

Compilation, Linking, Loading

Abhijit A M

Review of last few lectures

Boot sequence: BIOS, boot-loader, kernel

Boot sequence: Process world

kernel->init -> many forks+execs() ->

Hardware interrupts, system calls, exceptions

Event driven kernel

System calls

Fork, exec, ... open, read, ...

What are compiler, assembler, linker and loader, and C library

System Programs/Utilities

Most essential to make a kernel really usable

Standard C Library

A collection of some of the most frequently needed functions for C programs

`scanf`, `printf`, `getchar`, system-call wrappers (`open`, `read`, `fork`, `exec`, etc.), ...

An machine/object code file containing the machine code of all these functions

Not a source code! Neither a header file. More later.

Where is the C library on your computer?

`/usr/lib/x86_64-linux-gnu/libc-2.31.so`

Compiler

application program, which converts one (programming) language to another

Most typically compilers convert a high level language like C, C++, etc. to Machine code language

E.g. GCC /usr/bin/gcc



Usage: e.g.

```
$ gcc main.c -o main
```

Here main.c is the C code, and "main" is the object/machine code file generated

Assembler

application program, converts assembly code into machine code

What is assembly language?

Human readable machine code language

E.g. x86 assembly code

mov 50, r1

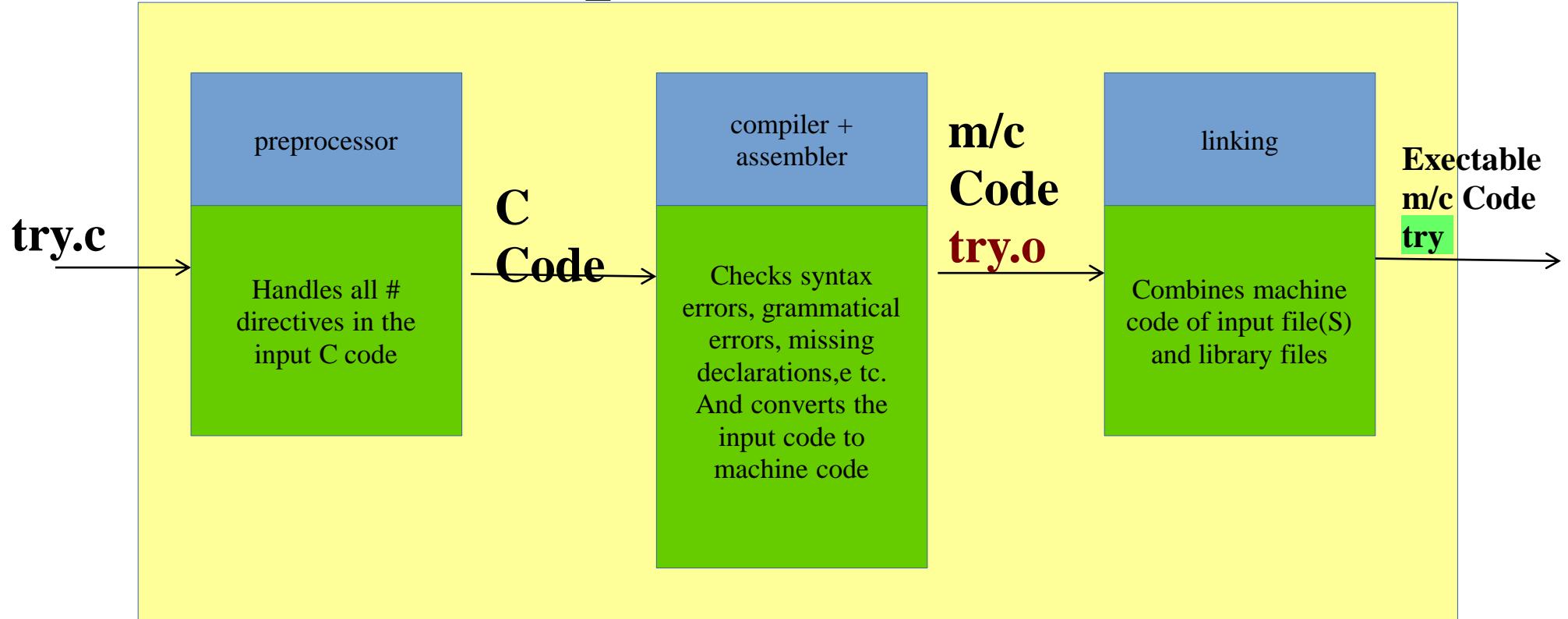
add 10, r1

mov r1, 500



Usage: cc

Compilation Process



gcc

Example

try.c

```
#include <stdio.h>
#define MAX 30
int f(int, int);
int main() {
    int i, j, k;
    scanf("%d%d", &i, &j);
    k = f(i, j) + MAX;
    printf("%d\n", k);
    return 0;
}
```

f.c

```
int g(int);
#define ADD(a, b) (a + b)
int f(int m, int n) {
    return ADD(m,n) + g(10);
}
```

g.c

```
int g(int x) {
    return x + 10;
}
```

Try these commands, observe the output/errors/warnings, and try to und

```
$ gcc try.c
$ gcc -c try.c
$ gcc -c f.c
$ gcc -c g.c
$ gcc try.o f.o g.o -o try
$ gcc -E try.c
$ gcc -E f.c
```

More about the steps

Pre-processor

#define ABC XYZ

cut ABC and paste XYZ

include <stdio.h>

copy-paste the file stdio.h

There is no CODE in stdio.h, only typedefs, #includes, #define, #ifdef, etc.

Linking

Normally links with the standard C-library by default

To link with other libraries, use the -l option of gcc

```
cc main.c -lm -lncurses -o main # links with libm.so and libncurses.so
```

Using gcc itself to understand the process

Run only the preprocessor

`cc -E test.c`

Shows the output on the screen

Run only till compilation (no linking)

`cc -c test.c`

Generates the “test.o” file , runs compilation + assembler

`gcc -S main.c`

One step before machine code generation, stops at assembly code

Combine multiple .o files (only linking part)

`cc test.o main.o try.o -o something`

Linking process

Linker is an application program

On linux, it's the "ld" program

E.g. you can run commands like \$ ld a.o b.o -o c.o

Normally you have to specify some options to ld to get a proper executable file.

When you run gcc

\$ cc main.o f.o g.o -o try

the CC will internally invoke "ld" . ld does the job of linking

The resultatnt file "try" here, will contain the codes of all the functions and linkages also.

What is linking?

"connecting" the call of a function with the code of the function.

What happens with the code of printf()?

The linker or CC will automatically pick up code from the libc.so.6 file for the functions.

Executable file format

An executable file needs to execute in an environment created by OS and on a particular processor

Contains machine code + other information for OS

Need for a structured-way of storing machine code in it

Different OS demand different formats

Windows: PE, Linux: ELF, Old Unixes: a.out, etc.

ELF : The format on Linux.

Try this

```
$ file /bin/ls
```

```
$ file /usr/lib/x86_64-linux-gnu/libc-2.31.so
```

```
$ file a.out # on any a.out created by you
```

Exec() and ELF

When you run a program

```
$ ./try
```

Essentially there will be a fork() and exec("./try", ...)

So the kernel has to read the file "./try" and understand it.

So each kernel will demand its own object code file format.

Hence ELF, EXE, etc. Formats

ELF is used not only for executable (complete machine code) programs, but also for partially compiled files e.g. main.o and library files like libc.so.6

What is a.out?

"a.out" was the name of a format used on earlier Unixes.

It so happened that the early compiler writers, also created executable with default name 'a.out'

Utilities to play with object code files

objdump

```
$ objdump -D -x /bin/ls
```

Shows all disassembled machine instructions
and “headers”

hexdump

```
$ hexdump /bin/ls
```

Just shows the file in hexadecimal

readelf

Alternative to objdump

ar

To create a “statically linked” library file

```
$ ar -crs libmine.a one.o two.o
```

Gcc to create shared library

```
$ gcc hello.o -shared -o libhello.so
```

To see how gcc invokes as, ld, etc; do this

```
$ gcc -v hello.c -o hello
```

/* <https://stackoverflow.com/questions/1170809/how-to-get-gcc-linker-command> */

Linker, Loader, Link-Loader

Linker or linkage-editor or link-editor

The “ld” program. Does linking.

Loader

The exec(). It loads an executable in the memory.

Link-Loader

Often the linker is called link-loader in literature. Because where were days when the linker and loader’s jobs were quite over-lapping.

Static, dynamic / linking, loading

**Both linking and loading can be
Static or dynamic**

More about this when we learn memory management

An important fundamental:

memory management features of processor, memory management architecture of kernel, executable/object-code file format, output of linker and job of loader, are all interdependent and in-separable.

They all should fit into each other to make a system work

Cross-compiler

Compiler on system-A, but generate object-code file for system-B (target system)

E.g. compile on Ubuntu, but create an EXE for windows

Normally used when there is no compiler available on target system

see gcc -m option

See https://wiki.osdev.org/GCC_Cross-Compiler

Calling Convention

Abhijit A M

The need for calling convention

An essential task of the compiler

Generates object code (file) for given source code (file)

Processors provide simple features

Registers, machine instructions (add, mov, jmp, call, etc.),
imp registers like stack-pointer, etc; ability to do
byte/word sized operations

No notion of data types, functions, variables, etc.

But languages like C provide high level features

Data types, variables, functions, recursion, etc

The need for calling convention

**Examples of some of the challenges before the compiler
“call” + “ret” does not make a C-function call!**

**A “call” instruction in processor simply does this
Pushes IP(that is PC) on stack + Jumps to given address
This is not like calling a C-function !**

Unsolved problem: How to handle parameters, return value?

Processor does not understand data types!

Although it has instructions for byte, word sized data and

Compiler and Machine code generation

Example, code inside a function

```
int a, b, c;
```

```
c = a + b;
```

What kind of code is generated by compiler for this?

sub 12, <esp> #normally local variables are located on stack, make space

mov <location of a in memory>, r1 #location is on stack, e,g. -4(esp)

mov <location of b in memory>, r2

Compiler and Machine code generation

Across function calls

```
int f(int m, n) {  
    int x = m, y = n;  
    return g(x, y);  
}
```

```
int x(int a) {  
    return g (a, a+ 1);  
}
```

```
int g(int p, int q) {
```

**g() may be called from f() or
from x()**

**Sequence of function calls
can NOT be predicted by
compiler**

**Compiler has to generate
machine code for each
function assuming nothing
about the caller**

Compiler and Machine code generation

Machine code generation for functions

Mapping C language features to existing machine code instructions.

Typical examples

a =100 ; ==> mov instruction

a = b+ c; ==> mov, add instructions

while(a < 5) { j++; } ==> mov, cmp, jlt, add, etc. Instruction

Where are the local variables in memory?

The only way to store them is on a stack.

Function calls

LIFO

Last in First Out

Must need a “stack” like feature to implement them

Processor Stack

Processors provide a stack pointer

%esp on x86

Instructions like push and pop are provided by hardware

They automatically increment/decrement the stack
pointer. On x86 stack grows downwards (subtract from

Function calls

System stack, compilers, Languages

Compilers generate machine code with the 'esp'. The pointer is initialized to a proper value at the time of fork-exec by the OS for each process.

Languages like C which provide for function calls, and recursion also assume that they will run on processors with a stack support in hardware

Convention needed

How to use the stack for effective implementation of function calls ?

Activation Record

Local Vars + parameters + return address

When functions call each other

One activation record is built on stack for each function call

On function return, the record is destroyed

On x86

ebp and esp pointers are used to denote the activation record.

How? We will see soon. You may start exploring with “gcc

X86 instructions

leave

Equivalent to

**mov %ebp, %esp # esp = push %eip
ebp**

pop %ebp

ret

Equivalent to

pop %ecx

jmp %ecx

call x

Equivalent to

**push %eip
jmp x**

X86 instructions

endbr64

Normally a NOP

Let's see some examples now

Let's compile using

gcc -S

See code and understand

simple.c and simple.s

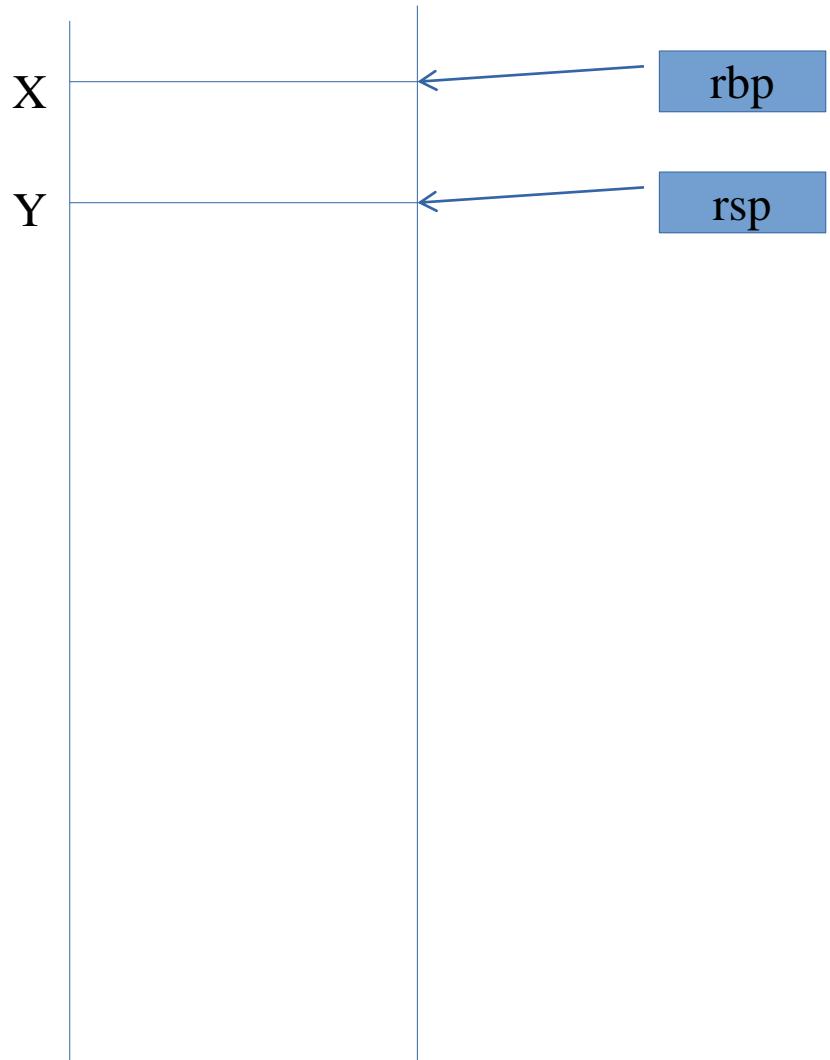
```
int f(int x) {  
    int y;  
    y = x + 3;  
    return y;  
}  
int main() {  
    int a = 20, b = 30;  
    b = f(a);  
    return b;  
}
```

main:

```
    endbr64  
    pushq %rbp  
    movq %rsp, %rbp  
    subq $16, %rsp  
    movl $20, -8(%rbp)  
    movl $30, -4(%rbp)  
    movl -8(%rbp), %eax  
    movl %eax, %edi  
    call f  
    movl %eax, -4(%rbp)  
    movl -4(%rbp), %eax  
    leave  
    ret
```

f:

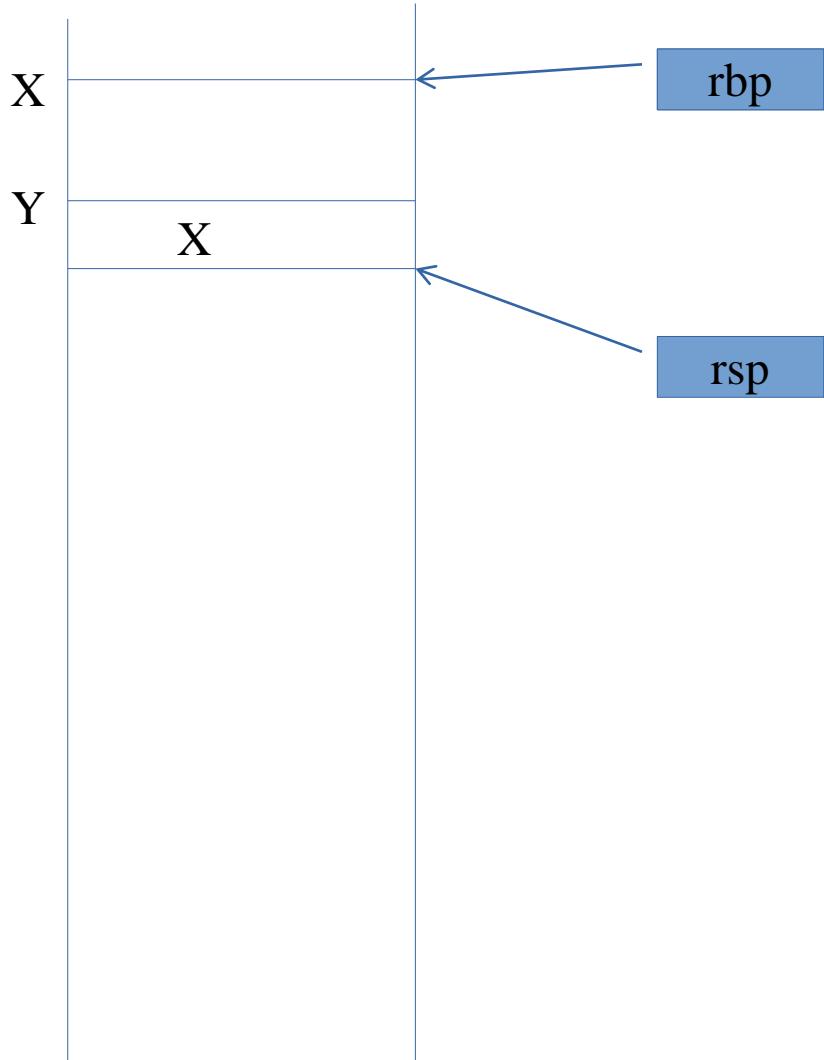
```
    endbr64  
    pushq %rbp  
    movq %rsp, %rbp  
    movl %edi, -20(%rbp)  
    movl -20(%rbp), %eax  
    addl $3, %eax  
    movl %eax, -4(%rbp)  
    movl -4(%rbp), %eax  
    popq %rbp  
    ret
```



main:

```
endbr64
pushq %rbp
movq %rsp, %rbp
subq $16, %rsp
movl $20, -8(%rbp)
movl $30, -4(%rbp)
movl -8(%rbp), %eax
movl %eax, %edi
call f
movl %eax, -4(%rbp)
movl -4(%rbp), %eax
leave
ret
```

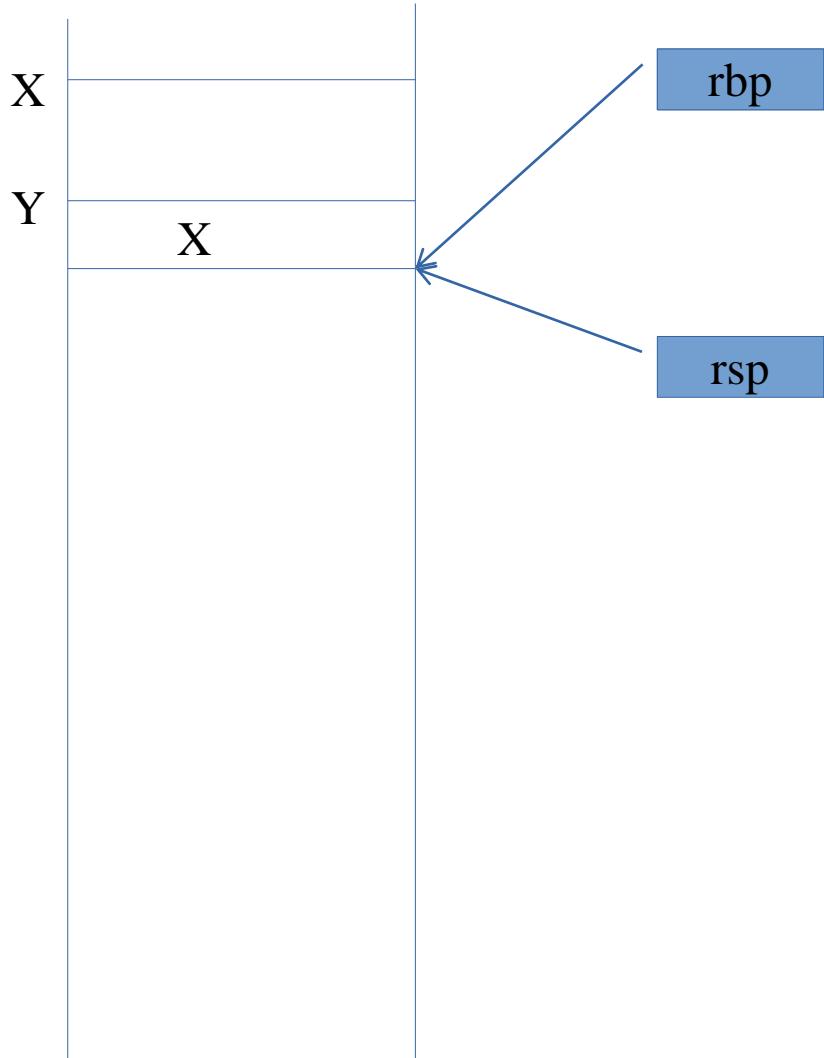
```
int main() {
    int a = 20, b = 30;
    b = f(a);
    return b;
}
```



main:

```
endbr64
pushq %rbp
movq %rsp, %rbp
subq $16, %rsp
movl $20, -8(%rbp)
movl $30, -4(%rbp)
movl -8(%rbp), %eax
movl %eax, %edi
call f
movl %eax, -4(%rbp)
movl -4(%rbp), %eax
leave
ret
```

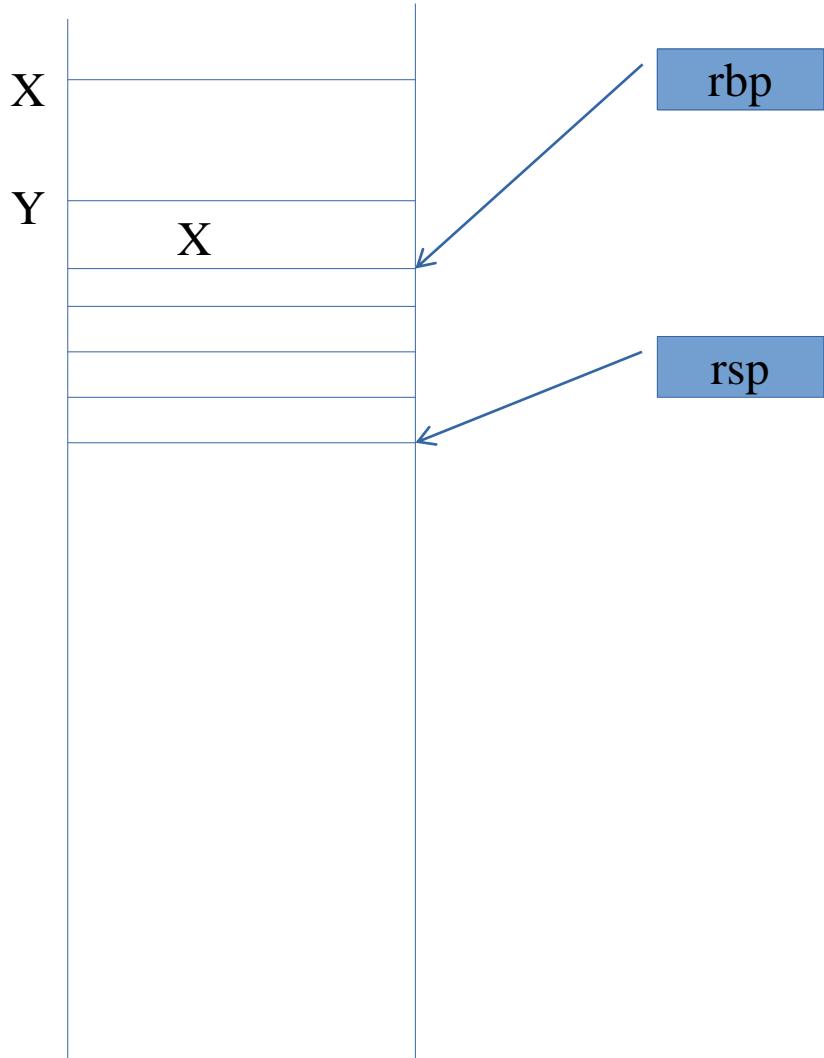
```
int main() {
    int a = 20, b = 30;
    b = f(a);
    return b;
}
```



main:

```
endbr64  
pushq %rbp  
movq %rsp, %rbp  
subq $16, %rsp  
movl $20, -8(%rbp)  
movl $30, -4(%rbp)  
movl -8(%rbp), %eax  
movl %eax, %edi  
call f  
movl %eax, -4(%rbp)  
movl -4(%rbp), %eax  
leave  
ret
```

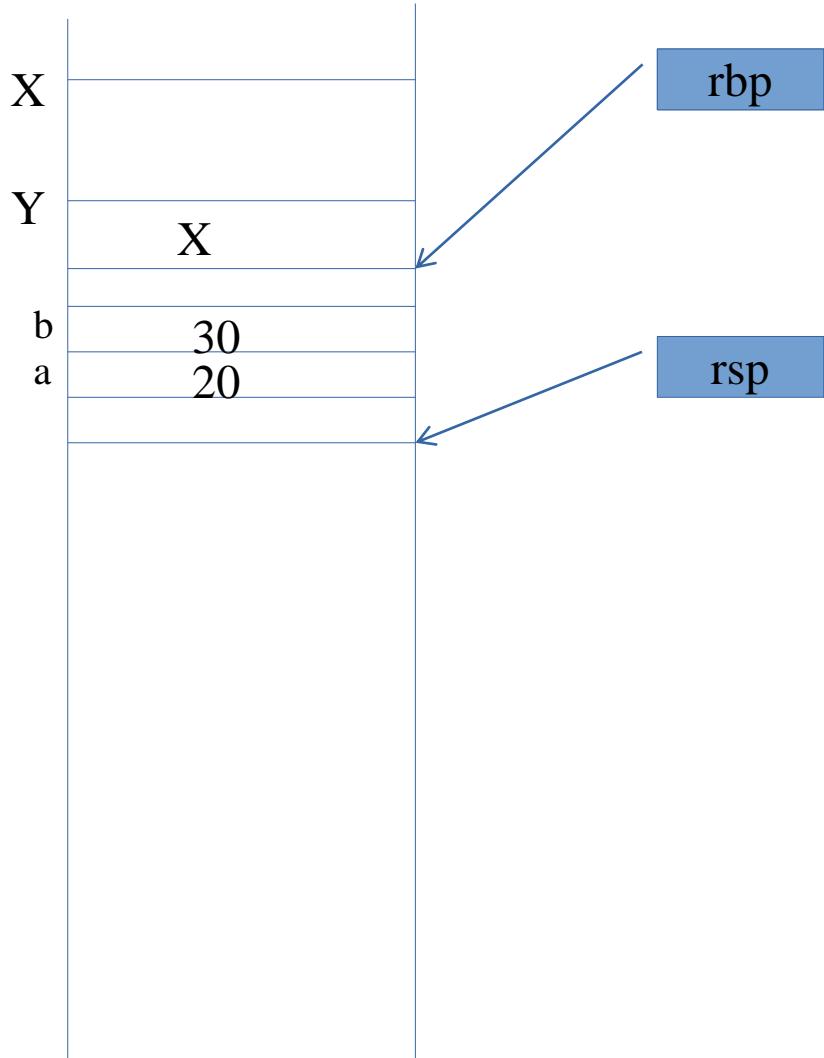
```
int main() {  
    int a = 20, b = 30;  
    b = f(a);  
    return b;  
}
```



main:

```
endbr64  
pushq %rbp  
movq %rsp, %rbp  
subq $16, %rsp  
movl $20, -8(%rbp)  
movl $30, -4(%rbp)  
movl -8(%rbp), %eax  
movl %eax, %edi  
call f  
movl %eax, -4(%rbp)  
movl -4(%rbp), %eax  
leave  
ret
```

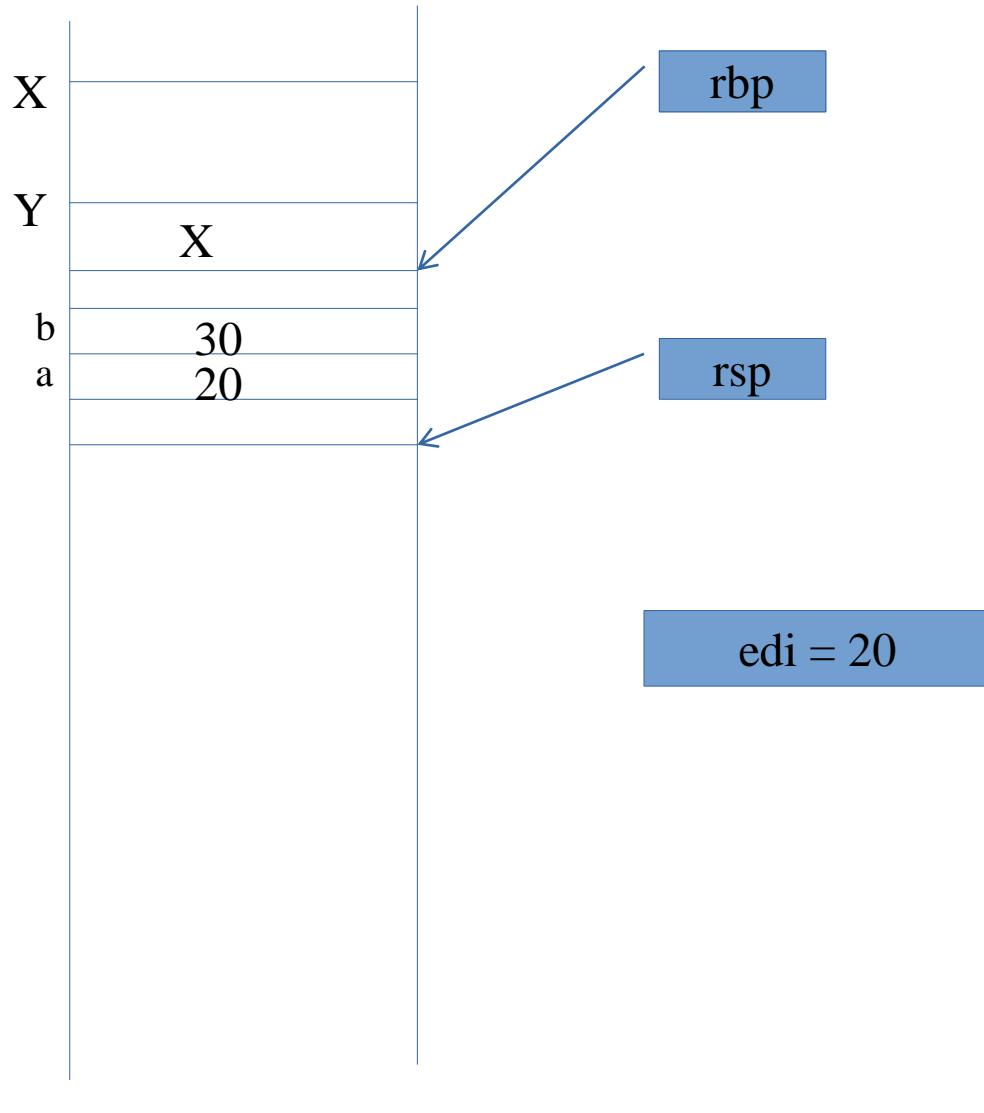
```
int main() {  
    int a = 20, b = 30;  
    b = f(a);  
    return b;  
}
```



main:

```
endbr64
pushq %rbp
movq %rsp, %rbp
subq $16, %rsp
movl $20, -8(%rbp)
movl $30, -4(%rbp)
movl -8(%rbp), %eax
movl %eax, %edi
call f
movl %eax, -4(%rbp)
movl -4(%rbp), %eax
leave
ret
```

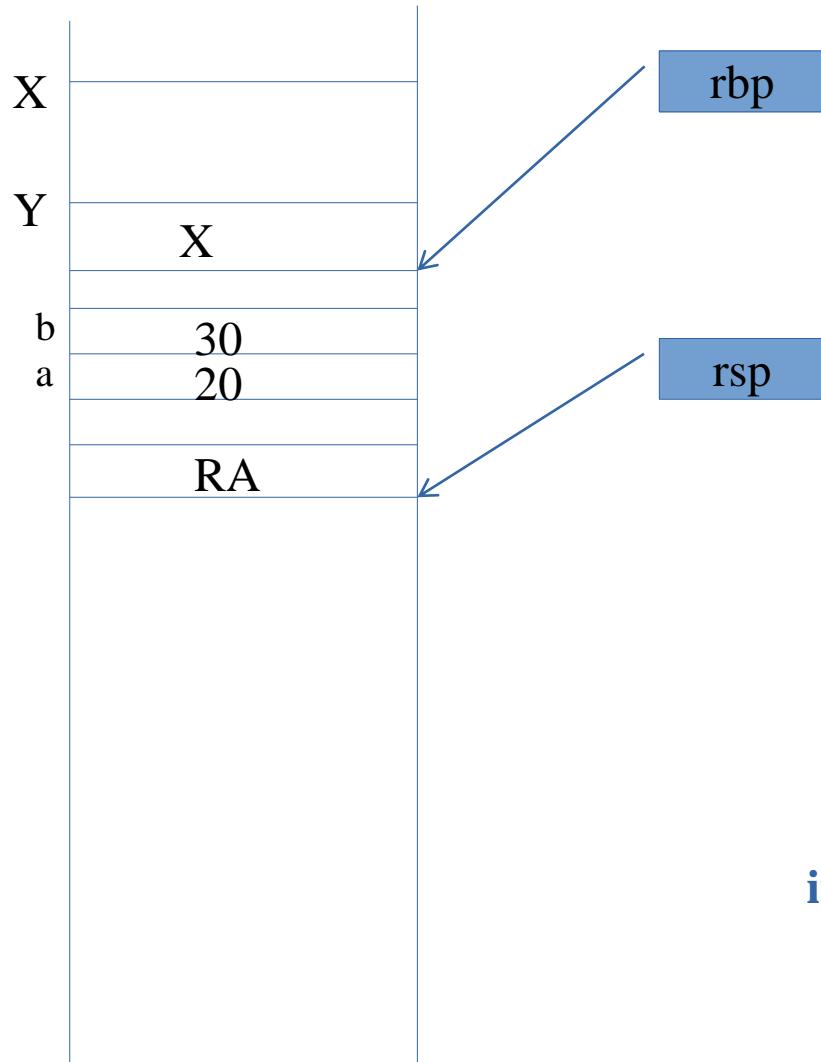
```
int main() {
    int a = 20, b = 30;
    b = f(a);
    return b;
}
```



main:

```
endbr64  
pushq %rbp  
movq %rsp, %rbp  
subq $16, %rsp  
movl $20, -8(%rbp)  
movl $30, -4(%rbp)  
movl -8(%rbp), %eax  
movl %eax, %edi  
call f  
movl %eax, -4(%rbp)  
movl -4(%rbp), %eax  
leave  
ret
```

```
int main() {  
    int a = 20, b = 30;  
    b = f(a);  
    return b;  
}
```



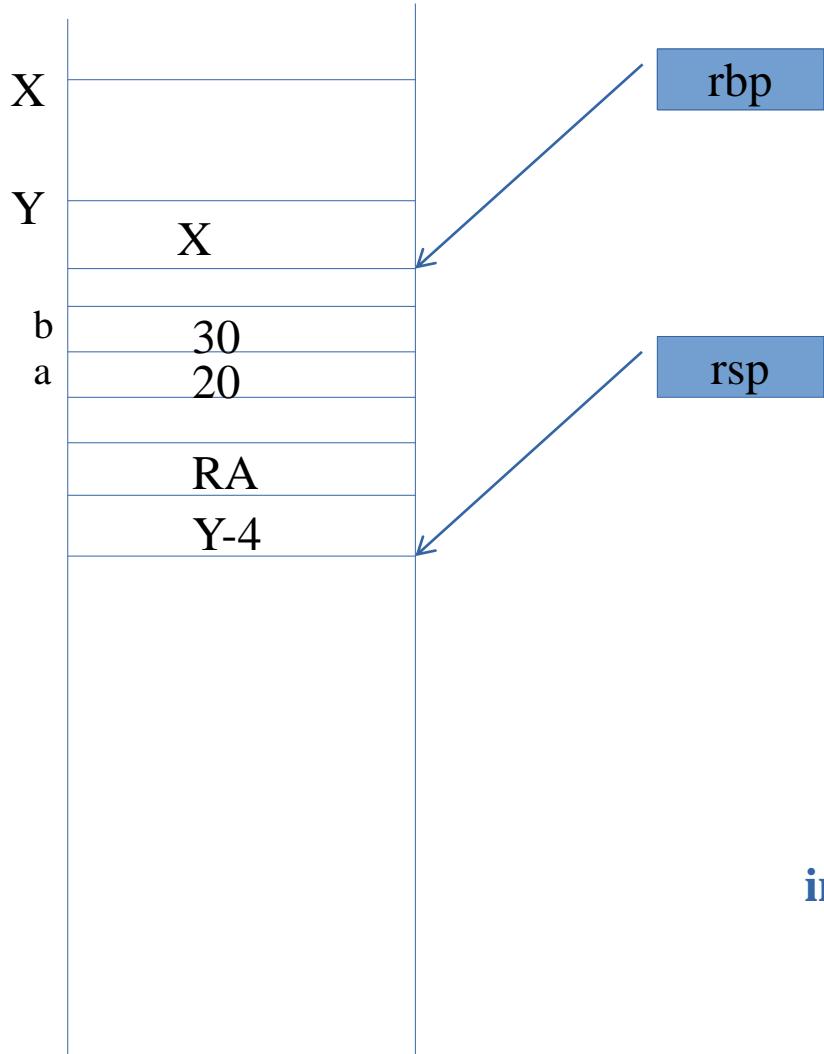
```
main:
endbr64
pushq %rbp
movq %rsp, %rbp
subq $16, %rsp
movl $20, -8(%rbp)
movl $30, -4(%rbp)
movl -8(%rbp), %eax
movl %eax, %edi
call f
movl %eax, -4(%rbp)
movl -4(%rbp), %eax
Leave
ret
```

```
int f(int x) {
    int y;
    y = x + 3;
    return y;
}
```

```
f:
endbr64
pushq %rbp
movq %rsp, %rbp
movl %edi, -20(%rbp)
movl -20(%rbp), %eax
addl $3, %eax
movl %eax, -4(%rbp)
movl -4(%rbp), %eax
Rpopq %rbp
ret
```

edi = 20

```
int main() {
    int a = 20, b = 30;
    b = f(a);
    return b;
}
```



```
main:
endbr64
pushq %rbp
movq %rsp, %rbp
subq $16, %rsp
movl $20, -8(%rbp)
movl $30, -4(%rbp)
movl -8(%rbp), %eax
movl %eax, %edi
call f
movl %eax, -4(%rbp)
movl -4(%rbp), %eax
Leave
ret
```

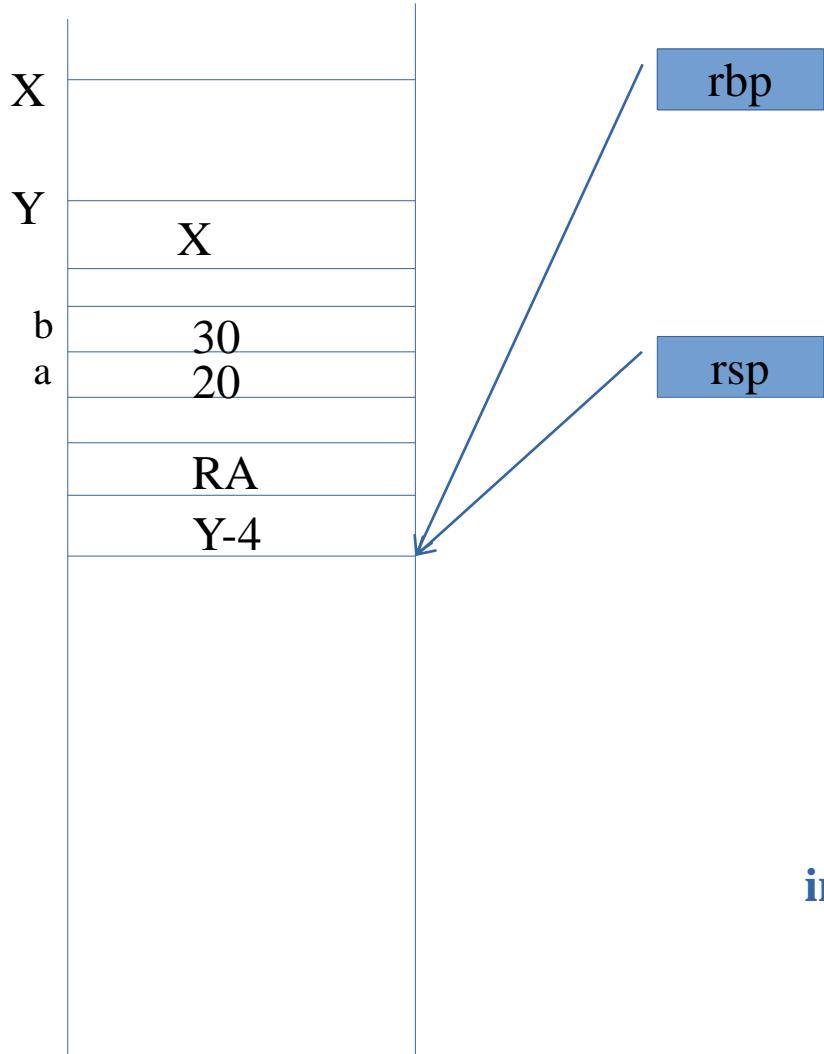
`f:`

```
endbr64
pushq %rbp
movq %rsp, %rbp
movl %edi, -20(%rbp)
movl -20(%rbp), %eax
addl $3, %eax
movl %eax, -4(%rbp)
movl -4(%rbp), %eax
Rpopq %rbp
ret
```

`edi = 20`

```
int f(int x) {
    int y;
    y = x + 3;
    return y;
}
```

```
int main() {
    int a = 20, b = 30;
    b = f(a);
    return b;
}
```



```

main:
endbr64
pushq %rbp
movq %rsp, %rbp
subq $16, %rsp
movl $20, -8(%rbp)
movl $30, -4(%rbp)
movl -8(%rbp), %eax
movl %eax, %edi
call f
movl %eax, -4(%rbp)
movl -4(%rbp), %eax
Leave
ret

```

```

int f(int x) {
    int y;
    y = x + 3;
    return y;
}

```

```

f:
endbr64
pushq %rbp
movq %rsp, %rbp
movl %edi, -20(%rbp)
movl -20(%rbp), %eax
addl $3, %eax
movl %eax, -4(%rbp)
movl -4(%rbp), %eax
Rpopq %rbp
ret

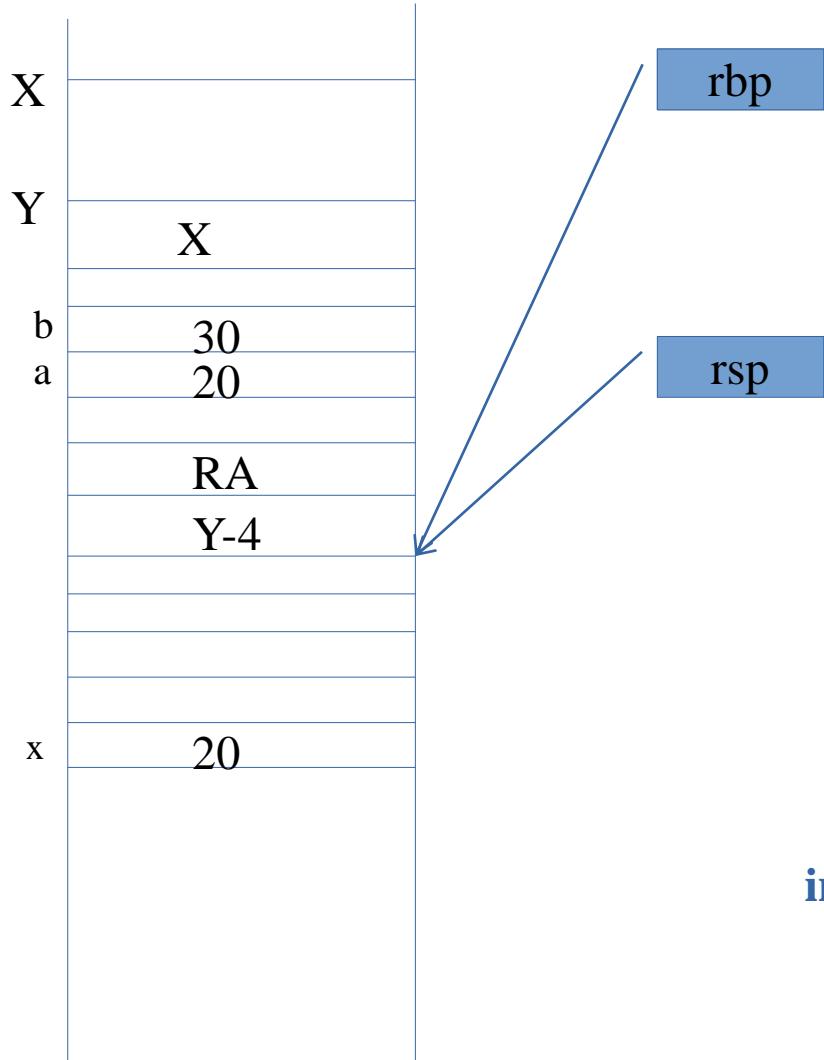
```

edi = 20

```

int main() {
    int a = 20, b = 30;
    b = f(a);
    return b;
}

```



```
main:
endbr64
pushq %rbp
movq %rsp, %rbp
subq $16, %rsp
movl $20, -8(%rbp)
movl $30, -4(%rbp)
movl -8(%rbp), %eax
movl %eax, %edi
call f
movl %eax, -4(%rbp)
movl -4(%rbp), %eax
Leave
ret
```

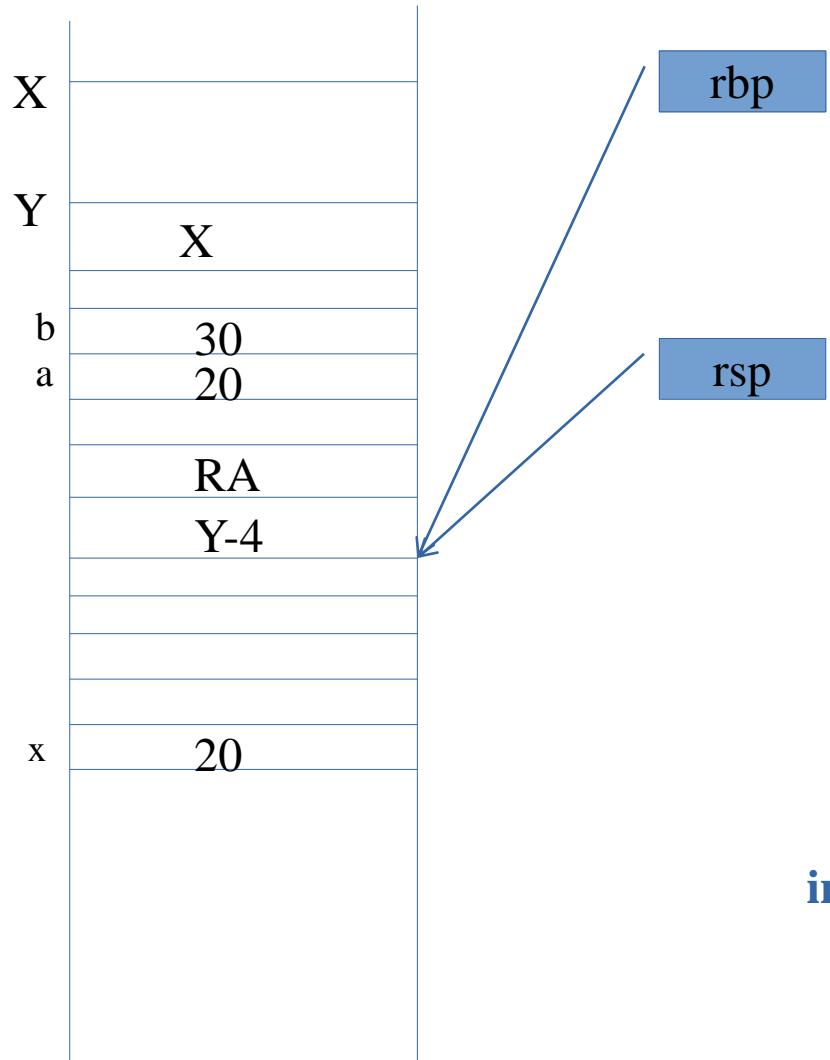
```
int f(int x) {
    int y;
    y = x + 3;
    return y;
}
```

f:

```
endbr64
pushq %rbp
movq %rsp, %rbp
movl %edi, -20(%rbp)
movl -20(%rbp), %eax
addl $3, %eax
movl %eax, -4(%rbp)
movl -4(%rbp), %eax
Rpopq %rbp
ret
```

edi = 20

```
int main() {
    int a = 20, b = 30;
    b = f(a);
    return b;
}
```



```
int f(int x) {
    int y;
    y = x + 3;
    return y;
}
```

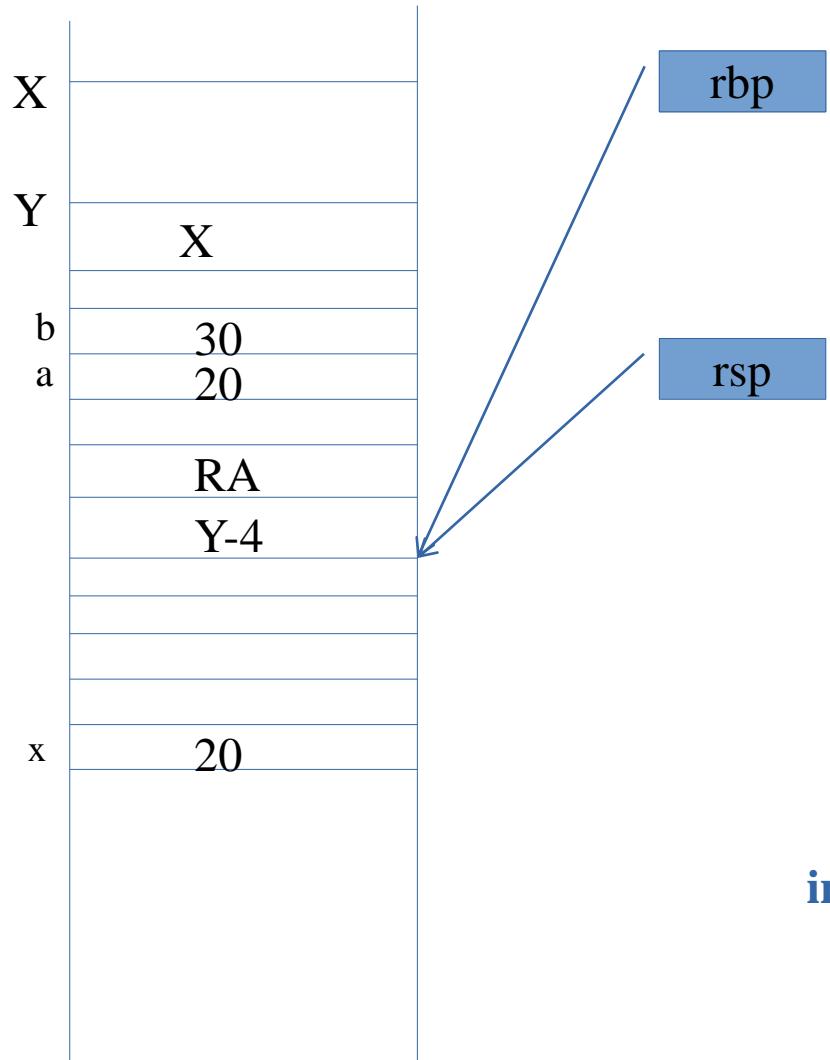
```
main:
endbr64
pushq %rbp
movq %rsp, %rbp
subq $16, %rsp
movl $20, -8(%rbp)
movl $30, -4(%rbp)
movl -8(%rbp), %eax
movl %eax, %edi
call f
movl %eax, -4(%rbp)
movl -4(%rbp), %eax
Leave
ret
```

```
f:
endbr64
pushq %rbp
movq %rsp, %rbp
movl %edi, -20(%rbp)
movl -20(%rbp), %eax
addl $3, %eax
movl %eax, -4(%rbp)
movl -4(%rbp), %eax
RApopq %rbp
ret
```

edi = 20

eax = 20

```
int main() {
    int a = 20, b = 30;
    b = f(a);
    return b;
}
```



```
int f(int x) {
    int y;
    y = x + 3;
    return y;
}
```

```
main:
endbr64
pushq %rbp
movq %rsp, %rbp
subq $16, %rsp
movl $20, -8(%rbp)
movl $30, -4(%rbp)
movl -8(%rbp), %eax
movl %eax, %edi
call f
movl %eax, -4(%rbp)
movl -4(%rbp), %eax
Leave
ret
```

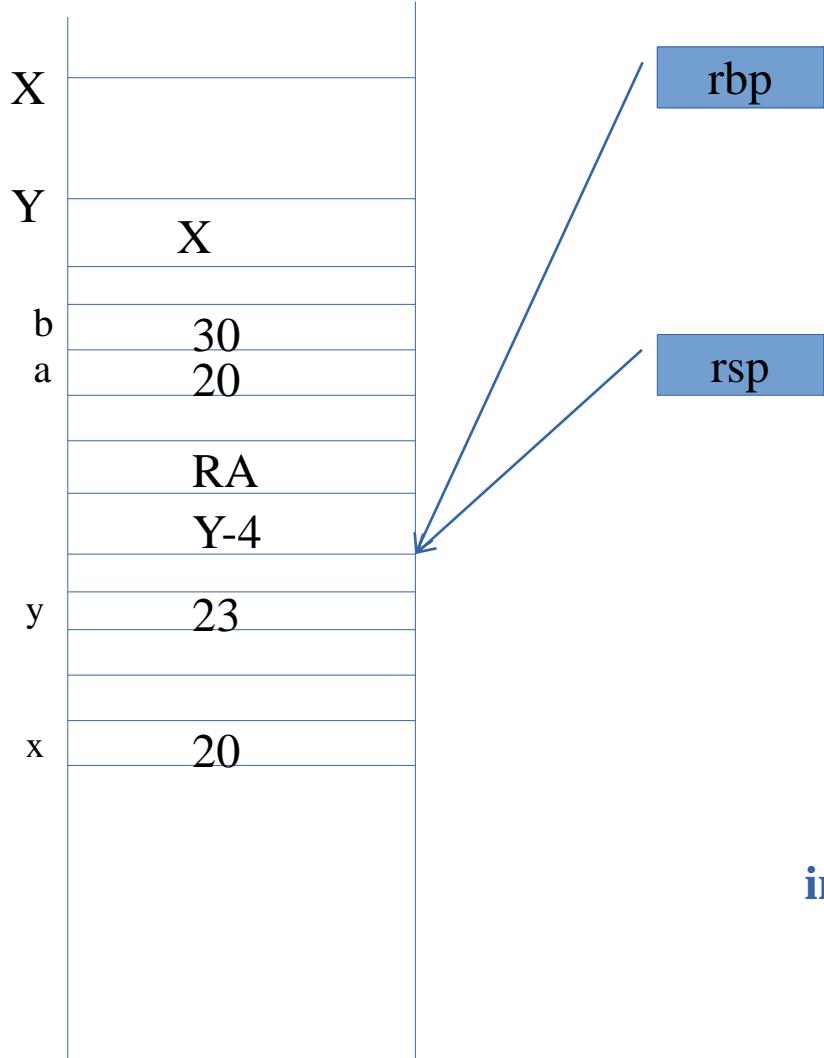
f:

```
endbr64
pushq %rbp
movq %rsp, %rbp
movl %edi, -20(%rbp)
movl -20(%rbp), %eax
addl $3, %eax
movl %eax, -4(%rbp)
movl -4(%rbp), %eax
RApopq %rbp
ret
```

edi = 20

eax = 23

```
int main() {
    int a = 20, b = 30;
    b = f(a);
    return b;
}
```



```
int f(int x) {
    int y;
    y = x + 3;
    return y;
}
```

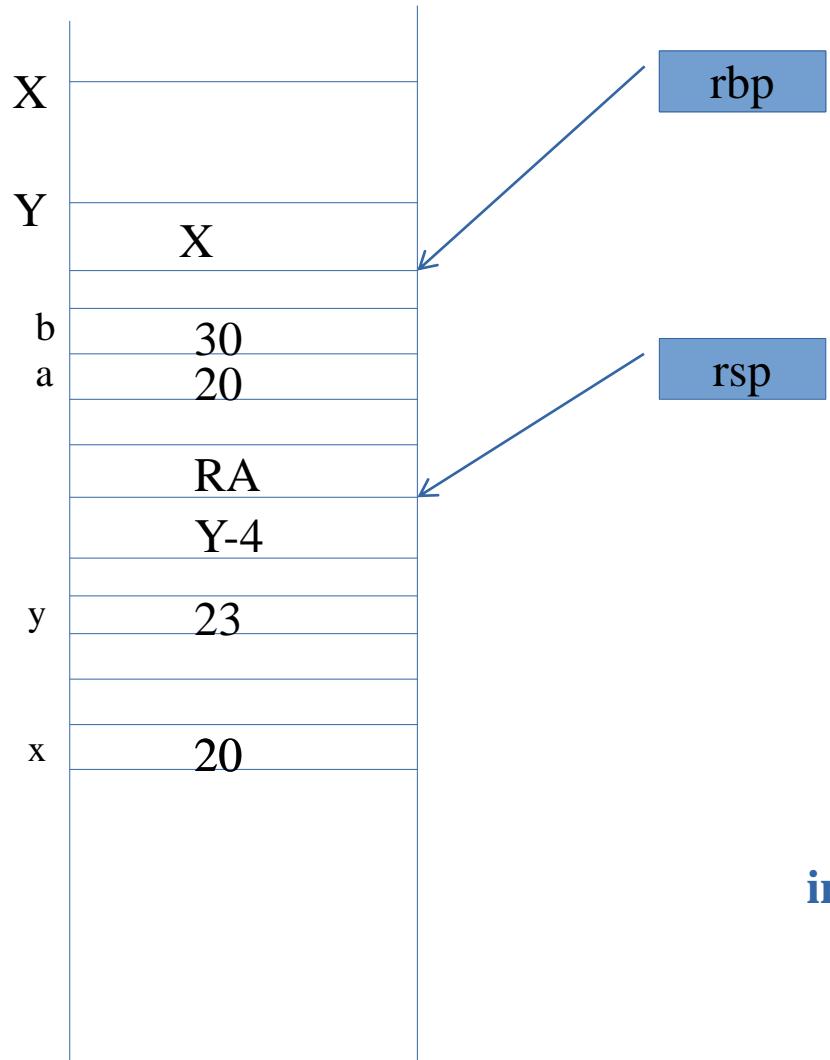
```
main:
endbr64
pushq %rbp
movq %rsp, %rbp
subq $16, %rsp
movl $20, -8(%rbp)
movl $30, -4(%rbp)
movl -8(%rbp), %eax
movl %eax, %edi
call f
movl %eax, -4(%rbp)
movl -4(%rbp), %eax
Leave
ret
```

```
f:
endbr64
pushq %rbp
movq %rsp, %rbp
movl %edi, -20(%rbp)
movl -20(%rbp), %eax
addl $3, %eax
movl %eax, -4(%rbp)
movl -4(%rbp), %eax
RApopq %rbp
ret
```

edi = 20

eax = 23

```
int main() {
    int a = 20, b = 30;
    b = f(a);
    return b;
}
```



```
main:
endbr64
pushq %rbp
movq %rsp, %rbp
subq $16, %rsp
movl $20, -8(%rbp)
movl $30, -4(%rbp)
movl -8(%rbp), %eax
movl %eax, %edi
call f
movl %eax, -4(%rbp)
movl -4(%rbp), %eax
Leave
ret
```

```
int f(int x) {
    int y;
    y = x + 3;
    return y;
}
```

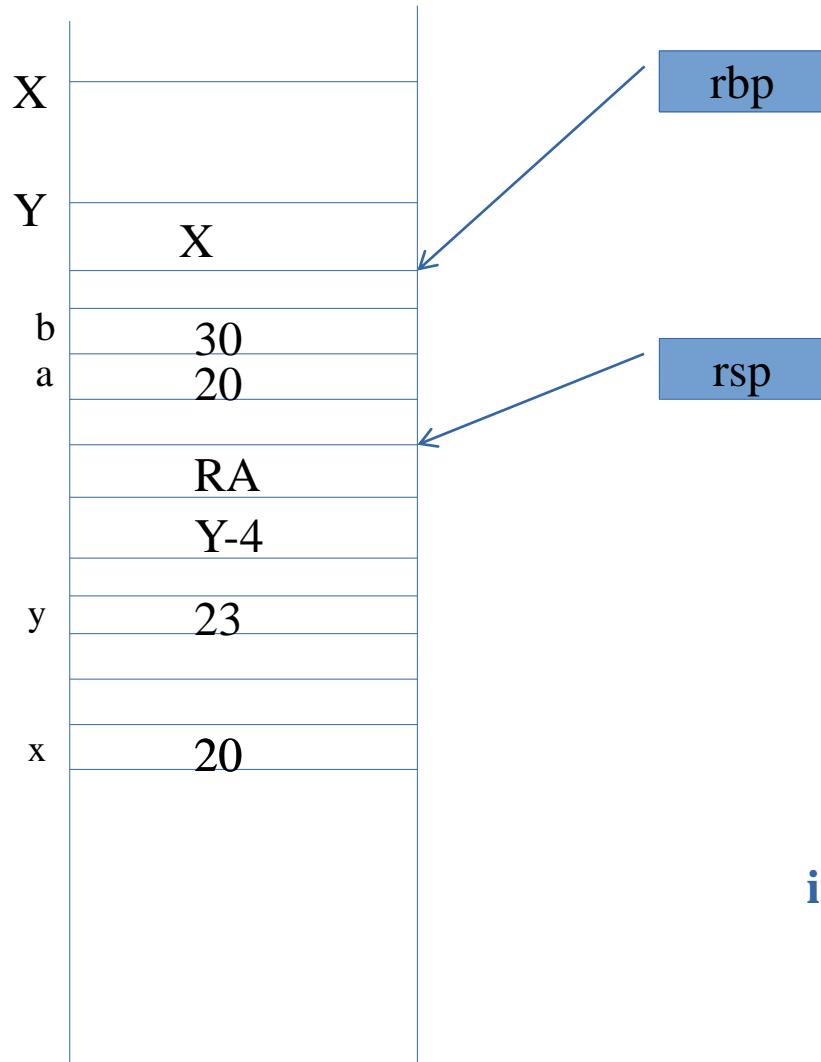
f:

```
endbr64
pushq %rbp
movq %rsp, %rbp
movl %edi, -20(%rbp)
movl -20(%rbp), %eax
addl $3, %eax
movl %eax, -4(%rbp)
movl -4(%rbp), %eax
RApopq %rbp
ret
```

edi = 20

Eax = 23

```
int main() {
    int a = 20, b = 30;
    b = f(a);
    return b;
}
```



```
main:
endbr64
pushq %rbp
movq %rsp, %rbp
subq $16, %rsp
movl $20, -8(%rbp)
movl $30, -4(%rbp)
movl -8(%rbp), %eax
movl %eax, %edi
call f
movl %eax, -4(%rbp)
movl -4(%rbp), %eax
Leave
ret
```

```
int f(int x) {
    int y;
    y = x + 3;
    return y;
}
```

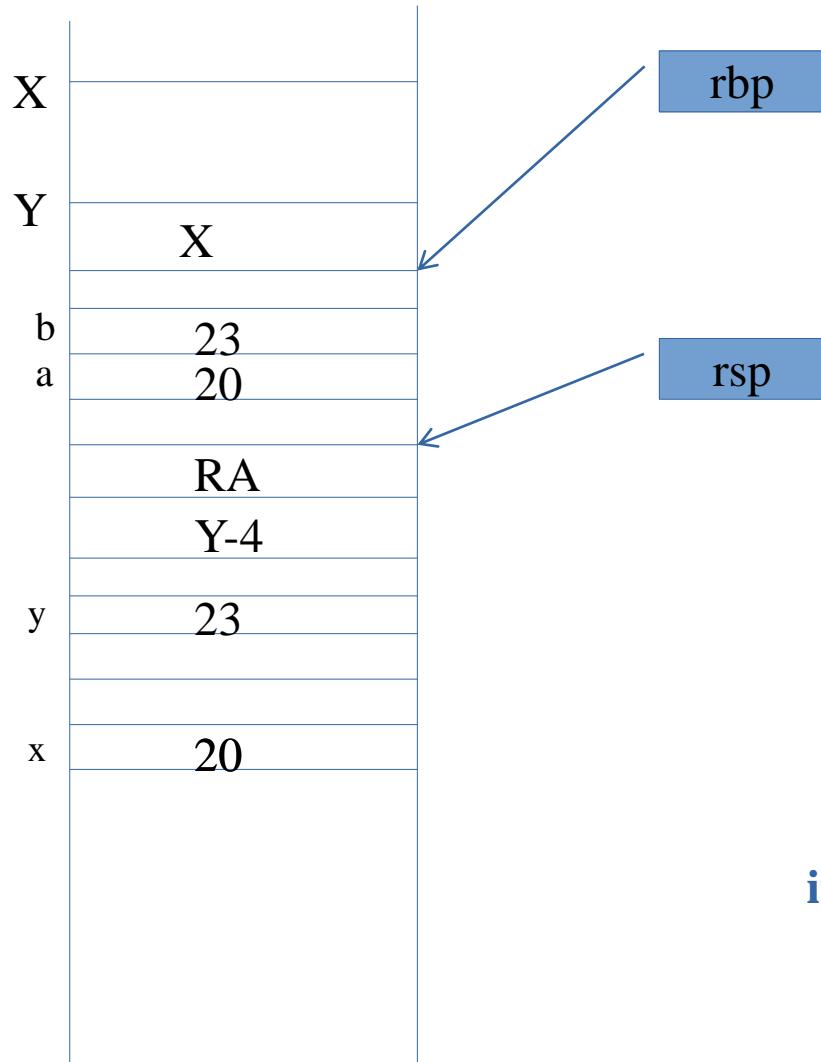
```
f:
endbr64
pushq %rbp
movq %rsp, %rbp
movl %edi, -20(%rbp)
movl -20(%rbp), %eax
addl $3, %eax
movl %eax, -4(%rbp)
movl -4(%rbp), %eax
RApopq %rbp
ret
```

edi = 20

eax = 23

```
int main() {
    int a = 20, b = 30;
    b = f(a);
    return b;
}
```

eip = RA



```
main:  
endbr64  
pushq %rbp  
movq %rsp, %rbp  
subq $16, %rsp  
movl $20, -8(%rbp)  
movl $30, -4(%rbp)  
movl -8(%rbp), %eax  
movl %eax, %edi  
call f  
movl %eax, -4(%rbp)  
movl -4(%rbp), %eax  
leave  
ret
```

```
int f(int x) {  
    int y;  
    y = x + 3;  
    return y;  
}
```

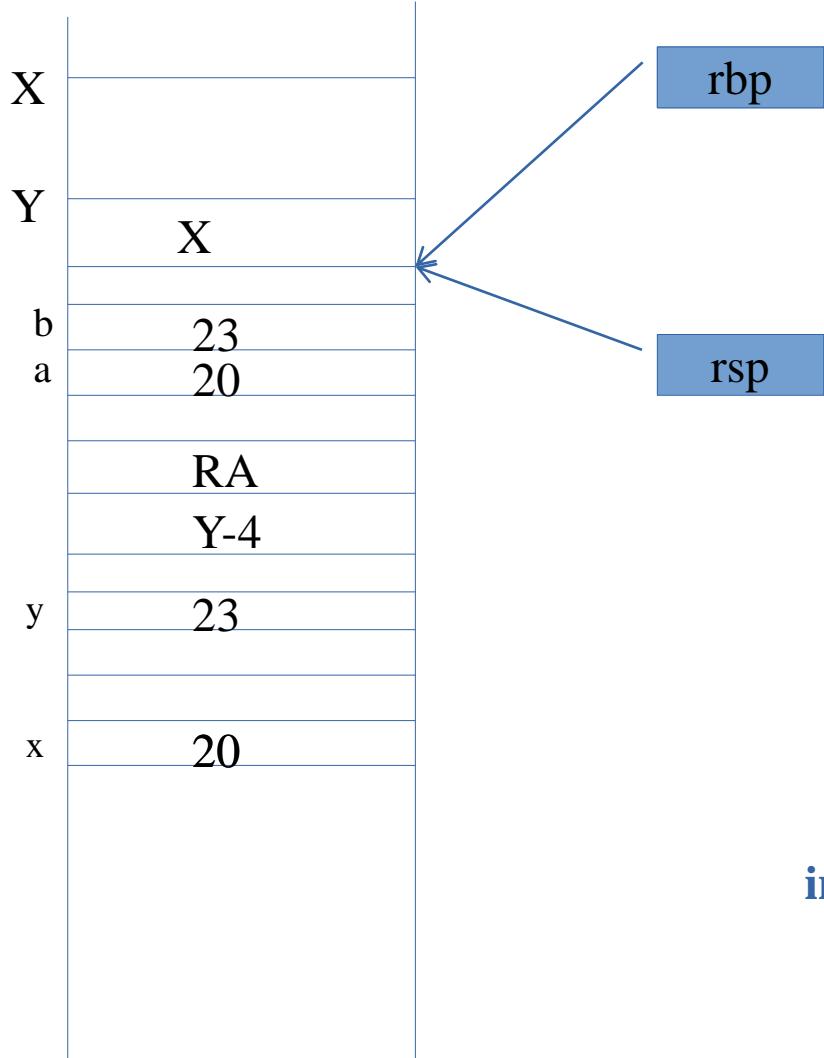
```
f:  
endbr64  
pushq %rbp  
movq %rsp, %rbp  
movl %edi, -20(%rbp)  
movl -20(%rbp), %eax  
addl $3, %eax  
movl %eax, -4(%rbp)  
movl -4(%rbp), %eax  
RApopq %rbp  
ret
```

edi = 20

eax = 23

```
int main() {  
    int a = 20, b = 30;  
    b = f(a);  
    return b;  
}
```

eip = RA



```
main:
endbr64
pushq %rbp
movq %rsp, %rbp
subq $16, %rsp
movl $20, -8(%rbp)
movl $30, -4(%rbp)
movl -8(%rbp), %eax
movl %eax, %edi
call f
movl %eax, -4(%rbp)
movl -4(%rbp), %eax
leave
# mov rbp rsp; pop rbp
ret
```

```
int f(int x) {
    int y;
    y = x + 3;
    return y;
}
```

```
int main() {
    int a = 20, b = 30;
    b = f(a);
    return b;
}
```

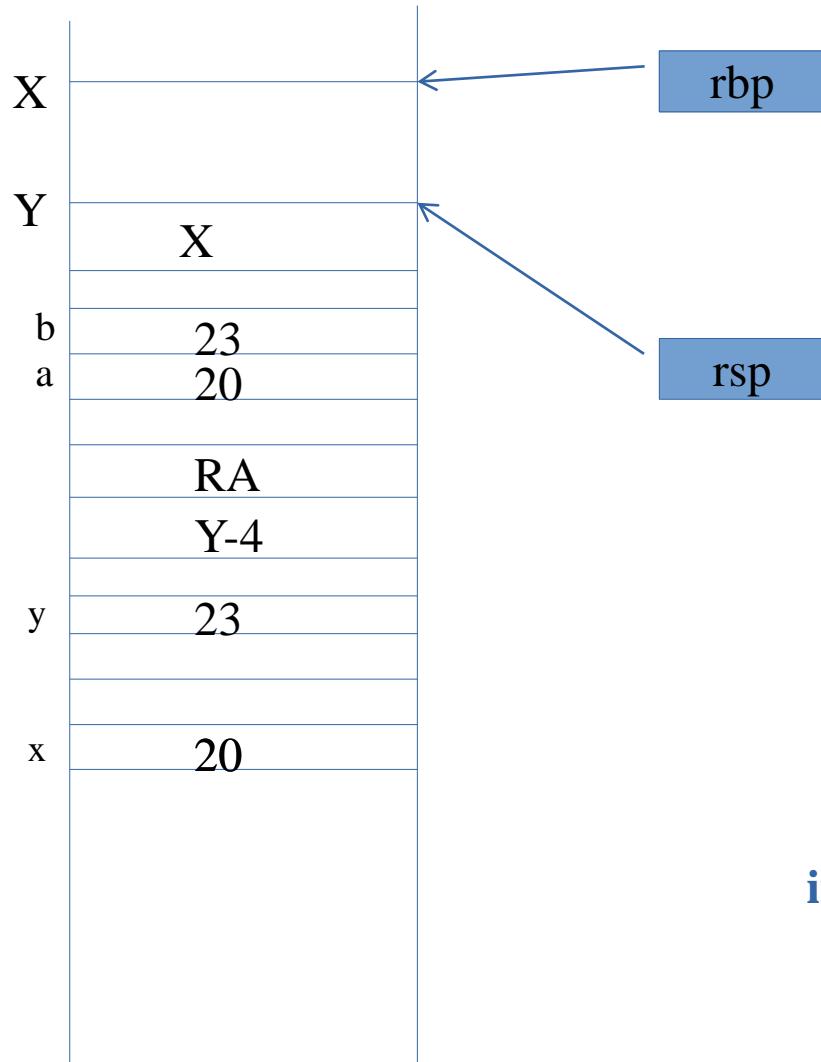
f:

```
endbr64
pushq %rbp
movq %rsp, %rbp
movl %edi, -20(%rbp)
movl -20(%rbp), %eax
addl $3, %eax
movl %eax, -4(%rbp)
movl -4(%rbp), %eax
popq %rbp
ret
```

edi = 20

eax = 23

eip = RA



```
int f(int x) {
    int y;
    y = x + 3;
    return y;
}
```

```
main:
endbr64
pushq %rbp
movq %rsp, %rbp
subq $16, %rsp
movl $20, -8(%rbp)
movl $30, -4(%rbp)
movl -8(%rbp), %eax
movl %eax, %edi
call f
movl %eax, -4(%rbp)
movl -4(%rbp), %eax
leave
# mov ebp esp; pop ebp
ret
```

```
f:
endbr64
pushq %rbp
movq %rsp, %rbp
movl %edi, -20(%rbp)
movl -20(%rbp), %eax
addl $3, %eax
movl %eax, -4(%rbp)
movl -4(%rbp), %eax
popq %rbp
ret
```

edi = 20

eax = 23

```
int main() {
    int a = 20, b = 30;
    b = f(a);
    return b;
}
```

eip = RA

Further on calling convention

This was a simple program

The parameter was passed in a register!

What if there were many parameters?

CPUs have different numbers of registers.

More parameters, more functions demand a more sophisticated convention

May be slightly different on different processors, or 32-bit, 64-bit variants also.

Caller save and Callee save registers

Local variables

Are visible only within the function

Recursion: different copies of variables

Stored on “stack”

Registers

Are only one copy

Are within the CPU

Local Variables & Registers conflict

Caller save and Callee save registers

Caller Save registers

Which registers need to be saved by caller function . They can be used by the callee function!

The caller function will push them (if already in use, otherwise no need) on the stack

Callee save registers

Will be pushed on to the stack by called (callee) function

How to return values?

On the stack itself – then caller will have to pop

X86 convention – caller, callee saved 32 bit

The caller-saved registers are EAX, ECX, EDX.

The callee-saved registers are EBX, EDI, and ESI

Activation record looks like this

F() called g()

**Parameters-i refers to
parameters passed by f() to
g()**

**Local variable is a variable
in g()**

**Return address is the
location in f() where call
should go back**

X86 caller and callee rules(32 bit)

Caller rules on call

Push caller saved registers on stack

Push parameters on the stack – in reverse order. Why?

Substract esp, copy data

call f() // push + jmp

Caller rules on return

return value is in eax

remove parameters from stack : Add to esp.

Restore caller saved registers (if any)

X86 caller and callee rules

Callee rules on call

1) **push ebp**

mov ebp, esp

ebp(+/-offset) normally used to locate local vars and parameters on stack

ebp holds a copy of esp

Ebp is pushed so that it can be later popped while returnig

<https://www.cs.virginia.edu/~evans/cs216/guides/x86.html>

2) **Allocate local variables**

X86 caller and callee rules

Callee rules on return

- 1) Leave return value in eax**
- 2) Restore callee saved registers**
- 3) Deallocate local variables**
- 4) restore the ebp**
- 5) return**

32 bit vs 64 bit calling convention

Registers are used for passing parameters in 64 bit , to a large extent

Upto 6 parameters

More parameters pushed on stack

See

<https://aaronbloomfield.github.io/pdr/book/x86-64bit-ccc-chapter.pdf>

Beware

**When you read assembly code generated using
gcc -S**

You will find

More complex instructions

But they will essentially follow the convention mentioned

Comparison

	MIPS	x86
Arguments:	First 4 in %a0-%a3, remainder on stack	Generally all on stack
Return values:	%v0-%v1	%eax
Caller-saved registers:	%t0-%t9	%eax, %ecx, & %edx
Callee-saved registers:	%s0-%s9	Usually none

Figure 6.2: A comparison of the calling conventions of MIPS and x86

From the textbook by Misru

simple3.c and simple3.s

```
int f(int x1, int x2, int x3, int x4, int x5, int x6, int x7, int x8, int x9, int x10) {  
    int h;  
    h = x1 + x2 + x3 + x4 + x5 + x6 + x7 + x8 + x9 + x10 + 3;  
    return h;  
}  
int main() {  
    int a1 = 10, a2 = 20, a3 = 30, a4 = 40, a5 = 50, a6 = 60, a7 = 70, a8 = 80, a9 = 90, a10 = 100;  
    int b;  
    b = f(a1, a2, a3, a4, a5, a6, a7, a8, a9, a10);  
    return b;  
}
```

simple3.c and simple3.s

main:

endbr64	movl -24(%rbp), %r9d	
pushq %rbp	movl -28(%rbp), %r8d	movl %eax, %edi
movq %rsp, %rbp	movl -32(%rbp), %ecx	call f
subq \$48, %rsp	movl -36(%rbp), %edx	addq \$32, %rsp
movl \$10, -44(%rbp)	movl -40(%rbp), %esi	movl %eax, -4(%rbp)
movl \$20, -40(%rbp)	movl -44(%rbp), %eax	movl -4(%rbp), %eax
movl \$30, -36(%rbp)	movl -8(%rbp), %edi	leave
movl \$40, -32(%rbp)	pushq %rdi	ret
movl \$50, -28(%rbp)	movl -12(%rbp), %edi	
movl \$60, -24(%rbp)	pushq %rdi	
movl \$70, -20(%rbp)	movl -16(%rbp), %edi	
movl \$80, -16(%rbp)	pushq %rdi	
movl \$90, -12(%rbp)	movl -20(%rbp), %edi	
movl \$100, -8(%rbp)	pushq %rdi	

simple3.c and simple3.s

f:

endbr64

```
pushq %rbp
movq %rsp, %rbp
movl %edi, -20(%rbp)
movl %esi, -24(%rbp)
movl %edx, -28(%rbp)
movl %ecx, -32(%rbp)
movl %r8d, -36(%rbp)
movl %r9d, -40(%rbp)
movl -20(%rbp), %edx
movl -24(%rbp), %eax
addl %eax, %edx
movl -28(%rbp), %eax
addl %eax, %edx
movl -32(%rbp), %eax
```

```
addl %eax, %edx
movl -36(%rbp), %eax
addl %eax, %edx
movl -40(%rbp), %eax
addl %eax, %edx
movl 16(%rbp), %eax
addl %eax, %edx
movl 24(%rbp), %eax
addl %eax, %edx
movl 32(%rbp), %eax
addl %eax, %edx
```

```
movl 40(%rbp), %eax
addl %edx, %eax
addl $3, %eax
movl %eax, -4(%rbp)
movl -4(%rbp), %eax
popq %rbp
ret
```

Let's see a demo of how the stack is built and destroyed during function calls, on a Linux machine using GCC.

