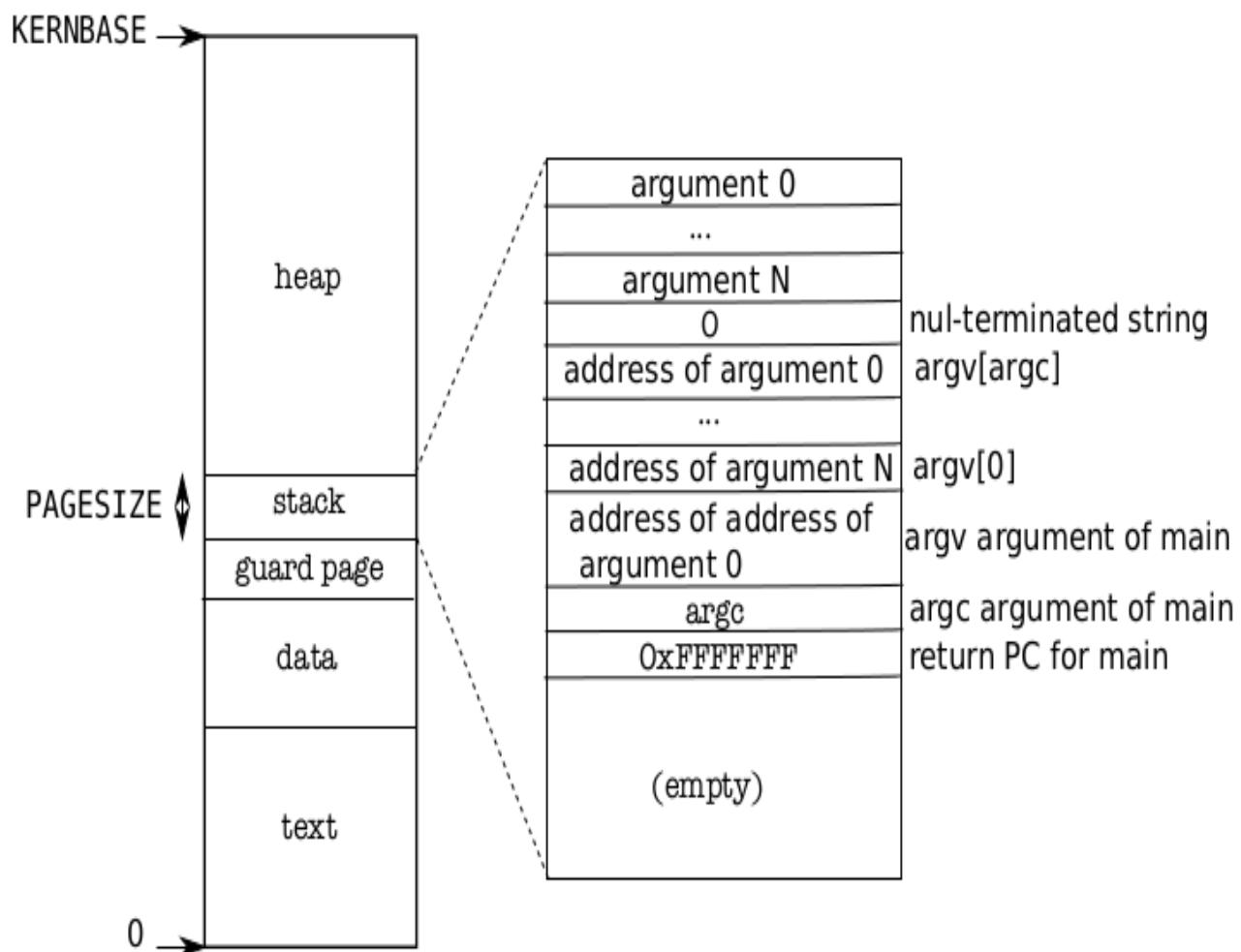


Memory Layout of a user process



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

Memory Layout of a user process



On sbrk()

The system call to grow process's address space.
Calls growproc()

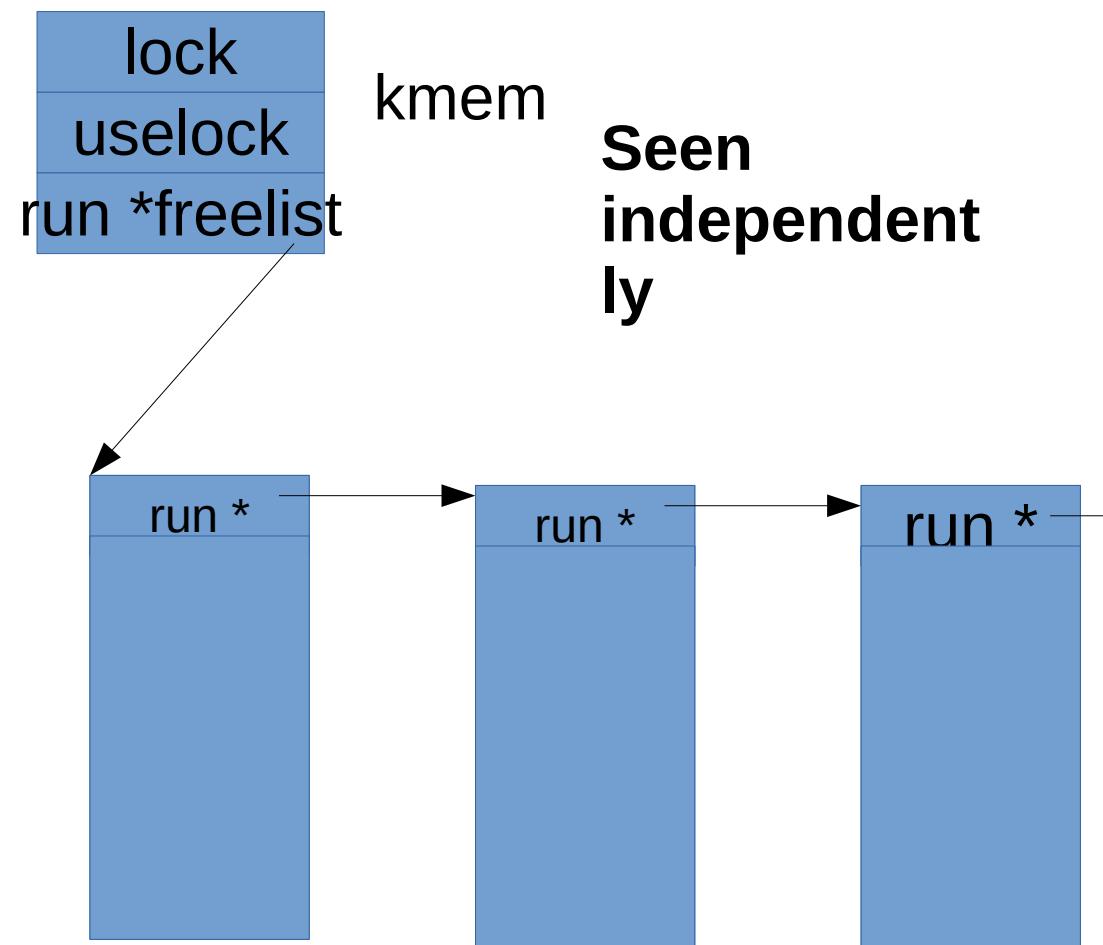
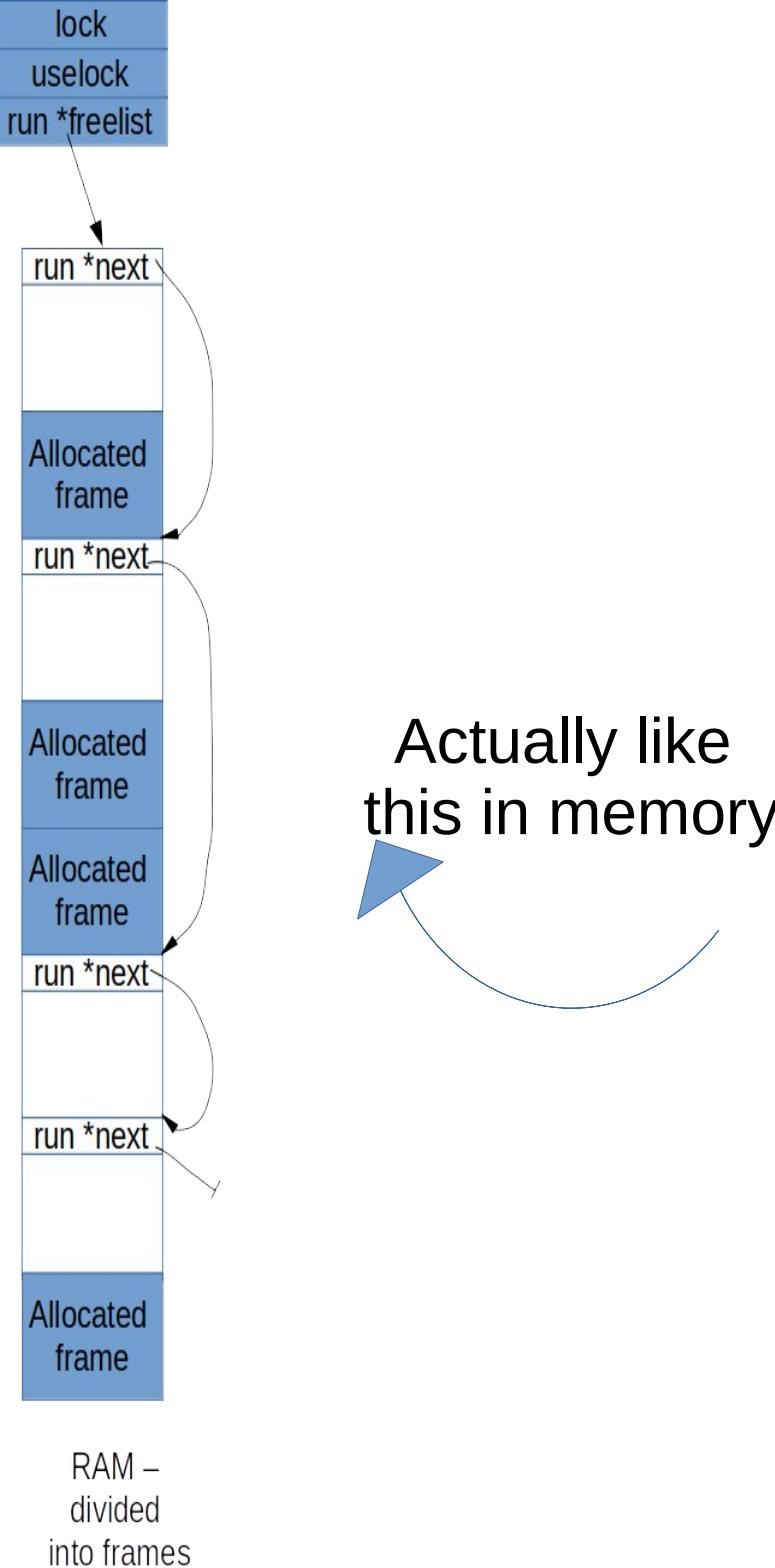
growproc()

Allocate a frame, Add an entry in page table at the top (above proc->sz)
//This entry can't go beyond KERNBASE
Calls switchuvvm()

Switchuvvm()

Ultimately loads CR3, invalidating cache

Free List in XV6



exec()

- **sys_exec()**

- exec(path, argv)**

- **exec(parth, argv)**

- ip = namei(path))**

- readi(ip, (char*)&elf, 0, sizeof(elf)) != sizeof(elf)**

- for(i=0, off=elf.phoff; i<elf.phnum; i++, off+=sizeof(ph)){**

- if(readi(ip, (char*)&ph, off, sizeof(ph)) != sizeof(ph))**

- if((sz = allocuvm(pgdir, sz, ph.vaddr + ph.memsz)) == 0)**

- if(loaduvm(pgdir, (char*)ph.vaddr, ip, ph.off, ph.filesz) < 0)**

- }**

exec()

- exec(part, argv)

// Allocate two pages at the next page boundary.

// Make the first inaccessible. Use the second as the user stack.

sz = PGROUNDUP(sz);

if((sz = allocuvm(pgdir, sz, sz + 2*PGSIZE)) == 0)

// Push argument strings, prepare rest of stack in ustack.

for(argc = 0; argv[argc]; argc++) {

sp = (sp - (strlen(argv[argc]) + 1)) & ~3;

if(copyout(pgdir, sp, argv[argc], strlen(argv[argc]) + 1) < 0)

Creation of first process by kernel

Why first process needs ‘special’ treatment?

- **Normally process is created using fork()**
 - and typically followed by a call to exec()
- **Fork will use the PCB of existing process to create a new process**
 - as a clone
- **The first process has nothing to copy from!**
- **So it’s PCB needs to “built” by kernel code**

Why first process needs ‘special’ treatment?

- **XV6 approach**
 - Create the process as if it was created by “fork”
 - Ensure that the process starts in a call to “exec”
 - Let “Exec” do the rest of the JOB as expected
 - In this case exec() will call
 - `exec("/init", NULL);`
- **See the code of init.c**
 - opens `console()` device for I/O; dups 0 on 1 and 2!
 - `Same device file for I/O`
 - forks a process and execs (“sh”) on it.
 - Itself keeps waiting for zombie processes

Why first process needs ‘special’ treatment?

- **What needs to be done ?**
 - Build struct proc by hand
 - How data structures (proc, stack, etc) are hand-crafted so that when kernel returns, the process starts in code of init

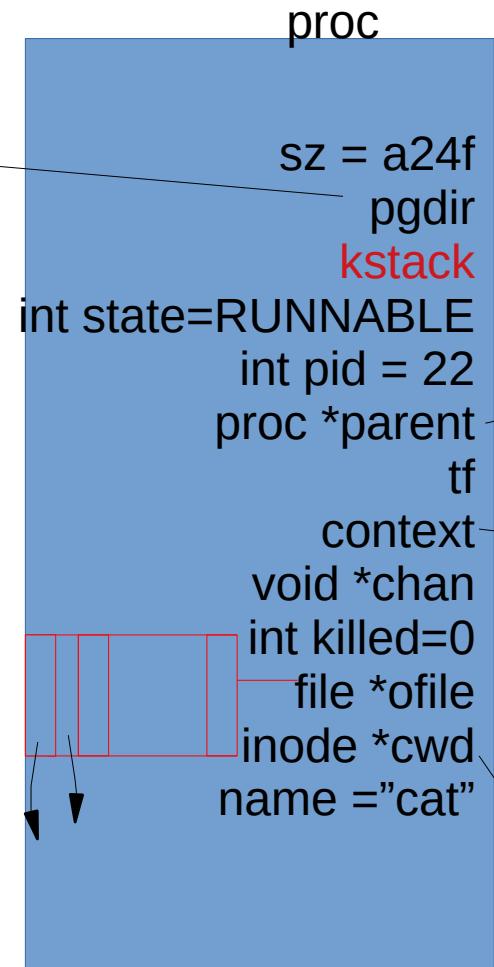
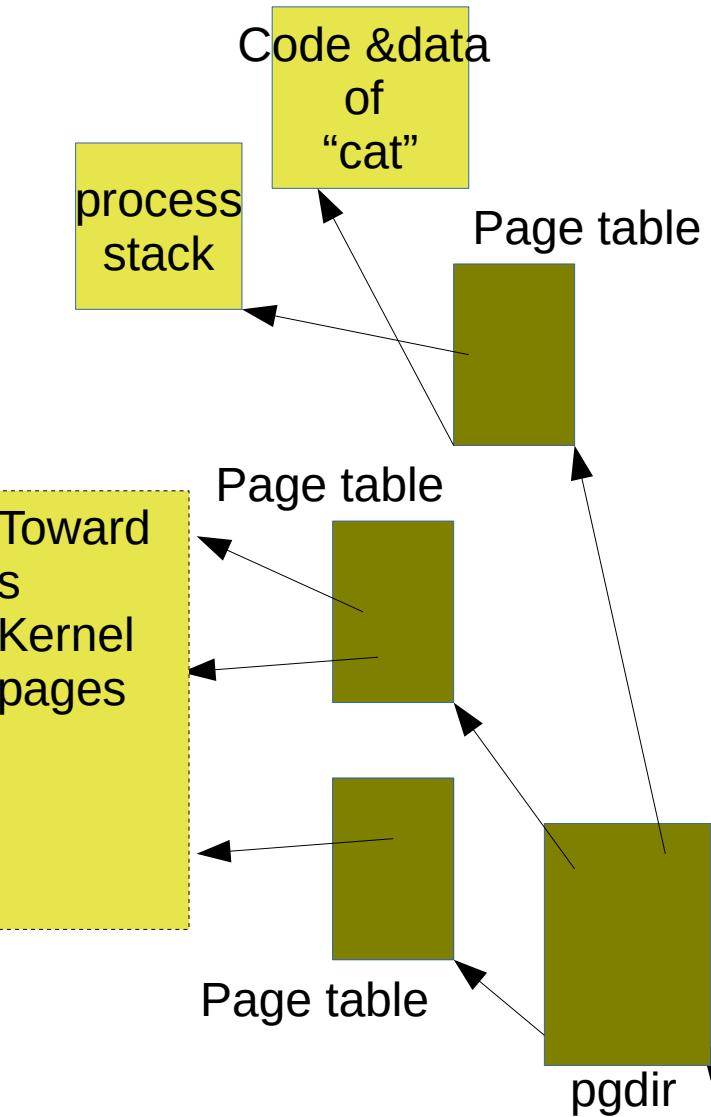
Imp Concepts