Consider this C code

```
int mult(int a, int b) {  
    int c, d = 20, e = 30, f;  
    f = add(d, e);  
    c = a * b + f;  
    return c;  
}  
  
int add(int x, int y) {  
    int z;  
    z = x + y;  
    return z;  
}
```

Translated to assembly as:

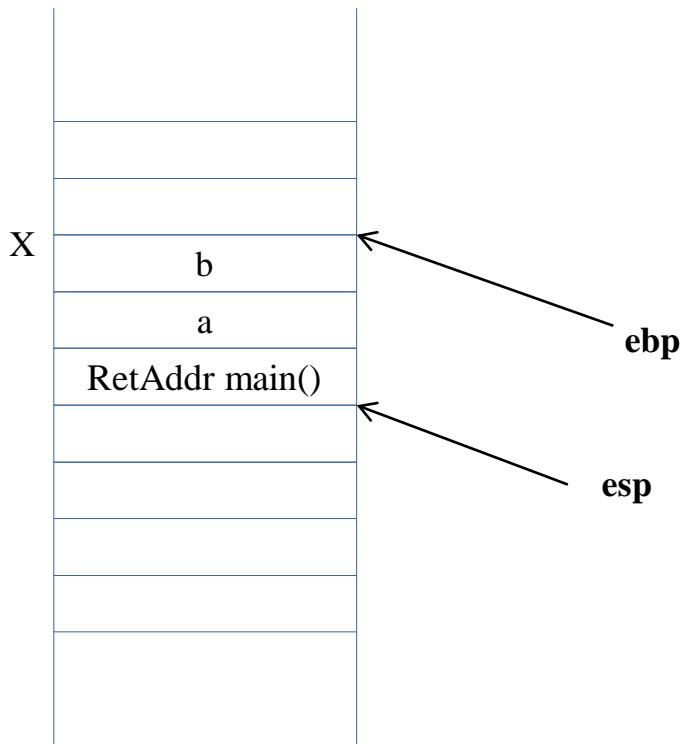
add:

```
pushl %ebp  
movl %esp, %ebp  
subl $16, %esp  
movl 8(%ebp), %edx  
movl 12(%ebp), %eax  
addl %edx, %eax  
movl %eax, -4(%ebp)  
movl -4(%ebp), %eax  
leave  
ret
```

mult:

```
pushl %ebp  
movl %esp, %ebp  
subl $24, %esp  
movl $20, -24(%ebp)  
movl $30, -20(%ebp)  
subl $8, %esp  
pushl -20(%ebp)  
pushl -24(%ebp)  
call add  
addl $16, %esp  
movl %eax, -16(%ebp)  
movl 8(%ebp), %eax  
imull 12(%ebp), %eax  
movl %eax, %edx  
movl -16(%ebp), %eax  
addl %edx, %eax  
movl %eax, -12(%ebp)  
movl -12(%ebp), %eax  
leave  
ret
```

Stack
↓



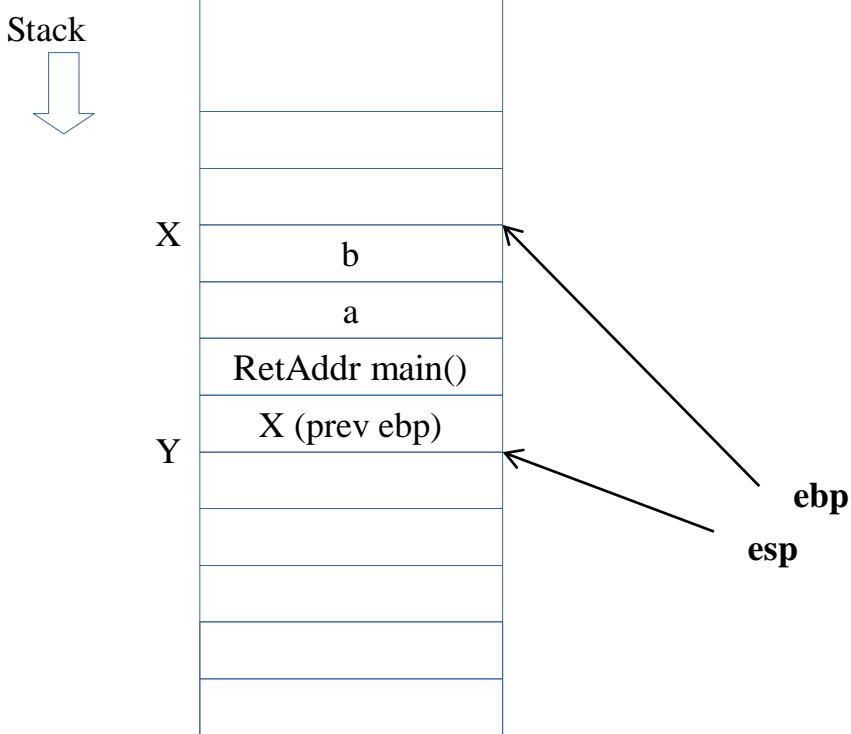
/ Control is here */*

```
int mult(int a, int b) {  
    int c, d = 20, e = 30, f;  
    f = add(d, e);  
    c = a * b + f;  
    return c;  
}  
int add(int x, int y) {  
    int z;  
    z = x + y;  
    return z;  
}
```

** Control is here */*

mult:

```
pushl %ebp  
movl %esp, %ebp  
subl $24, %esp  
movl $20, -24(%ebp)  
movl $30, -20(%ebp)  
subl $8, %esp  
pushl -20(%ebp)  
pushl -24(%ebp)  
call add
```



```

int mult(int a, int b) {
    int c, d = 20, e = 30, f;
    f = add(d, e);
    c = a * b + f;
    return c;
}
int add(int x, int y) {
    int z;
    z = x + y;
    return z;
}

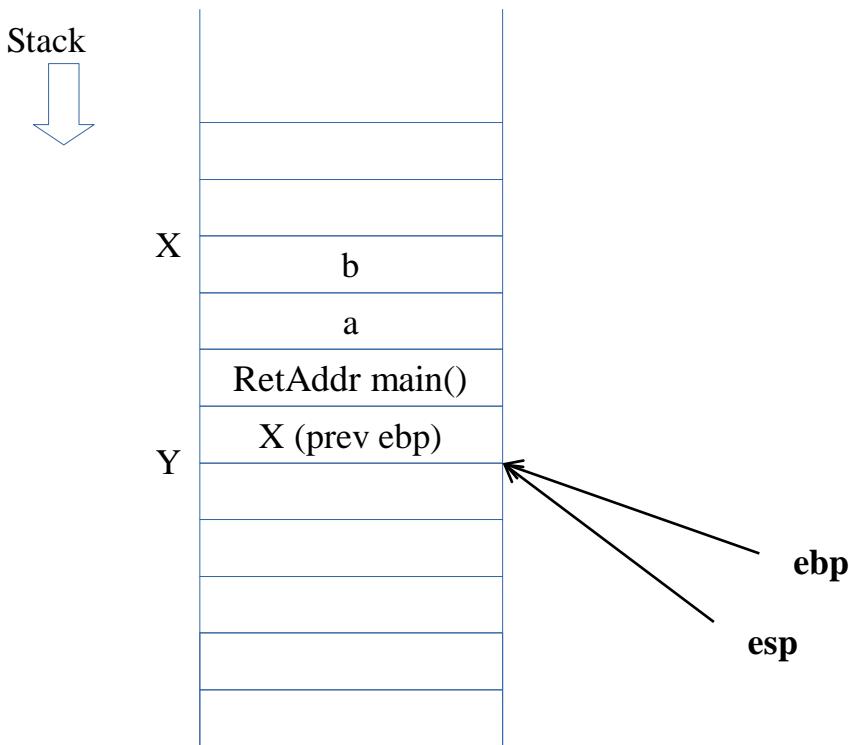
```

mult:

```

pushl %ebp
movl %esp, %ebp
subl $24, %esp
movl $20, -24(%ebp)
movl $30, -20(%ebp)
subl $8, %esp
pushl -20(%ebp)
pushl -24(%ebp)
call add

```



```

int mult(int a, int b) {
    int c, d = 20, e = 30, f;
    f = add(d, e);
    c = a * b + f;
    return c;
}
int add(int x, int y) {
    int z;
    z = x + y;
    return z;
}

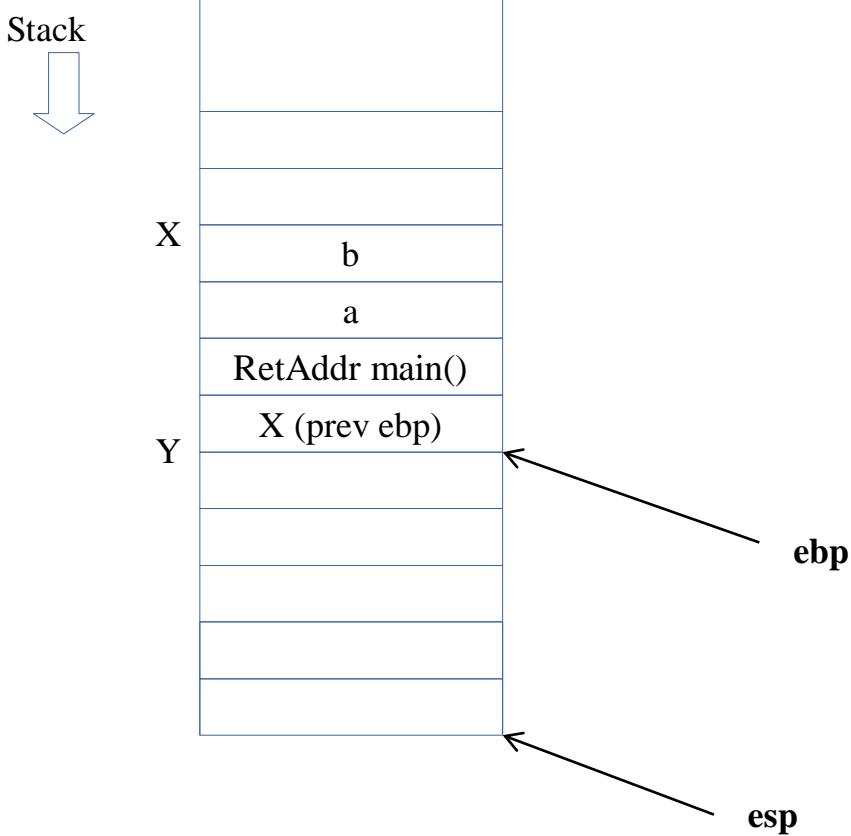
```

mult:

```

pushl %ebp
movl %esp, %ebp
subl $24, %esp
movl $20, -24(%ebp)
movl $30, -20(%ebp)
subl $8, %esp
pushl -20(%ebp)
pushl -24(%ebp)
call add

```



```

int mult(int a, int b) {
    int c, d = 20, e = 30, f;
    f = add(d, e);
    c = a * b + f;
    return c;
}
int add(int x, int y) {
    int z;
    z = x + y;
    return z;
}

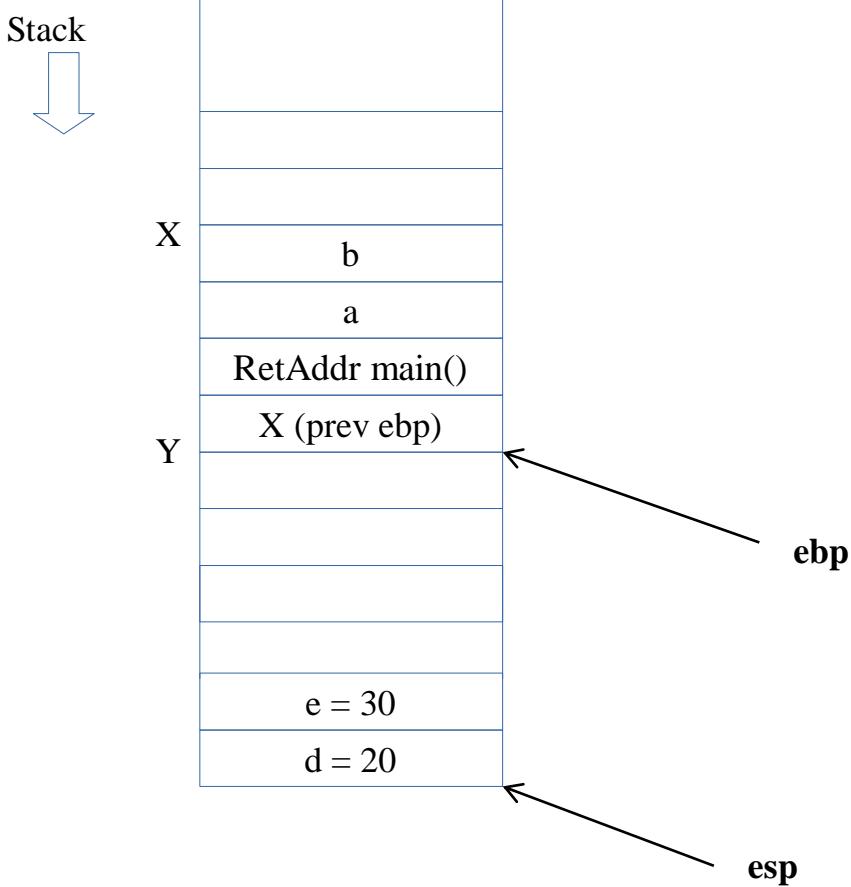
```

mult:

```

pushl %ebp
movl %esp, %ebp
subl $24, %esp
movl $20, -24(%ebp)
movl $30, -20(%ebp)
subl $8, %esp
pushl -20(%ebp)
pushl -24(%ebp)
call add

```



```

int mult(int a, int b) {
    int c, d = 20, e = 30, f;
    f = add(d, e);
    c = a * b + f;
    return c;
}
int add(int x, int y) {
    int z;
    z = x + y;
    return z;
}

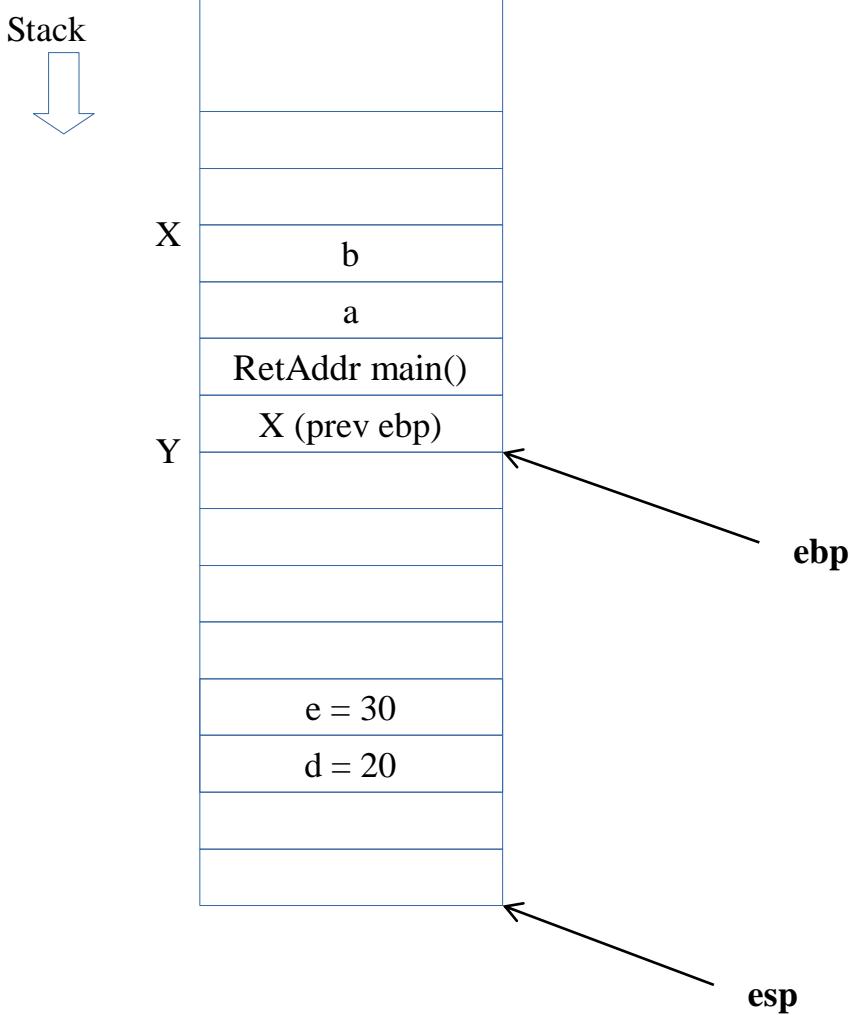
```

mult:

```

pushl %ebp
movl %esp, %ebp
subl $24, %esp
movl $20, -24(%ebp)
movl $30, -20(%ebp)
subl $8, %esp
pushl -20(%ebp)
pushl -24(%ebp)
call add

```



```

int mult(int a, int b) {
    int c, d = 20, e = 30, f;
    f = add(d, e);
    c = a * b + f;
    return c;
}
int add(int x, int y) {
    int z;
    z = x + y;
    return z;
}

```

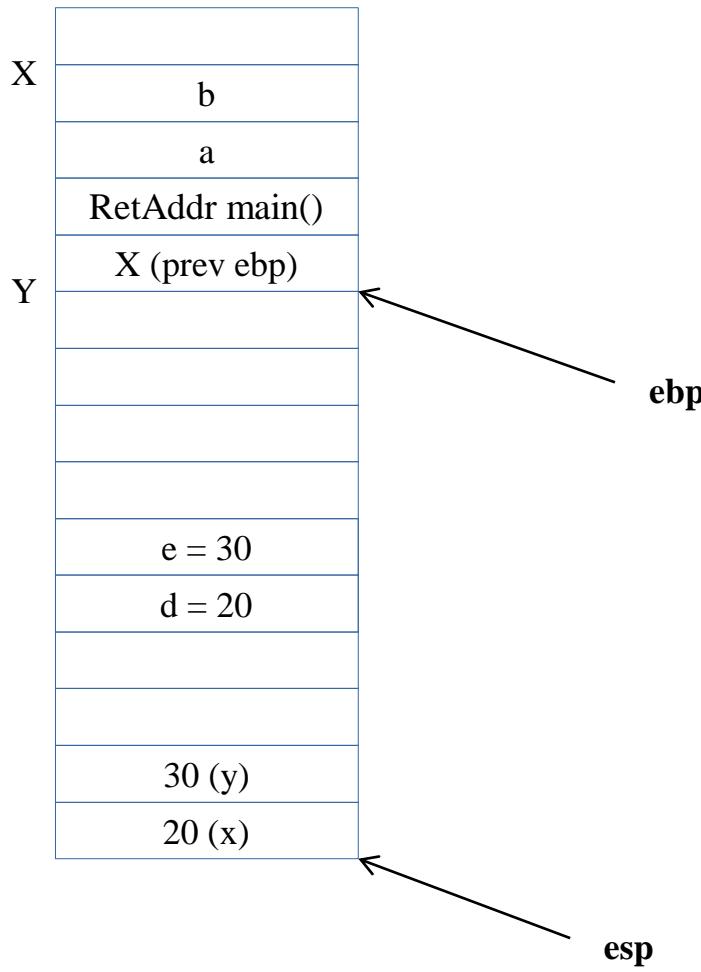
mult:

```

pushl %ebp
movl %esp, %ebp
subl $24, %esp
movl $20, -24(%ebp)
movl $30, -20(%ebp)
subl $8, %esp
pushl -20(%ebp)
pushl -24(%ebp)
call add

```

Stack

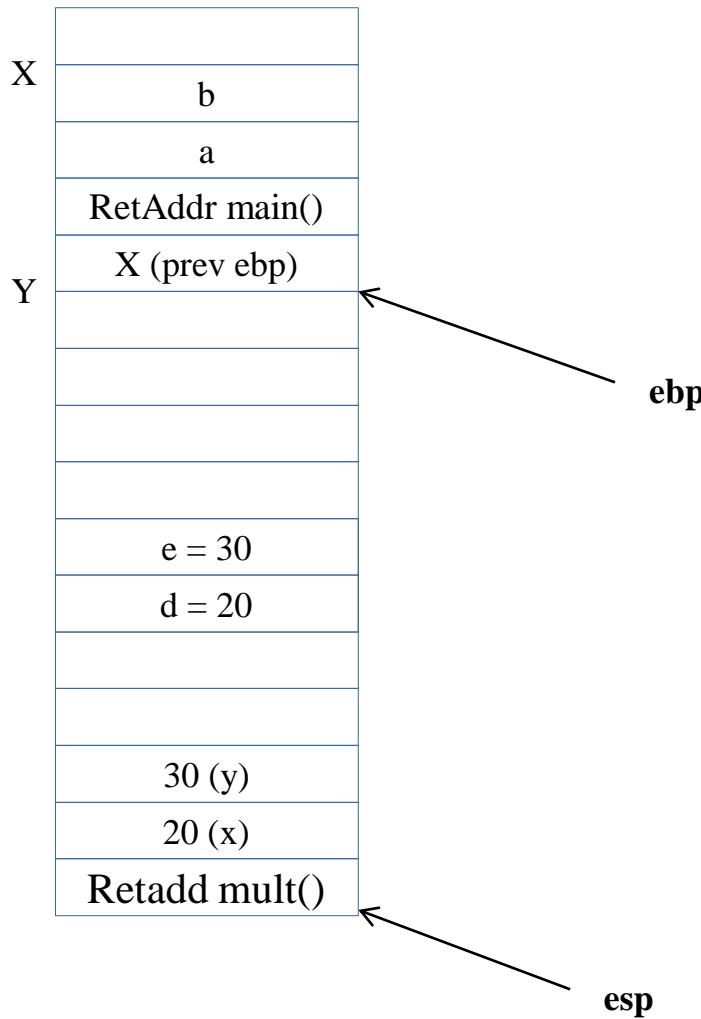


```
int mult(int a, int b) {  
    int c, d = 20, e = 30, f;  
    f = add(d, e);  
    c = a * b + f;  
    return c;  
}  
int add(int x, int y) {  
    int z;  
    z = x + y;  
    return z;  
}
```

mult:

```
pushl %ebp  
movl %esp, %ebp  
subl $24, %esp  
movl $20, -24(%ebp)  
movl $30, -20(%ebp)  
subl $8, %esp  
pushl -20(%ebp)  
pushl -24(%ebp)  
call add
```

Stack



```
int mult(int a, int b) {  
    int c, d = 20, e = 30, f;  
    f = add(d, e);  
    c = a * b + f;  
    return c;  
}  
int add(int x, int y) {  
    int z;  
    z = x + y;  
    return z;  
}
```

mult:

```
pushl %ebp  
movl %esp, %ebp  
subl $24, %esp  
movl $20, -24(%ebp)  
movl $30, -20(%ebp)  
subl $8, %esp  
pushl -20(%ebp)  
pushl -24(%ebp)  
call add
```

Stack



X	b
	a
	RetAddr main()
Y	X (prev ebp)
e = 30	
d = 20	
30 (y)	
20 (x)	
Retadd mult()	
Y(prev ebp)	

ebp

esp

```
int mult(int a, int b) {  
    int c, d = 20, e = 30, f;  
    f = add(d, e);  
    c = a * b + f;  
    return c;  
}
```

```
int add(int x, int y) {  
    int z;  
    z = x + y;  
    return z;  
}
```

add:

```
pushl %ebp  
movl %esp, %ebp  
subl $16, %esp  
movl 8(%ebp), %edx  
movl 12(%ebp), %eax  
addl %edx, %eax  
movl %eax, -4(%ebp)  
movl -4(%ebp), %eax  
leave  
ret
```

Stack



X	b
	a
	RetAddr main()
X (prev ebp)	
Y	
e = 30	
d = 20	
30 (y)	
20 (x)	
Retadd mult()	
Y(prev ebp)	

edx = 20
eax = 30
eax = eax + edx = 50

ebp
esp

```
int mult(int a, int b) {  
    int c, d = 20, e = 30, f;  
    f = add(d, e);  
    c = a * b + f;  
    return c;  
}  
int add(int x, int y) {  
    int z;  
    z = x + y;  
    return z;  
}
```

add:

```
pushl %ebp  
movl %esp, %ebp  
subl $16, %esp  
movl 8(%ebp), %edx  
movl 12(%ebp), %eax  
addl %edx, %eax  
movl %eax, -4(%ebp)  
movl -4(%ebp), %eax  
leave  
ret
```

Stack



X	b
	a
	RetAddr main()
Y	X (prev ebp)
e = 30	
d = 20	
30 (y)	
20 (x)	
Retadd mult()	
Y(prev ebp)	
50	

eax = 50

ebp
esp

```
int mult(int a, int b) {  
    int c, d = 20, e = 30, f;  
    f = add(d, e);  
    c = a * b + f;  
    return c;  
}
```

```
int add(int x, int y) {  
    int z;  
    z = x + y;  
    return z;  
}
```

add:

```
pushl %ebp  
movl %esp, %ebp  
subl $16, %esp  
movl 8(%ebp), %edx  
movl 12(%ebp), %eax  
addl %edx, %eax  
movl %eax, -4(%ebp)  
movl -4(%ebp), %eax  
leave  
ret
```

Some redundant code generated here.
Before “leave”. Result is in eax

Stack



X	b
	a
	RetAddr main()
Y	X (prev ebp)
e = 30	
d = 20	
30 (y)	
20 (x)	
Retadd mult()	
Y(prev ebp)	
50	

leave: step 1

eax = 50

esp

ebp

```
int mult(int a, int b) {  
    int c, d = 20, e = 30, f;  
    f = add(d, e);  
    c = a * b + f;  
    return c;  
}
```

```
int add(int x, int y) {  
    int z;  
    z = x + y;  
    return z;  
}
```

add:

```
pushl %ebp  
movl %esp, %ebp  
subl $16, %esp  
movl 8(%ebp), %edx  
movl 12(%ebp), %eax  
addl %edx, %eax  
movl %eax, -4(%ebp)  
movl -4(%ebp), %eax
```

**leave # # Set ESP to EBP, then pop
ret**

Stack



X	b
	a
	RetAddr main()
Y	X (prev ebp)
e = 30	
d = 20	
30 (y)	
20 (x)	
Retadd mult()	
Y(prev ebp)	
50	

leave: step 2

eax = 50

esp

ebp

```
int mult(int a, int b) {  
    int c, d = 20, e = 30, f;  
    f = add(d, e);  
    c = a * b + f;  
    return c;  
}
```

```
int add(int x, int y) {  
    int z;  
    z = x + y;  
    return z;  
}
```

add:

```
pushl %ebp  
movl %esp, %ebp  
subl $16, %esp  
movl 8(%ebp), %edx  
movl 12(%ebp), %eax  
addl %edx, %eax  
movl %eax, -4(%ebp)  
movl -4(%ebp), %eax
```

**leave # # Set ESP to EBP, then pop
ret**

Stack



X	b
	a
	RetAddr main()
Y	X (prev ebp)
	e = 30
	d = 20
	30 (y)
	20 (x)
	Retaddr mult()
	Y(prev ebp)
	50

eax = 50

esp

ebp

```
int mult(int a, int b) {  
    int c, d = 20, e = 30, f;  
    f = add(d, e); // here  
    c = a * b + f;  
    return c;  
}
```

```
int add(int x, int y) {  
    int z;  
    z = x + y;  
    return z;  
}
```

add:

```
pushl %ebp  
movl %esp, %ebp  
subl $16, %esp  
movl 8(%ebp), %edx  
movl 12(%ebp), %eax  
addl %edx, %eax  
movl %eax, -4(%ebp)  
movl -4(%ebp), %eax
```

leave # # Set ESP to EBP, then pop
ret

Stack



X	b
	a
	RetAddr main()
Y	X (prev ebp)
e = 30	
d = 20	
30 (y)	
20 (x)	
Retadd mult()	
Y(prev ebp)	
50	

eax = 50

esp

ebp

```
int mult(int a, int b) {  
    int c, d = 20, e = 30, f;  
    f = add(d, e); // here  
    c = a * b + f;  
    return c;  
}  
int add(int x, int y) {  
    int z;  
    z = x + y;  
    return z;  
}
```

Mult:

....

```
call add  
addl $16, %esp  
movl %eax, -16(%ebp)  
movl 8(%ebp), %eax  
imull 12(%ebp), %eax  
movl %eax, %edx  
movl -16(%ebp), %eax  
addl %edx, %eax  
movl %eax, -12(%ebp)  
movl -12(%ebp), %eax  
leave  
ret
```

Stack



X	b
	a
	RetAddr main()
Y	X (prev ebp)
	f = 50 (eax)
	e = 30
	d = 20
	30 (y)
	20 (x)
	Retadd mult()
	Y(prev ebp)
	50

eax = 50

esp

ebp

```
int mult(int a, int b) {  
    int c, d = 20, e = 30, f;  
    f = add(d, e);  
    c = a * b + f;  
    return c;  
}
```

```
int add(int x, int y) {  
    int z;  
    z = x + y;  
    return z;  
}
```

Mult:

```
....  
    call    add  
    addl   $16, %esp  
movl  %eax, -16(%ebp)  
    movl   8(%ebp), %eax  
    imull  12(%ebp), %eax  
    movl   %eax, %edx  
    movl   -16(%ebp), %eax  
    addl   %edx, %eax  
    movl   %eax, -12(%ebp)  
    movl   -12(%ebp), %eax  
    leave  
    ret
```

Stack



X	b
	a
	RetAddr main()
Y	X (prev ebp)
	f = 50
	e = 30
	d = 20
	30 (y)
	20 (x)
	Retadd mult()
	Y(prev ebp)
	50

eax = a
eax = eax * b
edx = eax
eax = f
eax = edx + eax
*// eax = a*b + f*

ebp

esp

```
int mult(int a, int b) {  
    int c, d = 20, e = 30, f;  
    f = add(d, e);  
    c = a * b + f;  
    return c;  
}  
int add(int x, int y) {  
    int z;  
    z = x + y;  
    return z;  
}
```

Mult:

....

call add
addl \$16, %esp
movl %eax, -16(%ebp)
movl 8(%ebp), %eax
imull 12(%ebp), %eax
movl %eax, %edx
movl -16(%ebp), %eax
addl %edx, %eax
movl %eax, -12(%ebp)
movl -12(%ebp), %eax
leave
ret

Stack



X	b
	a
	RetAddr main()
Y	X (prev ebp)
	c = eax
	f = 50
	e = 30
	d = 20
	30 (y)
	20 (x)
	Retadd mult()
	Y(prev ebp)
	50

// eax = a*b + f

ebp

esp

```
int mult(int a, int b) {  
    int c, d = 20, e = 30, f;  
    f = add(d, e);  
    c = a * b + f;  
    return c;  
}
```

```
int add(int x, int y) {  
    int z;  
    z = x + y;  
    return z;  
}
```

Mult:

....

```
call add  
addl $16, %esp  
movl %eax, -16(%ebp)  
movl 8(%ebp), %eax  
imull 12(%ebp), %eax  
movl %eax, %edx  
movl -16(%ebp), %eax  
addl %edx, %eax  
movl %eax, -12(%ebp)  
movl -12(%ebp), %eax  
leave  
ret
```

Again some redundant code

Stack



X	b
	a
	RetAddr main()
Y	X (prev ebp)
	c = eax
	f = 50
	e = 30
	d = 20
	30 (y)
	20 (x)
	Retadd mult()
	Y(prev ebp)
	50

After leave
// eax = a*b + f

ebp
esp

```
int mult(int a, int b) {  
    int c, d = 20, e = 30, f;  
    f = add(d, e);  
    c = a * b + f;  
    return c;  
}
```

```
int add(int x, int y) {  
    int z;  
    z = x + y;  
    return z;  
}
```

Mult:

....

```
call add  
addl $16, %esp  
movl %eax, -16(%ebp)  
movl 8(%ebp), %eax  
imull 12(%ebp), %eax  
movl %eax, %edx  
movl -16(%ebp), %eax  
addl %edx, %eax  
movl %eax, -12(%ebp)  
movl -12(%ebp), %eax  
leave  
ret
```

Lessons

- Calling function (caller)
 - Pushes arguments on stack , copies values
 - On call
 - Return IP is pushed
- Initially in called function (callee)
 - Old ebp is pushed
 - ebp = stack
 - Stack is decremented to make space for local variables

Lessons

- Before Return
- Ensure that result is in ‘eax’
- On Return
 - stack = ebp
 - Pop ebp (ebp = old ebp)
 - On ‘ret’
 - Pop ‘return IP’ and go back in old function

Lessons

- This was a demonstration for a
- User program, compiled with GCC, On Linux
- Followed the conventions we discussed earlier
- Applicable to
- C programs which work using LIFO function calls
- Compiler can't generate code using this mechanism for
- Functions like fork(), exec(), scheduler(), etc.
- Boot code of OS

Notes on reading xv6 code

Abhijit A. M.

abhijit.comp@coep.ac.in

Credits:

xv6 book by Cox, Kaashoek, Morris

Notes by Prof. Sorav Bansal

Introduction to xv6

Structure of xv6 code

Compiling and executing xv6 code

About xv6

- Unix Like OS
- Multi tasking, Single user
- On x86 processor
- Supports some system calls
- Small code, 7 to 10k
- Meant for learning OS concepts
- No : demand paging, no copy-on-write fork, no shared-memory, fixed size stack for user programs

Use cscope and ctags with VIM

- Go to folder of xv6 code and run

```
cscope -q *. [chS]
```

- Also run

```
ctags *. [chS]
```

- Now download the file

http://cscope.sourceforge.net/cscope_maps.vim as
.cscope_maps.vim in your ~ folder

- And add line "source

```
~/.cscope_maps.vim
```

 in your ~/.vimrc file

- Read this tutorial

Use call graphs (using doxygen)

- Doxygen – a documentation generator.
- Can also be used to generate “call graphs” of functions
- Download xv6
- Install doxygen on your Ubuntu machine.
- cd to xv6 folder
- Run “doxygen -g doxyconfig”
- This creates the file “doxyconfig”

Use call graphs (using doxygen)

- Create a folder “doxygen”
- Open “doxyconfig” file and make these changes.

PROJECT_NAME

= "XV6"

OUTPUT_DIRECTORY

= ./doxygen

CREATE_SUBDIRS

= YES

EXTRACT_ALL

= YES

EXCLUDE

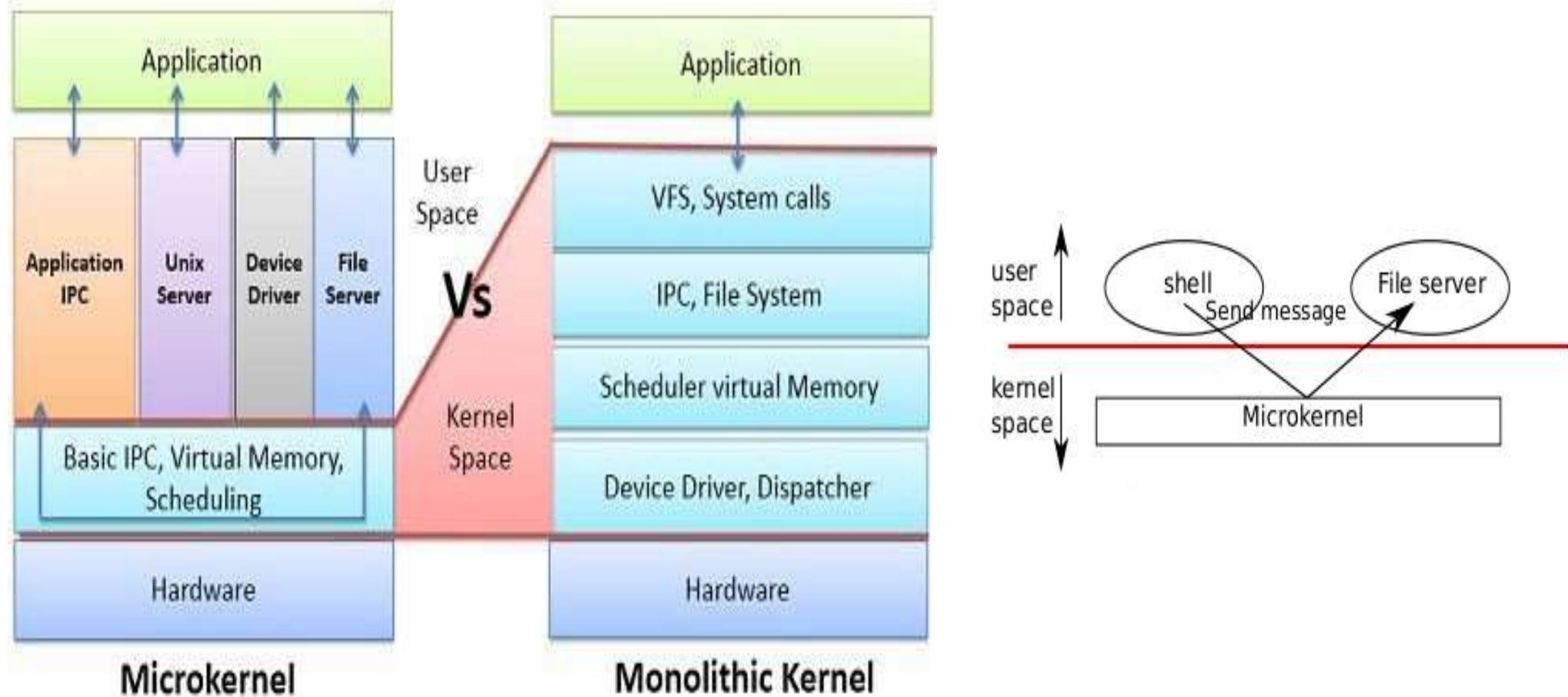
= usertests.c

cat.c yes.c echo.c forktest.c grep.c

init.c kill.c ln.c ls.c mkdir.c rm.c

sh.c stressfs.c wc.c zombie.c

Xv6 follows monolithic kernel approach



qemu

- A virtual machine manager, like Virtualbox
- Qemu provides us
 - BIOS
 - Virtual CPU, RAM, Disk controller, Keyboard controller
 - IOAPIC, LAPIC
- Qemu runs xv6 using this command

```
qemu -serial mon:stdio -drive  
file=fs.img,index=1,media=disk,format  
=raw -drive  
file=xv6.img,index=0,media=disk,forma
```

qemu

□ Understanding qemu command

□ -serial mon:stdio

□ the window of xv6 is also multiplexed in your normal terminal.

□ Run “make qemu”, then Press “Ctrl-a” and “c” in terminal and you get qemu prompt

□ -drive file=fs.img,index=1,media=disk,format=raw

□ Specify the hard disk in “fs.img”, accessible at first slot in IDE(or SATA, etc), as a “disk” , with “raw” format

□ -smp 2

About files in XV6 code

- ❑ **cat.c echo.c forktest.c grep.c
init.c kill.c ln.c ls.c mkdir.c rm.c
sh.c stressfs.c usertests.c wc.c
yes.c zombie.c**

- ❑ User programs for testing xv6

- ❑ **Makefile**

- ❑ To compile the code

- ❑ **dot-bochsrc**

- ❑ For running with emulator bochs

About files in XV6 code

- ❑ `bootasm.S` `entryother.S` `entry.S`
`initcode.S` `swtch.S` `trapasm.S`
`usys.S`

- ❑ Kernel code written in Assembly. Total 373 lines

- ❑ `kernel.ld`

- ❑ Instructions to Linker, for linking the kernel properly

- ❑ `README` `Notes` `LICENSE`

- ❑ Misc files

Using Makefile

- **make qemu**
 - Compile code and run using “qemu” emulator
- **make xv6.pdf**
 - Generate a PDF of xv6 code
- **make mkfs**
 - Create the mkfs program
- **make clean**
 - Remove all intermediary and final build files

Files generated by Makefile

- **.o files**

- Compiled from each .c file

- No need of separate instruction in Makefile to create .o files

- **_%: %.o \$(ULIB)** line is sufficient to build each .o for a _xyz file

Files generated by Makefile

❑asm files

❑Each of them has an equivalent object code file or C file. For example

```
❑bootblock: bootasm.S bootmain.c  
          $(CC) $(CFLAGS) -fno-pic -O  
-nostdinc -I. -c 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
```

Files generated by Makefile

- **_ln, _ls, etc**
- **Executable user programs**
- **Compilation process is explained after few slides**

Files generated by Makefile

- ❑ **xv6.img**

- ❑ Image of xv6 created

- ❑ **xv6.img: bootblock kernel**

- ❑ **dd if=/dev/zero of=xv6.img count=10000**

- ❑ **dd if=bootblock of=xv6.img conv=notrunc**

- ❑ **dd if=kernel of=xv6.img seek=1 conv=notrunc**

Files generated by Makefile

bootblock

```
bootblock: bootasm.S bootmain.c  
          $(CC) $(CFLAGS) -fno-pic -O -  
nostdinc -I. -c 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  
          $(OBJDUMP) -S bootblock.o >  
bootblock.asm
```

Files generated by Makefile

kernel

```
kernel: $(OBJS) entry.o entryother initcode kernel.ld  
        $(LD) $(LDFLAGS) -T kernel.ld  
-o kernel entry.o $(OBJS) -b binary  
initcode entryother  
        $(OBJDUMP) -S kernel >  
kernel.asm  
        $(OBJDUMP) -t kernel | sed  
'1,/SYMBOL TABLE/d; s/ .* //';  
/^$$/d' > kernel.sym
```

Files generated by Makefile

- **fs.img**

- A disk image containing user programs and README

- **fs.img: mkfs README \$ (UPROGS)**

- **./mkfs fs.img README**
 - \$ (UPROGS)**

- **.sym files**

- Symbol tables of different programs

- E.g. for file “kernel”

- **\$ (OBJDUMP) -t kernel | sed**

Size of xv6 C code

- `wc *[ch] | sort -n`
- **10595 34249 278455 total**
- Out of which
- **738 4271 33514 dot-bochssrc**
- **wc cat.c echo.c forktest.c grep.c init.c kill.c ln.c ls.c mkdir.c rm.c sh.c stressfs.c usertests.c wc.c yes.c zombie.c**
- **2849 6864 51993 total**
- So total code is $10595 - 2849 - 738 = 7008$ lines

List of commands to try (in given order)

usertests # Runs lot of tests and takes upto 10 minutes to run

stressfs # opens , reads and writes to files in parallel

ls # out put is filetype, inode number, type

cat README

ls;ls

cat README | grep BUILD

echo hi there

echo hi there | grep hi

List of commands to try (in this order)

echo README | grep Wa

echo README | grep Wa | grep ty # does not work

cat README | grep Wa | grep bl # works

ls > out # takes time!

mkdir test

cd test

ls .. # works from inside test

cd # fails

cd / # works

wc README

rm out

ls . test # listing both directories

ln cat xyz; ls

User Libraries: Used to link user land programs

- Ulib.c
- Strcpy, strcmp, strlen, memset, strchr, stat, atoi, memmove
- Stat uses open()
- Usys.S -> compiles into usys.o
- Assembly code file. Basically converts all calls like open() (e.g. used in ulib.c) into assembly code using “int” instruction.
- Run following command see the last 4 lines in the output

```
objdump -d usys.o
```

- 00000048 <open>:

□	48:	b8 0f 00 00 00	mov	\$0xf,%eax
□	4d:	cd 40	int	\$0x40
□	4f:	c3	ret	

User Libraries: Used to link user land programs

- printf.c
- Code for printf()!
- Interesting to read this code.
- Uses variable number of arguments. Normal technique in C is to use va_args library, but here it uses pointer arithmetic.
- Written using two more functions: printint() and putc() - both call write()
- Where is code for write()?

User Libraries: Used to link user land programs

❑ **umalloc.c**

- ❑ This is an implementation of malloc() and free()
- ❑ Almost same as the one done in “The C Programming Language” by Kernighan and Ritchie
- ❑ Uses sbrk() to get more memory from xv6 kernel

Understanding the build process in more details

□ Run

make qemu | tee make-output.txt

□ You will get all compilation commands in **make-output.txt**

Compiling user land programs

Normally when you compile a program on Linux

You compile it for the same ‘target’ machine (= CPU + OS)

The compiler itself runs on the same OS

To compile a user land program for xv6, we don’t have a compiler on xv6,

So we compile the programs (using make, cc) on Linux , for xv6

Obviously they can’t link with the standard libraries on Linux

Compiling user land programs

```
ULIB = ulib.o usys.o printf.o umalloc.o

_%: %.o $(ULIB)

    $(LD) $(LDFLAGS) -N -e main -Ttext 0 -o $@ $^
    $(OBJDUMP) -S $@ > $*.asm
    $(OBJDUMP) -t $@ | sed '1,/SYMBOL TABLE/d; s/ .* /'
    '/^$$/d' > $*.sym
```

\$@ is the name of the file being generated

\$^ is dependencies . i.e. \$(ULIB) and %.o in this case

Compiling user land programs

```
gcc -fno-pic -static -fno-builtin -fno-strict-
aliasing -O2 -Wall -MD -ggdb -m32 -Werror -fno-omit-
frame-pointer -fno-stack-protector -fno-pie -no-pie
-c -o cat.o cat.c

ld -m      elf_i386 -N -e main -Ttext 0 -o _cat cat.o
ulib.o usys.o printf.o umalloc.o

objdump -S _cat > cat.asm

objdump -t _cat | sed '1,/SYMBOL TABLE/d; s/ .* / /;
/^$/d' > cat.sym
```

Compiling user land programs

Mkfs is compiled like a Linux program !

```
gcc -Werror -Wall -o mkfs mkfs.c
```

How to read kernel code ?

- Understand the data structures
- Know each global variable, typedefs, lists, arrays, etc.
- Know the purpose of each of them
- While reading a code path, e.g. exec()
- Try to ‘locate’ the key line of code that does major work
- Initially (but not forever) ignore the ‘error checking’ code
- Keep summarising what you have read

Pre-requisites for reading the code

- Understanding of core concepts of operating systems
- Memory Management, processes, fork-exec, file systems, synchronization, x86 architecture, calling convention , computer organization
- 2 approaches:
 - 1) Read OS basics first, and then start reading xv6 code
 - Good approach, but takes more time !
 - 2) Read some basics, read xv6, repeat

Gives a headstart, but you will always have gaps in

Memory Management Basics

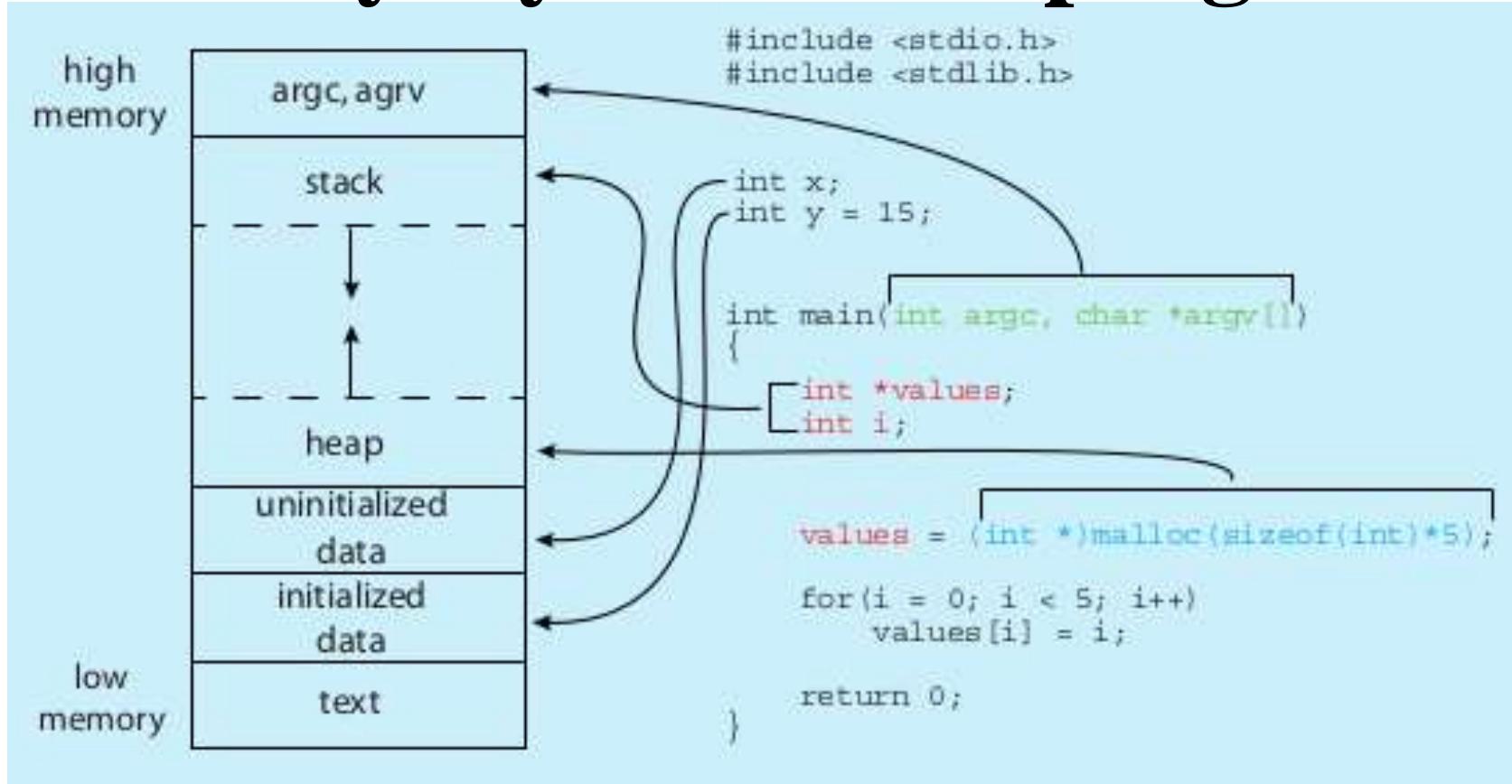
Summary

- 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 (multiprogramming)
 - 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

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()

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.

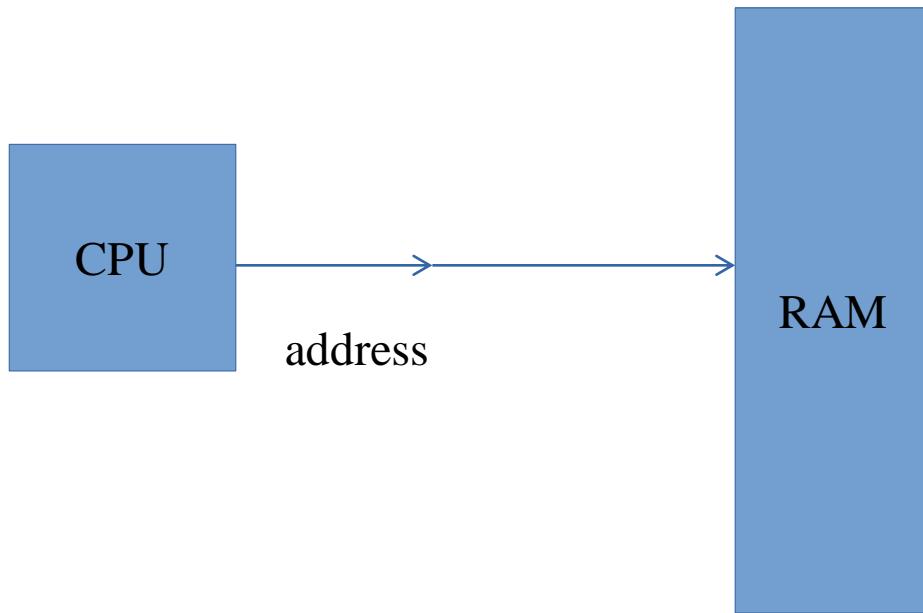
- Load time address binding

- Address of code/variables is fixed by loader

Location of process in RAM is decided at load

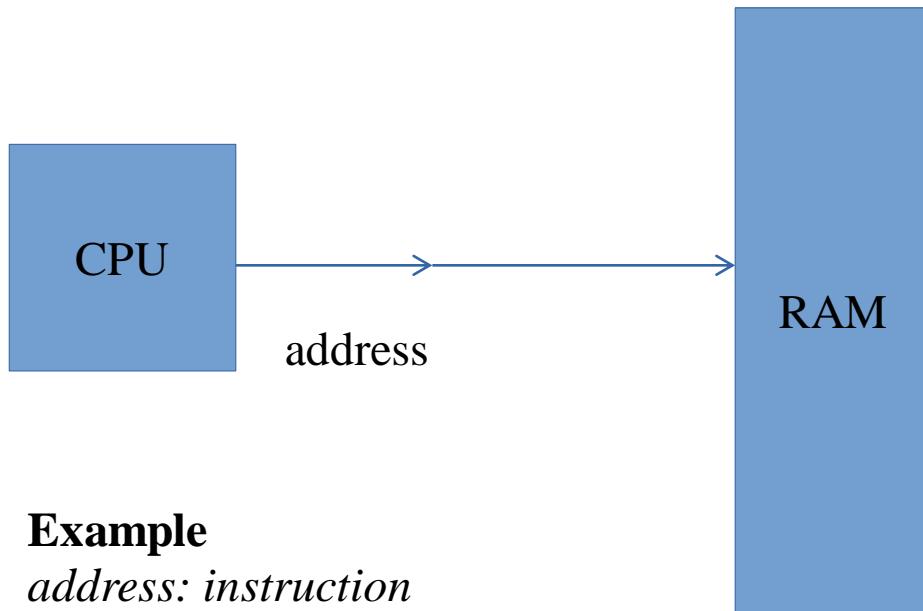
Which binding is actually used, i

Simplest case



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

Simplest case



Example

address: instruction

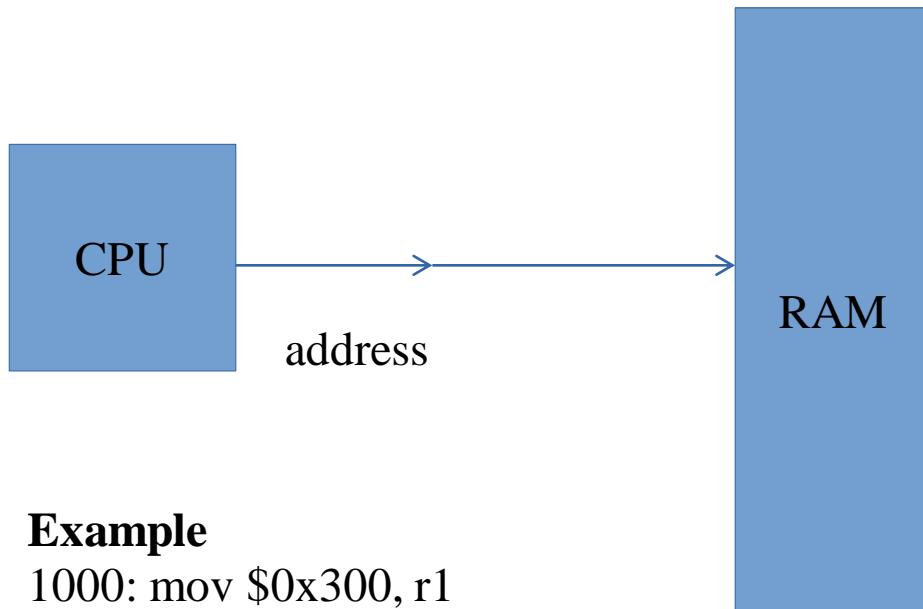
```
1000: mov $0x300, r1  
1004: add r1, -3  
1008: jnz 1000
```

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

Simplest case

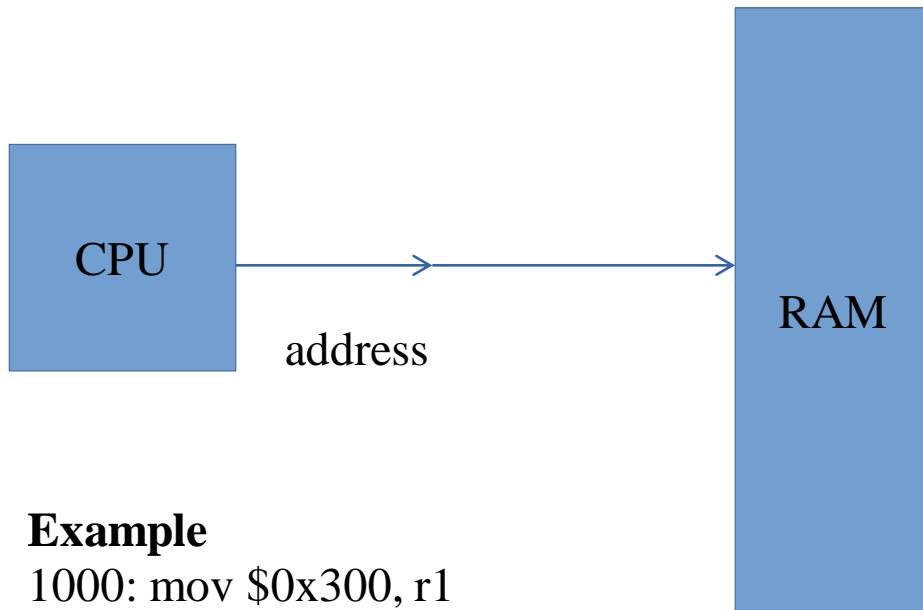


Example

```
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**

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

Base/Relocation + Limit scheme

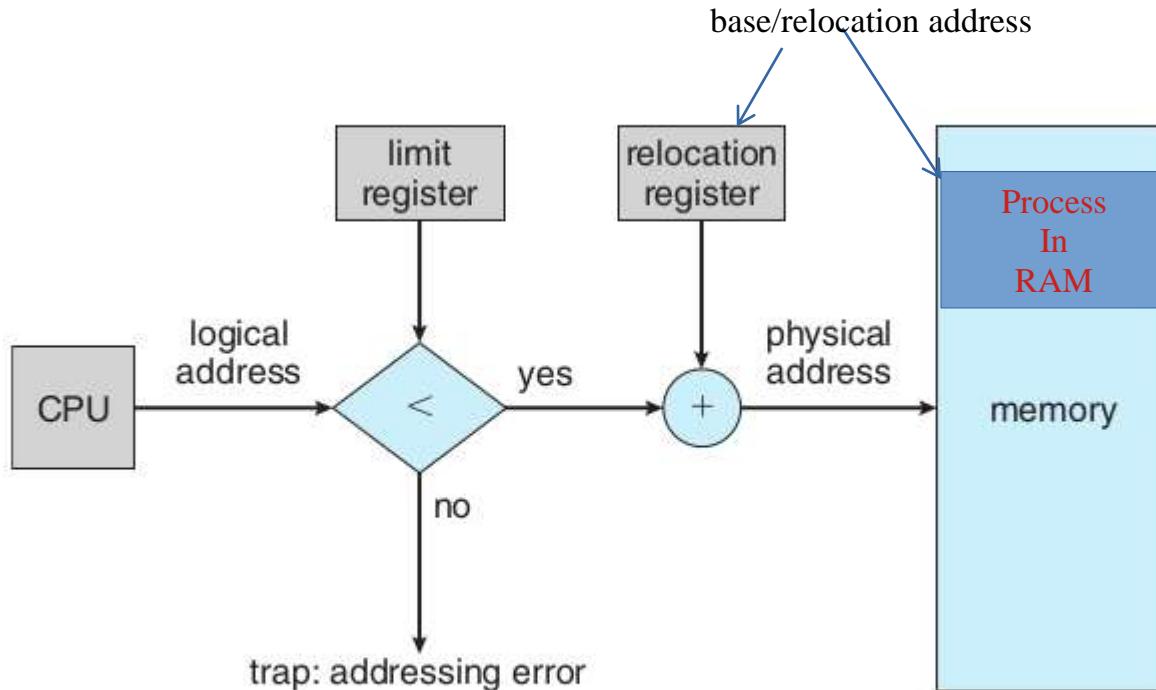


Figure 9.6 Hardware support for relocation and limit registers.

- Base and Limit are two registers inside CPU's Memory Management Unit
- 'base' is added to the address generated by CPU
- The result is

Memory Management Unit (MMU)

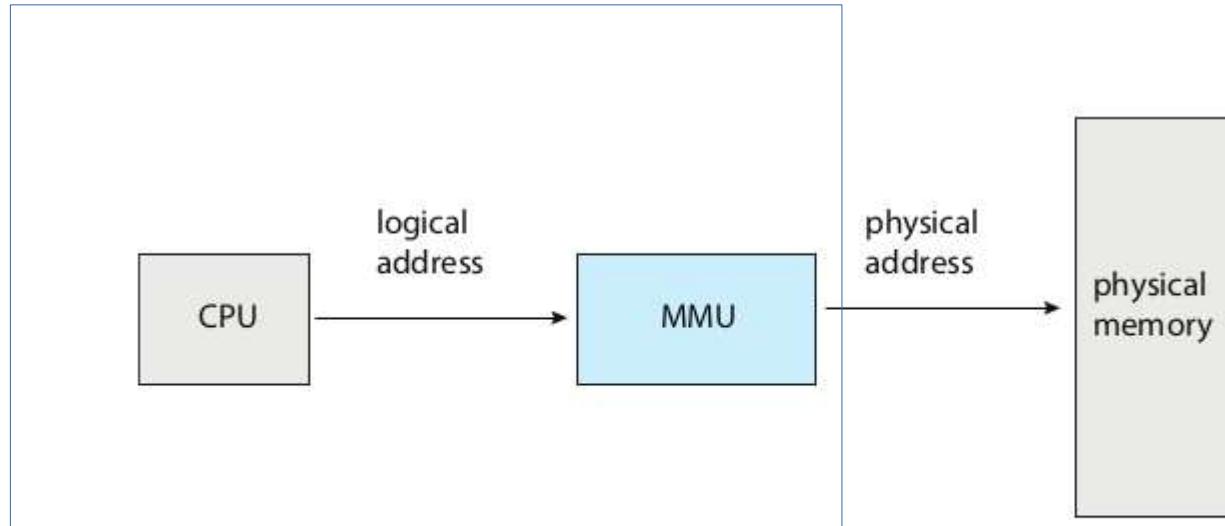
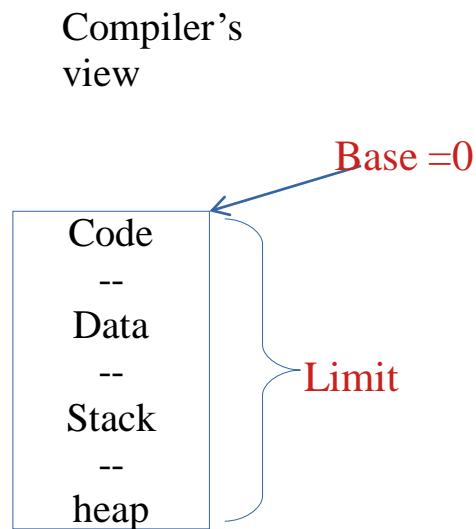
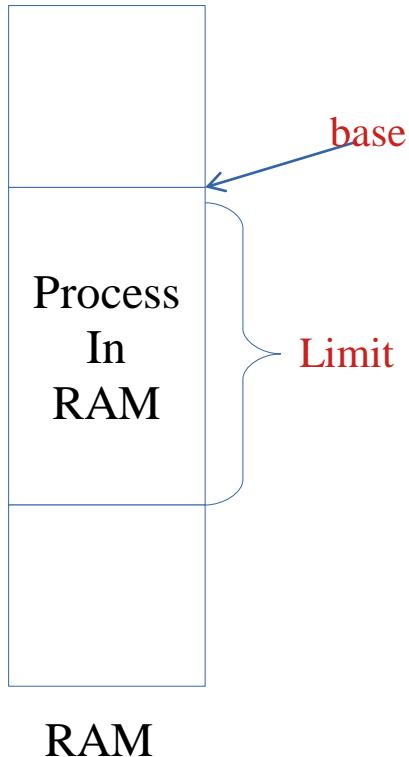


Figure 9.4 Memory management unit (MMU).

- Is part of the CPU chip, acts on every memory address issue by execution unit of the CPU

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 (!)

Base/Relocation + Limit scheme

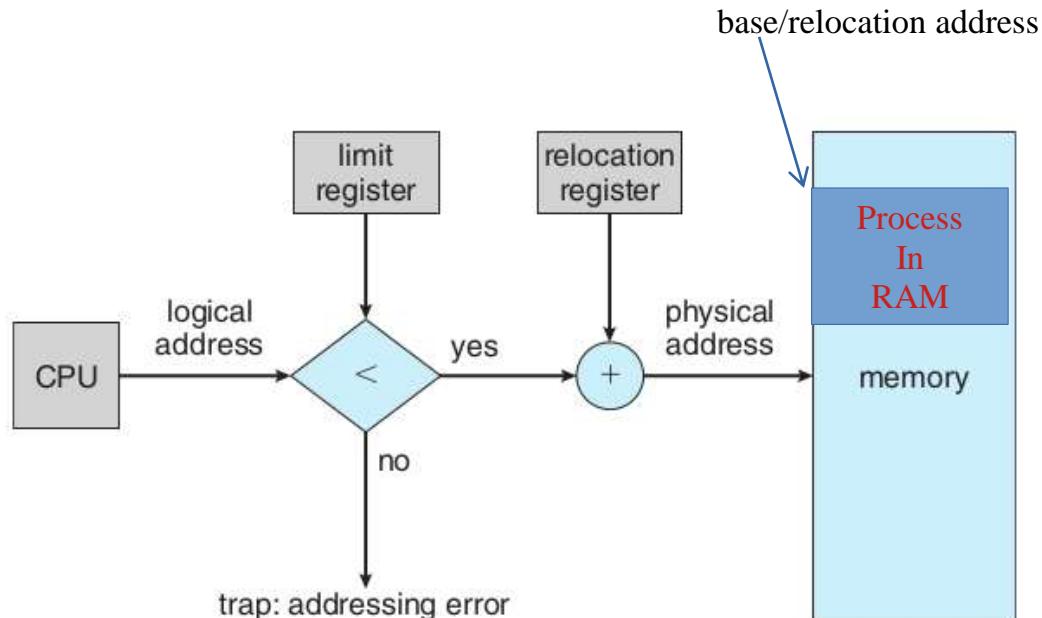


Figure 9.6 Hardware support for relocation and limit registers.

.OS's work

- While loading the process in memory – must load as one continuous segment
- Fill in the ‘base’ register with the

Base/Relocation + Limit scheme

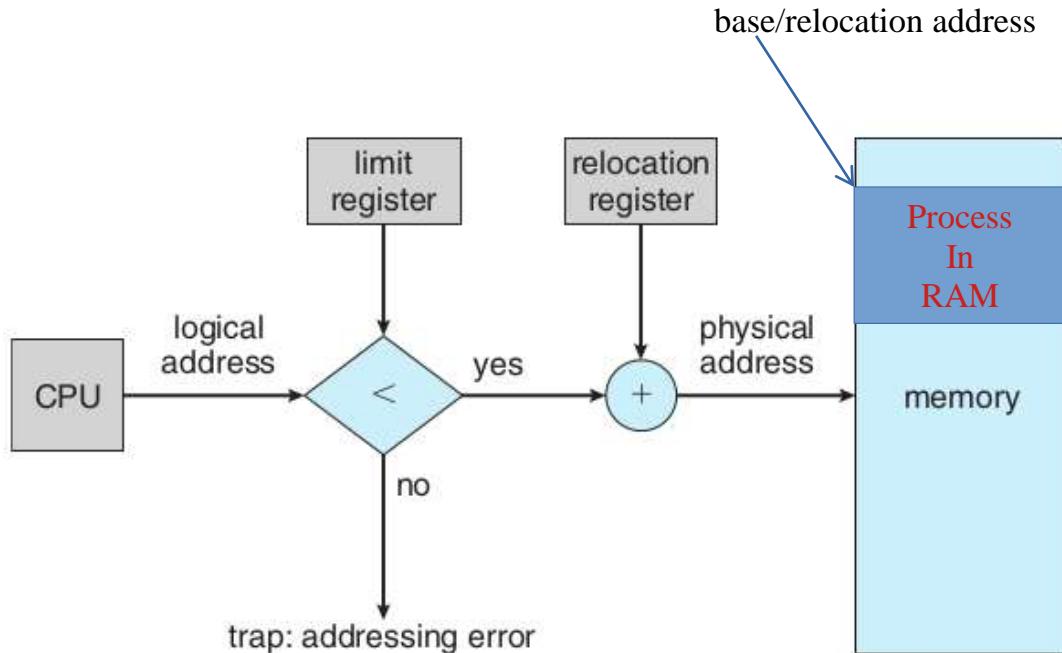
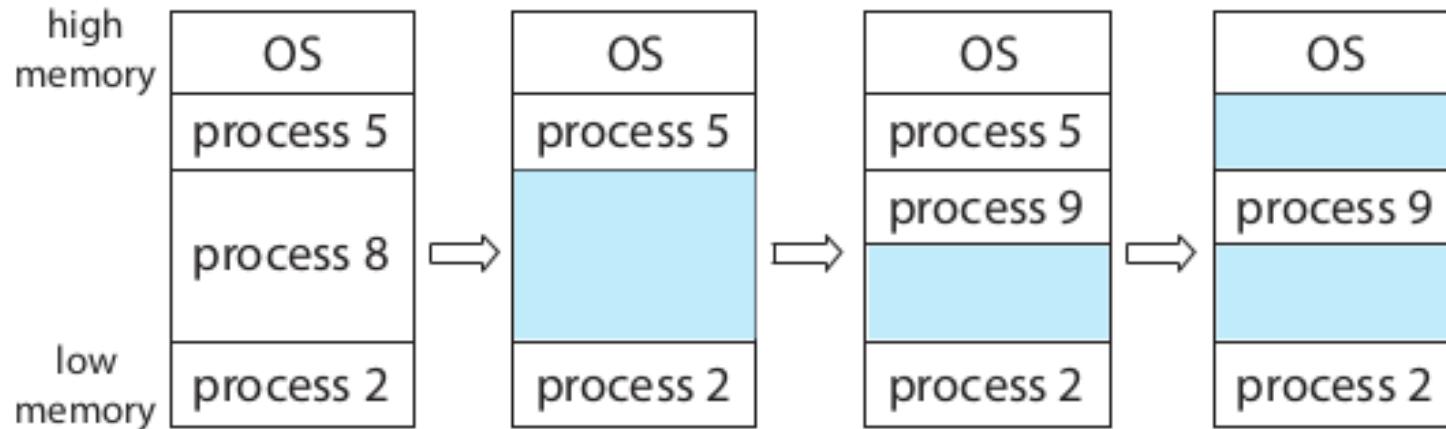


Figure 9.6 Hardware support for relocation and limit registers.

- .Combined effect
 - “**Relocatable code**” – the process can go anywhere in RAM at the time of loading
 - Some memory violations can be

Example scenario of memory in base+limit scheme



Process 8 called `fork()` Process 9 forked Process 5 terminated

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

By doing extra work during “loading”.

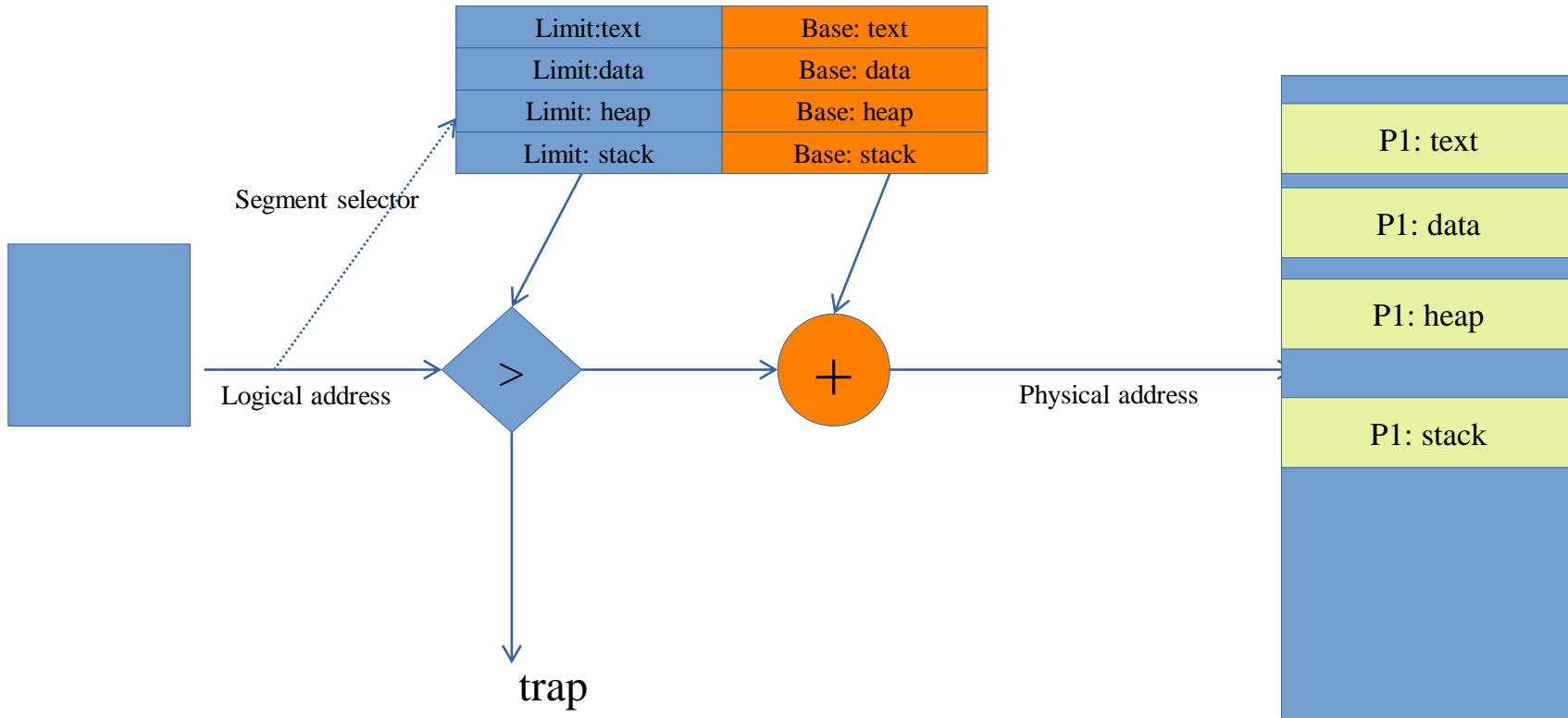
How?

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

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

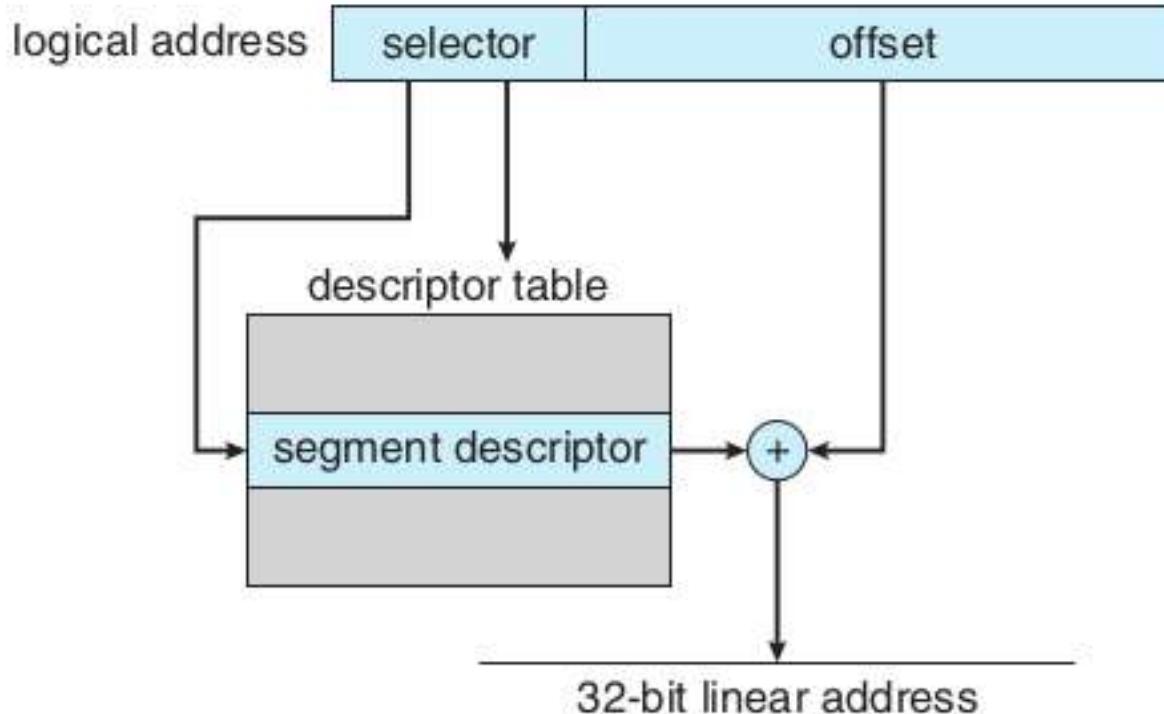


Figure 9.22 IA-32 segmentation.

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

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*

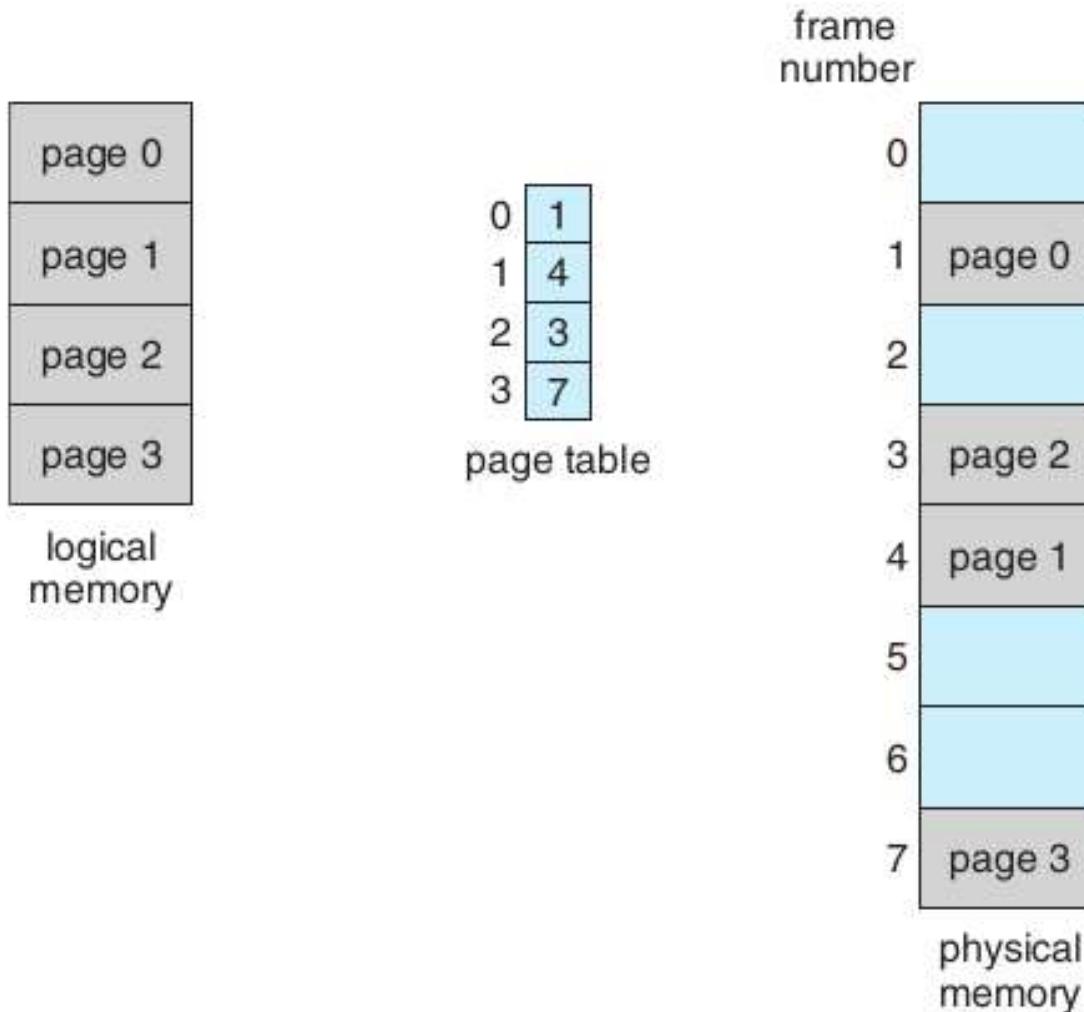


Figure 9.9 Paging model of logical and physical memory.

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.

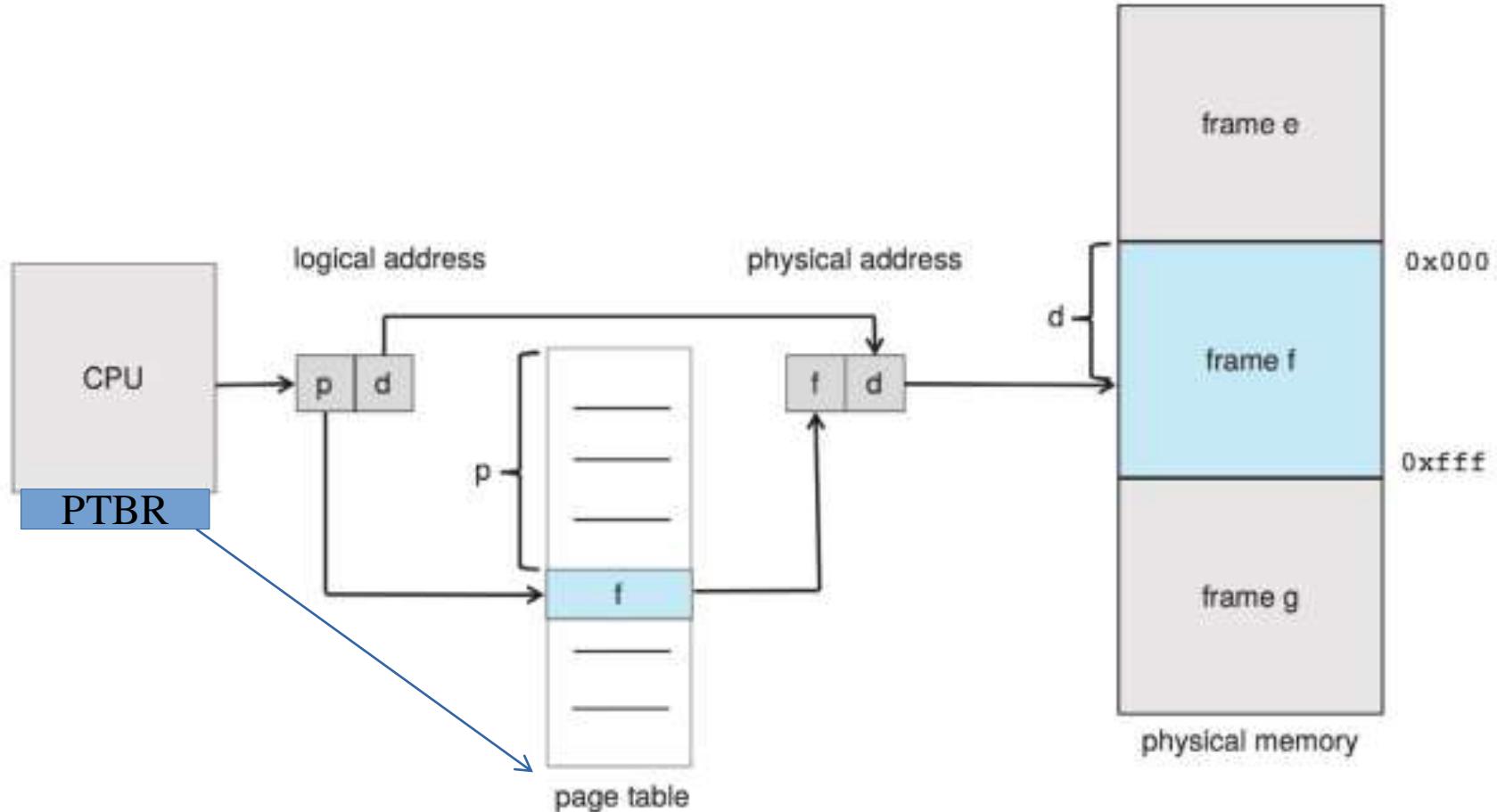


Figure 9.8 Paging hardware.

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

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

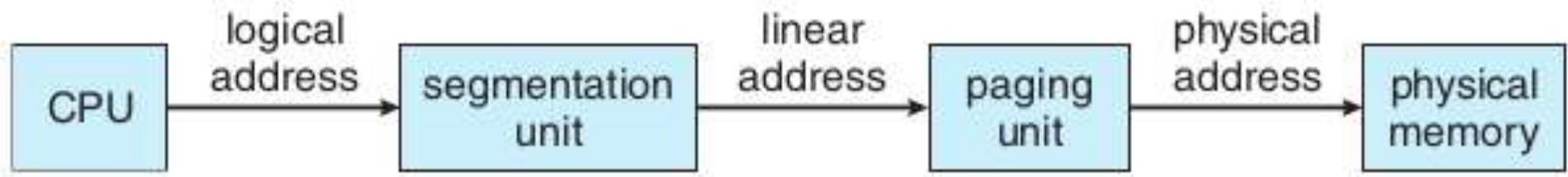


Figure 9.21 Logical to physical address translation in IA-32.

Segmentation in x86

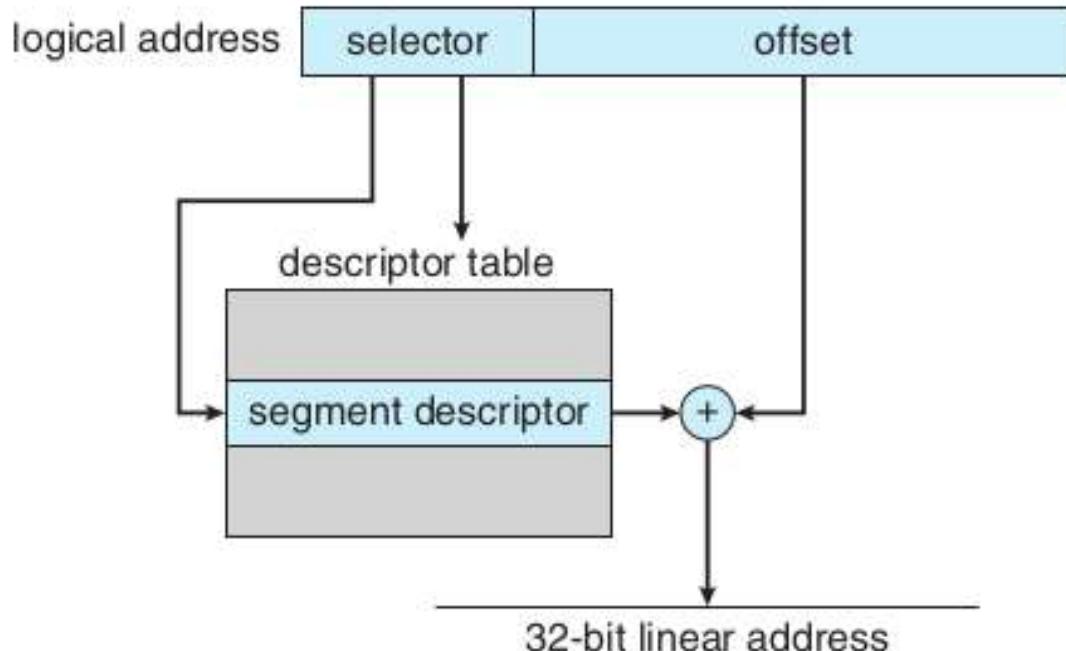


Figure 9.22 IA-32 segmentation.

.The selector is automatically chosen using Code Segment (CS) register, or Data Segment (DS) register depending on which type of

Paging in x86

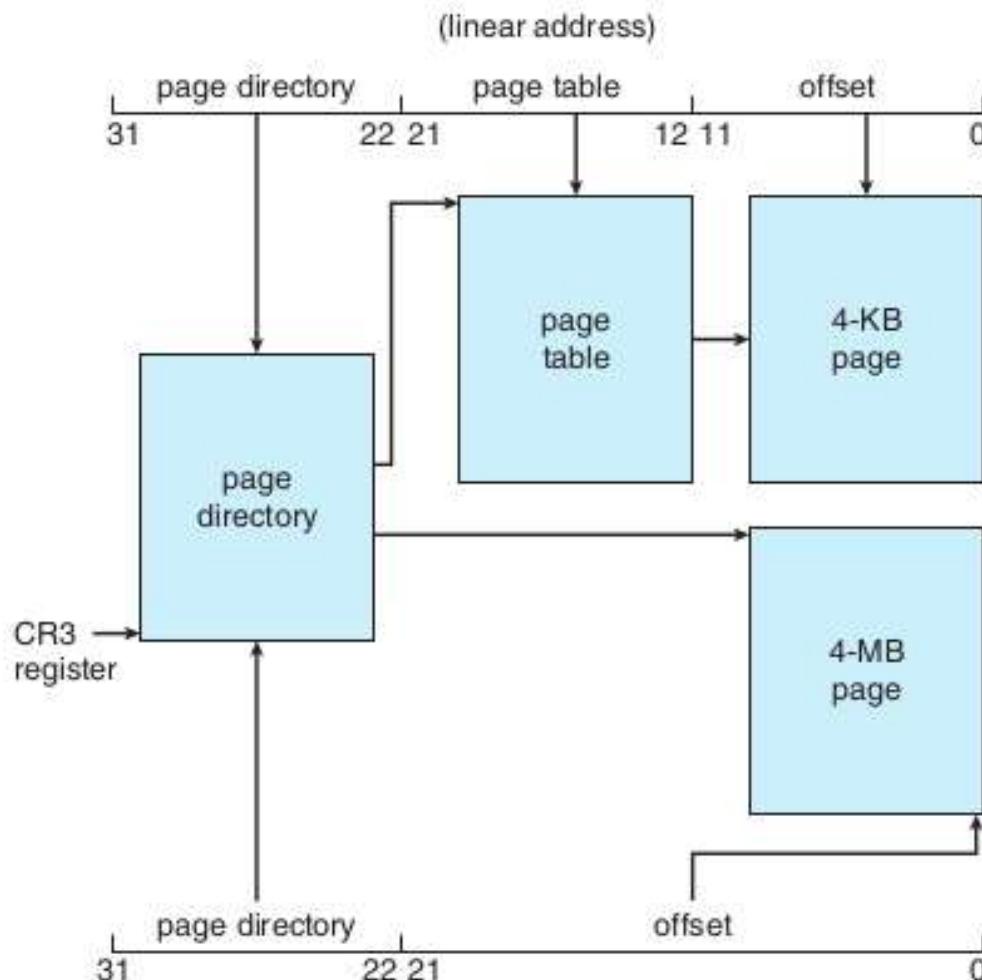


Figure 9.23 Paging in the IA-32 architecture.

- 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

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 [Module 1](#))

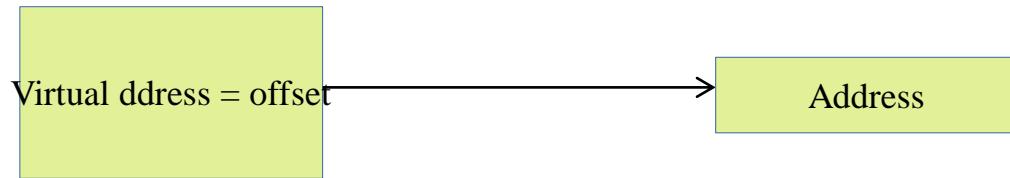
bootloader

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

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.

addr = seg << 4 + 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:

- Seg:off with 16 bit segments can actually address more than 20 bits of memory. After 0x100000 ($=2^{20}$), 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

After this

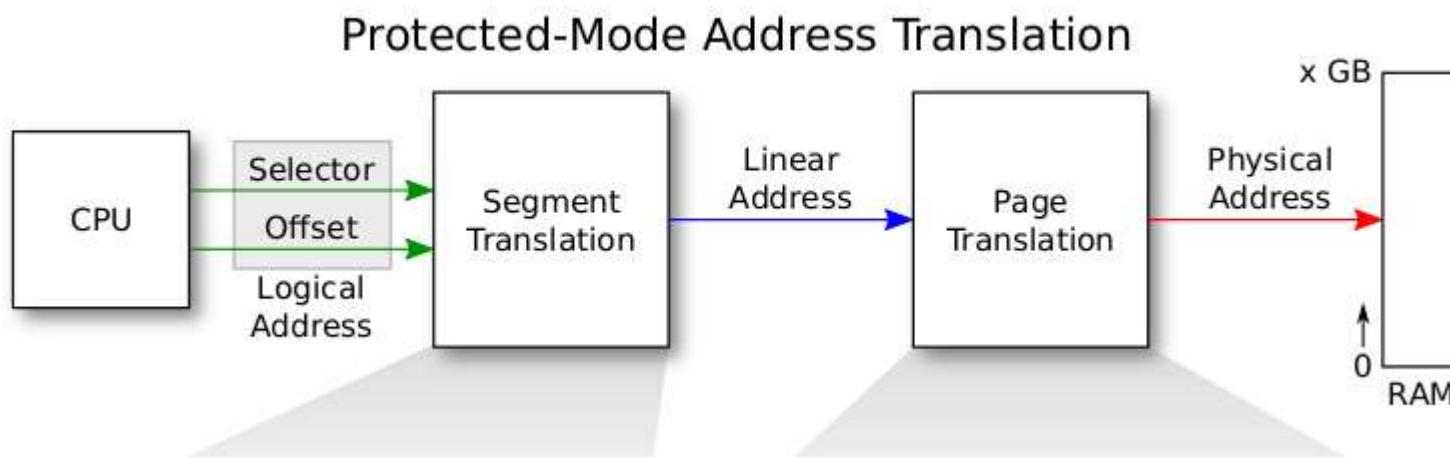
**Some instructions are run
to enter protected mode**

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

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

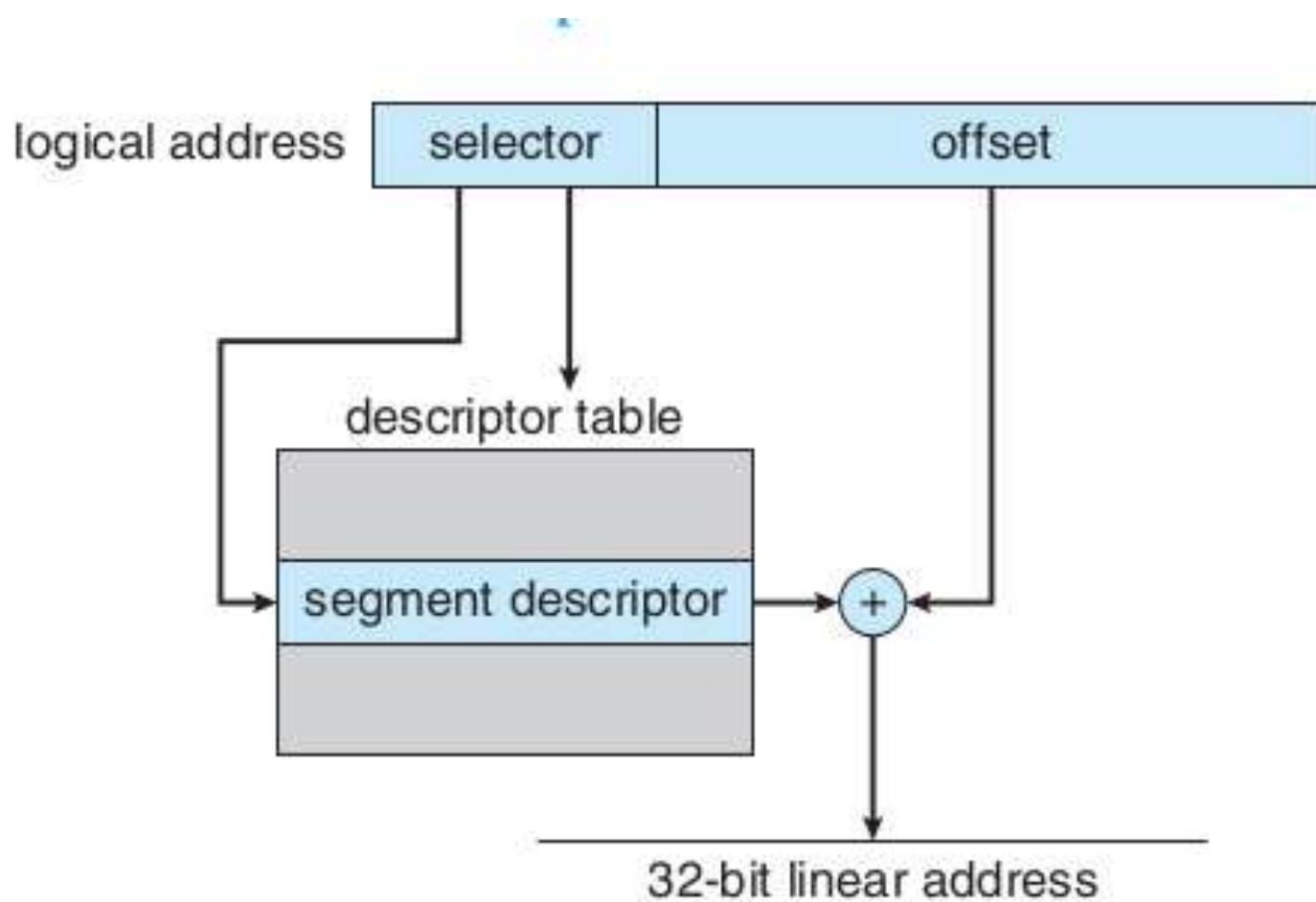


Figure 8.22 IA-32 segmentation.

Paging concept, hierarchical paging

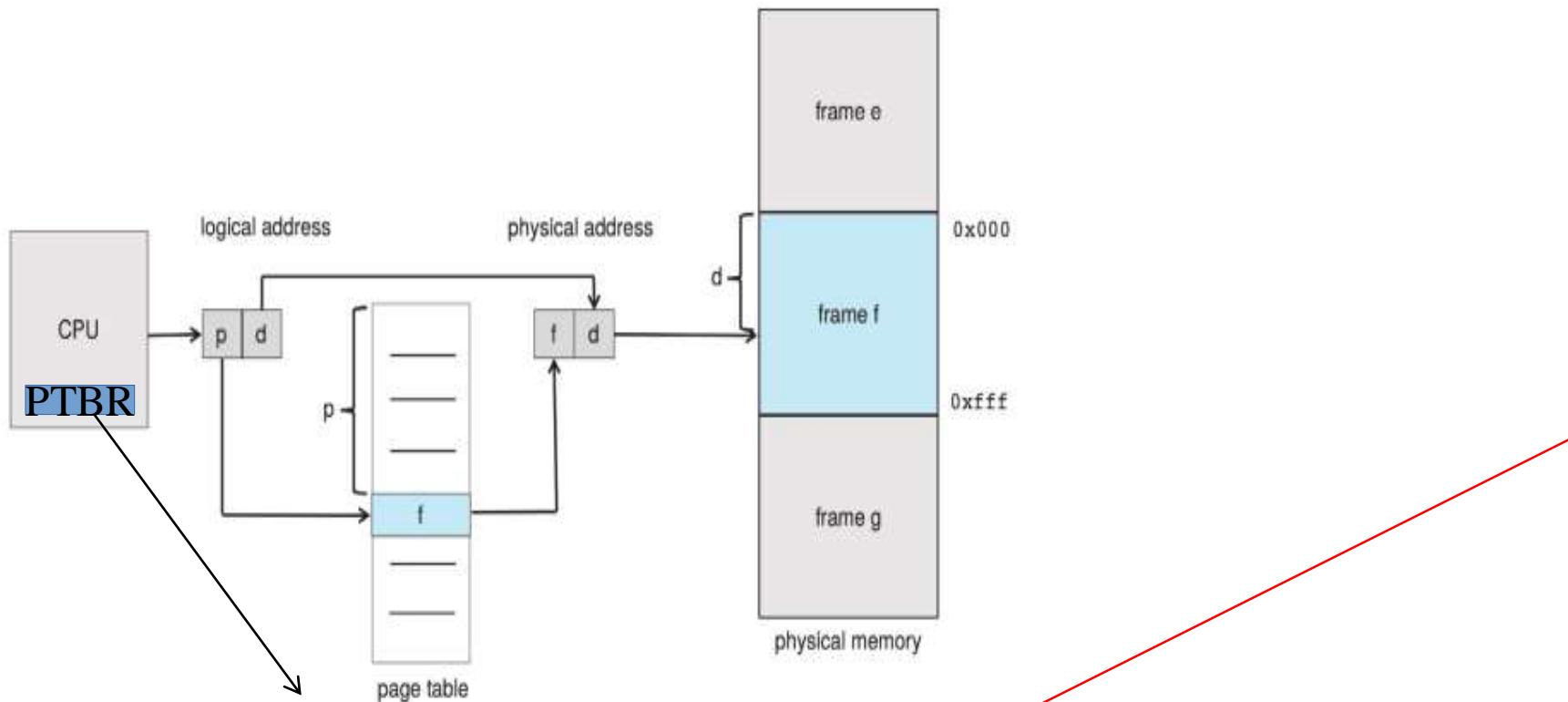


Figure 9.8 Paging hardware.

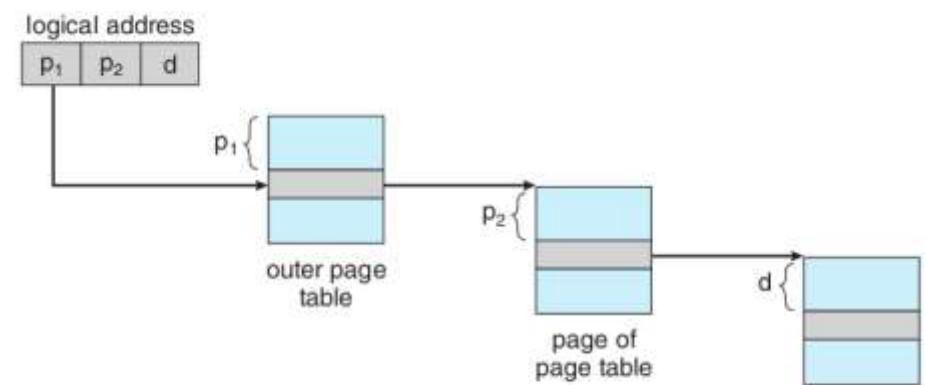


Figure 9.16 Address translation for a two-level 32-bit paging architecture.

X86 paging

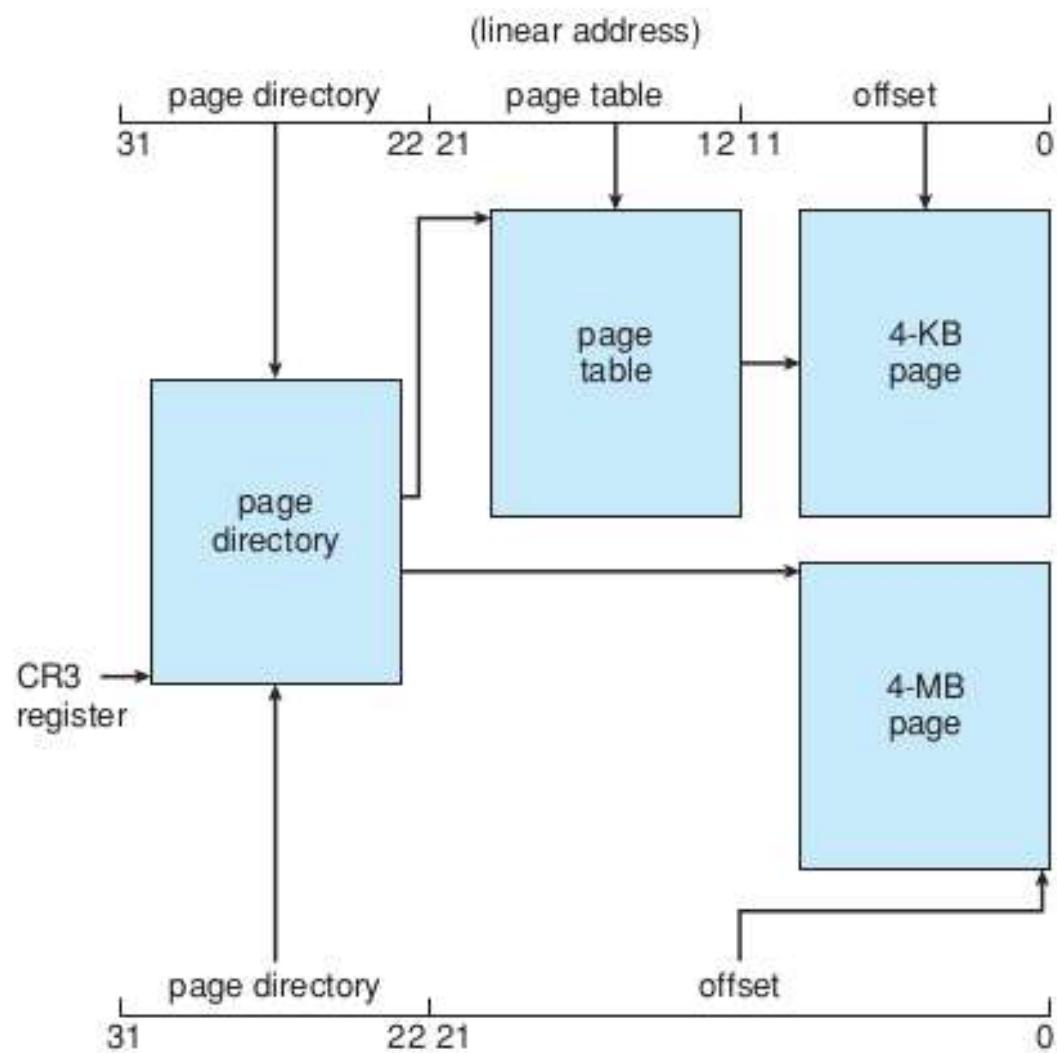


Figure 8.23 Paging in the IA-32 architecture.

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	G	P	S	0	A	C	W	D	T	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

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

PTE

CR3

CR3

	31	12 11	5 4 3 2	0
	Page-Directory-Table Base Address		P C D	P W T

PWT Page-level writes transparent

PCT Page-level cache disable

CR4

CR4

31	11	10	9	8	7	6	5	4	3	2	1	0
	O S X M	O S F X	P C G E	P G C E	M A S E	P A S E	P S E	D T S D	T P V I	P V M E		

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

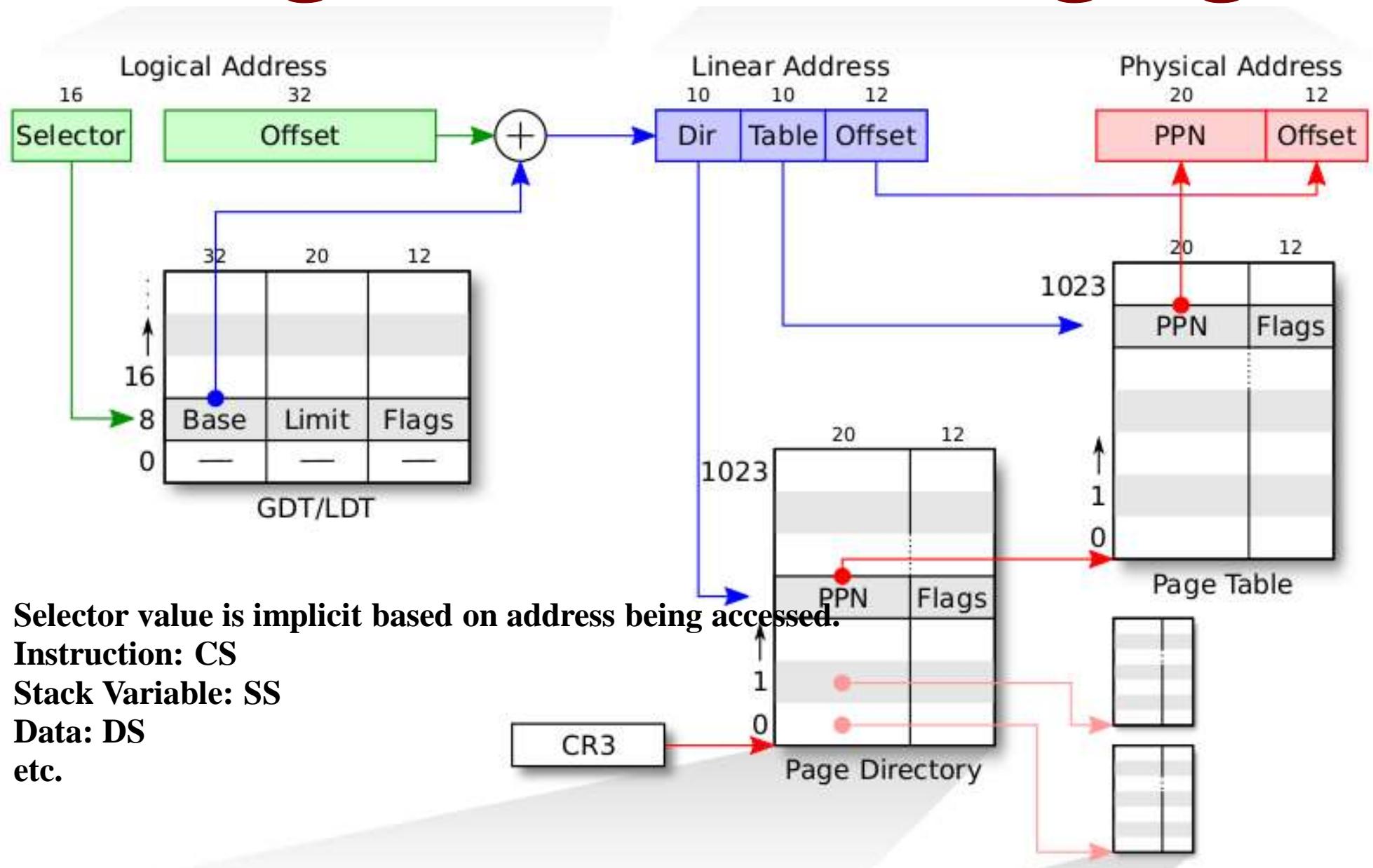
PCE Performance counter enable

OSFXSR OS FXSAVE/FXRSTOR support

OSXMM- OS unmasked exception support

EXCPT

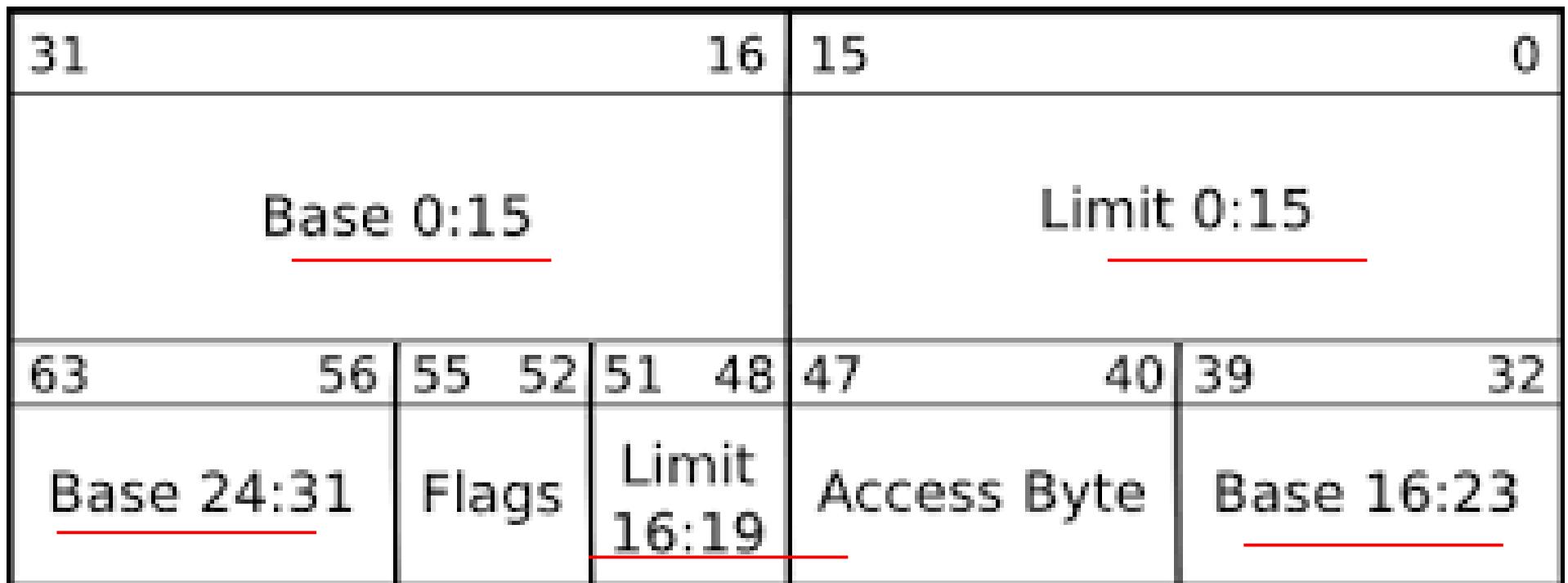
Segmentation + Paging



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

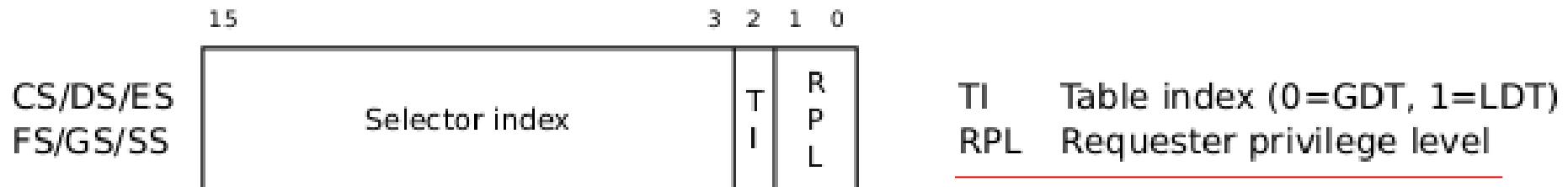
GDT Entry



asm.h

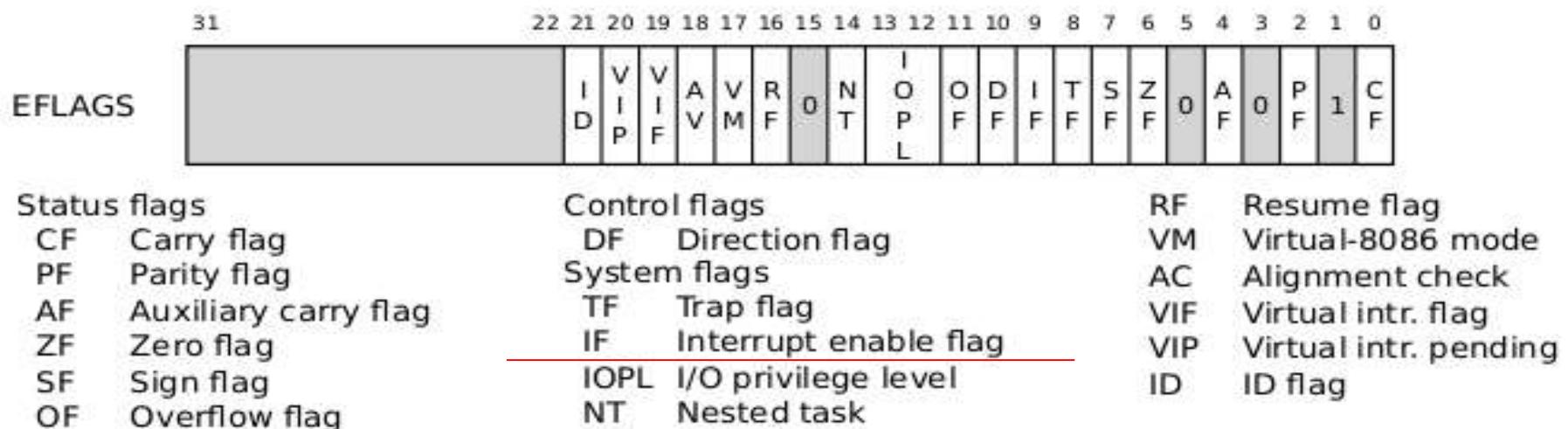
```
#define SEG_ASM(type,base,lim) \
    .word (((lim) >> 12) & 0xffff), ((base) & 0xffff); \
    .byte (((base) >> 16) & 0xff), (0x90 | (type)), \
          (0xC0 | (((lim) >> 28) & 0xf)), (((base) >> 24) & 0xff)
```

Segment selector



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

EFLAGS register



lgdt gdtdesc

...

Bootstrap GDT

.p2align 2 # force 4 byte alignment

gdt:

SEG_NULLASM #
null seg

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

lgdt

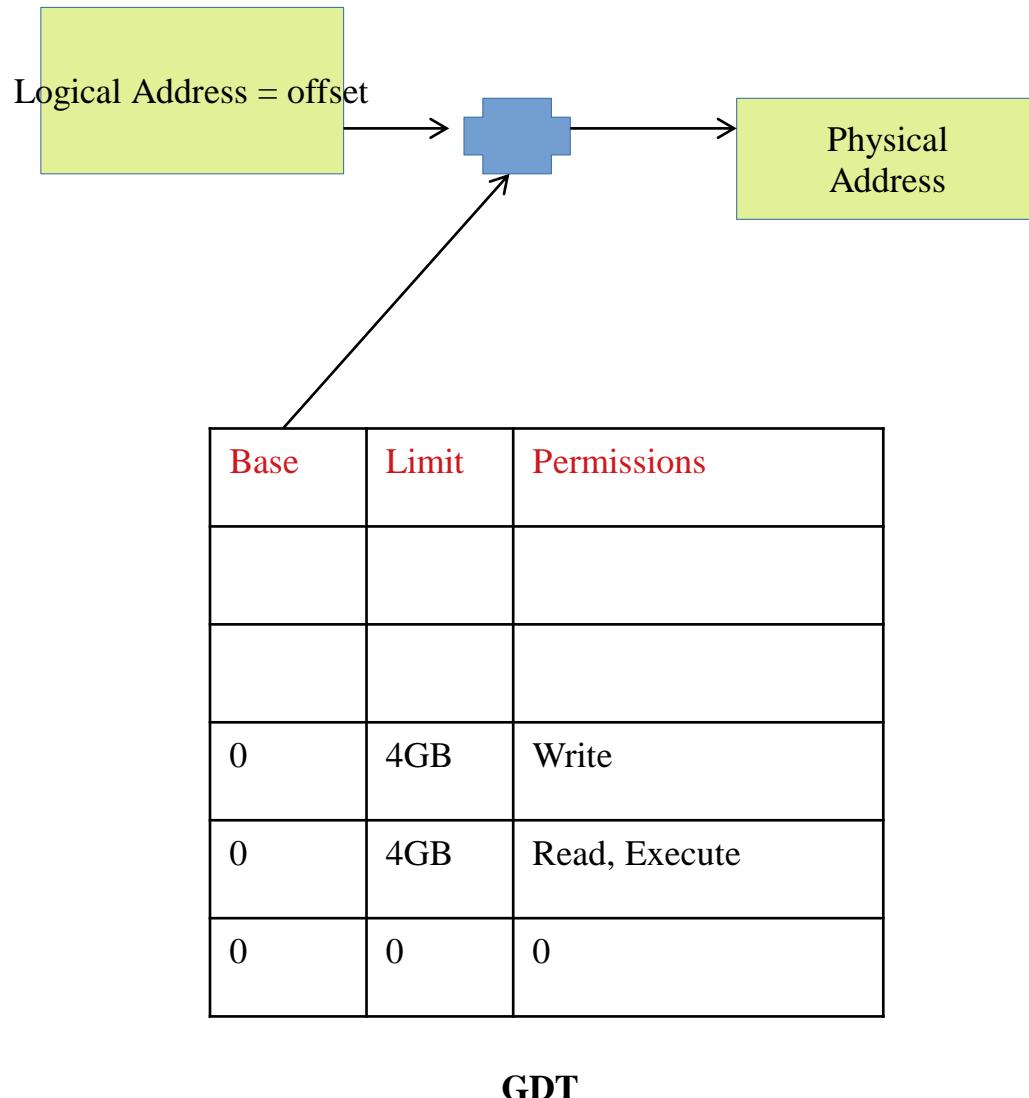
load the processor's (GDT) register with the value gdtdesc which points to the table gdt.

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

**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 p

Addresses in “kernel” file trans

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

	31 30 29 28	19 18 17 16 15	6 5 4 3 2 1 0
CR0	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
			NW Not write-through
			CD Cache disable
			PG Paging

PG: Paging enabled or not

WP: Write protection on/off

PE: Protection Enabled --> protected mode.

Complete transition to 32 bit mode