- **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

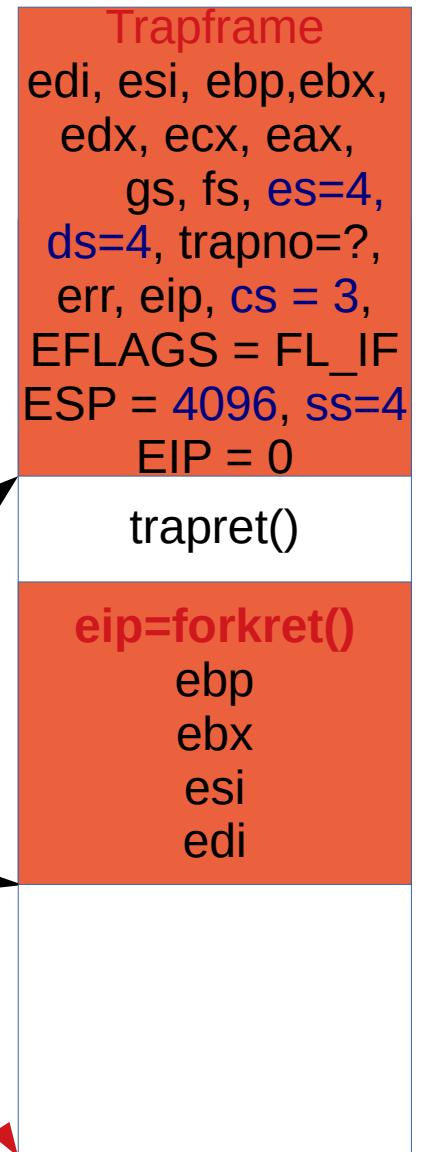
Imp Concepts

```
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  
    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  
    void *chan;                   // If non-zero, sleeping on chan  
    int killed;                   // If non-zero, have been killed  
    struct file *ofile[NOFILE];   // Open files  
    struct inode *cwd;            // Current directory  
    char name[16];                // Process name (debugging)  
};
```

struct proc diagram: Very imp!



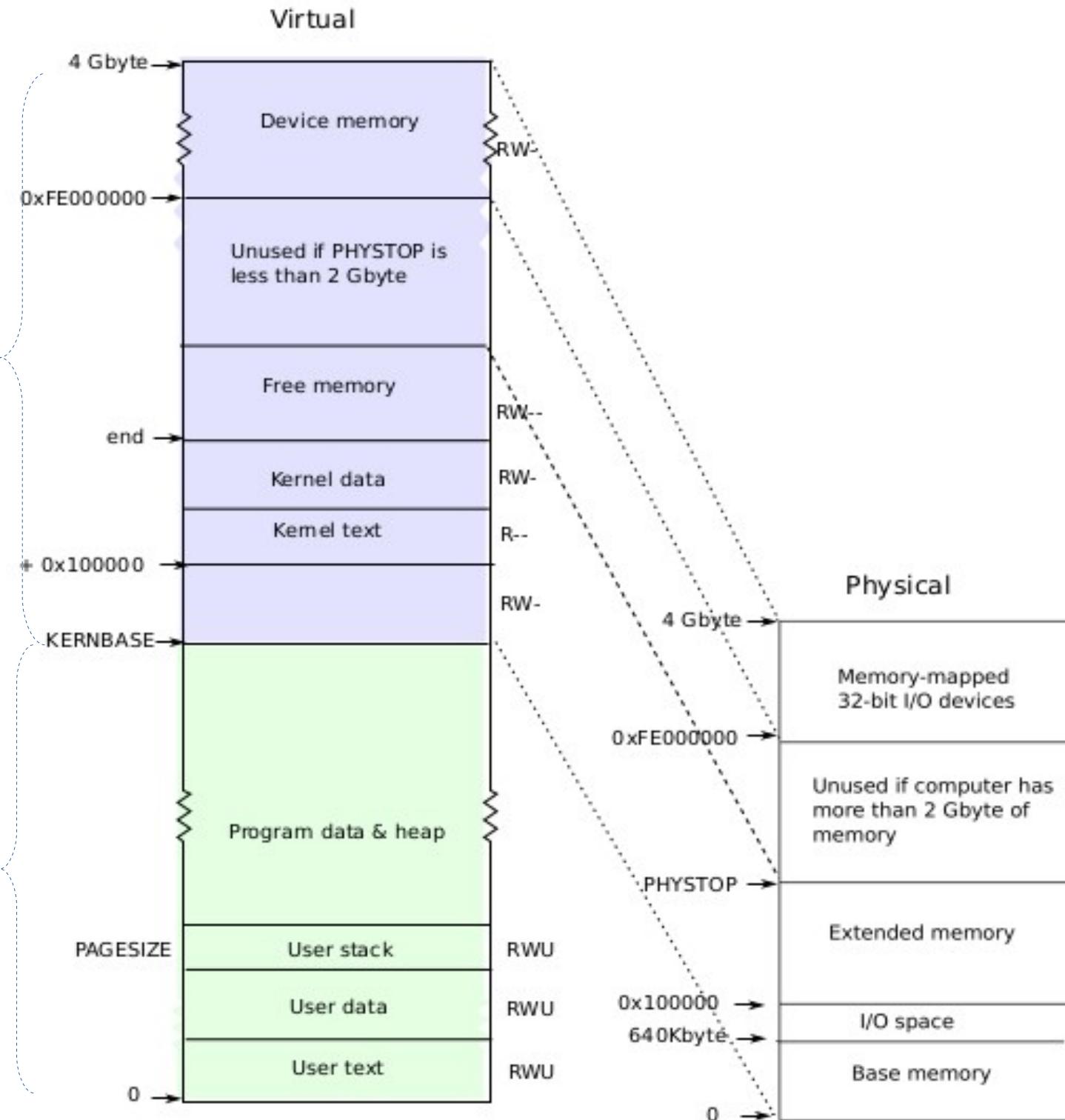
Inode for "/"



In use only when you are in kernel on a "trap" = interrupt/syscall. "tf" always used. trapret,forkret used during fork()

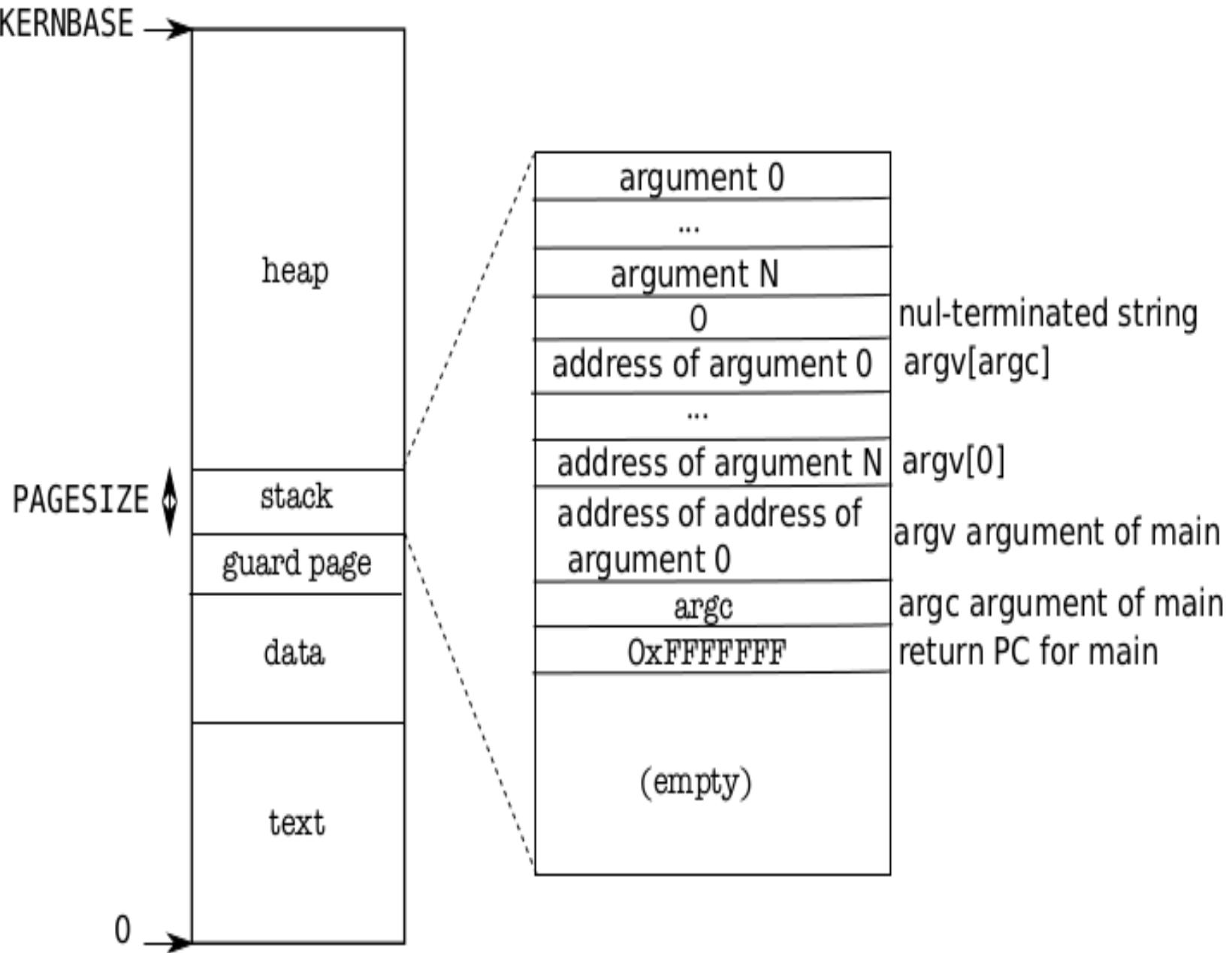
sz = ELF-code->memsz (includes data, check "ld -N") + 2*4096 (for stack)

setupkvm()
does this mapping



These mappings
need to be
created per
process

Memory Layout of a user process After exec()



Note the argc, argv on stack

stack is just one page.

size of text and data is derived from ELF file

main()->userinit()

Creating first process by hand

- **Code of the first process**
 - initcode.S and init.c
 - init.c is compiled into “/init” file
 - During make !
 - Trick:
 - Use initcode.S to “exec(“/init”)”
 - And let exec() do rest of the job
 - But before you do exec()
 - Process must exist as if it was forked() and running

main()->userinit() Creating first process by hand

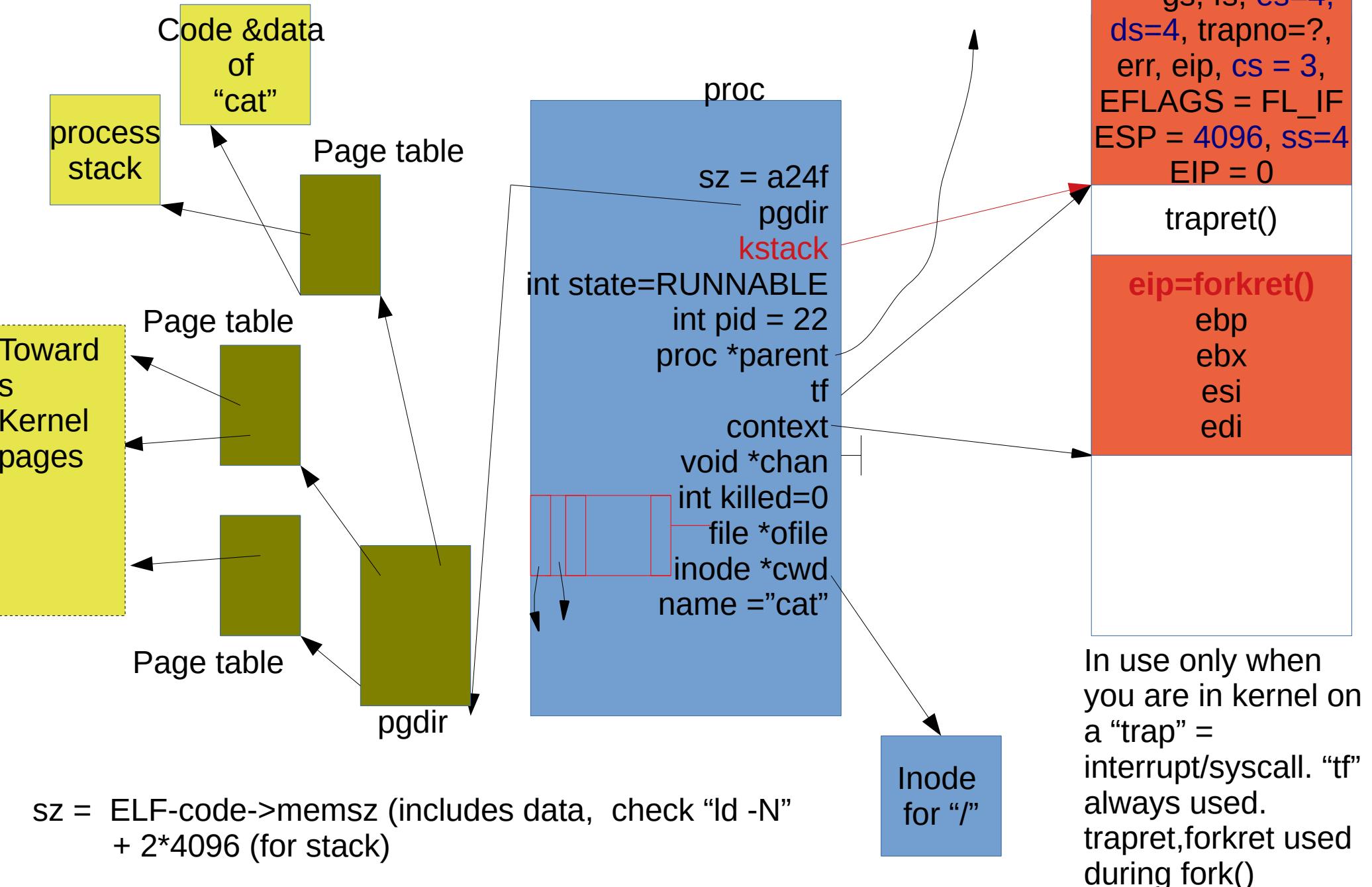
```
void  
userinit(void)  
{  
    struct proc *p;  
    extern char _binary_initcode_start[], _binary_initcode_size[];  
  
    // Abhijit: obtain proc 'p', with stack initialized  
    // and trapframe created and eip set to 'forkret'  
    p = allocproc();  
    // let's see what allocproc() does
```

First process creation

Let's revisit 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



allocproc()

```
static struct proc*
allocproc(void)
{
    struct proc *p;
    char *sp;
    acquire(&ptable.lock);
    for(p = ptable.proc; p <
&ptable.proc[NPROC]; p++)
        if(p->state == UNUSED)
            goto found;
    release(&ptable.lock);
    return 0;
}
```

found:

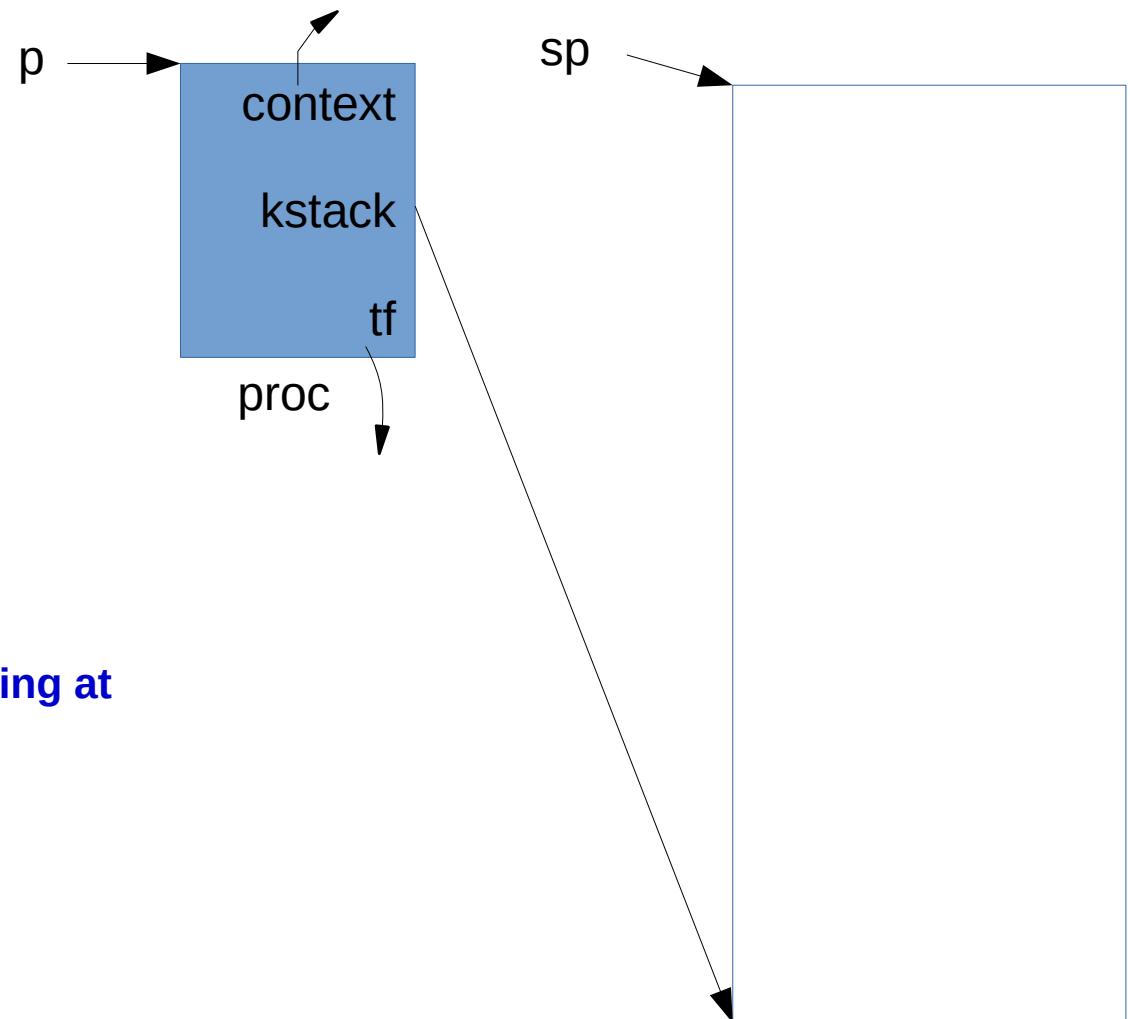
p->state = EMBRYO;

p->pid = nextpid++;

release(&ptable.lock);

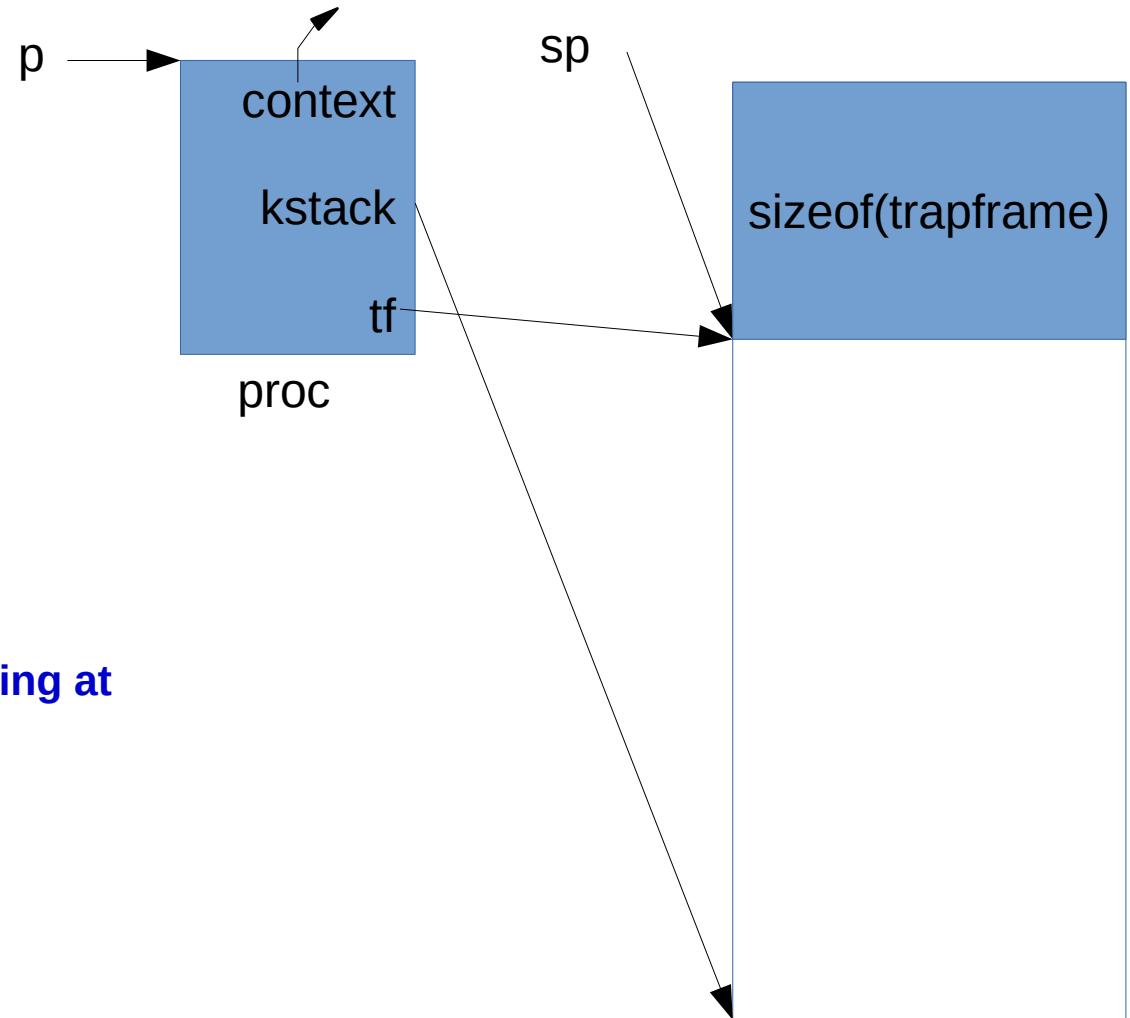
allocproc() setting up stack

```
if((p->kstack = kalloc()) == 0){  
    p->state = UNUSED;  
    return 0;  
}  
sp = p->kstack + KSTACKSIZE;  
// Abhijit KSTACKSIZE = PGSIZE  
// Leave room for trap frame.  
sp -= sizeof *p->tf;  
p->tf = (struct trapframe*)sp;  
// Set up new context to start executing at forkret,  
// which returns to trapret.  
sp -= 4;  
*(uint*)sp = (uint)trapret;  
sp -= sizeof *p->context;  
p->context = (struct context*)sp;  
memset(p->context, 0, sizeof *p->context);  
p->context->eip = (uint)forkret;
```



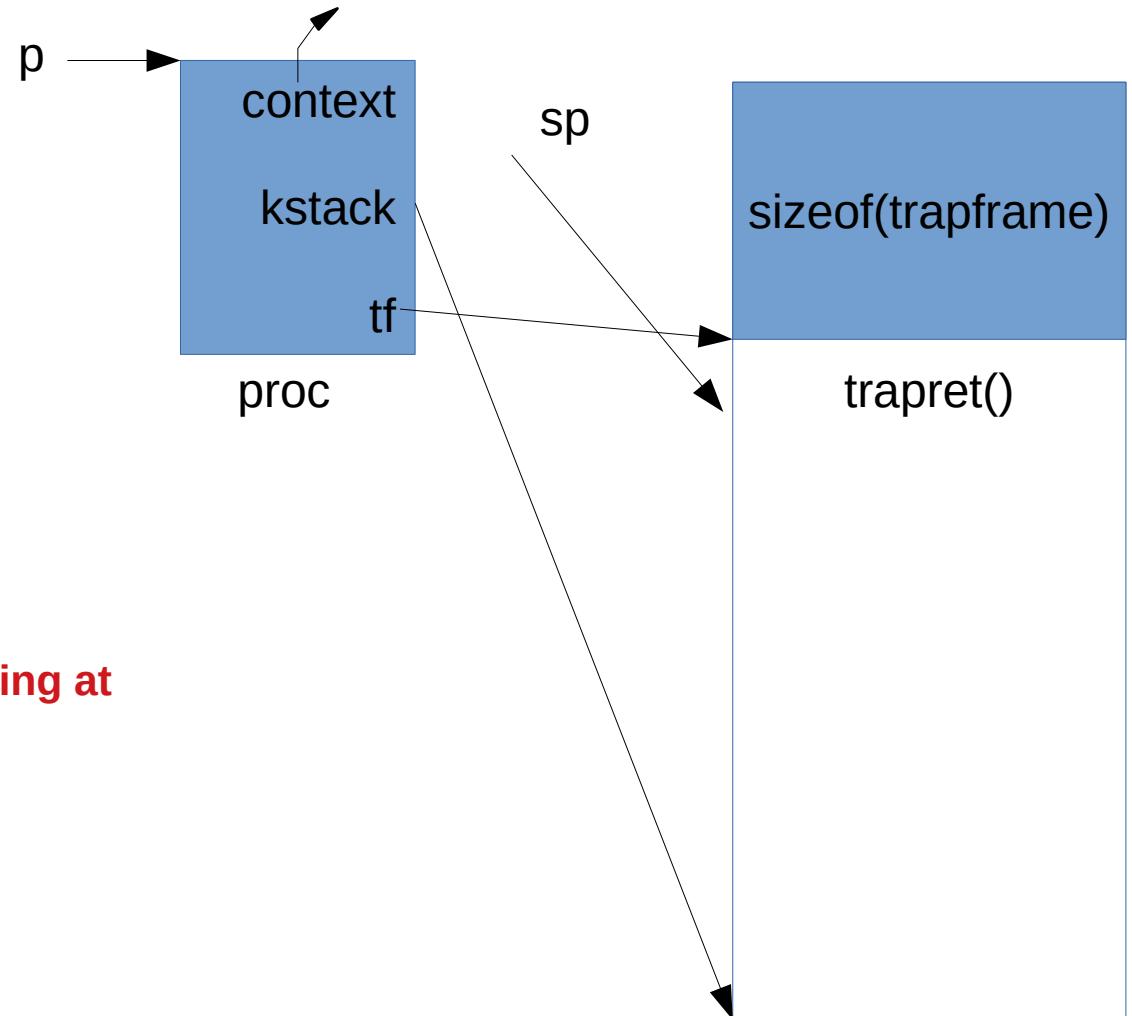
allocproc() setting up stack

```
if((p->kstack = kalloc()) == 0){  
    p->state = UNUSED;  
    return 0;  
}  
sp = p->kstack + KSTACKSIZE;  
// Abhijit KSTACKSIZE = PGSIZE  
// Leave room for trap frame.  
sp -= sizeof *p->tf;  
p->tf = (struct trapframe*)sp;  
// Set up new context to start executing at forkret,  
// which returns to trapret.  
sp -= 4;  
*(uint*)sp = (uint)trapret;  
sp -= sizeof *p->context;  
p->context = (struct context*)sp;  
memset(p->context, 0, sizeof *p->context);  
p->context->eip = (uint)forkret;
```



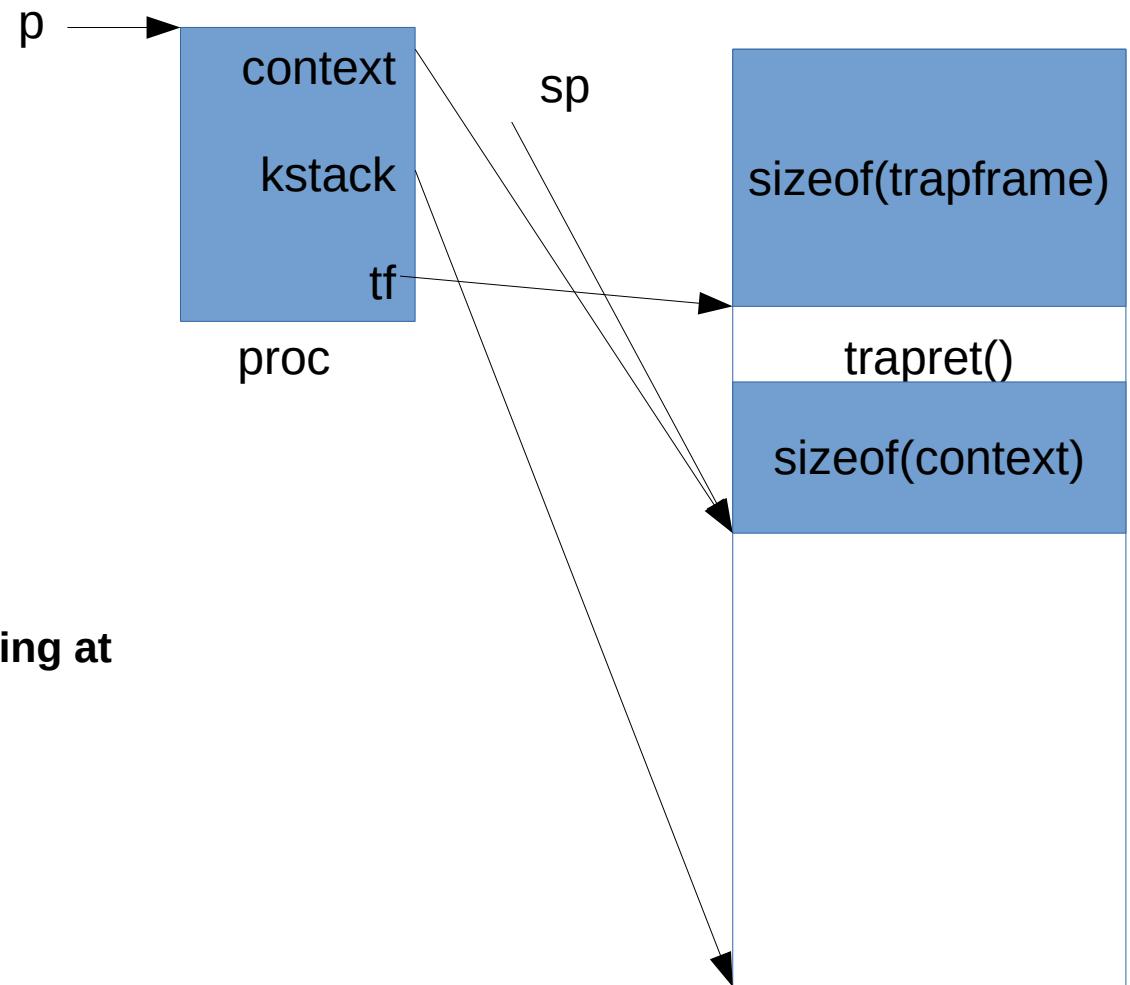
allocproc() setting up stack

```
if((p->kstack = kalloc()) == 0){  
    p->state = UNUSED;  
    return 0;  
}  
sp = p->kstack + KSTACKSIZE;  
// Abhijit KSTACKSIZE = PGSIZE  
// Leave room for trap frame.  
sp -= sizeof *p->tf;  
p->tf = (struct trapframe*)sp;  
// Set up new context to start executing at  
forkret,  
// which returns to trapret.  
sp -= 4;  
*(uint*)sp = (uint)trapret;  
sp -= sizeof *p->context;  
p->context = (struct context*)sp;  
memset(p->context, 0, sizeof *p->context);  
p->context->eip = (uint)forkret;
```