```
ljmp $(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 ljmp.

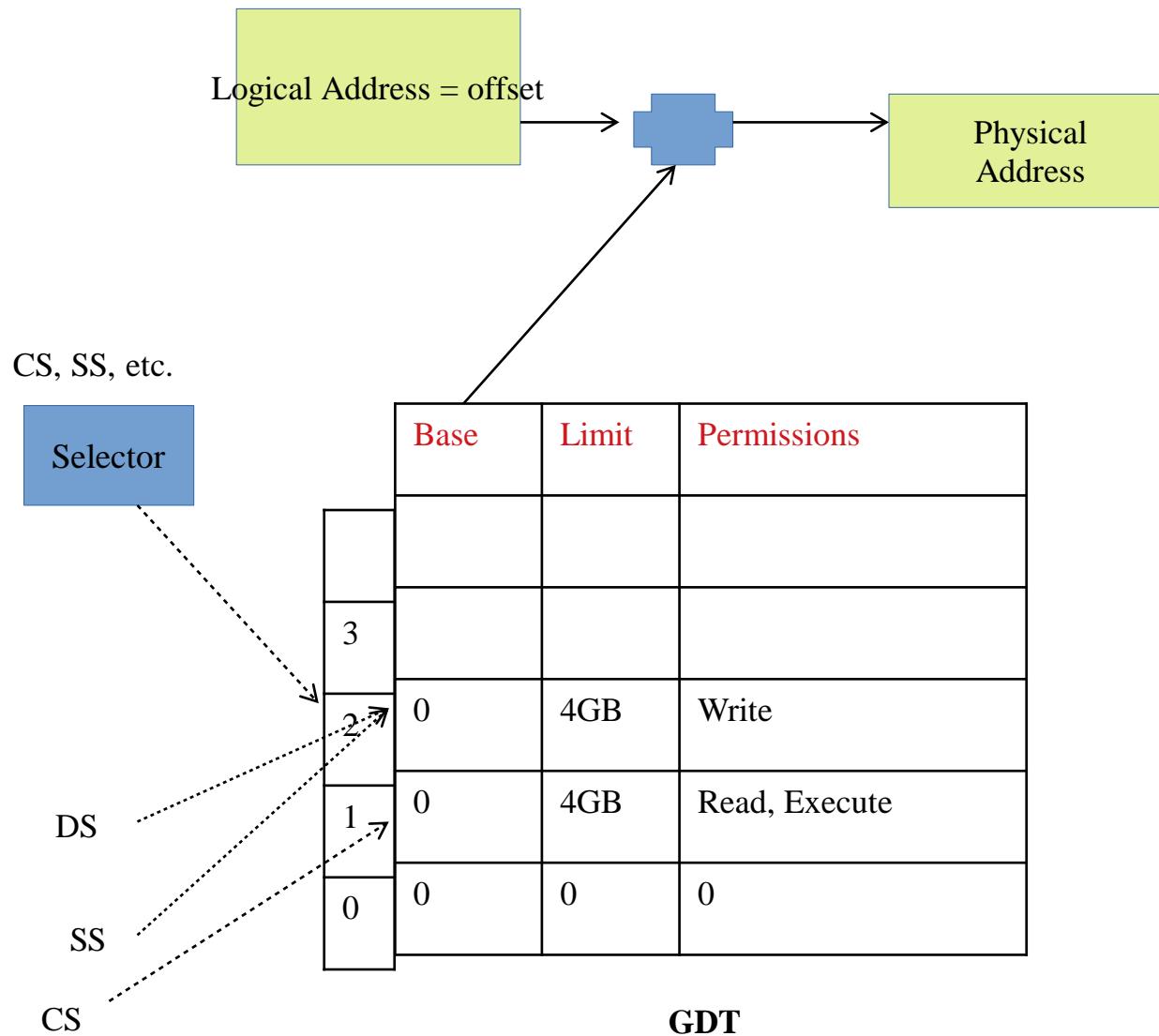
Note: The segment descriptors are set up with no translation (that

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
```

- 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-**

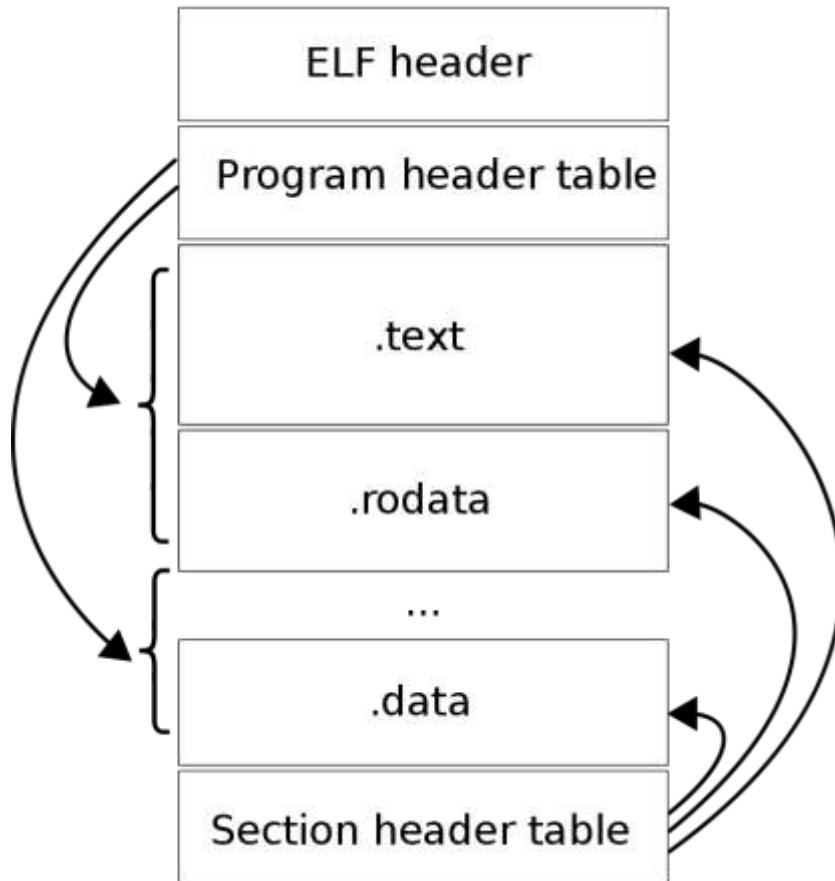
```
void  
bootmain(void)  
{  
    struct elfhdr *elf;  
    struct proghdr *ph,  
    *eph;  
    void (*entry)(void);  
    uchar* pa;
```

bootmain()

- Check if it's really ELF or not // Is this an ELF executable?

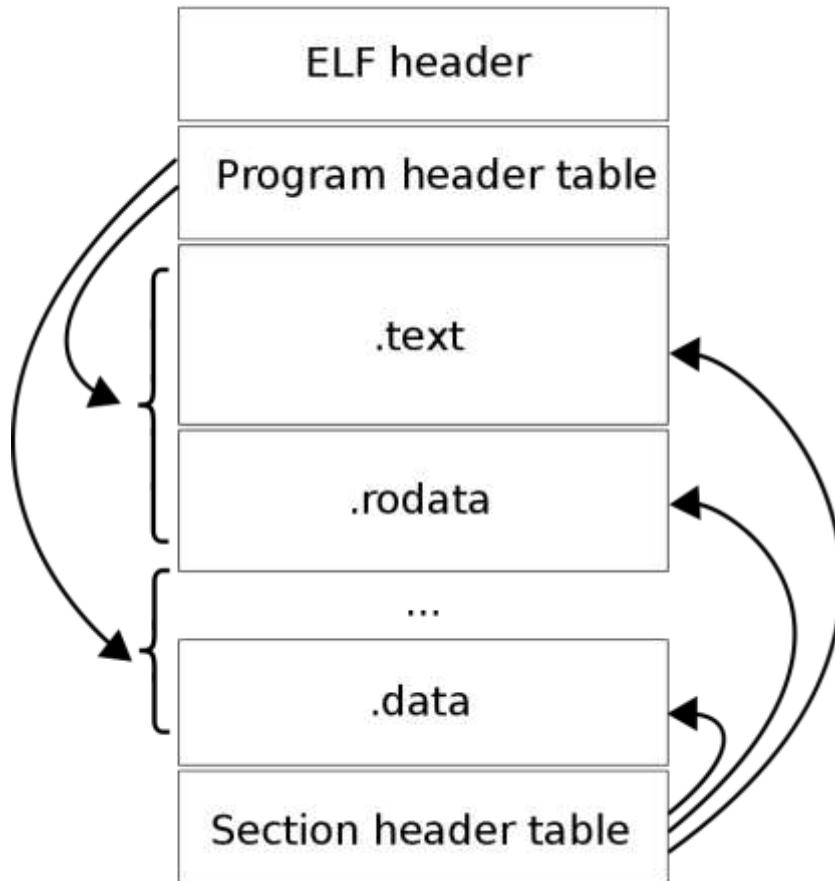
```
if(elf->magic != ELF_MAGIC)  
    return; // let  
bootasm.S handle error
```
- Next load kernel code from ELF file “kernel” into memory

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;
```

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.
```

kernel: Run ‘objdump -x -a kernel | head -15’ & see this
file format elf32-i386

kernel

architecture: i386, flags 0x00000112:

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

Stack : everything zeroes

Diff between mem
Code to be loaded at KERNBASE + KE

```
// Load each program segment (ignores ph flags).  
Load code from ELF to memory  
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).
```

To understand
further
code

Remember: 4
MB pages
are possible

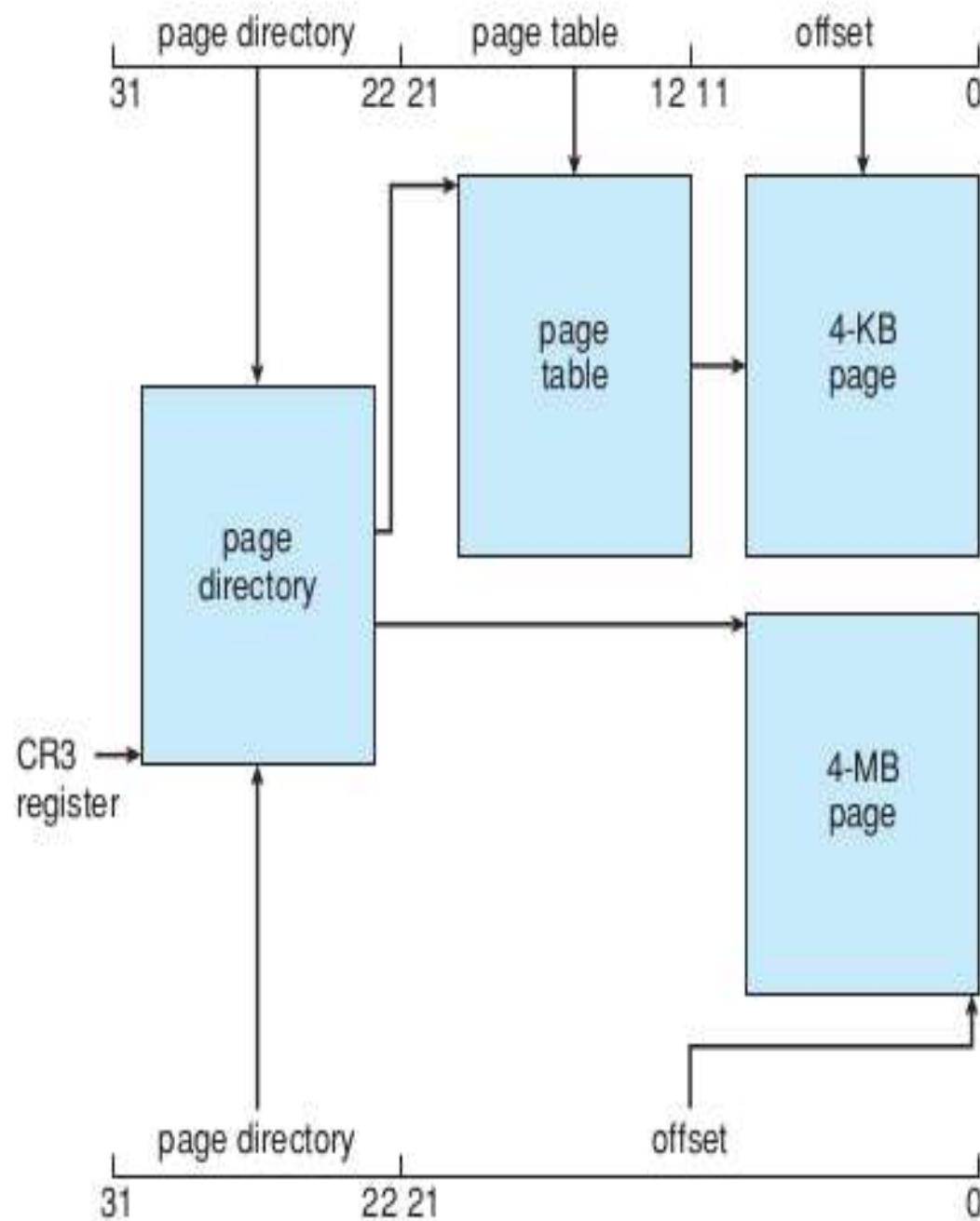
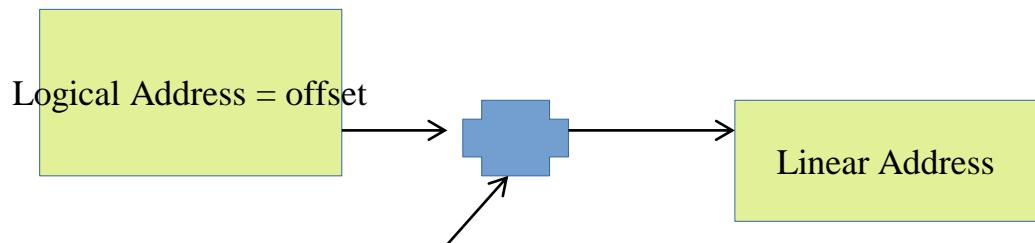


Figure 8.23 Paging in the IA-32 architecture.

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

RAM



CS, SS, etc.

Selector

	Base	Limit	Permissions
3			
2	0	4GB	Write
1	0	4GB	Read, Execute
0	0	0	0

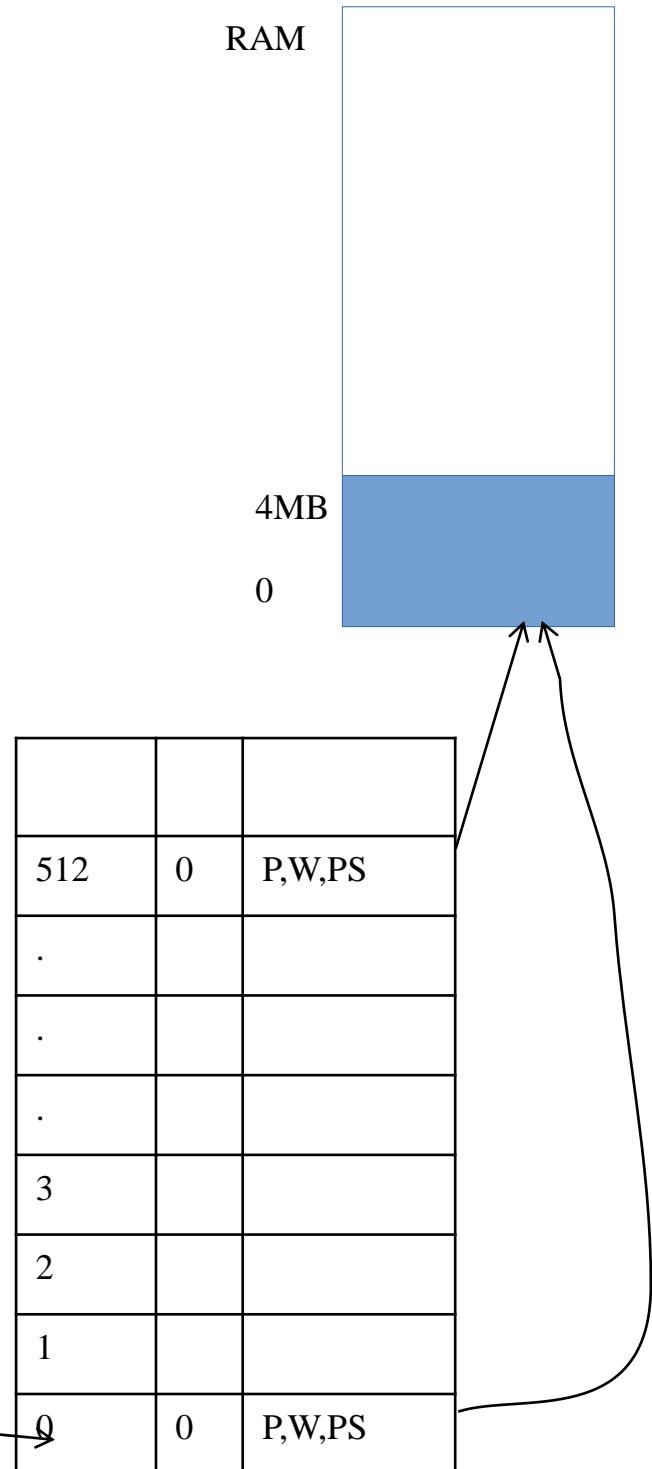
GDT

DS

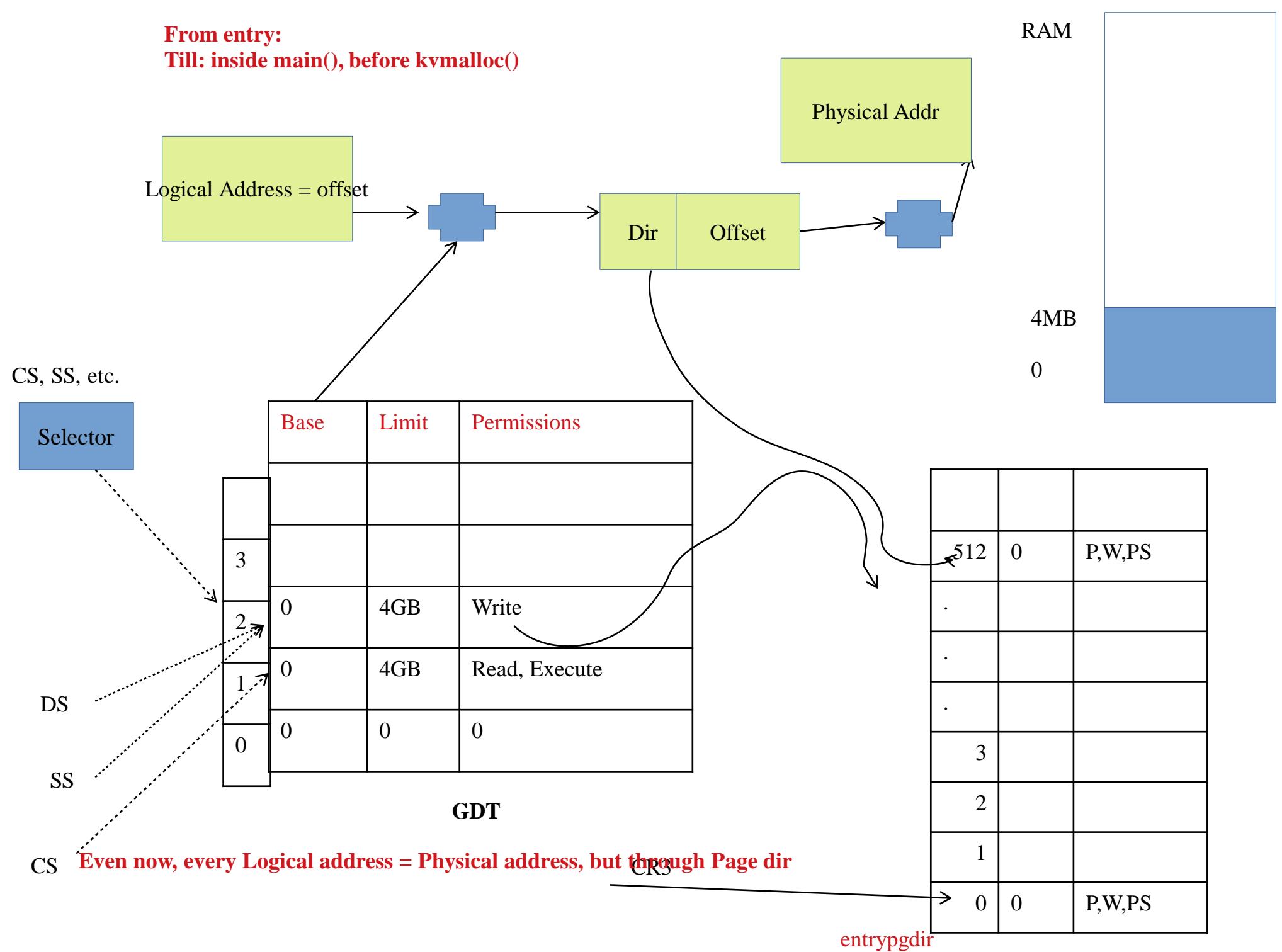
SS

CS

CR3 → entrypgdir



**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,  
  
};
```

#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 i

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.

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
```

- # 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

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

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

Introduction to the course on

Operating Systems

Sem VI 2021-22

3 Jan 2022



Remembering one
of the True teachers
on the **Teachers'**
day today

3rd Jan 2022

Before we go any further,

What do you want?

How do you want it to be different than CN?

Why is this course important?

In an era of “Data Science” why study “Operating systems” ?

Fashion is temporary, class is permanent.

Toppings keep changing, base does not!

Remember: DO NOT neglect the core courses in Computer Engineering!

Why is this course important?

What will this course teach you

To see “through” the black box called “system”

To see, describe, analyze

EVERYTHING that happens on your computer

Why is this course important?

What will this course teach you

**Remove many MISCONCEPTIONS about the
“system”**

**Unfortunately you kept assuming many things
about what happens “inside”**

Why is this course important?

What will this course teach you

**Clearly bring out what happens in hardware and
what happens in software**

**Fill in the gap between what you learnt in
Microprocessors+CO and any other programming
course**

Why is this course important?

This course will

**Challenge you to the best of your programming
abilities!**

Give you many (more) sleepless nights.

Why is this course important?

This course will

**Help you solve most of the problems that occur on
your computer! Also teach you to raise the most
critical unsolved problems in Operating systems.**

Be a mechanic and a scientist at the same time!

Why is this course important?

Do we have jobs in Operating Systems domain?

Yes!

**Veritas, IBM, Microsoft, NetApp, Amazon (AWS),
Oracle, Seagate, Vmware, RedHat, Apple, Google,
Intel, Honeywell, HP, ...**

Operating Systems: Introduction

Abhijit A. M.

abhijit.comp@coep.ac.in

(C) Abhijit A.M.

Available under Creative Commons Attribution-ShareAlike License V3.0+

Credits: Slides of “OS Book” ed10.

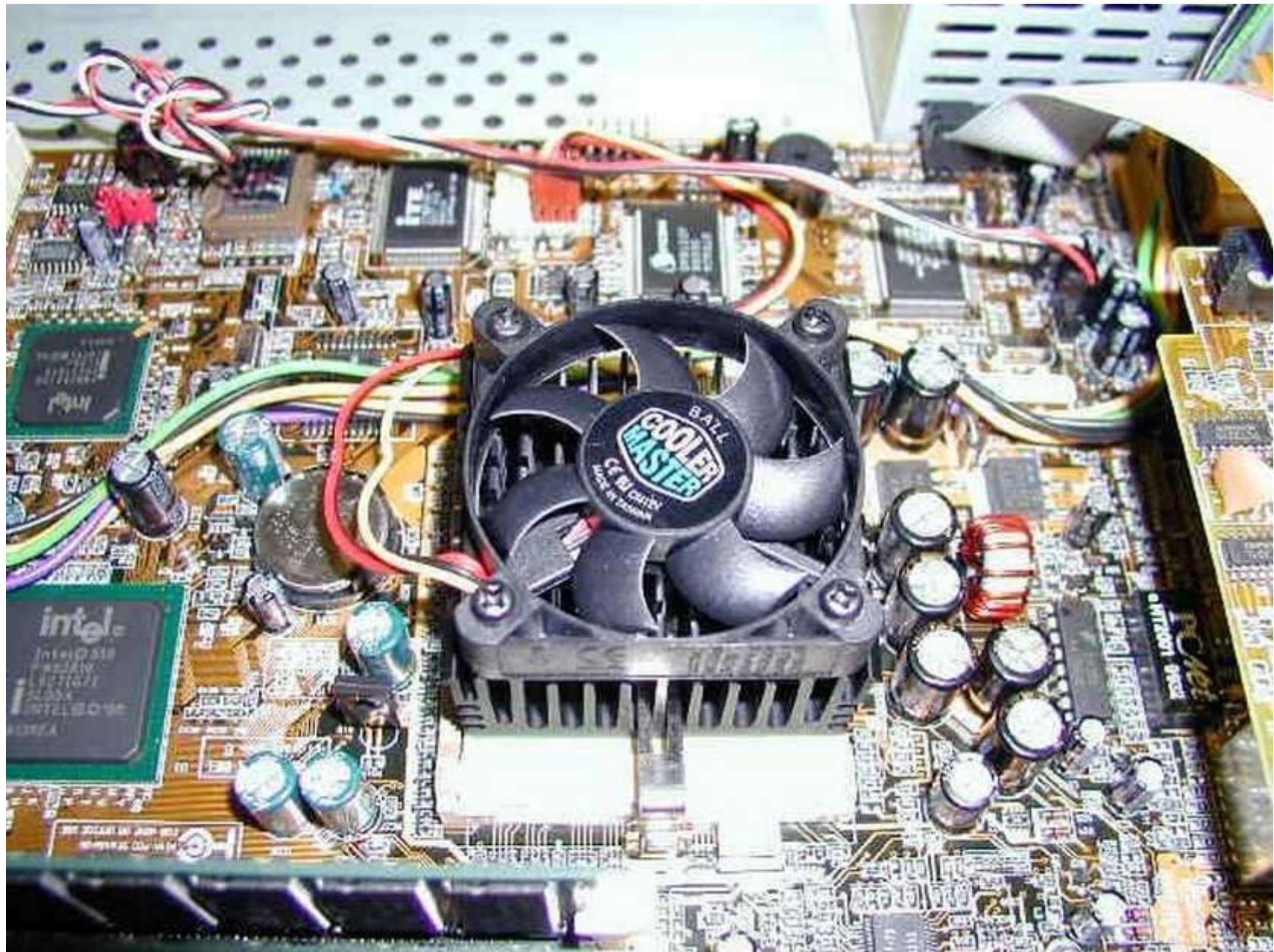
Initial lectures

We will solve a jigsaw puzzle
of how the computer system is built
with hardware, operating system and system programs

The “Ports”, what users see



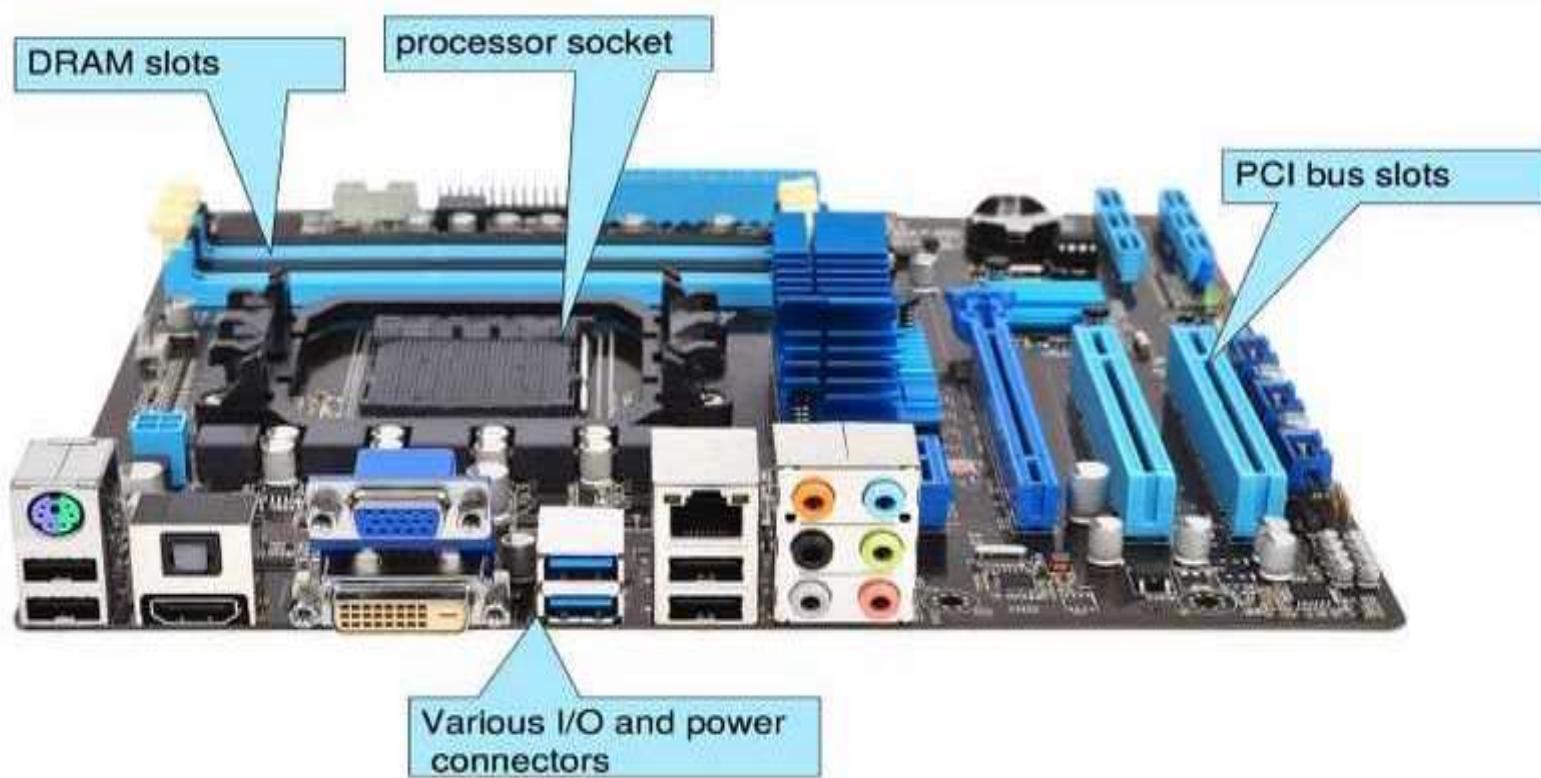
Revision: Hardware : The Motherboard



CPU/Processor

- I3,i5, etc.
- Speeds: Ghz
- “Brain”
- Runs “machine instructions”
- The actual “computer”
- Questions:
 - Where are the instructions that the processor runs?

What's on the motherboard?

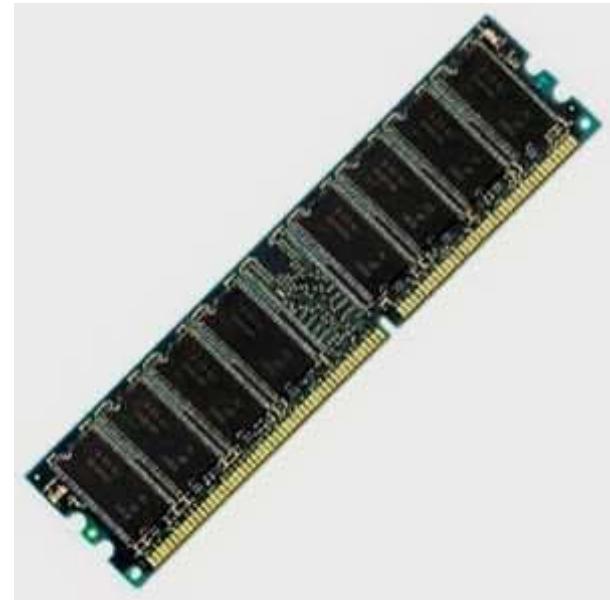


This board is a fully-functioning computer, once its slots are populated.

Memory

- Random Access Memory (RAM)

- Same time of access to any location – randomly accessible
- Semiconductor device



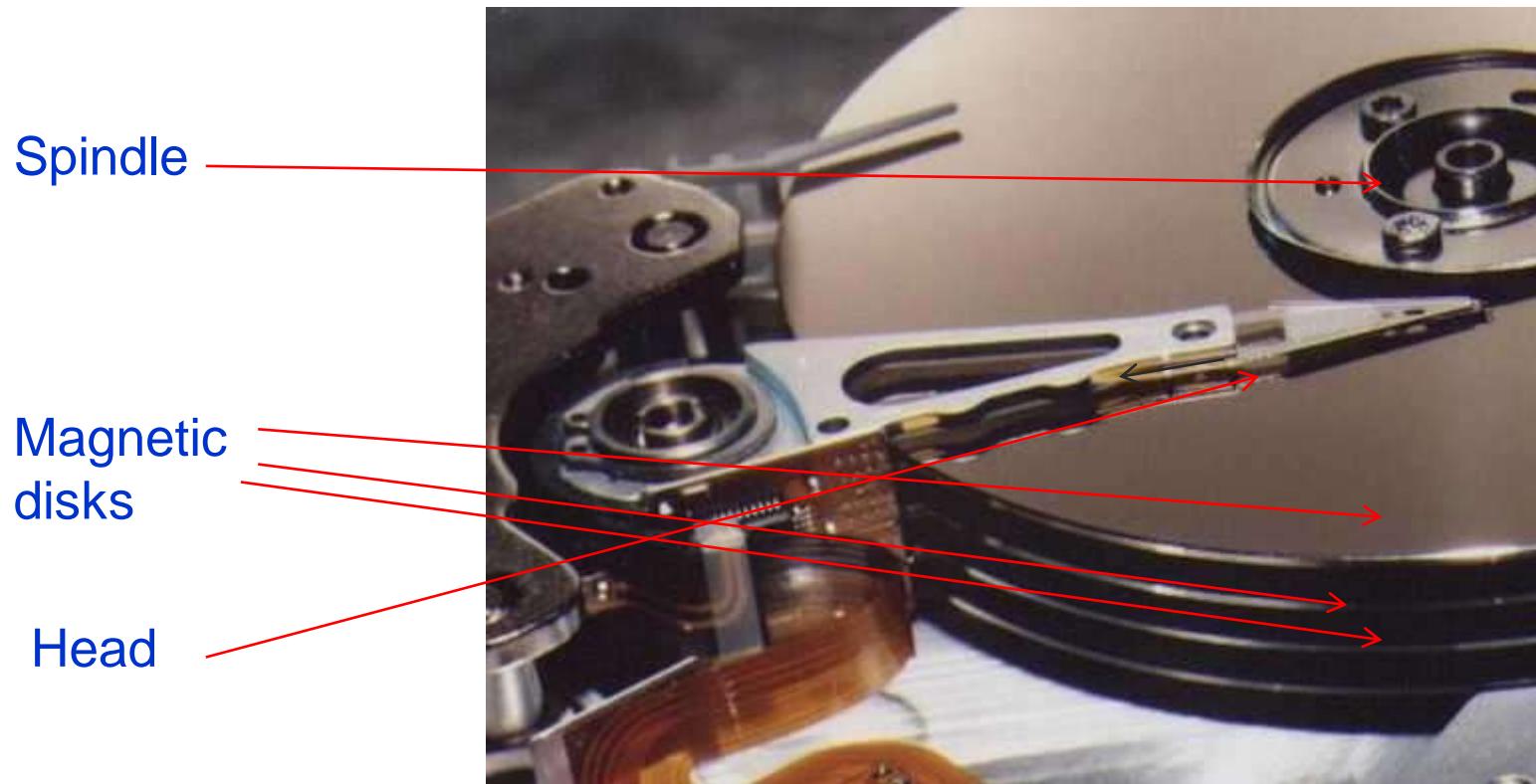
Question:

Can we add more RAM to a computer?

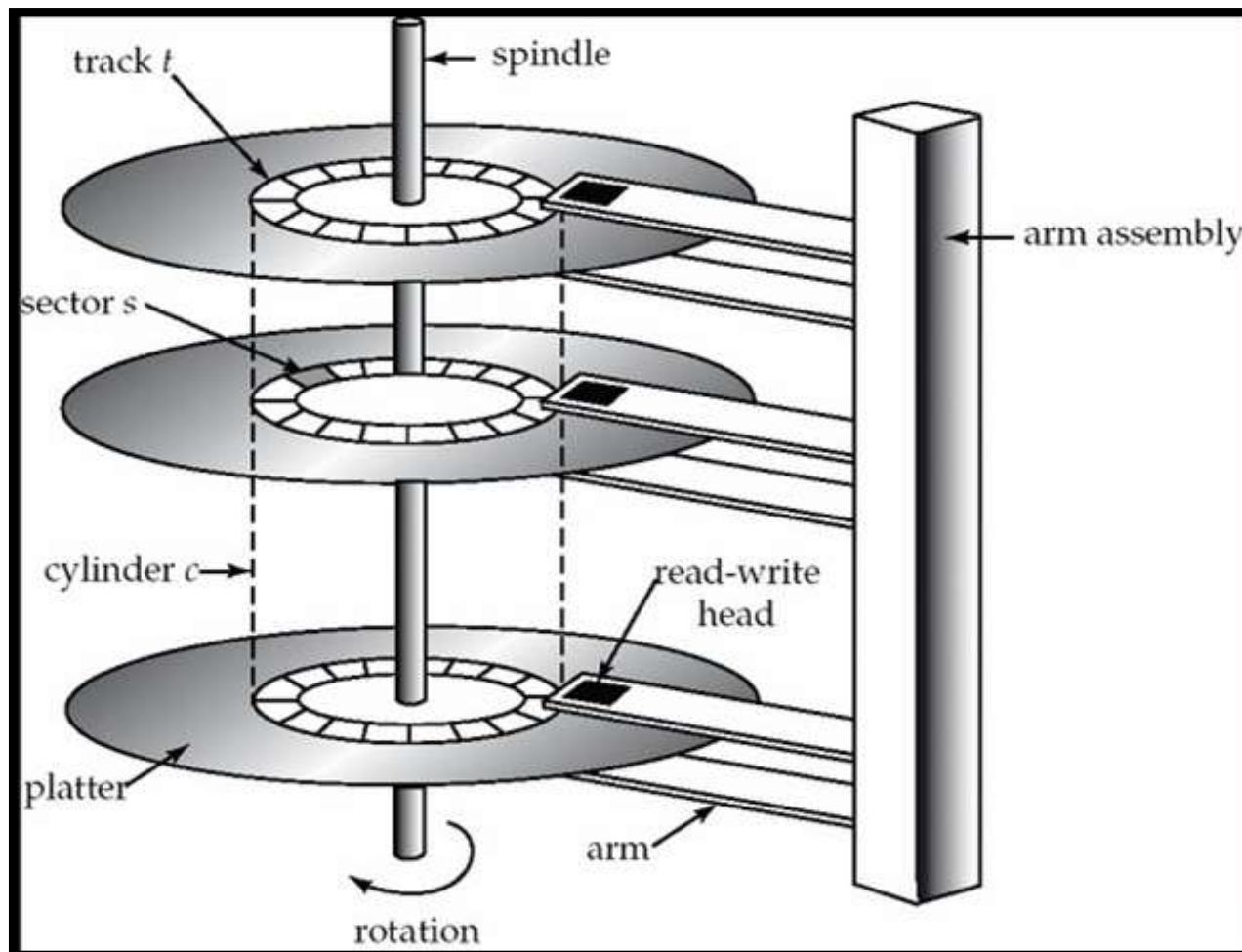
The Hard Drive



The Hard Drive



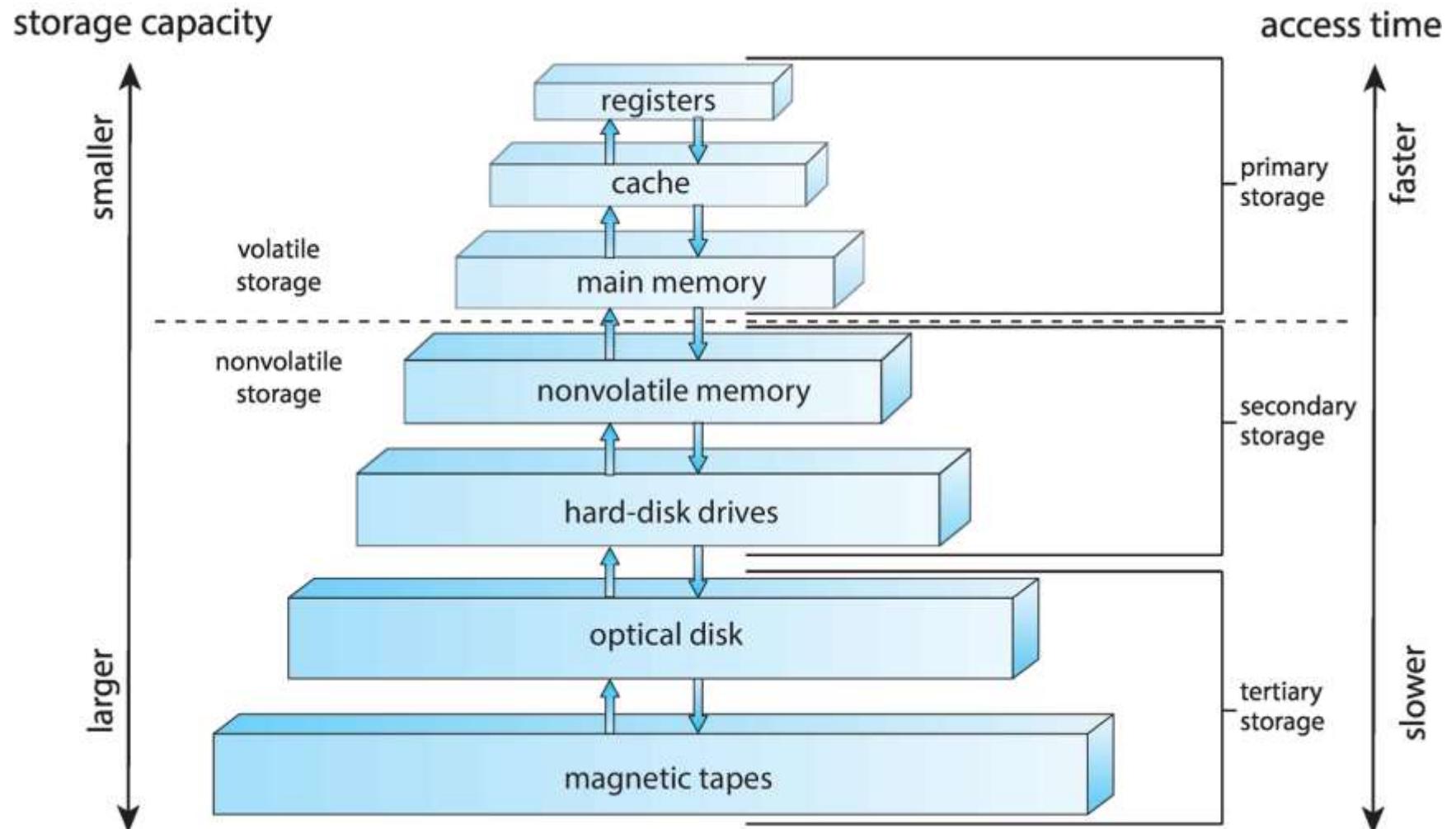
The hard Drive



The Hard Drive

- Is a Magnetic device
- Each disk divided into tiny magnetic spots, each representing 1 or 0
 - What's the physics ?
 - Two orientations of a magnet
- Is “persistent”
 - Data stays on powering-off
- Is slow
- IDE, SATA, SCSI, PATA, SAS, ...

Storage-Device Hierarchy



Level	1	2	3	4	5
Name	registers	cache	main memory	solid state disk	magnetic disk
Typical size	< 1 KB	< 16MB	< 64GB	< 1 TB	< 10 TB
Implementation technology	custom memory with multiple ports CMOS	on-chip or off-chip CMOS SRAM	CMOS SRAM	flash memory	magnetic disk
Access time (ns)	0.25 - 0.5	0.5 - 25	80 - 250	25,000 - 50,000	5,000,000
Bandwidth (MB/sec)	20,000 - 100,000	5,000 - 10,000	1,000 - 5,000	500	20 - 150
Managed by	compiler	hardware	operating system	operating system	operating system
Backed by	cache	main memory	disk	disk	disk or tape

Figure 1.11 Performance of various levels of storage.

Computer Organization

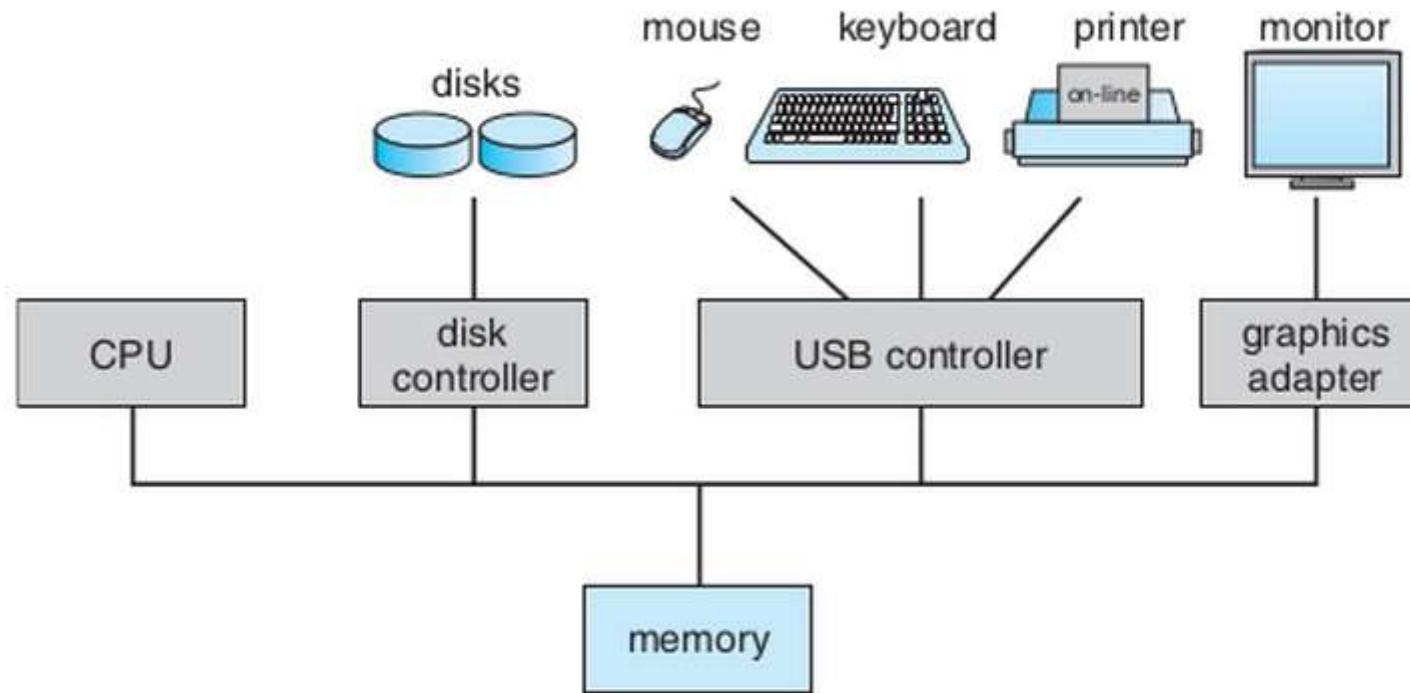
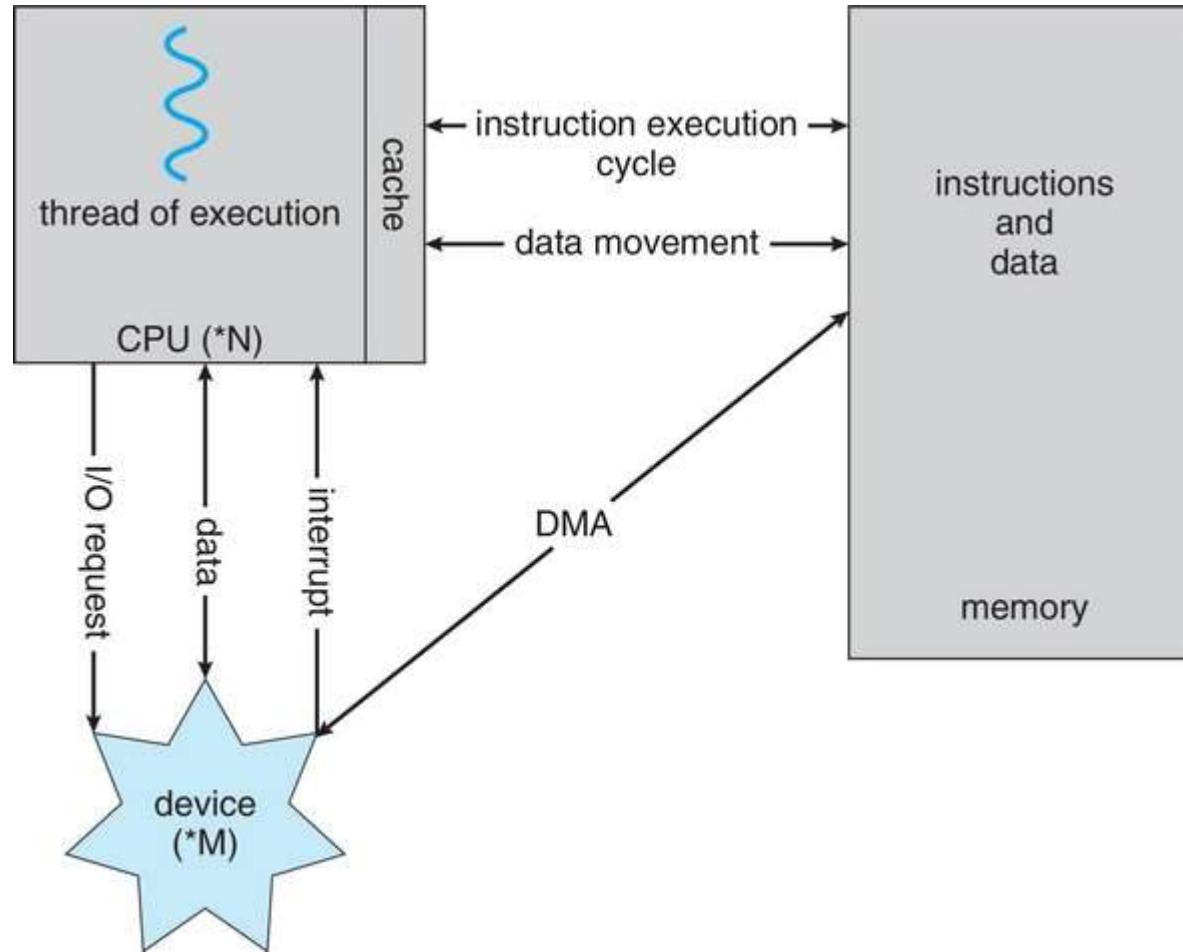


Figure 1.2 A modern computer system.

Important Facts

- Processor (CPU) transfers data between itself and main memory(RAM) only!
- No data transfer between CPU and Hard Disk, CPU and Keyboard, CPU and Mouse, etc.
- I/O devices transfer (how?) data to memory(RAM) and CPU instructions access data from the RAM

How a Modern Computer Works



A von Neumann architecture

What does the processor do?

- From the moment it's turned on until it's turned off, the processor simply does this
 - 1) Fetch the instruction from RAM (Memory).
Location is given by Program Counter (PC) register
 - 2) Decode the instruction and execute it
While doing this may fetch some data from the RAM
 - 3) While executing the instruction change/update the Program Counter
 - 4) Go to 1

Immediate questions

- What's the initial value of PC when computer starts ?
- Who puts "this" value in PC ?
- What is there at the initial location given by PC ?

A critical question you need to keep thinking about ...

- Throughout this course, With every concept that you study, Keep asking this question
- Which code is running on the processor?
 - Who wrote it?
 - Which code ran before it
 - Which code can run after it
- Basically try to understand the flow of instructions that execute on the processor

Few terms

▫ BIOS

- The code “in-built” into your hardware by manufacturer
- Runs “automatically” when you start computer
- Keeps looking for a “boot loader” to be loaded in RAM and to be executed

▫ Boot Loader

- A program that exists on (typically sector-0 of) a secondary storage
- Loaded by BIOS in RAM and passed over control to

Kernel, System Programs, Applications

❑ Kernel

- ❑ The code that is loaded and given control by BIOS initially when computer boots
- ❑ Takes control of hardware (how?)
- ❑ Creates an environment for “applications” to execute
- ❑ Controls access to hardware by applications,
- ❑ Etc.

❑ Everything else is “applications”

- ❑ System programs: applications that depend

How is a modern day
Desktop system
built
on top of
this type of hardware?

Components of a computer system

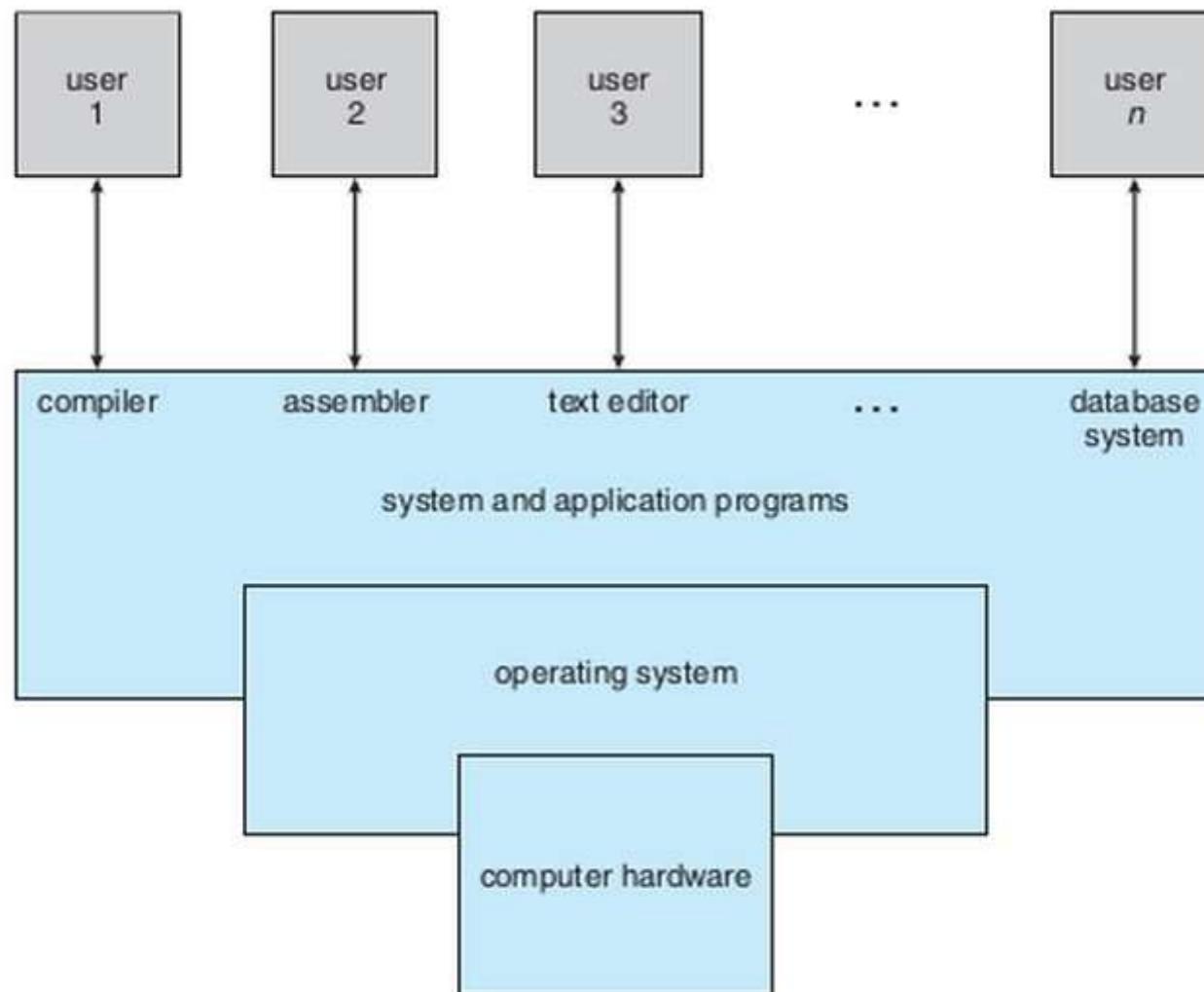


Figure 1.1 Abstract view of the components of a computer system.

Multiprocessor system: SMP

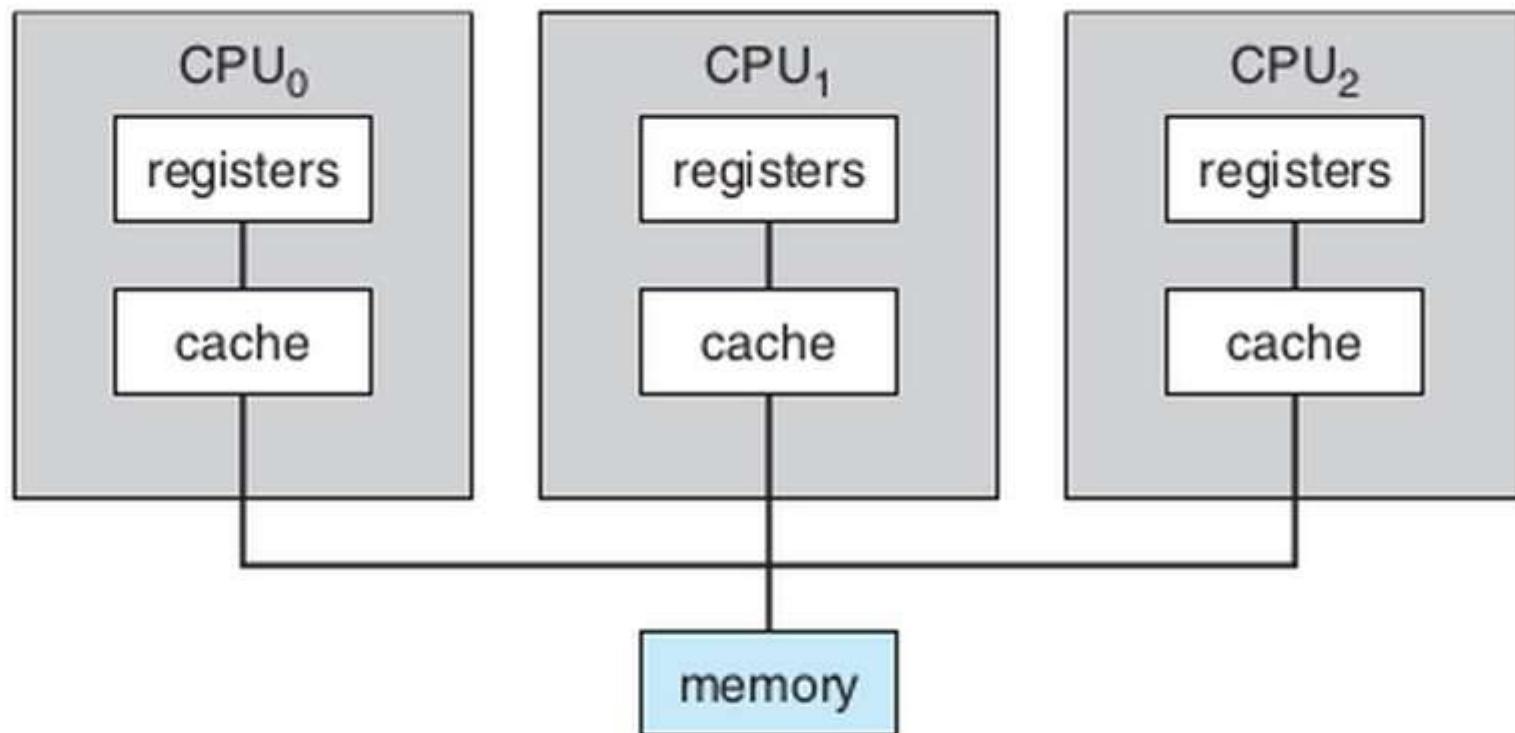


Figure 1.6 Symmetric multiprocessing architecture.

Dual Core: what's that?

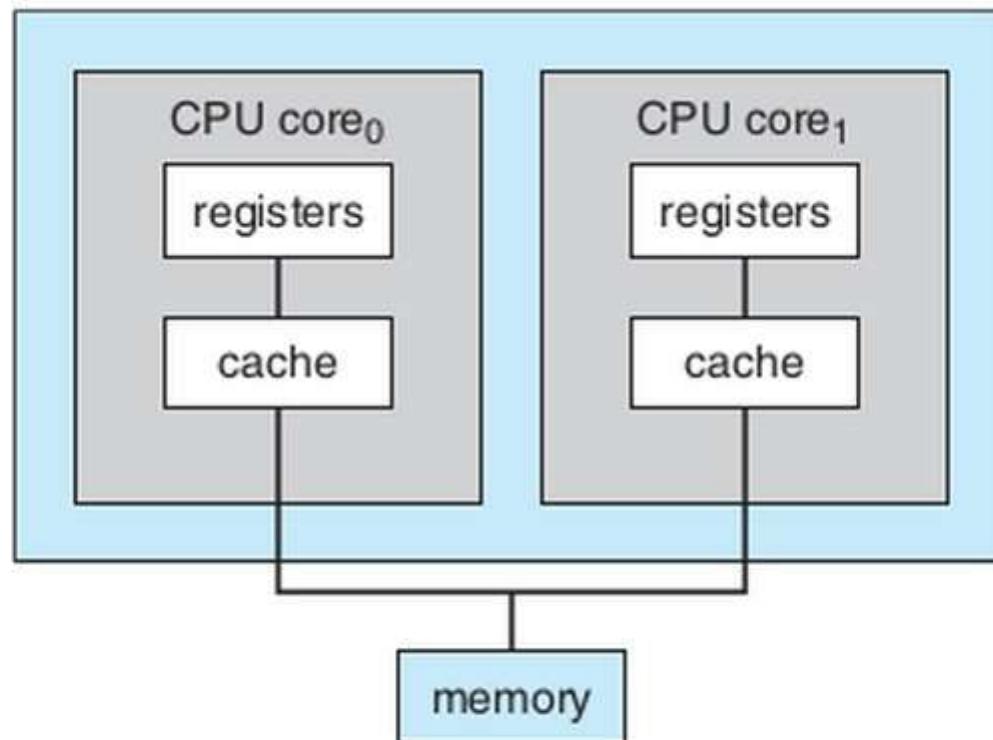


Figure 1.7 A dual-core design with two cores placed on the same chip.

Who does what?

What is done in hardware, by OS, by compiler, by linker, by loader, by human end-user?

There is no magic!

A very intelligent division of work/labour between different components of the computer system makes a system

Event Driven kernel

Multi-tasking, Multi-programming

Earlier...

- Boot process

- BIOS -> Boot Loader -> OS -> “init” process

- CPU doing

```
for(;; {
```

```
    fetch from @PC;
```

```
    deode+execute;
```

```
    PC++/change PC
```

```
}
```

Understanding hardware interrupts

- Hardware devices (keyboard, mouse, hard disk, etc) can raise “hardware interrupts”
- Basically create an electrical signal on some connection to CPU (/bus)
- This is notified to CPU (in hardware)
- Now CPU’s normal execution is interrupted!
 - What’s the normal execution?
 - CPU will not continue doing the fetch, decode, execute, change PC cycle !
 - What happens then?

Understanding hardware interrupts

□ On Hardware interrupt

- The PC changes to a location pre-determined by CPU manufacturers!
- Now CPU resumes normal execution
 - What's normal?
 - Same thing: Fetch, Decode, Execute, Change PC, repeat!
 - But...
 - But what's there at this pre-determined address?

Boot Process

- BIOS runs “automatically” because at the initial value of PC (on power ON), the manufacturers have put in the BIOS code in ROM
 - CPU is running BIOS code . BIOS code also occupies all the addresses corresponding to hardware interrupts.
- BIOS looks up boot loader (on default boot device) and loads it in RAM, and passes control over to it
 - Pass control? - Like a JMP instruction
 - CPU is running boot loader code

Boot loader gets boot option from human user and

Hardware interrupts and OS

□ When OS starts running initially

- It copies relevant parts of it's own code at all possible memory addresses that a hardware interrupt can lead to!**
- Intelligent, isn't' it?**

□ Now what?

- Whenever there is a hardware interrupt – what will happen?**
- The PC will change to predetermined location, and control will jump into OS code**

So remember: whenever there is a hardware

Key points

- Understand the interplay of hardware features + OS code + clever combination of the two to achieve OS control over hardware
- Most features of computer systems / operating systems are derived from hardware features
 - We will keep learning this throughout the course
 - Hardware support is needed for many OS features

Time Shared CPU

- Normal use: after the OS has been loaded and Desktop environment is running
- The OS and different application programs keep executing on the CPU alternatively (more about this later)
 - The CPU is time-shared between different applications and OS itself
- Questions to be answered later
 - How is this done?
 - How does OS control the time allocation to

Multiprogramming

□ Program

- Just a binary (machine code) file lying on the hard drive. E.g. /bin/ls
- Does not do anything!

□ Process

- A program that is executing
- Must exist in RAM before it executes. Why?
- One program can run as multiple processes. What does that mean?

Multiprogramming

□ Multiprogramming

- A system where multiple processes(!) exist at the same time in the RAM
- But only one runs at a time!
 - Because there is only one CPU

□ Multi tasking

- Time sharing between multiple processes in a multi-programming system
- The OS enables this. How? To be seen later.

Question

□ Select the correct one

- 1) A multiprogramming system is not necessarily multitasking
- 2) A multitasking system is not necessarily multiprogramming

Events , that interrupt CPU's functioning

□ Three types of “traps” : Events that make the CPU run code at a pre-defined address

1) Hardware interrupts

2) Software interrupt instructions (trap)

E.g. instruction “int”

3) Exceptions

e.g. a machine instruction that does division by zero

Illegal instruction, etc.

Some are called “faults”, e.g. “page fault”,

Multi tasking requirements

- Two processes should not be
 - Able to steal time of each other
 - See data/code of each other
 - Modify data/code of each other
 - Etc.
- The OS ensures all these things. How?
 - To be seen later.

**But the OS is “always” “running”
“in the background”
Isn’t it?**

Absolutely No!

**Let's understand
What kind of
Hardware, OS interplay
makes
Multitasking possible**

Two types of CPU instructions and two modes of CPU operation

- CPU instructions can be divided into two types

- Normal instructions

- mov, jmp, add, etc.

- Privileged instructions

- Normally related to hardware devices

- E.g.

- IN, OUT # write to I/O memory locations

- INTR # software interrupt, etc.

Two types of CPU instructions and two modes of CPU operation

- CPUs have a mode bit (can be 0 or 1)**
- The two values of the mode bit are called: User mode and Kernel mode**
- If the bit is in user mode**
 - Only the normal instructions can be executed by CPU**
- If the bit is in kernel mode**
 - Both the normal and privileges instructions can be executed by CPU**
- If the CPU is “made” to execute privileged**

Two types of CPU instructions and two modes of CPU operation

- The operating system code runs in kernel mode.**
 - How? Wait for that!**
- The application code runs in user mode**
 - How? Wait !**
 - So application code can not run privileged hardware instructions**
- Transition from user mode to kernel mode and vice-versa**
 - Special instruction called “software interrupt” instructions**

Software interrupt instruction

- E.g. INT on x86 processors

- Does two things at the same time!

- Changes mode from user mode to kernel mode in CPU
- + Jumps to a pre-defined location! Basically changes PC to a pre-defined value.
 - Close to the way a hardware interrupt works. Isn't it?
- Why two things together?
- What's there are the pre-defined location?

Software interrupt instruction

- What's the use of these type of instructions?
 - An application code running INT 0x80 on x86 will now cause
 - Change of mode
 - Jump into OS code
 - Effectively a request by application code to OS to do a particular task!
 - E.g. read from keyboard or write to screen !
 - OS providing hardware services to applications !

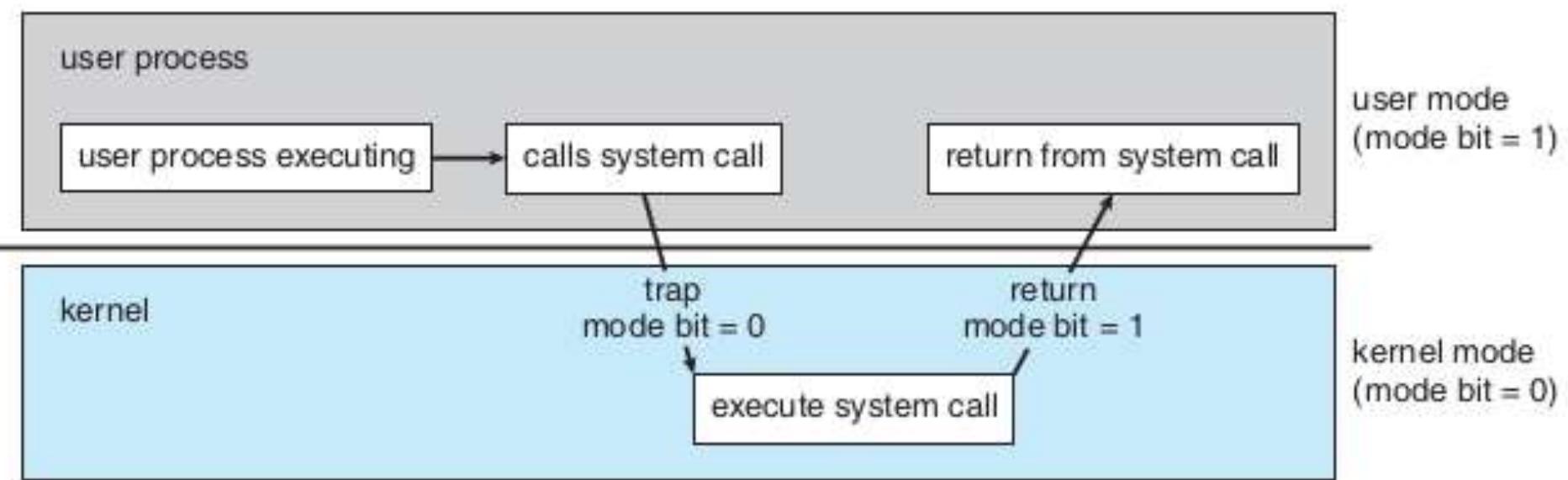


Figure 1.10 Transition from user to kernel mode.

Software interrupt instruction

- How does application code run INT instruction?
 - C library functions like printf(), scanf() which do I/O requests contain the INT instruction!
 - Control flow
 - Application code -> printf -> INT -> OS code -> back to printf code -> back to application code

Example: C program

```
int main() {  
    int i, j, k;  
    k = 20;  
    scanf("%d", &i); // This jumps into OS and  
    returns back  
    j = k + i;  
    printf("%d\n", j); // This jumps into OS and  
    returns back  
    return 0;
```

Interrupt driven OS code

- OS code is sitting in memory , and runs intermittantly . When?
 - On a software or hardware interrupt or exception!
 - Event/Interrupt driven OS!
 - Hardware interrupts and exceptions occur asynchronously (un-predictedly) while software interrupts are caused by application code

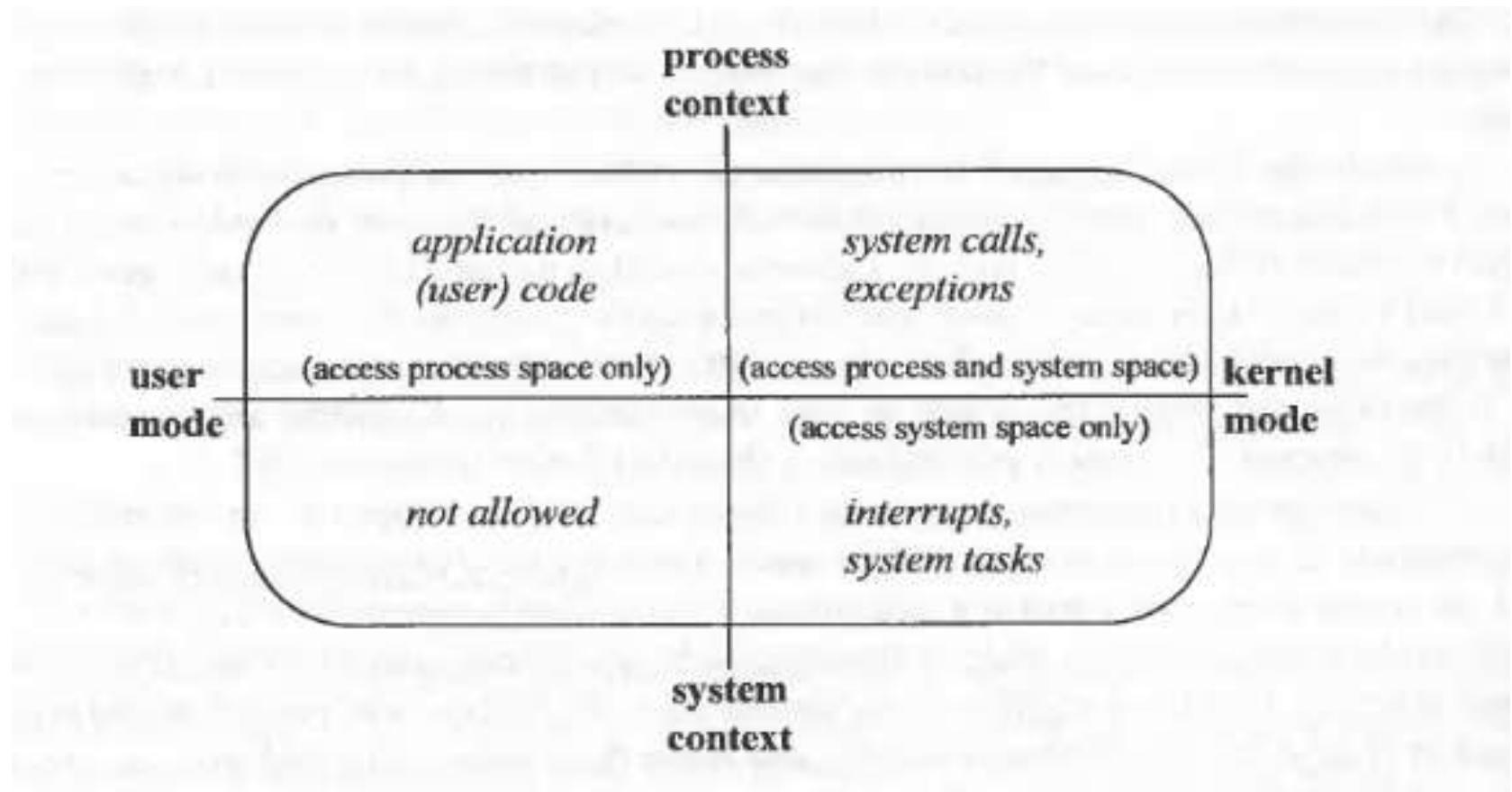
Interrupt driven OS code

□ Timer interrupt and multitasking OS

- Setting timer register is a privileged instruction.
- After setting a value in it, the timer keeps ticking down and on becoming zero the timer interrupt is raised again (in hardware automatically)
- OS sets timer interrupt and “passes control” over to some application code. Now only application code running on CPU !
- When time allocated to process is over, the timer interrupt occurs and control jumps back to OS (hardware interrupt mechanism)

What runs on the processor ?

- 4 possibilities.



System Calls, fork(), exec()

Abhijit A. M.

abhijit.comp@coep.ac.in

(C) Abhijit A.M.

Available under Creative Commons Attribution-ShareAlike License V3.0+

Credits: Slides of “OS Book” ed10.

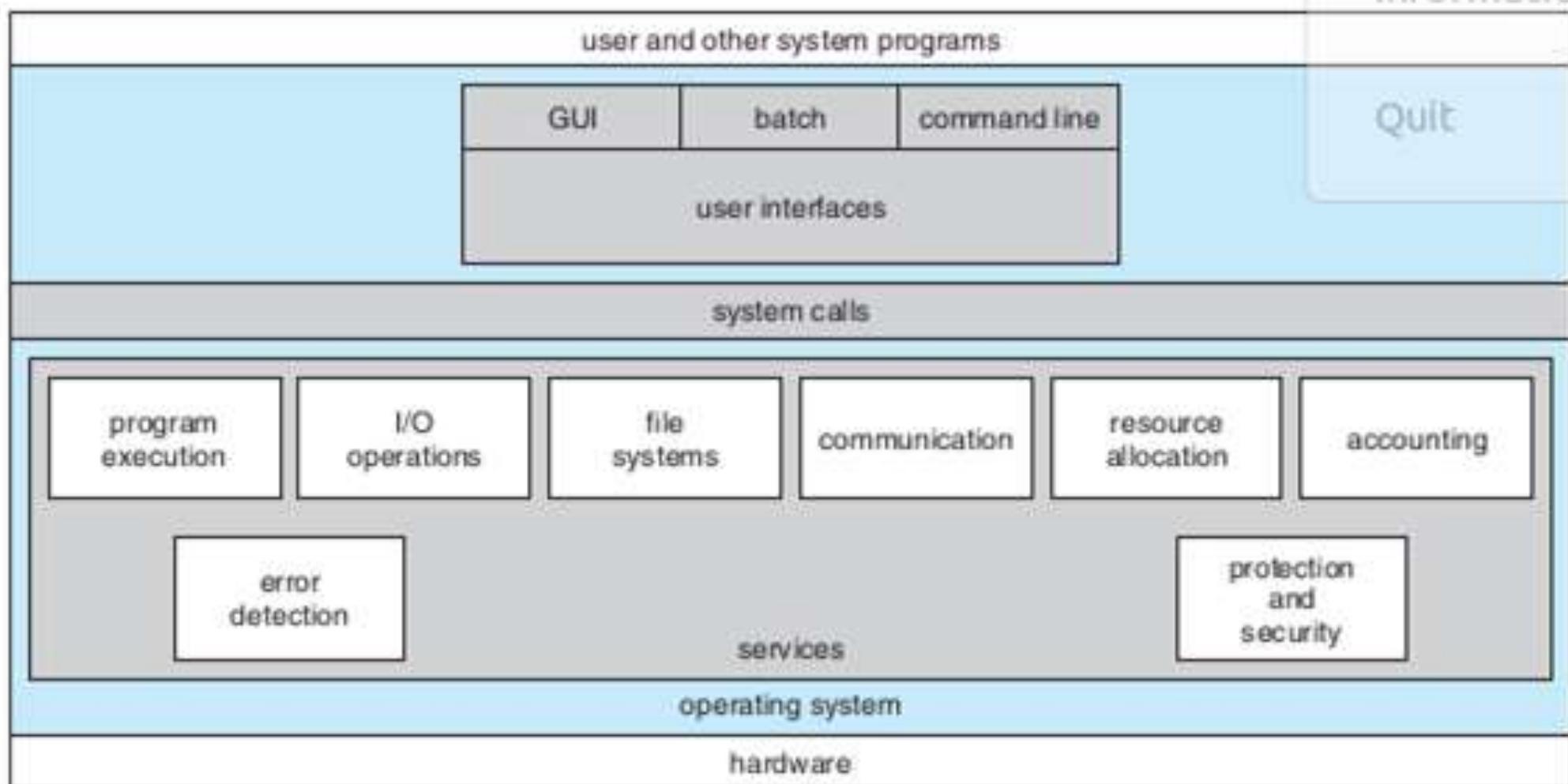


Figure 2.1 A view of operating system services.

System Calls

- Services provided by operating system to applications
 - Essentially available to applications by calling the particular software interrupt application
 - All system calls essentially involve the “INT 0x80” on x86 processors + Linux
 - Different arguments specified in EAX register inform the kernel about different system calls
 - The C library has wrapper functions for each of the system calls

Types of System Calls

□ File System Related

- Open(), read(), write(), close(), etc.

□ Processes Related

- Fork(), exec(), ...

□ Memory management related

- Mmap(), shm_open(), ...

□ Device Management

□ Information maintainance – time,date

https://linuxhint.com/list_of_linux_syscalls/

□ Communication between processes (IPC)

EXAMPLES OF WINDOWS AND UNIX SYSTEM CALLS

	Windows	Unix
Process Control	CreateProcess() ExitProcess() WaitForSingleObject()	fork() exit() wait()
File Manipulation	CreateFile() ReadFile() WriteFile() CloseHandle()	open() read() write() close()
Device Manipulation	SetConsoleMode() ReadConsole() WriteConsole()	ioctl() read() write()
Information Maintenance	GetCurrentProcessID() SetTimer() Sleep()	getpid() alarm() sleep()
Communication	CreatePipe() CreateFileMapping() MapViewOfFile()	pipe() shm_open() mmap()
Protection	SetFileSecurity() InitializeSecurityDescriptor() SetSecurityDescriptorGroup()	chmod() umask() chown()