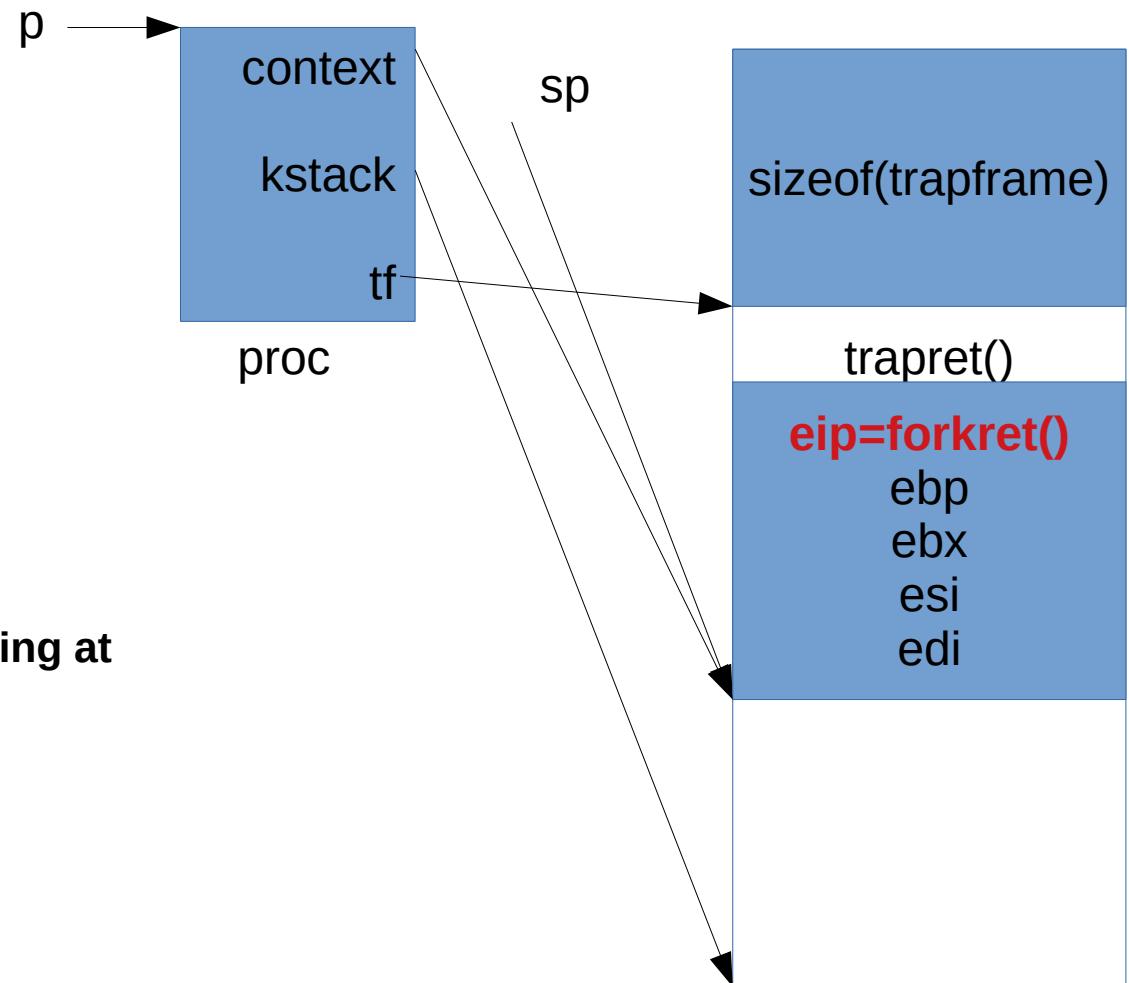
allocproc() setting up stack

```
if((p->kstack = kalloc()) == 0){  
    p->state = UNUSED;  
    return 0;  
}  
  
sp = p->kstack + KSTACKSIZE;  
// Abhijit KSTACKSIZE = PGSIZE  
// Leave room for trap frame.  
sp -= sizeof *p->tf;  
p->tf = (struct trapframe*)sp;  
// Set up new context to start executing at  
forkret,  
// which returns to trapret.  
  
sp -= 4;  
*(uint*)sp = (uint)trapret;  
sp -= sizeof *p->context;  
p->context = (struct context*)sp;  
memset(p->context, 0, sizeof *p->context);  
p->context->eip = (uint)forkret;
```



allocproc() setting up stack

```
if((p->kstack = kalloc()) == 0){  
    p->state = UNUSED;  
    return 0;  
}  
  
sp = p->kstack + KSTACKSIZE;  
// Abhijit KSTACKSIZE = PGSIZE  
// Leave room for trap frame.  
sp -= sizeof *p->tf;  
p->tf = (struct trapframe*)sp;  
// Set up new context to start executing at  
forkret,  
// which returns to trapret.  
  
sp -= 4;  
*(uint*)sp = (uint)trapret;  
sp -= sizeof *p->context;  
p->context = (struct context*)sp;  
memset(p->context, 0, sizeof *p->context);  
p->context->eip = (uint)forkret;
```



Next in userinit()

```
initproc = p;
if((p->pgdir = setupkvm()) == 0)
    panic("userinit: out of
memory");

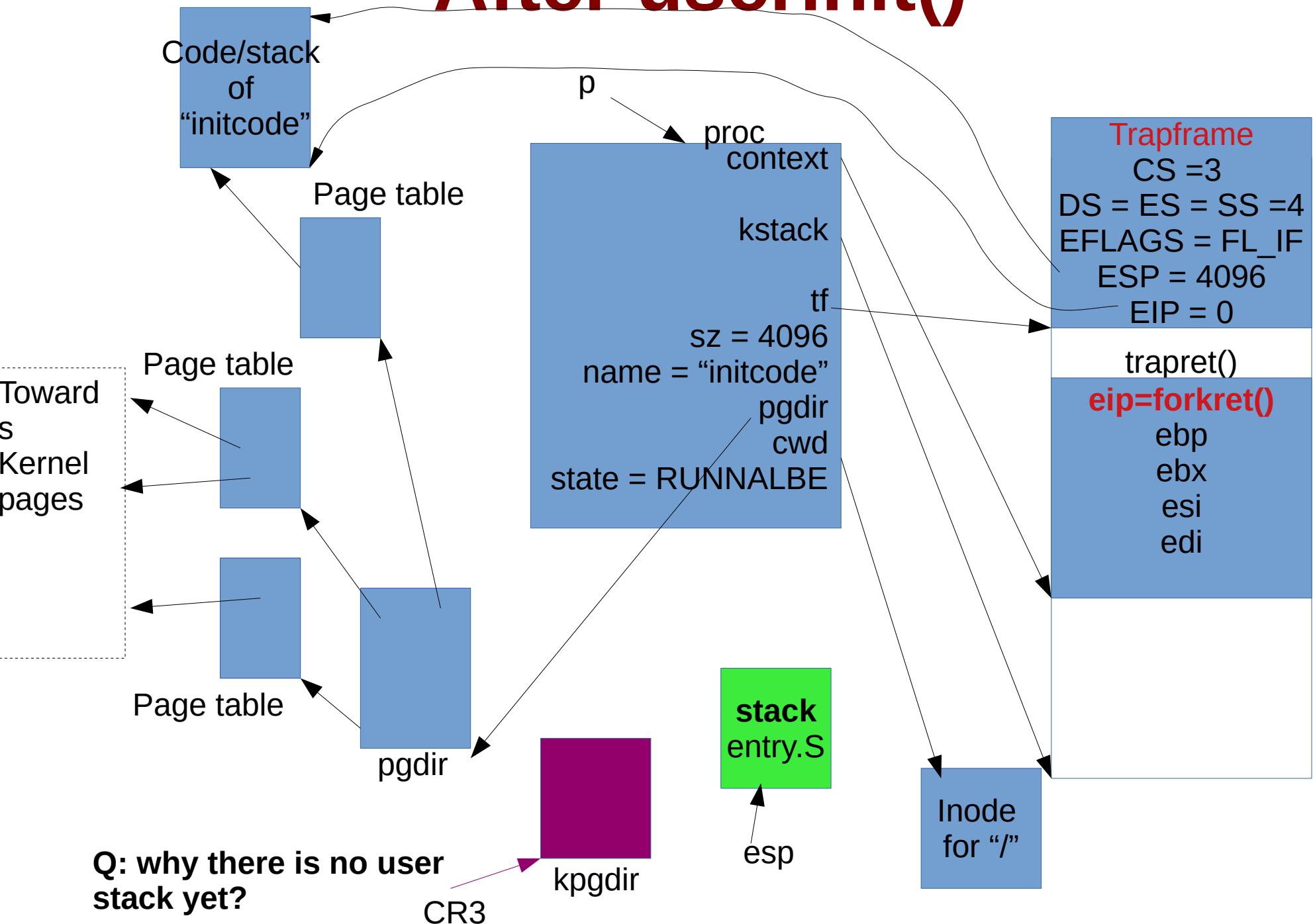
inituvm(p->pgdir,
_binary_initcode_start,
(int)_binary_initcode_size);
p->sz = PGSIZE;
memset(p->tf, 0, sizeof(*p->tf));
p->tf->cs = (SEG_UCODE << 3) |
DPL_USER;
p->tf->ds = (SEG_UDATA << 3) |
DPL_USER;
p->tf->es = p->tf->ds;
p->tf->ss = p->tf->ds;

p->tf->eflags = FL_IF;
p->tf->esp = PGSIZE;
p->tf->eip = 0; // beginning of
initcode.S
safestrcpy(p->name, "initcode",
sizeof(p->name));

p->cwd = namei("/");
acquire(&ptable.lock);
p->state = RUNNABLE;

release(&ptable.lock);
```

After userinit()



main() -> mpmain()

```
static void  
mpmain(void)  
{  
    cprintf("cpu%d: starting %d\n",  
cpuid(), cpuid());  
    idtinit(); // load idt register  
    xchg(&(mycpu()->started), 1); //  
tell startothers() we're up  
    scheduler(); // start running  
processes  
}
```

- **Load IDT register**
 - Copy from `idt[]` array into `IDTR`
- **Call `scheduler()`**
 - One process has already been made runnable
 - Let's enter scheduler now

Before reading scheduler(): Note

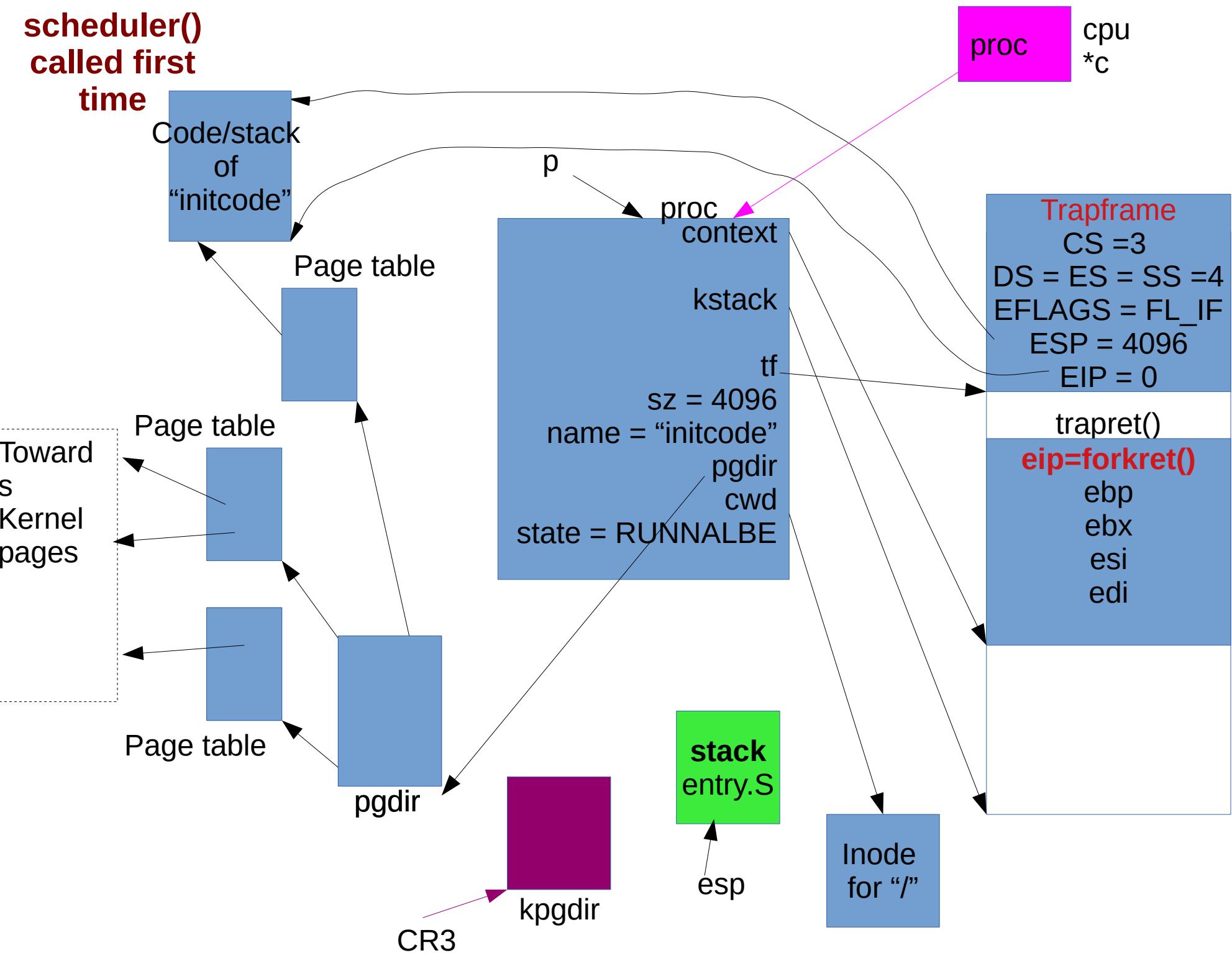
- The **esp** is still pointing to the stack which was allocated in **entry.S**!
 - this is the kernel only stack
 - Not the per process kernel stack.
- **CR3** points to **kpgdir**
- **Struct cpu[] has been setup up already**
 - apicid – in mpinit()
 - segdesc gdt – in seginit()
 - started – in mpmain()
- **Fields in cpu[] not yet set**
 - context * scheduler --> will be setup in sched()
 - taskstate ts --> large structure, only parts used in switchuvvm()
 - ncli, intena --> used while locking
 - proc *proc -> set during scheduler()

scheduler()

```
void
scheduler(void)
{
    struct proc *p;
    struct cpu *c = mycpu();
    c->proc = 0;

    for(;;){
        sti();
        // Loop over process table looking for process to run.
        acquire(&ptable.lock);
        for(p = ptable.proc; p < &ptable.proc[NPROC]; p++){
            if(p->state != RUNNABLE)
                continue;
            // Switch to chosen process.  It is the process's job
            // to release ptable.lock and then reacquire it
            // before jumping back to us.
            c->proc = p;
        }
    }
}
```

scheduler()
called first
time



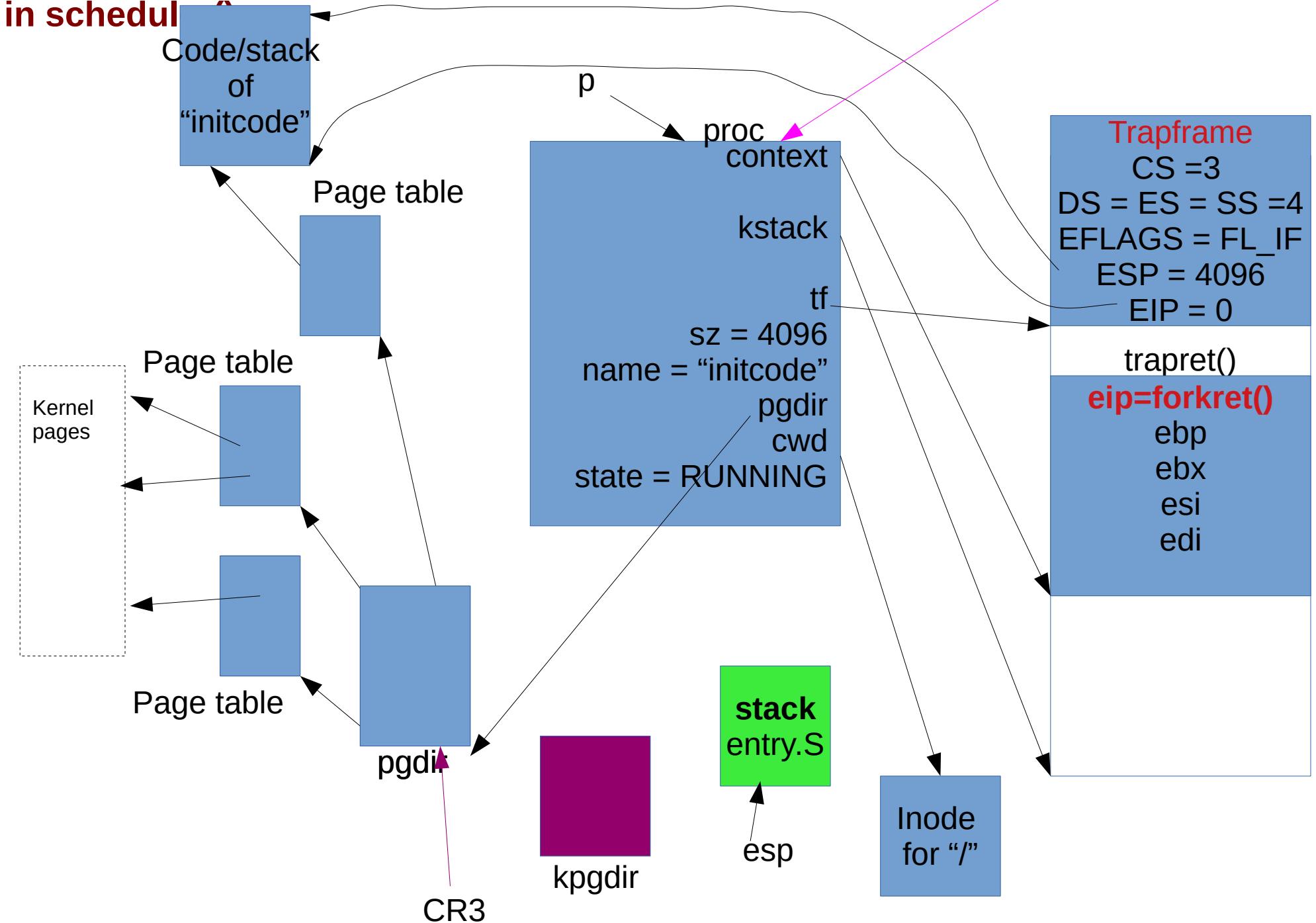
scheduler()

```
acquire(&ptable.lock);
for(p = ptable.proc; p < &ptable.proc[NPROC]; p++){
    if(p->state != RUNNABLE)
        continue;
```

```
// Switch to chosen process. It is the process's job
// to release ptable.lock and then reacquire it
// before jumping back to us.
```

```
c->proc = p;
switchuvm(p);
p->state = RUNNING;
```

after
switchuvvm()
in scheduler

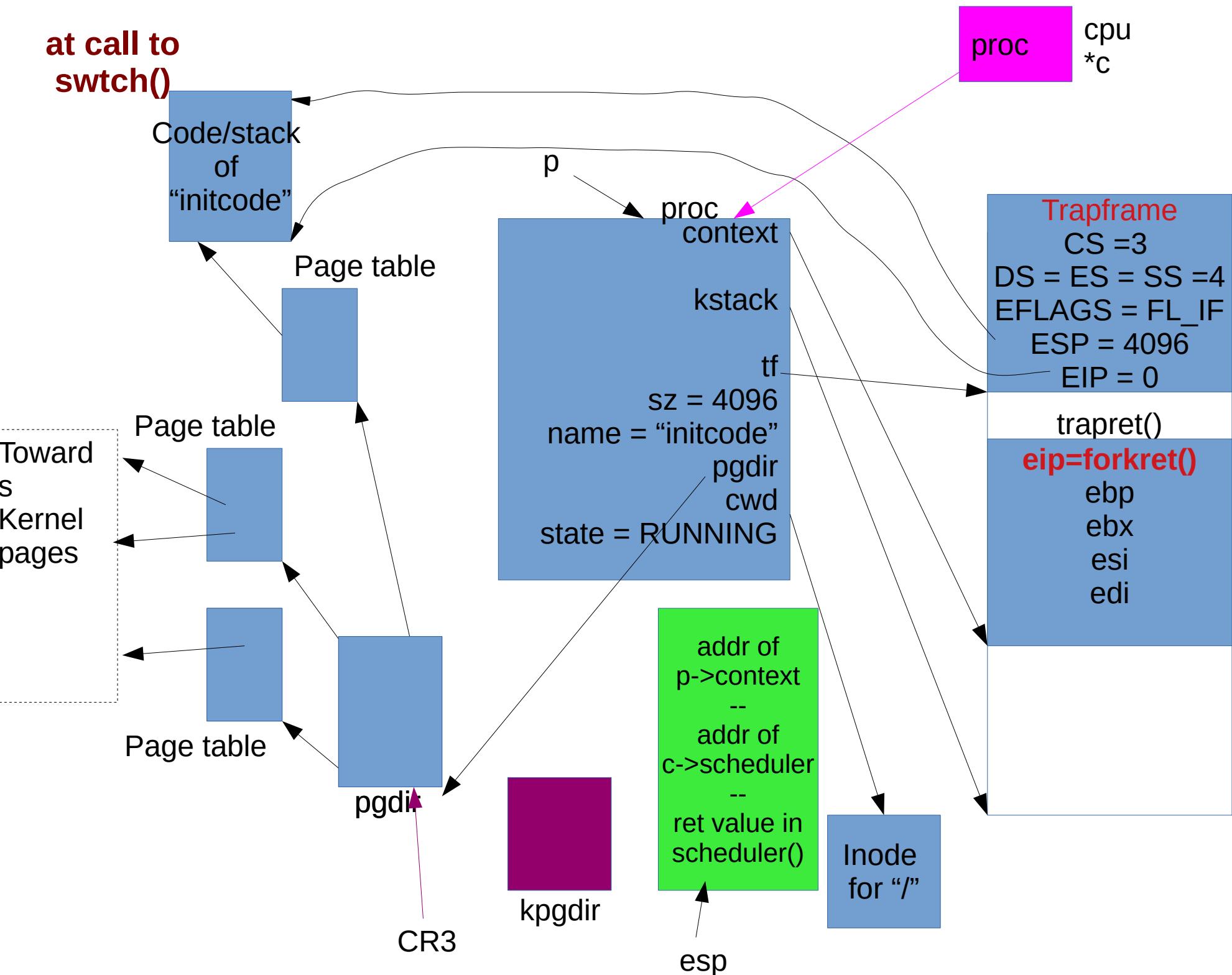


scheduler()

```
acquire(&ptable.lock);
for(p = ptable.proc; p < &ptable.proc[NPROC]; p++){
    if(p->state != RUNNABLE)
        continue;

    // Switch to chosen process. It is the process's job
    // to release ptable.lock and then reacquire it
    // before jumping back to us.
    c->proc = p;
    switchuvm(p);
    p->state = RUNNING
    swtch(&(c->scheduler), p->context);
};
```

at call to
swtch()



swtch

swtch:

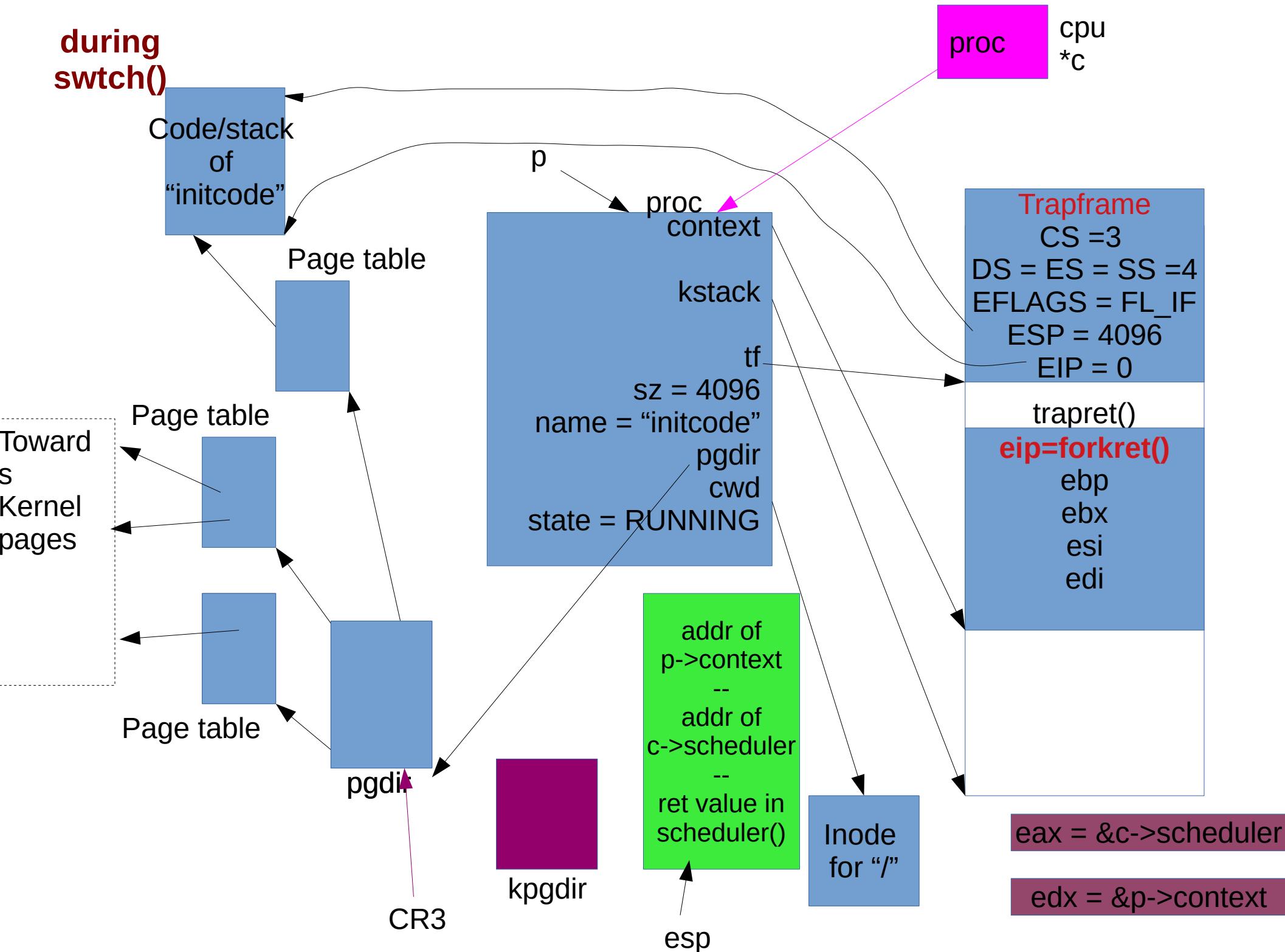
#Abhijit: swtch was called through a function call.

#So %eip was saved on stack already

movl 4(%esp), %eax # Abhijit: eax = old

movl 8(%esp), %edx # Abhijit: edx = new

**during
swtch()**



swtch

swtch:

#Abhijit: swtch was called through a function call.

#So %eip was saved on stack already

movl 4(%esp), %eax # Abhijit: eax = old

movl 8(%esp), %edx # Abhijit: edx = new

Save old callee-saved registers

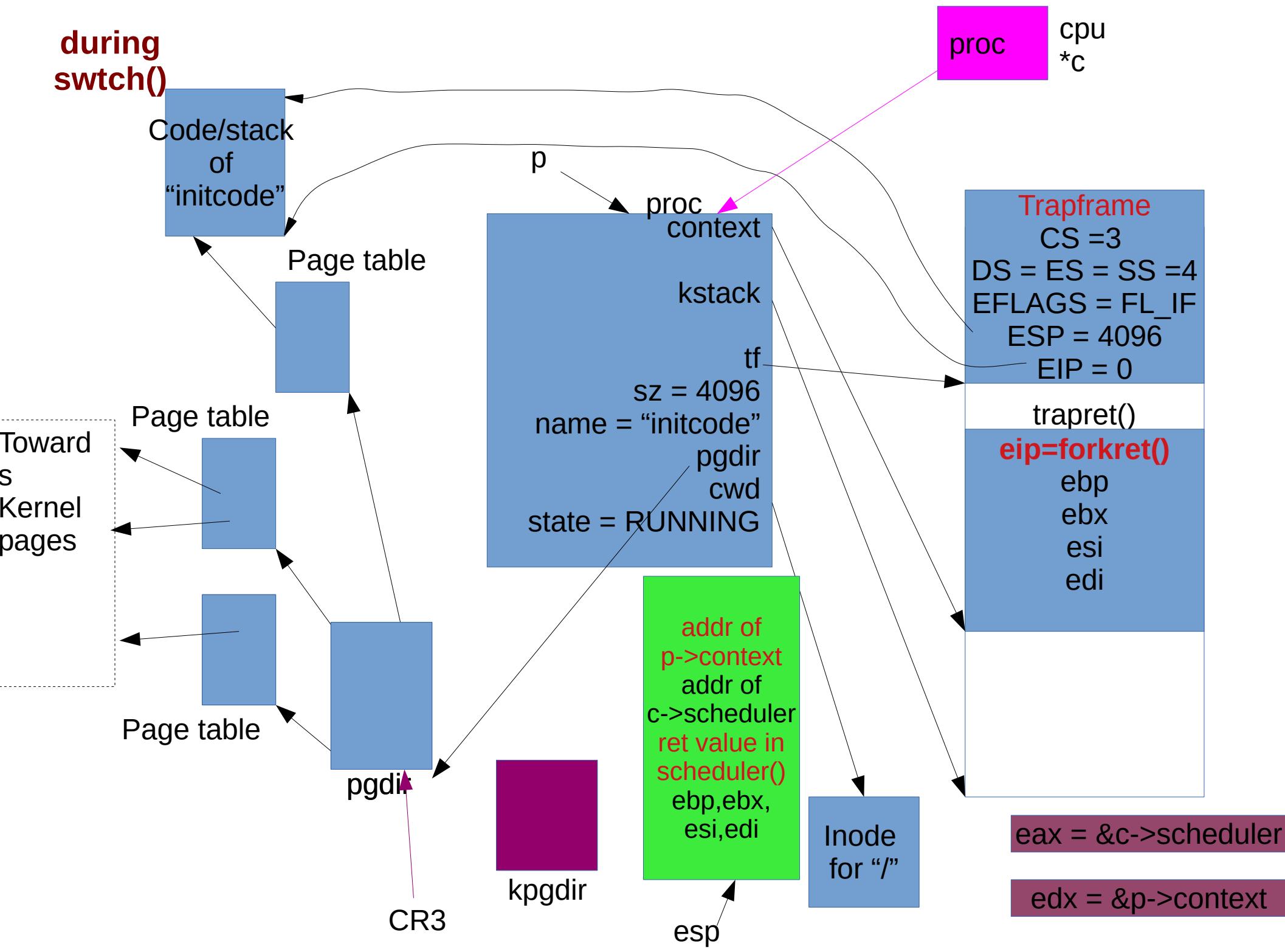
pushl %ebp

pushl %ebx

pushl %esi

pushl %edi # Abhijit: esp = esp + 16

**during
swtch()**

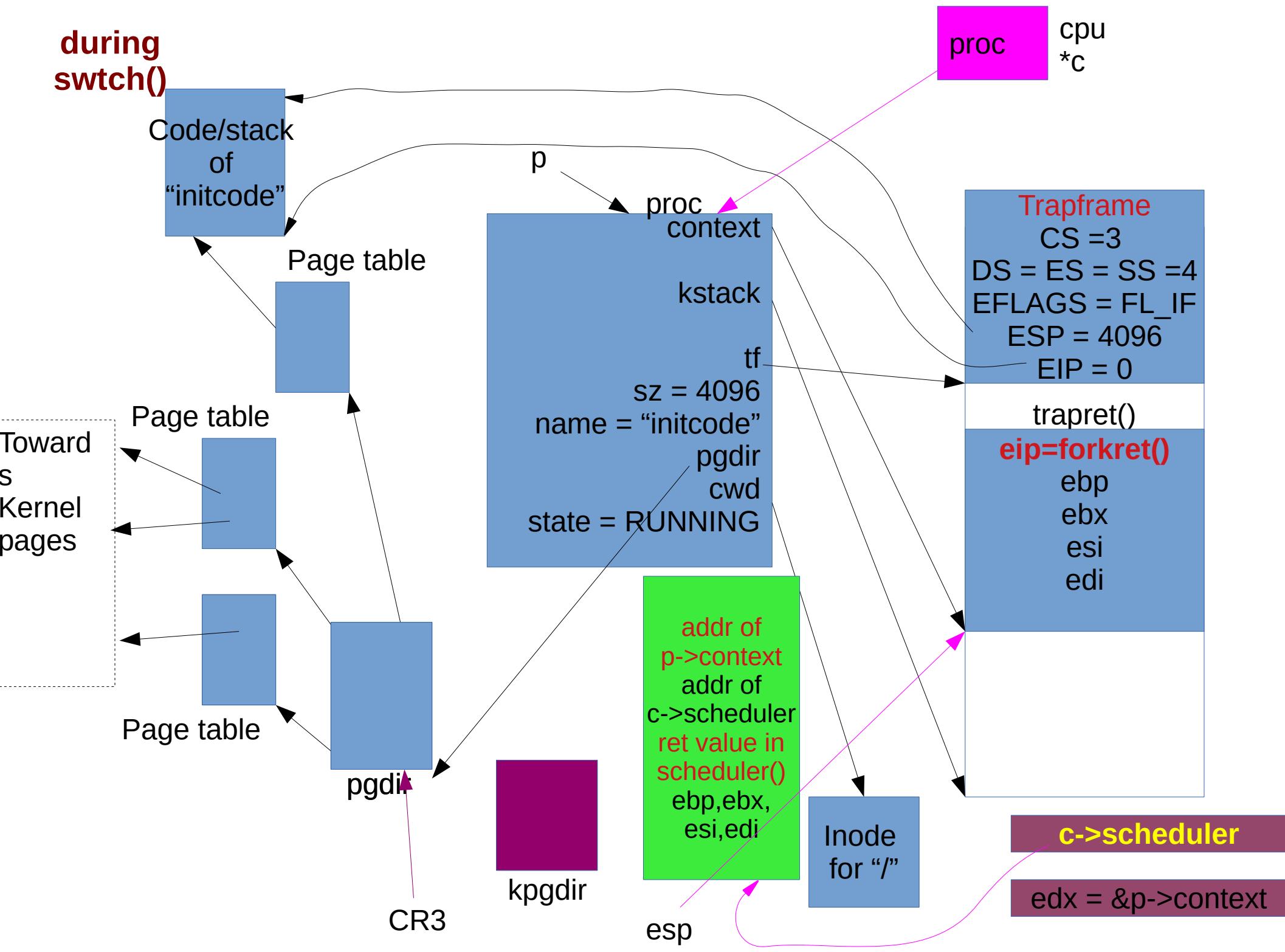


swtch

swtch:

```
#Abhijit: swtch was called through a function call.  
#So %eip was saved on stack already  
movl 4(%esp), %eax # Abhijit: eax = old  
movl 8(%esp), %edx # Abhijit: edx = new  
# Save old callee-saved registers  
pushl %ebp  
pushl %ebx  
pushl %esi  
pushl %edi      # Abhijit: esp = esp + 16  
# Switch stacks  
movl %esp, (%eax) # Abhijit: *old = updated old stack  
movl %edx, %esp # Abhijit: esp = new
```

during
swtch()



swtch

swtch:

#Abhijit: swtch was called through a function call.