```
int main() {  
    int a = 2;  
  
    printf("hi\n");  
}  
-----
```

C Library

```
-----
```

```
int printf("void *a, ...){  
    ...  
    write(1, a, ...);  
}
```

Code schematic

-----user-kernel-mode-boundary-----

//OS code

```
int sys_write(int fd, char  
*, int len) {
```

figure out location on
disk

where to do the write
and

carry out the

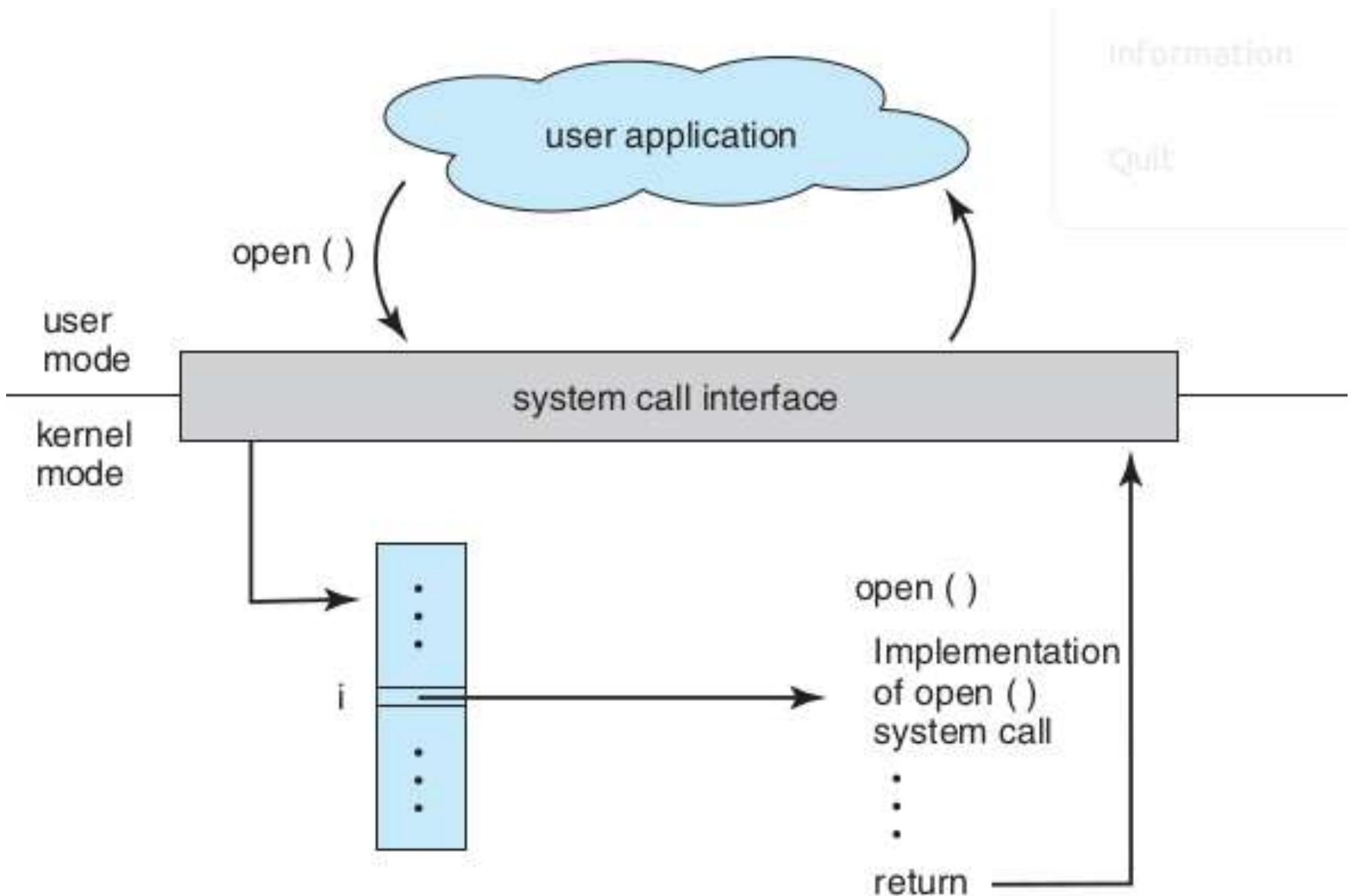


Figure 2.6 The handling of a user application invoking the `open()` system call.

Two important system calls

Related to processes

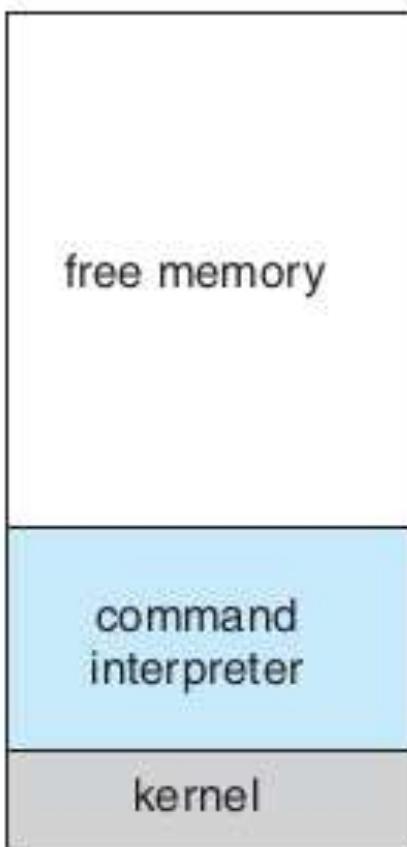
fork() and exec()

Process

- A program in execution
- Exists in RAM
- Scheduled by OS
 - In a timesharing system, intermittantly scheduled by allocating a time quantum, e.g. 20 microseconds
- The “ps” command on Linux

Process in RAM

- Memory is required to store the following components of a process
 - Code
 - Global variables (data)
 - Stack (stores local variables of functions)
 - Heap (stores malloced memory)
 - Shared libraries (e.g. code of printf, etc)
 - Few other things, may be



(a)

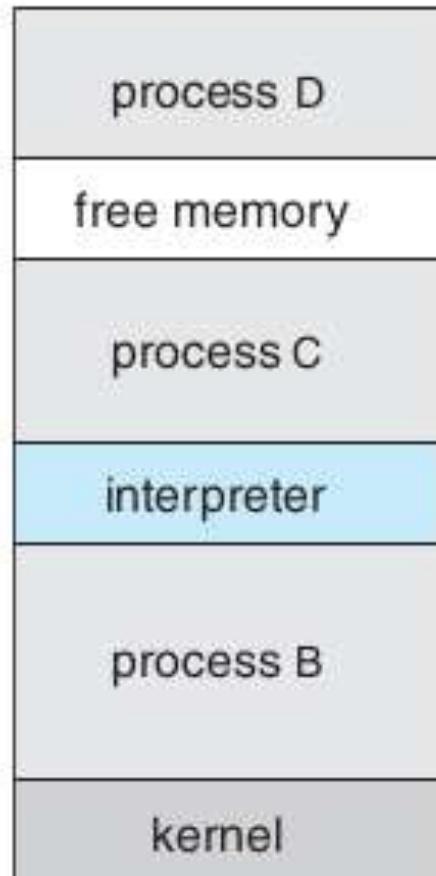


(b)

Figure 2.9 MS-DOS execution. (a) At system startup. (b) Running a program.

MS-DOS: a single tasking operating system

Only one program in RAM at a time, and only one program can run at a time



A multi tasking system
With multiple programs loaded in memory
Along with kernel

(A very simplified conceptual diagram. Things are more complex in reality)

fork()

- A running process creates it's duplicate!
- After call to fork() is over
 - Two processes are running
 - Identical
 - The calling function returns in two places!
 - Caller is called parent, and the new process is called child
 - PID is returned to parent and 0 to child

exec()

- Variants: execvp(), execl(), etc.
- Takes the path name of an executable as an argument
- Overwrites the existing process using the code provided in the executable
- The original process is OVER ! Vanished!
- The new program starts running overwritting the existing process!

Shell using fork and exec

□ Demo

- The only way a process can be created on Unix/Linux is using `fork()` + `exec()`
- All processes that you see were started by some other process using `fork()` + `exec()` , except the initial “*init*” process
- *When you click on “firefox” icon, the user-interface program does a `fork()` + `exec()` to start firefox; same with a command line shell program*
- The “bash” shell you have been using is nothing but a command line interface application.

The boot process, once again

- BIOS

- Boot loader

- OS – kernel

- Init created by kernel by Hand(kernel mode)

- Kernel schedules init (the only process)

- Init fork-execs some programs (user mode)

- Now these programs will be scheduled by OS

- Init -> GUI -> terminal -> shell

- One of the typical parent-child relationships

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]) {
```

Function f() may not be called

```
    int i, j;
```

```
    for(i = 0; i < 1024; i++)
```

```
        m[0][i] = 200;
```

```
}
```

```
int main() {
```

```
    int i, j;
```

```
    for(i = 0; i < 1024; i++)
```

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

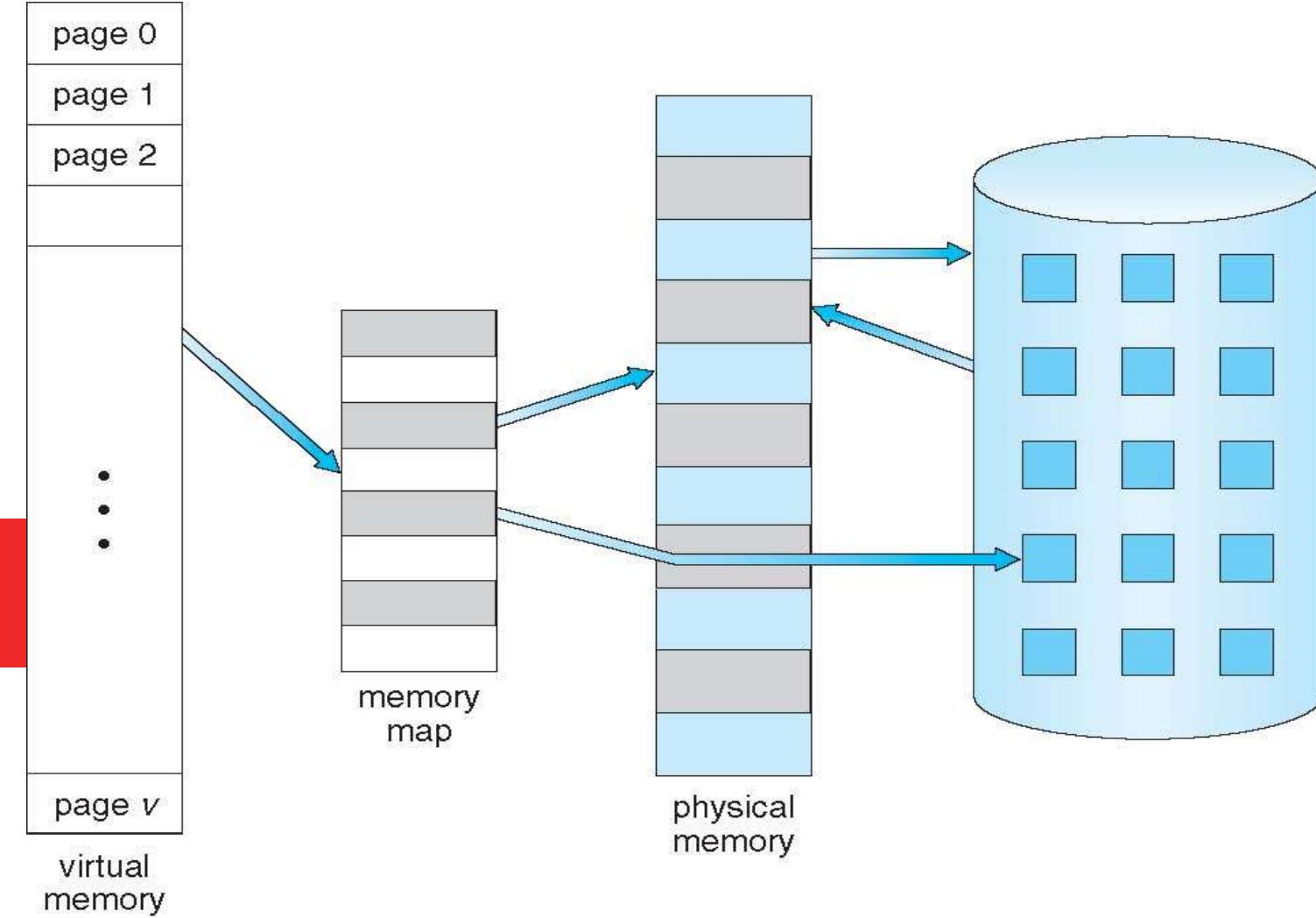
- Less I/O needed to load or swap processes

- Virtual memory can be implemented via:

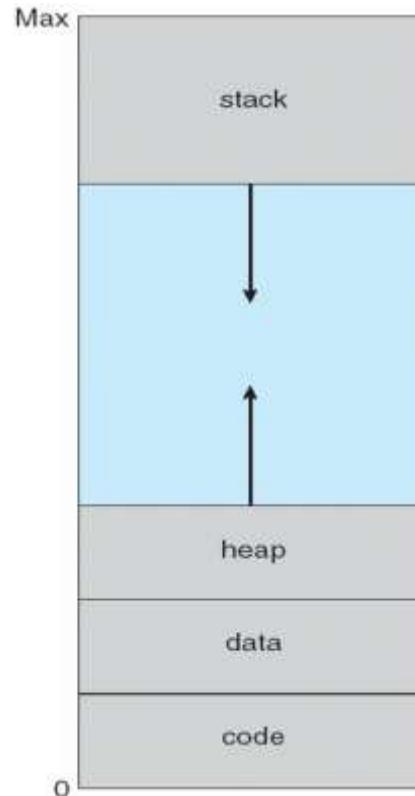
- Demand paging

- Demand segmentation

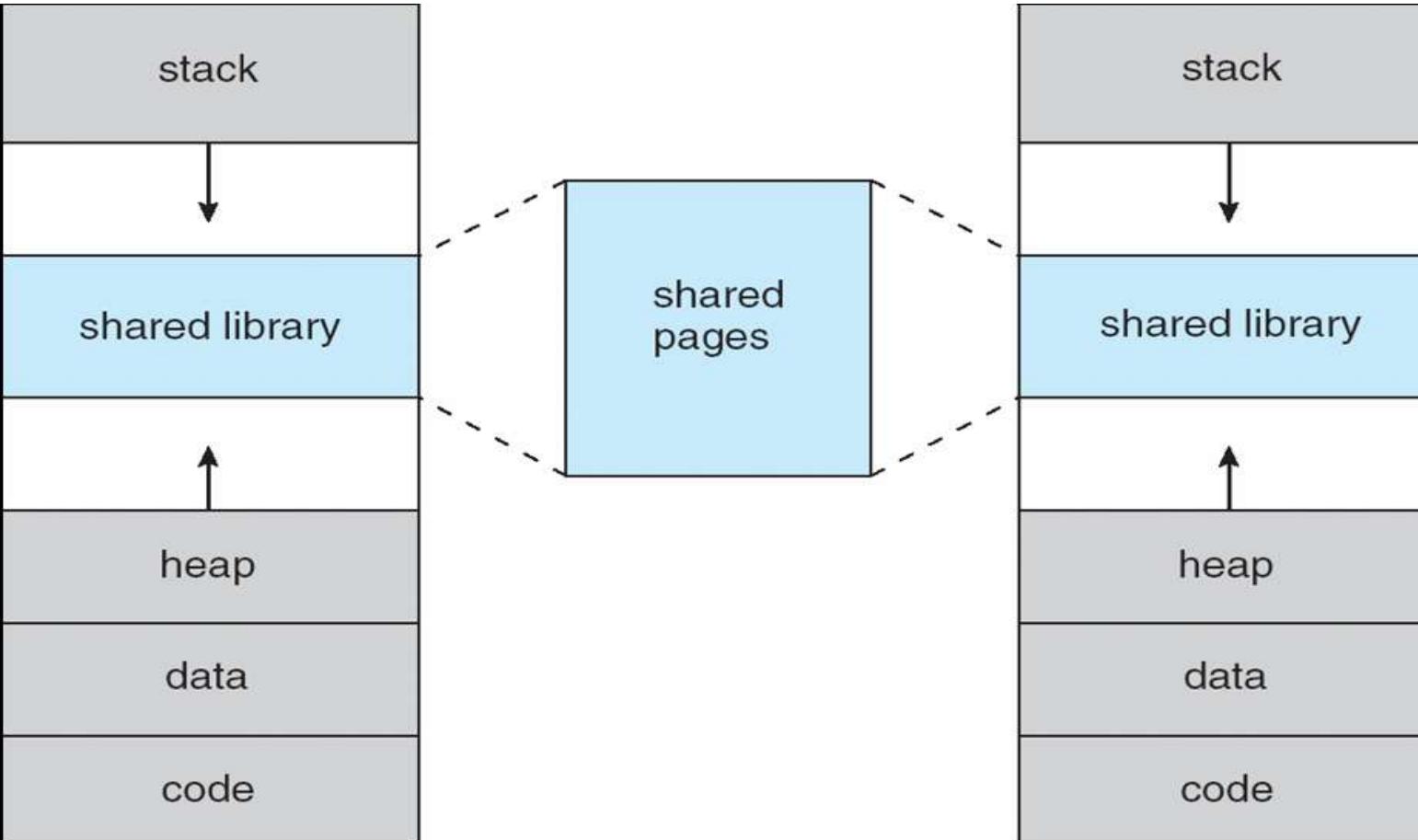
Virtual Memory
Larger
Than
Physical
Memory



Virtual Address space



Enables **sparse** address spaces with holes



Shared pages Using Virtual Memory

System libraries shared
Shared memory by mapping
Pages can be shared during



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:

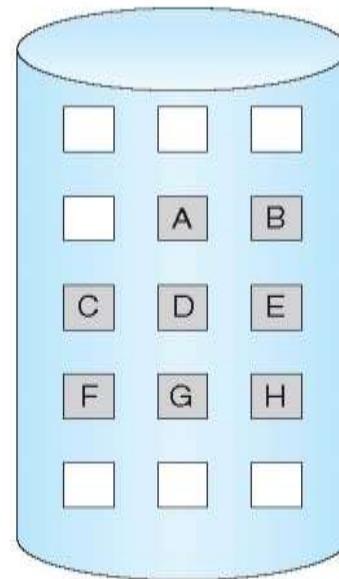
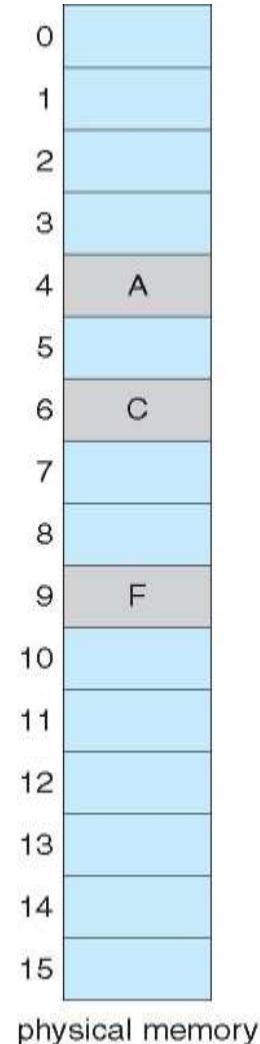
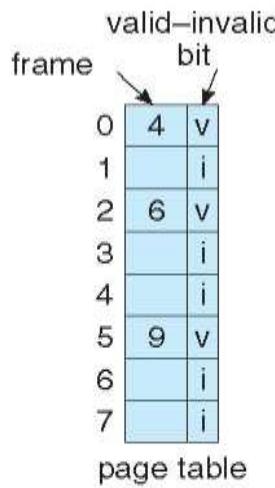
Load entire process at load time : achieves little

Load some pages at load time: good

Load no pages at load time: pure demand paging

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

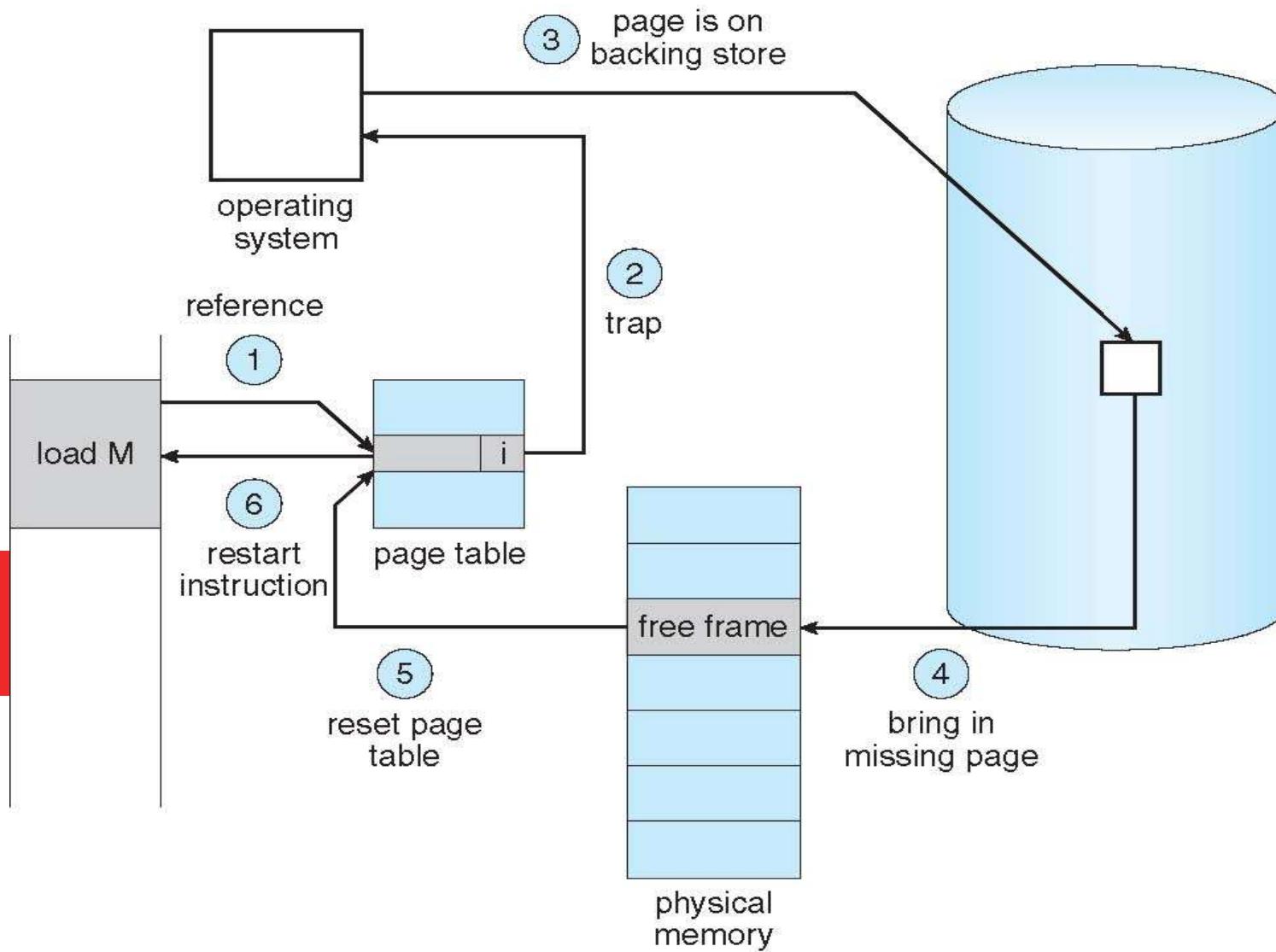
And for every other process pages on first access

Pure demand paging

- .Actually, a given instruction could access multiple pages -> multiple page faults

Pain decreased because of **locality of reference**

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

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

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

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

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

$$p < .0000025$$

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) {
```

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

An optimization: Copy on write

.Copy-on-Write (COW) allows both parent and child processes to initially share the same pages in memory

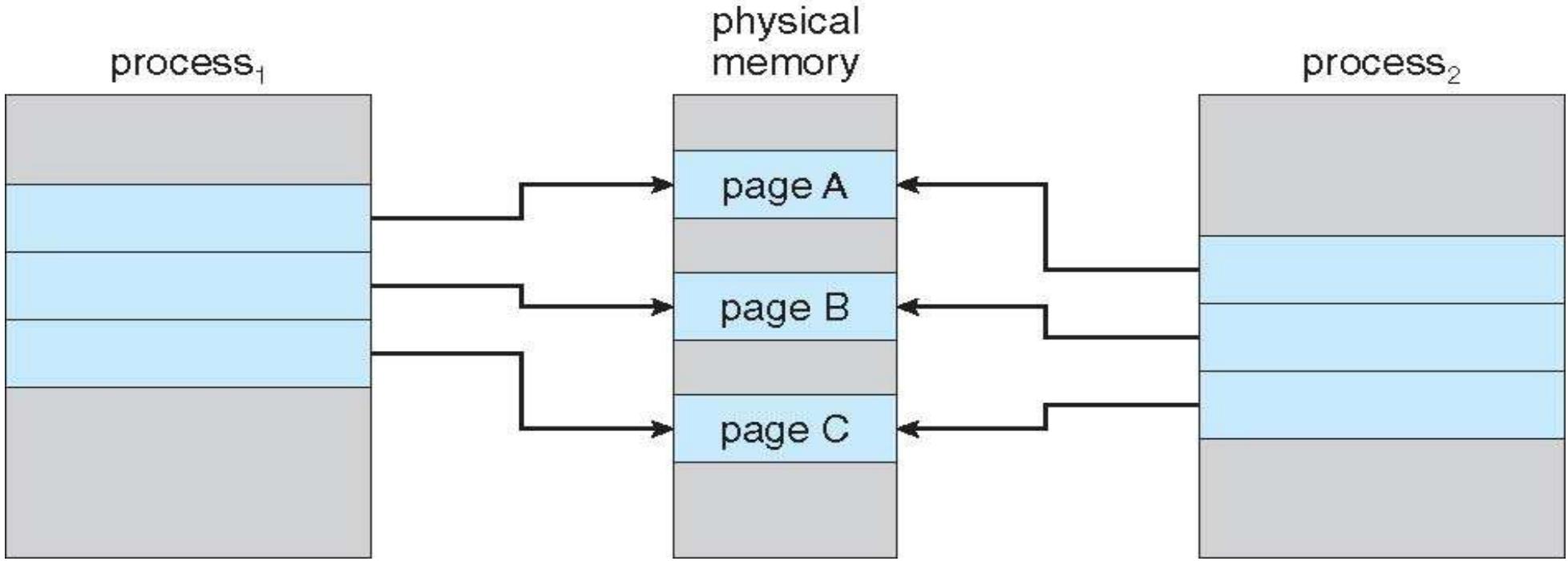
If either process modifies a shared page, only then is the page copied

.COW allows more efficient process creation as only modified pages are copied

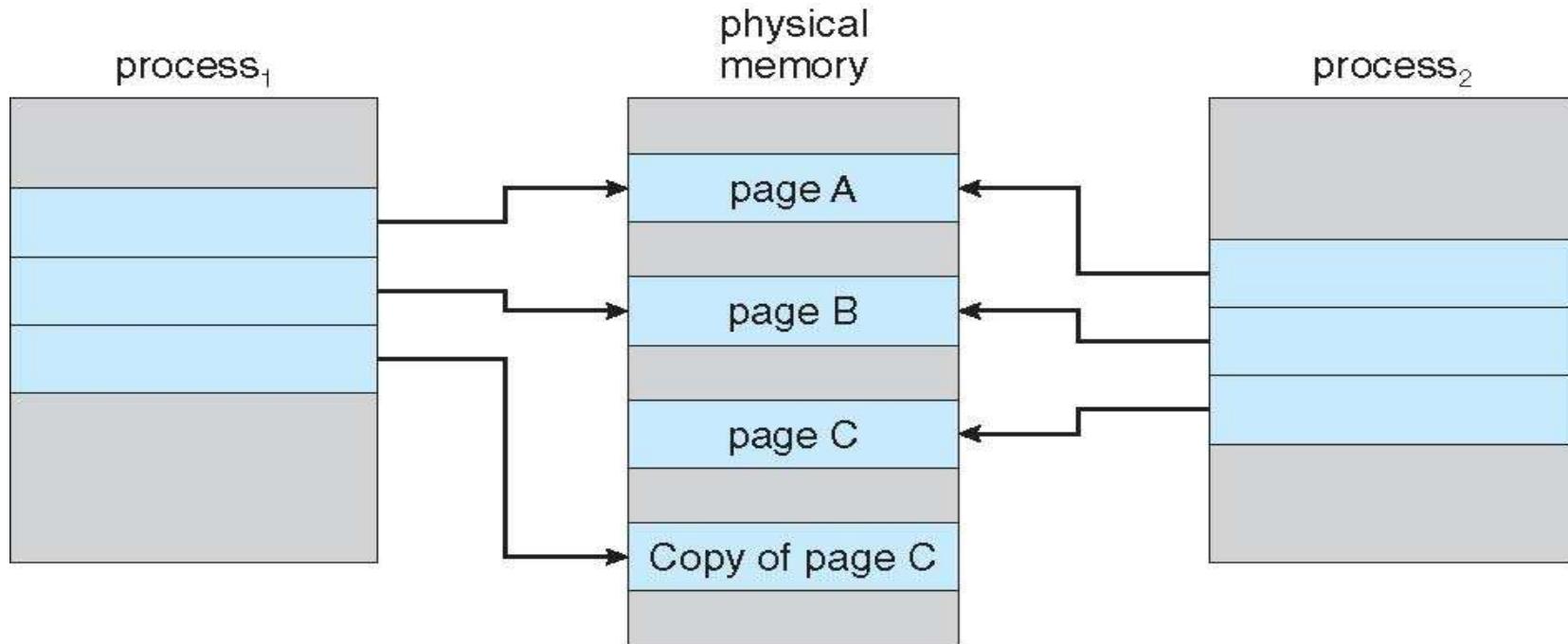
.vfork() variation on fork() system call has parent suspend and child using copy-on-write address space of parent

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 pagable?

- 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



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

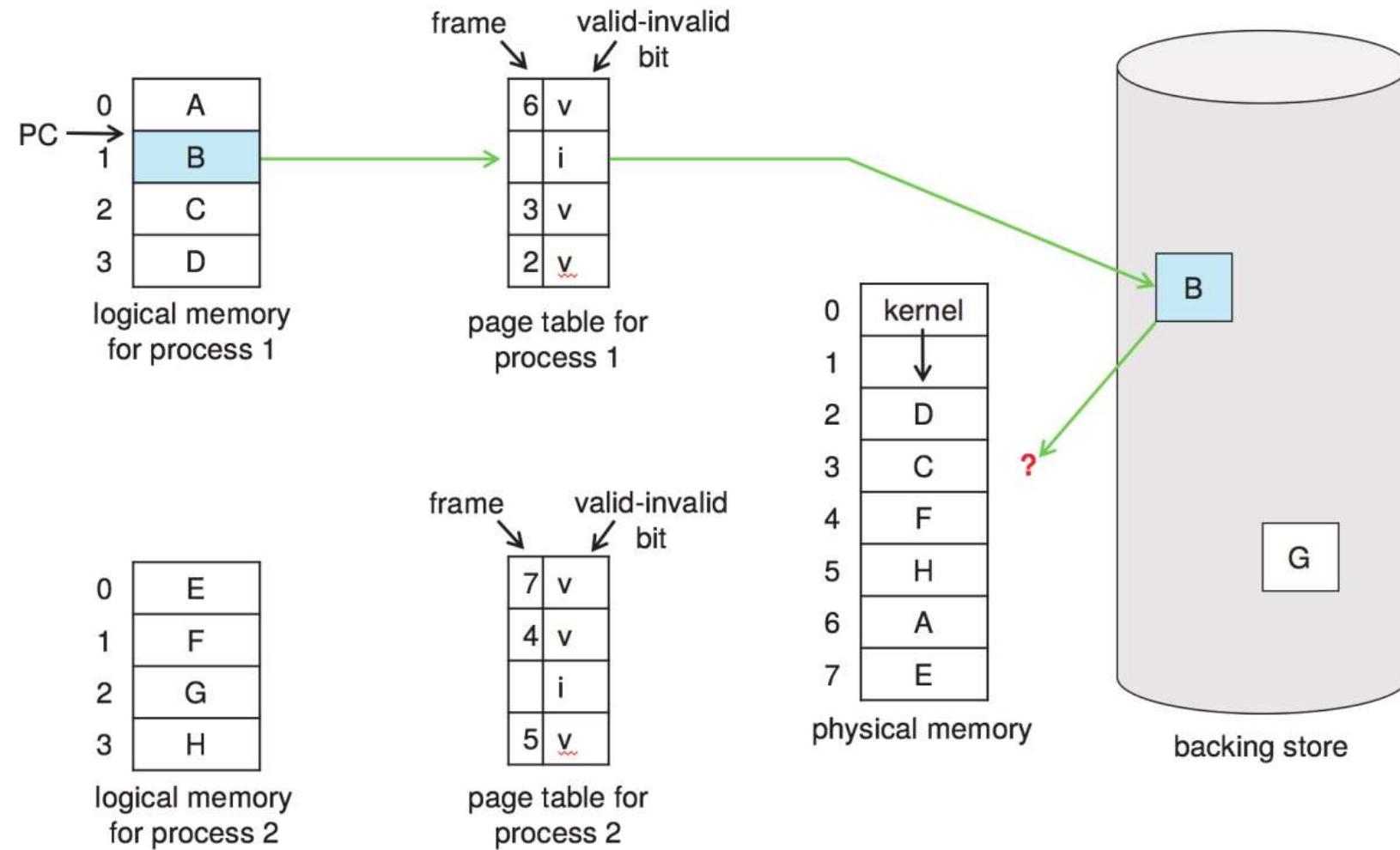


Figure 10.9 Need for page replacement.

Page replacement

.Strategies for performance

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

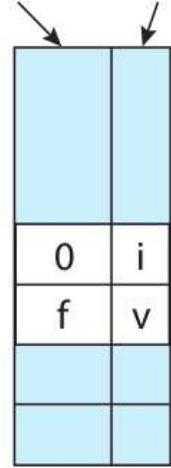
Use modify (dirty) bit in page table. To reduce overhead of page transfers – only modified pages are written to disk. If page is not modified, just reuse it (a copy is already there in backing store)

Basic Page replacement

- 1) Find the location of the desired page on disk
 - 2) Find a free frame:
 - 3) - If there is a free frame, use it
 - 4) - If there is no free frame, use a page replacement algorithm to select a victim frame & write victim frame to disk if dirty
 - 5) Bring the desired page into the free frame; update the page table of process and global frame table/list
 - 6) 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

f

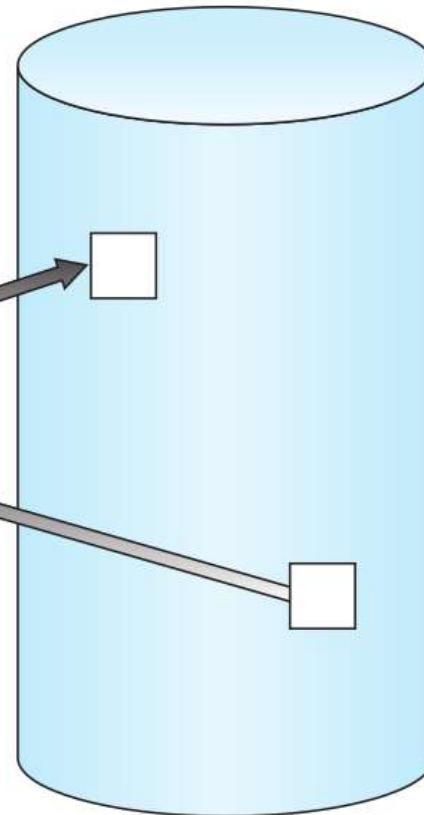
physical
memory

victim

page out
victim
page

1

3 page in
desired
page



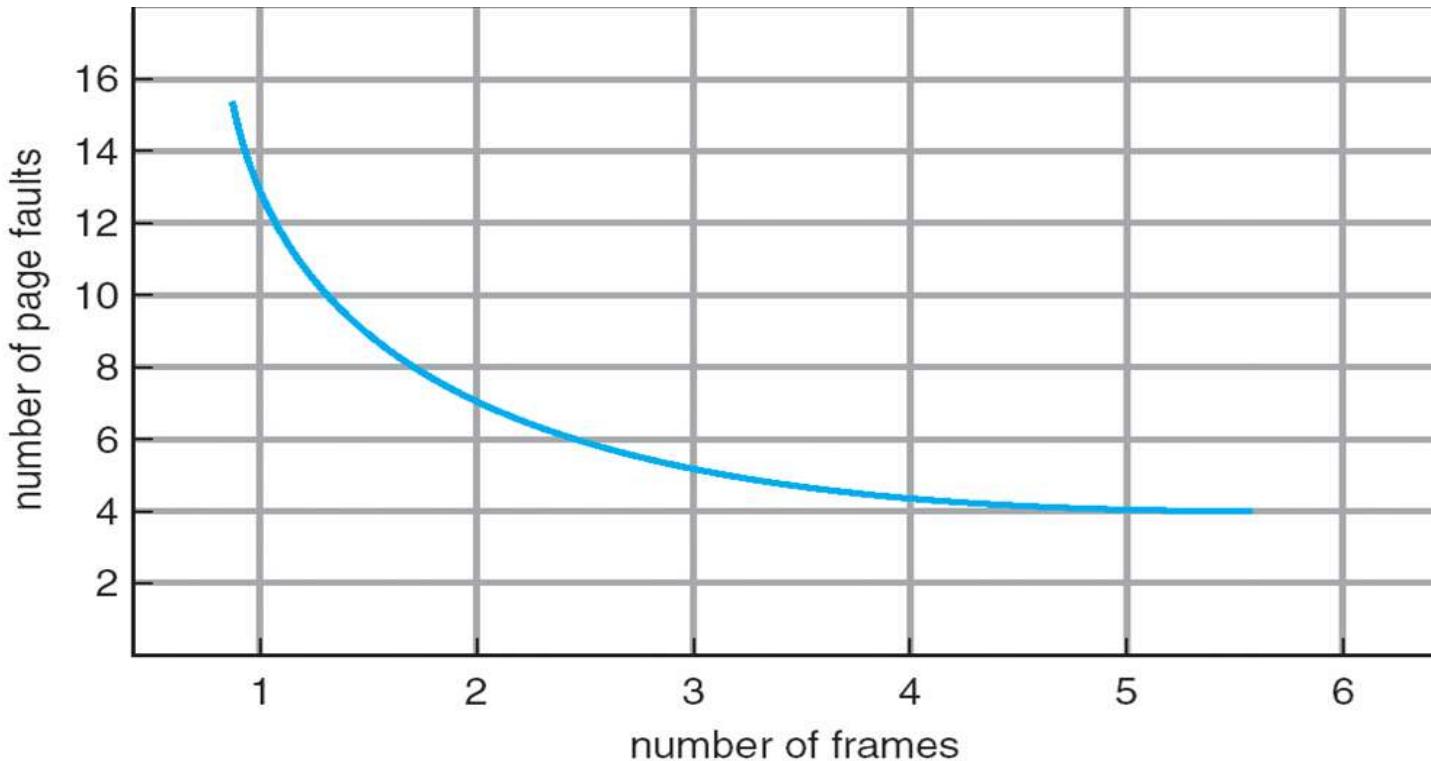
backing store

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



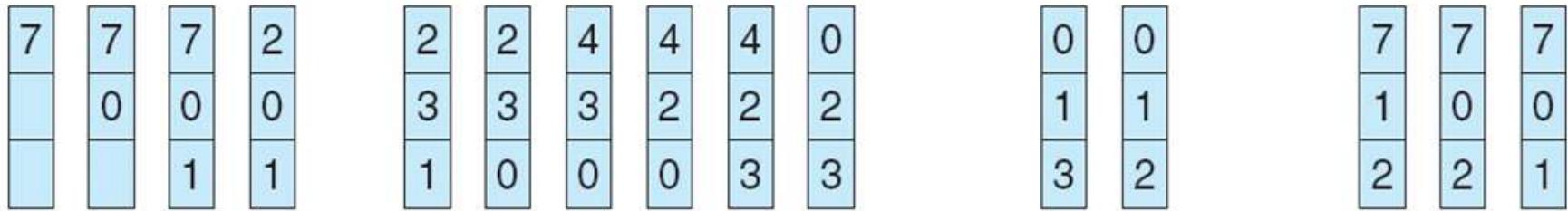
An
Expectation

More page
Frames
Means less
faults

FIFO Algorithm

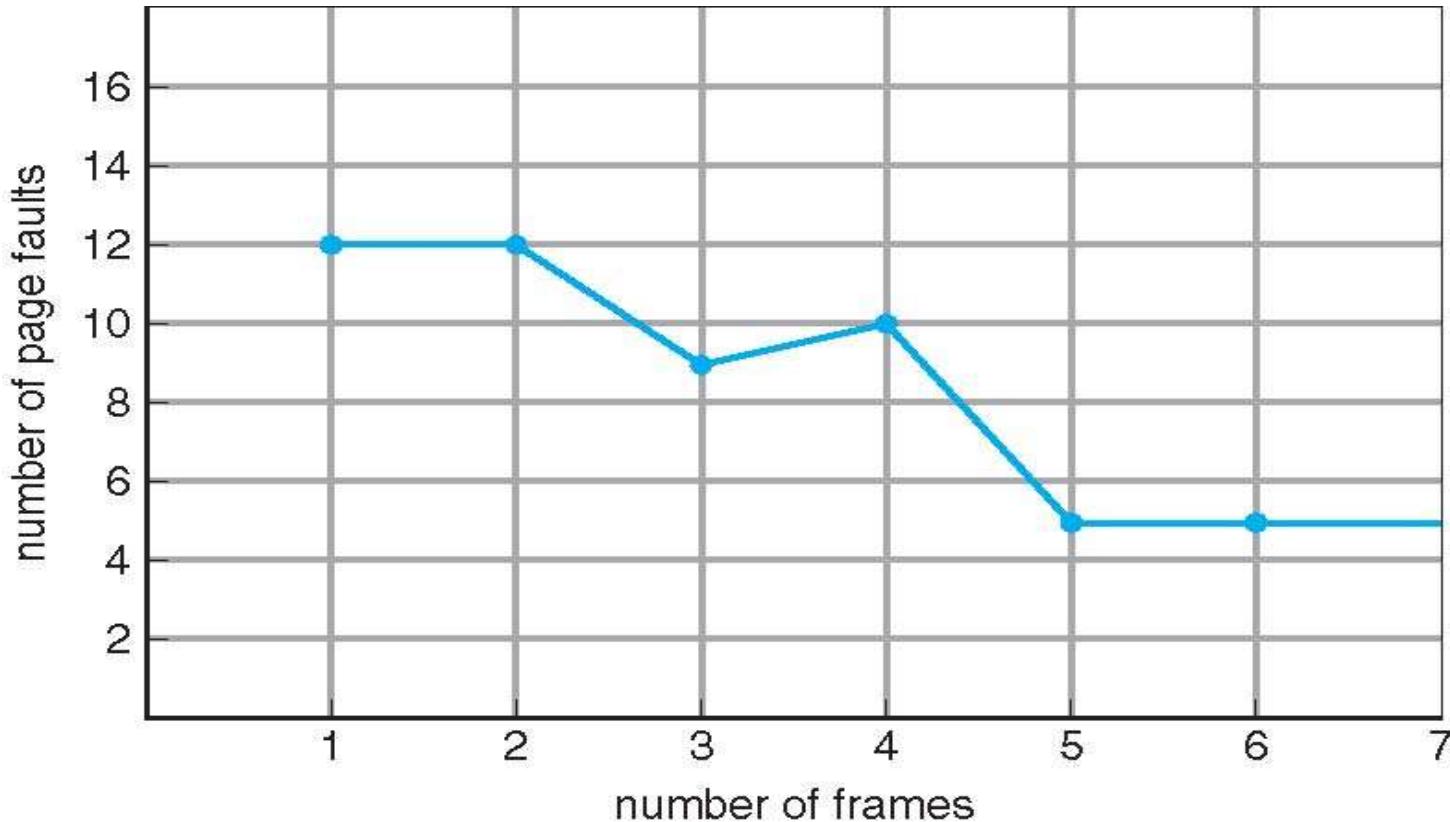
reference string

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



page frames

FIFO Algorithm: Balady's anamoly



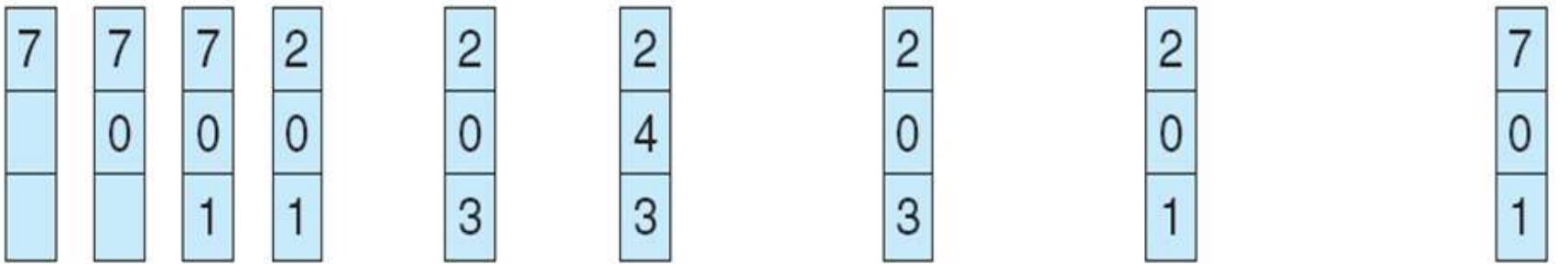
Optimal Algorithm

- 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

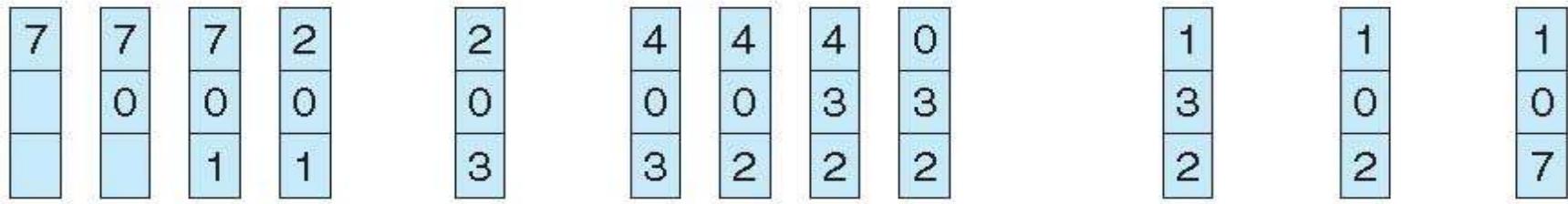


page frames

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 Most Recent

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

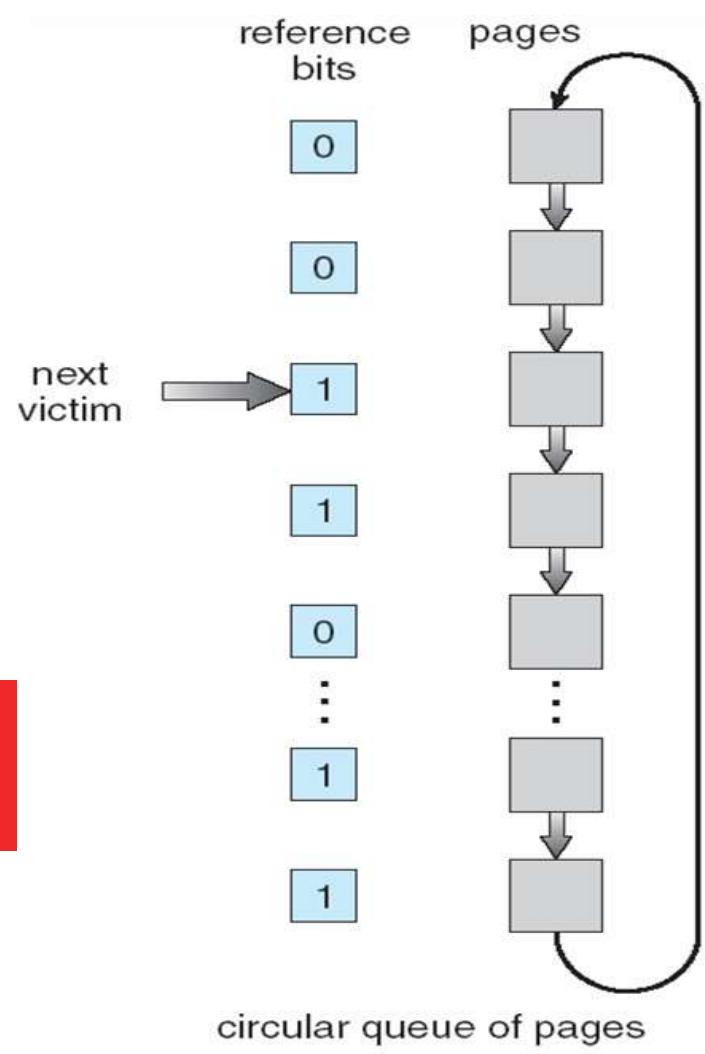
. Reference bit

- . 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

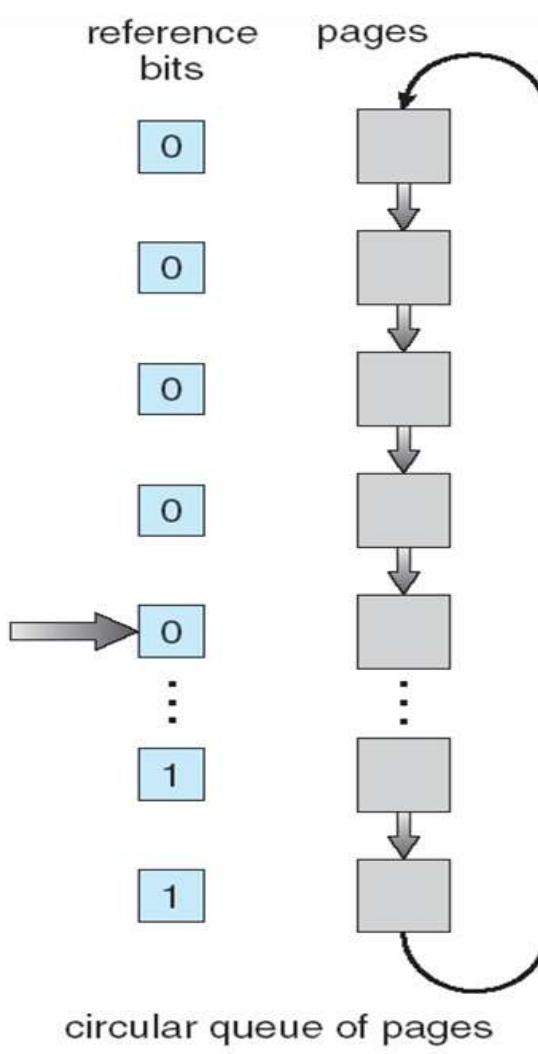
LRU: Approximation algorithms

- .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

clock) Page-Repla



(a)



(b)

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

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

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

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

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

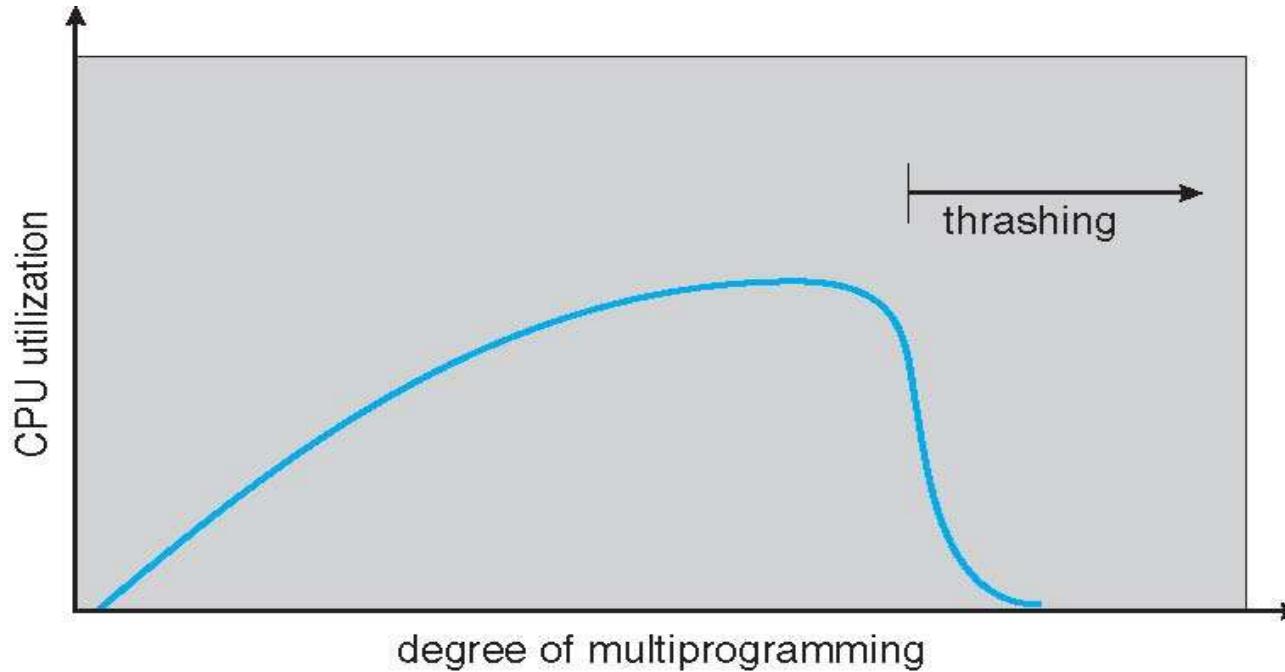


Thrashing

Thrashing

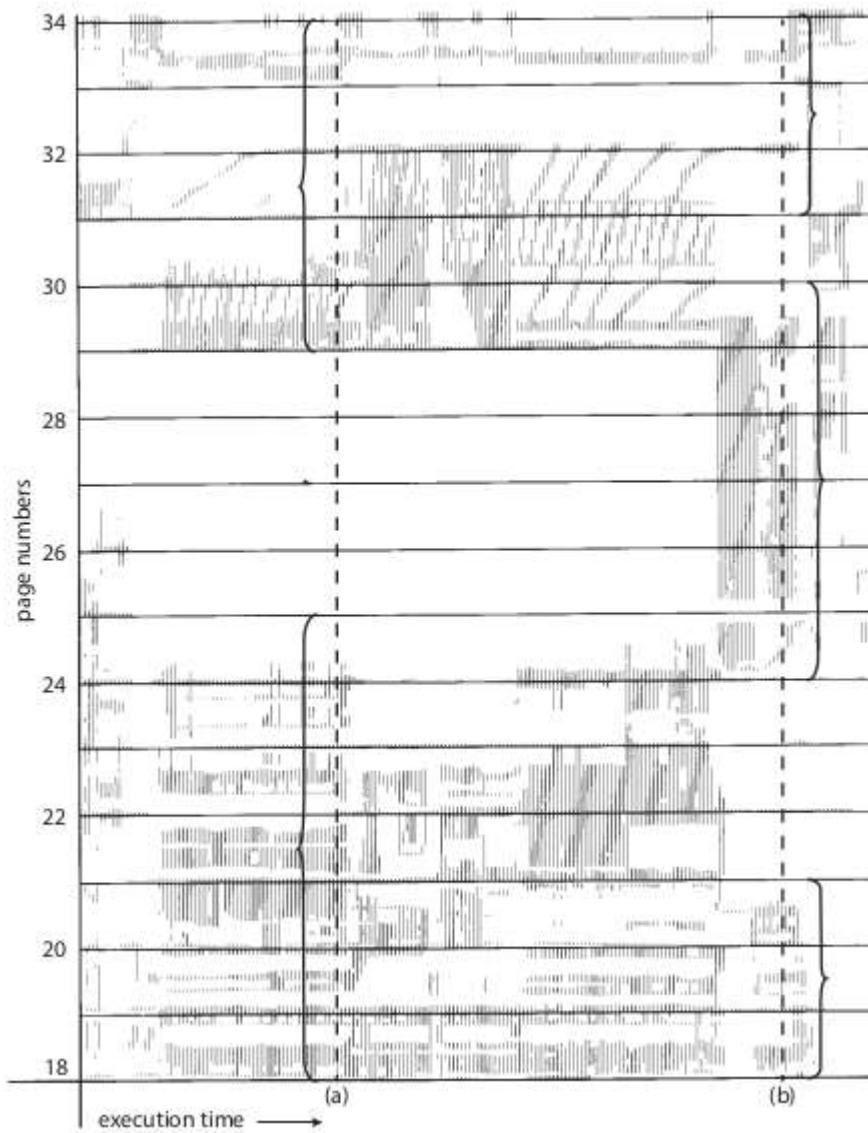
- If a process does not have “enough” pages, the page-fault rate is very high
- Page fault to get page
- Replace existing frame
- But quickly need replaced frame back
- This leads to:
 - Low CPU utilization
 - Operating system thinking that it needs to increase the degree of multiprogramming

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-Refer

Working set model

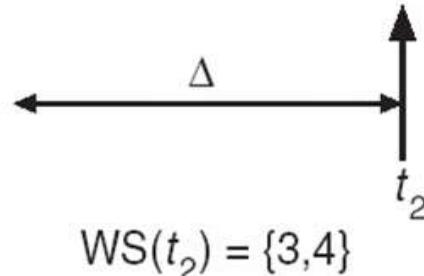
- $\Delta \equiv$ working-set window \equiv a fixed number of page references
- Example: 10,000 instructions
- Working Set Size, WSS_i (working set of Process P_i) =
 - total number of pages referenced in the most recent Δ (varies in time)
 - if Δ too small will not encompass entire locality
 - if Δ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

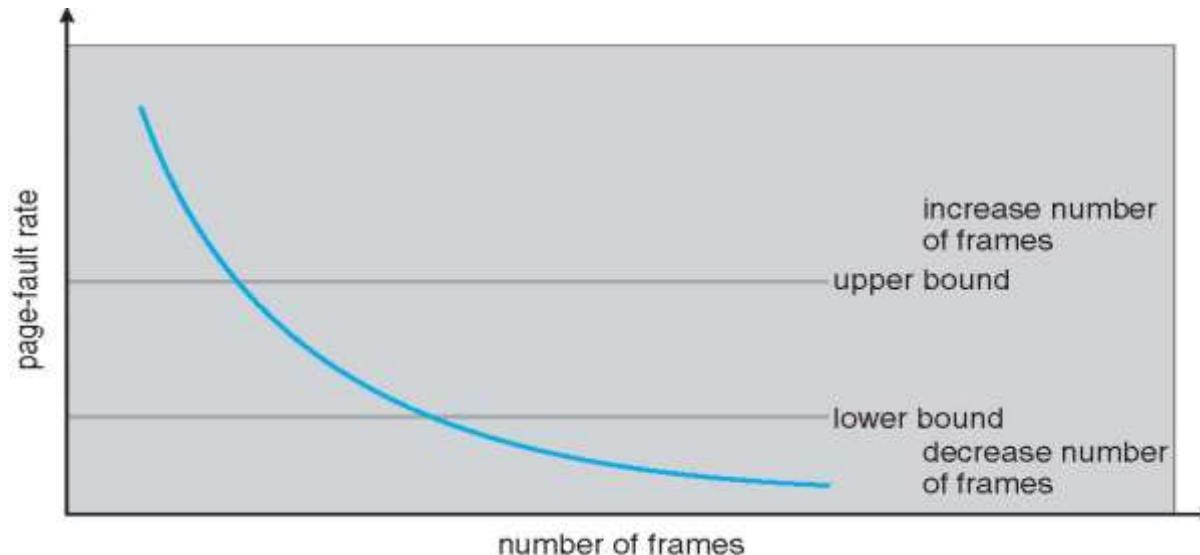
- Why is this not completely accurate?
- Improvement = 10 bits and interrupt every 1000 time units

Page fault frequency

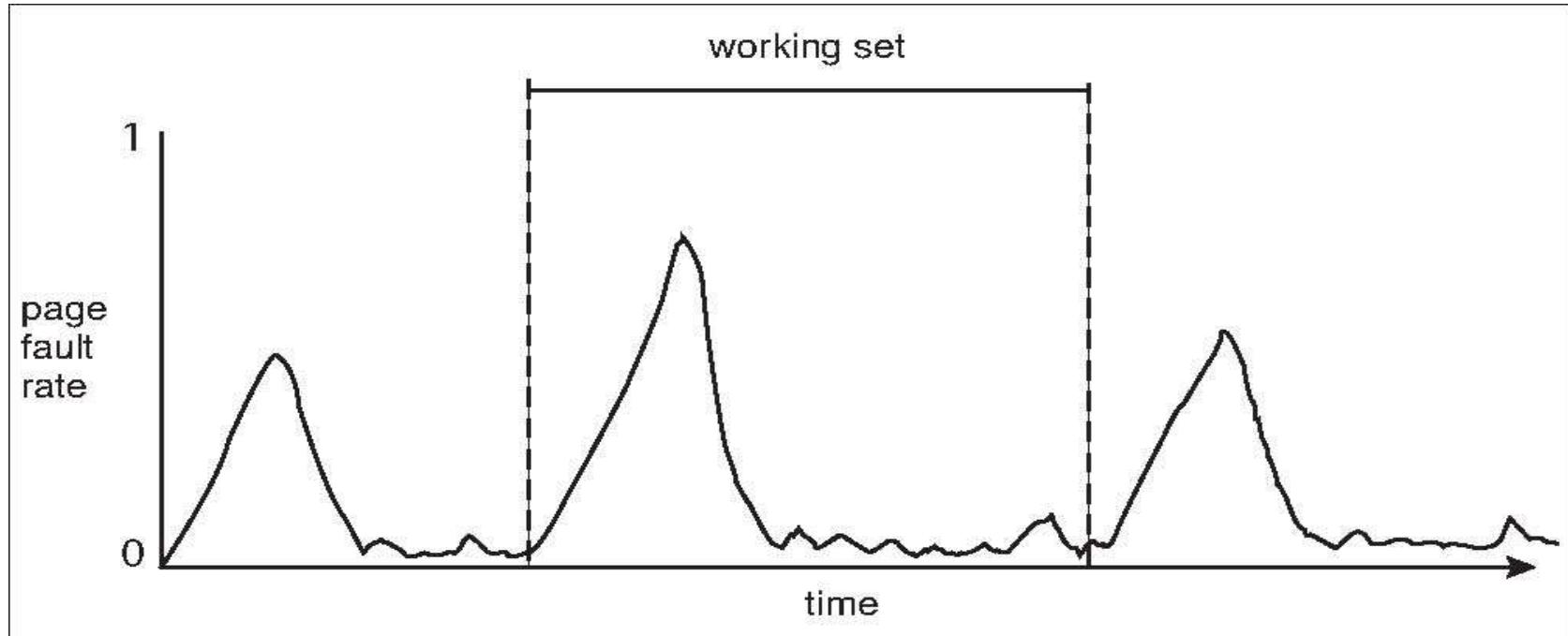
- More direct approach than WSS
- Establish “acceptable” page-fault frequency rate and use local replacement policy

If actual rate too low, process loses frame

If actual rate too high, process gains frame



Working Sets and Page F



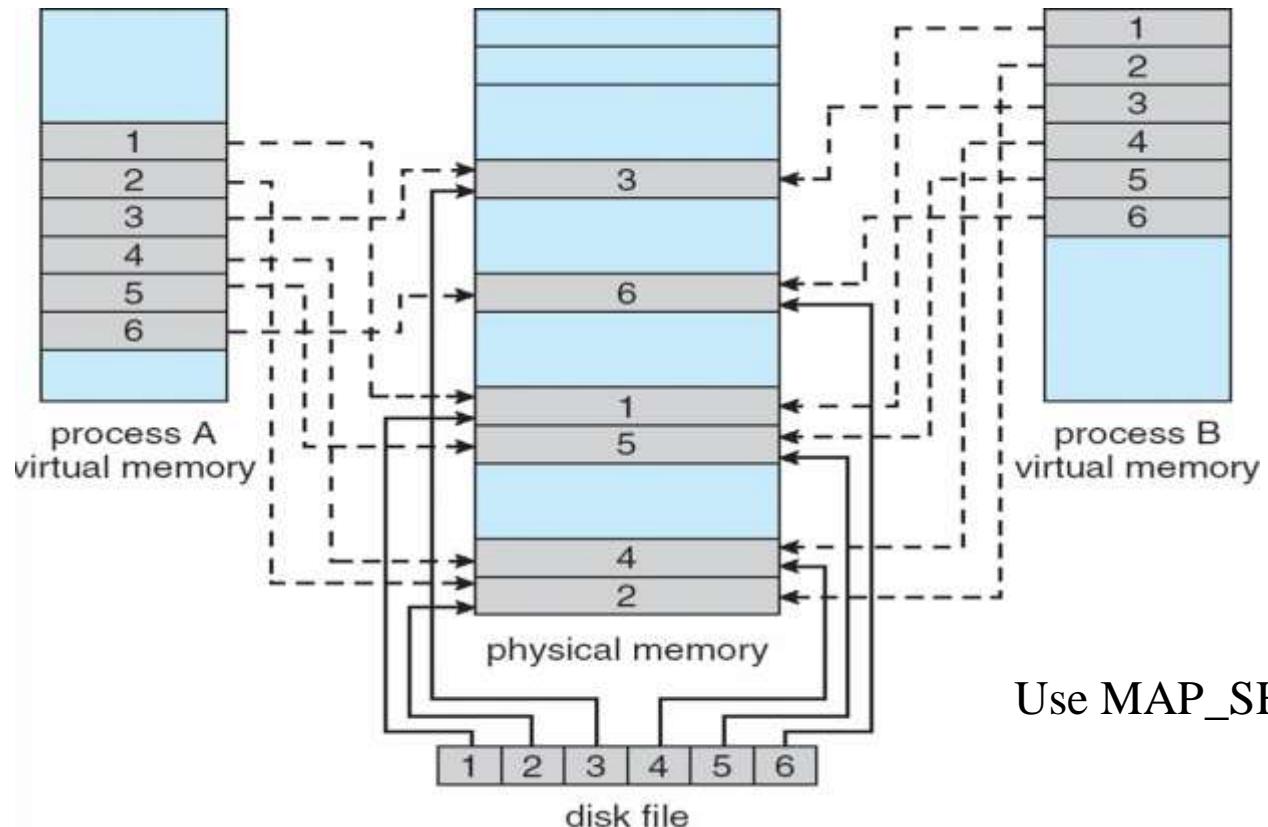


Memory Mapped Files

Memory-Mapped Files

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

Memory-Mapped Files



Use `MAP_SHARED` flag

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

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



Allocating Kernel Memory

Allocating kernel memory

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

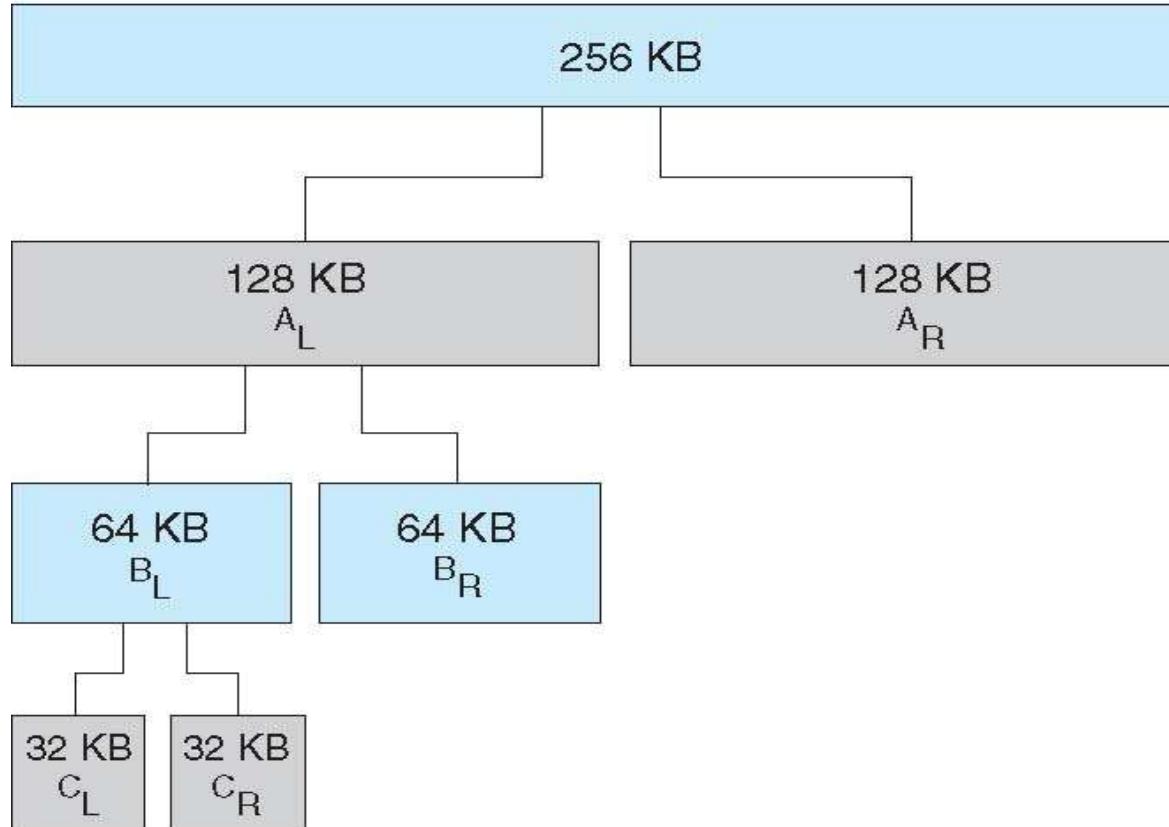
Kernel requests memory for structures of varying sizes

Some kernel memory needs to be contiguous

i.e. for device I/O

Buddy Allocator

physically contiguous pages



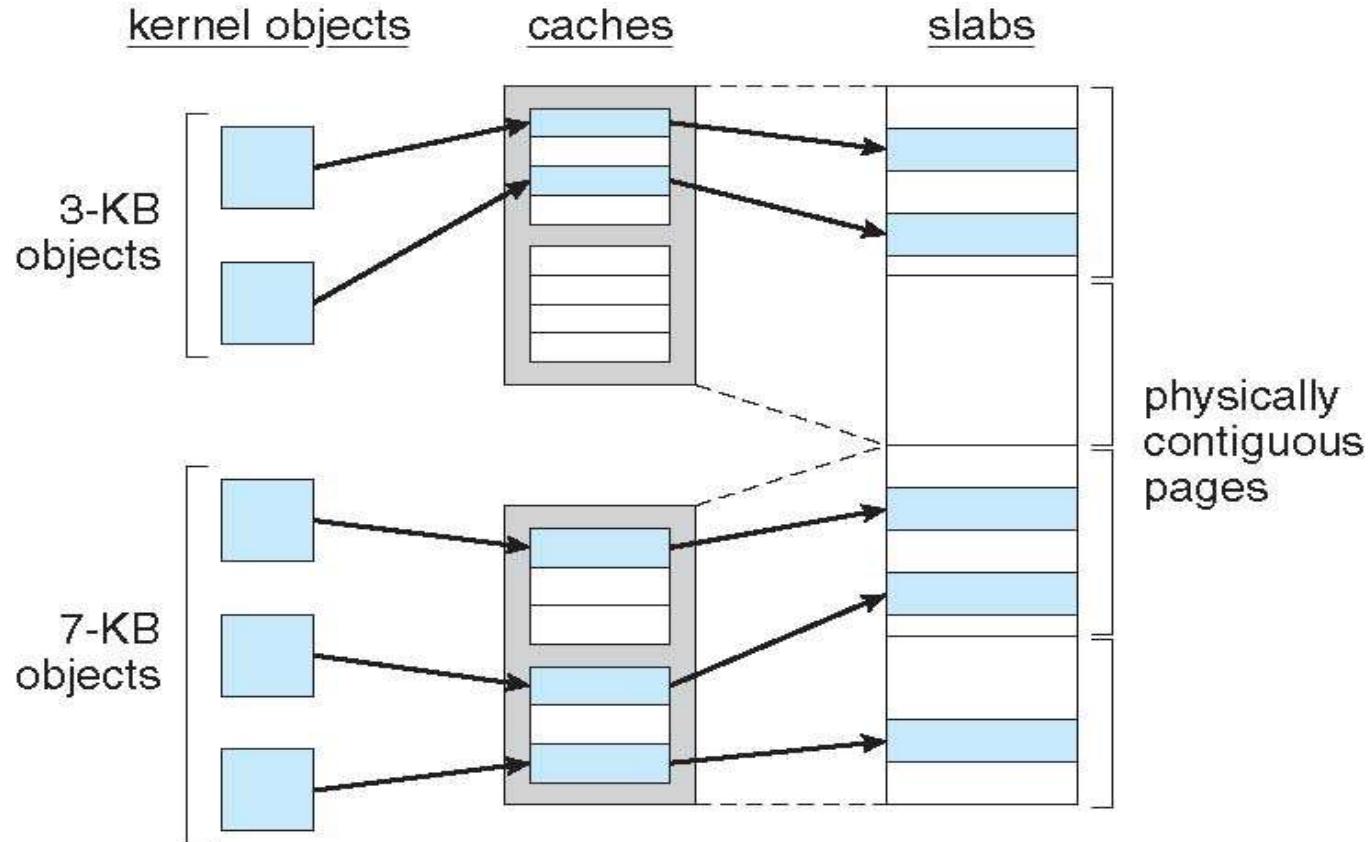
Buddy Allocator

- 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



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?

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

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;
```

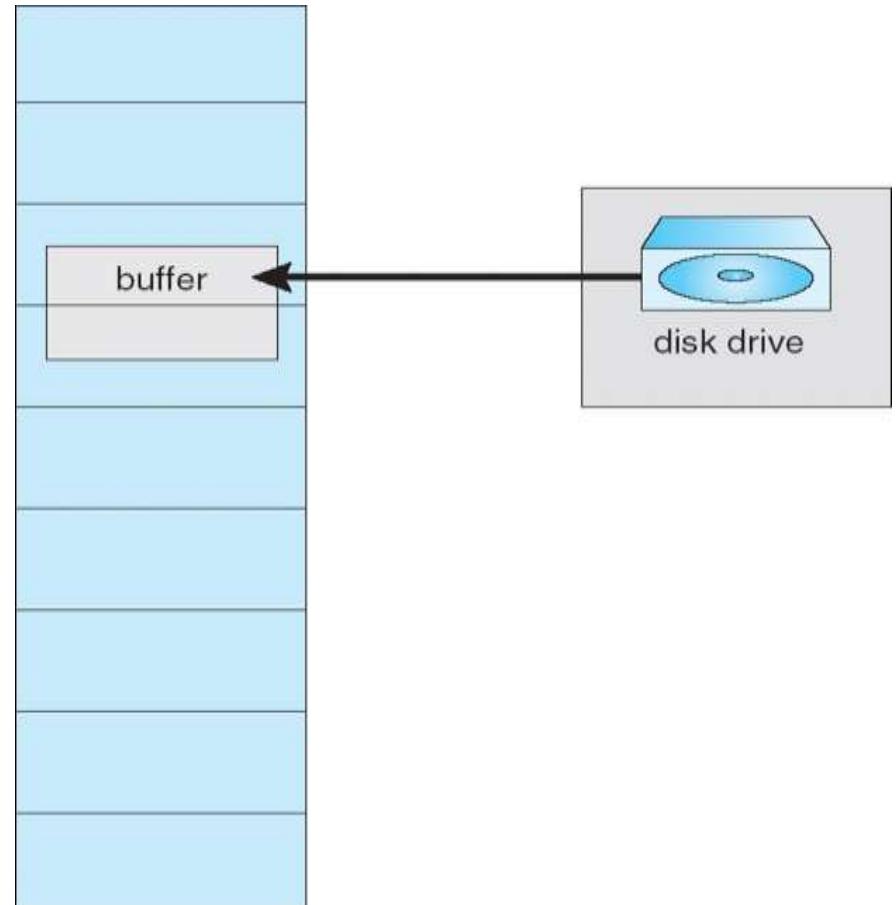
128 x 128 = 16,384 page faults

.Program 2

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

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

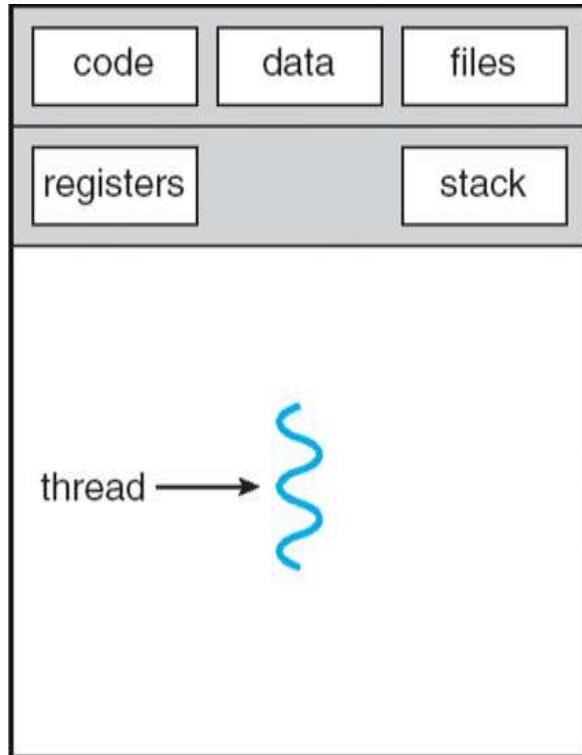
Threads

- **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**

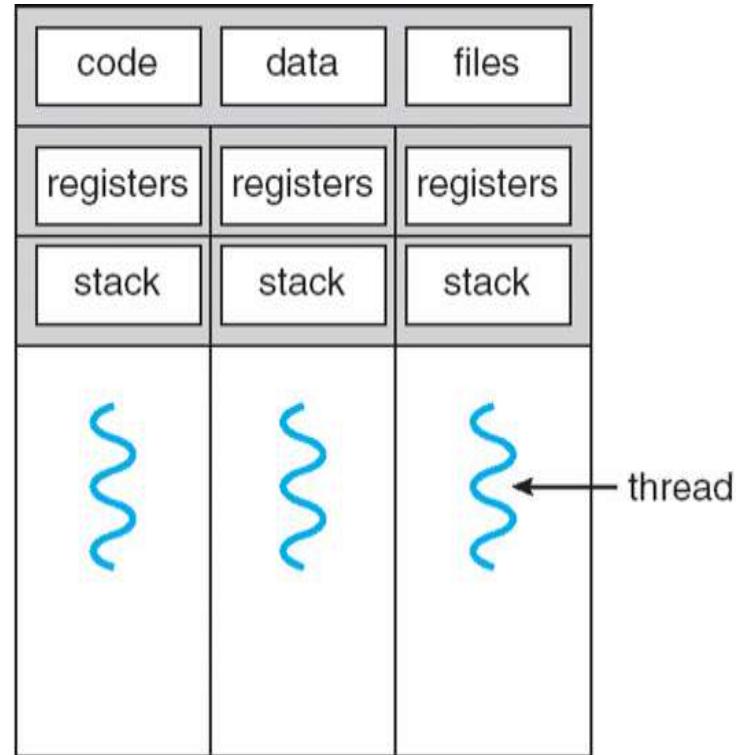
Threads

- 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

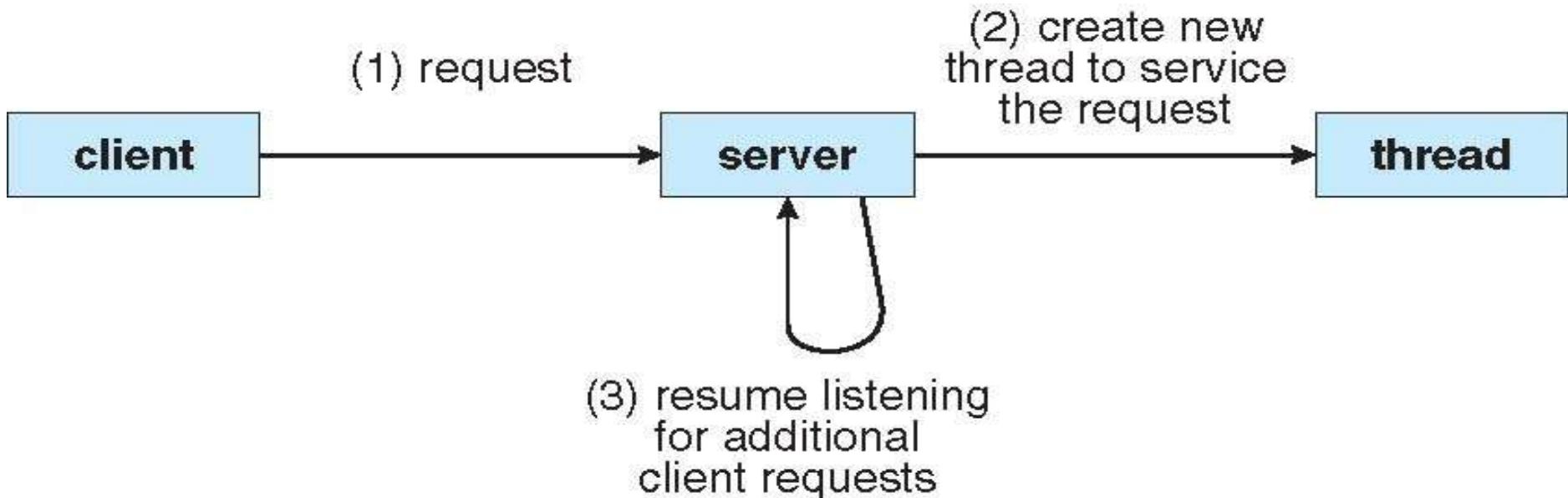


single-threaded process



multithreaded process

A multithreaded server

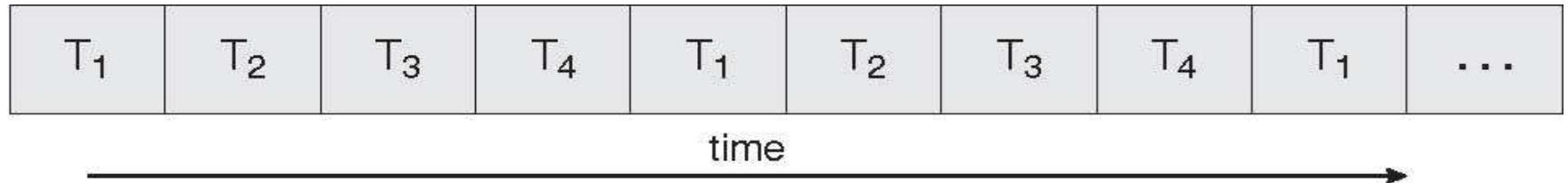


Benefits of threads

- 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

User threads vs Kernel Threads

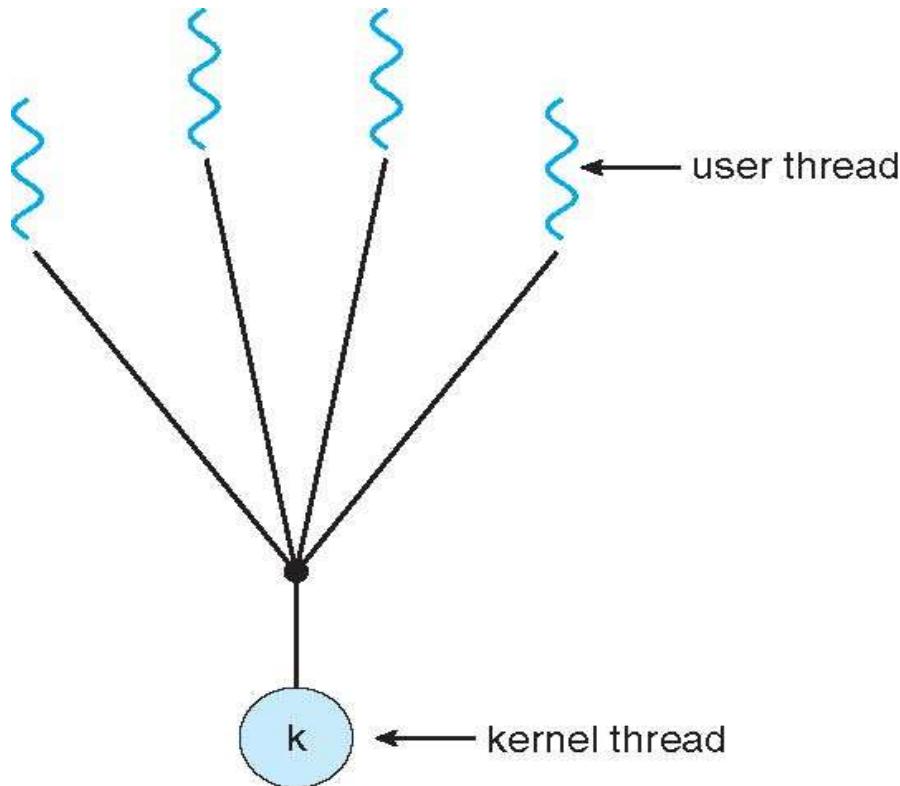
- User threads
- User level library provides a “typedef” called threads
- The scheduling of threads needs to be implemented in the user level library
- Need some type of timer

- 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

Multithreading models

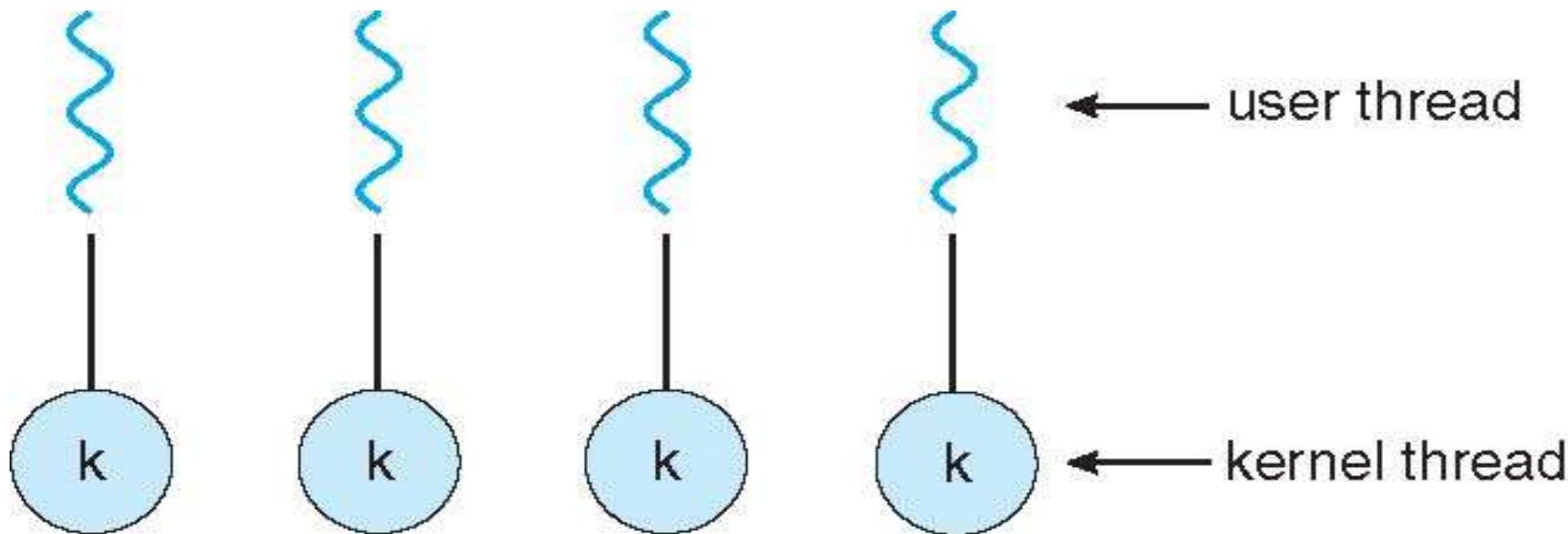
- How to map user threads to kernel threads?
 - Many-to-One
 - One-to-One
 - Many-to-Many
- What if there are no kernel threads?
 - Then only “one” process. Hence many-one mapping possible, to be done by user level thread library

Many-One Model



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

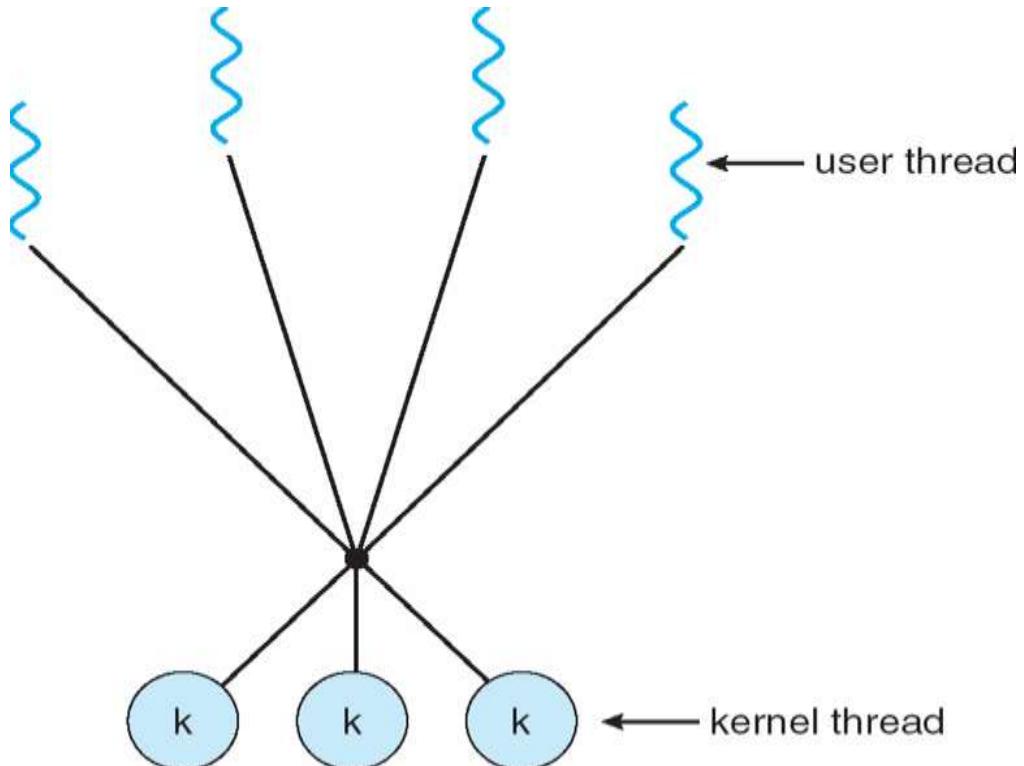
One-One Model



- Each user-level thread maps to kernel thread

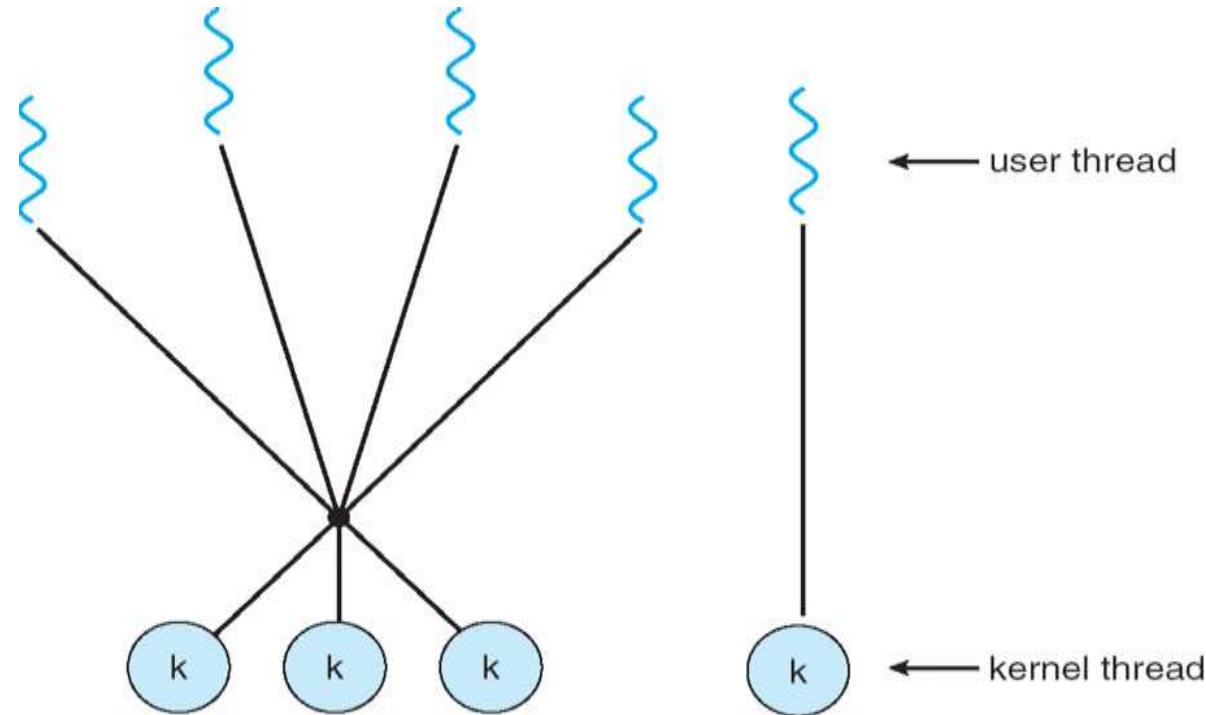
- Examples

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

Two Level Model



- Similar to M:M, except that it allows a user thread to be bound to kernel thread

- Examples

- IRIX

- HP-UX

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.