#So %eip was saved on stack already

movl 4(%esp), %eax # Abhijit: eax = old

movl 8(%esp), %edx # Abhijit: edx = new

Save old callee-saved registers

pushl %ebp

pushl %ebx

pushl %esi

pushl %edi # Abhijit: esp = esp + 16

Switch stacks

movl %esp, (%eax) # Abhijit: *old = updated old stack

movl %edx, %esp # Abhijit: esp = new

Load new callee-saved registers

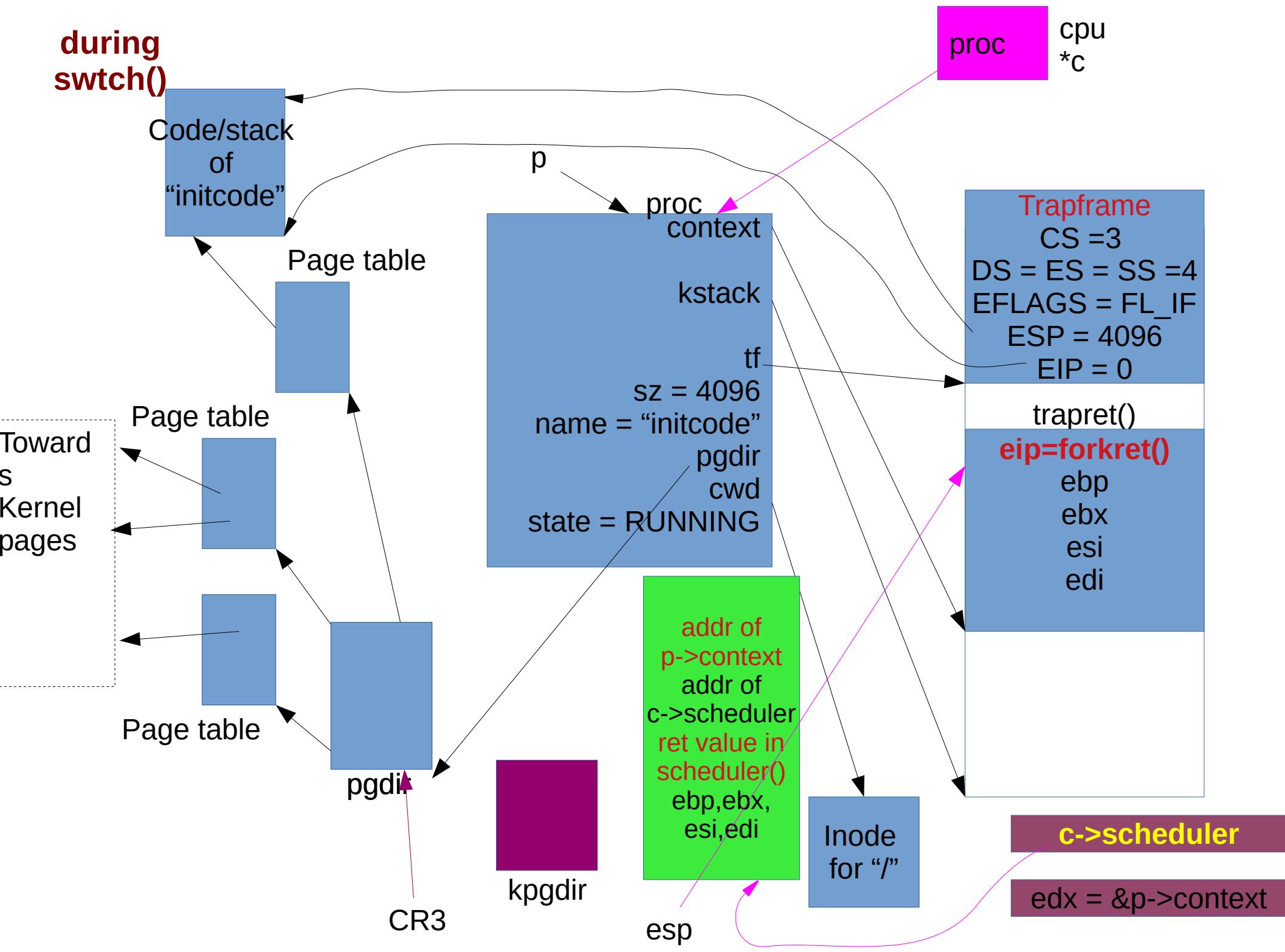
popl %edi

popl %esi

popl %ebx

popl %ebp # Abhijit: newesp = newesp - 16, context restored

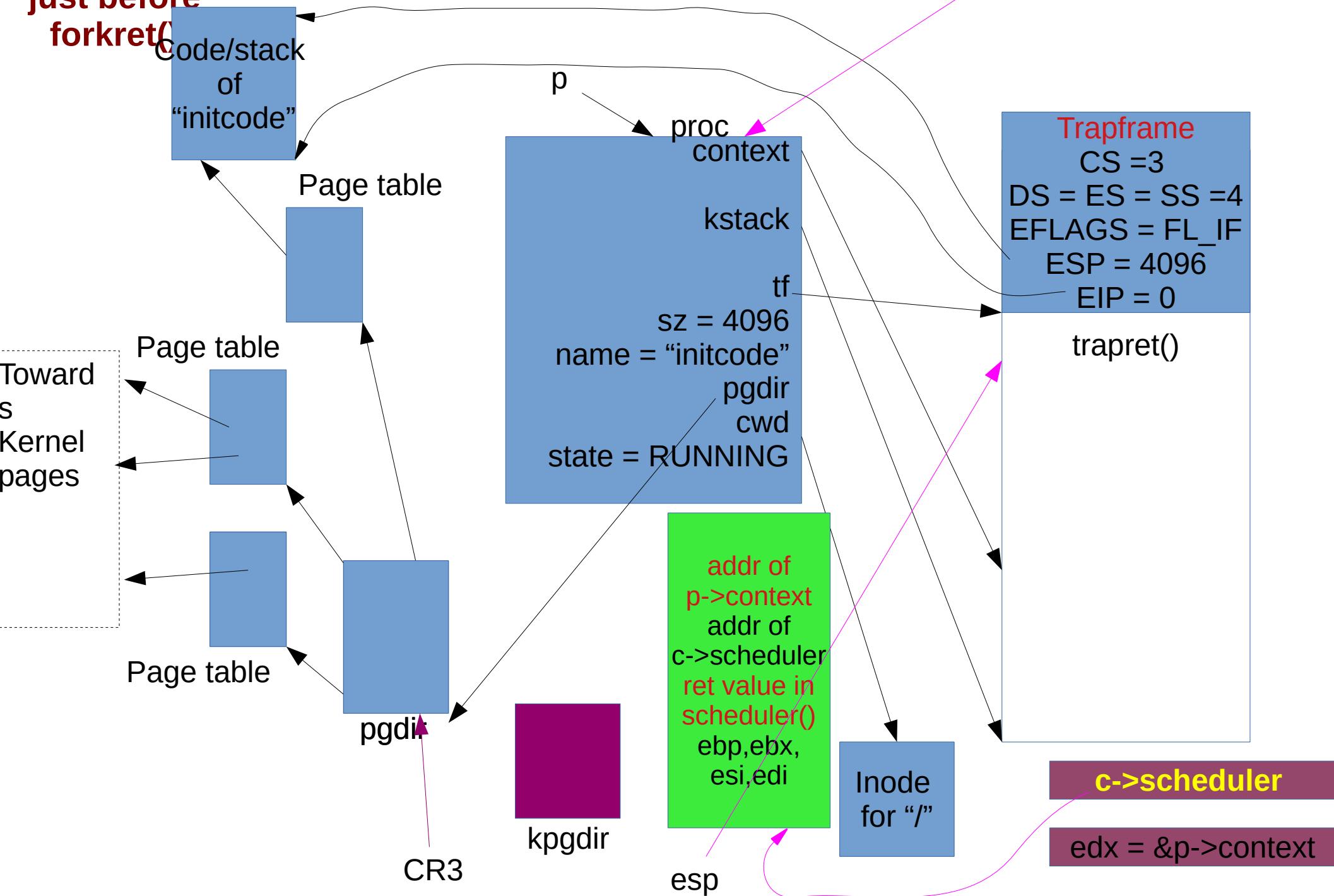
**during
swtch()**



swtch:

#Abhijit: swtch was called through a function call. **Swtch**
#So %eip was saved on stack already
movl 4(%esp), %eax # Abhijit: eax = old
movl 8(%esp), %edx # Abhijit: edx = new
Save old callee-saved registers
pushl %ebp
pushl %ebx
pushl %esi
pushl %edi # Abhijit: esp = esp + 16
Switch stacks
movl %esp, (%eax) # Abhijit: *old = updated old stack
movl %edx, %esp # Abhijit: esp = new
Load new callee-saved registers
popl %edi
popl %esi
popl %ebx
popl %ebp # Abhijit: newesp = newesp - 16, context restored
ret # Abhijit: will pop from esp now -> function where to return.

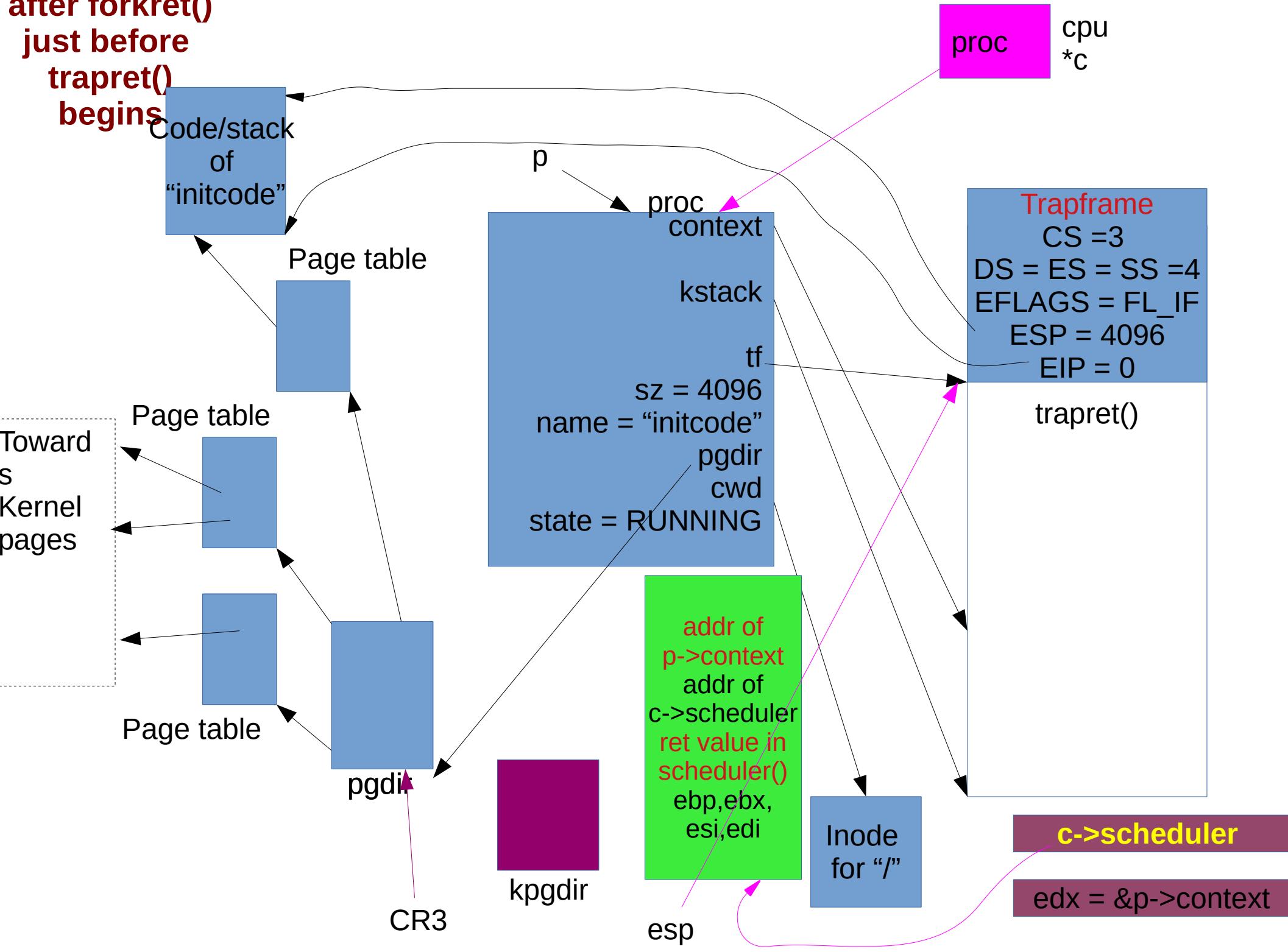
after "ret"
from swtch()
just before
forkret()



After swtch()

- Process is running in forkret()
- c->csheduler has saved the old kernel stack
 - with the context of p, return value in scheduler, ebp, ebx, esi, edi on stack
 - remember {edi, esi, ebx, ebp, ret-value } = context
 - The c->scheduler is pointing to old context
- CR3 is pointing to process pgdir