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)

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

```
int arr[16];  
  
int f() {  
    a(); b(); c();  
}  
  
int g() {  
    x(); y();  
}
```

Scheduler activations for threads

Library

```
th_setup(int n) {  
    max_threads = n;  
    curr_threads = 0;  
}  
  
th_create(..., fn,...) {
```

application

```
f0 {  
    scanf();  
}  
  
g0 {  
    recv();  
}
```

Scheduler activations for threads

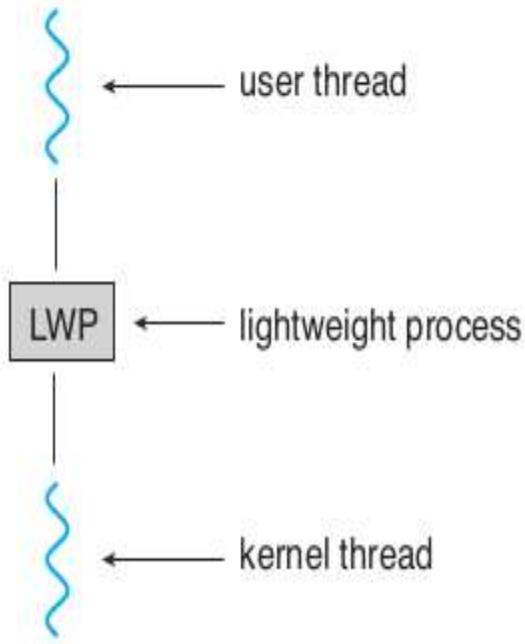
- Scheduler Activations

- 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

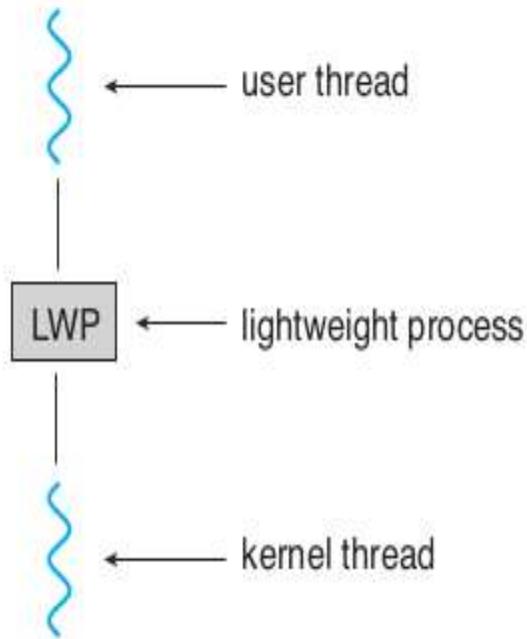
- This communication allows an application to maintain the correct number kernel threads

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.**

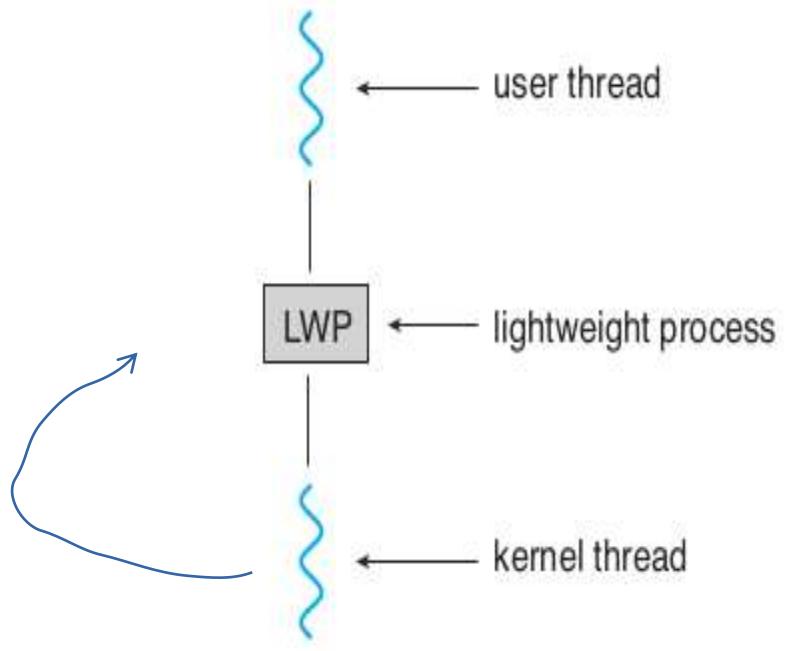
Issues with threads



- **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**

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)

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(),

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

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

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

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
now

Threads and Signals

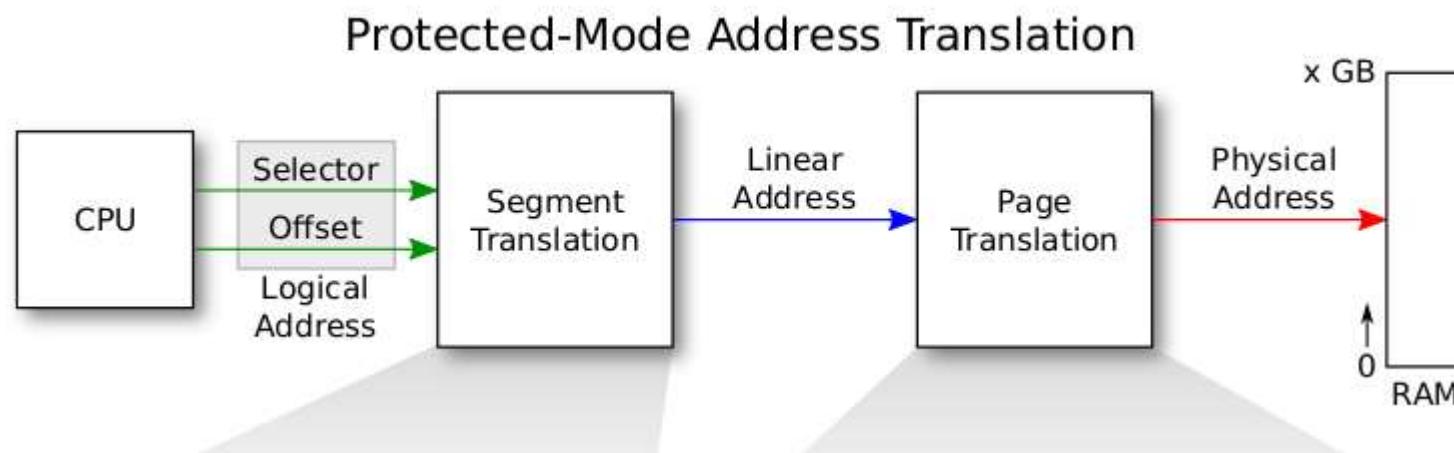
- **Signal handling Options:**
 - **Deliver the signal to the thread to which the signal applies**
 - **Deliver the signal to every thread in the process**
 - **Deliver the signal to certain threads in the process**
 - **Assign a specific thread to receive all signals for the process**

Some numbers and their ‘meaning’

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

X86 Memory Management

X86 address : protected mode address translation



X86 paging

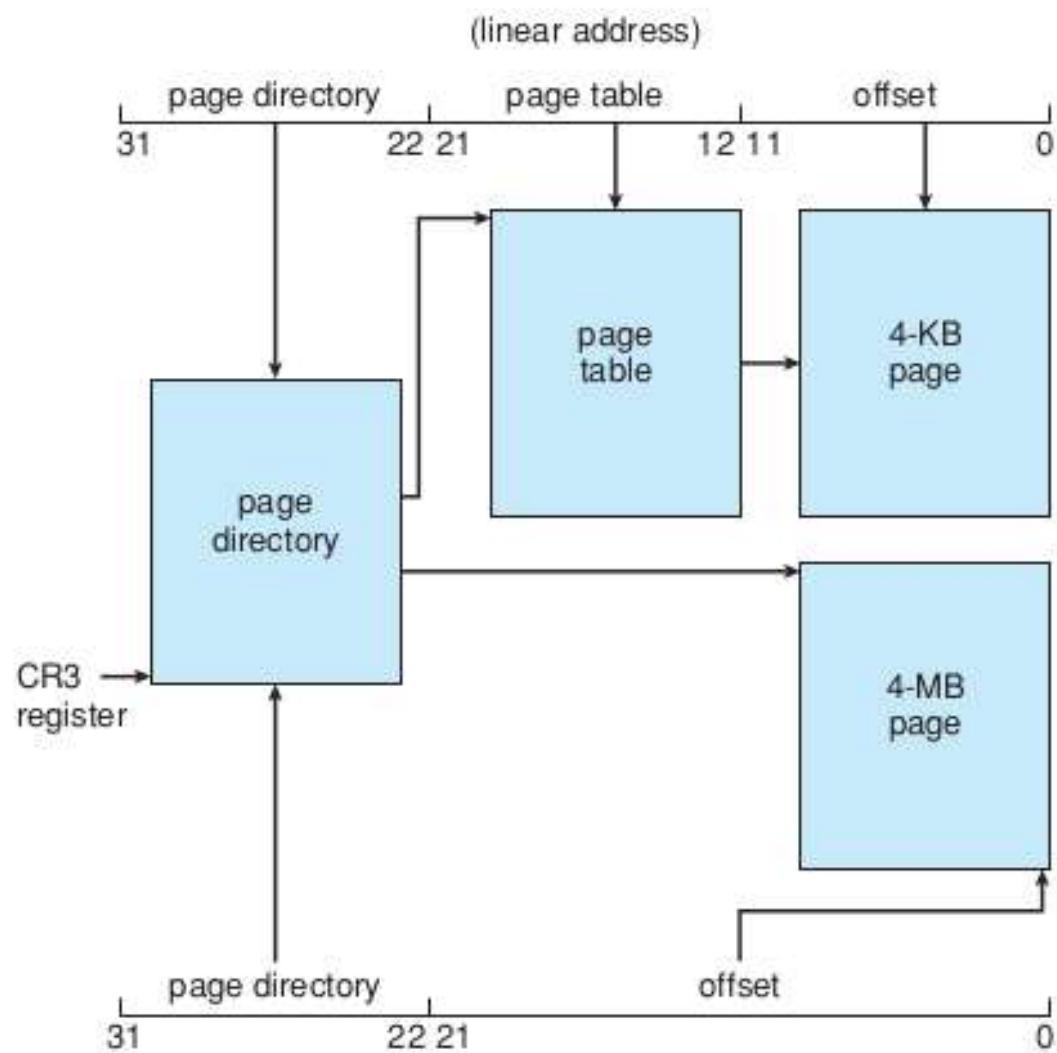
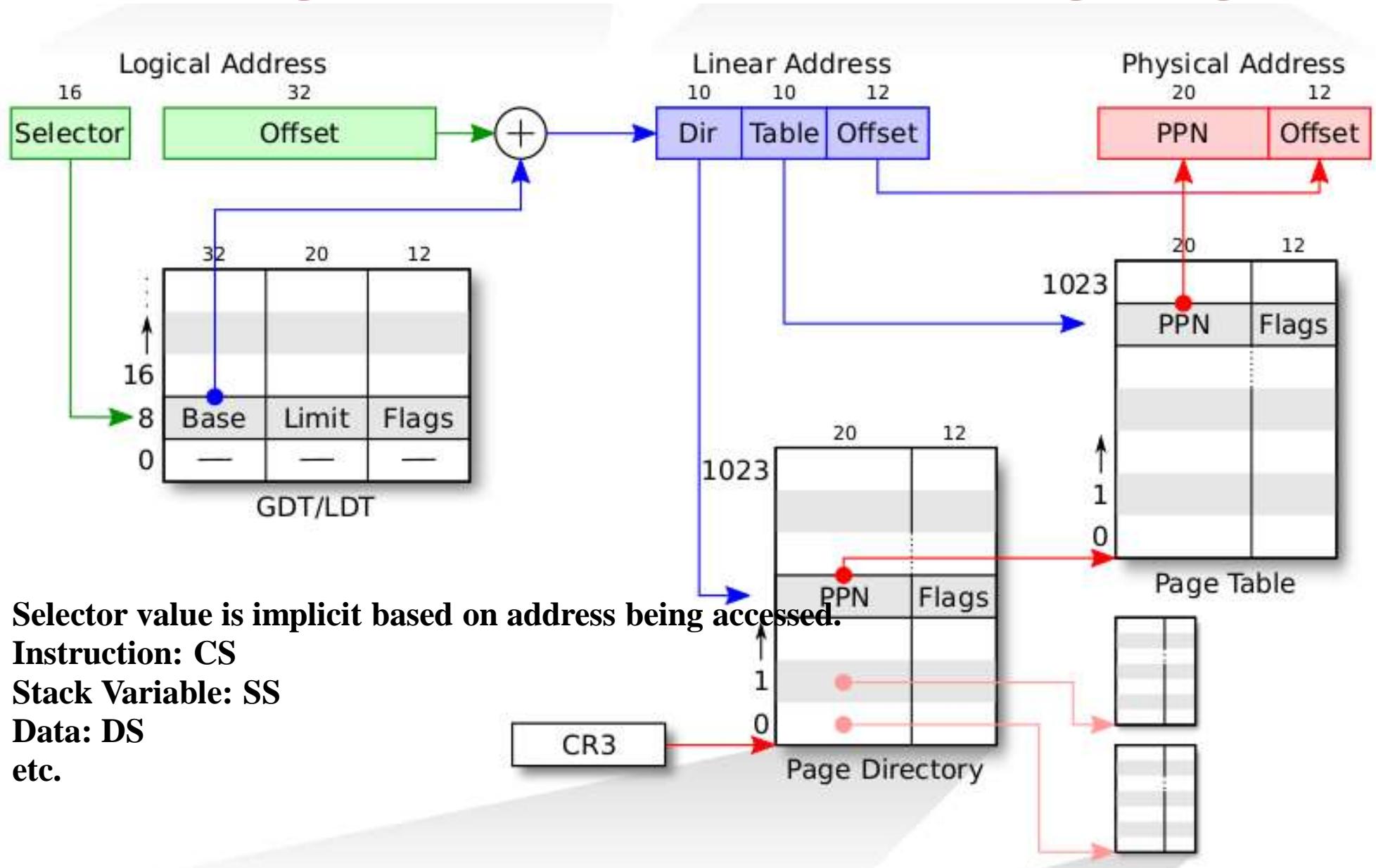


Figure 8.23 Paging in the IA-32 architecture.

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	G	P	S	0	A	C	W	D	T	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

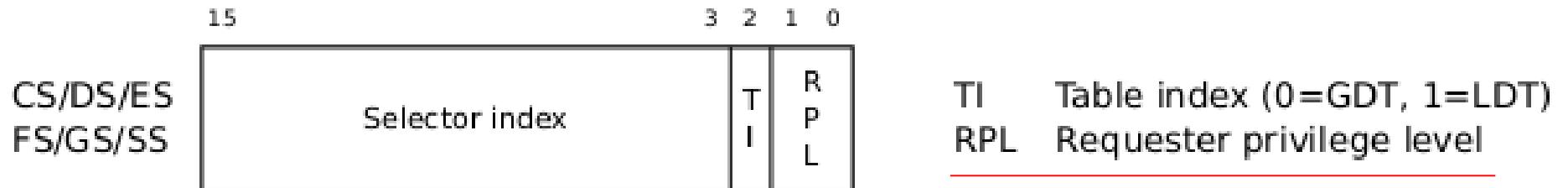
AVL Available for system use

31

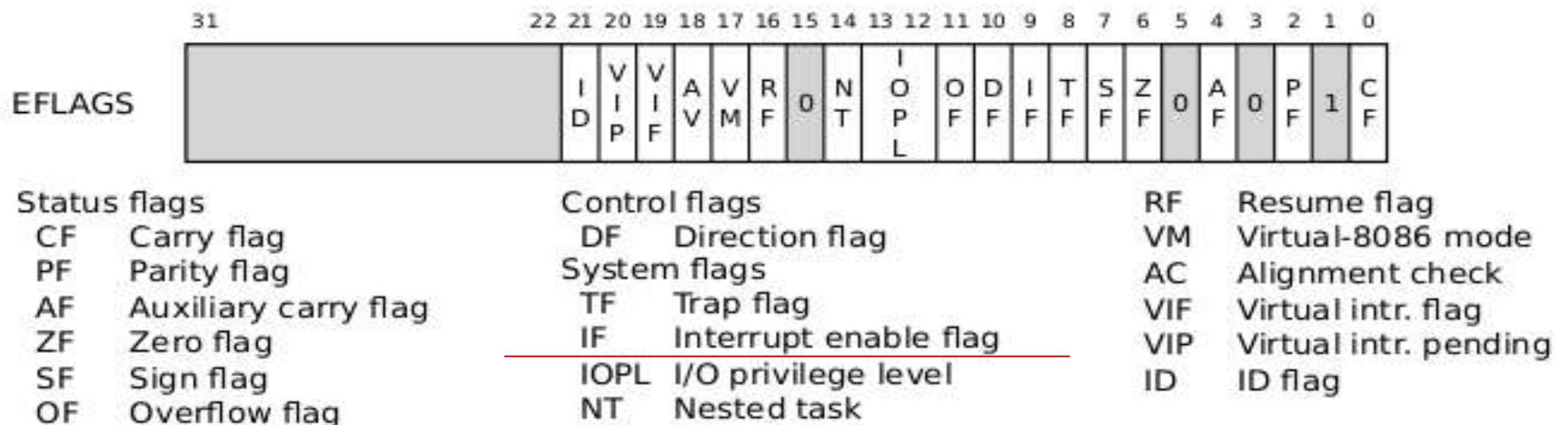
Physical page number	12	11	10	9	8	7	6	5	4	3	2	1	0
	A	G	P	A	D	A	C	W	D	T	U	W	P

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

	31	12 11	5 4 3 2	0
	Page-Directory-Table Base Address		P C D	P W T

PWT Page-level writes transparent

PCT Page-level cache disable

CR4

CR4

31	11	10	9	8	7	6	5	4	3	2	1	0
	O S X M	O S F X	P C G E	P G C E	M A S E	P A S E	P S E	D T S D	T P V I	P V M E		

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

di nos temos à dizer

mmu.h : paging related macros

```
// Page directory and page  
table constants.
```

```
#define NPENTRIES  
1024      // # directory  
entries per page directory
```

```
#define NPTENTRIES  
1024      // # PTEs per page  
table
```

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
```

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**

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**

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
```

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)

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
- Initializes
 - LAPIC on each processor, IOAPIC
 - Disables PIC
 - “Console” hardware (the standard I/O)
 - Serial Port
 - Interrupt Descriptor Table
 - Buffer Cache

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);  
    }  
}
```

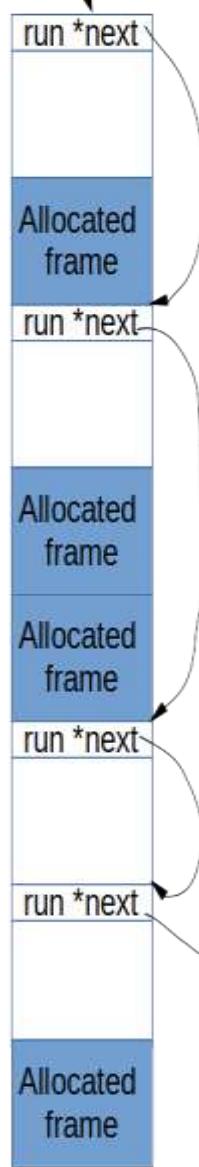
main() in main.c

void

```
freerange(void *vstart, void
*vend)
{
    char *p;
    p =
(char*)PGROUNDUP((uint)vstar
t);
    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

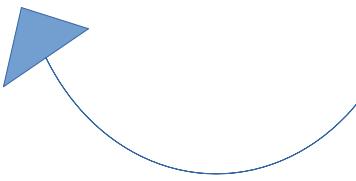


RAM –
divided
into frames

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.

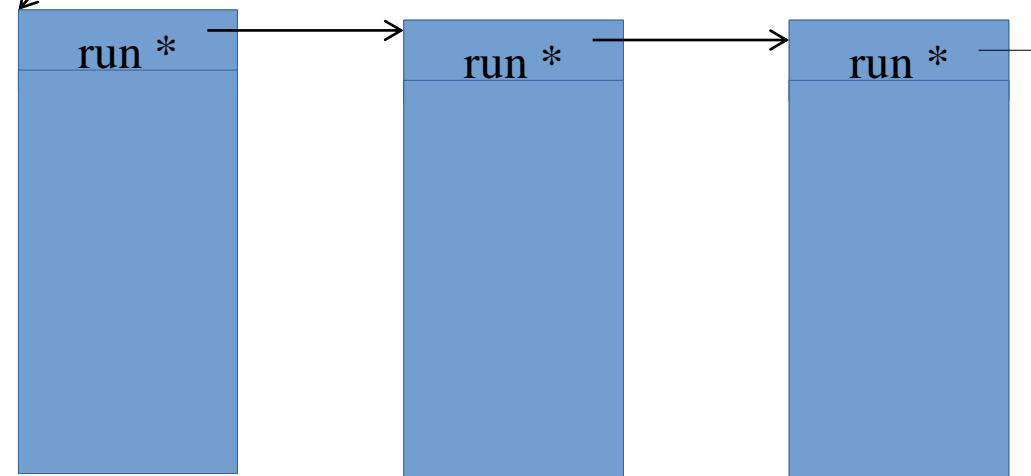
Actually like
this in memory



lock
uselock
run *freelist

kmem

Seen independently



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

 kmalloc(); //
 kernel page table

// Allocate one page
table for the machine
for the kernel address

// space for scheduler
processes.

void

kmalloc(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;
```

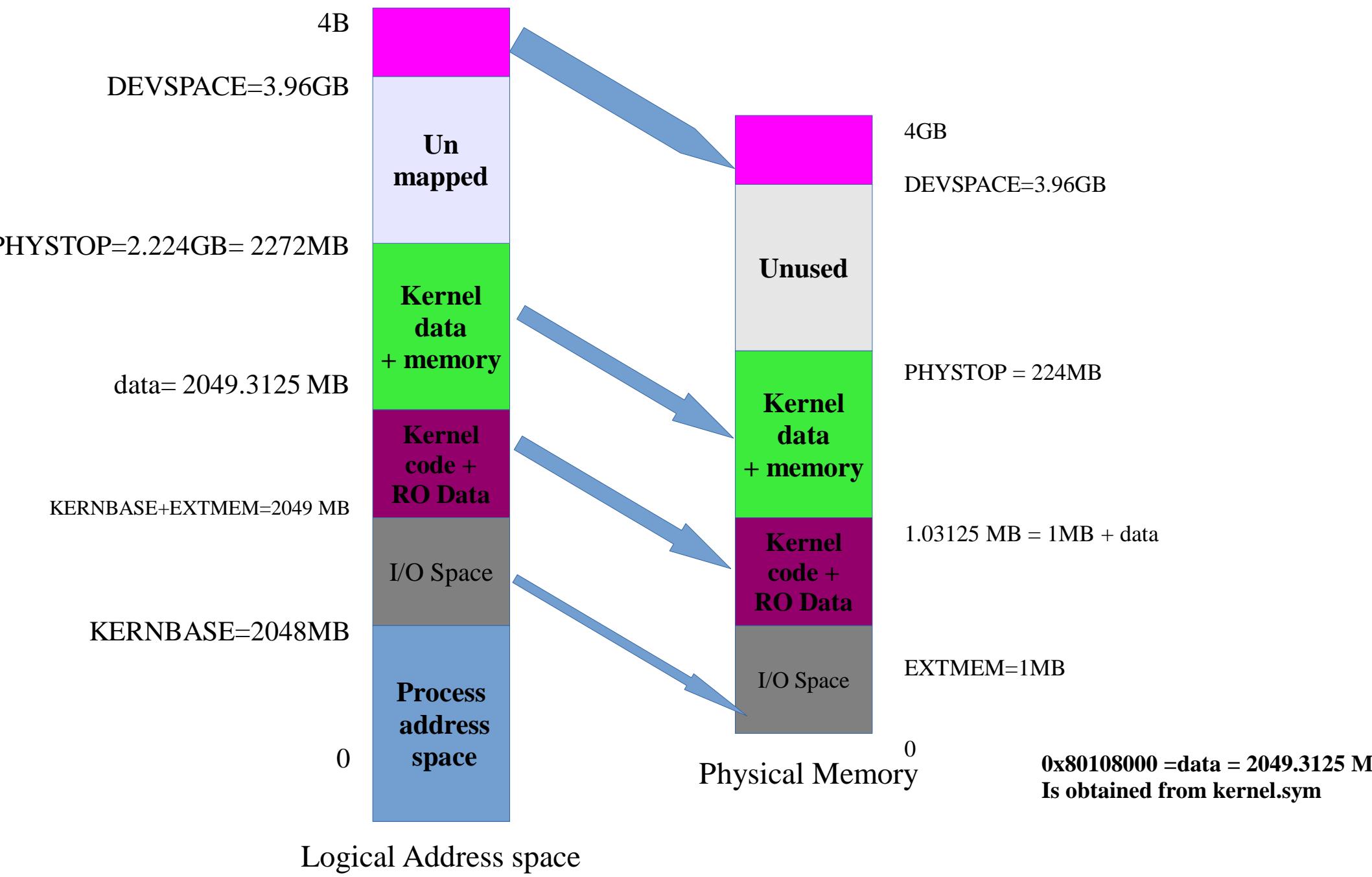
```
        }
```

```
    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 corre



Remidner: PDE and PTE entries

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	D A	A D T	C W U	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

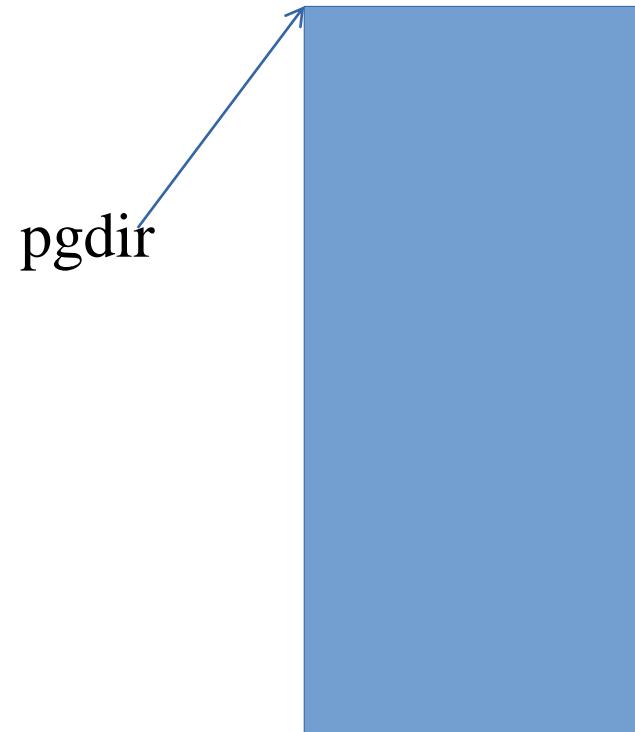
AVL Available for system use

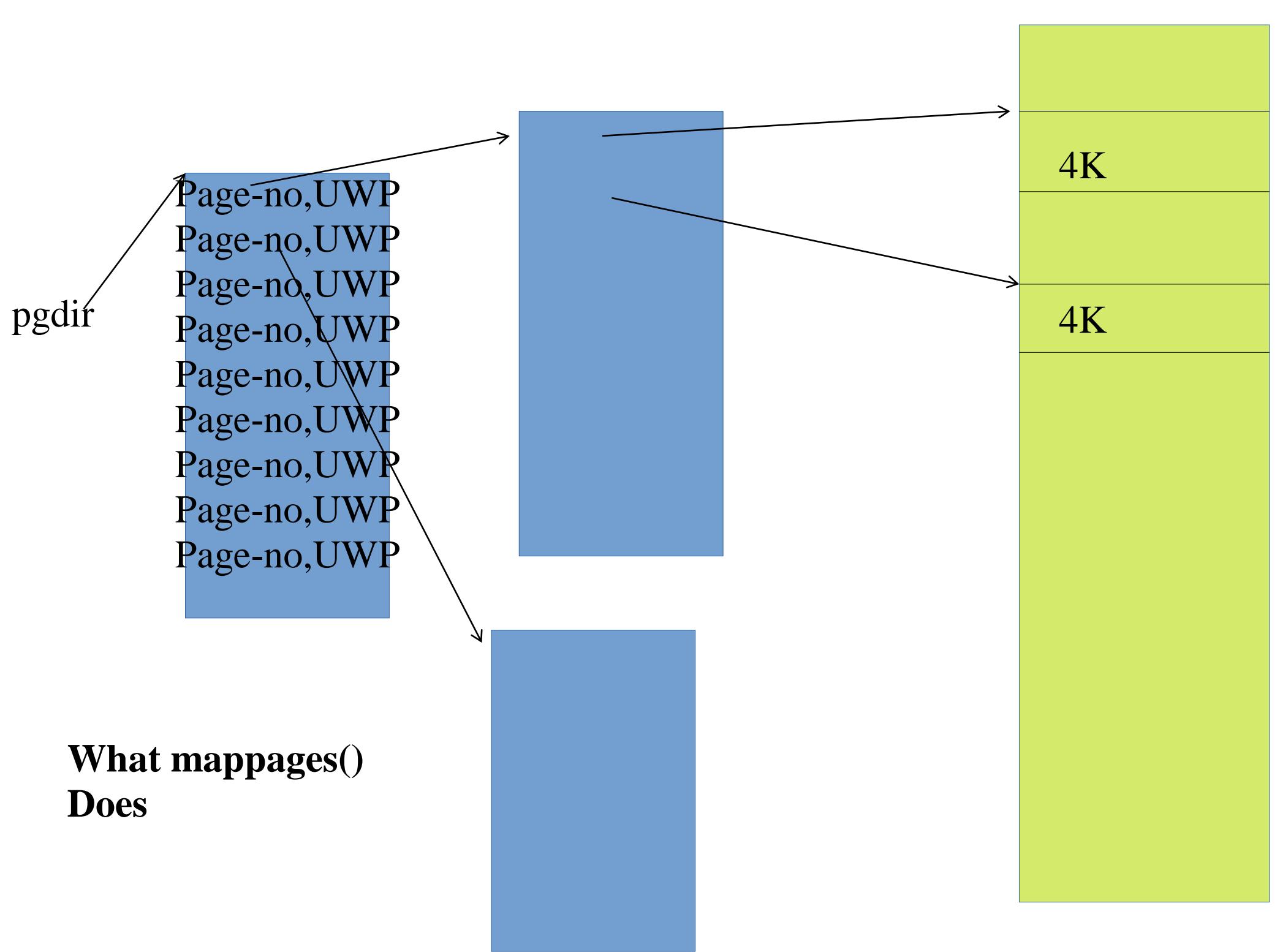
31

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

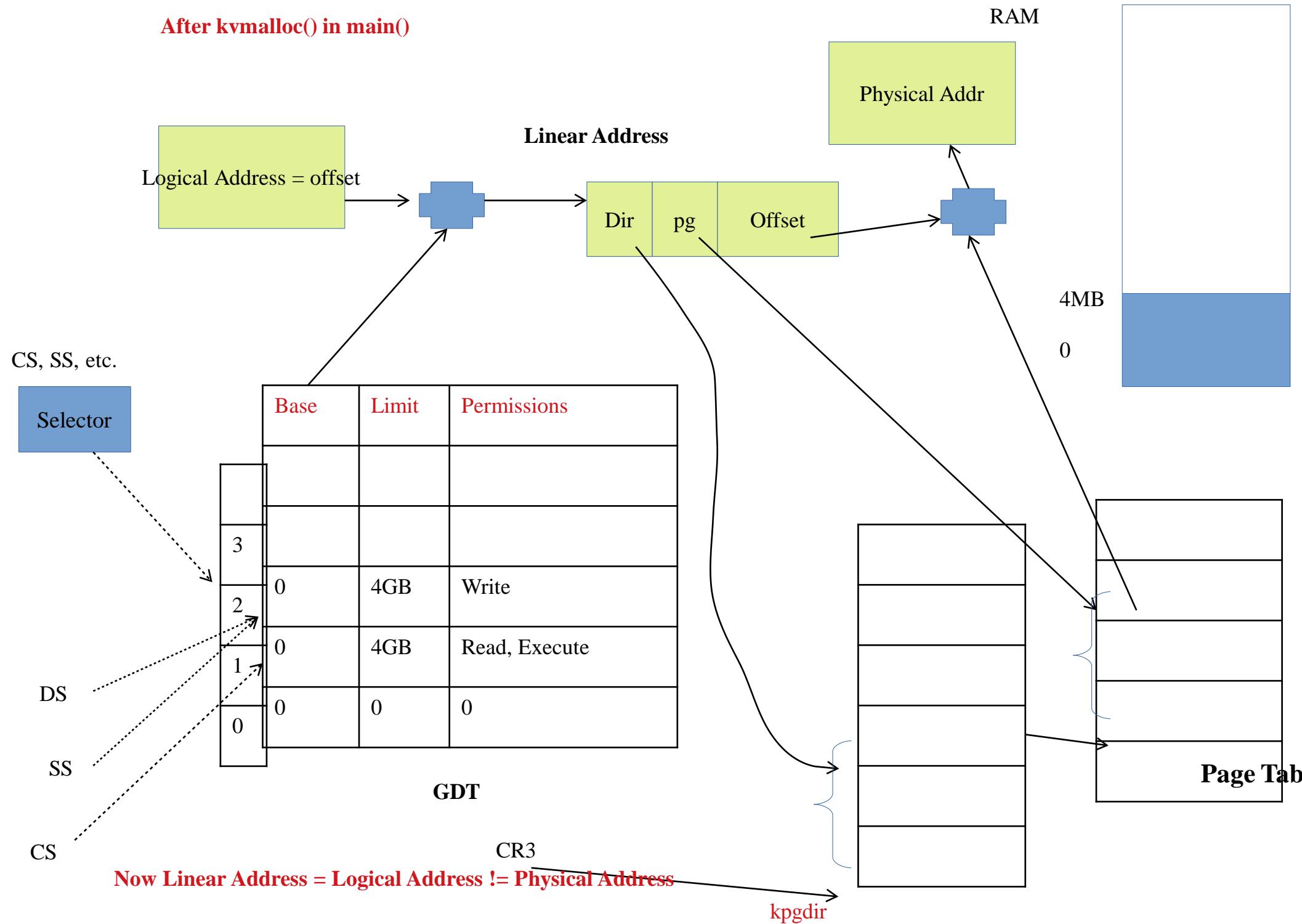
PTE

Before mappages()





After kvmalloc() in main()

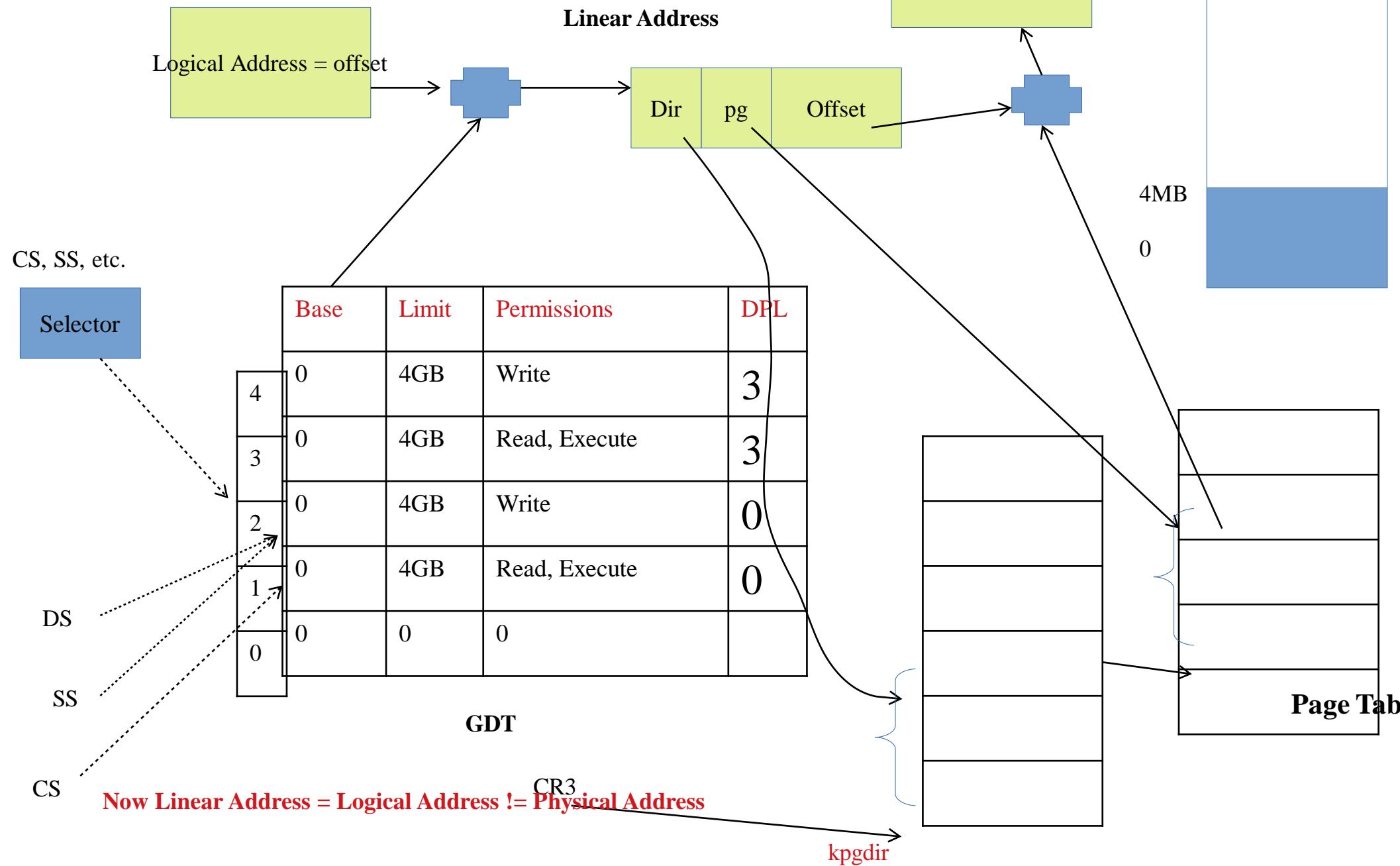


main() -> seginit()

- 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().

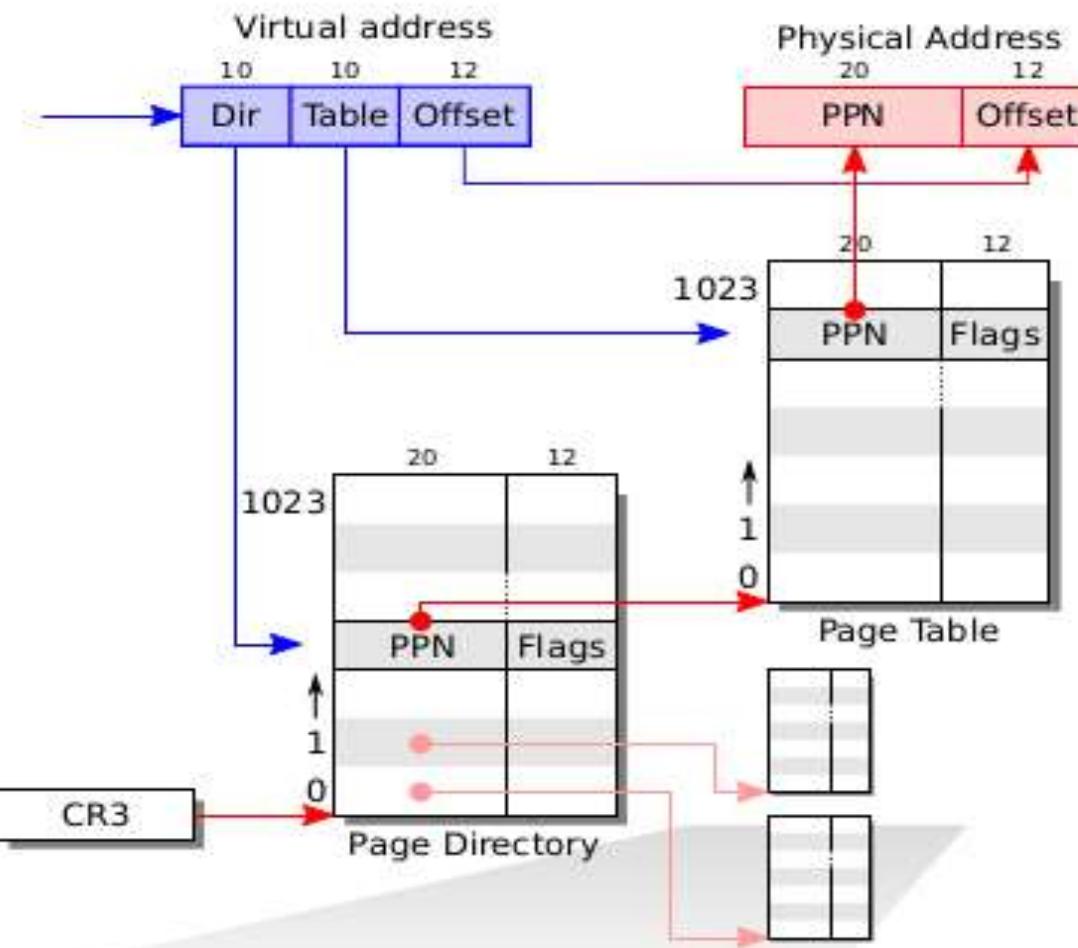
On the processor where we started booting



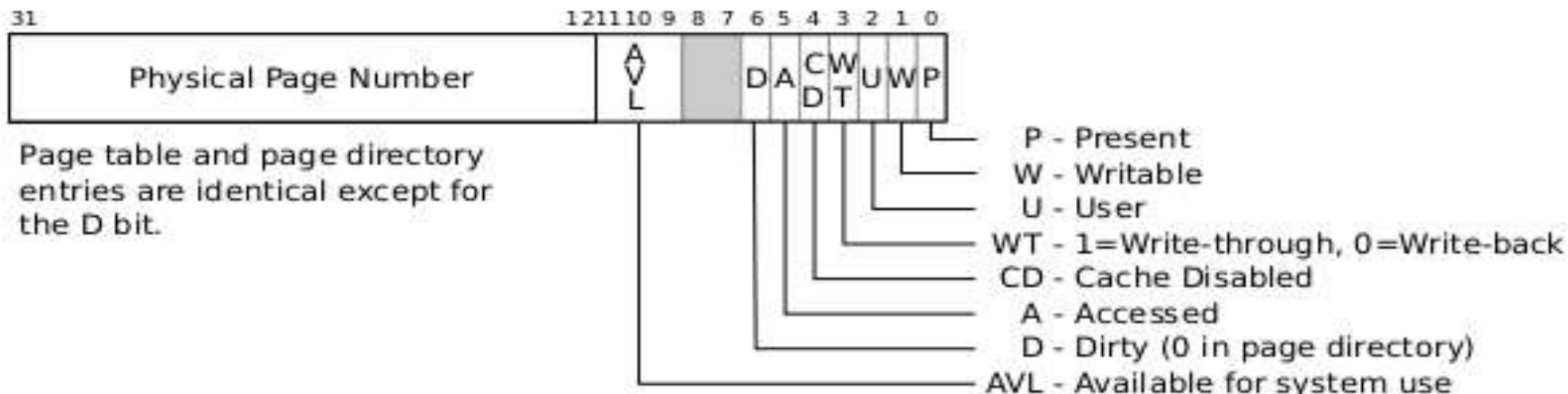
After seginit()

- 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)

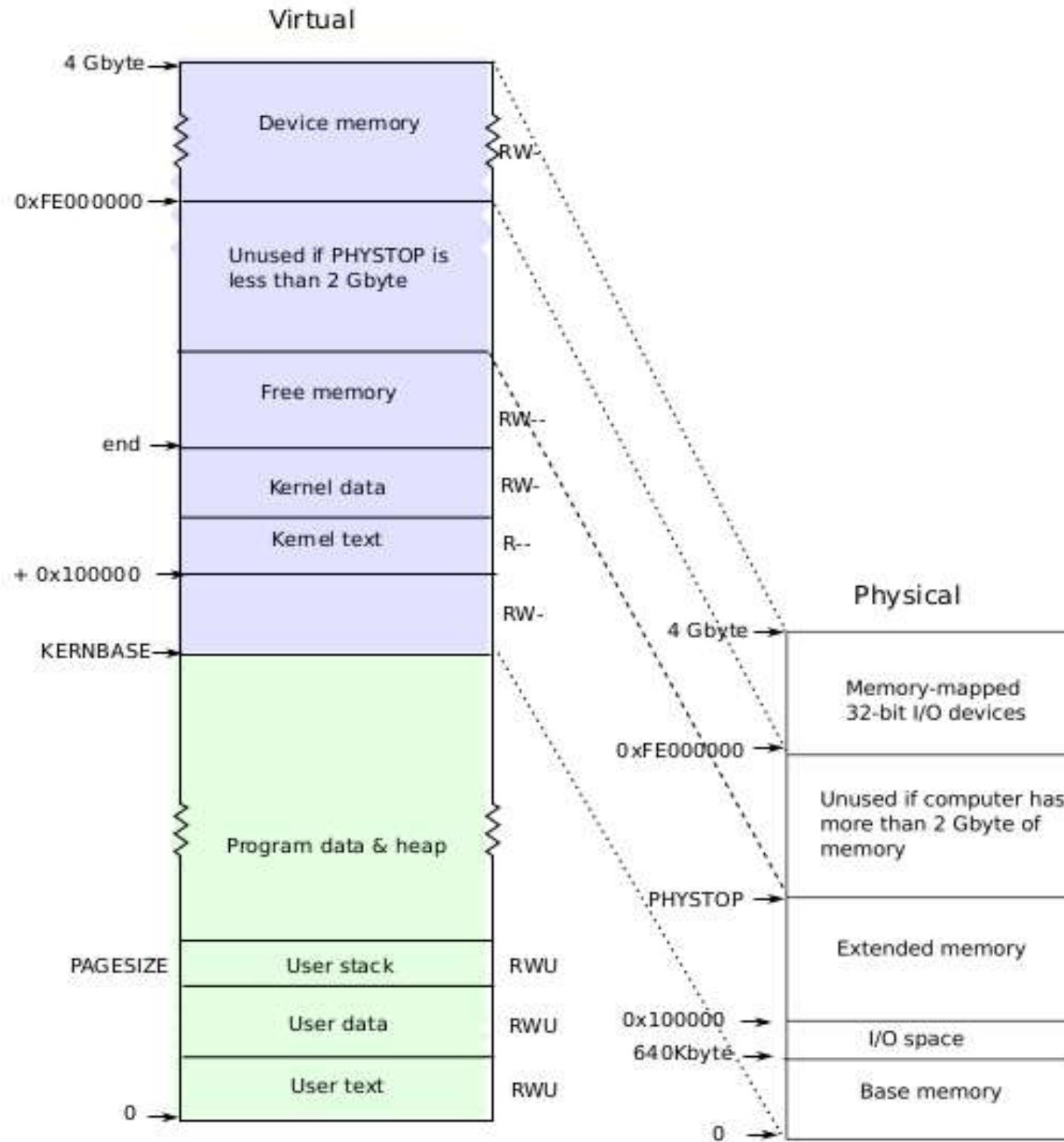
Memory Management

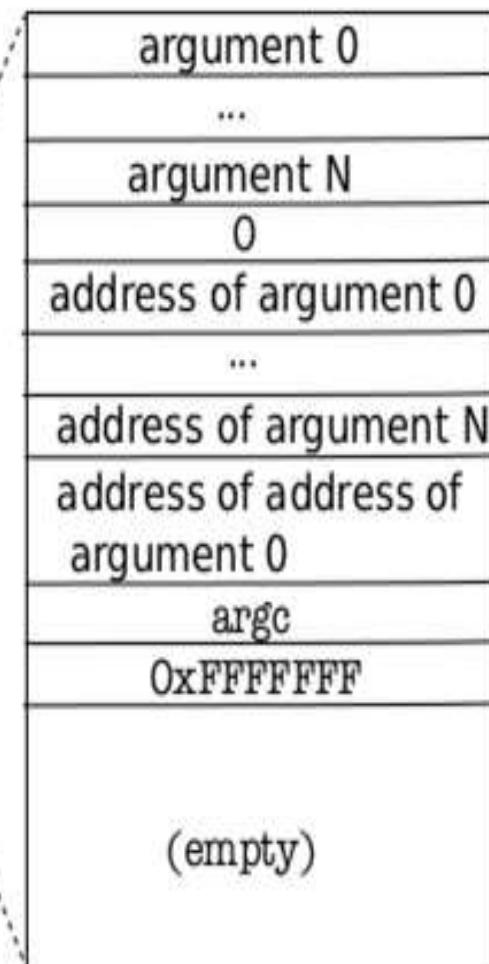
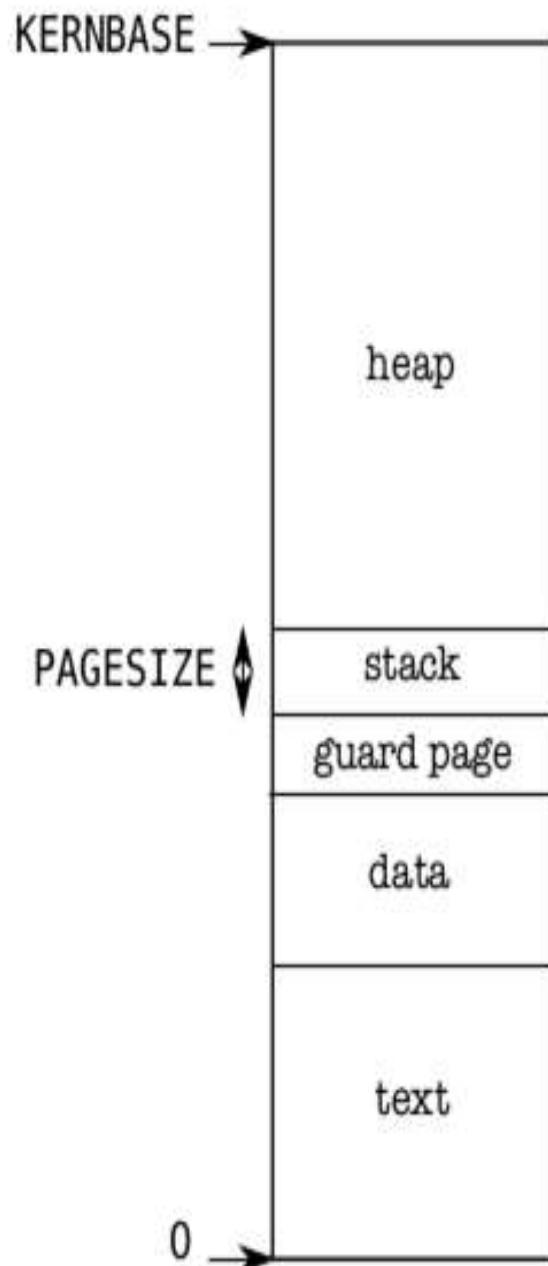


X86 page table hardware



Layout of process's VA space





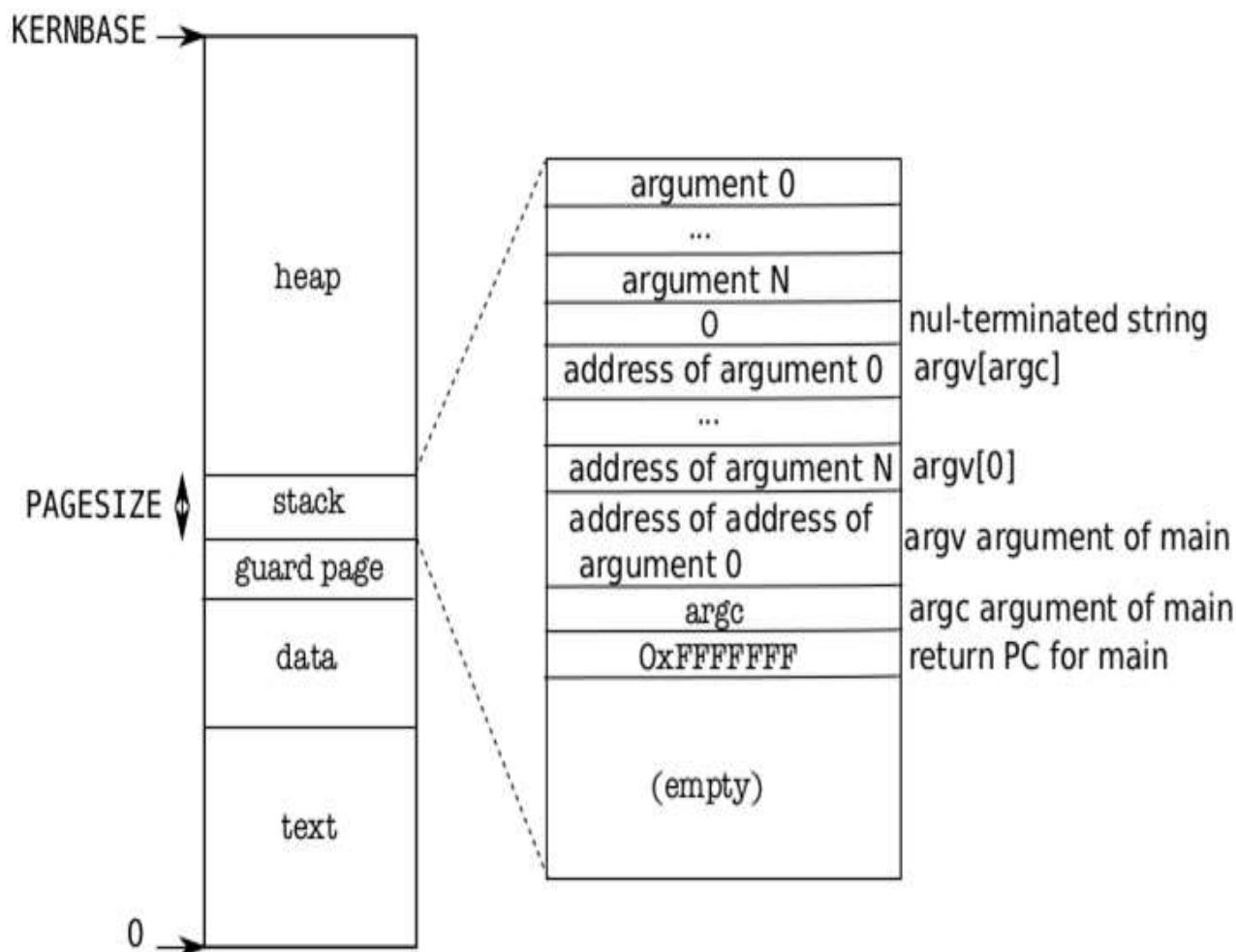
Memory Layout of a user

After exec()

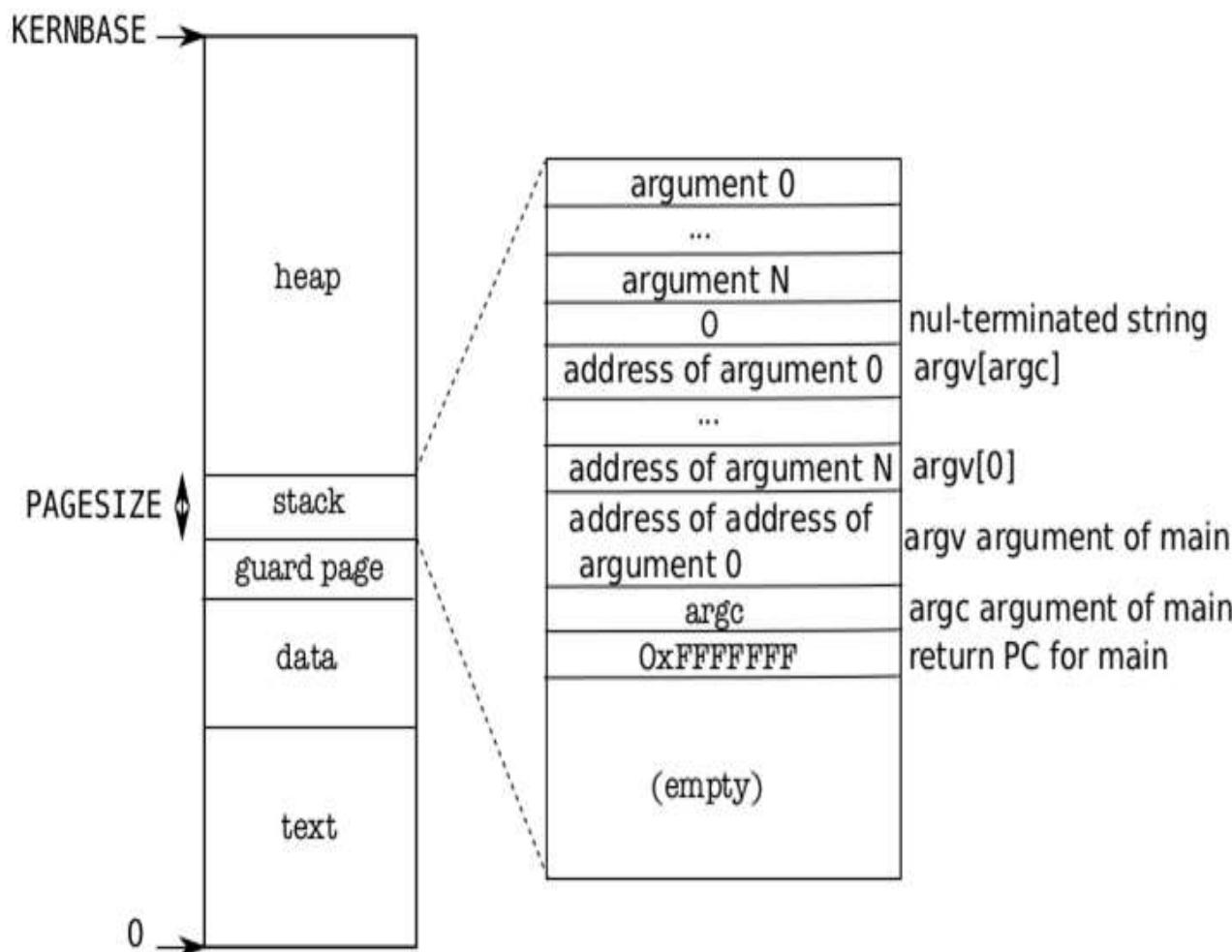
Note the argc, argv on
nul-terminated string
argv[argc]
argv[0]
argv argument of main
argc argument of main
return PC for main

Memory Layout of a user process

The “guard page” is just a map



Memory Layout of a user process



On sbrk()

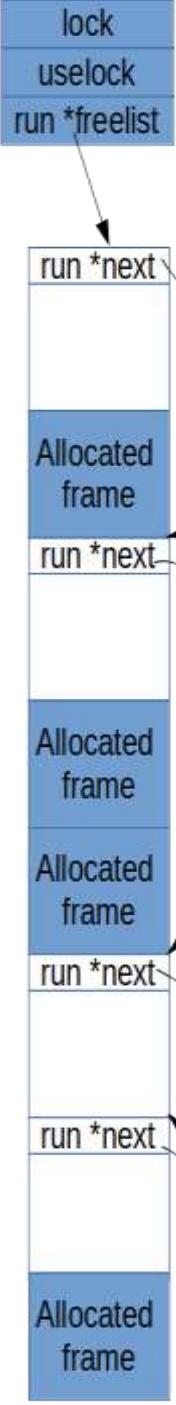
The system call to grow process's ad

growproc()

Allocate a frame, Add an entry in pag
//This entry can't go beyond KERNE
Calls switchuvm()

Switchuvm()

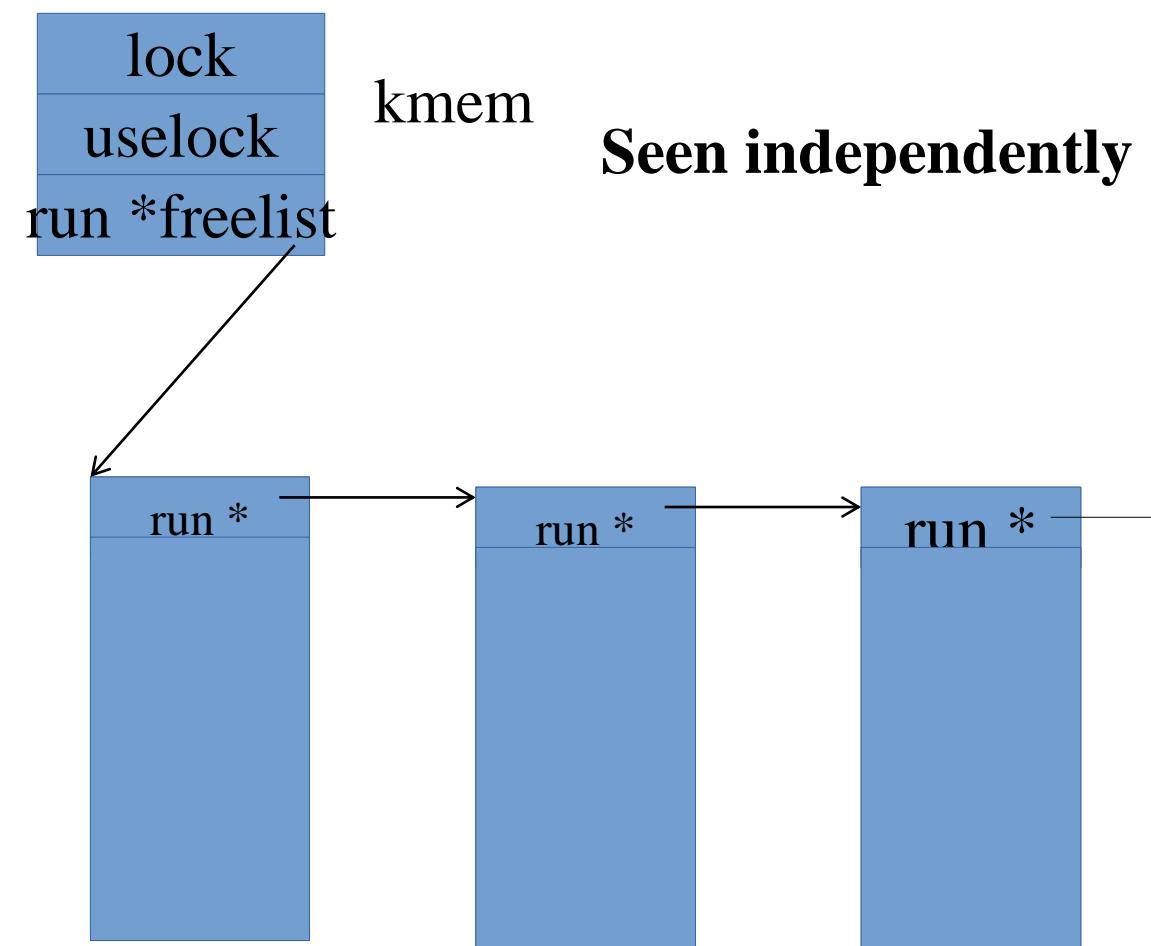
Ultimately loads CR3, invalidating c



RAM –
divided
into frames

Actually like
this in memory

Free List in XV6



Seen independently

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 +

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 = allocuvvm(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;

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

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

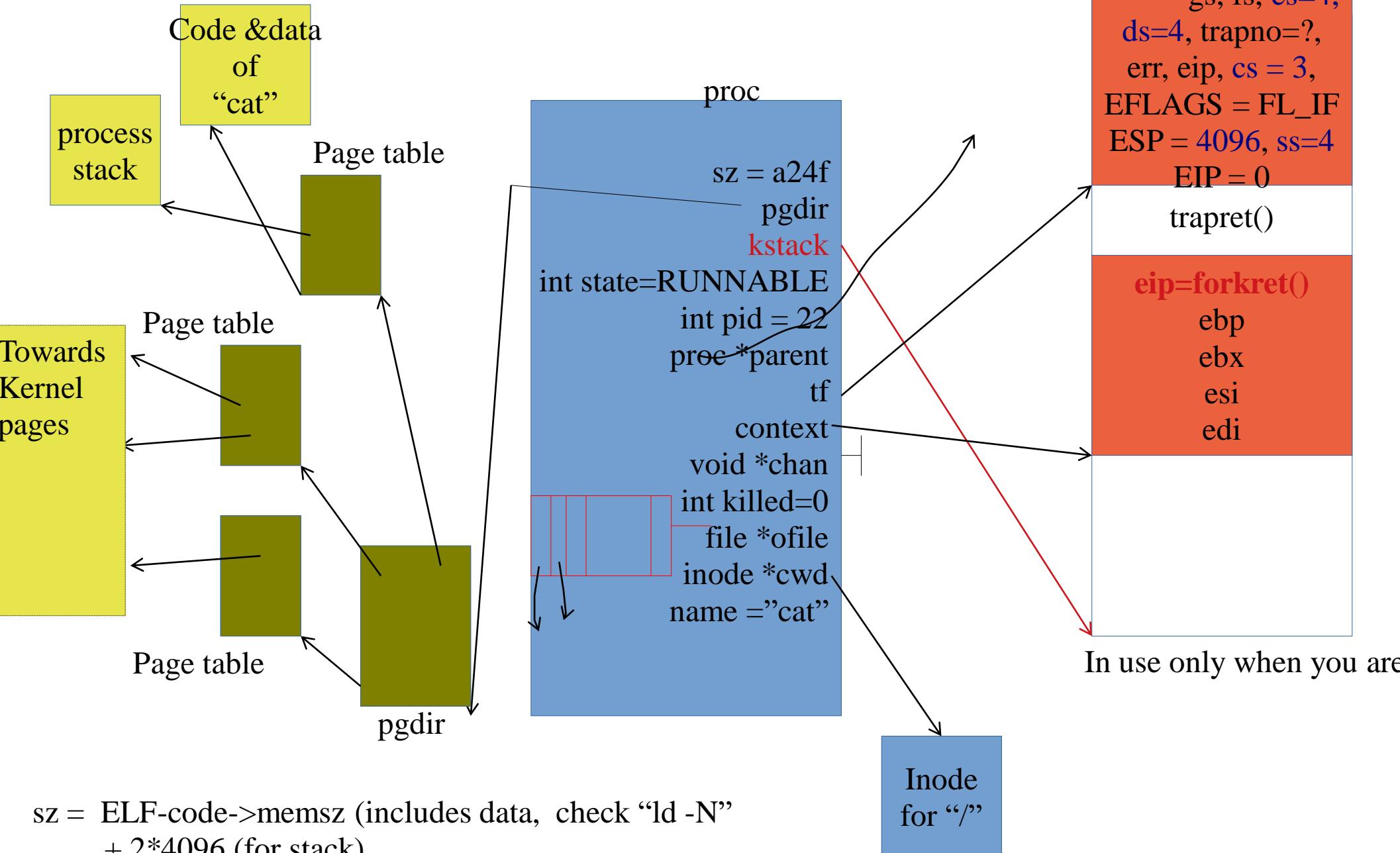
Imp Concepts

- 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

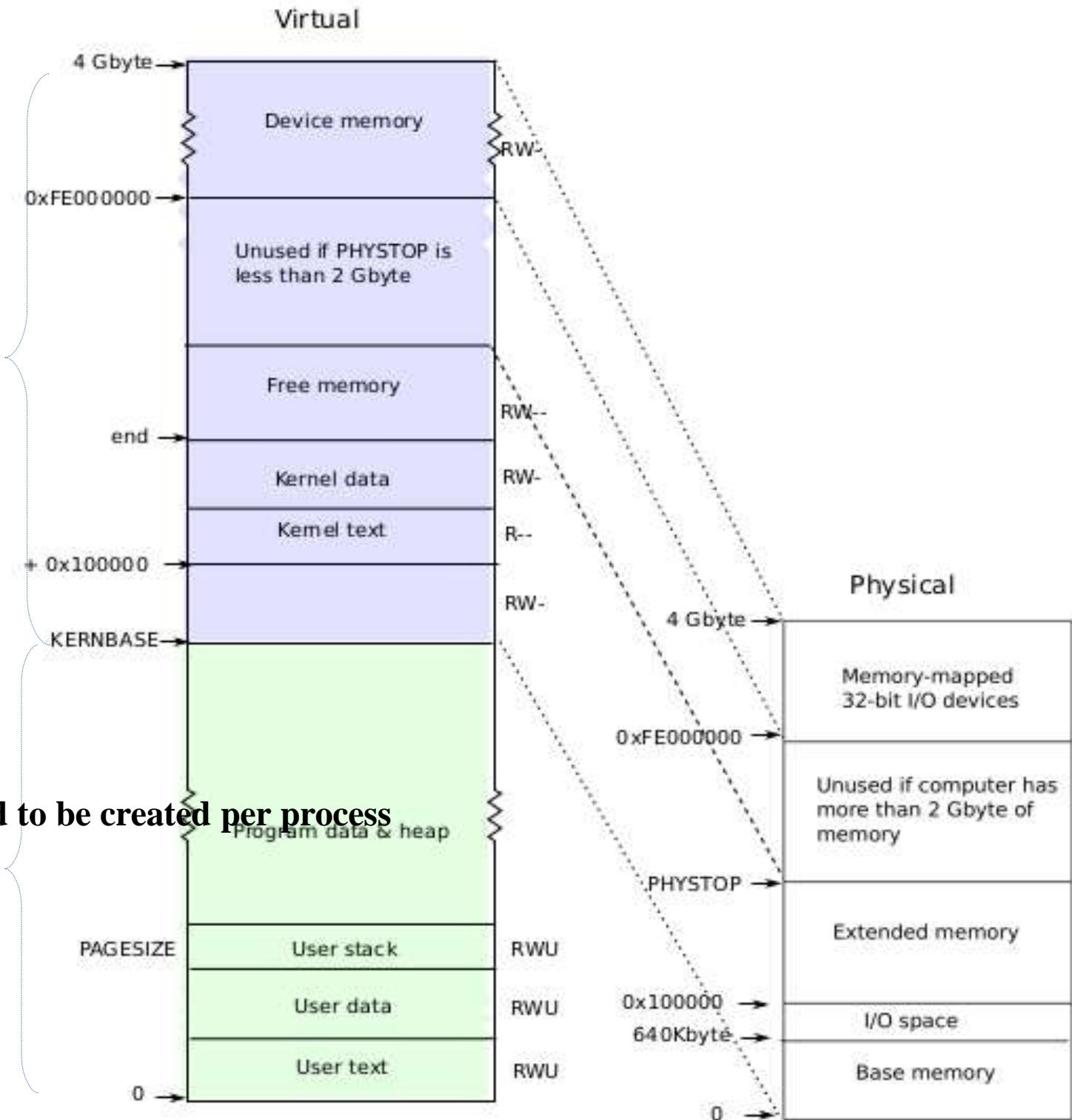
```
struct proc {  
    uint sz;                      // Size of process memory (bytes)  
    pde_t* pgdir;                 // Page table  
    char *kstack;                  // Bottom of kernel stack for this process  
    enum procstate state;         // Process state  
    int pid;                      // 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)  
};
```

Imp Concepts

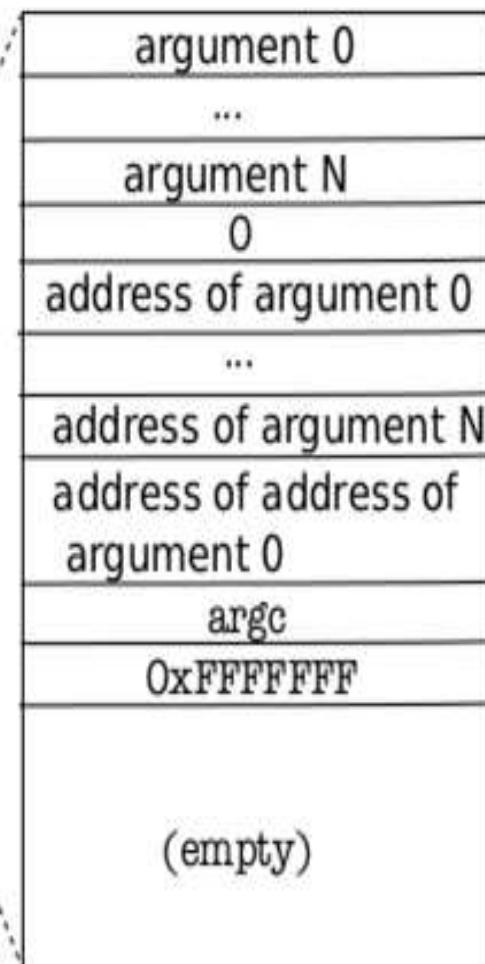
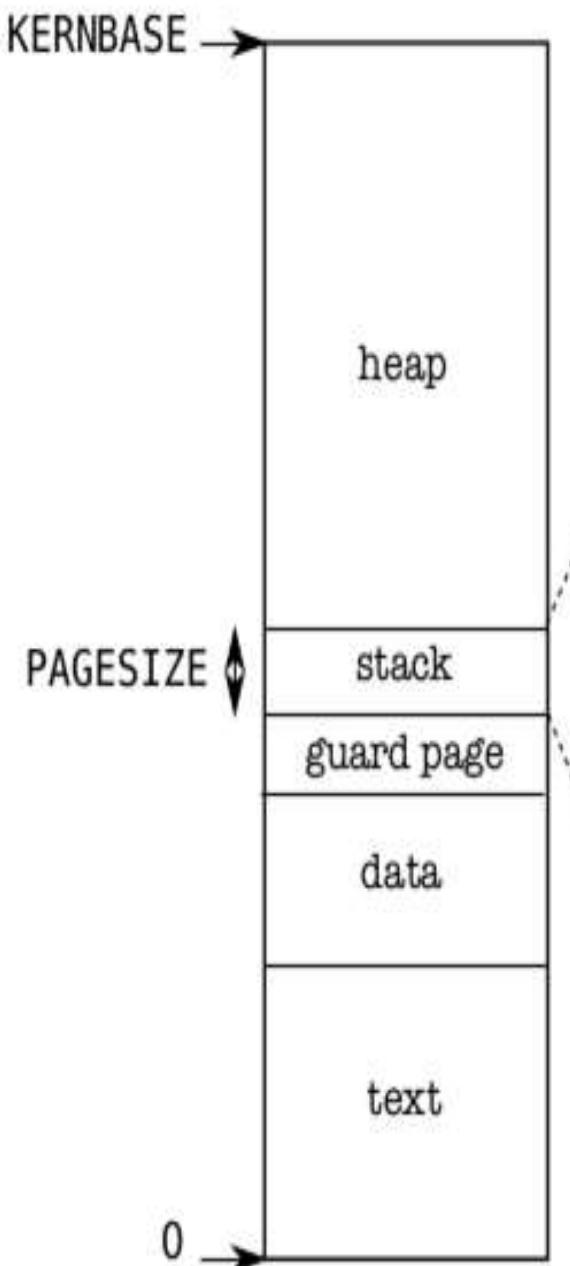
struct proc diagram: Very imp!



setupkvm()
does this mapping



Memory Layout of a user After exec()



Note the argc, argv on
stack is just one page

size of text and data is derived
nul-terminated string
argv[argc]
argv[0]
argv argument of main
argc argument of main
return PC for main

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'

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

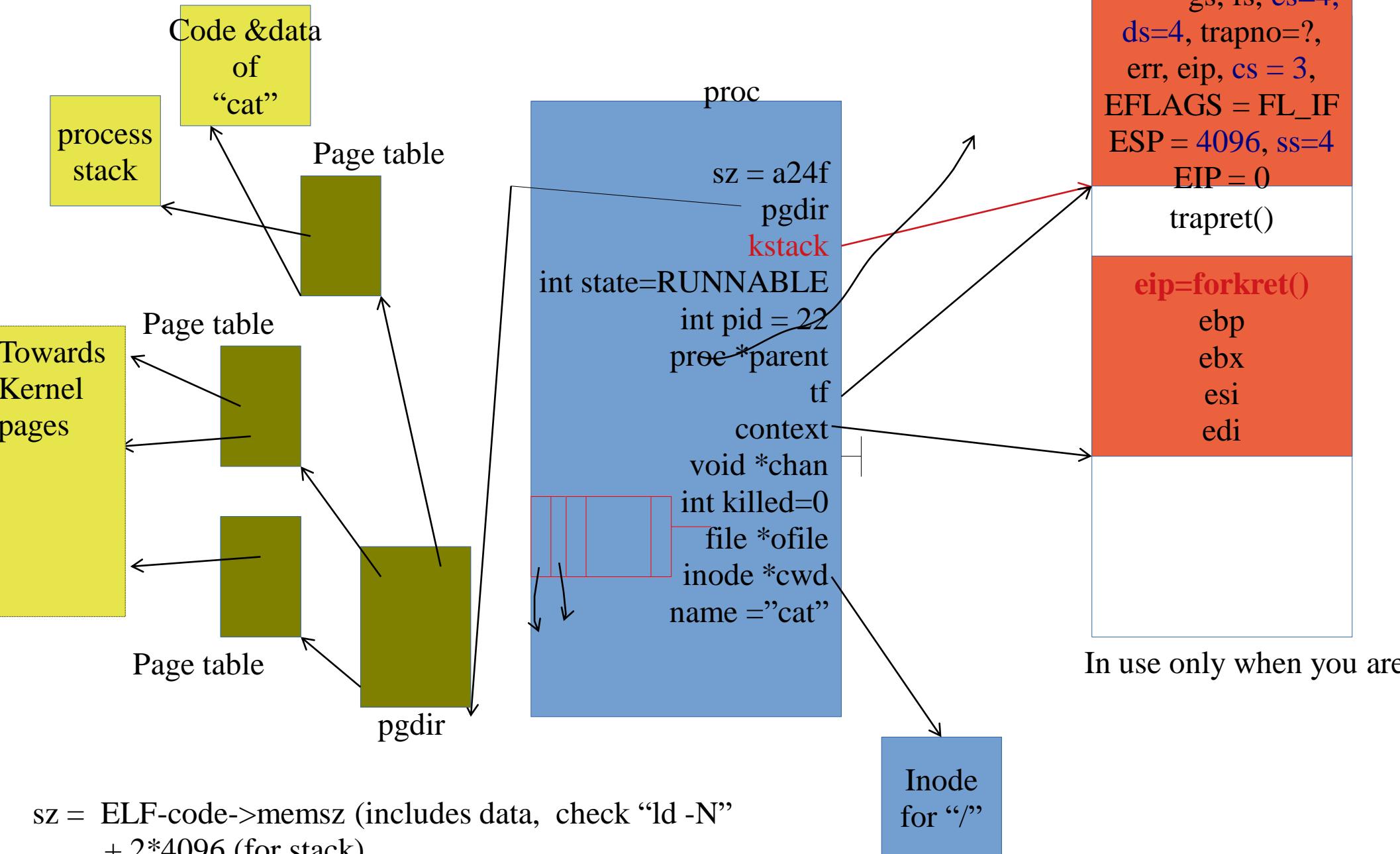
 kernel stack for this process

 enum procstate state;

 // Process state.

 allocated, ready to run, running, wait-

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++)
```

found:

```
p->state = EMBRYO;
p->pid = nextpid++;
release(&ptable.lock);
```

allocproc() setting up stack

```
if((p->kstack = kalloc())  
== 0){
```

```
    p->state = UNUSED;
```

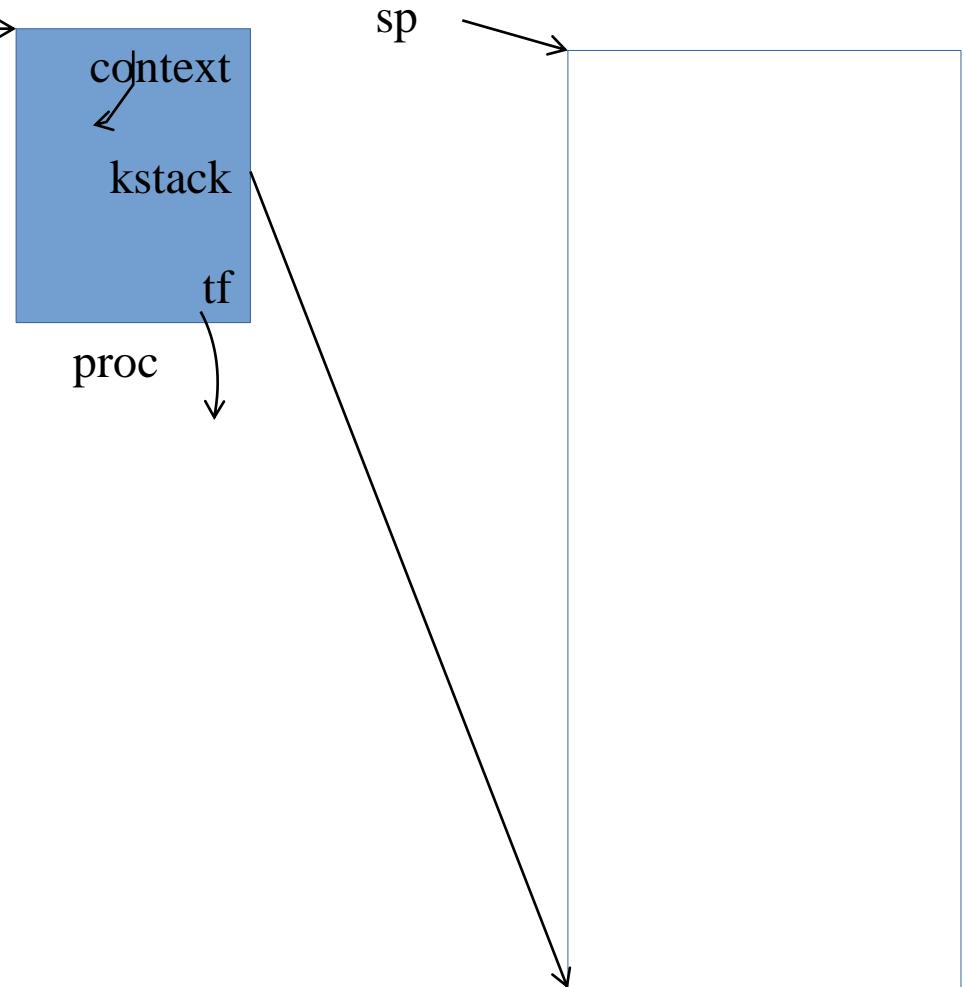
```
    return 0;
```

```
}
```

```
sp = p->kstack +  
KSTACKSIZE;
```

```
// Abhijit KSTACKSIZE =  
PGSIZE
```

```
// Leave room for trap
```



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
```

The diagram illustrates the memory layout for a process. A pointer **p** points to a **proc** structure. This structure contains fields **context**, **kstack**, and **tf**. The **kstack** field points to a stack area. The stack area starts at address **sp** and has a size of **sizeof(trapframe)**.

allocproc() setting up stack

```
if((p->kstack = kalloc())  
== 0){
```

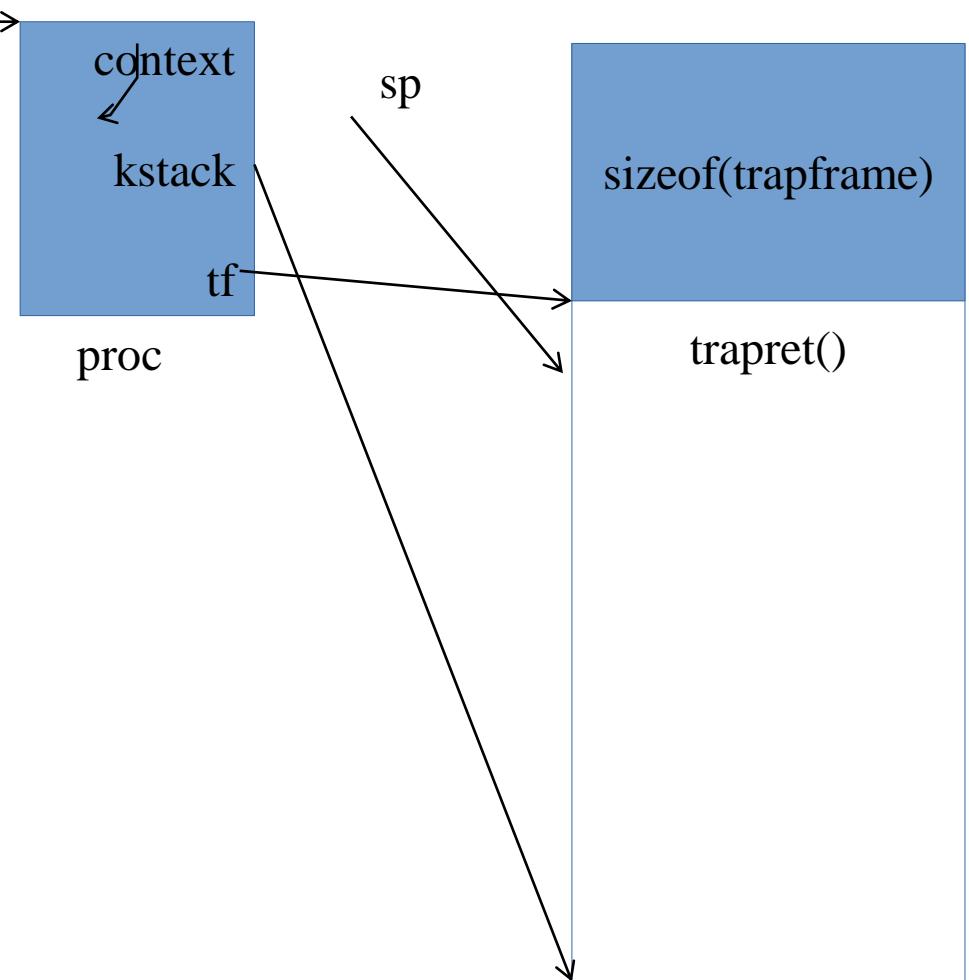
```
    p->state = UNUSED;
```

```
    return 0;
```

```
}
```

```
sp = p->kstack +  
KSTACKSIZE;
```

```
// Abhijit KSTACKSIZE =  
PGSIZE
```



```
// Leave room for trap
```

allocproc() setting up stack

```
if((p->kstack = kalloc())  
== 0){
```

```
    p->state = UNUSED;
```

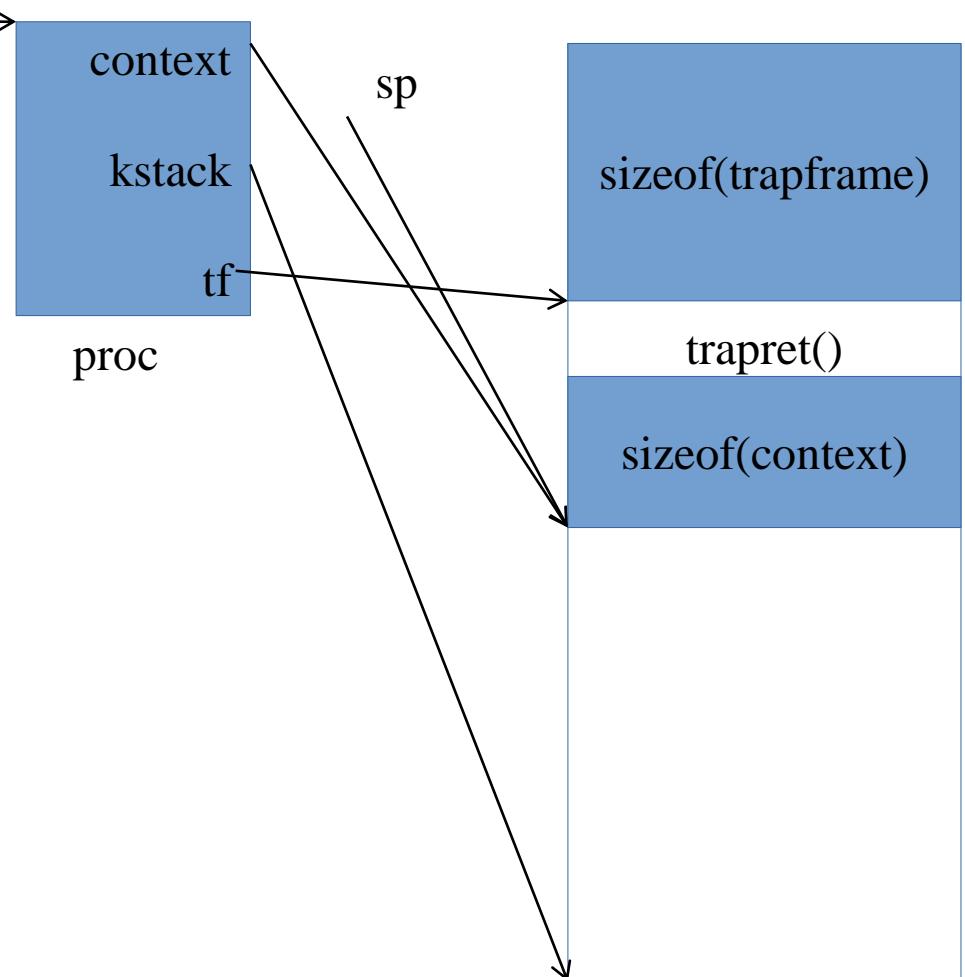
```
    return 0;
```

```
}
```

```
sp = p->kstack +  
KSTACKSIZE;
```

```
// Abhijit KSTACKSIZE =  
PGSIZE
```

```
// Leave room for trap
```



allocproc() setting up stack

```
if((p->kstack = kalloc())  
== 0){
```

```
    p->state = UNUSED;
```

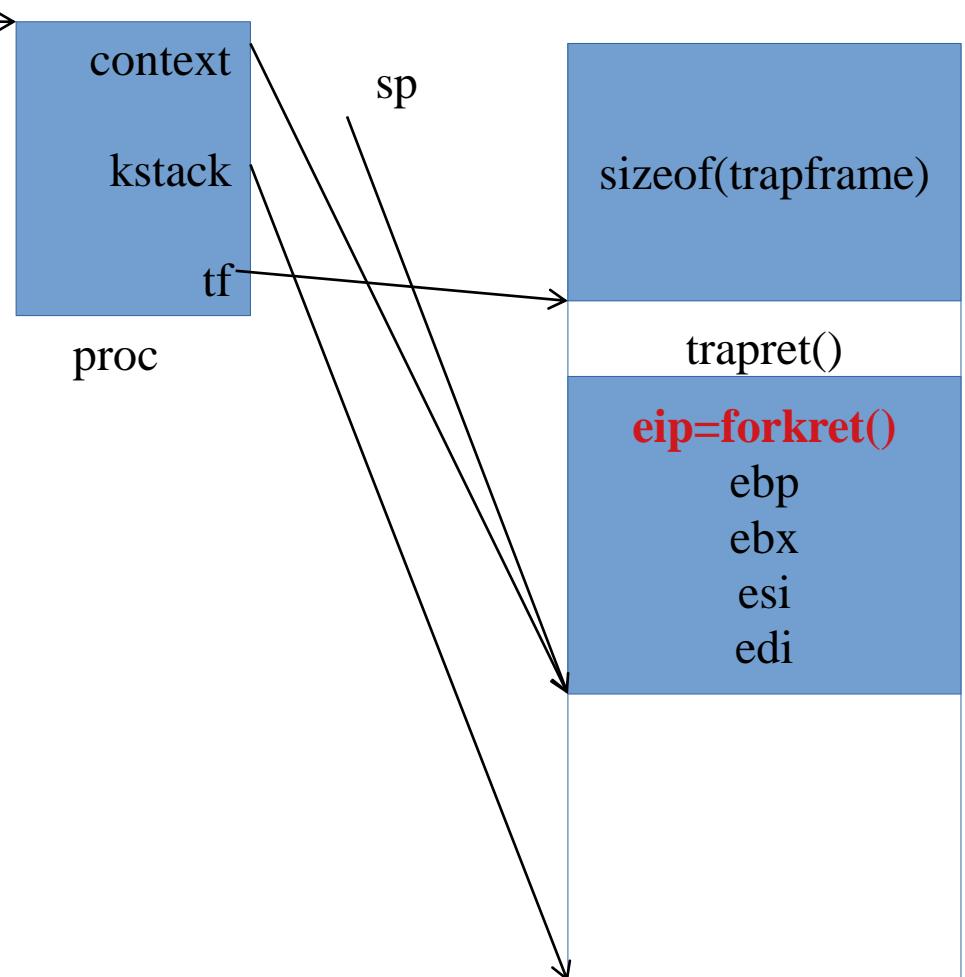
```
    return 0;
```

```
}
```

```
sp = p->kstack +  
KSTACKSIZE;
```

```
// Abhijit KSTACKSIZE =  
PGSIZE
```

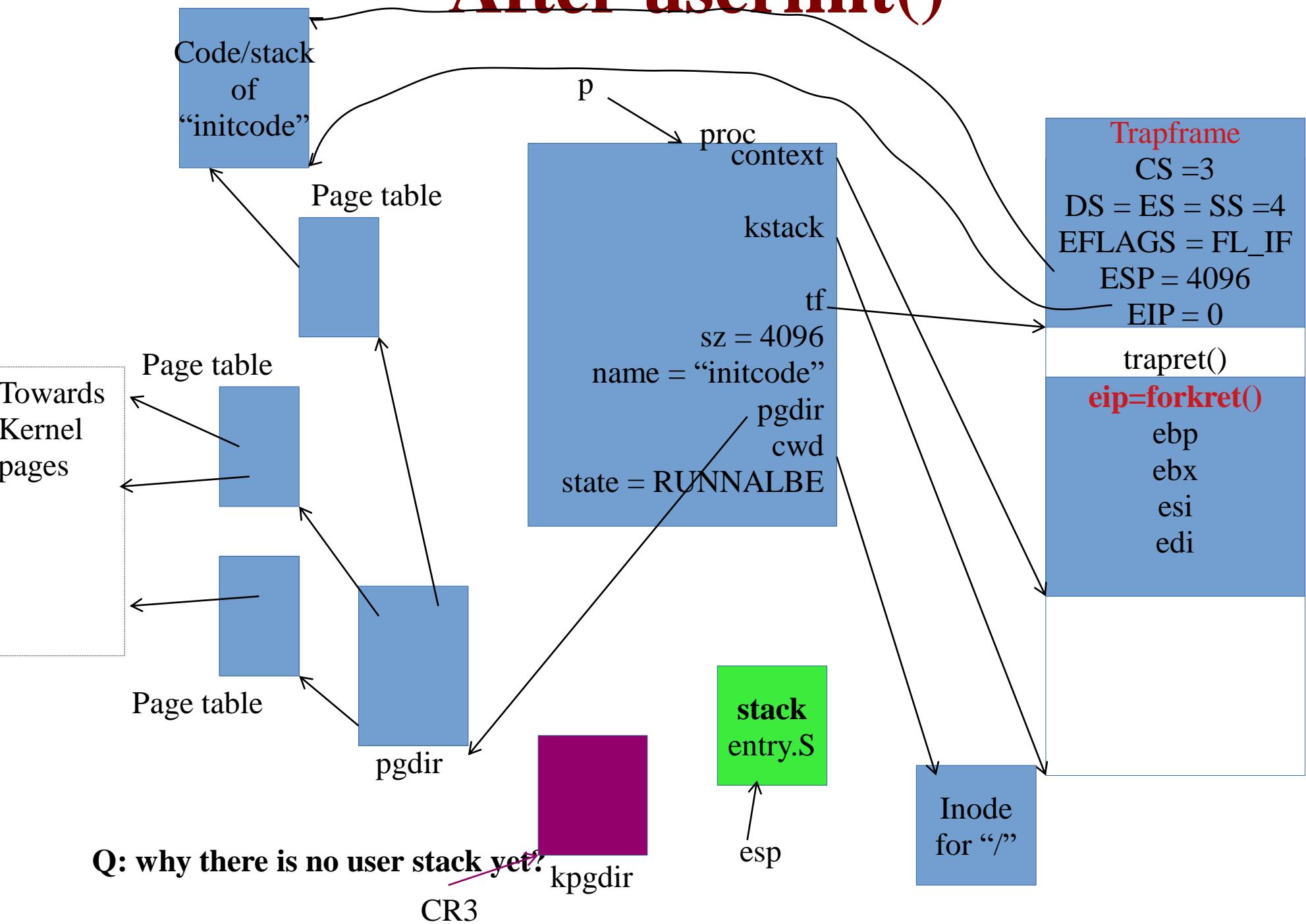
```
// Leave room for trap
```



Next in userinit()

```
initproc = p;  
  
if((p->pgdir ==  
setupkvm()) == 0)  
    panic("userinit: out of  
memory?");  
  
inituvm(p->pgdir,  
        _binary_initcode_start,  
        (int)_binary_initcode_siz  
e);  
  
p->sz = PGSIZE;  
  
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)  
{  
    printf("cpu%d:  
starting %d\n", cpuid(),  
cpuid());  
  
    idtinit(); // load idt  
register  
  
    xchg(&(mycpu()-  
>started), 1); // tell
```

- 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
- Fields in cpu[] not yet set
- context * scheduler --> will be setup in sched()
- taskstate ts --> large structure, only parts used in switchuvm()
- ncli, intena --> used while locking
- proc *proc -> set during scheduler()

```
{
```

```
struct proc *p;
```

scheduler()

```
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)
```

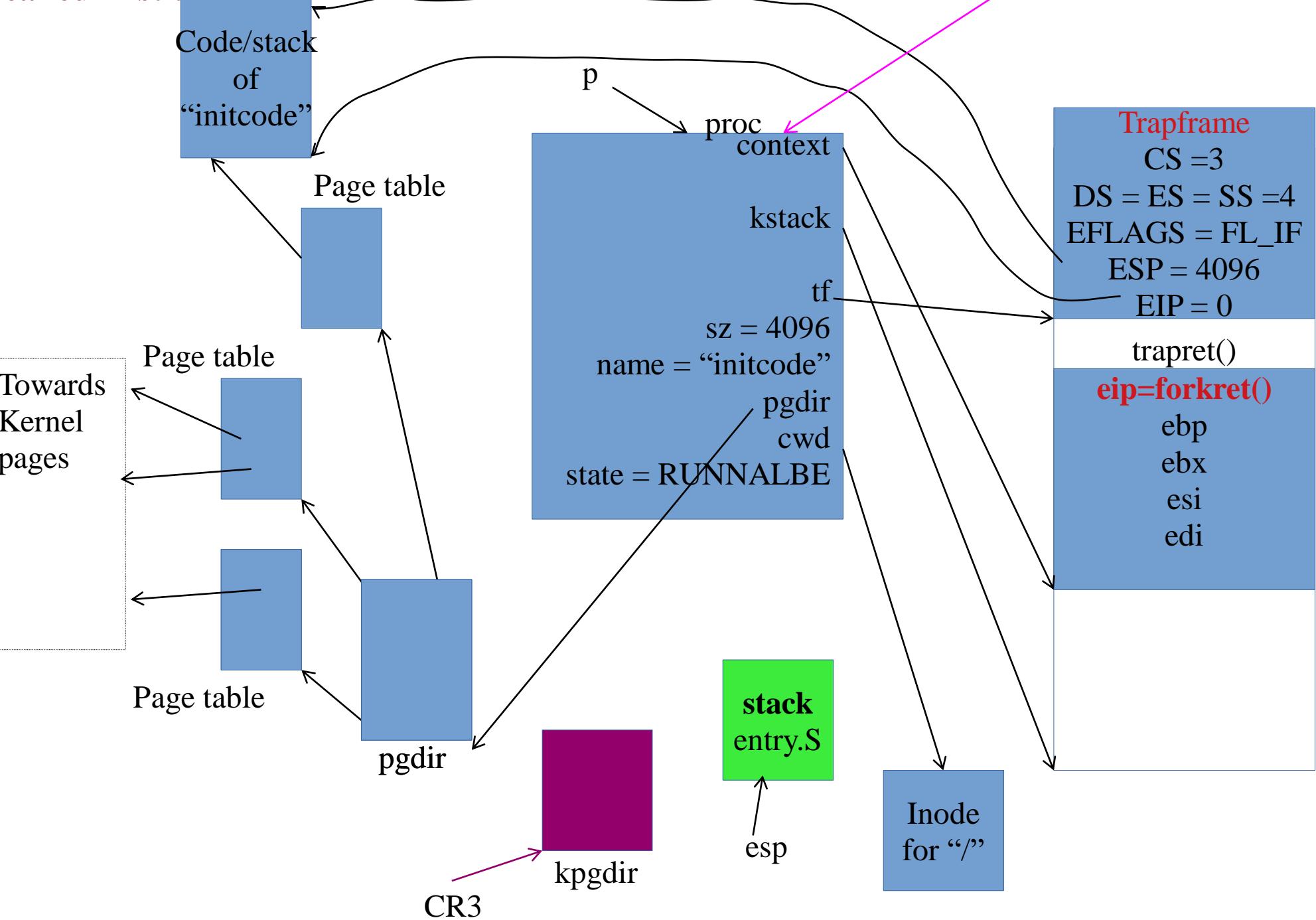
```
            continue;
```

```
        // Switch to chosen process. It is the process's job
```

```
        // to release ptable.lock and then reacquire it
```

```
        // before jumping back to us.
```

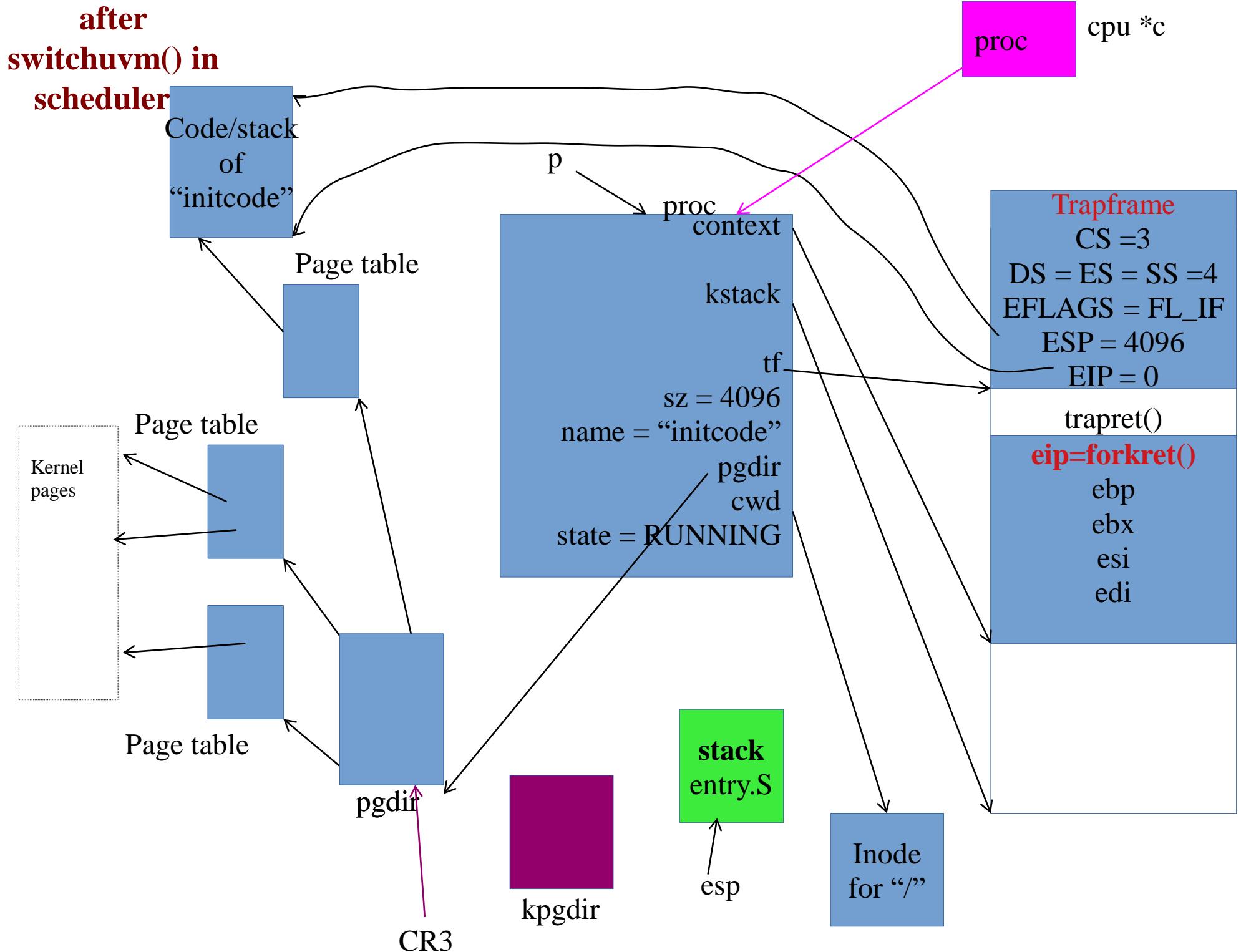
scheduler()
called first time



```
acquire(&ptable.lock); scheduler()
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;
    switchuvvm(p);
    p->state = RUNNING;
```



```
acquire(&ptable.lock);

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

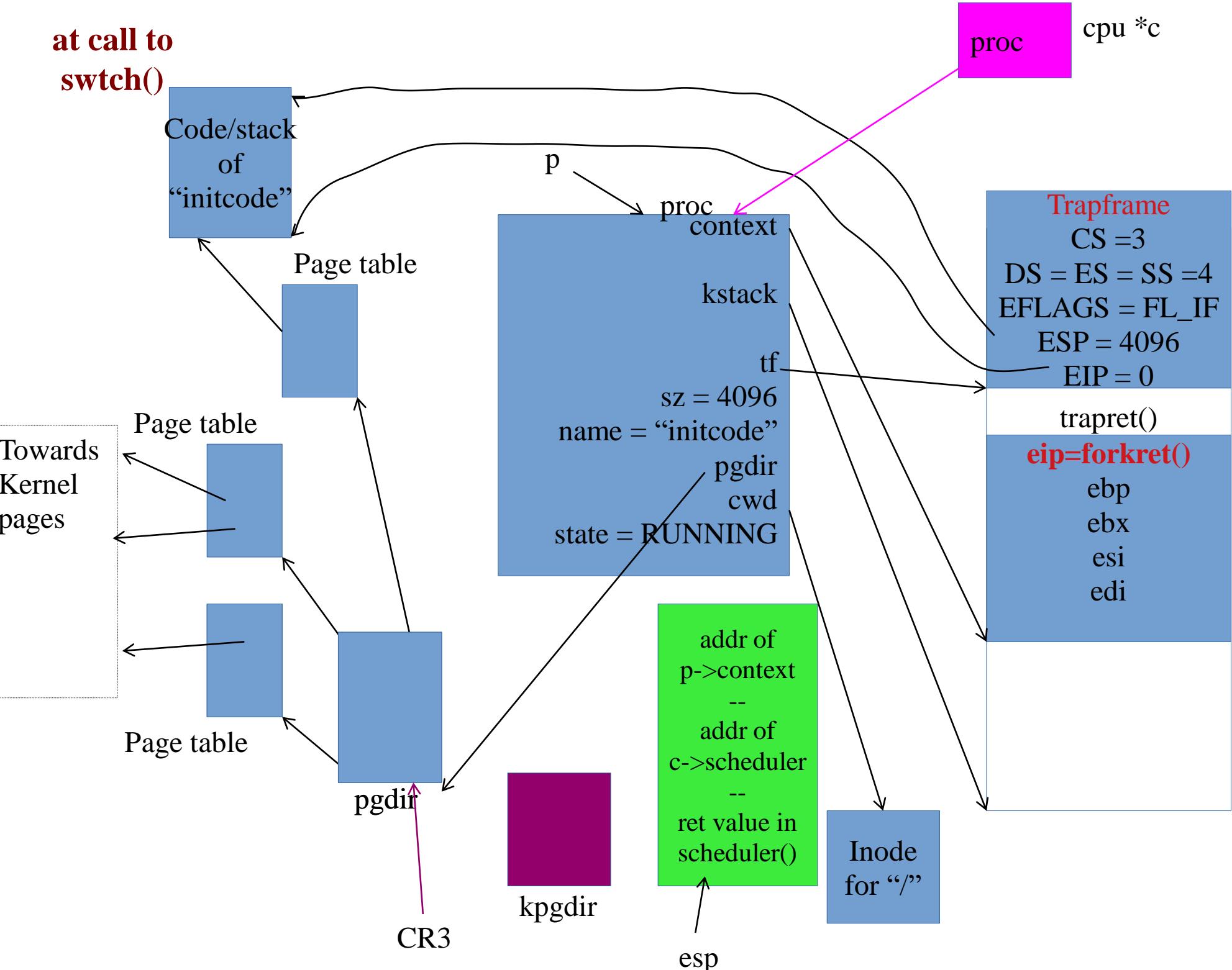
    // Switch to chosen process. It is the process's job
    // to release ptable.lock and then reacquire it
    // before jumping back to us.

    c->proc = p;
    switchuvvm(p);

    p->state = RUNNING
    swtch(&(c->scheduler), p->context);

};
```

at call to
swtch()



swtch

swtch:

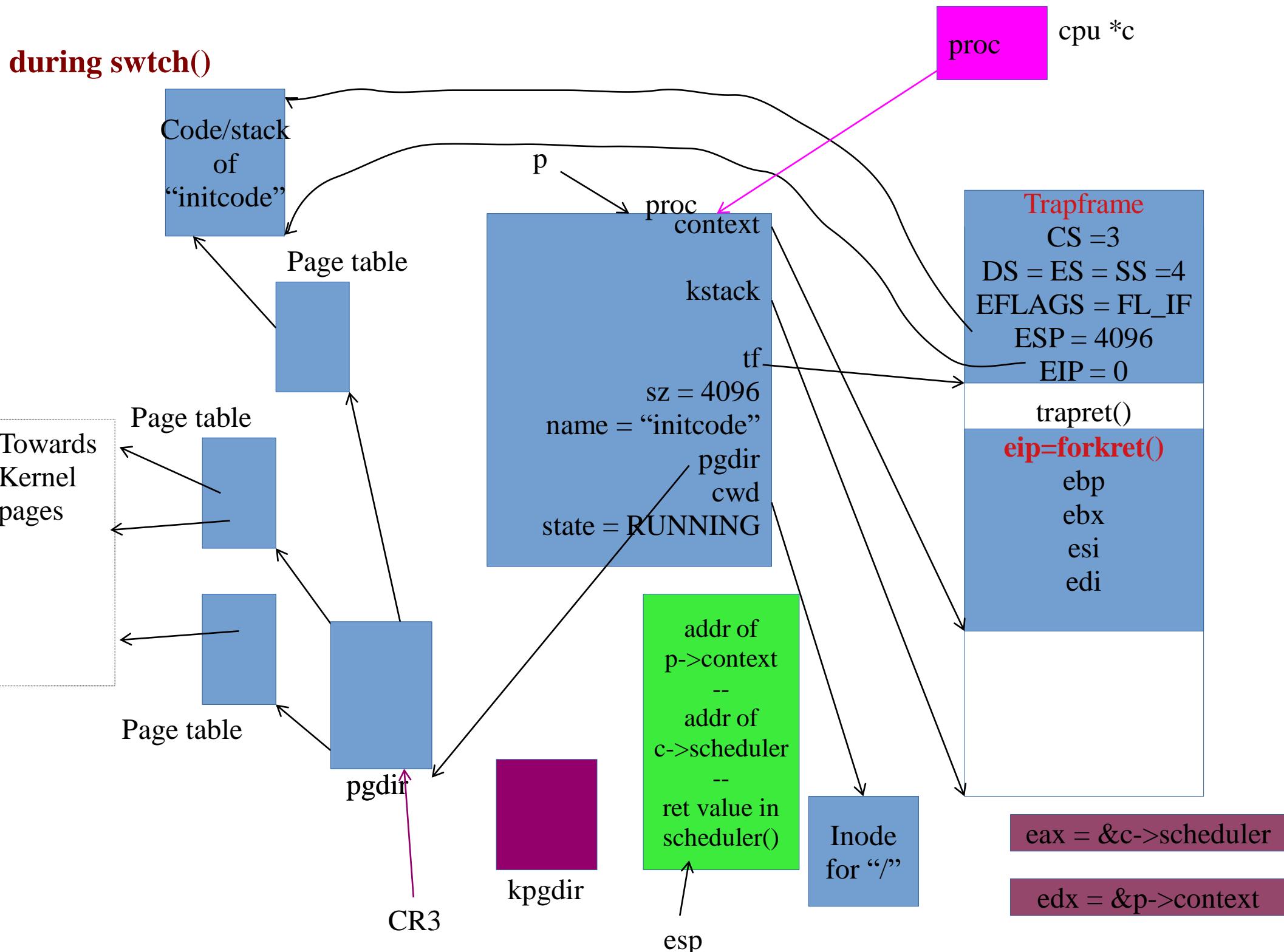
#Abhijit: swtch was called through a function call.

#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

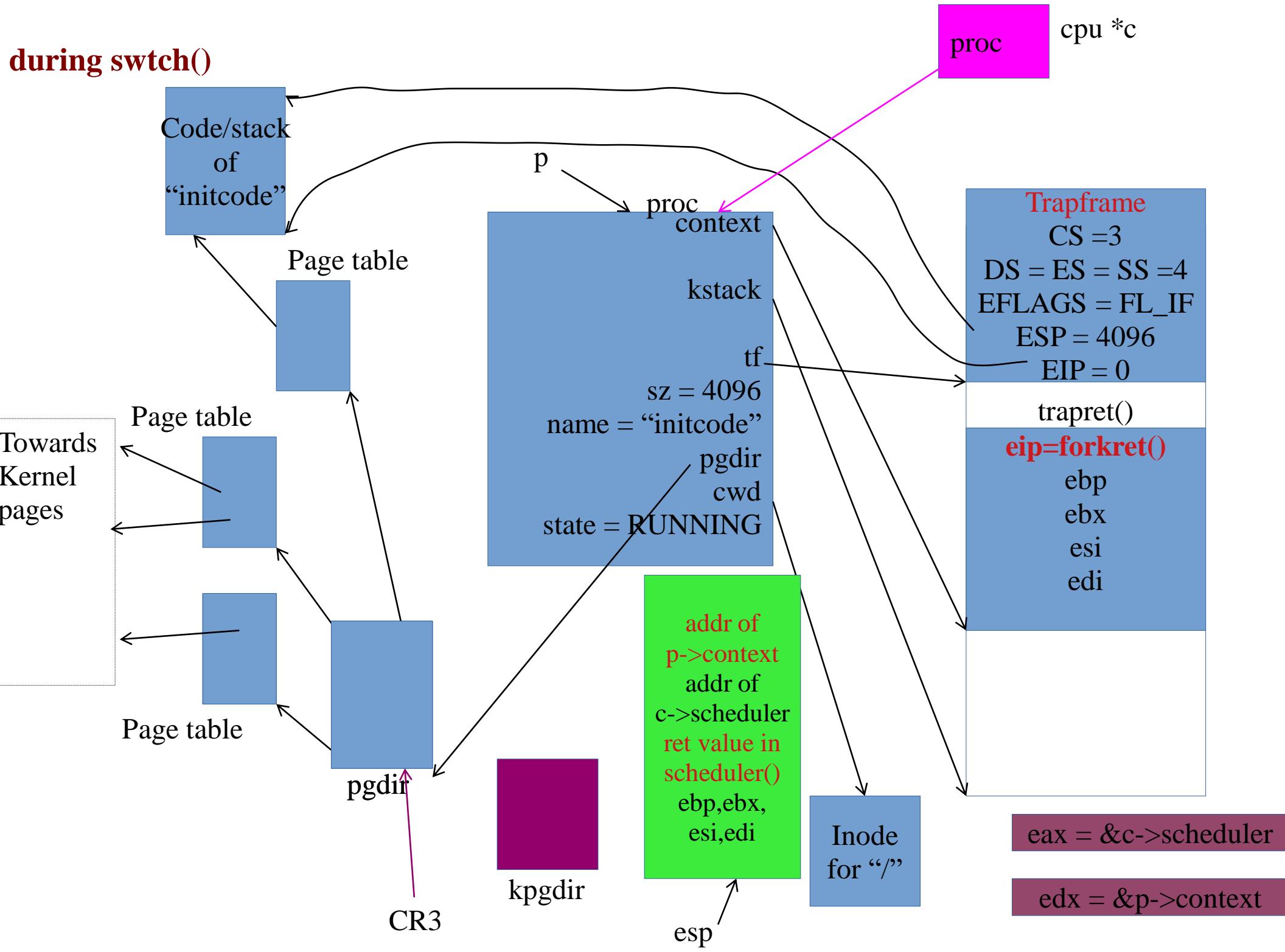
pushl %ebp

pushl %ebx

pushl %esi

pushl %edi # Abhijit: esp = esp + 16

during swtch()



swtch:

#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

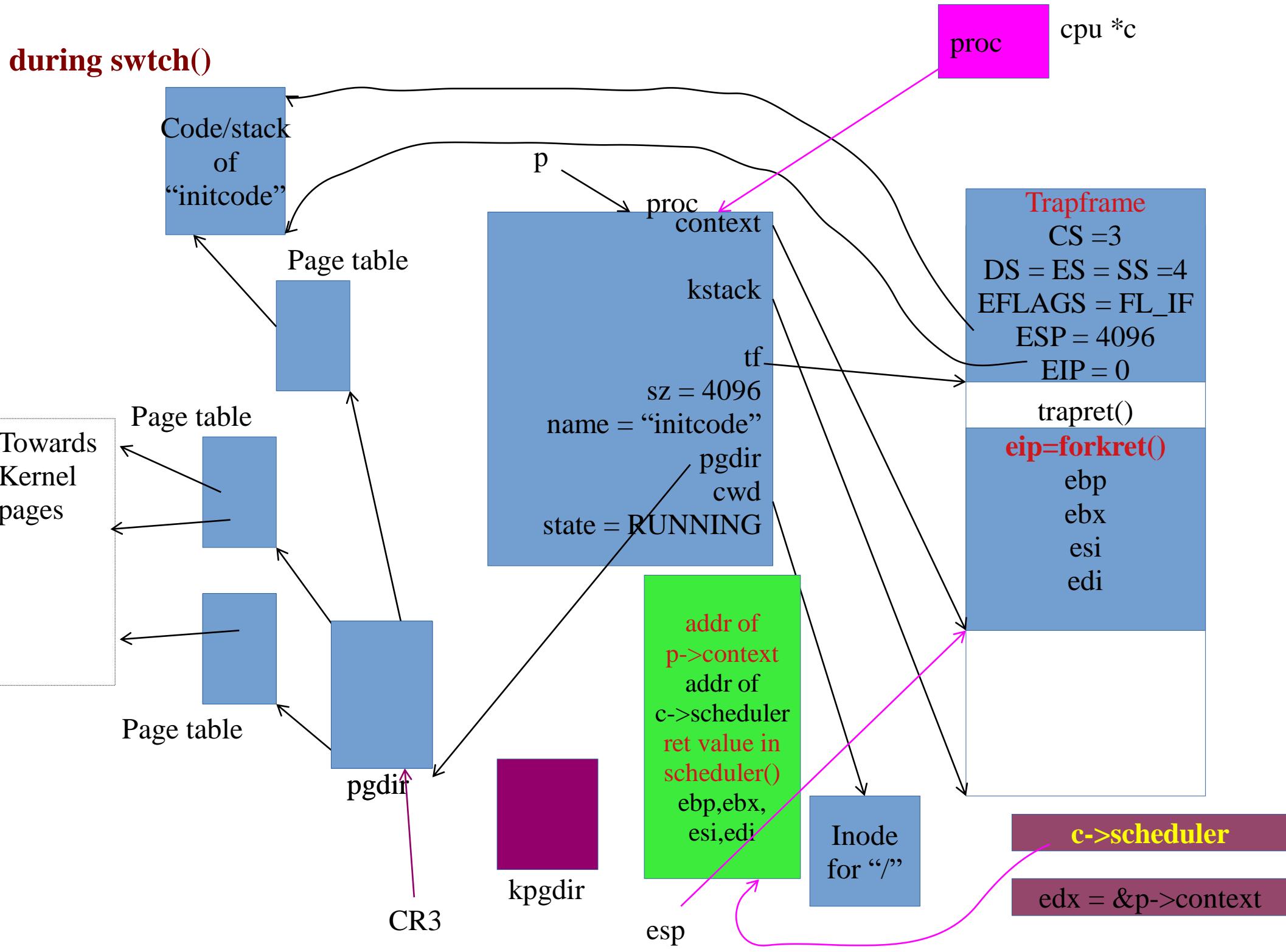
pushl %edi # Abhijit: esp = esp + 16

Switch stacks

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

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

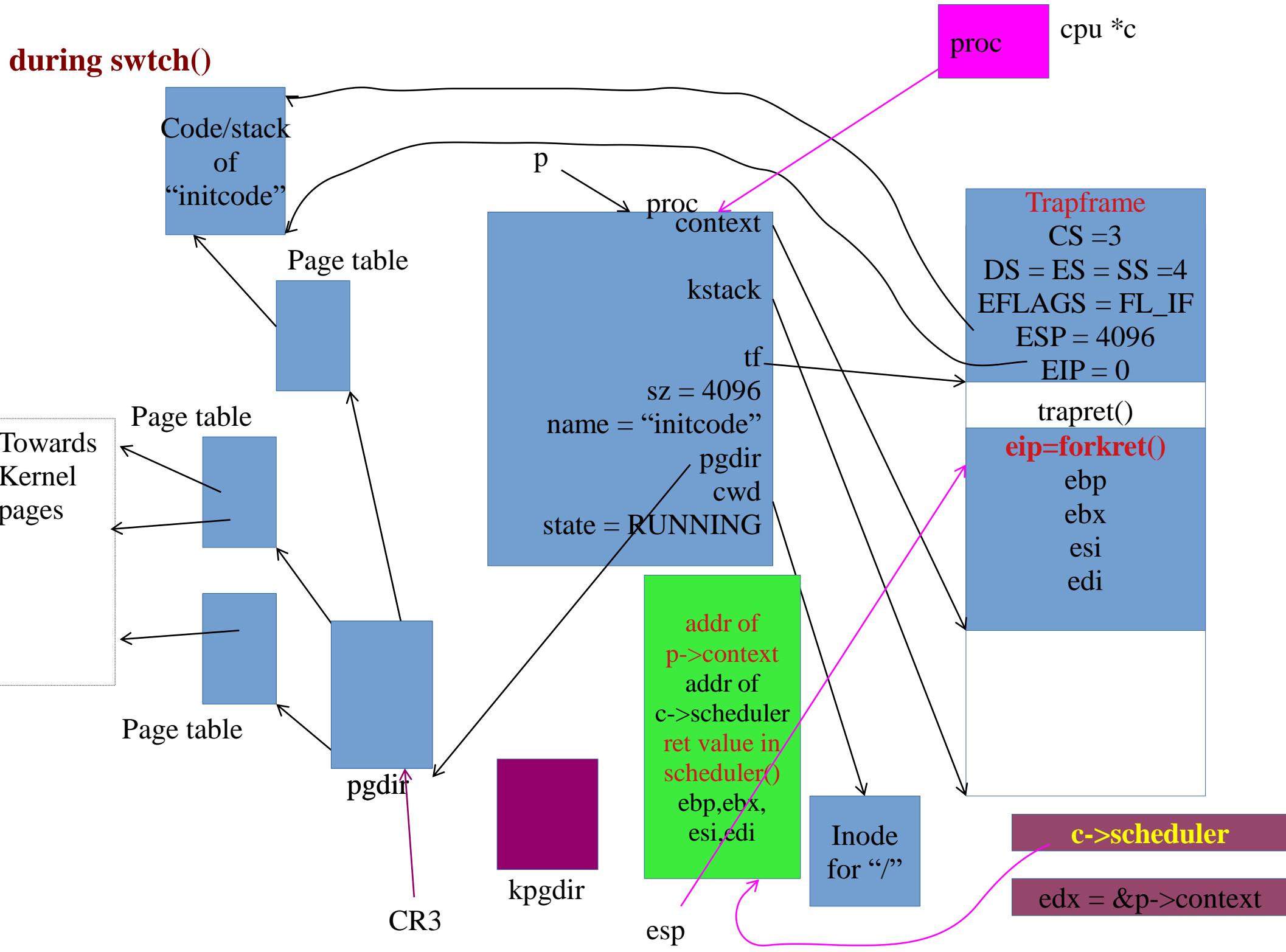
during swtch()



```
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
```

swtch

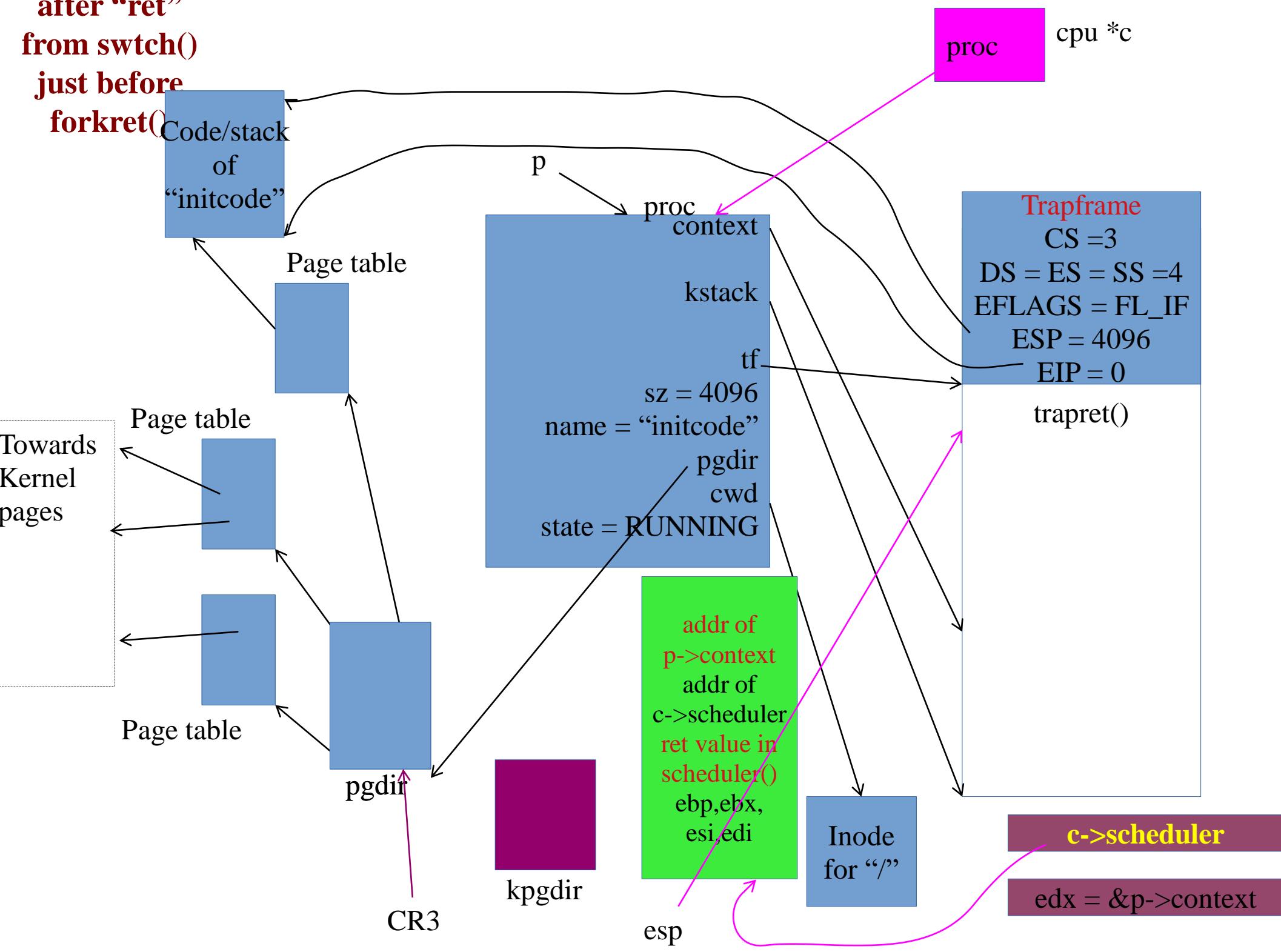
during swtch()



```
movl 8(%esp), %edx    # Abhijit: edx = new  
# Save old callee-saved registers  
swtch  
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()

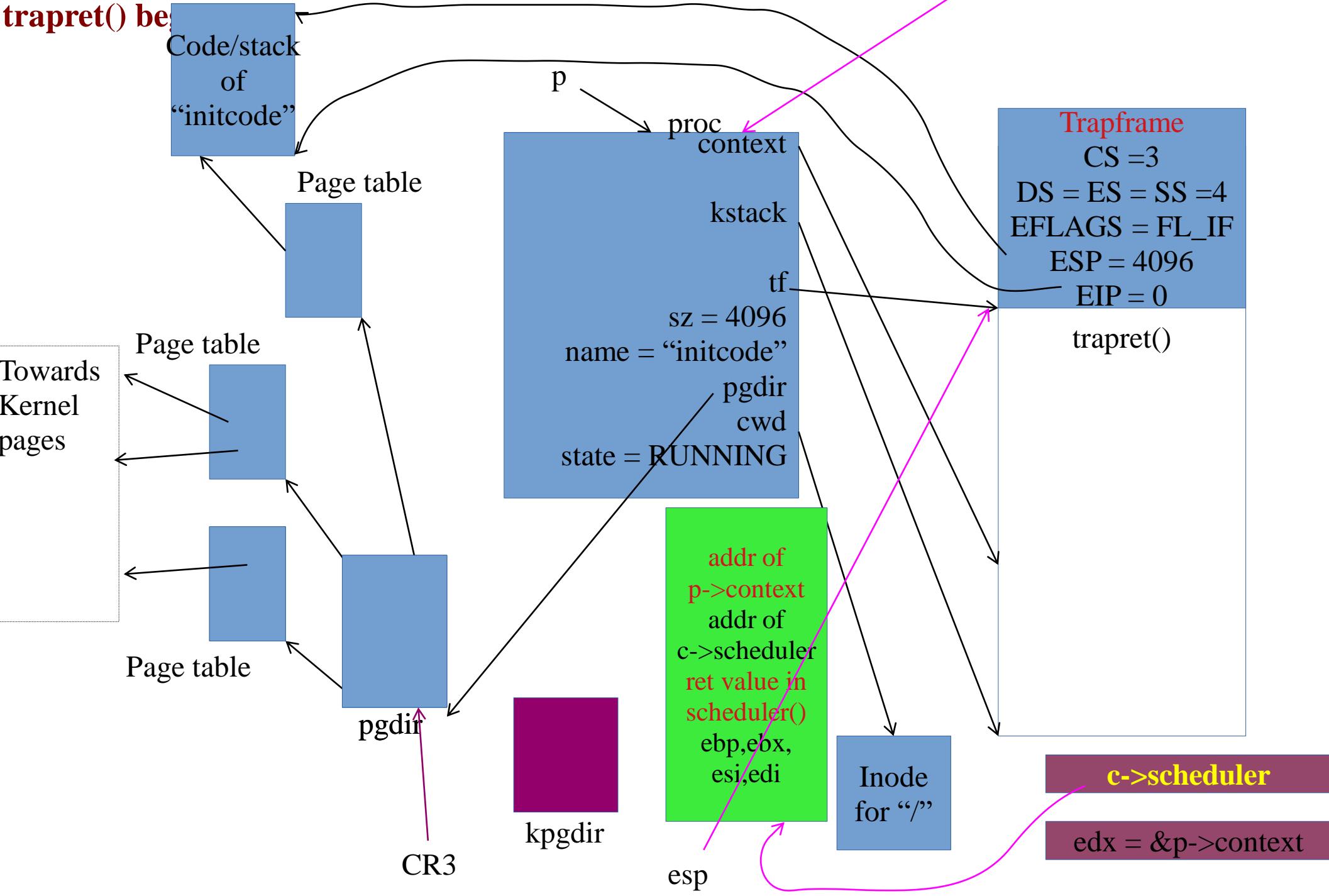
just before
`forkret()`



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() be

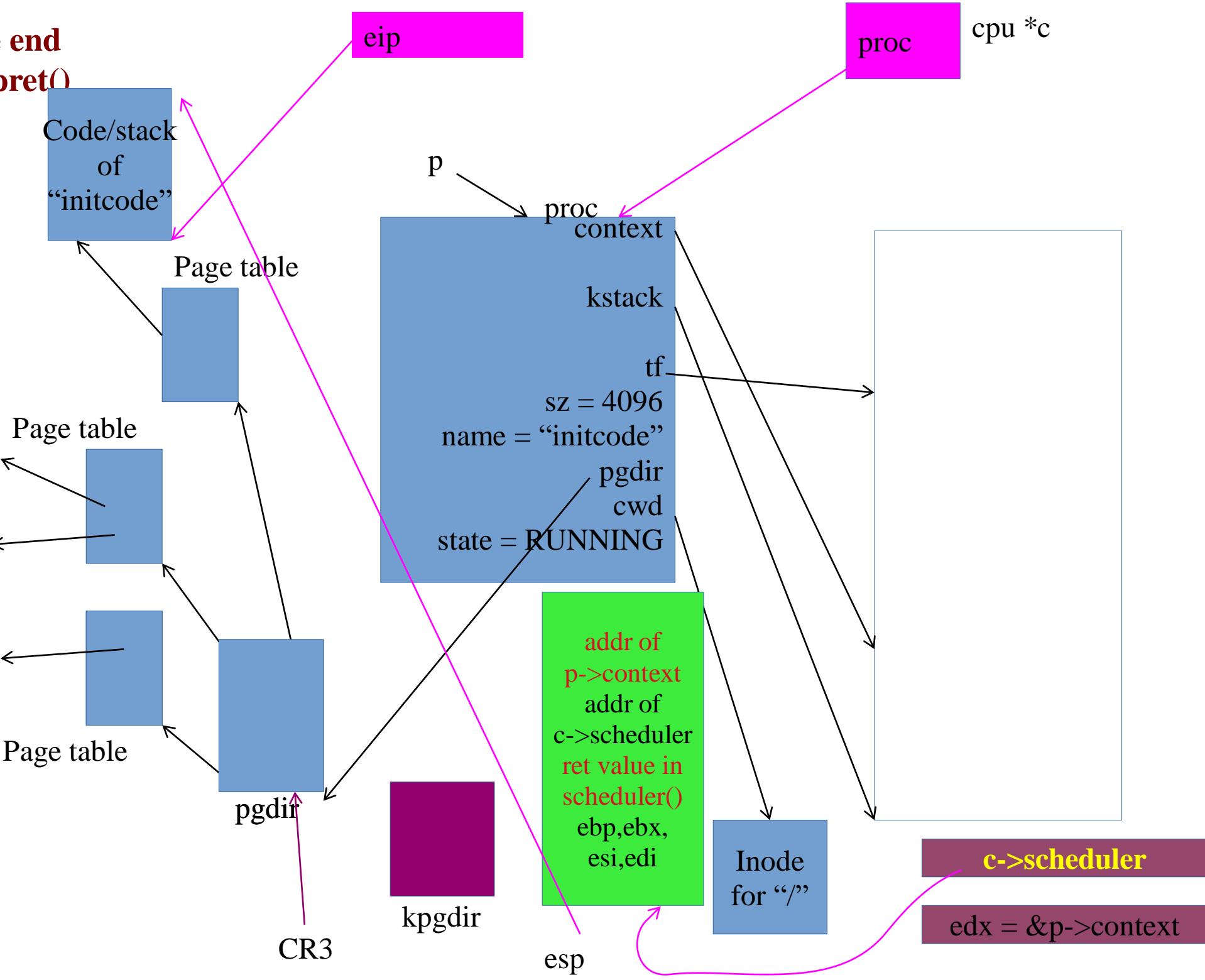


After iret in trapret

- 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()

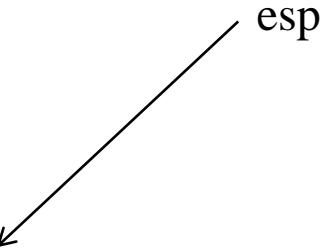
Towards
Kernel
pages



initcode

```
# char init[] = "/init\0";      start:  
  
init:                                pushl $argv  
  
.string "/init\0"                      pushl $init  
  
# char *argv[] = { init, 0             pushl $0 // where caller  
};                                     pc would be  
  
.p2align 2                            movl $SYS_exec, %eax  
  
argv:                                  int $T_SYSCALL  
  
.long init                           # for(;;) exit();
```

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



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>

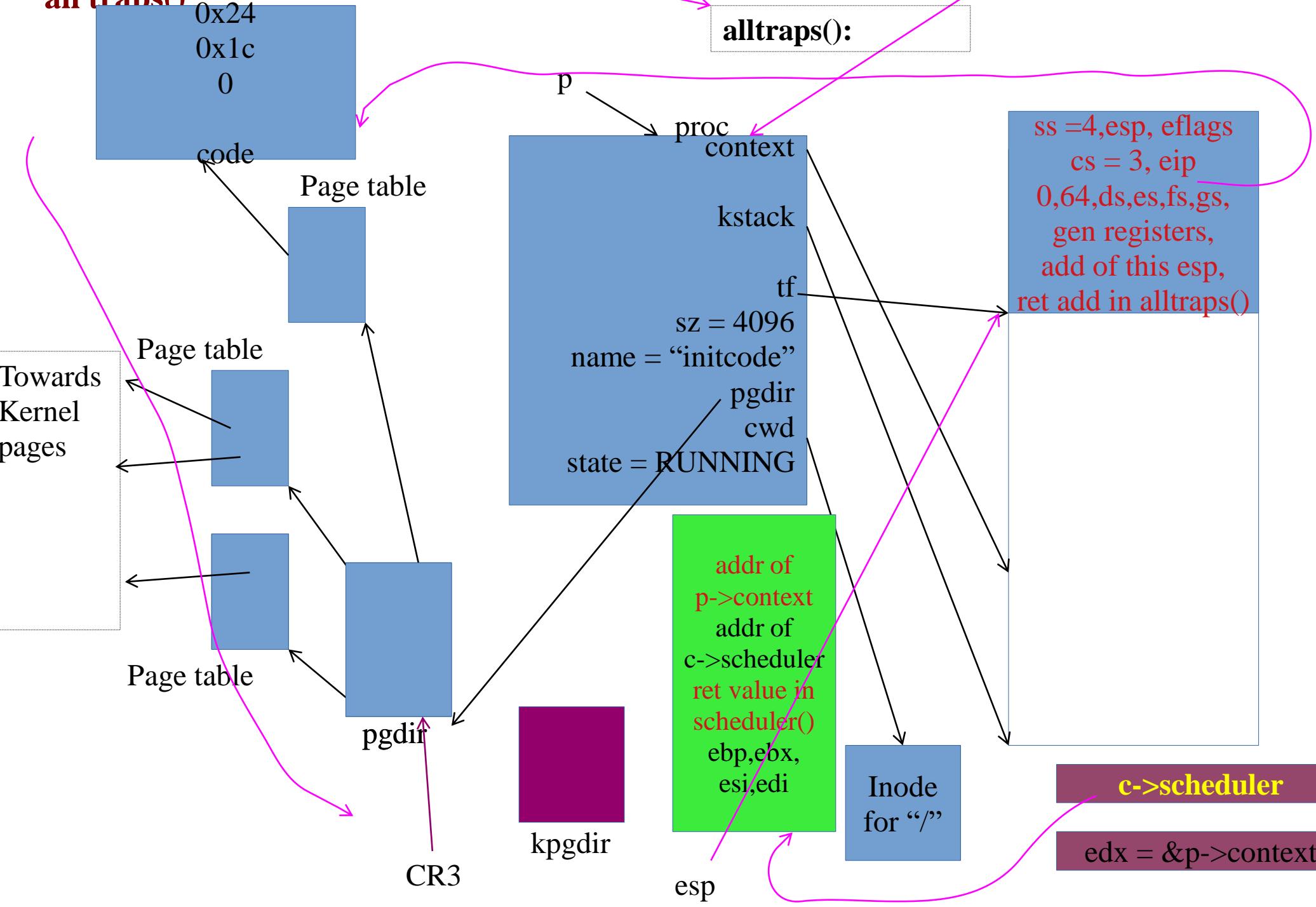
0000001c <init>:

"/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

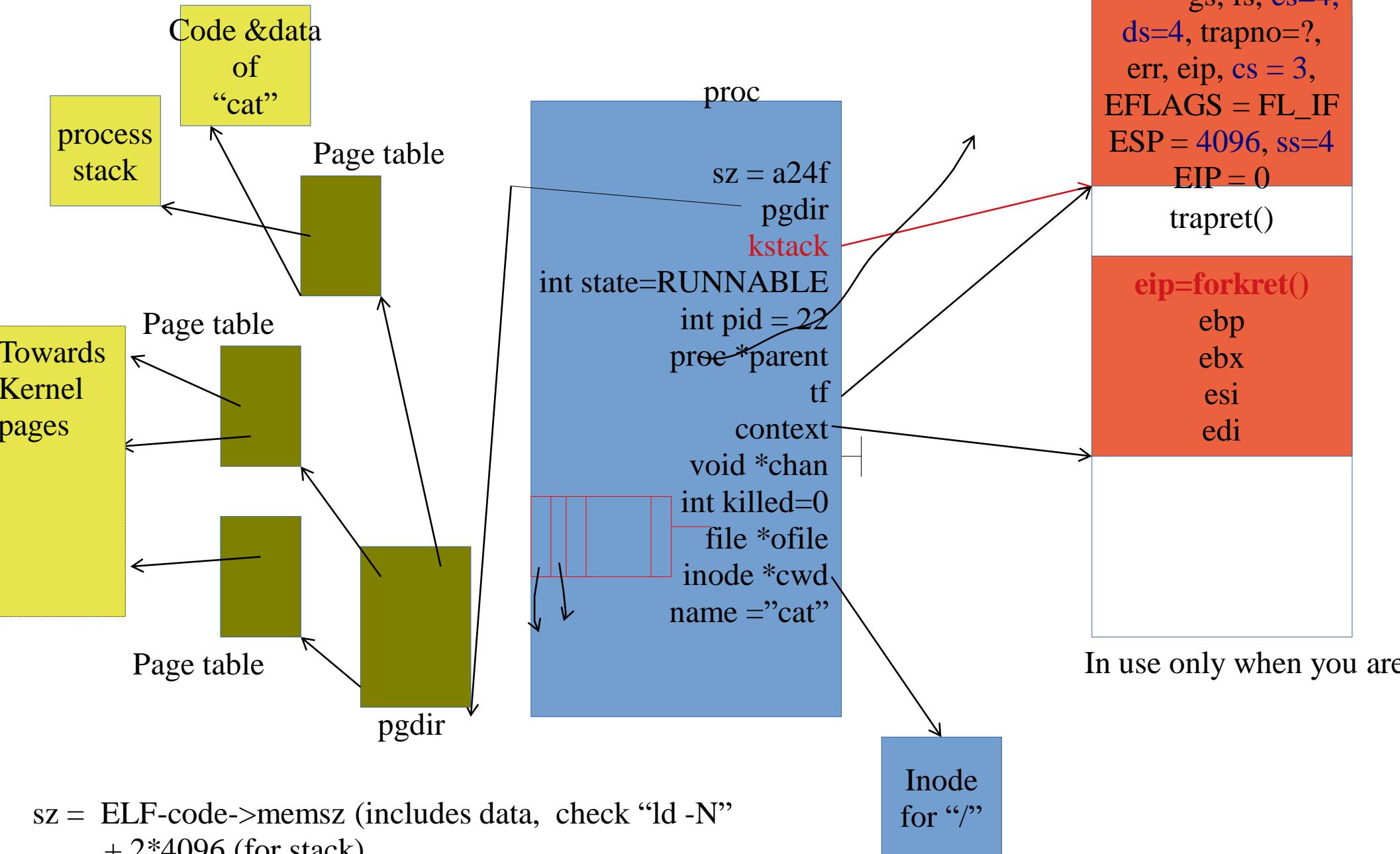
 kernel stack for this process

 enum procstate state;

 // Process state.

 allocated, ready to run, running, wait-

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**
- 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**

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

```
# 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
```

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();
```

trap()

- Argument is trapframe

- In alltraps

- Before “call trap”, there was “push %esp” and stack had the trapframe

- Remember calling convention --> when a function is called, the stack contains the arguments in reverse order (here only 1 arg)

trap()

- Has a switch
- **switch(tf->trapno)**
- Q: who set this trapno?
- Depending on the type of trap
- Call interrupt handler
 - Timer
 - wakeup(&ticks)
 - IDE: disk interrupt
 - Ideintr()
 - KBD
 - Kbdintr()
 - COM1
 - Uatrintr()
 - If Timer

when trap() returns

❑#Back in alltraps

call trap

addl \$4, %esp

Return falls through
to trapret...

.globl trapret

trapret:

popal

❑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

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()

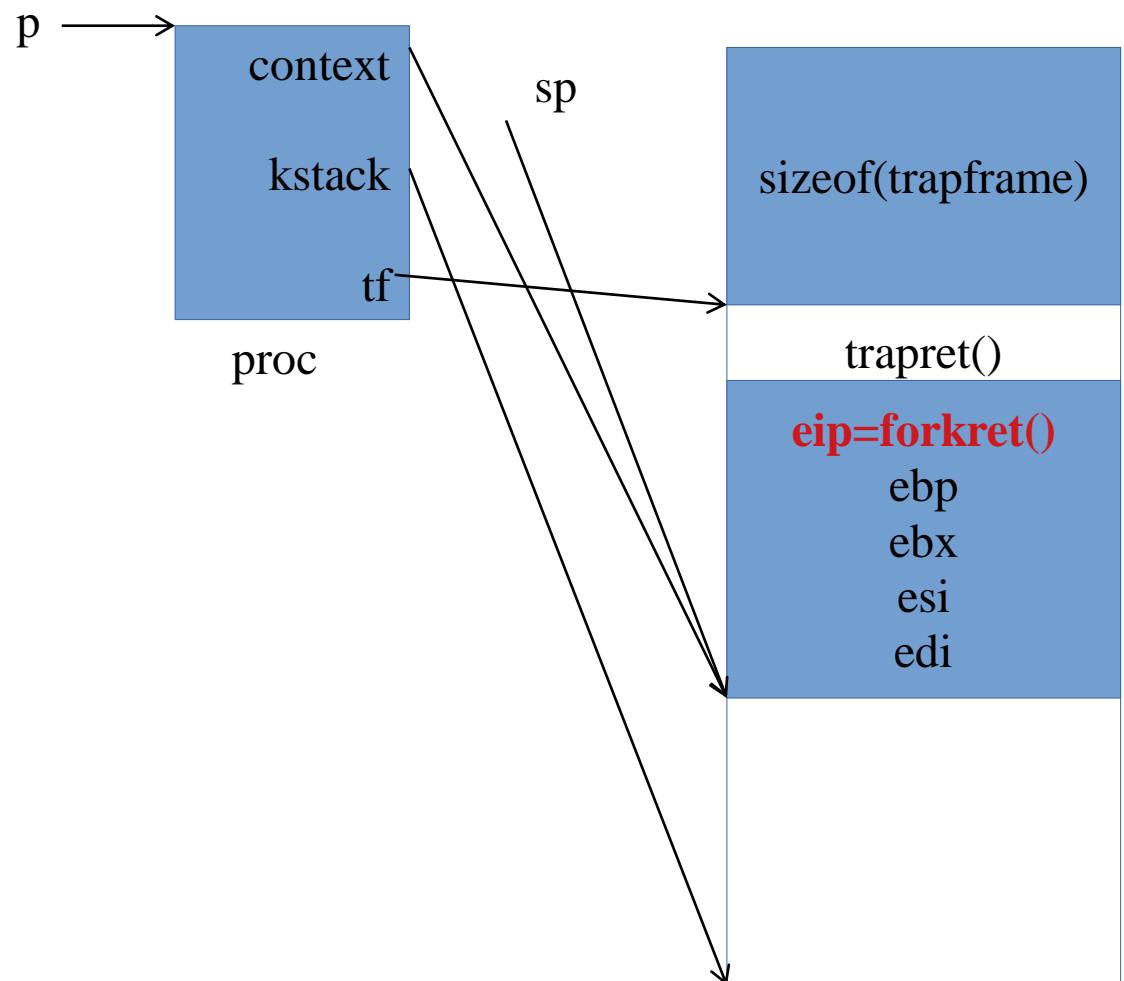
```
int sys_fork(void)
{
    return fork();
}

int
fork(void)
{
    int i, pid;
    struct proc *np;
    struct proc *curproc =
    myproc();

    // Allocate process.
```

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;
```

- .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*

copyuvm(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 =

understanding
fork()->copyuvm()

- .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] =  
fileup(curproc-  
>ofile[i]);  
  
np->cwd =  
idup(curproc->cwd);  
  
safestrcpy(np->name,
```

- 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 UP

```
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;
    }
}
```

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));
```

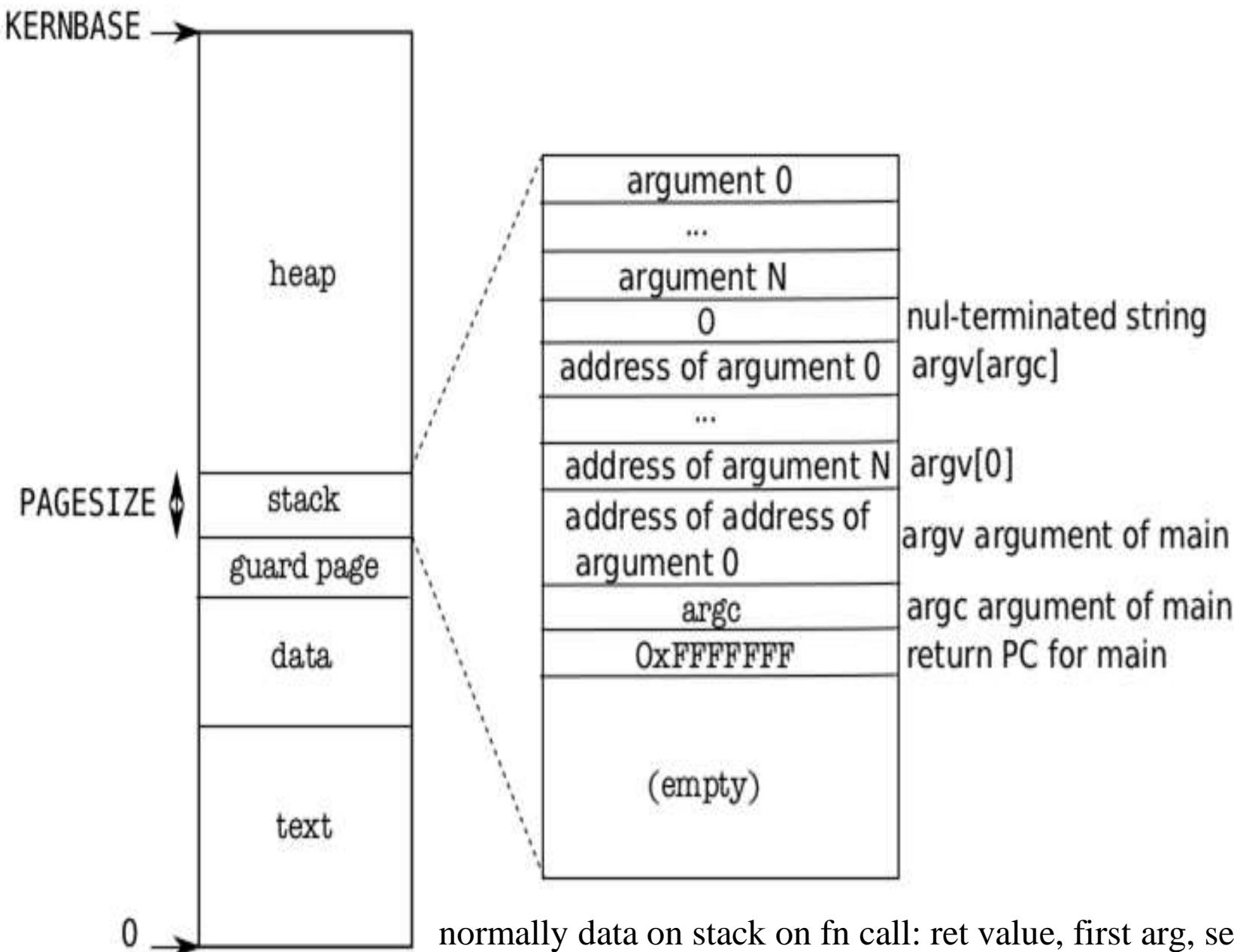
sys_exec()

- 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

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

User stack after
call
to exec()
is over



normally data on stack on fn call: ret value, first arg, second arg, ...
main(int argc, char *argv[])
argv[] is address of array of string; string itself is an address. Hence 2 levels

exec()

```
int  
exec(char *path, char  
**argv)  
{  
...  
    uint argc, sz, sp,  
    ustack[3+MAXARG+1];  
...  
    if((ip = namei(path)) ==  
0){
```

- 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;
```

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

```
if((nogdir = setunkvm())
```

```
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;
```

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.

exec()

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

if(argc >= MAXARG)

goto bad;

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

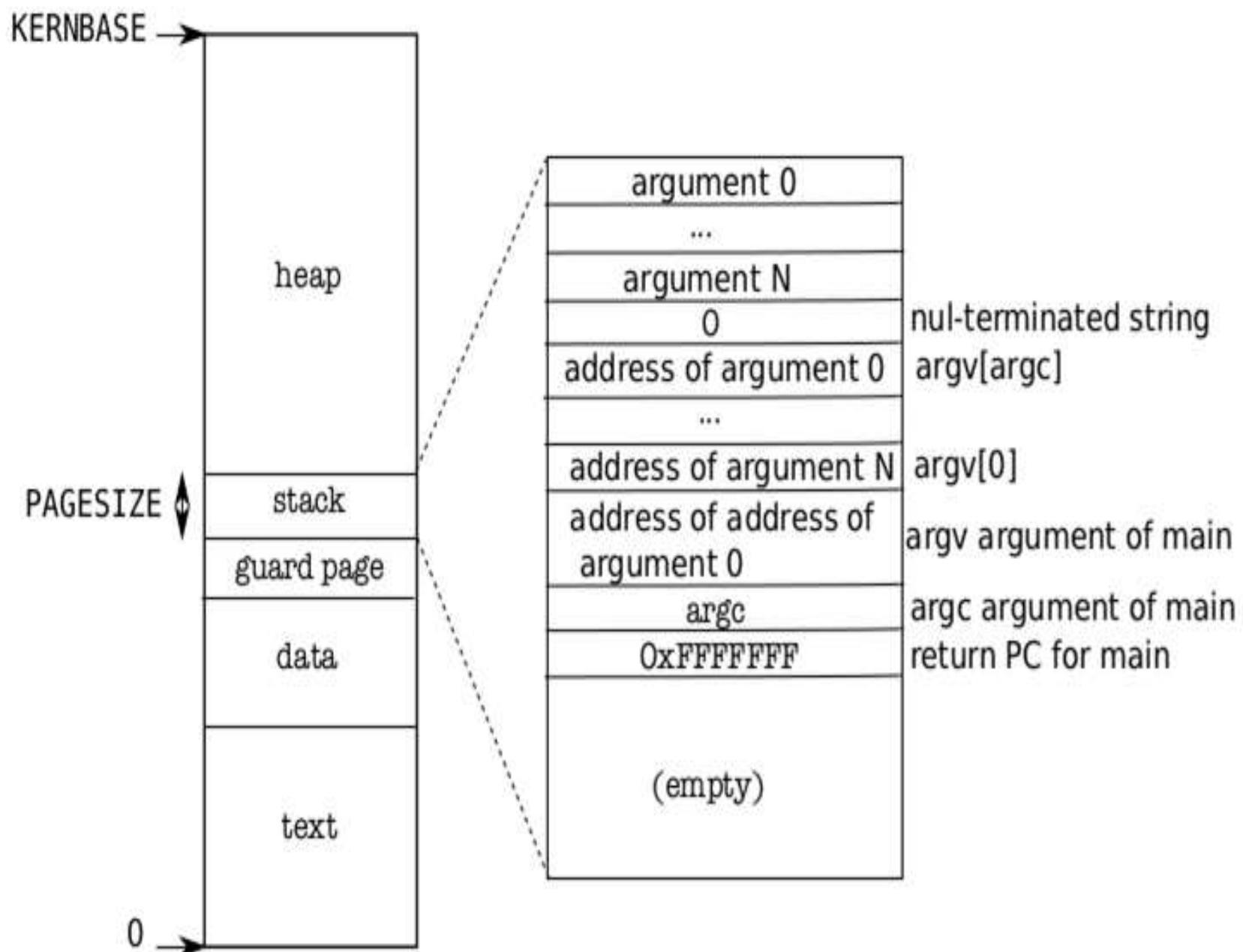
if(copyout(pkdir, sp,
argv[argc], strlen(argv[argc]) +
1) < 0)

goto bad;

ustack[3+argc] = sp;

- 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 earl



// Save program name for
debugging.

exec()

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;

- 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

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 !

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

Process Control Block

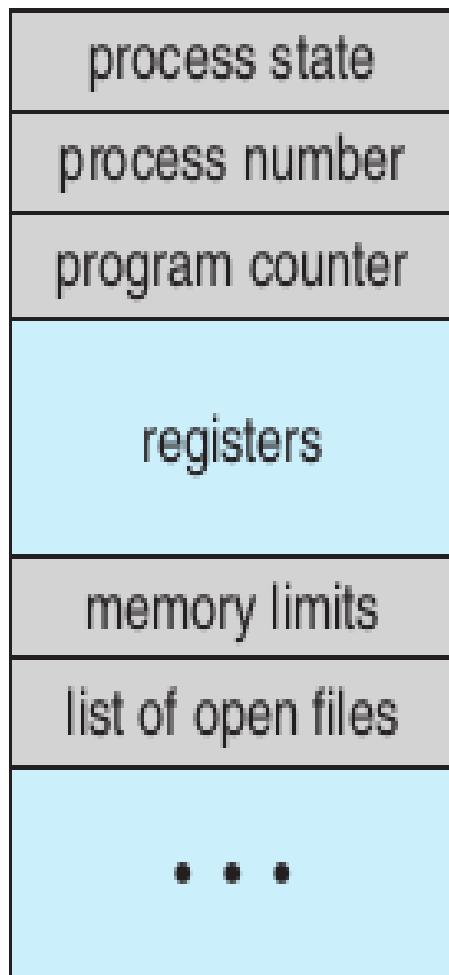


Figure 3.3 Process control block (PCB).

- A record representing a process in operating system's data structures
- OS maintains a “list” of PCBs, one for each process
- Called “struct

Fields in PCB



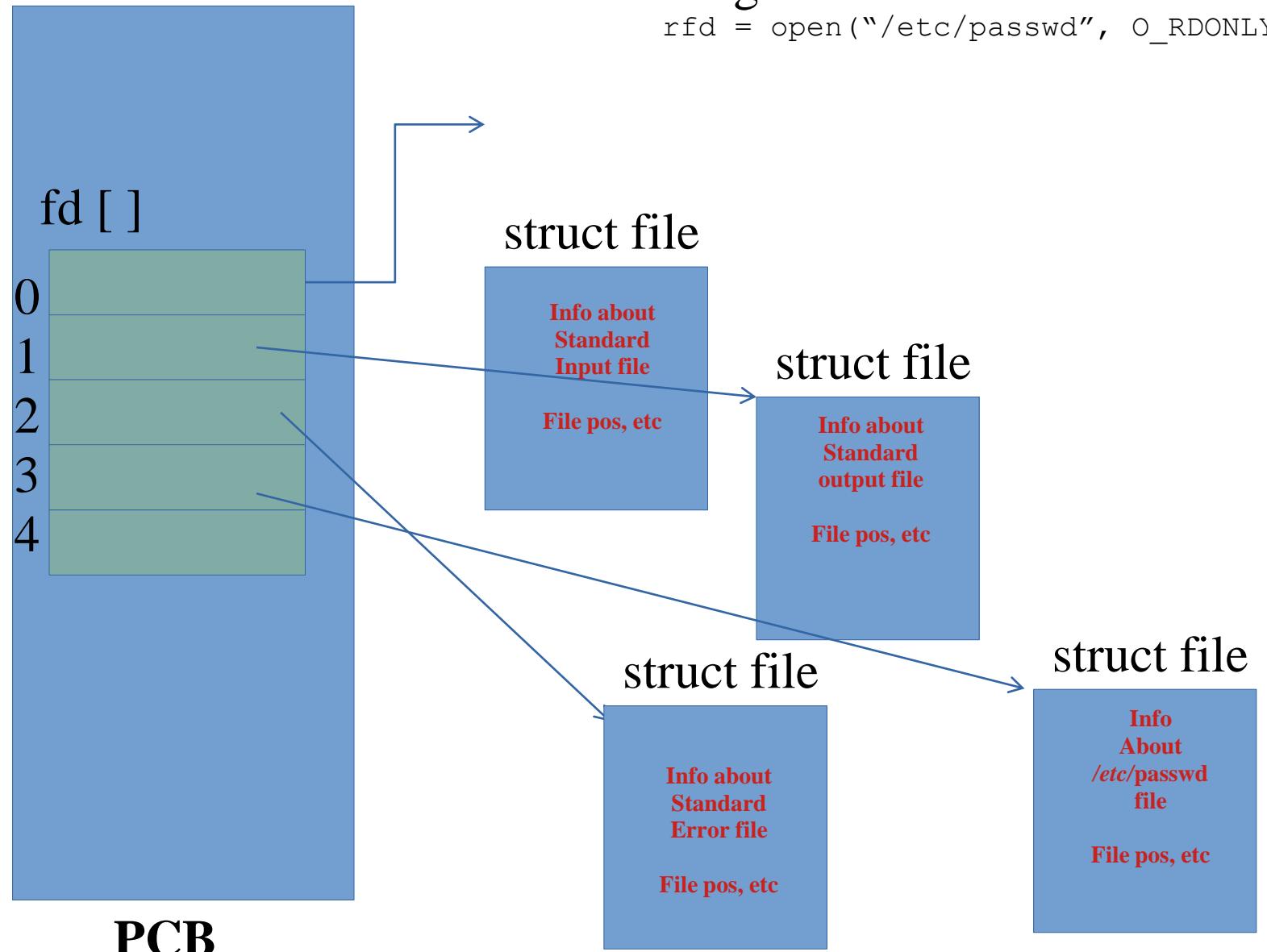
Figure 3.3 Process control block (PCB).

- Process ID (PID)
- Process State
- Program counter
- Registers
- Memory limits of the process
- Accounting information
- I/O status

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

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

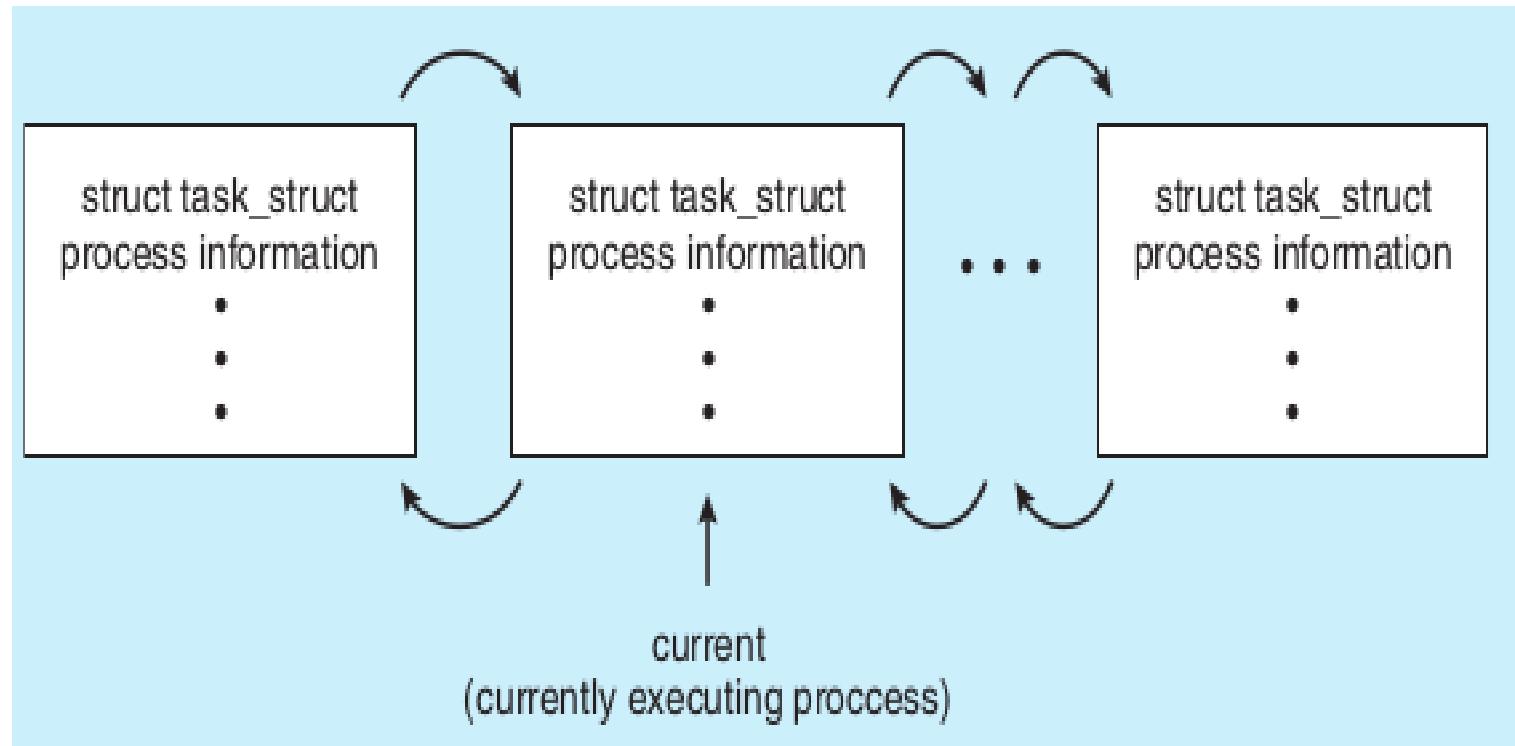
struct proc {
    uint sz;           // Size of process memory (bytes)
    pde_t* pgdir;     // Page table
    char *kstack;     // Bottom of kernel stack for this process
    enum procstate state; // Process state
    int pid;          // Process ID
    struct proc *parent; // Parent process
    struct 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[NOFILE]; // 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 };
    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



// 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;
};
```

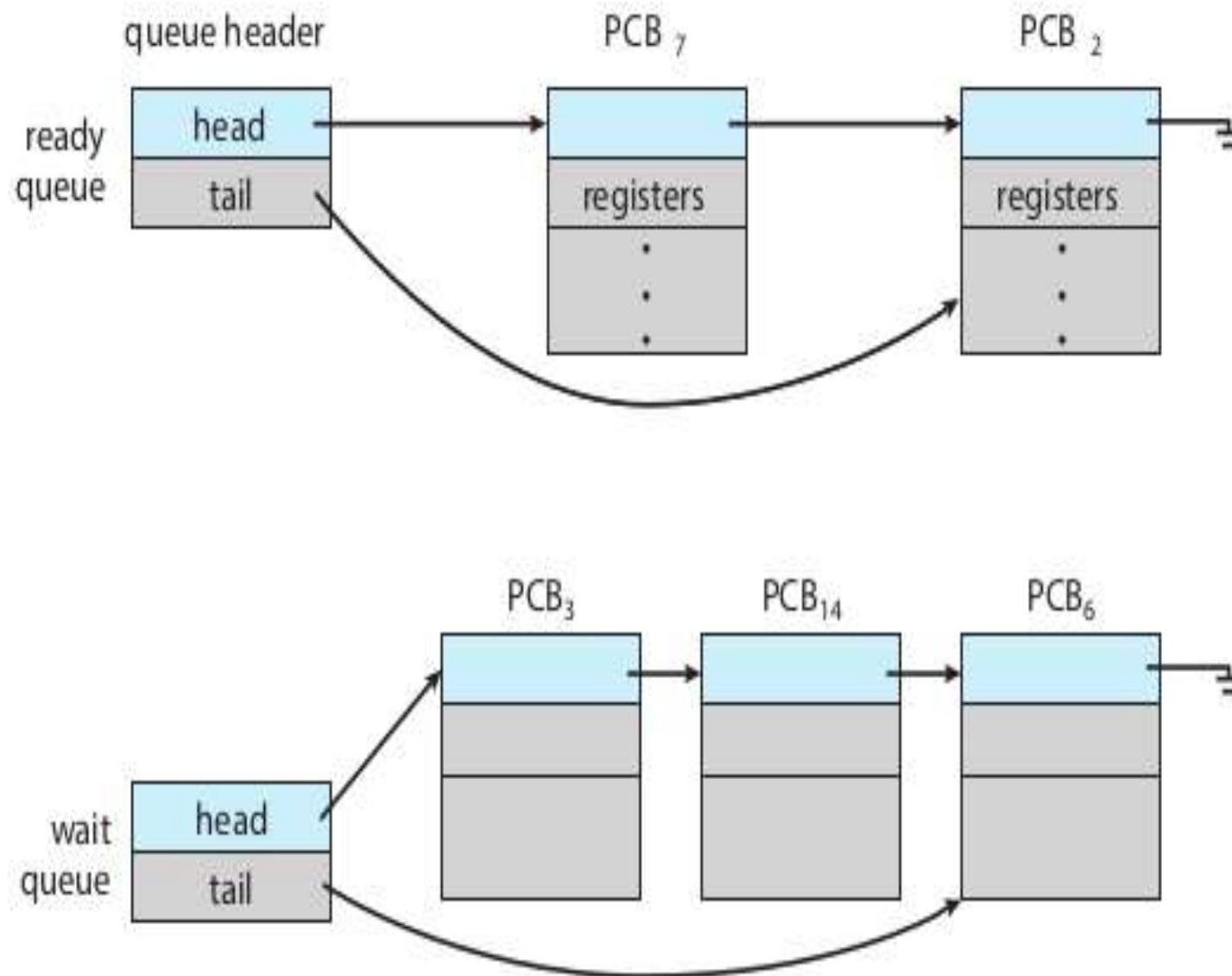


Figure 3.4 The ready queue and wait queues.

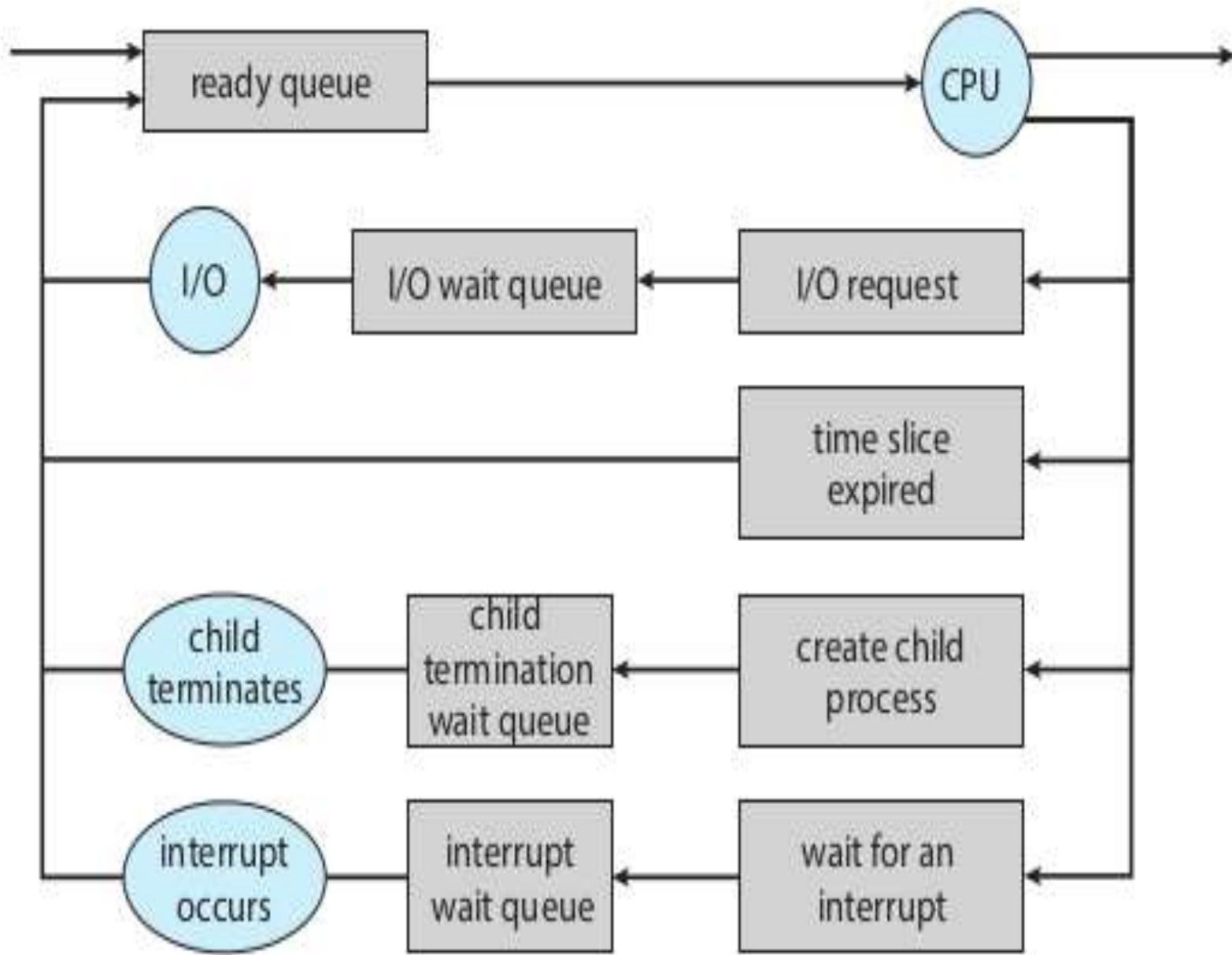


Figure 3.5 Queueing-diagram representation of process scheduling.

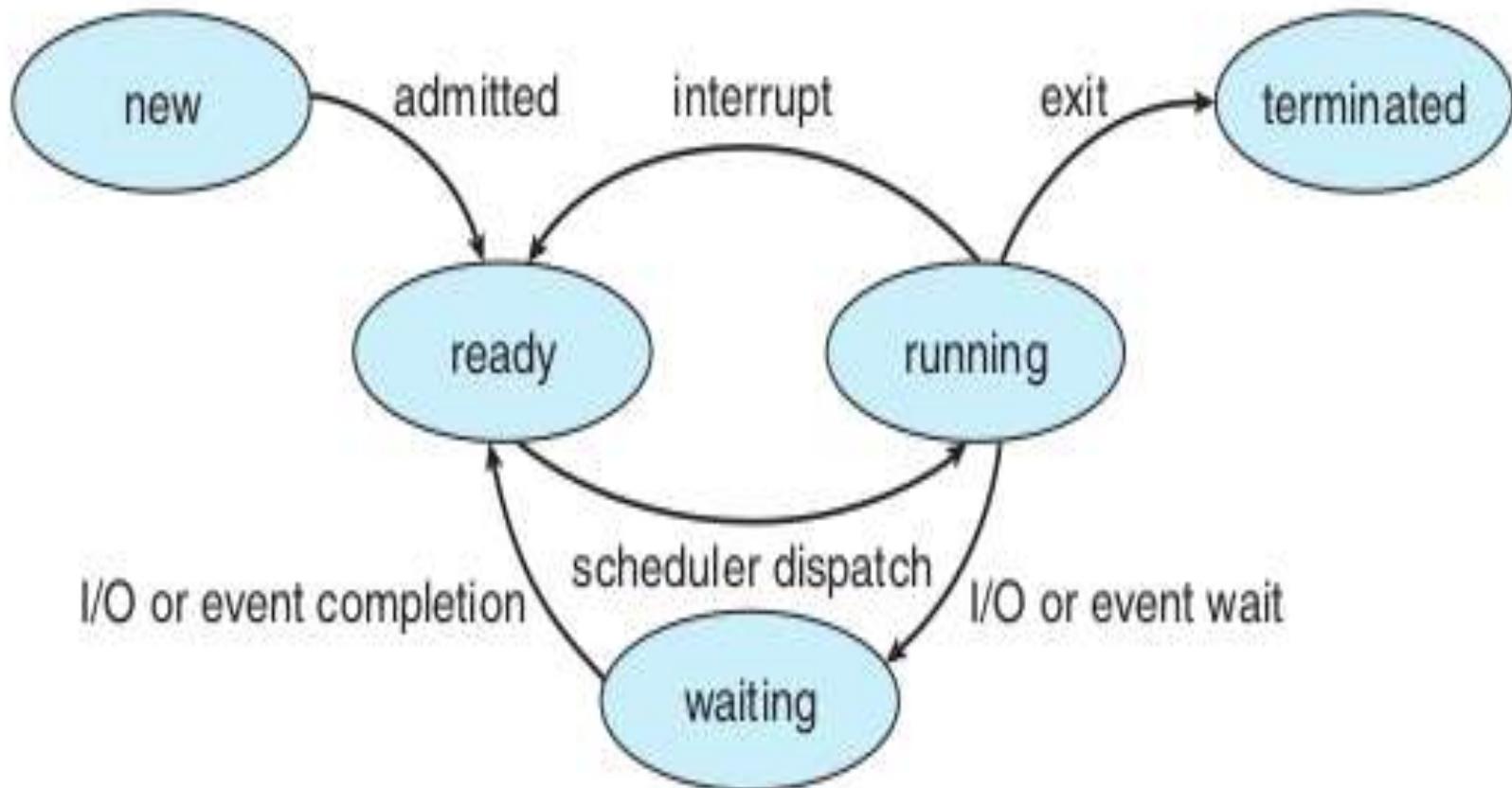


Figure 3.2 Diagram of process state.

Conceptual diagram

“Giving up” CPU by a process or blocking

```
int main() {  
    i = j + k;  
    scanf("%d", &k);  
}  
  
int scanf(char *x,  
...) {  
    ...  
}
```

OS Syscall

```
sys_read(int fd,  
char *buf, int len) {  
    file f = current-  
        >fdarray[fd];  
    int offset = f-  
        >position;  
    ...  
}
```

“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

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

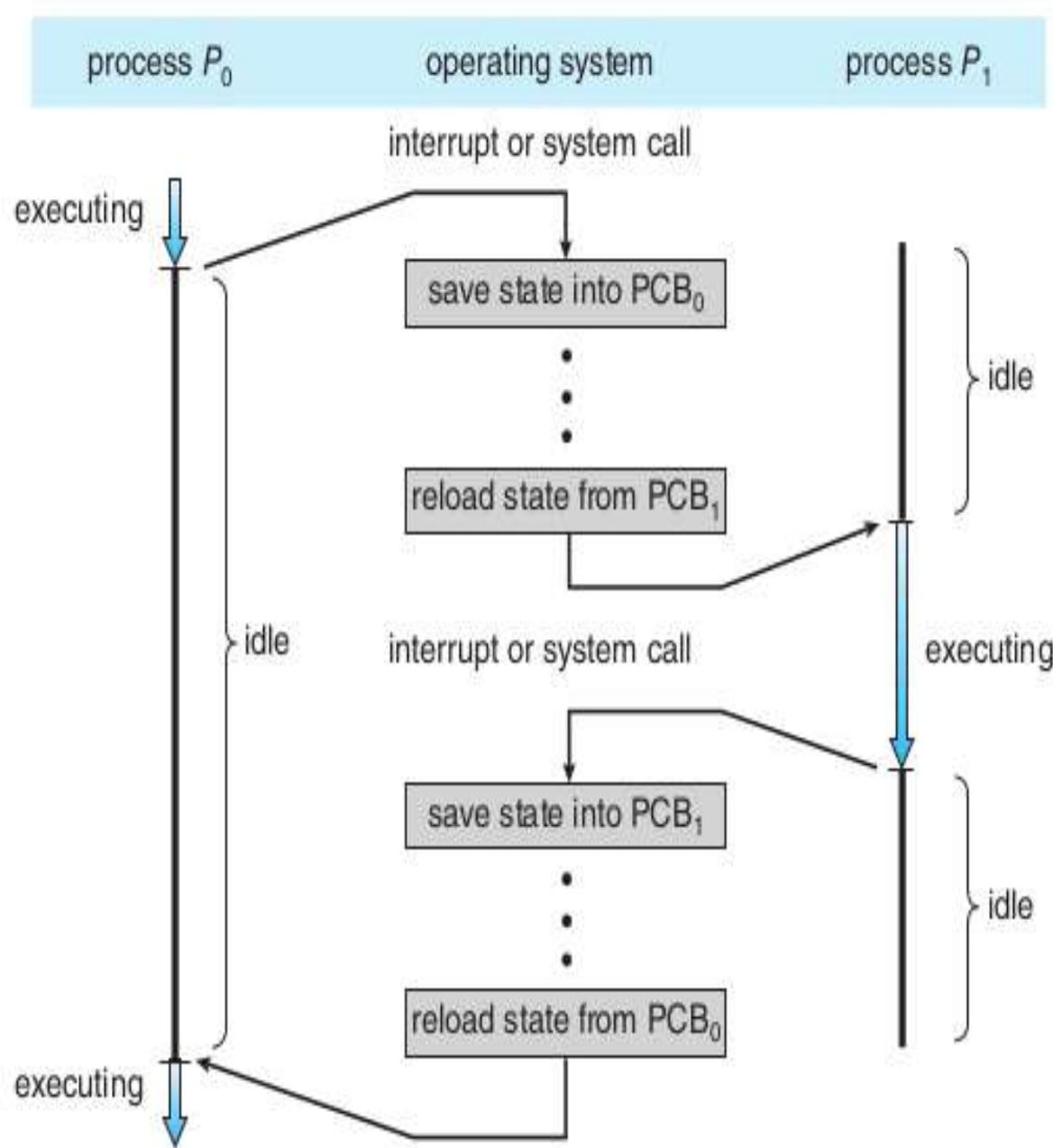


Figure 3.6 Diagram showing context switch from process to process.

Pecularity 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 a context switch

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

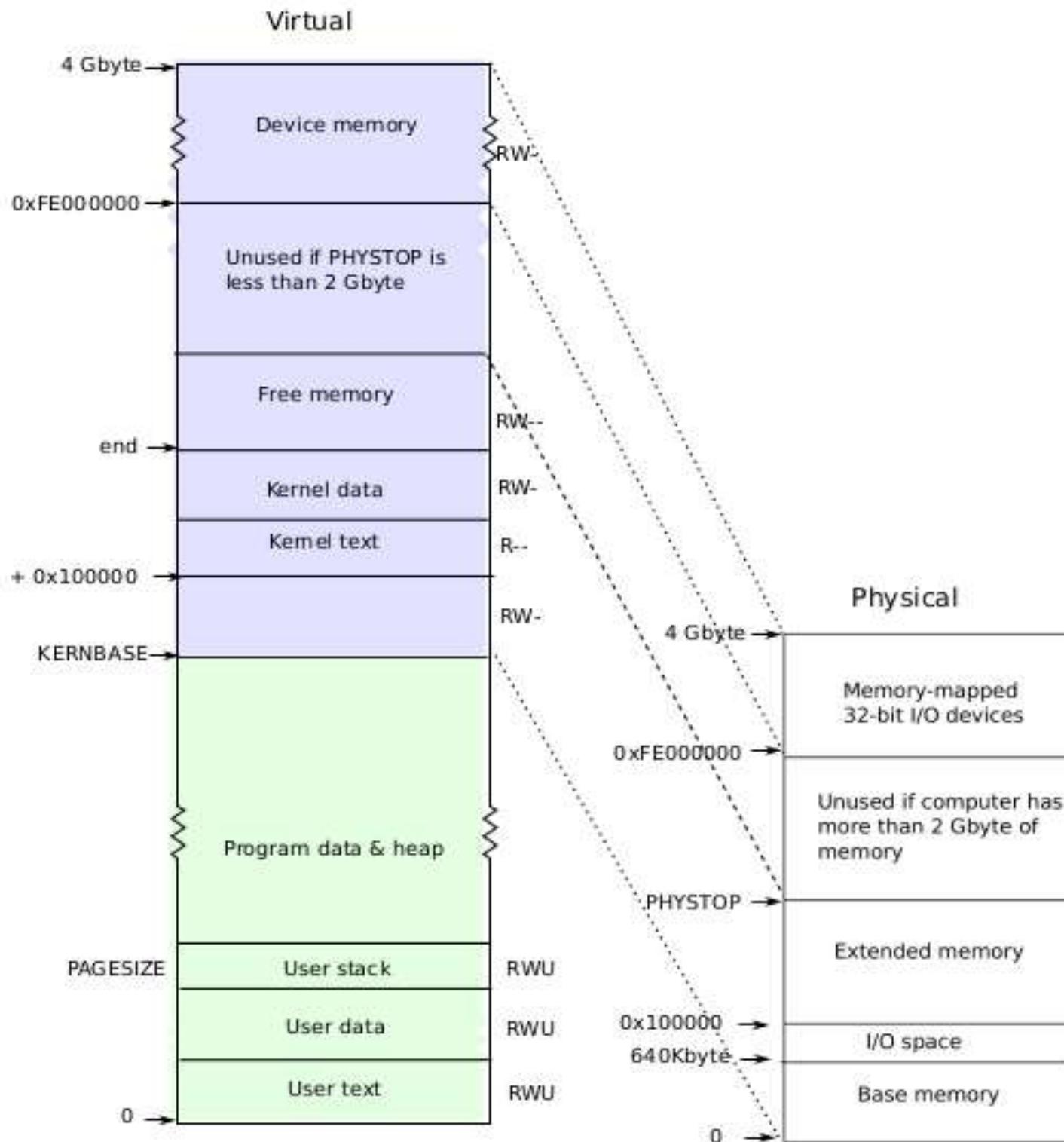
```
struct {  
  
    struct spinlock lock;  
  
    struct proc proc[NPROC];  
  