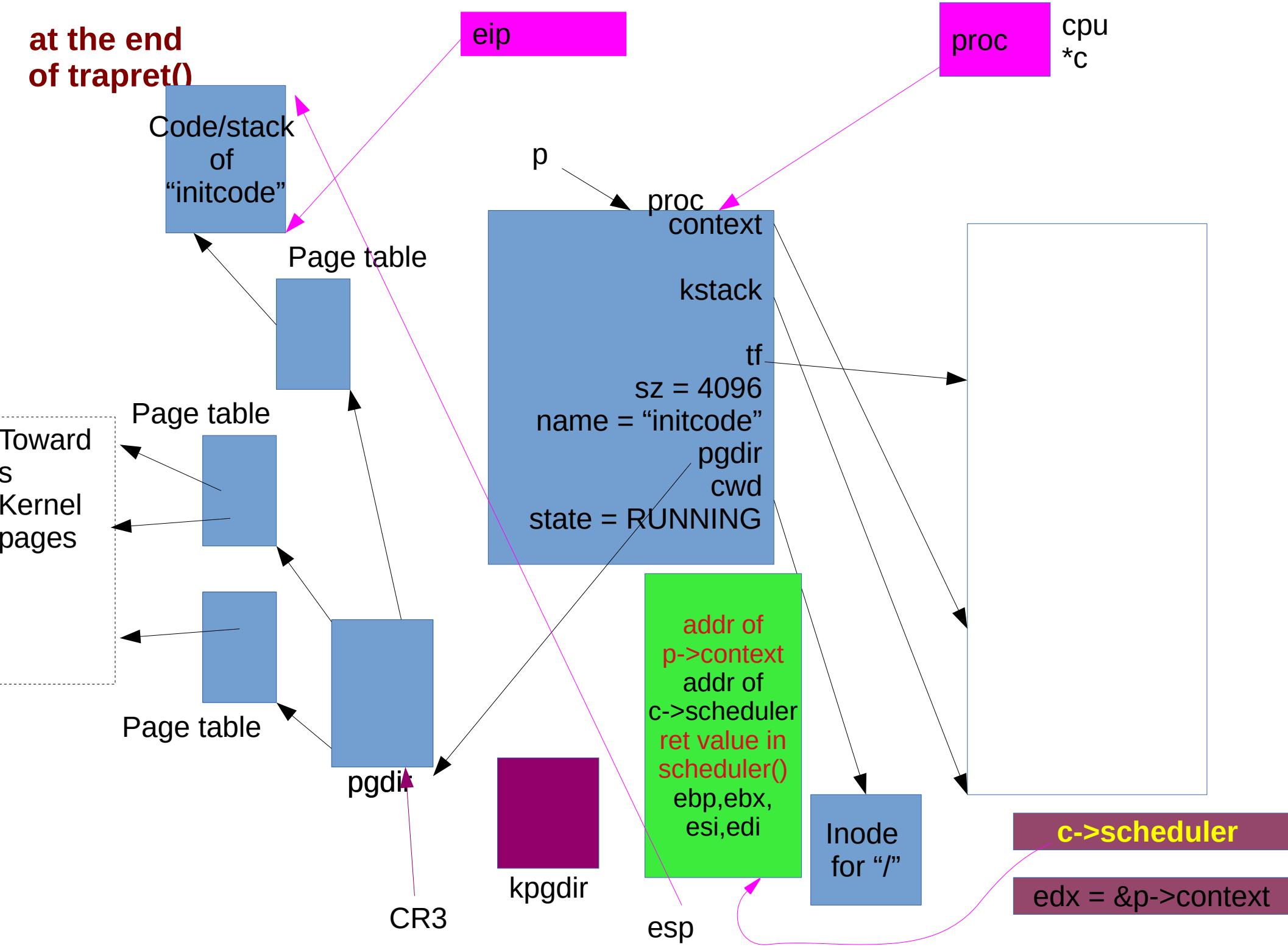
after forkret()
just before
trapret()
begins



After iret in trapret

- **The CS, EIP, ESP will be changed**
 - to values already stored on trapframe
 - this is done by iret
- **Hence after this user code will run**
 - On user stack!
- **Hence code of *initcode* will run now**

at the end
of trapret()



initcode

```
# char init[] = "/init\0";
```

init:

```
.string "/init\0"
```

```
# char *argv[] = { init, 0 };
```

.p2align 2

argv:

```
.long init
```

```
.long 0
```

start:

```
pushl $argv
```

```
pushl $init
```

```
pushl $0 // where caller pc  
would be
```

```
movl $SYS_exec, %eax
```

```
int $T_SYSCALL
```

for(;;) exit();

exit:

```
movl $SYS_exit, %eax
```

```
int $T_SYSCALL
```

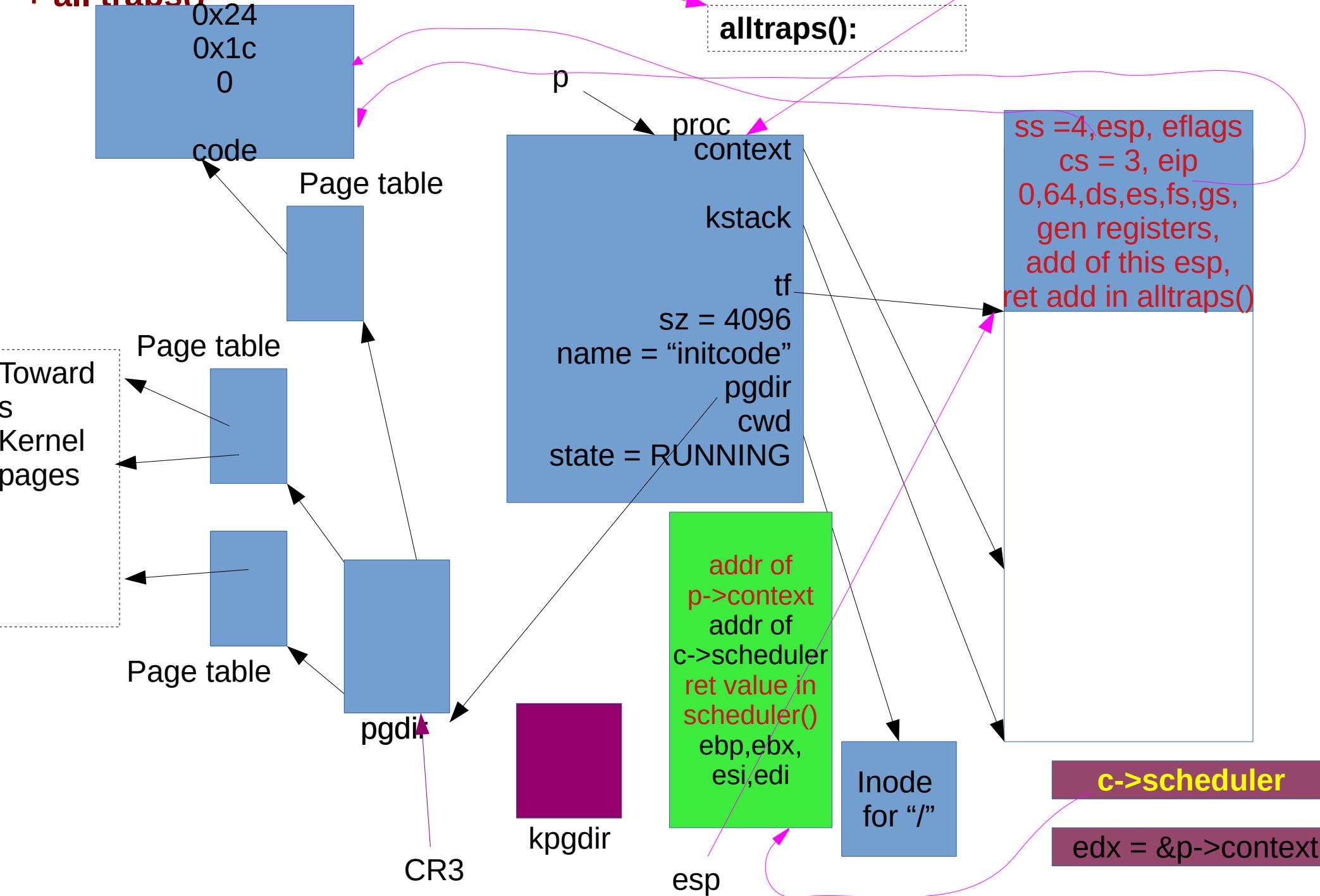
```
jmp exit
```

0x24 = addr of argv
0x1c = addr of init
0x0

esp

```
00000000 <start>:  
 0: 68 24 00 00 00      push $0x24  
 5: 68 1c 00 00 00      push $0x1c  
 a: 6a 00                push $0x0  
 c: b8 07 00 00 00      mov $0x7,%eax  
 11: cd 40               int $0x40  
  
00000013 <exit>:  
 13: b8 02 00 00 00     mov $0x2,%eax  
 18: cd 40               int $0x40  
 1a: eb f7               jmp 13 <exit>  
  
0000001c <init>:  
           "/init\0"  
  
00000024 <argv>:  
 1c 00  
 00 00
```

**on sys_exec()
+ all traps()**



Understanding fork() and exec()

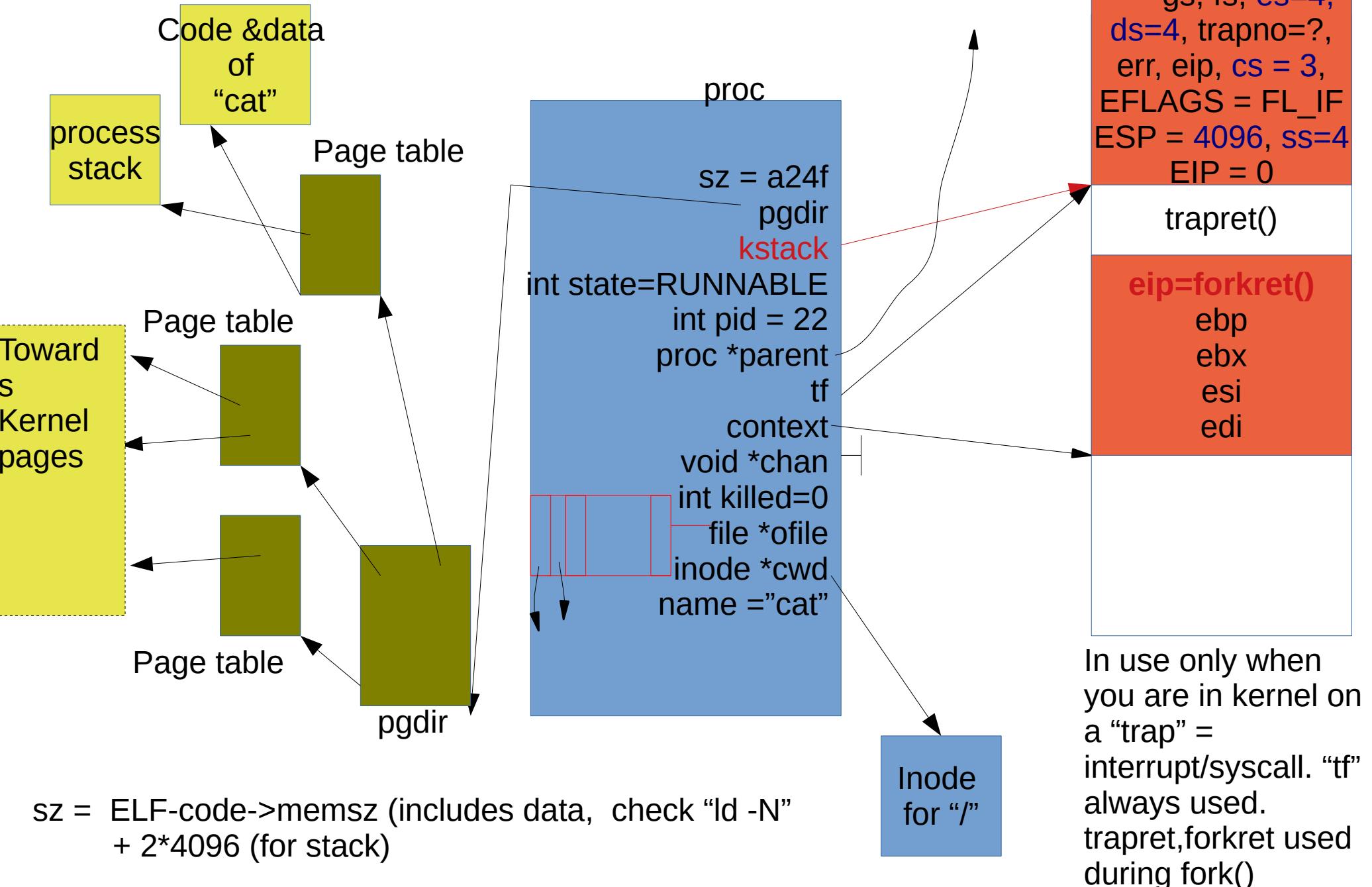
**First, revising some concepts already learnt
then code of fork(), exec()**

First process creation

Let's revisit 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



fork()/exec() are syscalls. On every syscall this happens

- Fetch the n'th descriptor from the IDT, where n is the argument of int.
- Check that CPL in %cs is <= DPL, where DPL is the privilege level in the descriptor.
- 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 switchuvm()
- **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, during idtinit()
 - Remember vectors.S – gives jump locations for each interrupt
- “int 64” ->jump to 64th entry in vector table

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

```
# 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
```

alltraps:

- 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
 - `UatINTR()`
- 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,

understanding fork()

- **What should fork do?**
 - Create a copy of the existing process
 - child is same as parent, except pid, parent-child relation, return value (pid or 0)
 - Please go through every member of struct proc, understand it's meaning to appreciate what fork() should do
 - create a struct proc, and
 - **duplicate pages, page directory, sz, state, trapframe, context, ofile (and files!), cwd, name**
 - **modify: pid, parent, trapframe, state**

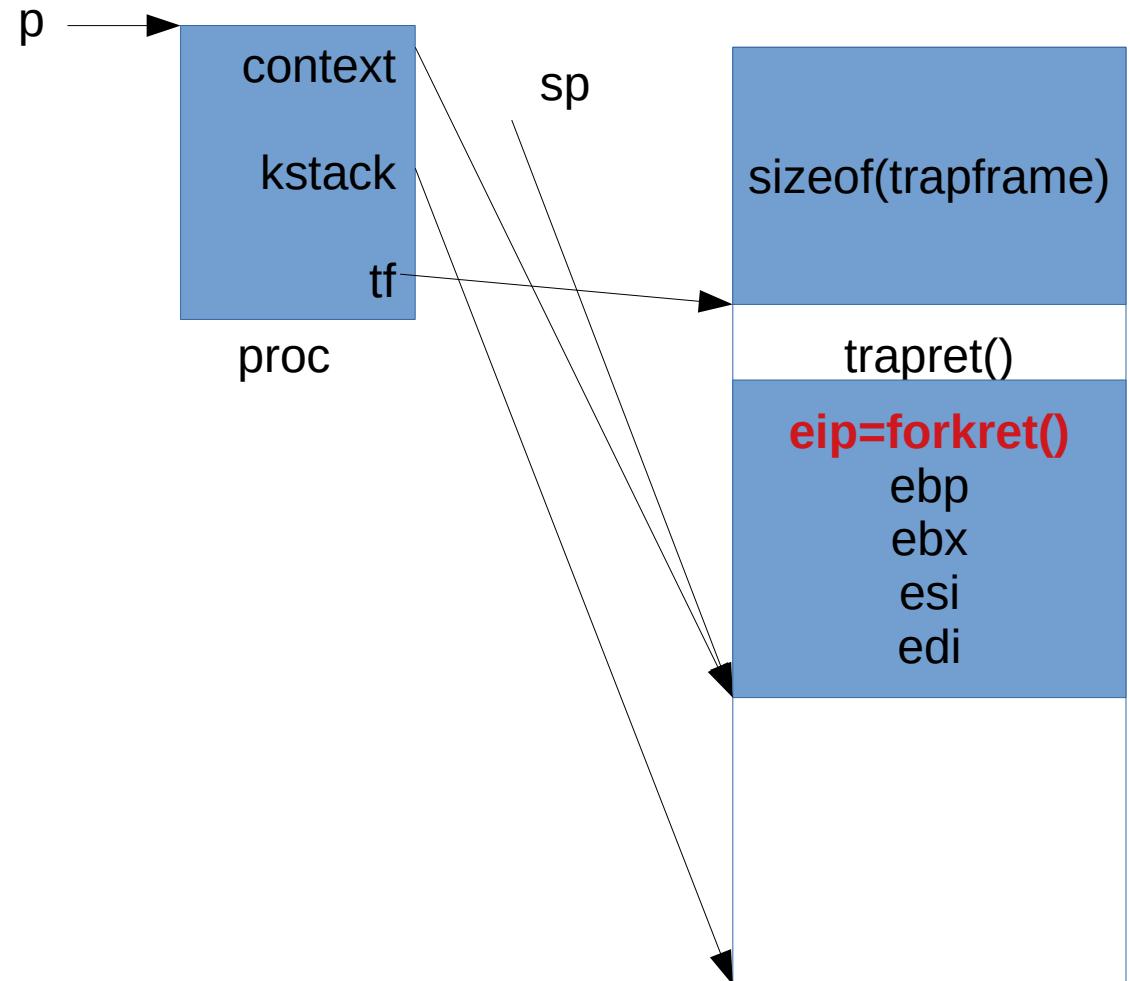
understanding fork()

```
int  
sys_fork(void)  
{  
    return fork();  
}
```

```
int  
fork(void)  
{  
    int i, pid;  
    struct proc *np;  
    struct proc *curproc = myproc();  
  
    // Allocate process.  
    if((np = allocproc()) == 0){  
        return -1;  
    }
```

after allocproc()

-- we studied this -- same as creation of first process



understanding fork()

```
// Copy process state from proc.  
if((np->pgdir = copyuvm(curproc->pgdir, curproc->sz)) == 0){  
    kfree(np->kstack);  
    np->kstack = 0;  
    np->state = UNUSED;  
    return -1;  
}  
  
np->sz = curproc->sz;  
np->parent = curproc;  
*np->tf = *curproc->tf;
```

- **copy the pages, page tables, page directory**
 - no copy on write here!
 - Rewind if operation of `copyuvm()` fails
- **copy size**
- **set parent of child**
- **copy trapframe
(structure is copied)**

```

pde_t*
copyuvvm(pde_t *pgdir, uint sz)
{
    pde_t *d; pte_t *pte; uint pa, i, flags;
    char *mem;

    if((d = setupkvm()) == 0)
        return 0;
    for(i = 0; i < sz; i += PGSIZE){
        if((pte = walkpgdir(pgdir, (void *) i, 0)) == 0)
            panic("copyuvvm: pte should exist");
        if(!(pte & PTE_P))
            panic("copyuvvm: page not present");
        pa = PTE_ADDR(*pte);
        flags = PTE_FLAGS(*pte);
        if((mem = kalloc()) == 0)
            goto bad;
        memmove(mem, (char*)P2V(pa), PGSIZE);
        if(mappages(d, (void*)i, PGSIZE, V2P(mem), flags) < 0) {
            kfree(mem);
            goto bad;
        }
    }
    return d;
bad:
    freevm(d);
    return 0;
}

```

understanding fork()->copyuvvm()

- **Map kernel pages**
- **for every page in parent's VM address space**
 - **allocate a PTE for child**
 - **set flags**
 - **copy data**
 - **map pages in child's page directory/tables**

understanding fork()

```
np->tf->eax = 0;  
for(i = 0; i < NOFILE; i++)  
    if(curproc->ofile[i])  
        np->ofile[i] = filedup(curproc->ofile[i]);  
    np->cwd = idup(curproc->cwd);  
    safestrcpy(np->name, curproc->name, sizeof(curproc->name));  
    pid = np->pid;  
    acquire(&phtable.lock);  
    np->state = RUNNABLE;  
    release(&phtable.lock);
```

- **set return value of child to 0**
 - eax contains return value, it's on TF
- **copy each struct file**
- **copy current working dir inode**
- **copy name**
- **set pid of child**
- **set child “RUNNABLE”**

exec() - different prototype

- **int exec(char*, char**);**
 - usage: to print README and test.txt using “cat”

```
int main(int argc, char *argv[])
{
    char *cmd = "/cat";
    char *argstr[4] = { "/cat", "README",
"test.txt", 0};
    exec(cmd, argstr);
}
```

note: to really run this code in xv6, you need to make changes to Makefile. First, add this program to UPROGS, then write a file test.txt using Linux, and add ‘test.txt’ to list of files in ‘mkfs’ target in Makefile

```

int
sys_exec(void)
{
    char *path, *argv[MAXARG];
    int i;
    uint uargv, uarg;
    if(argstr(0, &path) < 0 || argint(1, (int*)&uargv) < 0){
        return -1;
    }
    memset(argv, 0, sizeof(argv));
    for(i=0;; i++){
        if(i >= NELEM(argv))
            return -1;
        if(fetchint(uargv+4*i, (int*)&uarg) < 0)
            return -1;
        if(uarg == 0){
            argv[i] = 0;
            break;
        }
        if(fetchstr(uarg, &argv[i]) < 0)
            return -1;
    }
    return exec(path, argv);
}

```

sys_exec()

- **argstr(n,), argint(n,)**
 - Fetch the n'th argument from *process stack* using p->tf->esp + offset
 - Again: revise calling conventions
- **0'th argument: name of executable file**
- **1st Argument: address of the array of arguments**
 - store in *uargv*

```

int sys_exec(void)
{
    char *path, *argv[MAXARG];
    int i;  uint uargv, uarg;
    if(argstr(0, &path) < 0 || argint(1,
(int*)&uargv) < 0){
        return -1;
    }
    memset(argv, 0, sizeof(argv));
    for(i=0;; i++){
        if(i >= NELEM(argv))  return -1;
        if(fetchint(uargv+4*i, (int*)&uarg) < 0)
            return -1;
        if(uarg == 0){
            argv[i] = 0;    break;
        }
        if(fetchstr(uarg, &argv[i]) < 0)
            return -1;
    }
    return exec(path, argv);
}

```

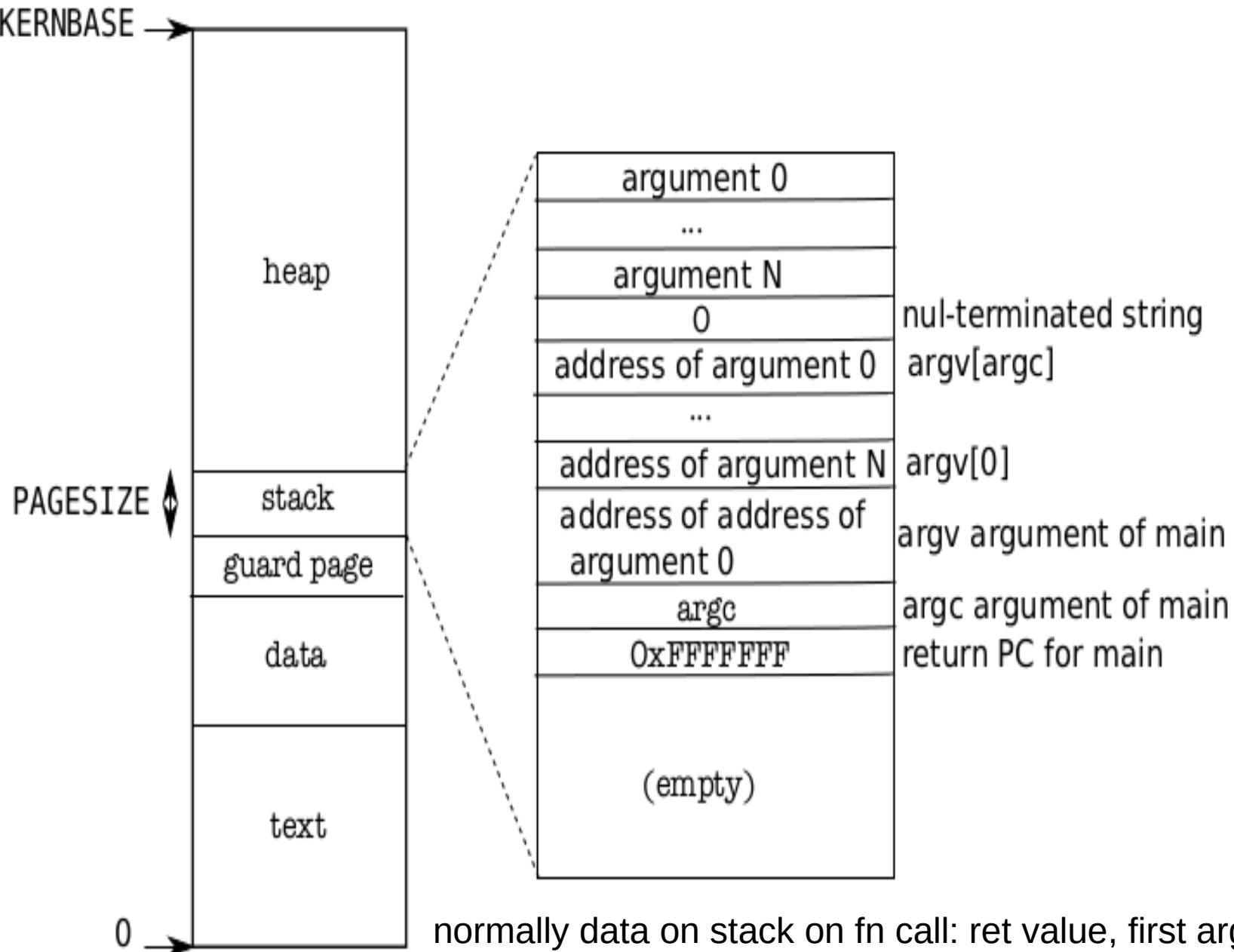
sys_exec()

- the local array argv[] (allocated on kernel stack, obviously) set to 0
- fetch every next argument from array of arguments
 - Sets the address of argument in argv[1]
- call exec
 - beware: mistake to assume that this exec() is the exec() called from user code! NO!

What should exec() do?

- **Remember, it came from fork()**
 - so proc & within it tf, context, kstack, pgdir-tables-pages, all exist.
 - Code, stack pages exist, and mappings exist through proc->pgdir
- **Hence**
 - read the ELF executable file (argv[0])
 - create a new page dir – create mappings for kernel and user code+data; copy data from ELF to these pages (later discard old pagedir)
 - Copy the argv onto the user stack – so that when new process starts it has its main(argc, argv[]) built
 - set values of other fields in proc to start program correctly

User stack after call to exec() is over



exec()

```
int
exec(char *path, char **argv)
{
...
    uint argc, sz, sp,
ustack[3+MAXARG+1];
...
if((ip = namei(path)) == 0){
    end_op();
    cprintf("exec: fail\n");
    return -1;
}
```

- **ustack**
 - used to build the arguments to be pushed on user-stack
- **namei**
 - get the inode of the executable file

exec()

```
// Check ELF header  
  
if(readi(ip, (char*)&elf, 0,  
sizeof(elf)) != sizeof(elf))  
    goto bad;  
  
if(elf.magic != ELF_MAGIC)  
    goto bad;  
  
if((pgdir = setupkvm()) == 0)  
    goto bad;
```

- **readi**
 - read ELF header
- **setupkvm()**
 - creating a *new* page directory and mapping kernel pages

```

sz = 0;
for(i=0, off=elf.phoff; i<elf.phnum; i++, off+=sizeof(ph)){
    if(readi(ip, (char*)&ph, off, sizeof(ph)) != sizeof(ph))
        goto bad;
    if(ph.type != ELF_PROG_LOAD)
        continue;
    if(ph.memsz < ph.filesz)
        goto bad;
    if(ph.vaddr + ph.memsz < ph.vaddr)
        goto bad;
    if((sz = allocuvvm(pgdир, sz, ph.vaddr + ph.memsz)) == 0)
        goto bad;
    if(ph.vaddr % PGSIZE != 0)
        goto bad;
    if(loaduvvm(pgdир, (char*)ph.vaddr, ip, ph.off, ph.filesz) <
0)
        goto bad;
}

```

exec()

- Read ELF program headers from ELF file
- Map the code/data into pagedir-pagetable-pages
- Copy data from ELF file into the pages allocated

exec()

```
sz = PGROUNDUP(sz);
if((sz = allocuvm(pgdir, sz, sz +
2*PGSIZE)) == 0)
    goto bad;
clearpteu(pgdir, (char*)(sz -
2*PGSIZE));
sp = sz;
```

- Allocate 2 pages on top of proc->sz
- One page for stack
- one page for guard page
- Clear the valid flag on guard page

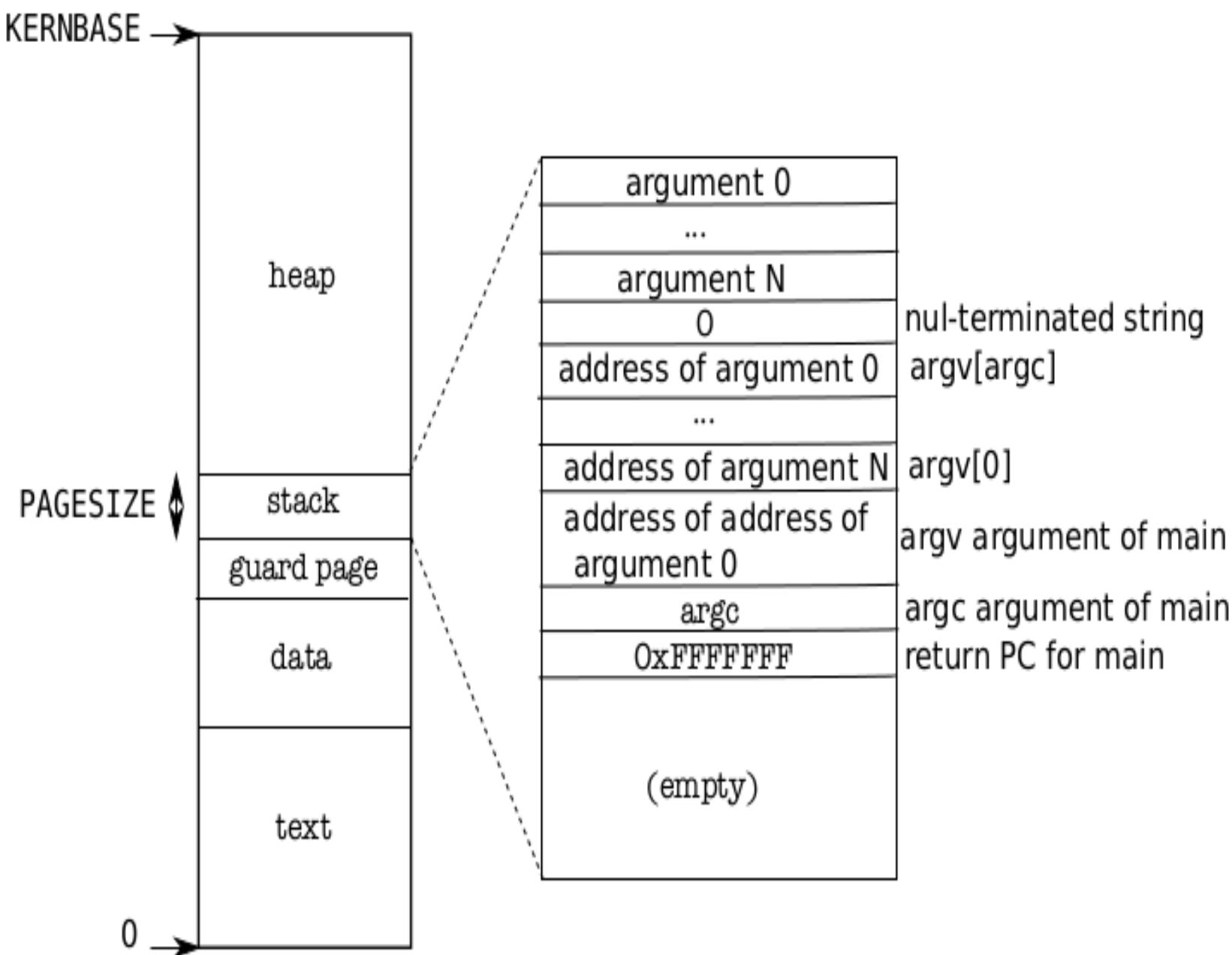
// Push argument strings, prepare rest of stack
in ustack.

```
for(argc = 0; argv[argc]; argc++) {  
    if(argc >= MAXARG)  
        goto bad;  
  
    sp = (sp - (strlen(argv[argc]) + 1)) & ~3;  
  
    if(copyout(pkdir, sp, argv[argc],  
    strlen(argv[argc]) + 1) < 0)  
        goto bad;  
  
    ustack[3+argc] = sp;  
}  
  
ustack[3+argc] = 0;  
ustack[0] = 0xffffffff; // fake return PC  
ustack[1] = argc;  
ustack[2] = sp - (argc+1)*4; // argv pointer  
sp -= (3+argc+1) * 4;  
  
if(copyout(pkdir, sp, ustack, (3+argc+1)*4) < 0)  
    goto bad;
```

exec()

- For each entry in argv[]
 - copy it on user-stack
 - remember it's location on user stack in ustack
- add extra entries (to be copied to user stack) to ustack
- copy argc, argv pointer
- take sp to bottom
- copy ustack to user stack

This is
what the
code on
earlier
slide did



// Save program name for debugging.

```
for(last=s=path; *s; s++)
```

```
if(*s == '/')
```

```
last = s+1;
```

```
safestrcpy(curproc->name, last,  
sizeof(curproc->name));
```

// Commit to the user image.

```
oldpgdir = curproc->pgdir;
```

```
curproc->pgdir = pgdir;
```

```
curproc->sz = sz;
```

```
curproc->tf->eip = elf.entry; // main
```

```
curproc->tf->esp = sp;
```

```
switchuvm(curproc);
```

```
freevm(oldpgdir);
```

```
return 0;
```

exec()

- copy name of new process in proc->name
- change to new page directory
- change new size
- tf->eip will be used when we return from exec() to jump to user code. Set to first instruction of code, given by elf.entry
- Set user stack pointer to “sp” (bottom of stack of arguments)
- Update TSS, change CR3 to newpagedir
- free old page dir

return 0 from exec()?

- We know exec() does not return !
- This was exec() function !
 - Returns to sys_exec()
- sys_exec() also returns , where?
 - Remember we are still in kernel code, running on kernel stack.
p->kstack has the trapframe setup
 - There is context struct on stack. Why?
 - sys_exec() returns to trapret(), the trap frame will be popped !
 - with “iret” jump into new program !
 - New program is not old program , which could have accessed
return value of sys_exec()