} ptable;
```

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

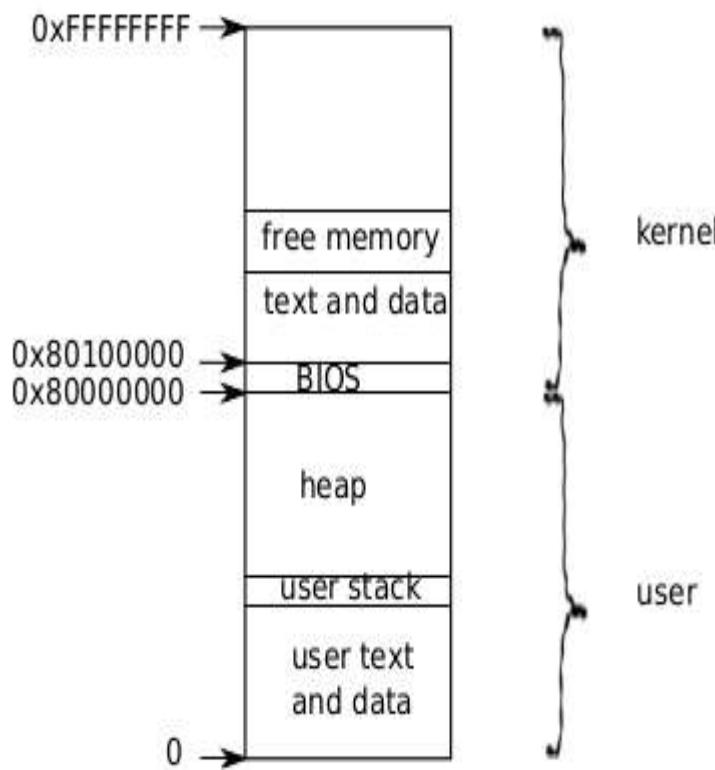
Layout of process's VA space

kv6 schema!

different from Linux

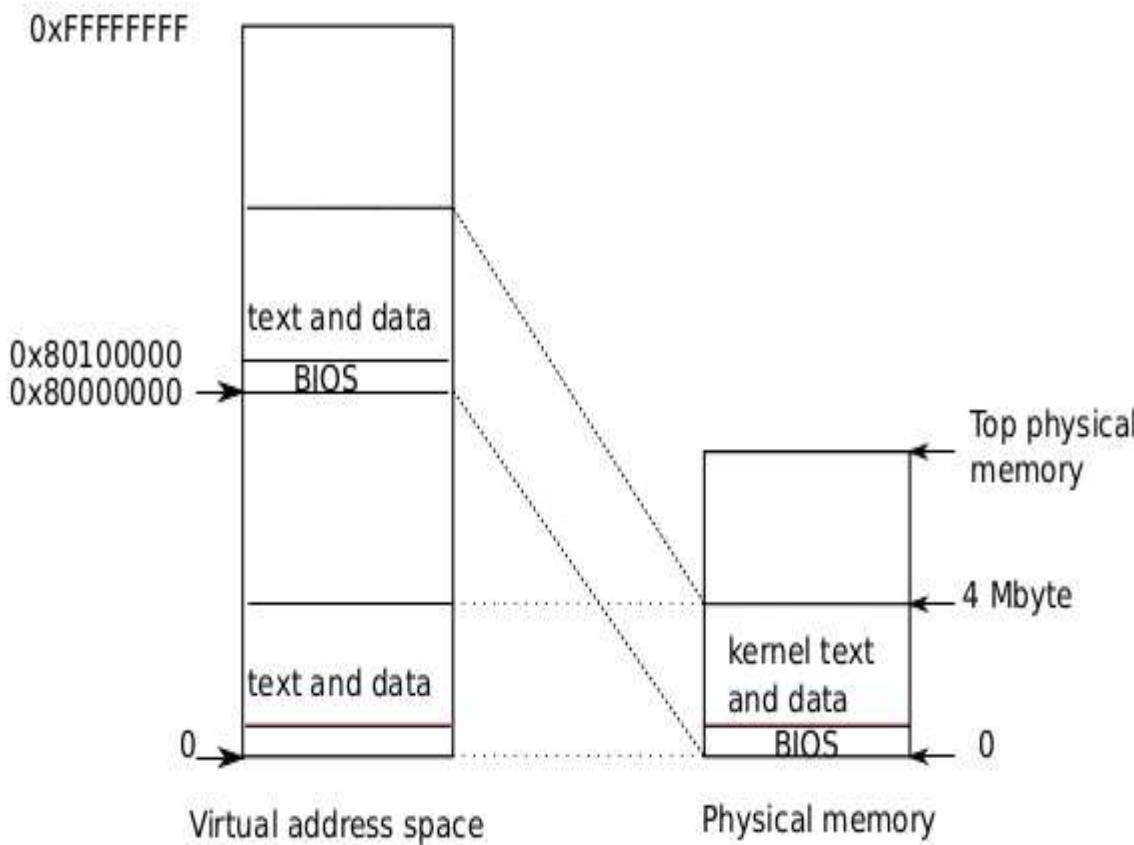


Logical layout of memory for a process



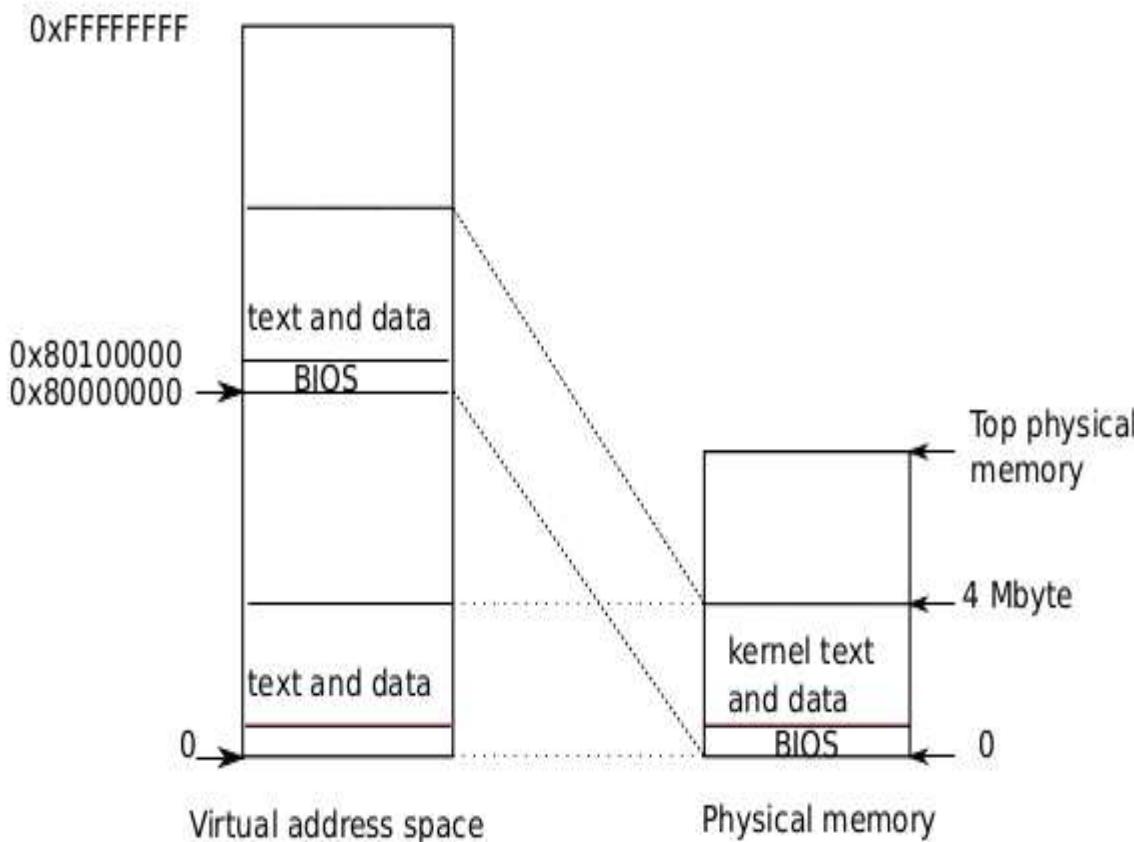
- Address 0: code
- Then globals
- Then stack
- Then heap
- Each process's address space maps kernel's text,

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

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

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

Struct proc

// Per-process state

struct proc {

uint sz;

// Size of process

memory (bytes)

pde_t* pgdir;

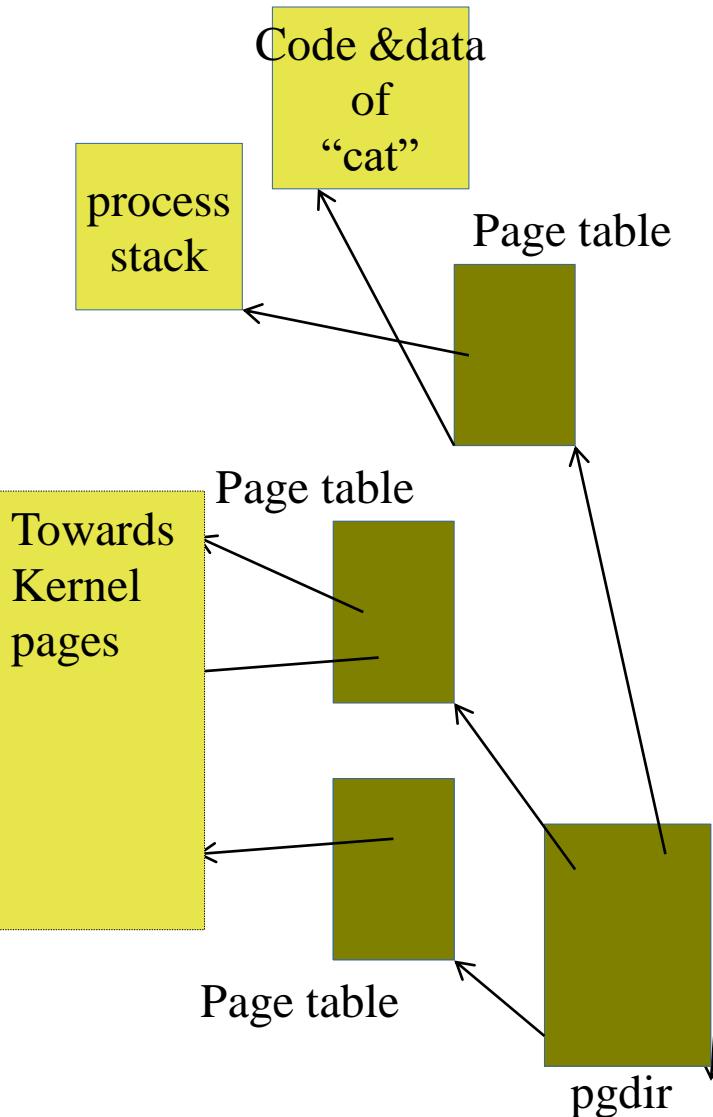
// Page table

```
char *kstack; // Be  
kernel stack for this process
```

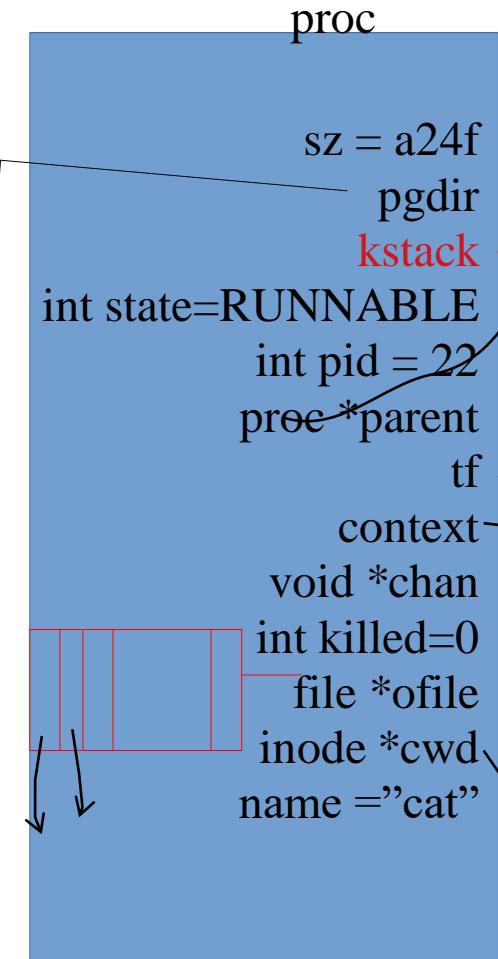
```
enum procstate state;
```

allocated, ready to run, running, wait-

struct proc diagram: Very imp!



`sz = ELF-code->memsz` (includes data, check “`ld -N`”
+ 2×4096 (for stack))



The diagram illustrates a call stack with three frames:

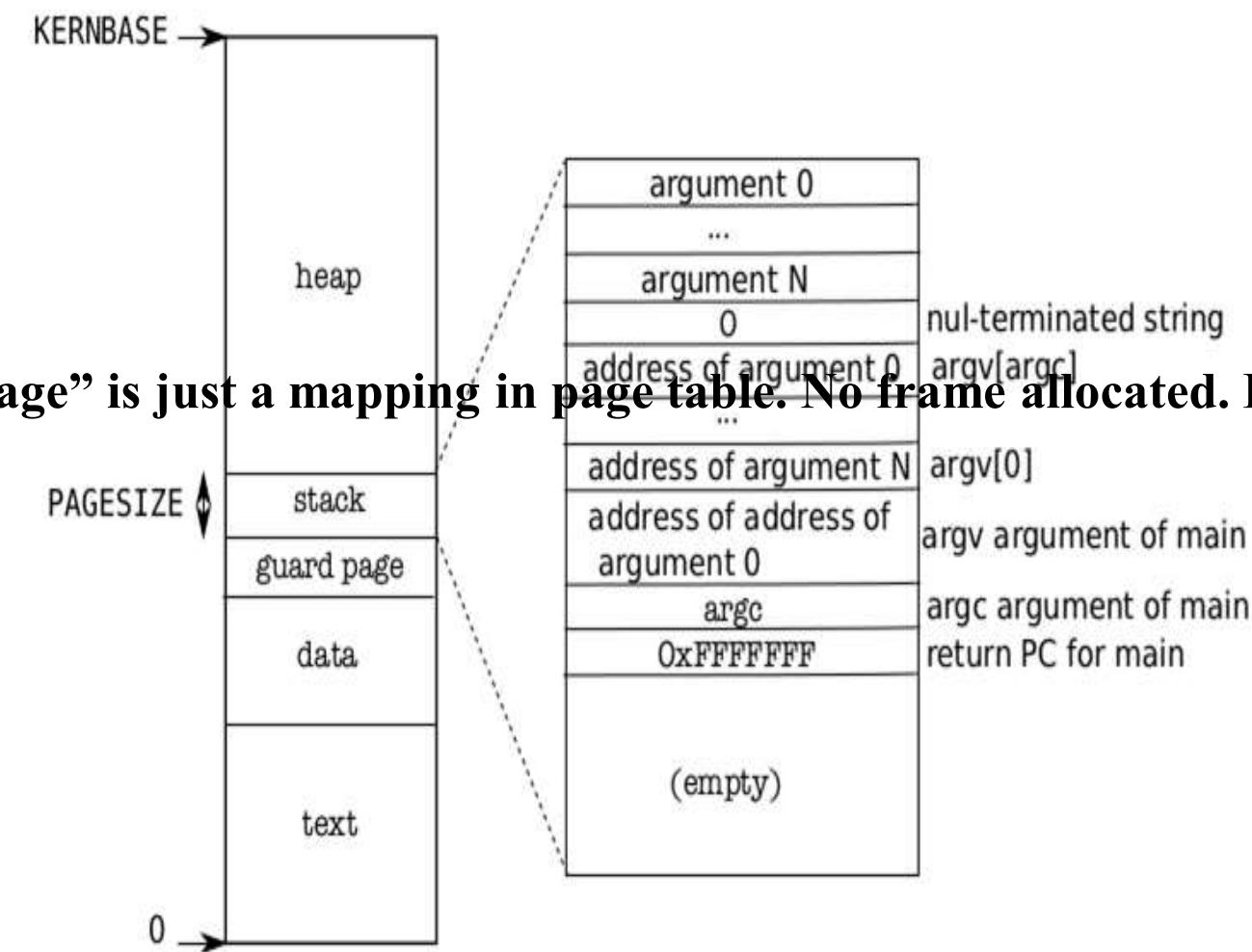
- Trapframe**: The top frame is orange and contains registers: edi, esi, ebp, ebx, edx, ecx, eax, gs, fs, es=4, ds=4, trapno=? , err, eip, cs = 3, EFLAGS = FL_IF, ESP = 4096, ss=4, and EIP = 0.
- trapret()**: The middle frame is white and contains the instruction eip=forkret().
- eip=forkret()**: The bottom frame is orange and contains registers: ebp, ebx, esi, and edi.

Arrows indicate the flow of the stack: an upward-pointing arrow from the bottom frame to the middle frame, and a downward-pointing arrow from the middle frame to the bottom frame.

In use only when you are

Memory Layout of a user process

Memory Layout of a user process



After exec()

Note the argc, argv on stack

"page" is just a mapping in page table. No frame allocated. It's marked as invalid. So if stack grows

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;          //
```

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,
```

Setting IDT entries

```
#define SETGATE(gate, istrap, sel,
off, d)          \
{                  \
    (gate).off_15_0 = (uint)(off) & \
0xffff;           \
    (gate).cs = (sel); \
    (gate).args = 0; \
}
```

Setting IDT entries

Vectors.S

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

```
# handlers
```

```
.globl alltraps
```

```
.globl vector0
```

```
vector0:
```

```
    pushl $0
```

```
    li t0
```

trapasm.S

```
#include "mmu.h"  
  
# vectors.S sends all  
traps here.
```

```
.globl alltraps
```

```
alltraps:
```

```
    # Build trap frame.
```

```
    pushl %ds
```

```
    pushl %es
```

```
    pushl %fs
```

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
- 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

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
```

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();

trap()

- Argument is trapframe

- In alltraps

- Before “call trap”, there was “push %esp” and stack had the trapframe

- Remember calling convention --> when a function is called, the stack contains the arguments in reverse order (here only 1 arg)

trap()

- Has a switch
- **switch(tf->trapno)**
- Q: who set this trapno?
- Depending on the type of trap
- Call interrupt handler
 - Timer
 - wakeup(&ticks)
 - IDE: disk interrupt
 - Ideintr()
 - KBD
 - Kbdintr()
 - COM1
 - Uatrintr()
 - If Timer

when trap() returns

❑#Back in alltraps

call trap

addl \$4, %esp

Return falls through
to trapret...

.globl trapret

trapret:

popal

❑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

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

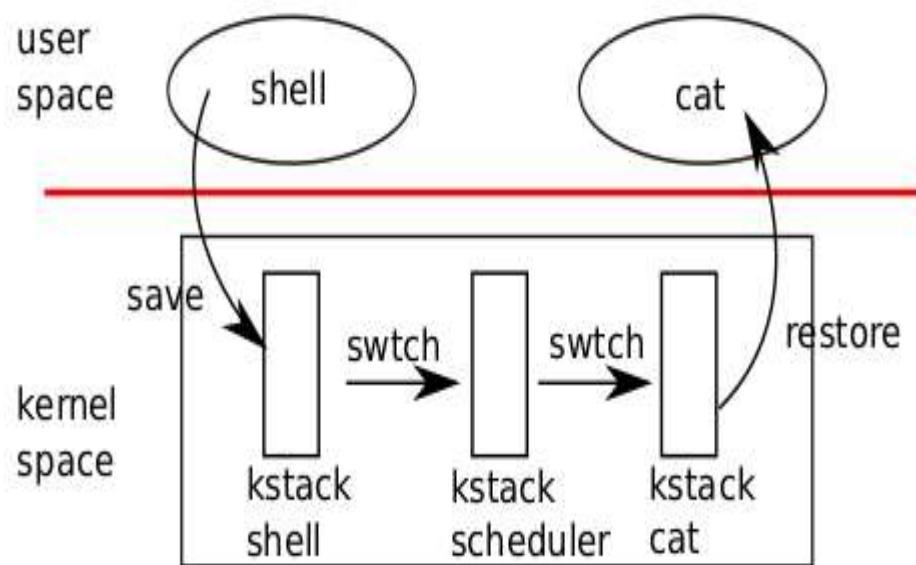
F() { i++; j++; }

❑ P2 was doing

G() { x--; y++; }

❑ P1 will experience a timer interrupt,

4 stacks need to change!



- User stack of process ->
- kernel stack of process
- Switch to kernel stack
- The normal sequence on any

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

swtch

swtch:

movl 4(%esp), %eax

movl 8(%esp), %edx

Save old callee-saved registers

pushl %ebp

pushl %ebx

pushl %esi

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

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

void

sched(void)

{

int intena;

struct proc *p =
myproc();

if(!holding(&ptabl
e.lock))

sched()

□ get current
process

□ Error checking
code (ignore as of
now)

□ get interrupt
enabled status on
current CPU
(ignore as of now)

sched() and schduler()

sched()

scheduler(void) {

3

3

```
    swtch(&p->context, mycpu(), &scheduler), p->scheduler); sched() saves context in p*/>context
}
}
```

- after `switch()` call in `sched()`, the control jumps to Y in scheduler

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

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()
- Now the loop in scheduler()
 - calls switchkvm()
 - Then continues to find next process (P2) to run
 - Then calls switchuvm(p): changing the page

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

Dynamic Linking on Linux

```
#include <stdio.h>

int main() {
    int a, b;
    scanf("%d%d", &a, &b);
    PLP!printf("%d%d\n", a, b);
    used to call external procedures/functions whose address is to be resolved by the dynamic linker at run time
    return 0;
}
```

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: .endbrf64

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

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)

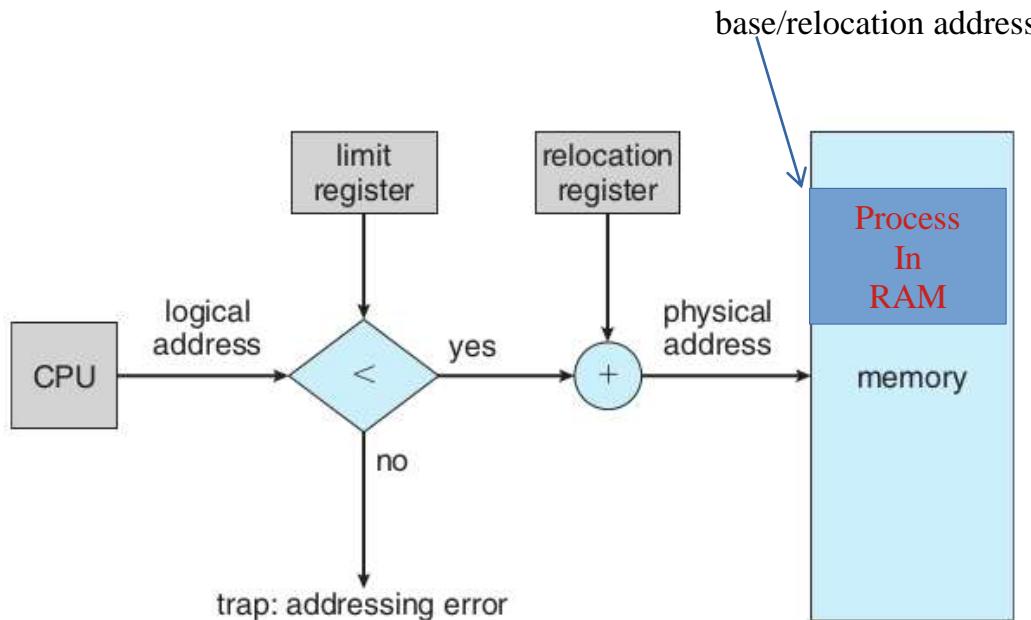


Figure 9.6 Hardware support for relocation and limit registers.

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**

Variable partition scheme

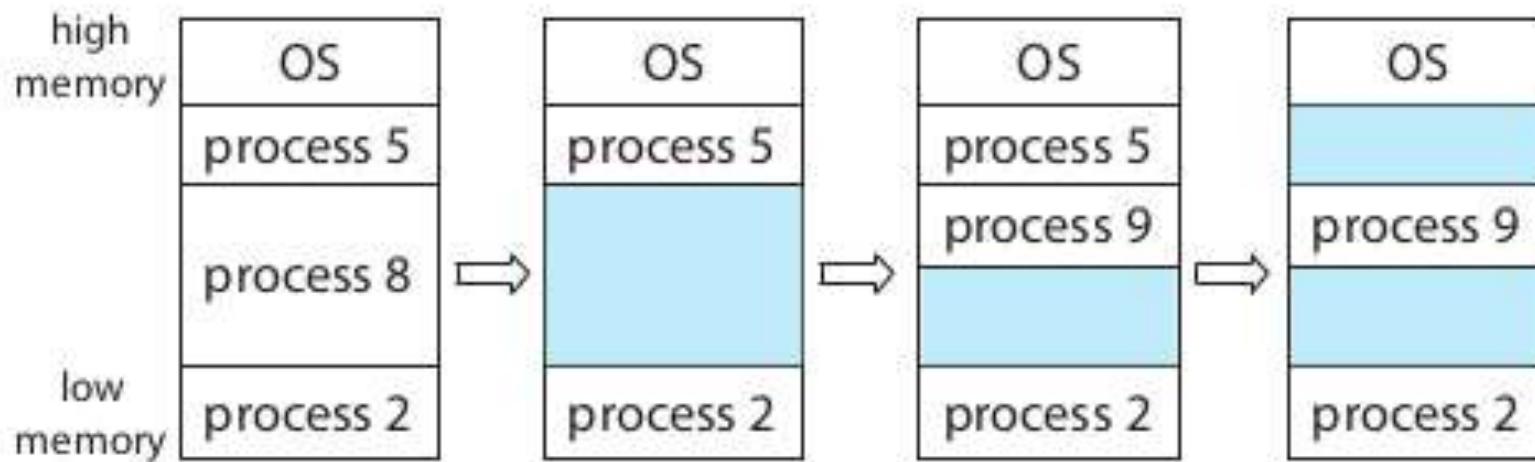


Figure 9.7 Variable partition.

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

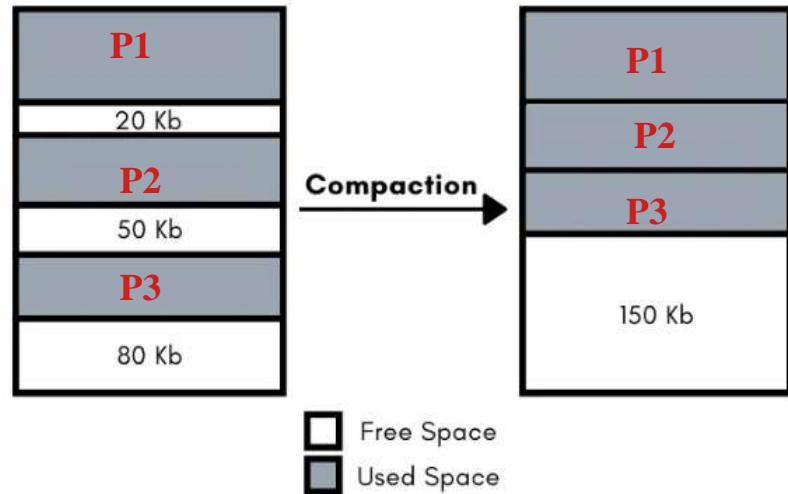
- Free chunks: 30k, 40k, 20k
- The newly created process (during fork() + exec()) needs 50k
- Total free memory: $30+40+20 = 90\text{k}$
- But can't allocate 50k !

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.



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



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

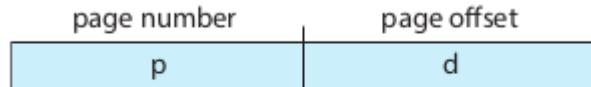
- .Reduce the size of the fixed sized partition**
- .How small then ?**
 - Smaller partitions mean more overhead for the operating system in allocating deallocating**

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

Logical address seen as



Paging hardware

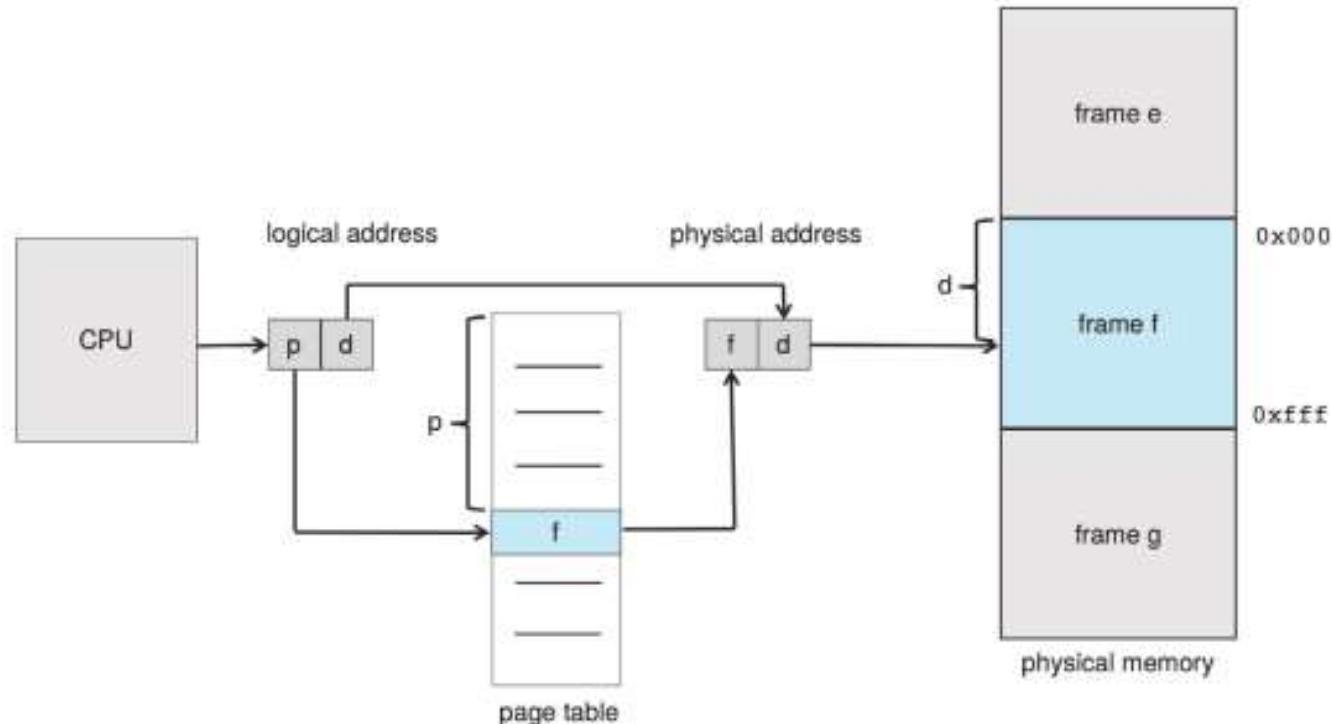


Figure 9.8 Paging hardware.

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**

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

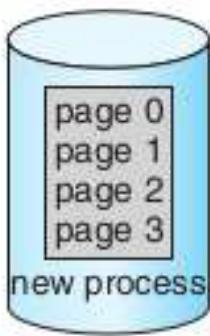
14

13

18

20

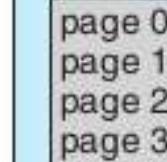
15



(a)

free-frame list

15



(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%**

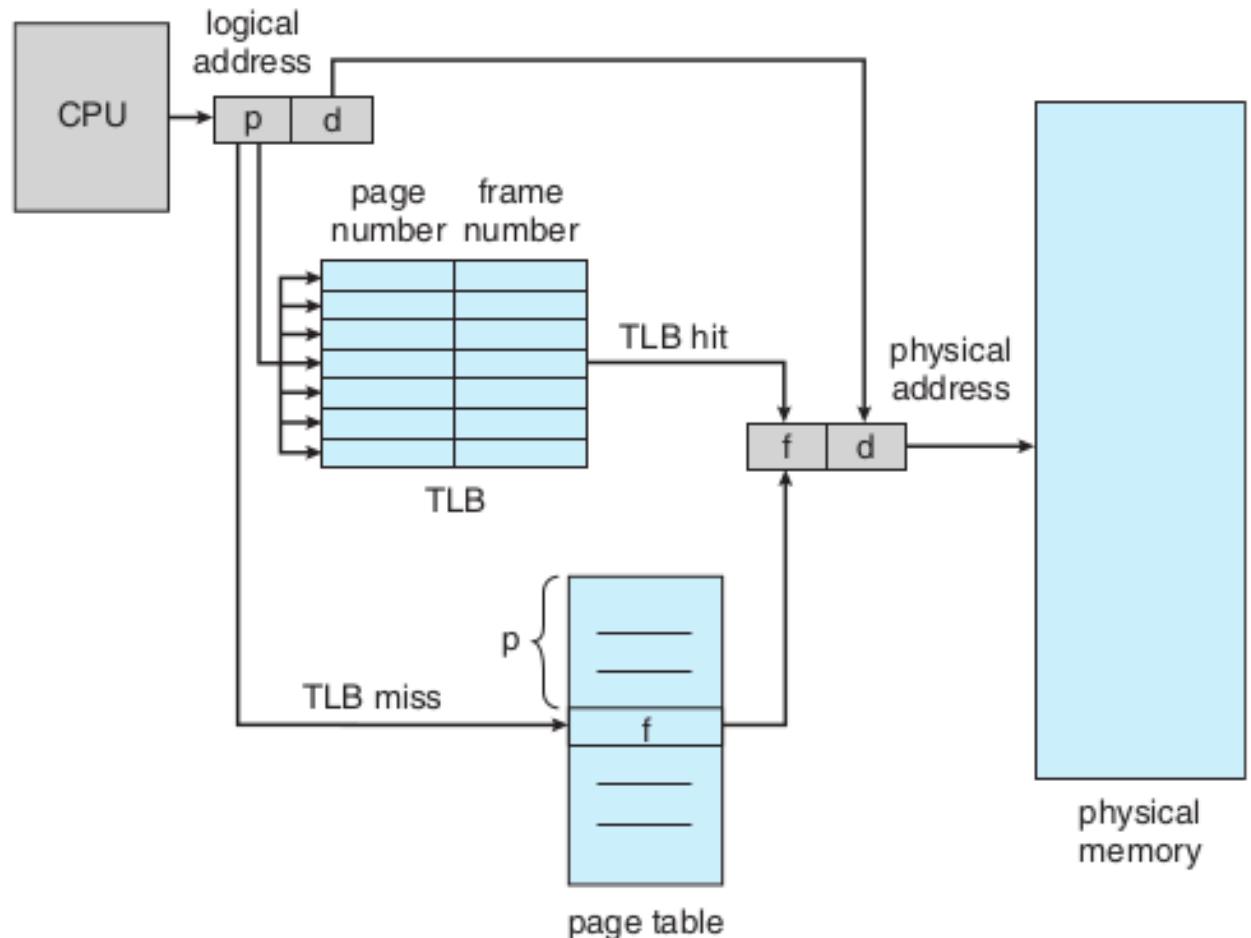
Speeding up paging

- 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 = $0.80 \times 10 + 0.20 \times 20$

Memory protection with paging

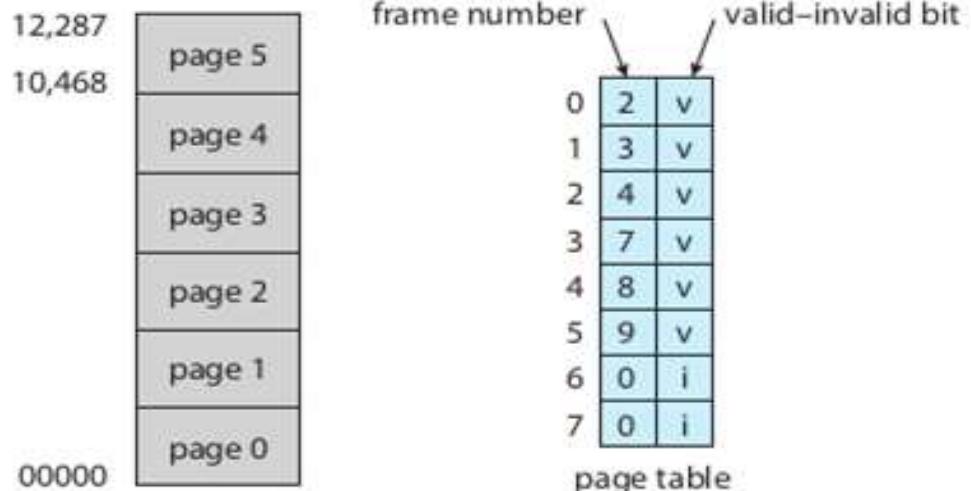


Figure 9.13 Valid (v) or invalid (i) bit in a page table.

X86 PDE and PTE

31

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

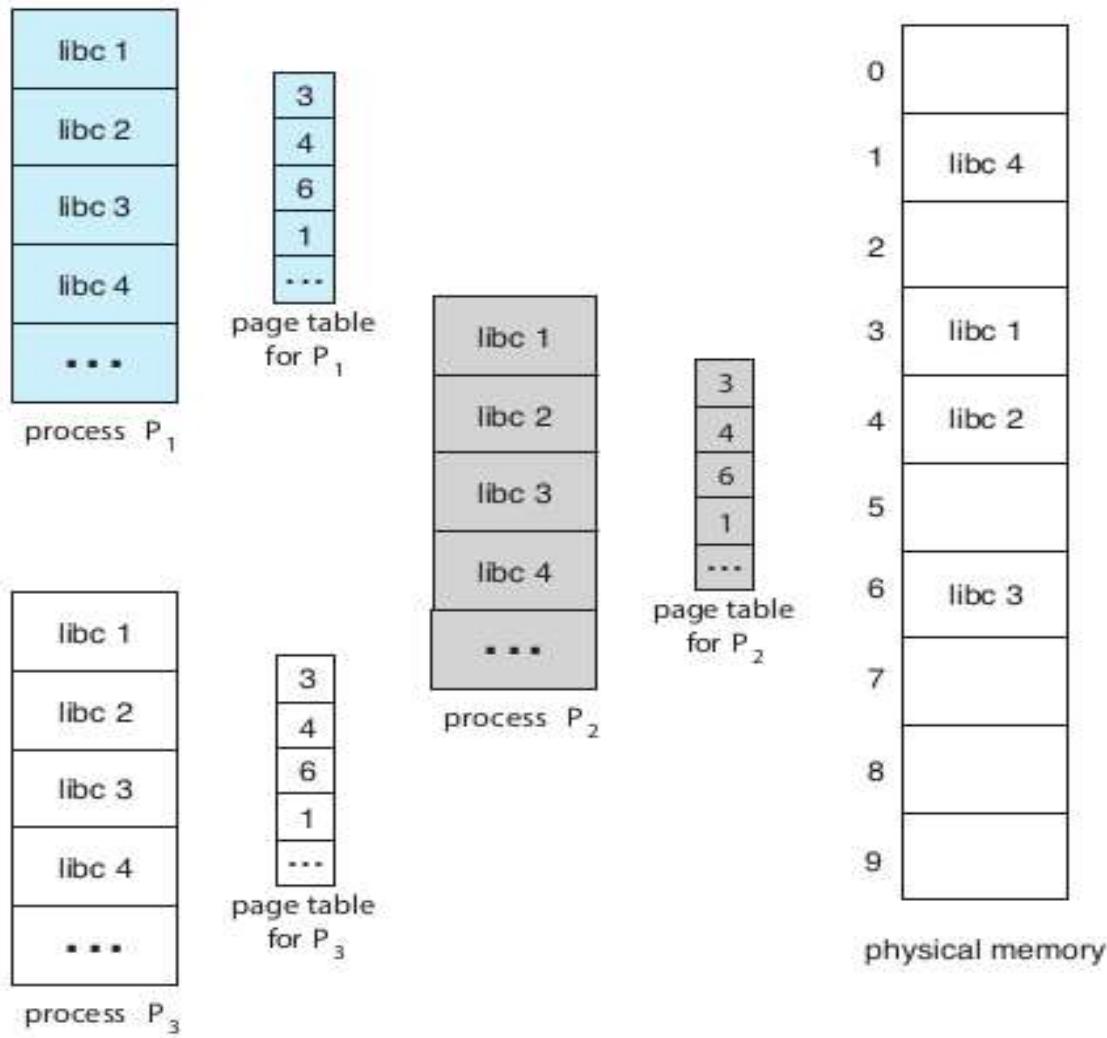
G Global page

AVL Available for system use

31

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

PTE



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

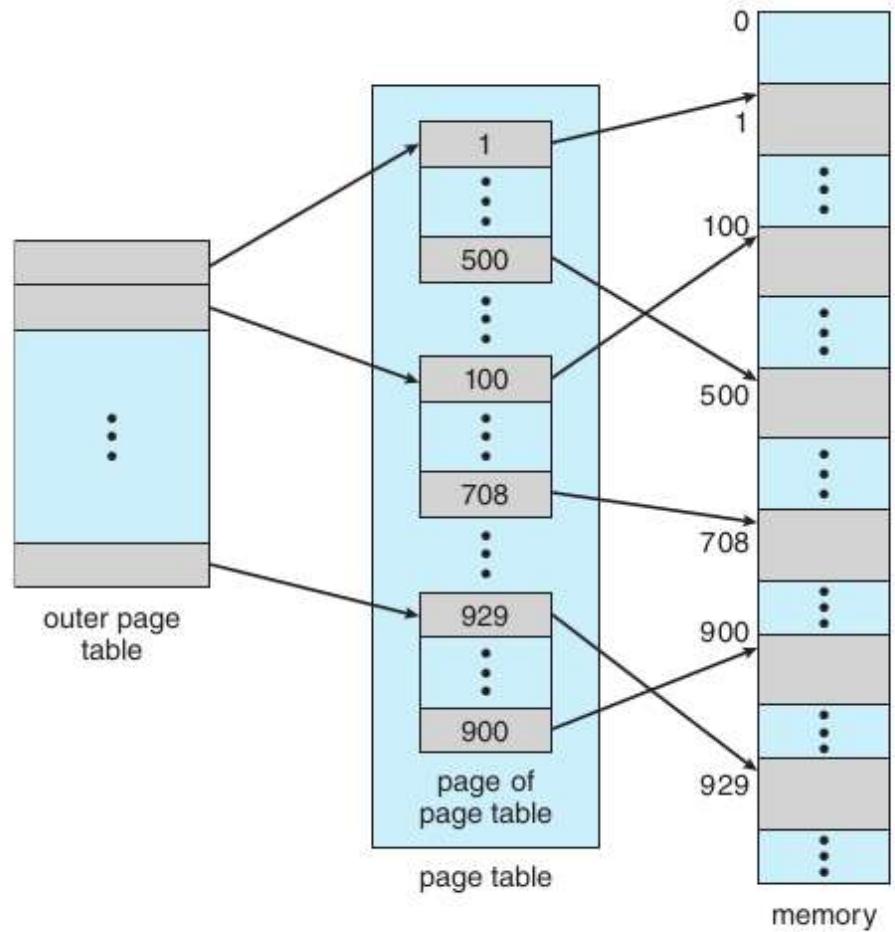
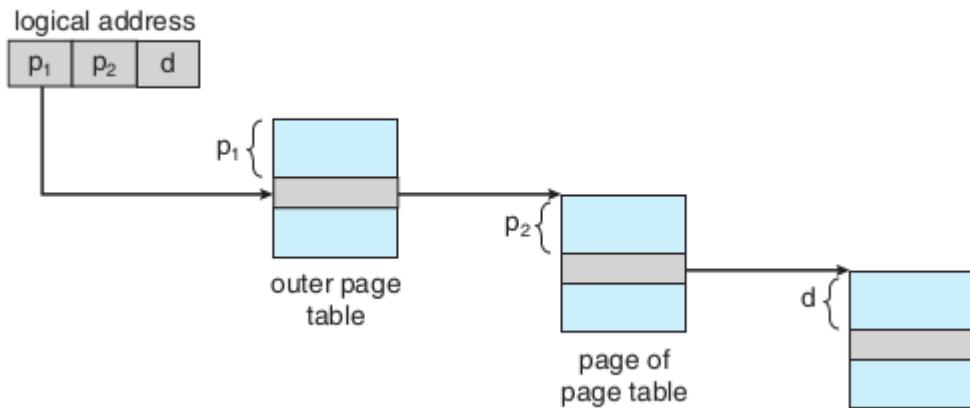
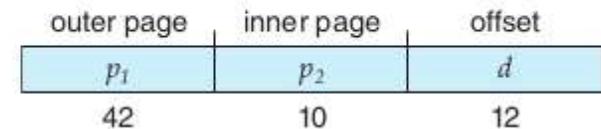


Figure 9.15 A two-level page-table scheme.



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

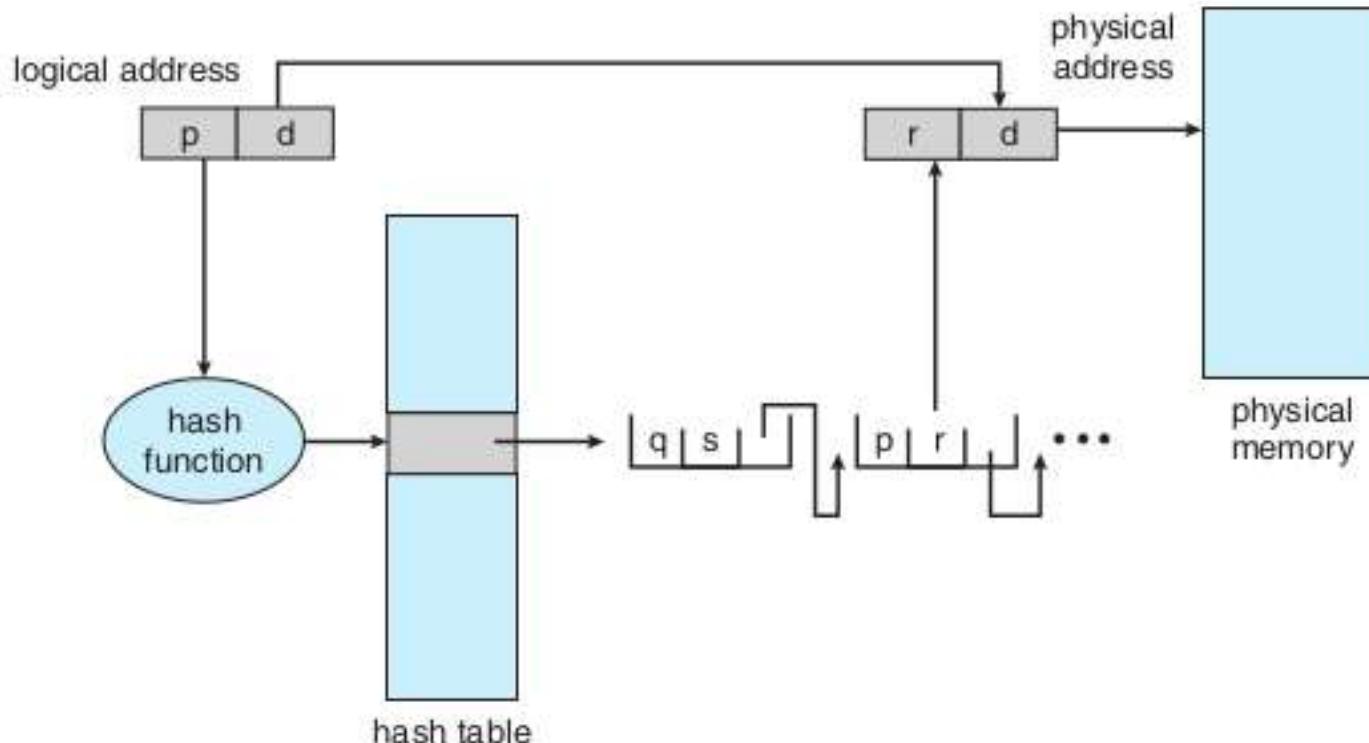
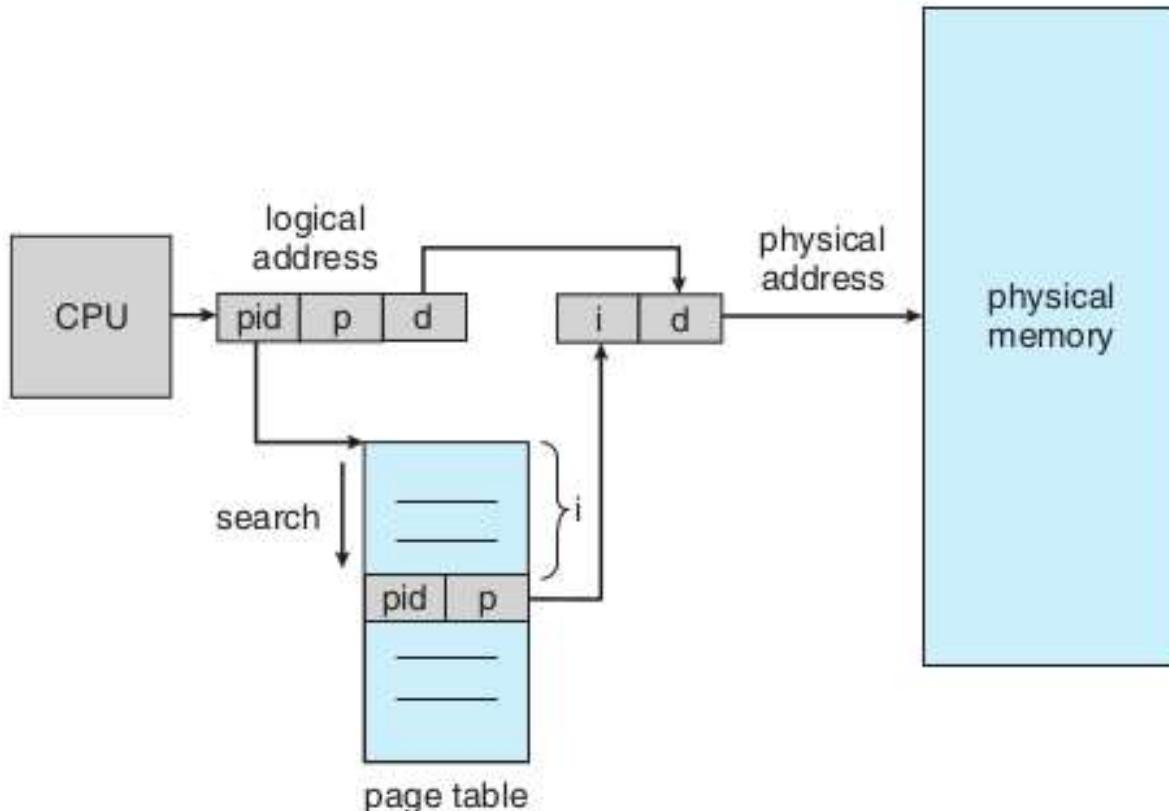


Figure 9.17 Hashed page table.

Inverted page table



Normal page table – one per process

Inverted page table : global table – one for all processes
Needs to store PID in the table entry

Examples of systems using inverted page tables include the 64-bit Ultra SPAF

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

Figure 9.18 Inverted page table.

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**

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

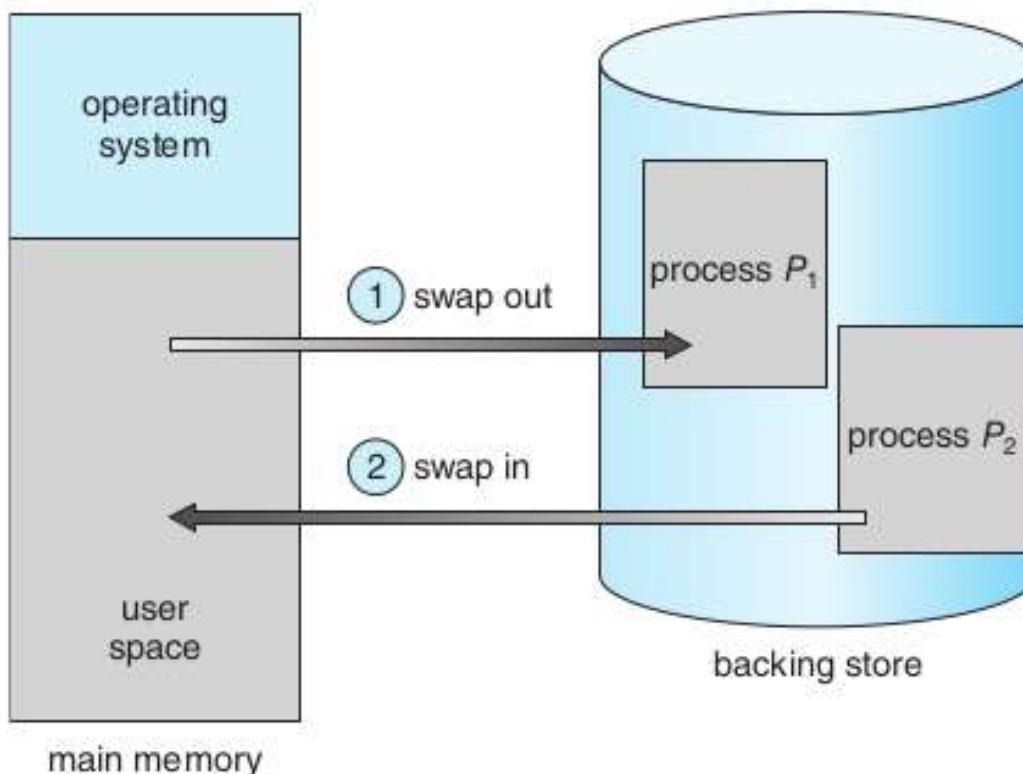


Figure 9.19 Standard swapping of two processes using a disk as a backing store.

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

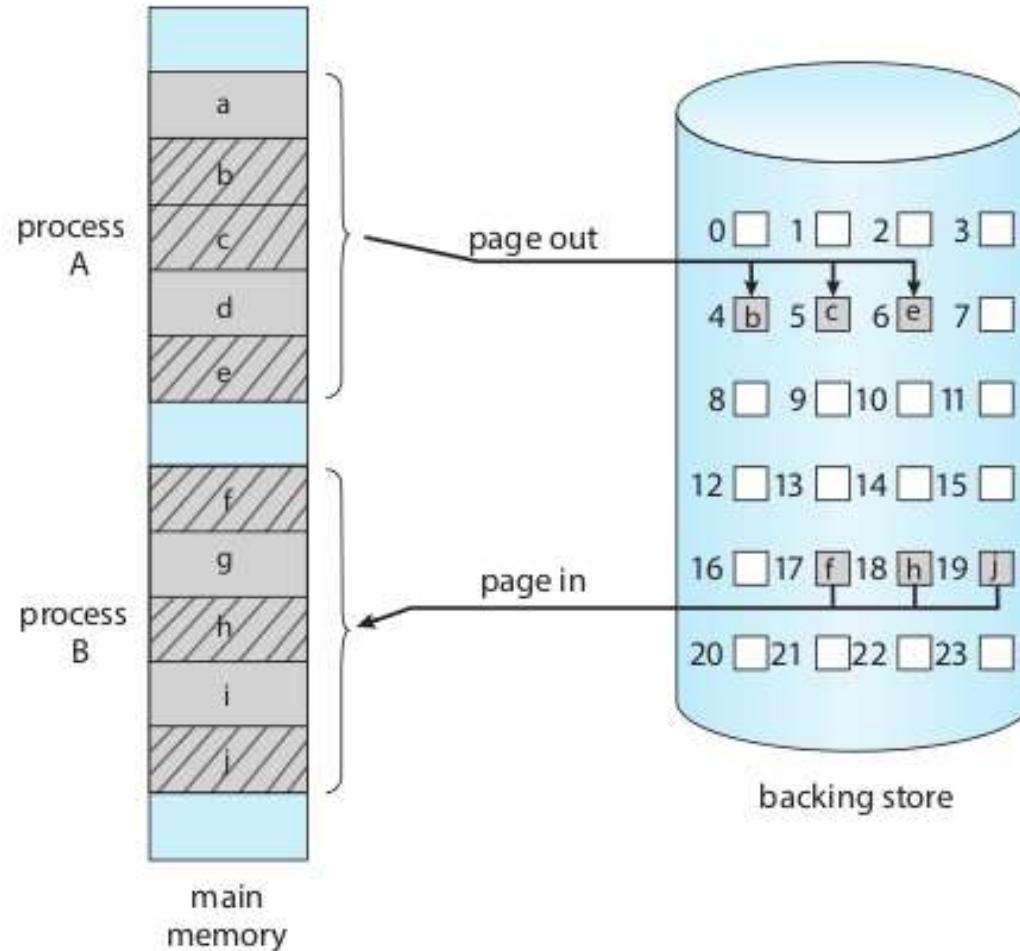


Figure 9.20 Swapping with paging.

Words of caution about ‘paging’

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

File Systems

Abhijit A M

abhijit.comp@coep.ac.in

System calls related to files/file-system

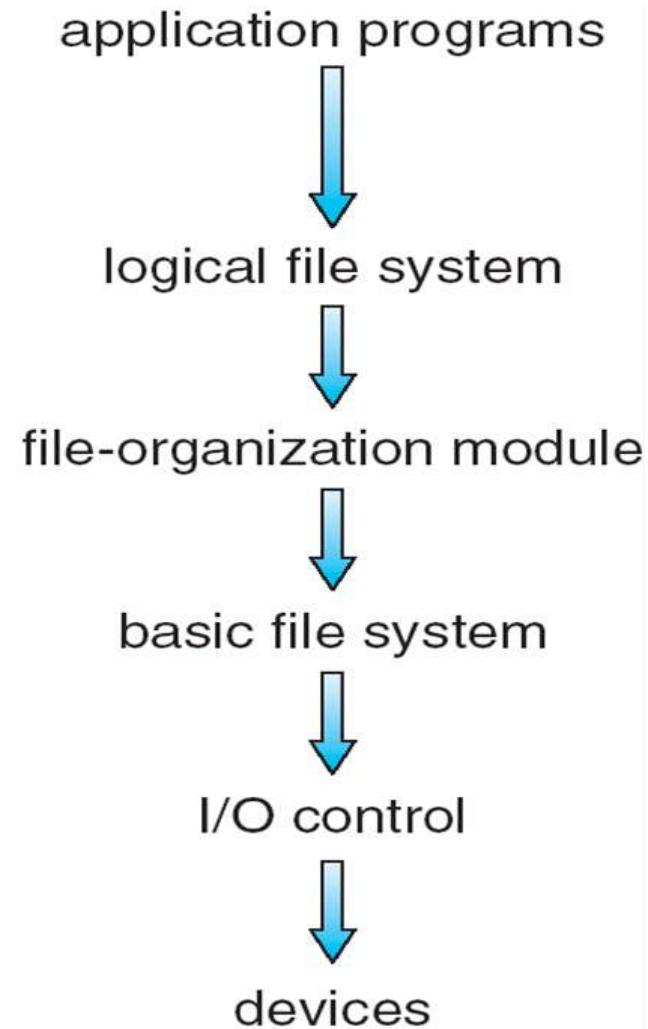
- ❑ **Open(2), chmod(2), chown(2), close(2), dup(2), fcntl(2), link(2), lseek(2), mknod(2), mmap(2), mount(2), read(2), stat(2), umask(2), unlink(2), write(2), fstat(2), access(2), readlink(2), ...**

Implementing file systems

File system on disk

- Disk I/O in terms of sectors (512 bytes)
- File system: implementation of acyclic graph using the linear sequence of sectors
- Device driver: available to rest of the OS code to access disk using a block number

File system implementation: layering



Application programs

```
int main() {  
    char buf[128]; int count;  
    fd = open(...);  
    read(fd, buf, count);  
}
```

OS

Logical file system:

```
sys_read(int fd, char *buf, int count) {  
    file *fp = currproc->fdarray[fd];  
    file_read(fp, ...);  
}
```

File organization module:

```
file_read(file *fp, char *buf, int count) {  
    offset = fp->current-offset;  
    translate offset into blockno;  
    basic_read(blockno, buf, count);  
}
```

Basic File system:

```
basic_read(int blockno, char *buf, ...) {  
    os_buffer *bp;  
    sectorno = calculation on blockno;  
    disk_driver_read(sectorno, bp );  
    move-process-to-wait-queue;  
    copy-data-to-user-buffer(bp, buf);  
}
```

IO Control, Device driver:

```
disk_driver_read(sectorno) {  
    issue instructions to disk controller (ofte  
    to read sectorno    into specific location  
}
```

XV6 does it slightly differently, but following the

A typical file control block (inode)

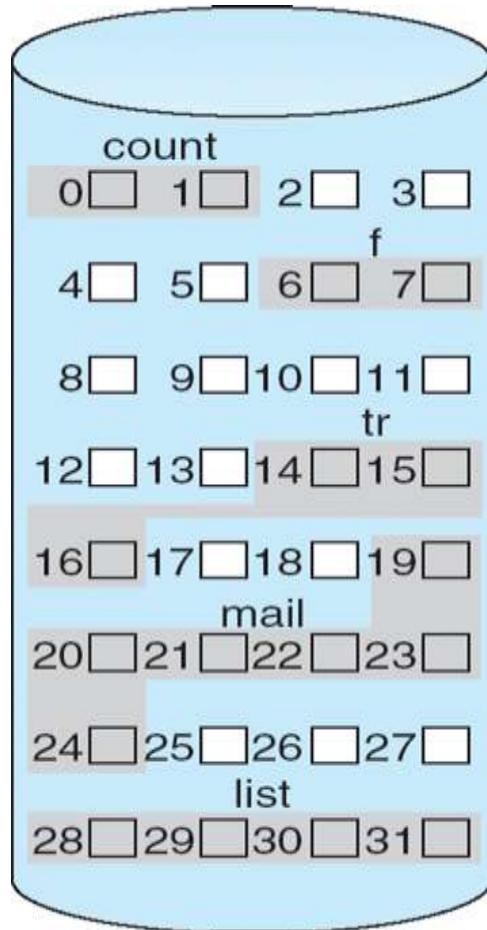
file permissions
file dates (create, access, write)
file owner, group, ACL
file size
file data blocks or pointers to file data blocks

**Why does it NOT contain the
Name of the file ?**

Disk space allocation for files

- File contain data and need disk blocks/sectors for storing it
- File system layer does the allocation of blocks on disk to files
- Files need to
 - Be created, expanded, deleted, shrunk, etc.
 - How to accommodate these requirements?

Contiguous Allocation of Disk Space



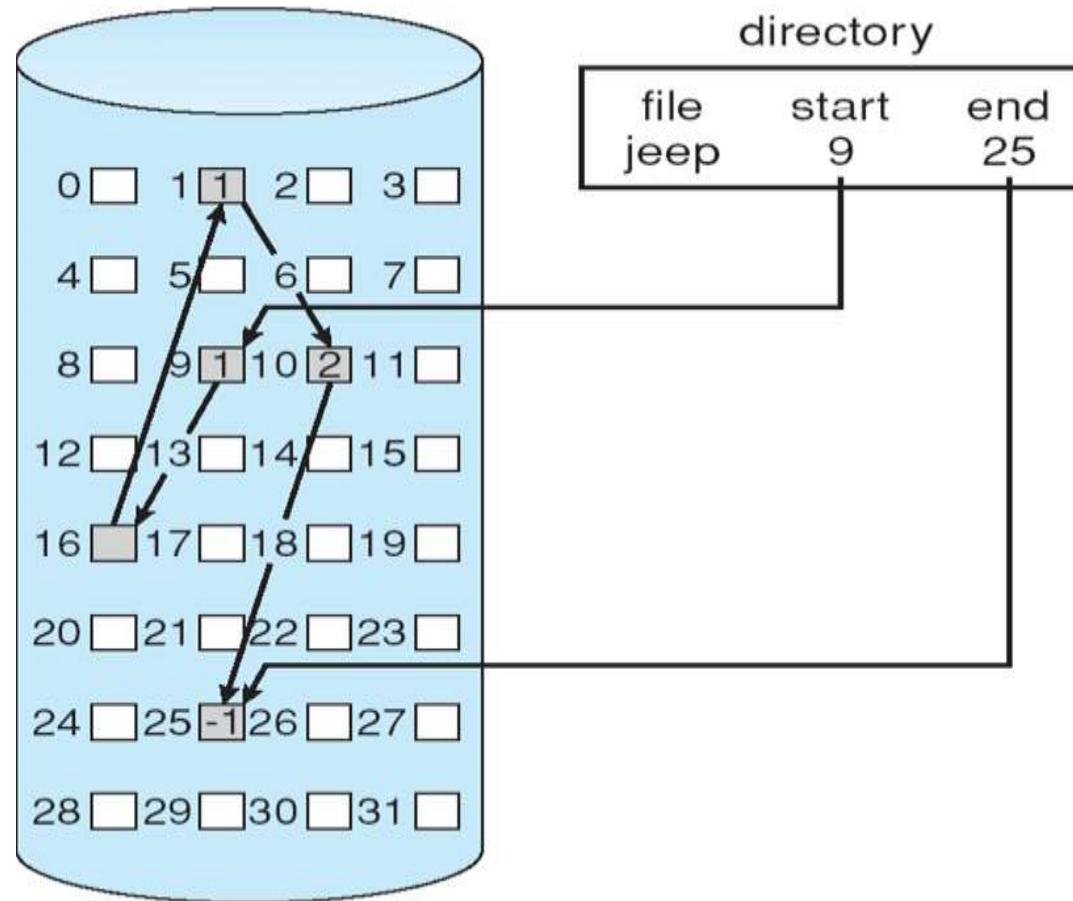
directory

file	start	length
count	0	2
tr	14	3
mail	19	6
list	28	4
f	6	2

Contiguous allocation

- Each file occupies set of contiguous blocks
- Best performance in most cases
- Simple – only starting location (block #) and length (number of blocks) are required
- Problems include finding space for file, knowing file size, external fragmentation, need for compaction off-line (downtime) or on-line

Linked allocation of blocks to a file

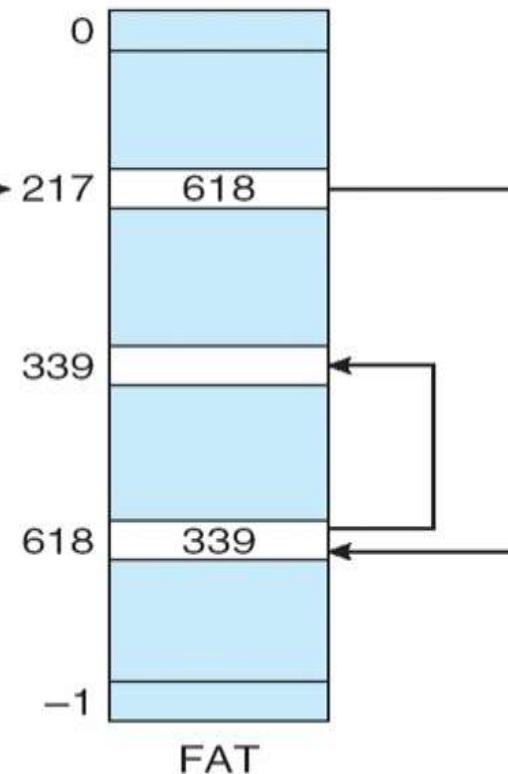
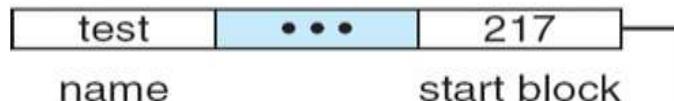


Linked allocation of blocks to a file

- Linked allocation
- Each file a linked list of blocks
- File ends at nil pointer
- No external fragmentation
- Each block contains pointer to next block (i.e.
- Free space management system called when new block needed
- Improve efficiency by clustering blocks into groups but increases internal fragmentation
- Reliability can be a problem

FAT: File Allocation Table

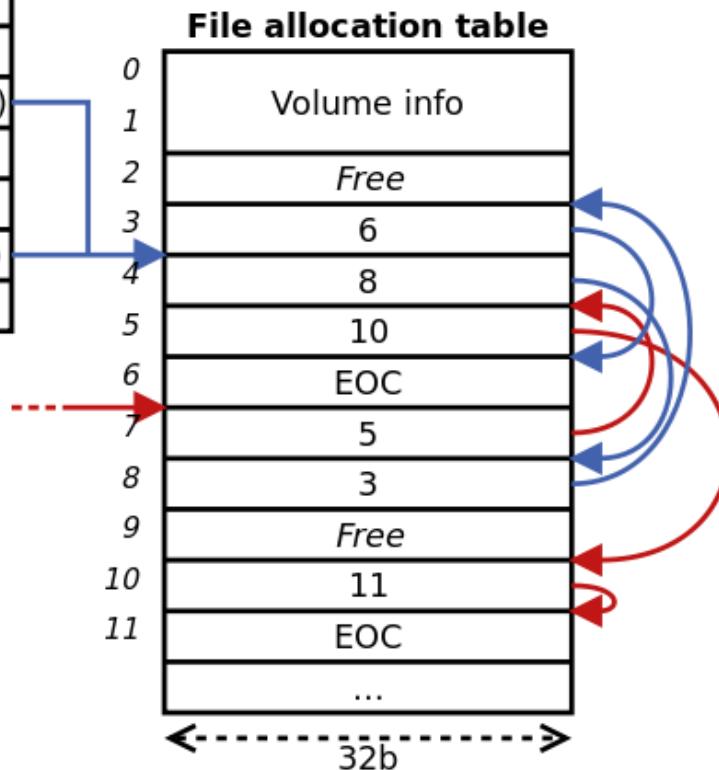
directory entry



- FAT (File Allocation Table), a variation
- Beginning of volume has table, indexed by block number
- Much like a linked list, but faster on disk and cacheable
- New block allocation simple

Directory table entry (32B)

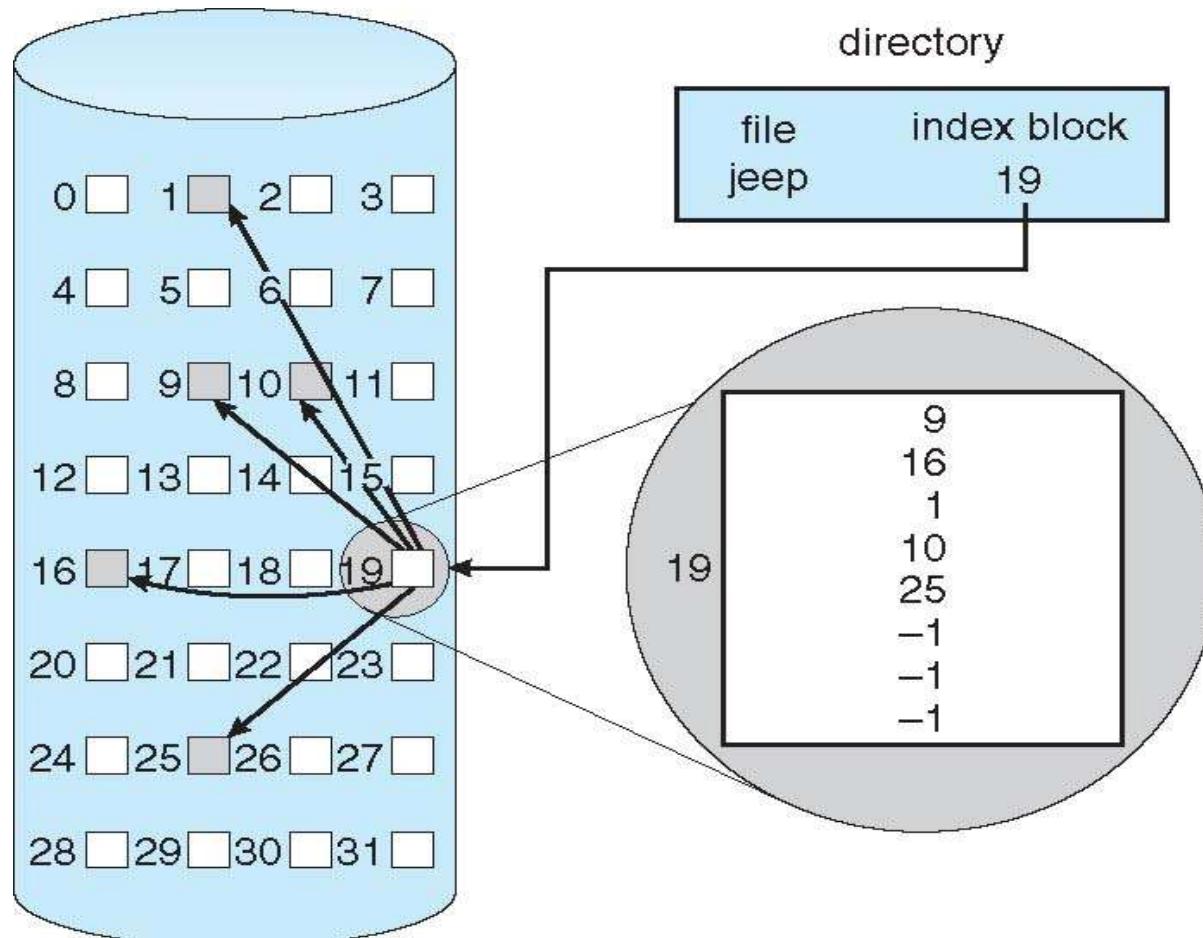
Filename (8B)
Extension (3B)
Attributes (1B)
Reserved (1B)
Create time (3B)
Create date (2B)
Last access date (2B)
First cluster # (MSB, 2B)
Last mod. time (2B)
Last mod. date (2B)
First cluster # (LSB, 2B)
File size (4B)



FAT: File Allocation Table

Variants: FAT8, FAT12, FAT16, FAT32, VFAT, ...

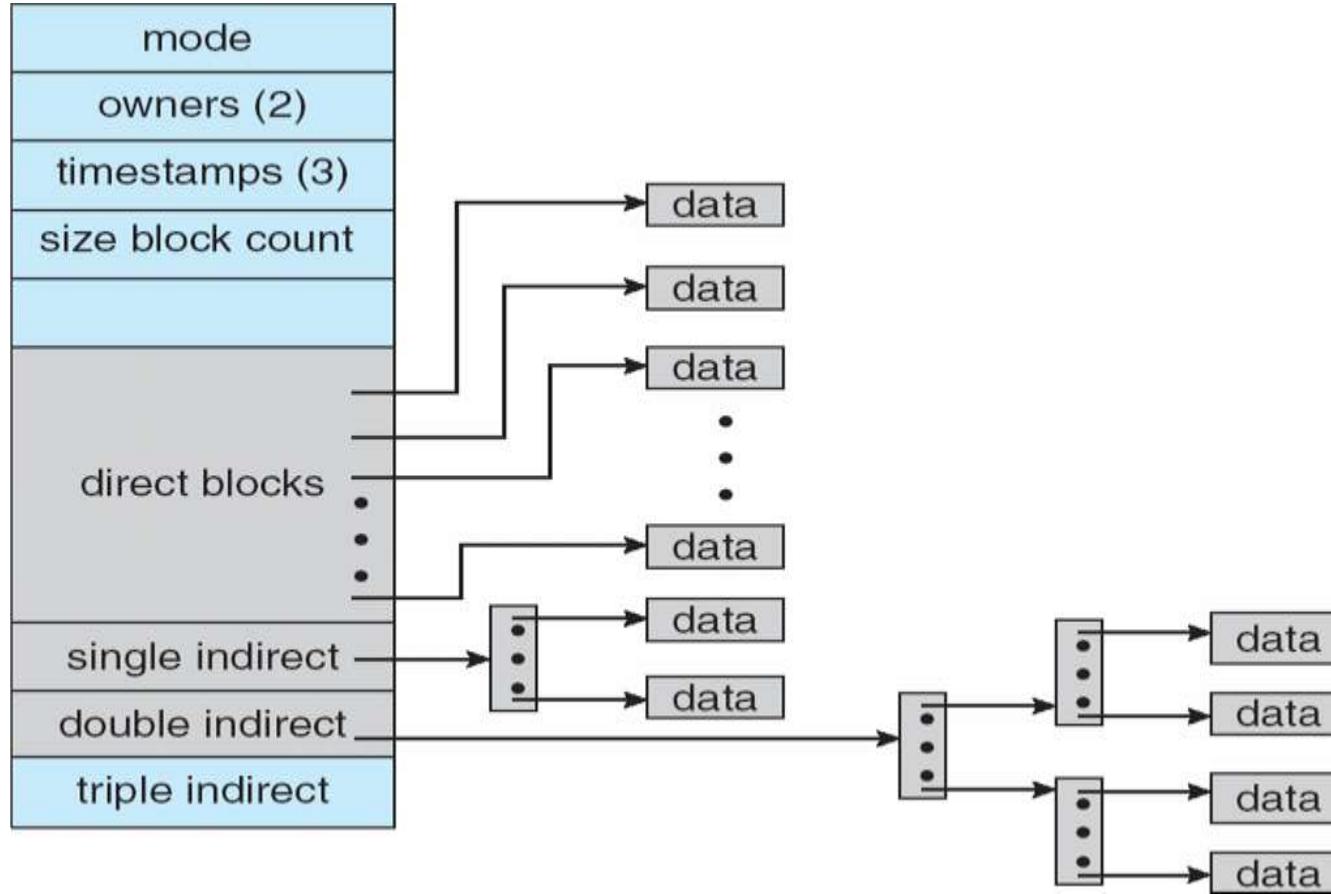
Indexed allocation



Indexed allocation

- Need index table
- Random access
- Dynamic access without external fragmentation,
but have overhead of index block
- Mapping from logical to physical in a file of
maximum size of 256K bytes and block size of 512
bytes. We need only 1 block for index table

Unix UFS: combined scheme for block allocation



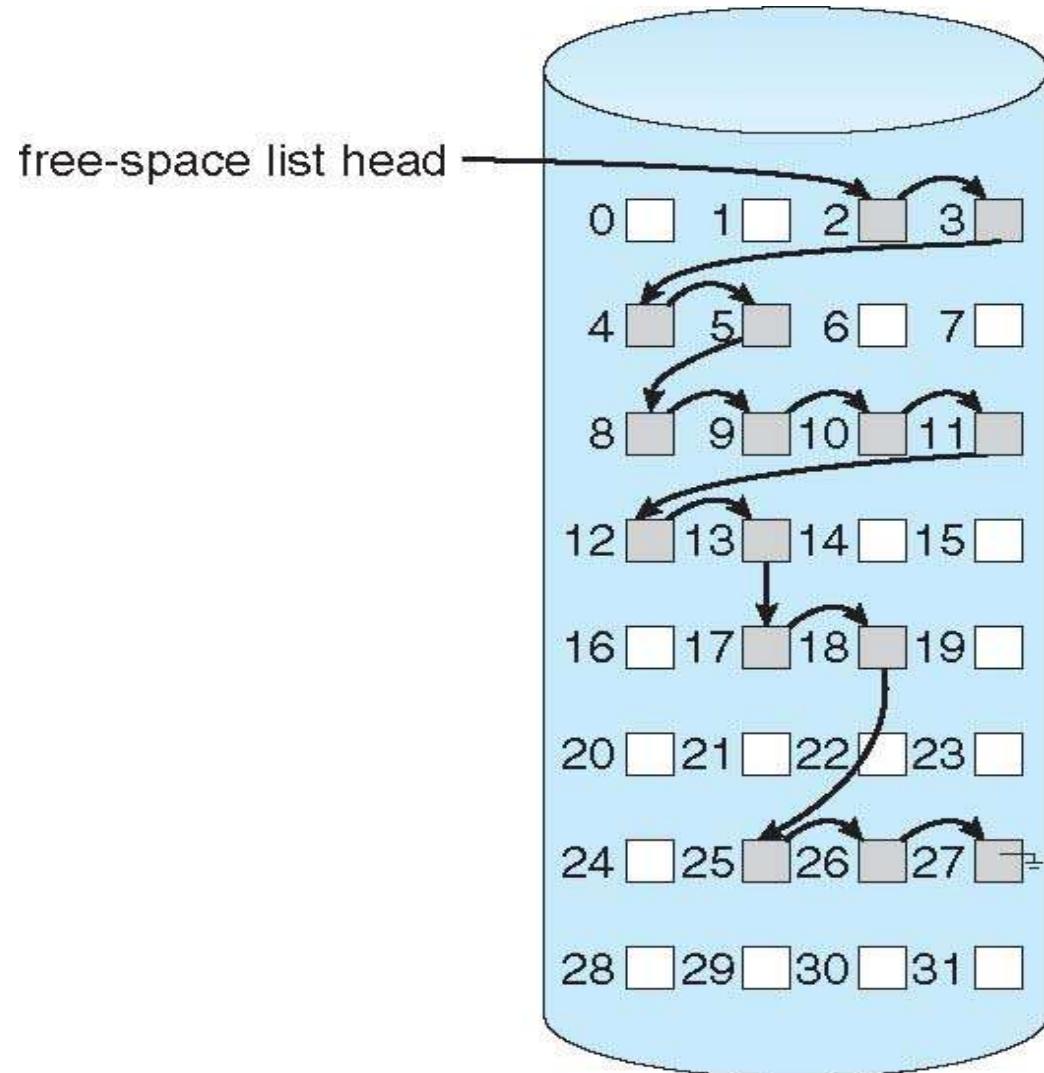
Free Space Management

- ❑ File system maintains free-space list to track available blocks/clusters
- ❑ Bit vector or bit map (n blocks)
- ❑ Or Linked list

Free Space Management: bit vector

- Each block is represented by 1 bit.
- If the block is free, the bit is 1; if the block is allocated, the bit is 0.
- For example, consider a disk where blocks 2, 3, 4, 5, 8, 9, 10, 11, 12, 13, 17, 18, 25, 26, and 27 are free and the rest of the blocks are allocated. The free-space bitmap would be 00111100111110001100000011100000 ...

Free Space Management: Linked list (not in memory, on disk!)



Further improvements on link list method of free-blocks

- Grouping
- Counting
- Space Maps (ZFS)
- Read as homework

Directory Implementation

- **Problem**

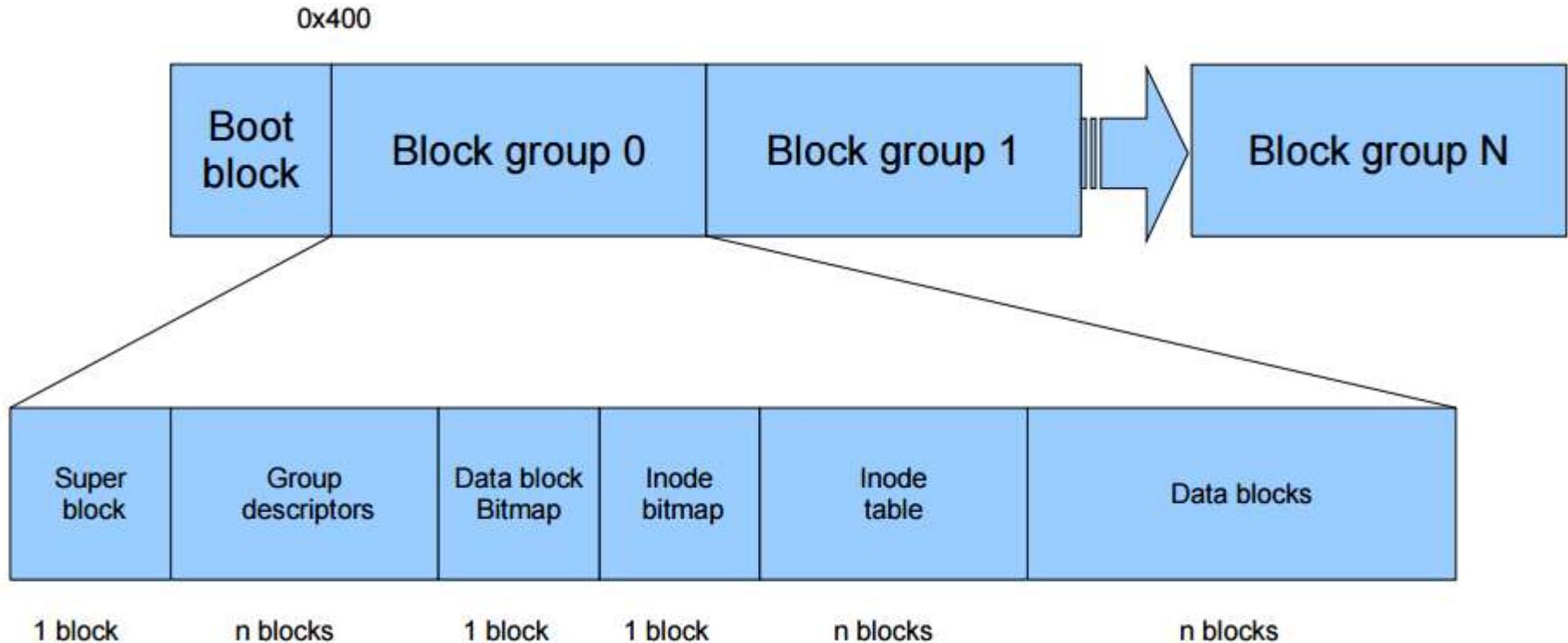
- **Directory contains files and/or subdirectories**
 - Operations required – create files/directories, access files/directories, search for a file (during lookup), etc.
- **Directory needs to give location of each file on disk**

Directory Implementation

- Linear list of file names with pointer to the data blocks
- Simple to program
- Time-consuming to execute
- Linear search time
- Could keep ordered alphabetically via linked list or use B+ tree
- Ext2 improves upon this approach.

Ext2 FS layout

Ext2 FS Layout



```
struct ext2_super_block {
    __le32 s_inodes_count; /* Inodes count */
    __le32 s_blocks_count; /* Blocks count */
    __le32 s_r_blocks_count; /* Reserved blocks count */
    __le32 s_free_blocks_count; /* Free blocks count */
    __le32 s_free_inodes_count; /* Free inodes count */
    __le32 s_first_data_block; /* First Data Block */
    __le32 s_log_block_size; /* Block size */
    __le32 s_log_frag_size; /* Fragment size */
    __le32 s_blocks_per_group; /* # Blocks per group */
    __le32 s_frags_per_group; /* # Fragments per group */
    __le32 s_inodes_per_group; /* # Inodes per group */
    __le32 s_mtime; /* Mount time */
    __le32 s_wtime; /* Write time */
    __le16 s_mnt_count; /* Mount count */
    __le16 s_max_mnt_count; /* Maximal mount count */
    __le16 s_magic; /* Magic signature */
    __le16 s_state; /* File system state */
    __le16 s_errors; /* Behaviour when detecting errors */
```

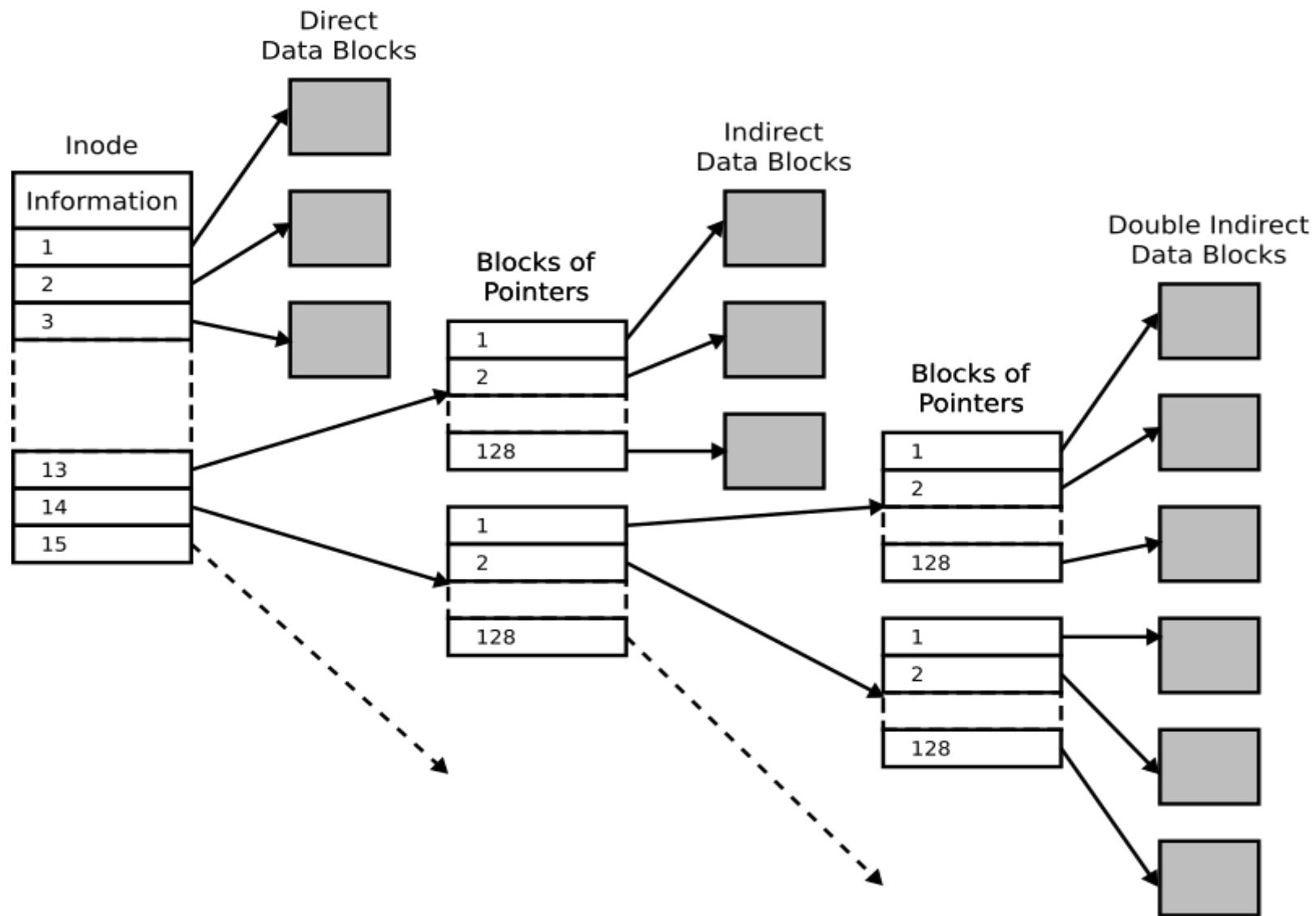
```
struct ext2_super_block {  
...  
    __le16 s_minor_rev_level; /* minor revision level */  
    __le32 s_lastcheck;      /* time of last check */  
    __le32 s_checkinterval; /* max. time between checks */  
    __le32 s_creator_os;    /* OS */  
    __le32 s_rev_level;     /* Revision level */  
    __le16 s_def_resuid;   /* Default uid for reserved blocks */  
    __le16 s_def_resgid;   /* Default gid for reserved blocks */  
    __le32 s_first_ino;    /* First non-reserved inode */  
    __le16 s_inode_size;   /* size of inode structure */  
    __le16 s_block_group_nr; /* block group # of this superblock */  
    __le32 s_feature_compat; /* compatible feature set */  
    __le32 s_feature_incompat; /* incompatible feature set */  
    __le32 s_feature_ro_compat; /* readonly-compatible feature set */  
    __u8  s_uuid[16];       /* 128-bit uuid for volume */  
    char s_volume_name[16]; /* volume name */  
    char s_last_mounted[64]; /* directory where last mounted */  
    __le32 s_algorithm_usage_bitmap; /* For compression */
```

```
struct ext2_super_block {  
...  
    __u8    s_prealloc_blocks; /* Nr of blocks to try to preallocate*/  
    __u8    s_prealloc_dir_blocks; /* Nr to preallocate for dirs */  
    __u16   s_padding1;  
/*  
 * Journaling support valid if EXT3_FEATURE_COMPAT_HAS_JOURNAL set.  
 */  
    __u8    s_journal_uuid[16]; /* uuid of journal superblock */  
    __u32   s_journal_inum;    /* inode number of journal file */  
    __u32   s_journal_dev;    /* device number of journal file */  
    __u32   s_last_orphan;    /* start of list of inodes to delete */  
    __u32   s_hash_seed[4];    /* HTREE hash seed */  
    __u8    s_def_hash_version; /* Default hash version to use */  
    __u8    s_reserved_char_pad;  
    __u16   s_reserved_word_pad;  
    __le32  s_default_mount_opts;  
    __le32  s_first_meta_bg;   /* First metablock block group */  
    __u32   s_reserved[190];   /* Padding to the end of the block */
```

```
struct ext2_group_desc
{
    __le32 bg_block_bitmap;      /* Blocks bitmap block */
    __le32 bg_inode_bitmap;     /* Inodes bitmap block */
    __le32 bg_inode_table;     /* Inodes table block */
    __le16 bg_free_blocks_count; /* Free blocks count */
    __le16 bg_free_inodes_count; /* Free inodes count */
    __le16 bg_used_dirs_count; /* Directories count */
    __le16 bg_pad;
    __le32 bg_reserved[3];
};
```

```
struct ext2_inode {  
    __le16 i_mode;      /* File mode */  
    __le16 i_uid;       /* Low 16 bits of Owner Uid */  
    __le32 i_size;      /* Size in bytes */  
    __le32 i_atime;     /* Access time */  
    __le32 i_ctime;     /* Creation time */  
    __le32 i_mtime;     /* Modification time */  
    __le32 i_dtime;     /* Deletion Time */  
    __le16 i_gid;       /* Low 16 bits of Group Id */  
    __le16 i_links_count; /* Links count */  
    __le32 i_blocks;    /* Blocks count */  
    __le32 i_flags;     /* File flags */
```

Inode in ext2



```
struct ext2_inode {  
    ...  
    union {  
        struct {  
            __le32 l_i_reserved1;  
        } linux1;  
        struct {  
            __le32 h_i_translator;  
        } hurd1;  
        struct {  
            __le32 m_i_reserved1;  
        } masix1;  
    } osd1;          /* OS dependent 1 */  
    __le32 i_block[EXT2_N_BLOCKS];/* Pointers to blocks */  
    __le32 i_generation; /* File version (for NFS) */  
    __le32 i_file_acl; /* File ACL */  
    __le32 i_dir_acl; /* Directory ACL */  
    __le32 i_faddr;   /* Fragment address */
```

```
struct ext2_inode {  
    ...  
    union {  
        struct {  
            __u8 l_i_frag; /* Fragment number */      __u8 l_i_fsize; /* Fragment size */  
            __u16 i_pad1;          __le16 l_i_uid_high; /* these 2 fields */  
            __le16 l_i_gid_high; /* were reserved2[0] */  
            __u32 l_i_reserved2;  
        } linux2;  
        struct {  
            __u8 h_i_frag; /* Fragment number */      __u8 h_i_fsize; /* Fragment size */  
            __le16 h_i_mode_high;      __le16 h_i_uid_high;  
            __le16 h_i_gid_high;  
            __le32 h_i_author;  
        } hurd2;  
        struct {  
            __u8 m_i_frag; /* Fragment number */      __u8 m_i_fsize; /* Fragment size */  
            __u16 m_pad1;          __u32 m_i_reserved2[2];  
        } masix2;  
    } osd2;           /* OS dependent 2 */
```

Ext2 FS Layout: Directory entry

	inode	rec_len	file_type	name_len	name				
0	21	12	1	2	.	\0	\0	\0	
12	22	12	2	2	.	.	\0	\0	
24	53	16	5	2	h	o	m	e	1 \0 \0 \0 \0
40	67	28	3	2	u	s	r	\0	
52	0	16	7	1	o	l	d	f	i 1 e \0
68	34	12	4	2	s	b	i	n	

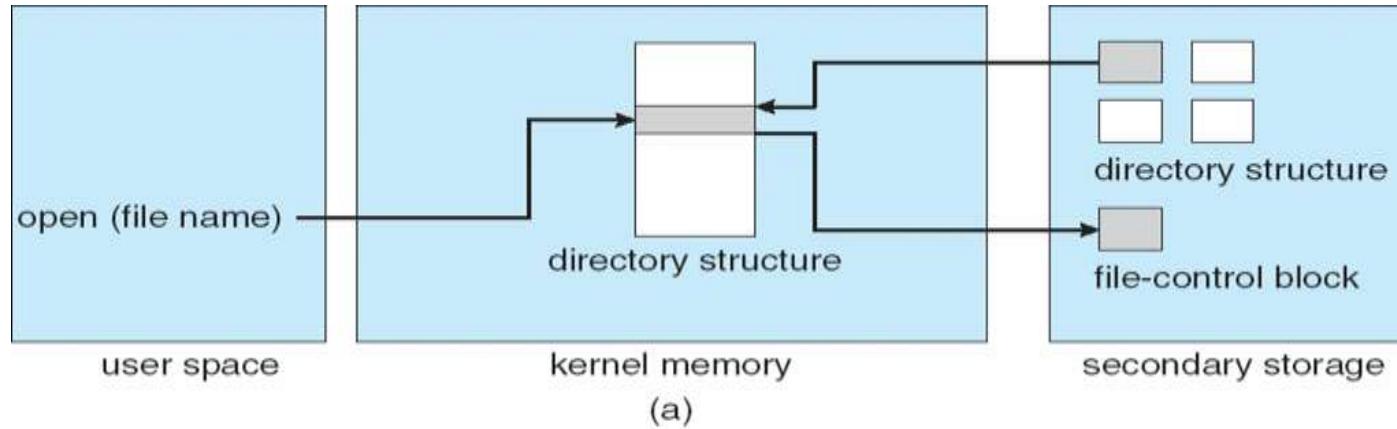
Let's see a program to read superblock of an ext2 file system.

Efficiency and Performance (and the risks created while trying to achieve it!)

In memory data structures

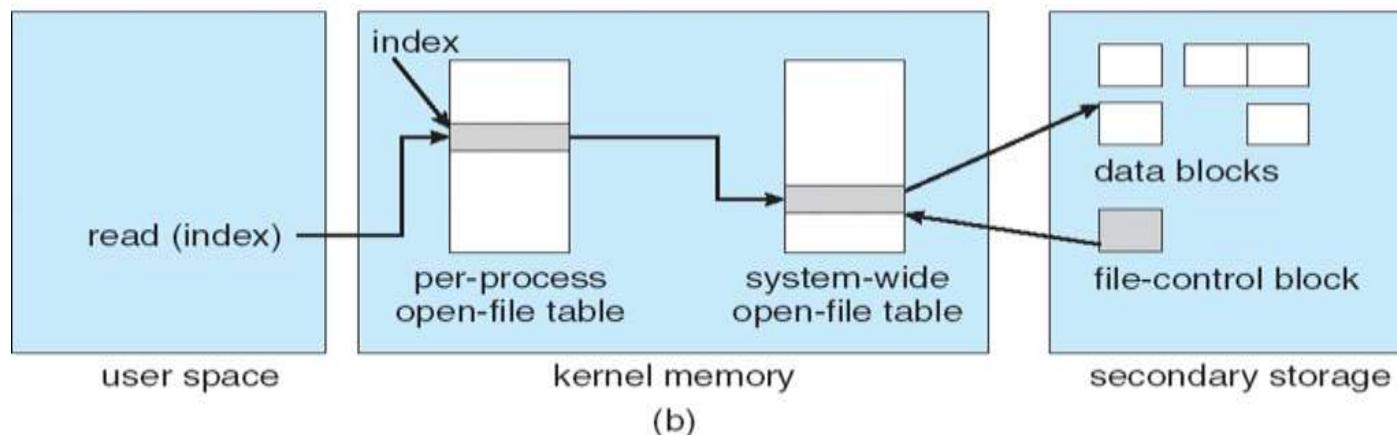
- Mount table
 - storing file system mounts, mount points, file system types
 - See next slide for “file” realated data structures
- Buffers
 - hold data blocks from secondary storage

In memory data structures: for open,read,write, ...



Open returns a file handle for

Data from read eventually comes



Efficiency

- Efficiency dependent on:
 - Disk allocation and directory algorithms
 - Types of data kept in file's directory entry
 - Pre-allocation or as-needed allocation of metadata structures
 - Fixed-size or varying-size data structures

Performance

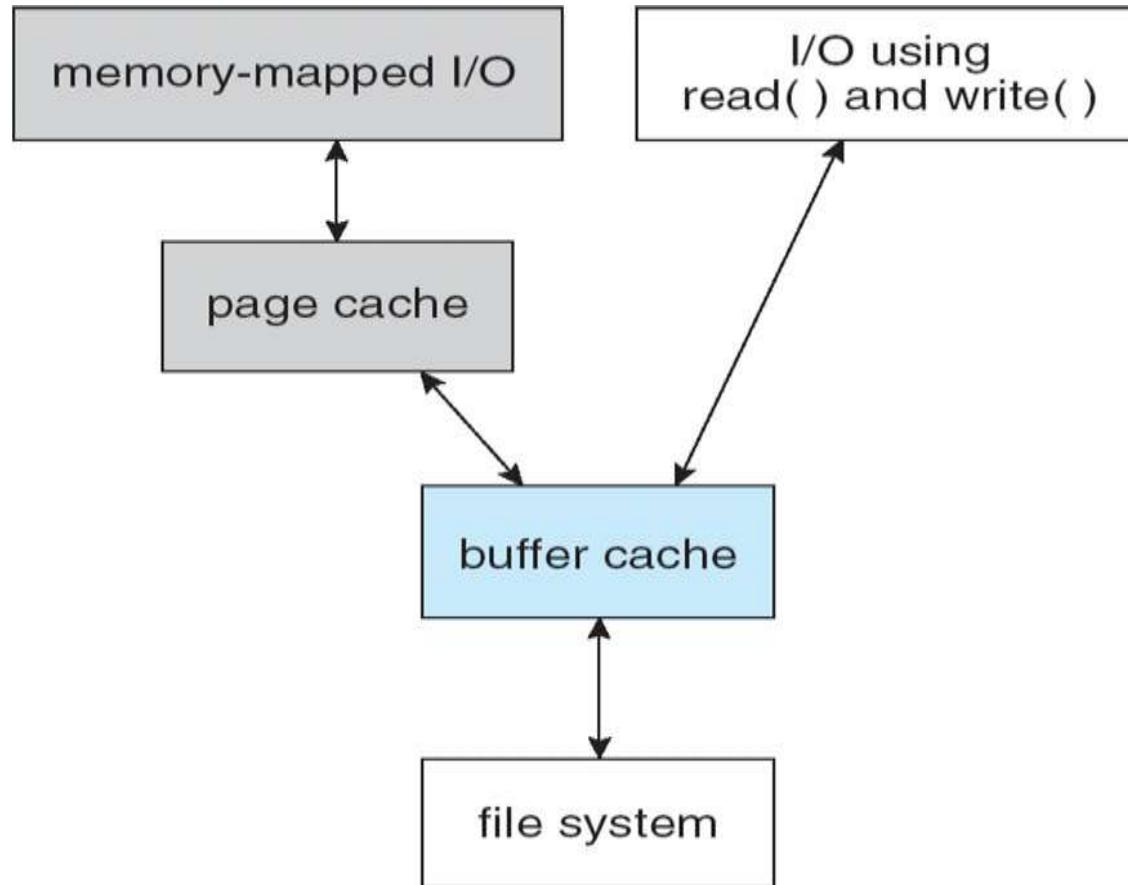
- Keeping data and metadata close together
- Buffer cache – separate section of main memory for frequently used blocks
- Synchronous writes sometimes requested by apps or needed by OS
- No buffering / caching – writes must hit disk before acknowledgement

A synchronous write is more common, bufferable

Page cache

- A page cache caches pages rather than disk blocks using virtual memory techniques and addresses
- Memory-mapped I/O uses a page cache
- Routine I/O through the file system uses the buffer (disk) cache
- This leads to the following figure

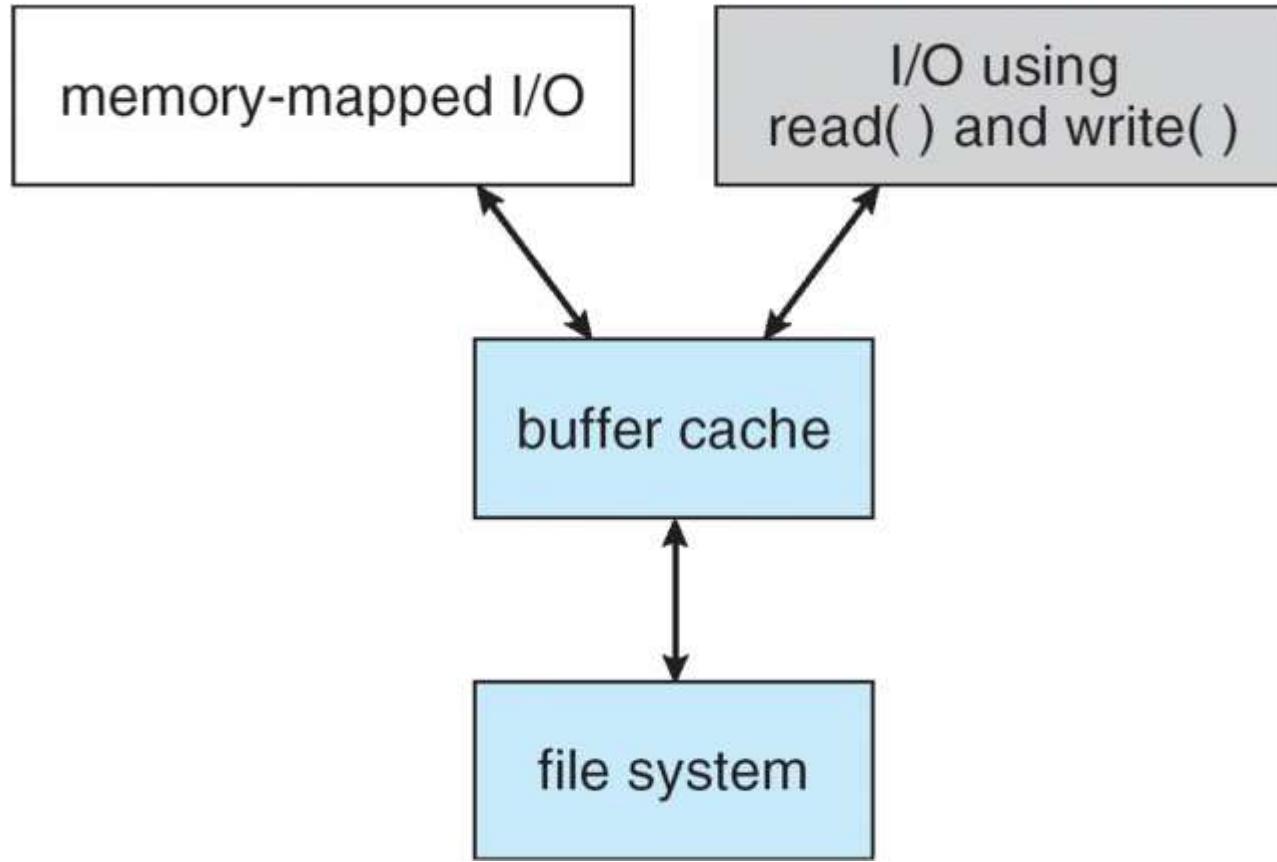
I/O Without a Unified Buffer Cache



Unified buffer cache

- A unified buffer cache uses the same page cache to cache both memory-mapped pages and ordinary file system I/O to avoid double caching
- But which caches get priority, and what replacement algorithms to use?

I/O Using a Unified Buffer Cache



Recovery

- Problem. Consider creating a file on ext2 file system.
- Following on disk data structures will/may get modified
- Directory data block, new directory data block, block bitmap, inode table, inode table bitmap, group descriptor, super block, data blocks for new file, more data block bitmaps, ...
- All cached in memory by OS

Recovery

- Consistency checking – compares data in directory structure with data blocks on disk, and tries to fix inconsistencies
- Can be slow and sometimes fails
- Use system programs to back up data from disk to another storage device (magnetic tape, other magnetic disk, optical)
- Recover lost file or disk by restoring data from

Log structured file systems

- Log structured (or journaling) file systems record each metadata update to the file system as a transaction
- All transactions are written to a log
- A transaction is considered committed once it is written to the log (sequentially)
- Sometimes to a separate device or section of disk
- However, the file system may not yet be updated

Journaling file systems

- ❑ Veritas FS
- ❑ Ext3, Ext4
- ❑ Xv6 file system!

File System Code

**open, read, write, close, pipe, fstat, chdir, dup,
mknod, link, unlink, mkdir,**

Files, Inodes, Buffers

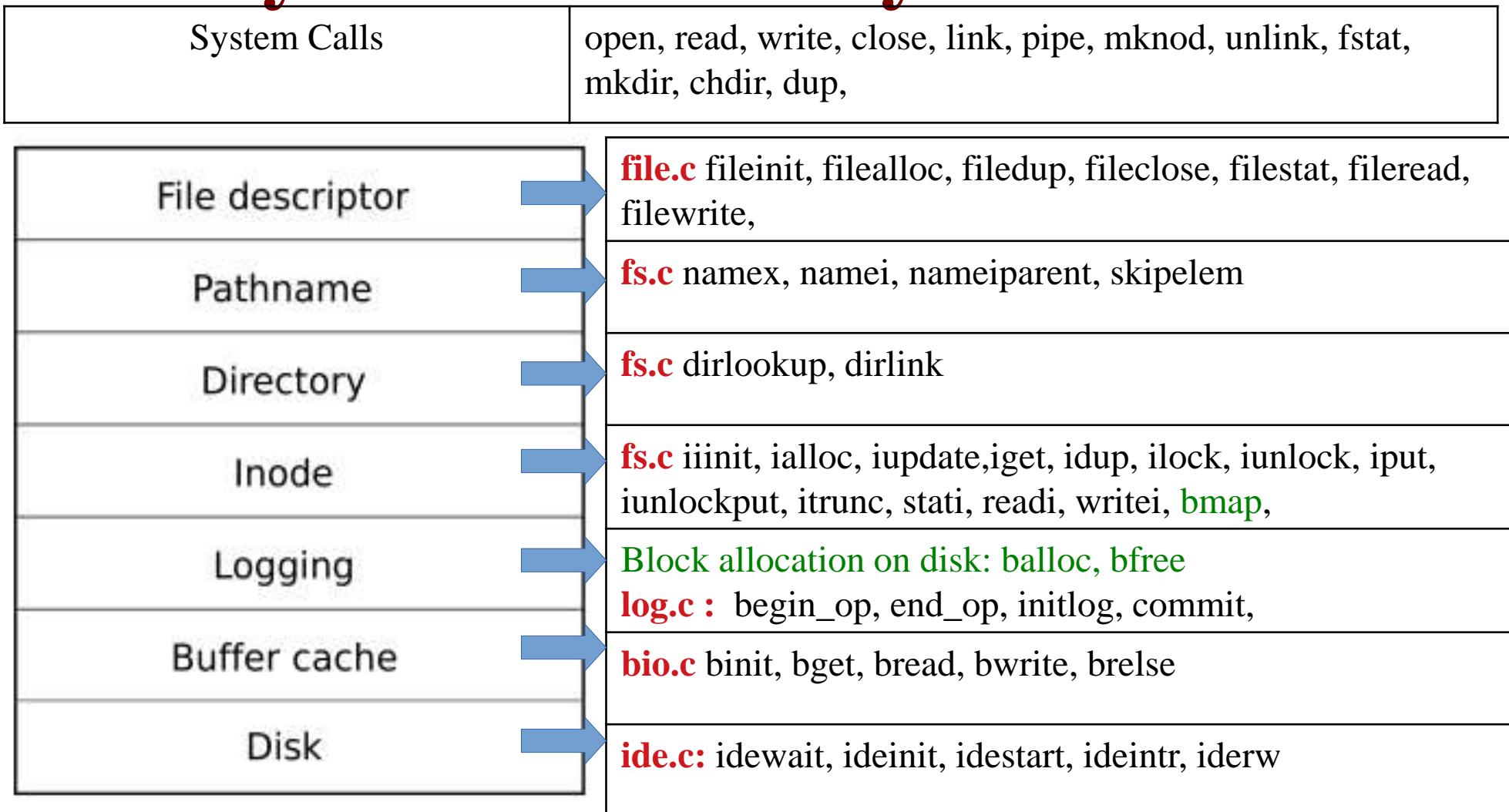
What we already know

- File system related system calls
- deal with ‘fd’ arrays (ofile in xv6). **open()** returns first empty index. open should ideally locate the inode on disk and initialize some data structures
- maintain ‘offsets’ within a ‘file’ to support sequential read/write
- **dup()** like system calls duplicate pointers in fd-array
- read/write like system calls, going through ‘ofile’ array, should locate data of file on disk
- We need functions to read/write from disk – that is

xv6 file handling code

- Is a very good example in ‘design’ of a layered and modular architecture
- Splits the entire work into different modules, and modules into functions properly
- The task of each function is neatly defined and compartmentalized

Layers of xv6 file system code

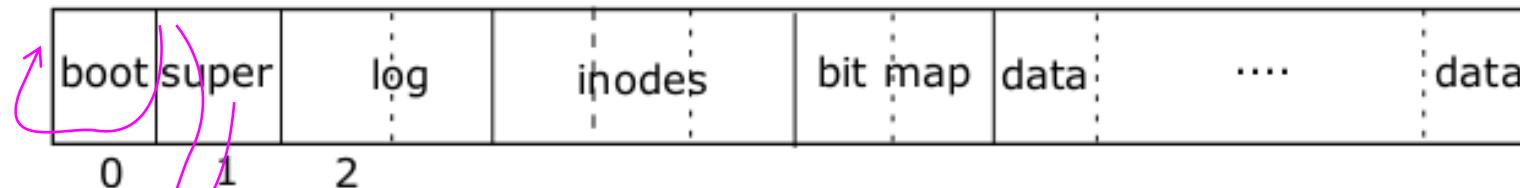


Normally, any upper layer can call any lower layer below

Abhijit: Block allocator should be considered as another Layer!

Layout of xv6 file system

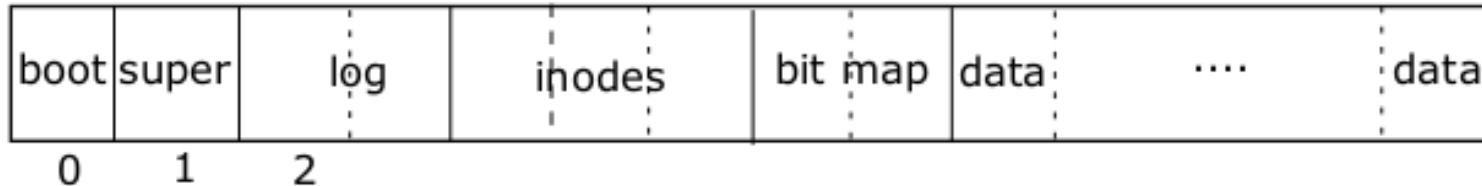
Pointer shown are conceptual



May see the code of `mkfs.c` to get insight into the layout

```
struct superblock {  
    uint size;      // Size of file system image (blocks)  
    uint nblocks;   // Number of data blocks  
    uint ninodes;   // Number of inodes.  
    uint nlog;      // Number of log blocks  
    uint logstart;  // Block number of first log block  
    uint inodestart; // Block number of first inode block  
    uint bmapstart; // Block number of first free map block  
};  
#define ROOTINO 1 // root i-number  
#define BSIZE 512 // block size
```

Layout of xv6 file system



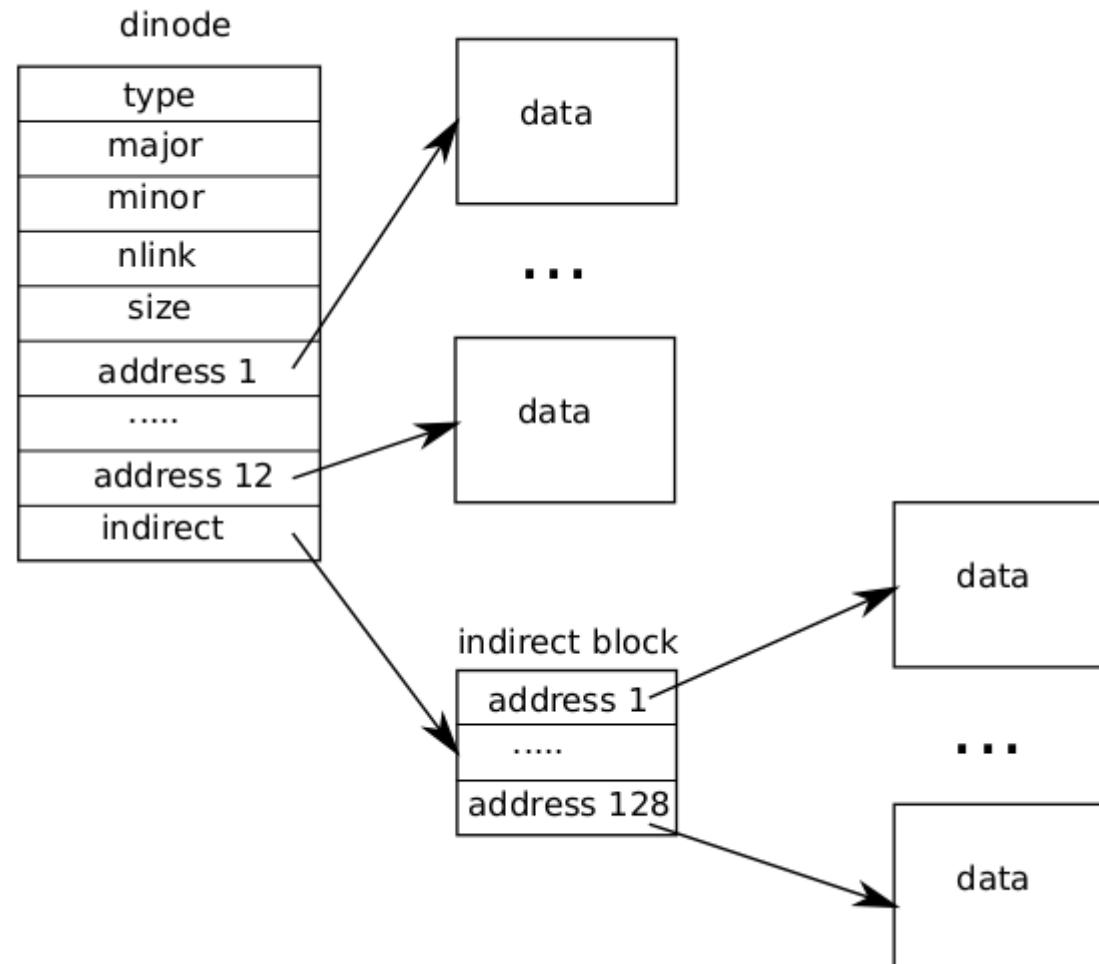
```
#define NDIRECT 12
#define NINDIRECT (BSIZE / sizeof(uint))
#define MAXFILE (NDIRECT + NINDIRECT)
// On-disk inode structure
struct dinode {
    short type;          // File type
    short major;         // Major device number (T_DEV only)
    short minor;         // Minor device number (T_DEV only)
    short nlink;         // Number of links to inode in file system
    uint size;           // Size of file (bytes)
    uint addrs[NDIRECT+1]; // Data block addresses
};


```

```
#define DIRSIZ 14
```

```
struct dirent {
    ushort inum;
    char name[DIRSIZ];
};
```

File on disk



Let's discuss lowest layer first

System Calls	open, read, write, close, link, pipe, mknod, unlink, fstat, mkdir, chdir, dup,
File descriptor	file.c fileinit, filealloc, filedup, fileclose, filestat, fileread, filewrite,
Pathname	fs.c namex, namei, nameiparent, skipelem
Directory	fs.c dirlookup, dirlink
Inode	fs.c iiinit, ialloc, iupdate, igeet, idup, ilock, iunlock, iput, iunlockput, itrunc, stati, readi, writei, bmap ,
Logging	Block allocation on disk: balloc , bfree log.c : begin_op, end_op, initlog, commit,
Buffer cache	bio.c binit, bget, bread, bwrite, brelse
Disk	ide.c : idewait, ideinit, idestart, ideintr, iderw

Normally, any upper layer can call any lower layer below

ide.c: idewait, ideinit, idestart, ideintr, iderw

static struct spinlock idelock;

static struct buf *idequeue;

static int havedisk1;

□ **ideinit**

□ **was called from main.c: main()**

□ **Initialized IDE controller by writing to certain ports**

□ **havedisk=1 setup**

□ **Initialize idelock**

□ **idewait**

ide.c: idewait, ideinit, idestart, ideintr, iderw

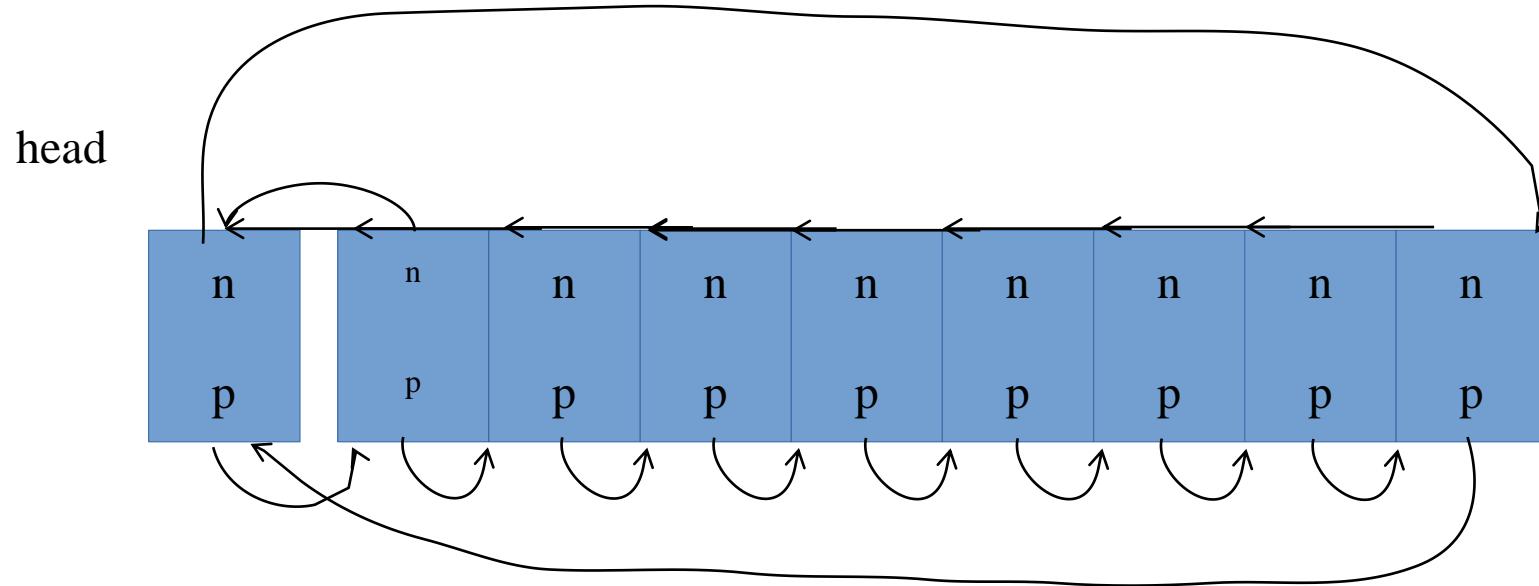
- **void idestart(buf *b)**
- **static void idestart(struct buf *b)**
- Calculate sector number on disk using b->blockno
- Issue a read/write command to IDE controller.
- (This is the first buf on **idequeue**)
- **ideintr**
 - Take **idelock**. Called on IDE interrupt (through alltraps()->trap())
 - Wakeup the process waiting on first buffer in **buffer *idequeue**;

Let's see buffer cache layer

System Calls	open, read, write, close, link, pipe, mknod, unlink, fstat, mkdir, chdir, dup,
File descriptor	file.c fileinit, filealloc, filedup, fileclose, filestat, fileread, filewrite,
Pathname	fs.c namex, namei, nameiparent, skipelem
Directory	fs.c dirlookup, dirlink
Inode	fs.c iiinit, ialloc, iupdate, igeet, idup, ilock, iunlock, iput, iunlockput, itrunc, stati, readi, writei, bmap ,
Logging	Block allocation on disk: balloc , bfree log.c : begin_op, end_op, initlog, commit,
Buffer cache	bio.c binit, bget, bread, bwrite, brelse
Disk	ide.c : idewait, ideinit, idestart, ideintr, iderw

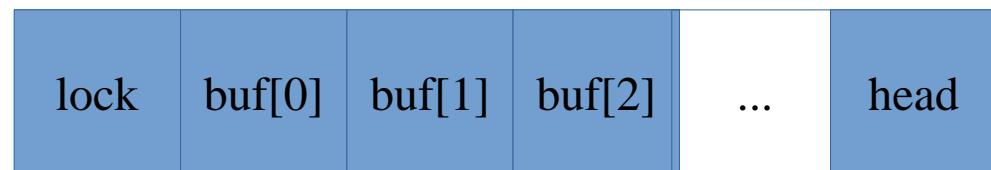
Normally, any upper layer can call any lower layer below

Reminder: After main() -> binit()



Conceptually Linear

Buffers keep moving



`struct bcache`

struct buf

```
struct buf {  
    int flags; // 0 or B_VALID or B_DIRTY  
    uint dev; // device number  
    uint blockno; // seq block number on device  
    struct sleeplock lock; // Lock to be held by process using it  
    uint refcnt; // Number of live accesses to the buf  
    struct buf *prev; // cache list  
    struct buf *next; // cache list  
    struct buf *qnext; // disk queue  
    uchar data[BSIZE]; // data 512 bytes  
};  
#define B_VALID 0x2 // buffer has been read from disk  
#define B_DIRTY 0x4 // buffer needs to be written to disk
```

buffer cache:

static struct buf* bget(uint dev, uint blockno)

- The bcache.head list is maintained on Most Recently Used (MRU) basis
- head.next is the Most Recently Used (MRU) buffer
- hence head.prev is the Least Recently Used (LRU)
- Look for a buffer with b->blockno = blockno and b->dev = dev
- Search the head.next list for existing buffer (MRU order)
- Else search the head.prev list for empty buffer
- panic() if found in-use or empty buffer

buffer cache:

struct buf* bread(uint dev, uint blockno)

struct buf*

**bread(uint dev, uint
blockno)**

{

struct buf *b;

b = bget(dev, blockno);

**if((b->flags &
B_VALID) == 0) {**

Recollect: **iderw** moves buf to tail of **idequeue**, calls **dequeue()** and **sleep()**

iderw(b);

void

bwrite(struct buf *b)

{

if(!holdingsleep(&b->lock))

panic("bwrite");

b->flags |= B_DIRTY;

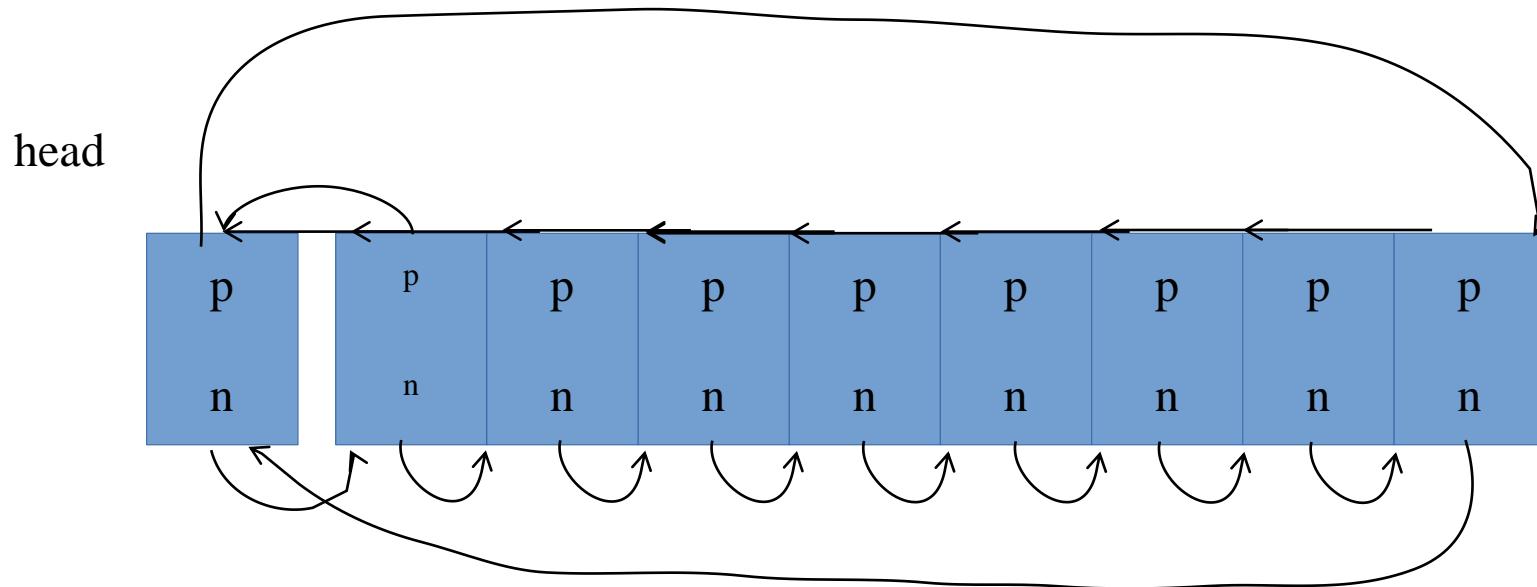
iderw(b);

}

buffer cache:
void brelse(struct buf *b)

- **release lock on buffer**
- **b->refcnt = 0**
- **If b->refcnt = 0**
- **Means buffer will no longer be used**
- **Move it to front of the front of bcache.head**

Overall in this diagram



Buffers keep moving to the front of the list and around
The list always contains **NBUF=30** buffers
head.next is always the MRU and **head.prev** is always LRU buffer

Let's see logging layer

System Calls	open, read, write, close, link, pipe, mknod, unlink, fstat, mkdir, chdir, dup,
File descriptor	file.c fileinit, filealloc, filedup, fileclose, filestat, fileread, filewrite,
Pathname	fs.c namex, namei, nameiparent, skipelem
Directory	fs.c dirlookup, dirlink
Inode	fs.c iiinit, ialloc, iupdate, igeet, idup, ilock, iunlock, iput, iunlockput, itrunc, stati, readi, writei, bmap ,
Logging	Block allocation on disk: balloc, bfree log.c : begin_op, end_op, initlog, commit,
Buffer cache	bio.c binit, bget, bread, bwrite, brelse
Disk	ide.c : idewait, ideinit, idestart, ideintr, ide rw Normally, any upper layer can call any lower layer below

log in xv6

- a mechanism of recovery from disk
- Concept: multiple write operations needed for system calls (e.g. ‘open’ system call to create a file in a directory)
- some writes succeed and some don’t leading to inconsistencies on disk
- In the log, all changes for a ‘transaction’ (an operation) are either written completely or not at all
- During recovery, completed operations can be

log in xv6

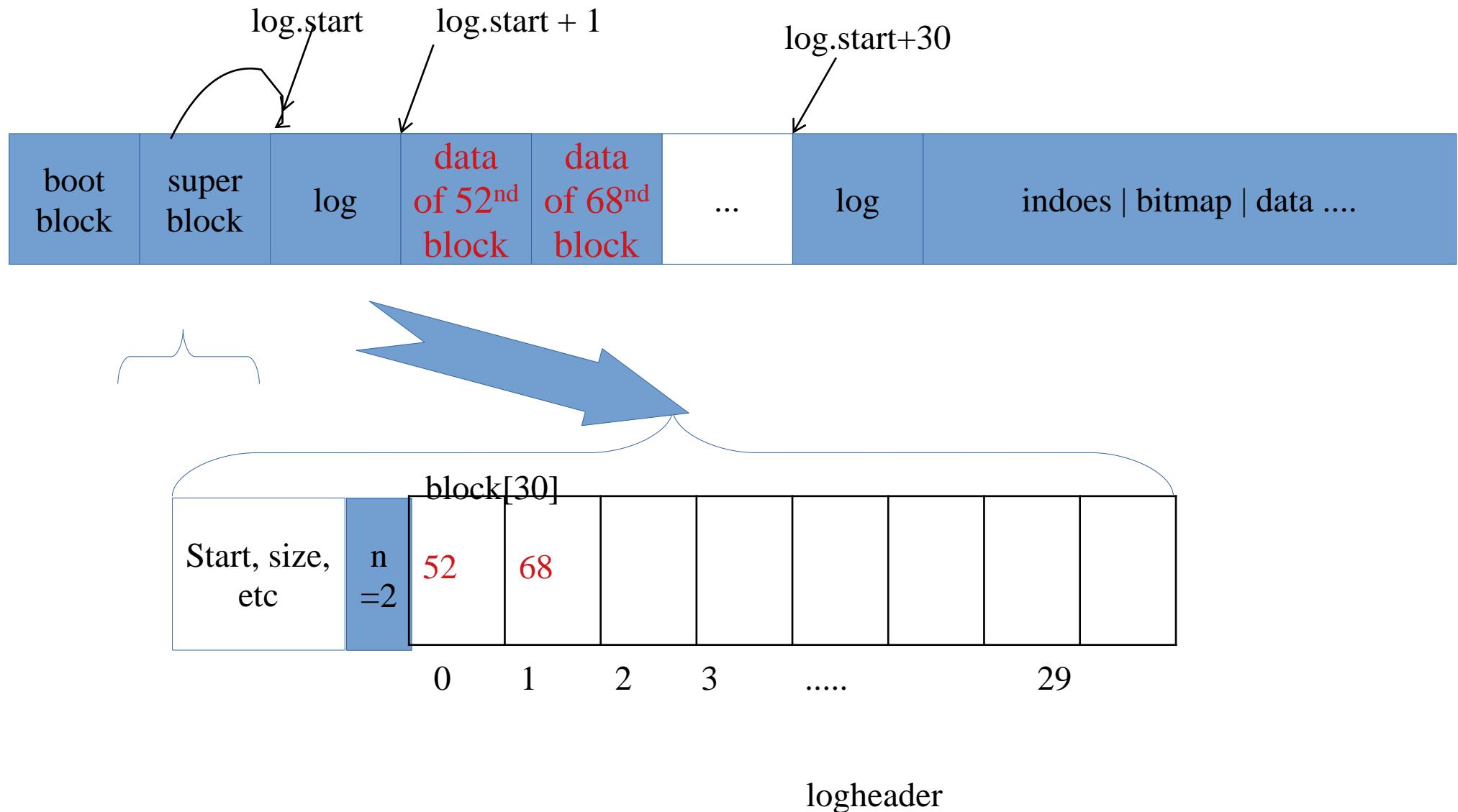
- xv6 system call does not directly write the on-disk file system data structures.
- A system call calls begin_op() at beginning and end_op() at end
 - begin_op() increments log.outstanding
 - end_op() decrements log.outstanding, and if it's 0, then calls commit()
- During the code of system call, whenever a buffer is modified, (and done with)
 - log_write() is called

log

```
struct logheader { // ON DISK
    int n; // number of entries in use in block[] below
    int block[LOGSIZE]; // List of block numbers
    stored
};

struct log { // only in memory
    struct spinlock lock;
    int start; // first log block on disk (starts with
    logheader)
```

log on disk



Typical use case of logging

```
/* In a system call code */  
/  
begin_op();
```

...

```
bp = bread(...);  
bp->data[...] = ...;
```

```
log_write(bp);
```

...

```
end_op();
```

prepare for logging.
Wait if logging system is
not ready or
'committing'.
++outstanding

read and get access to a
data block – as a buffer
modify buffer

note down this buffer for
writing, in log. proxy for
`bwrite()`. Mark

match colors in code and comments on right side

Example of calls to logging

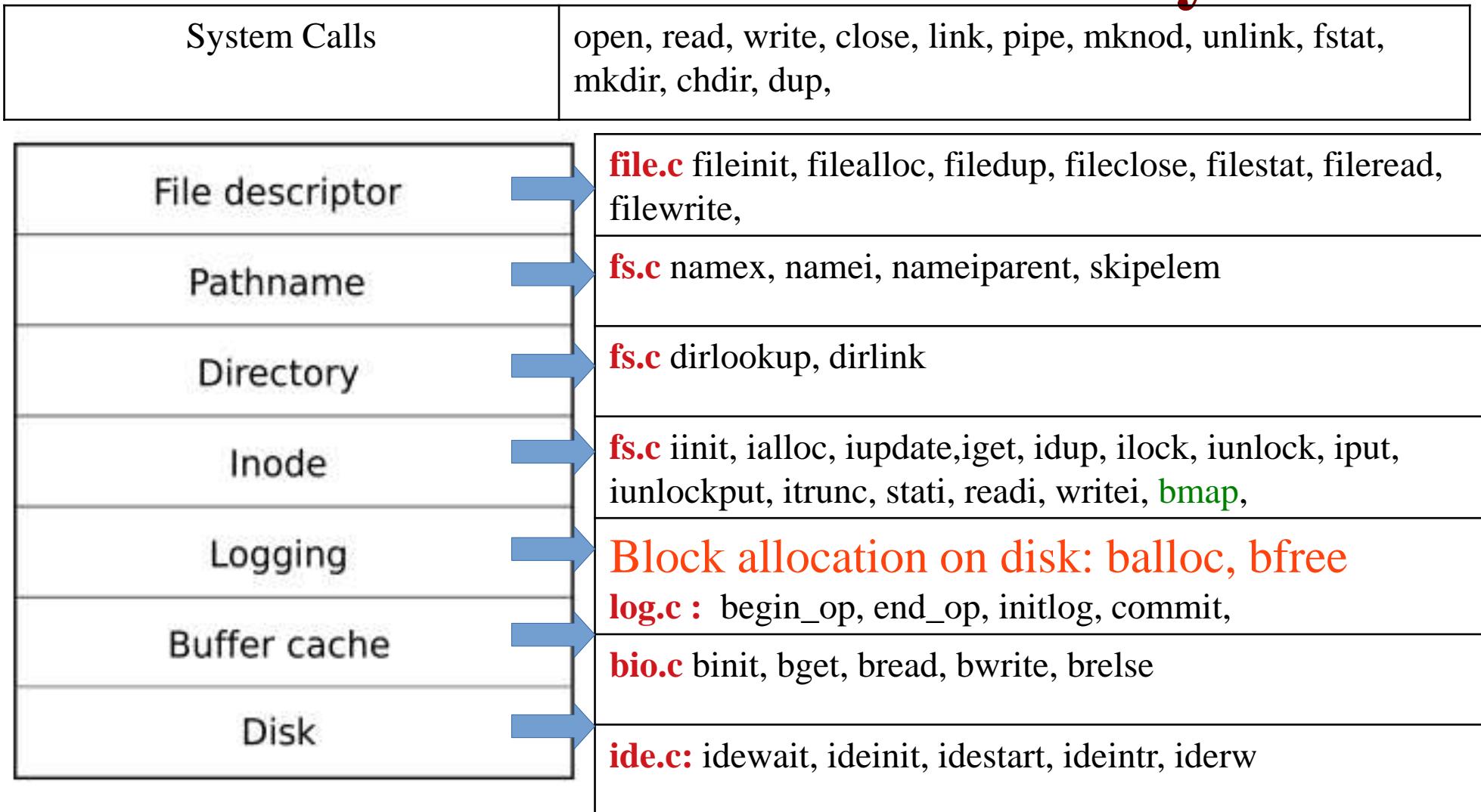
```
//file_write() code  
begin_op();  
ilock(f->ip);  
/*loop */ r = writei(f->ip, ...);  
iunlock(f->ip);  
end_op();
```

- each writei() in turn calls bread(), log_write() and brelse()
- also calls iupdate(ip) which also calls bread, log_write and brelse
- Multiple writes are combined between begin_op() and end_op()

Logging functions

- **Initlog()**
- Set fields in global **log.xyz** variables, using FS superblock
- Recovery if needed
- Called from first forkret()
- Following three called by FS code
- **begin_op(void)**
- **write_log(void)**
 - Called only from commit()
 - Use block numbers specified in **log.lh.block** and copy those blocks from memory to log-blocks
- **commit(void)**
 - Called only from end_op()

Let's see block allocation layer



Normally, any upper layer can call any lower layer below

Abhijit: Block allocator should be considered as another Layer!

allocating & deallocating blocks on DISK

- **balloc(devno)**
- looks for a block whose bitmap bit is zero, indicating that it is free.
- On finding updates the bitmap and returns the block.
- **balloc()** calls **bread()->bget** to get a block from disk in a buffer.
- Race prevented by the fact that the buffer cache
- **bfree(devno, blockno)**
- finds the right bitmap block and clears the right bit.
- Also calls **log_write()**

Let's see Inode Layer

System Calls	open, read, write, close, link, pipe, mknod, unlink, fstat, mkdir, chdir, dup,
File descriptor	file.c fileinit, filealloc, filedup, fileclose, filestat, fileread, filewrite,
Pathname	fs.c namex, namei, nameiparent, skipellem
Directory	fs.c dirlookup, dirlink
Inode	fs.c iinit, ialloc, iupdate, iguret, idup, ilock, iunlock, iput, iunlockput, itrunc, stati, readi, writei, bmap,
Logging	Block allocation on disk: balloc, bfree
Buffer cache	log.c : begin_op, end_op, initlog, commit,
Disk	bio.c binit, bget, bread, bwrite, brelse
	ide.c : idewait, ideinit, idestart, ideintr, ideintrw

On disk & in memory inodes

```
struct {  
    struct spinlock  
    lock;  
  
    struct inode  
    inode[NINODE];  
}  
icache;
```

```
// in-memory copy of an  
inode  
  
struct inode {  
    uint dev;           // Device  
    number  
    uint inum;         // Inode  
    number  
    int ref;           //  
    Reference count  
    struct sleeplock lock; //
```

In memory inodes

- Kernel keeps a subset of on disk inodes, those in use, in memory
- as long as ‘ref’ is >0
- The **iget** and **iput** functions acquire and release pointers to an inode, modifying the **ref** count.
- See the caller graph of **iget()**
- all those who call **iget()**
- Sleep lock in ‘inode’ protects
- fields in inode
- data blocks of inode

iget and iupdate

□ **iget**

- **searches for an existing/free inode in icache and returns pointer to one**
- **if found, increments ref and returns pointer to inode**
- **else gets empty inode , initializes, ref=1 and return**

□ **iupdate(inode *ip)**

- **read on disk block of inode**
- **get on disk inode**
- **modify it as specified in ‘ip’**
- **modify disk block of inode**
- **log_write(disk block of inode)**

No lock held after iget()

itrunc , iput

- **iput(ip)**
- **if ref is 1**
- **itrunc(ip)**
- **type = 0**
- **iupdate(ip)**
- **i->valid = 0 // free in memory**
- **else**
- **ref--**
- **itrunc(ip)**
- **write all data blocks of inode to disk**
- **using bfree()**
- **ip->size = 0**
- **Inode is freed from use**
- **iupdate(ip)**
- **called from iput() only when ‘ref’ becomes zero**

race in iput ?

- A concurrent thread might be waiting in ilock to use this inode
- and won't be prepared to find the inode is not longer allocated
- This is not possible.
Why?
- no way for a syscall to get a ref to a inode with ip->ref = 1

```
void  
iput(struct inode *ip)  
{  
    acquireSleep(&ip->lock);  
    if(ip->valid && ip->nlink == 0){  
        acquire(&icache.lock);  
        int r = ip->ref;
```

buffer and inode cache

- to read an **inode**, it's block must be read in a buffer
- So the buffer always contains a copy of the on-disk **dinode**
- duplicate copy in in-memory **inode**
- The inode cache is write-through,
- code that modifies a cached inode must immediately write it to disk with **iupdate**
- Inode may still exist in the buffer cache

allocating inode

- **ialloc(dev, type)**
- Loop over all disk inodes
 - read inode (from its block)
 - if it's free (note inum)
 - zero on disk inode
 - write on disk inode (as zeroes)
 - return iget(dev, inum)
- **ilock**
 - code must acquire ilock before using inode's data/fields
 - Illock reads inode if it's already not in memory

Trouble with `iput()` and crashes

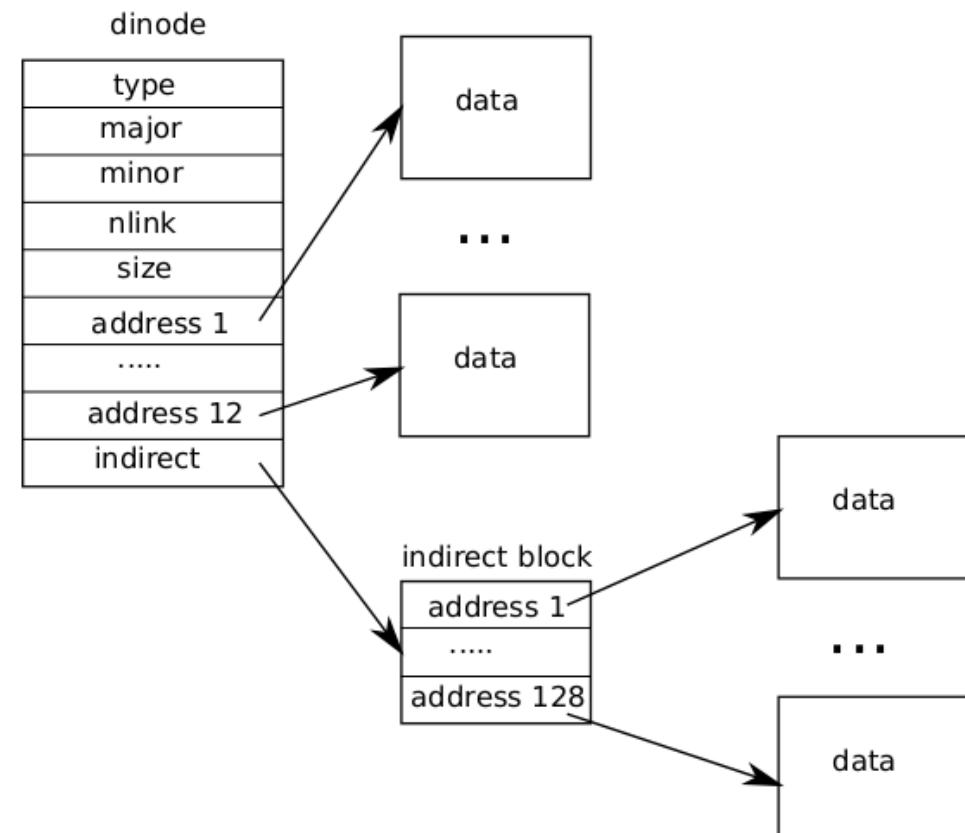
- **`iput()` doesn't truncate a file immediately when the link count for the file drops to zero, because**
- **some process might still hold a reference to the inode in memory: a process might still be reading and writing to the file, because it successfully opened it.**
- **if a crash happens before the last process closes the file descriptor for the file,**
- **then the file will be marked allocated on disk but no directory entry points to it**
- **Unsolved problem.**
- **How to solve it?**

Get Inode data: bmap(ip, bn)

- Allocate ‘bn’th block for the file given by inode ‘ip’

- Allocate block on disk and store it in either direct entries or block of indirect entries

- allocate block of indirect entries if needed using `balloc()`



writing/reading data at a given offset in file

**readi(struct inode *ip,
char *dst, uint off, uint
n)**

**writei(struct inode *ip,
char *src, uint off, uint
n)**

- Calculate the block number in file where ‘off’ belongs
- Read sufficient blocks to read ‘n’ bytes
 - using bread(), brelse()
- Call devsw.read if inode is a device Inode.
- Writei() also updates size if required

Reading Directory Layer

System Calls	open, read, write, close, link, pipe, mknod, unlink, fstat, mkdir, chdir, dup,
File descriptor	file.c fileinit, filealloc, filedup, fileclose, filestat, fileread, filewrite,
Pathname	fs.c namex, namei, nameiparent, skipelem
Directory	fs.c dirlookup, dirlink
Inode	fs.c iiinit, ialloc, iupdate, igeet, idup, ilock, iunlock, iput, iunlockput, itrunc, stati, readi, writei, bmap ,
Logging	Block allocation on disk: balloc , bfree log.c : begin_op, end_op, initlog, commit,
Buffer cache	bio.c binit, bget, bread, bwrite, brelse
Disk	ide.c : idewait, ideinit, idestart, ideintr, iderw

directory entry

```
#define DIRSIZ 14
```

```
struct dirent {  
    ushort inum;  
    char name[DIRSIZ];  
};
```

**Data of a directory file is a sequence of such entries.
To find a name, just get all the data blocks and
search the name**

struct inode*

dirlookup(struct inode *dp, char *name, uint *poff)

- Given a pointer to directory inode (dp), name of file to be searched
- return the pointer to inode of that file (NULL if not found)
- set the ‘offset’ of the entry found, inside directories data blocks, in poff
- How was ‘dp’ obtained? Who should be calling dirlookup? Why is poff returned?
- During resolution of pathnames?
- Code: call readi() to get data of dp, search name in it, name comes with inode num - iget() that inode

int
dirlink(struct inode *dp, char *name, uint inum)

- Create a new entry for ‘name’ _ ‘inum’ in directory given by ‘dp’
- inode number must have been obtained before calling this. How to do that?
- Use **dirlookup()** to verify entry does not exist!
- Get empty slot in directory’s data block
- Make directory entry
- Update directory inode! **writei()**

namex

- Called by namei(), or nameiparent()
- Just iteratively split a path using “/” separator and get inode for last component
- iget() root inode, then
 - Repeatedly calls
 - split on “/”, dirlookup() for next component

races in namex()

- Crucial. Called so many times!
- one kernel thread is looking up a pathname
another kernel thread may be changing the
directory by calling unlink
- when executing dirlookup in namex, the lookup
thread holds the lock on the directory and
dirlookup() returns an inode that was obtained
using iget.
- Deadlock? next points to the same inode as ip
when looking up "..". Locking next before releasing
the lock on ip would result in a deadlock.

File descriptor layer code

System Calls	open, read, write, close, link, pipe, mknod, unlink, fstat, mkdir, chdir, dup,
File descriptor	file.c fileinit, filealloc, filedup, fileclose, filestat, fileread, filewrite,
Pathname	fs.c namex, namei, nameiparent, skipelem
Directory	fs.c dirlookup, dirlink
Inode	fs.c iiinit, ialloc, iupdate, igeet, idup, ilock, iunlock, iput, iunlockput, itrunc, stati, readi, writei, bmap ,
Logging	Block allocation on disk: balloc , bfree
Buffer cache	log.c : begin_op, end_op, initlog, commit,
Disk	bio.c binit, bget, bread, bwrite, brelse
	ide.c : idewait, ideinit, idestart, ideintr, ide rw

data structures related to “file” layer

```
struct file {  
    enum { FD_NONE,  
FD_PIPE, FD_INODE }  
    type;  
    int ref; // reference  
    count  
    char readable;  
    char writable;  
    struct pipe *pipe; //  
    used only if it works as a
```

```
struct proc {  
    ...  
    struct file  
    *ofile[NOFILE]; // Open  
    files per process  
    ...  
}  
struct {  
    struct spinlock lock;
```

Multiple processes accessing same file.

- Each will get a different ‘struct file’
- but share the inode !
- different offset in struct file, for each process
- Also true, if same process opens file many times
- File can be a PIPE (more later)
- what about STDIN, STDOUT, STDERR files ?
- Figure out!
- ref
- used if the file was ‘duped’ or process forked . in that case the ‘struct file’ is shared

file layer functions

- **filealloc**
- find an empty struct file in ‘ftable’ and return it
- set ref = 1
- **filedup(file *)**
- simply ref++
- **fileclose**
 - --ref
 - if ref = 0
 - free struct file
 - iput() / pipeclose()
 - note – transaction if iput() called
 - **filestat**
 - simply return fields from inode, after holding

file layer functions

- **fileread**
 - call readi() or piperead()
 - readi() later calls device-read or inode read (using bread())
- **filewrite**
 - call pipewrite() or writei()
 - writei() is called in a loop, within a transaction
- Why does readi() call read on the device , why not fileread() itself call device read ?

pipes

```
struct pipe {  
    struct spinlock lock;  
    char data[PIPE_SIZE];  
    uint nread;  
    // number of bytes read  
    uint nwrite;  
    // number of bytes  
    written  
    int readopen;
```

- functions
 - pipealloc
 - pipeclose
 - piperead
 - pipewrite

pipes

- pipealloc
- allocate two struct file
- allocate pipe itself using kalloc (it's a big structure with array)
- init lock
- initialize both struct file as 2 ends (r/w)
- pipewrite
 - wait if pipe full
 - write to pipe
 - wakeup processes waiting to read
- piperead
 - wait if no data
 - read from pipe
 - wakeup processes waiting to write

Further to reading system call code now

- Now we are ready to read the code of system calls on file system
- sys_open, sys_write, sys_read , etc.
- Advise: Before you read code of these, contemplate on what these functions should do using the functions we have studied so far.
- Also think of locks that need to be held.

File Systems

Abhijit A M

abhijit.comp@coep.ac.in

Introduction

- Human end user's view of file system on a modern desktop operating system
- Files, directories(folders), hierarchy – acyclic graph like structure
- Windows Vs Linux logical organization: multiple partitions (C:, D:,etc.), vs single logical namespace starting at “/”

Introduction

- Secondary and Tertiary memory
- Hard disks, Pen drives, CD-ROMs, DVDs, Magnetic Tapes, Portable disks, etc. Used for storing files
- Each comes with a hardware “controller” that acts as an intermediary in the hardware/software boundary
- IDE, SATA, SCSI, SAS, etc. Protocols : Different types of cables, speeds, signaling mechanisms

Introduction

- OS and File system
- OS bridges the gap between end user and stoage hardware controller
- Provides data structure to map the logical view of end users onto disk storage
- Essentially an implementation of the acyclic graph on the sequential sector-based disk storage
- Both in memory and on-disk

What we are going to learn

- The operating system interface (system calls, commands/utilities) for accessing files in a file-system
- Design aspects of OS to implement the file system

What is a file?

- A sequence of bytes , with
- A name
- Permissions
- Owner
- Timestamps,
- Etc.
- Types: Text files, binary files

file type	usual extension	function
executable	exe, com, bin or none	ready-to-run machine-language program
object	obj, o	compiled, machine language, not linked
source code	c, cc, java, pas, asm, a	source code in various languages
batch	bat, sh	commands to the command interpreter
text	txt, doc	textual data, documents
word processor	wp, tex, rtf, doc	various word-processor formats
library	lib, a, so, dll	libraries of routines for programmers
print or view	ps, pdf, jpg	ASCII or binary file in a format for printing or viewing
archive	arc, zip, tar	related files grouped into one file, sometimes compressed, for archiving or storage
multimedia	mpeg, mov, rm, mp3, avi	binary file containing audio or A/V information

What is a file?

- The sequence of bytes can be interpreted to be
 - Just a sequence of bytes
 - E.g. a text file
 - Sequence of records/structures
 - E.g. a file of student records
 - A complexly organized, collection of records and bytes

File attributes

- Run

- \$ ls -l

- on Linux

- To see file listing with different attributes

- Different OSes and file-systems provide different sets of file attributes

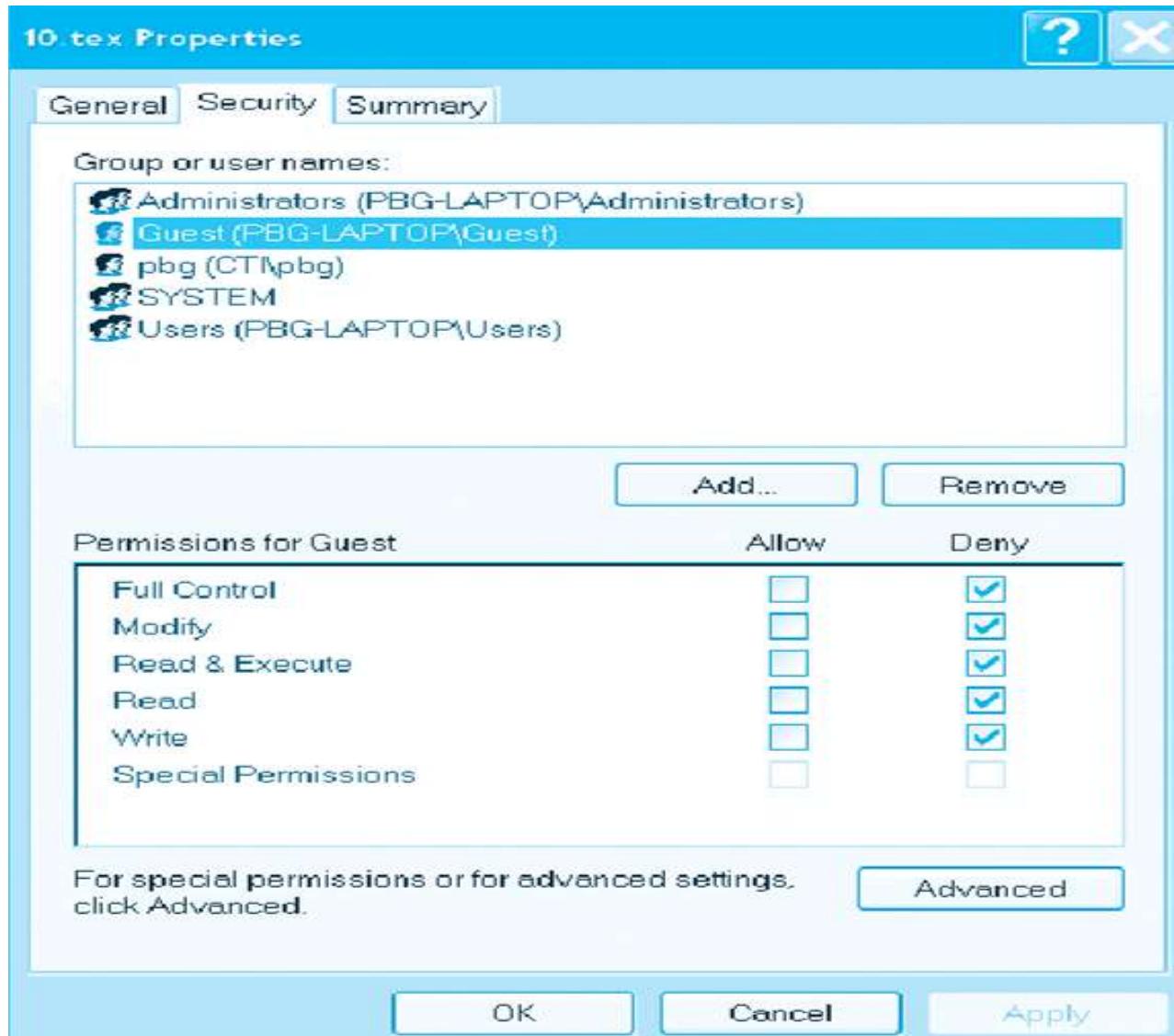
- Some attributes are common to most, while some

File protection attributes

- File owner/creator should be able to control:
 - what can be done by whom
 - Types of access
 - Read
 - Write
- Linux file permissions
 - For owner, group and others
 - Read, write and execute
 - Total 9 permissions

A Sample UNIX Directory Listing

-rw-rw-r--	1	pbg	staff	31200	Sep 3 08:30	intro.ps
drwx-----	5	pbg	staff	512	Jul 8 09:33	private/
drwxrwxr-x	2	pbg	staff	512	Jul 8 09:35	doc/
drwxrwx---	2	pbg	student	512	Aug 3 14:13	student-proj/
-rw-r--r--	1	pbg	staff	9423	Feb 24 2003	program.c
-rwxr-xr-x	1	pbg	staff	20471	Feb 24 2003	program
drwx--x--x	4	pbg	faculty	512	Jul 31 10:31	lib/
drwx-----	3	pbg	staff	1024	Aug 29 06:52	mail/
drwxrwxrwx	3	pbg	staff	512	Jul 8 09:35	test/



Windows XP Access List

Access methods

- OS system calls may provide two types of access to files
 - Direct Access
 - read n
 - write n
 - position to n
 - Sequential Access
 - read next
 - write next
 - reset
 - no read after

Device Drivers

- Hardware manufacturers provide “hardware controllers” for their devices
- Hardware controllers can operate the hardware, as instructed
- Hardware controllers are instructed by writing to particular I/O ports using CPU’s machine instructions
- This bridges the hardware-software gap

Disk device driver

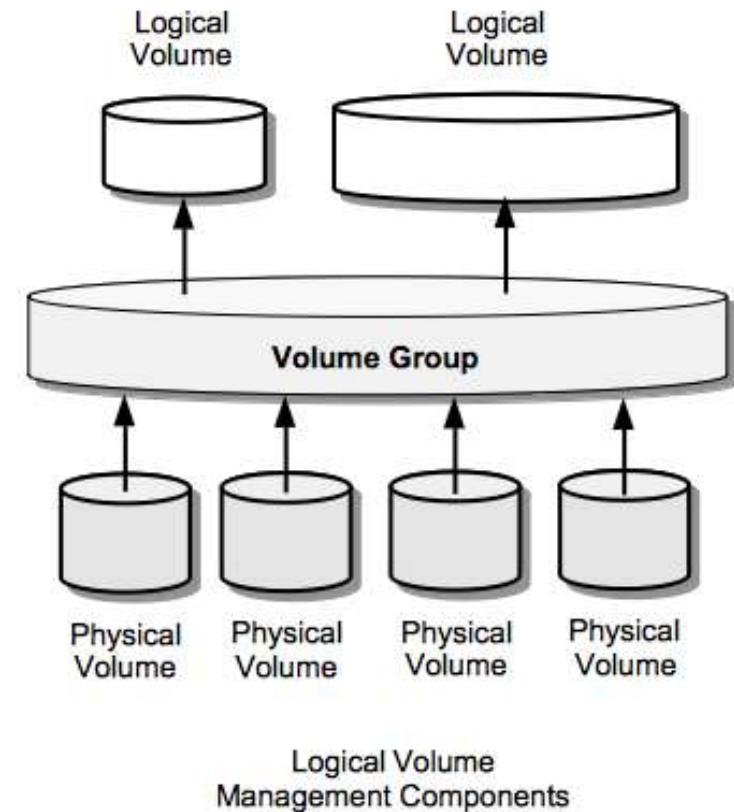
- OS views the disk as a logical sequence of blocks
- OS's assumed block size may be > sector size
- OS Talks to disk controller
- Helps the OS Convert it's view of “logical block” of the disk, into physical sector numbers
- Acts as a translator between rest of OS and hardware controller

OS's job now

- To implement the logical view of file system as seen by end user
- Using the logical block-based view offered by the device driver

Volume Managers

- Special type of kernel device drives, which reside on top of disk device drivers
- Provide a more abstract view of the underlying hardware
- E.g. Can combine two physical hard disks, and

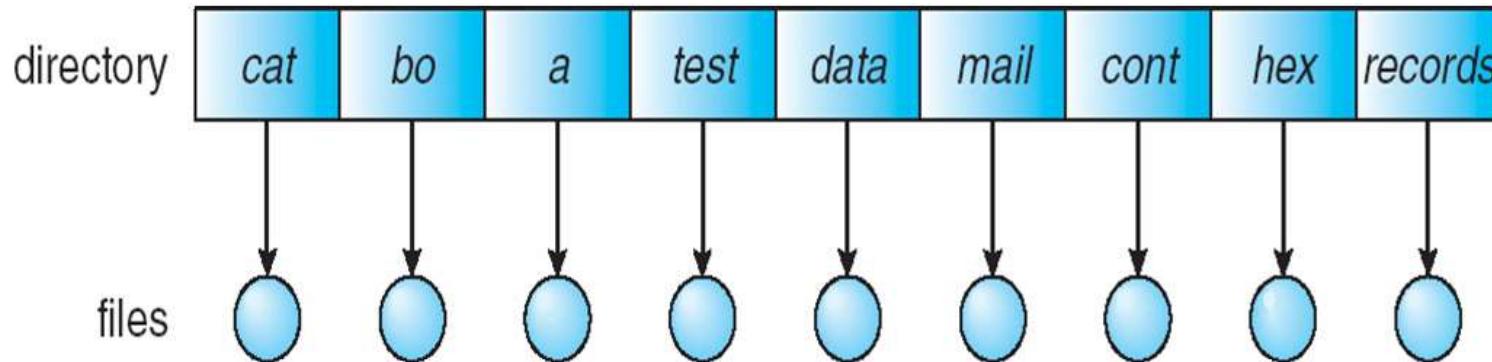


Formatting

- Physical hard disk divided into partitions
- Partitions also known as minidisks, slices
- A raw disk partition is accessible using device driver – but no block contains any data !
- Like an un-initialized array, or sectors/blocks
- Formatting
- Creating an initialized data structure on the partition, so that it can start storing the acyclic

Different types of “layouts”

Single level directory

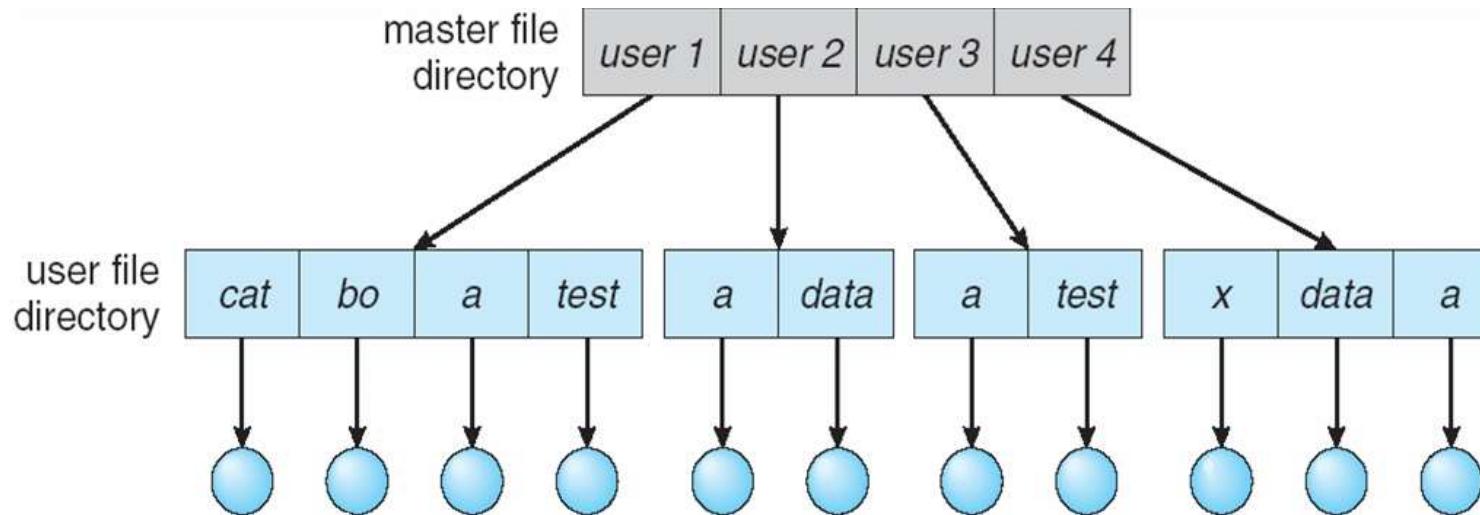


Naming problem

Grouping problem

Different types of “layouts”

Two level directory



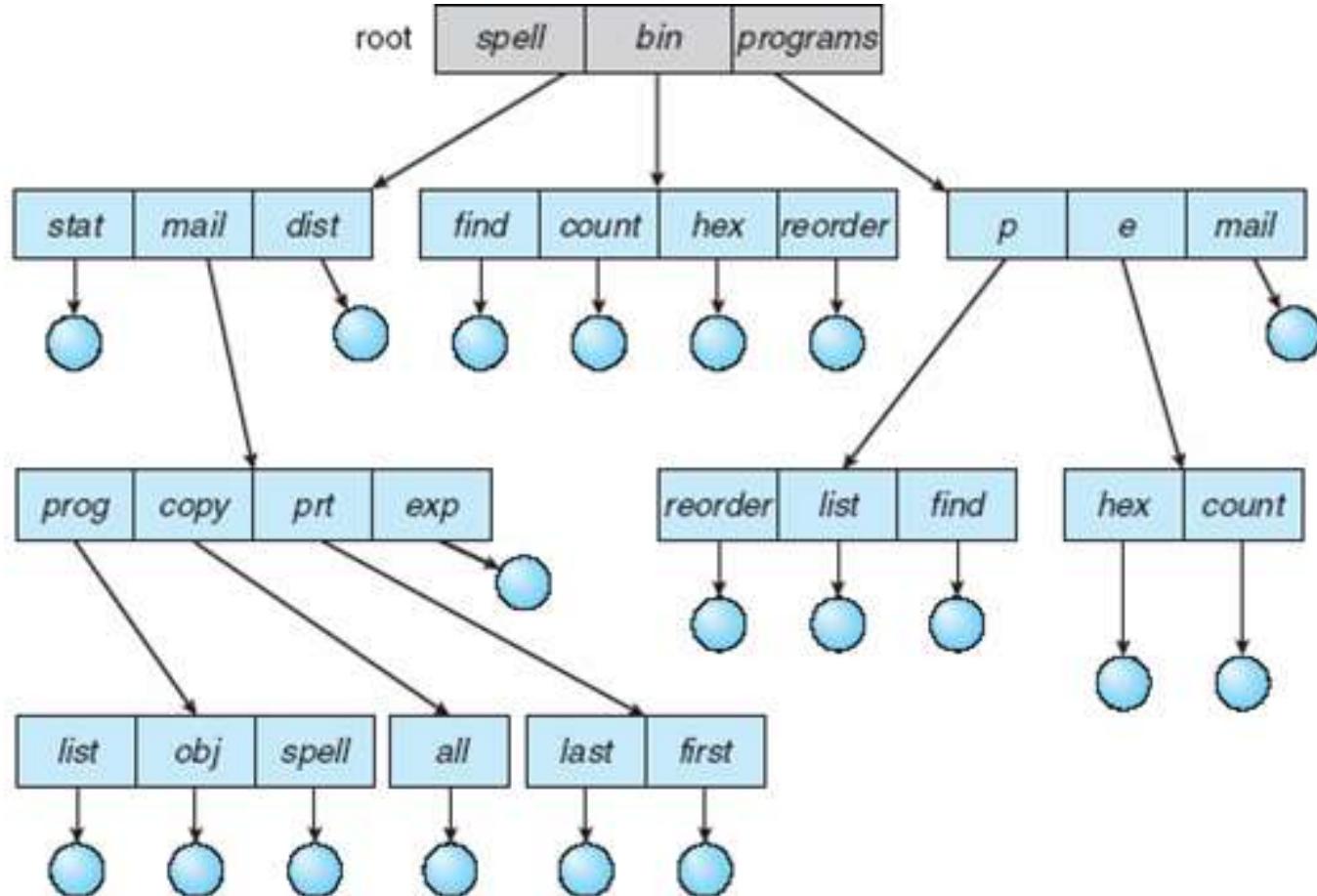
Path name

Can have the same file name for different user

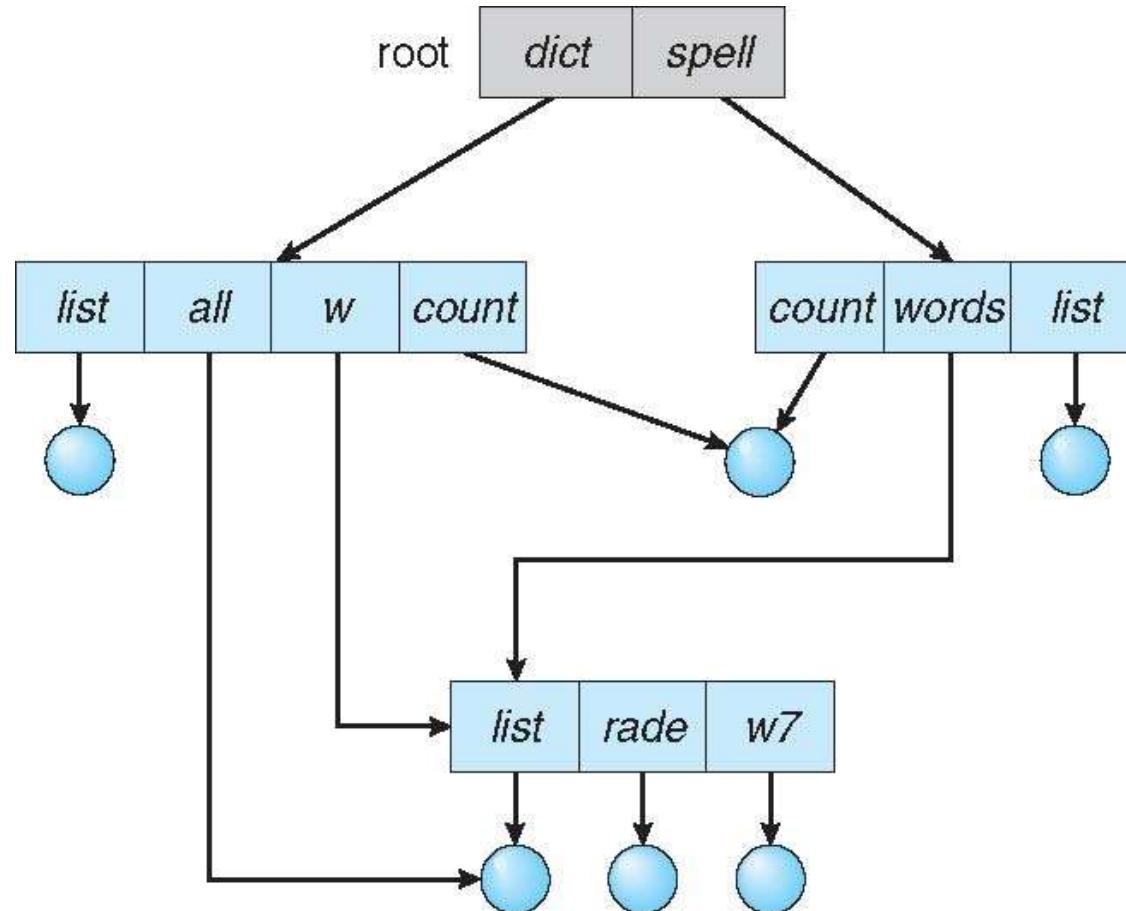
Efficient searching

No grouping capability

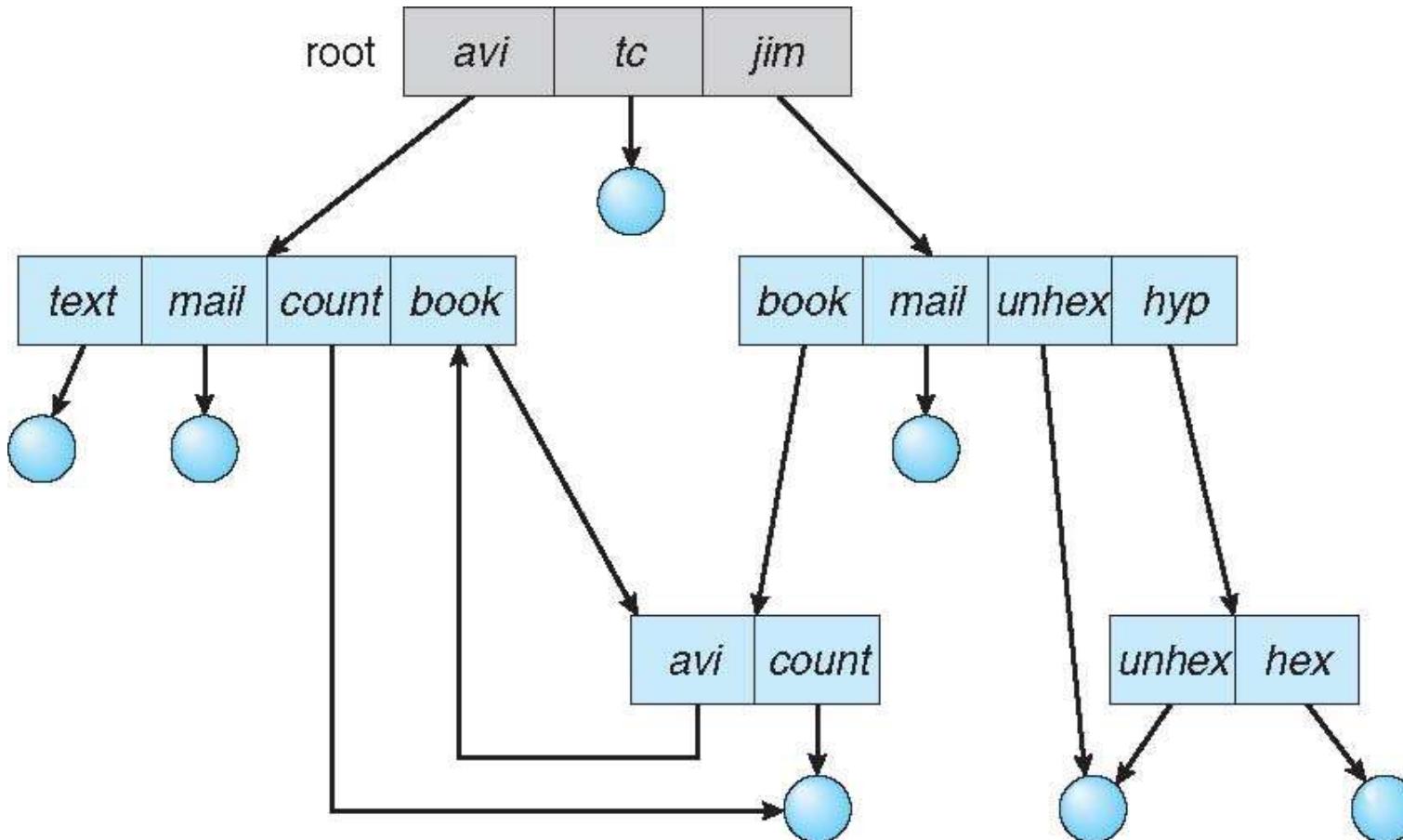
Tree Structured directories



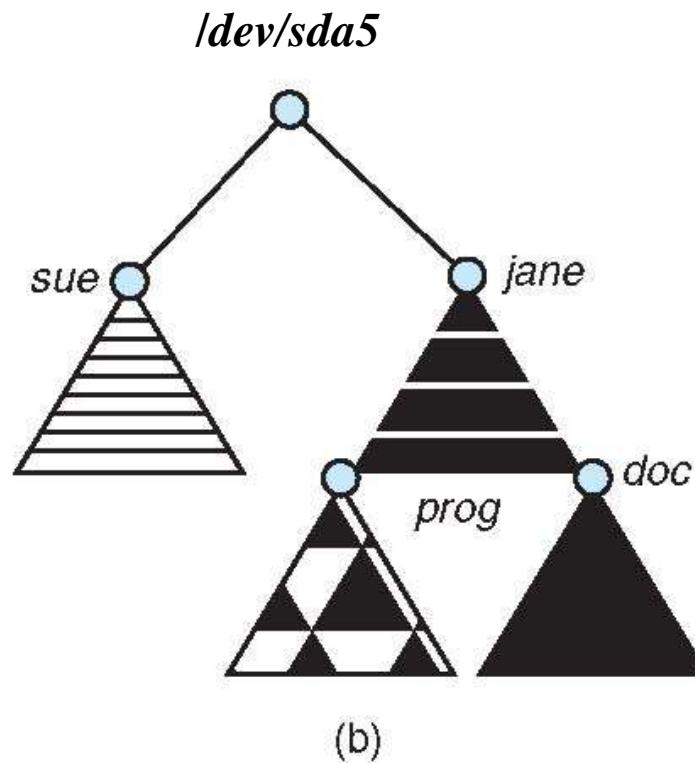
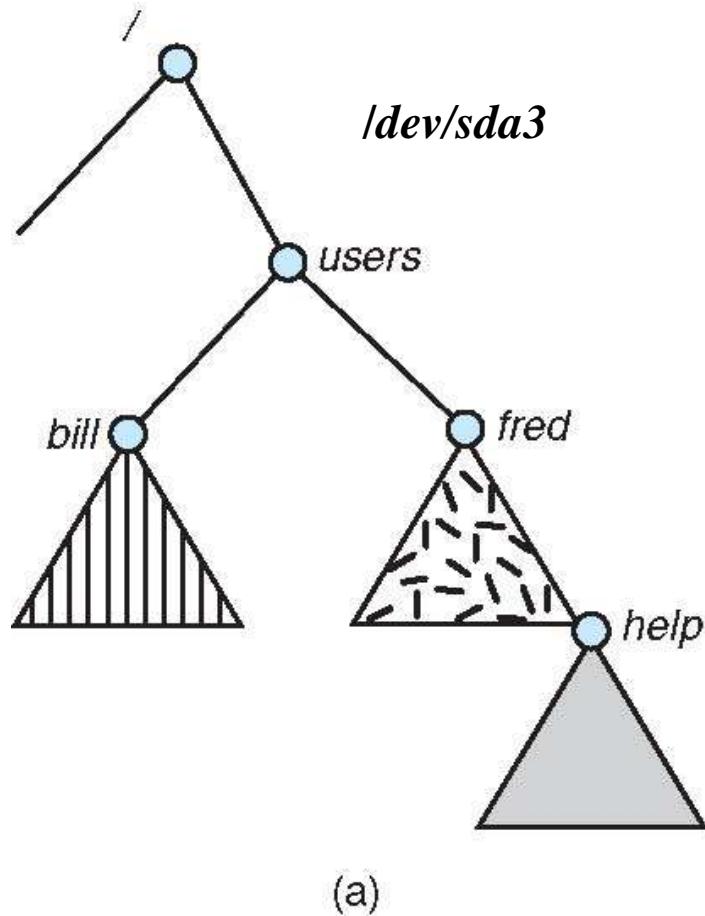
Acyclic Graph Directories



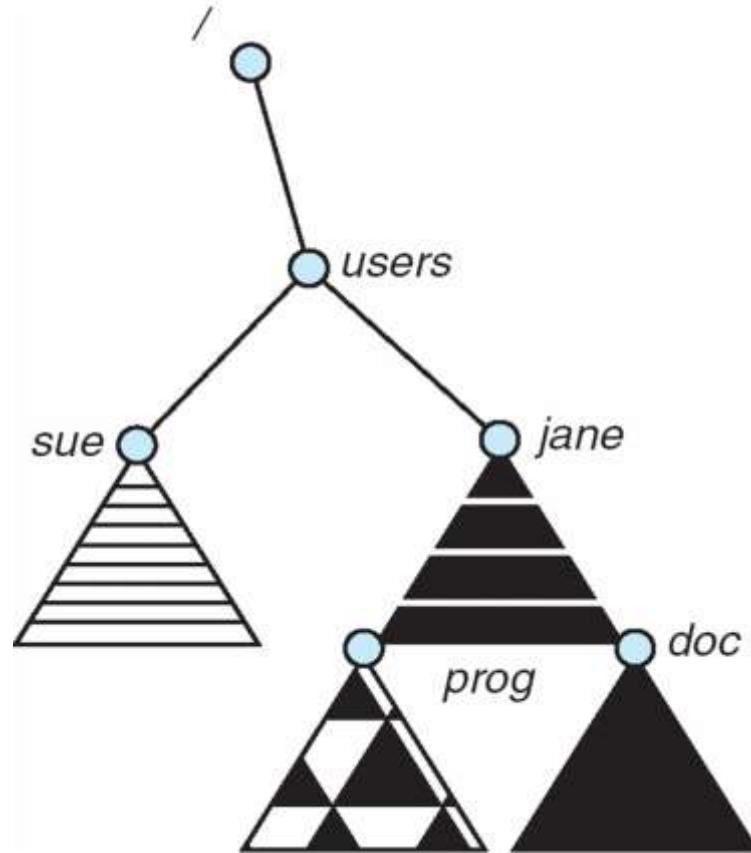
General Graph directory



Mounting of a file system: before



Mounting of a file system: after



`$sudo mount /dev/sda5 /users`

Remote mounting: NFS

- Network file system

- \$ sudo mount 10.2.1.2:/x/y /a/b

- The /x/y partition on 10.2.1.2 will be made available under the folder /a/b on this computer

File sharing semantics

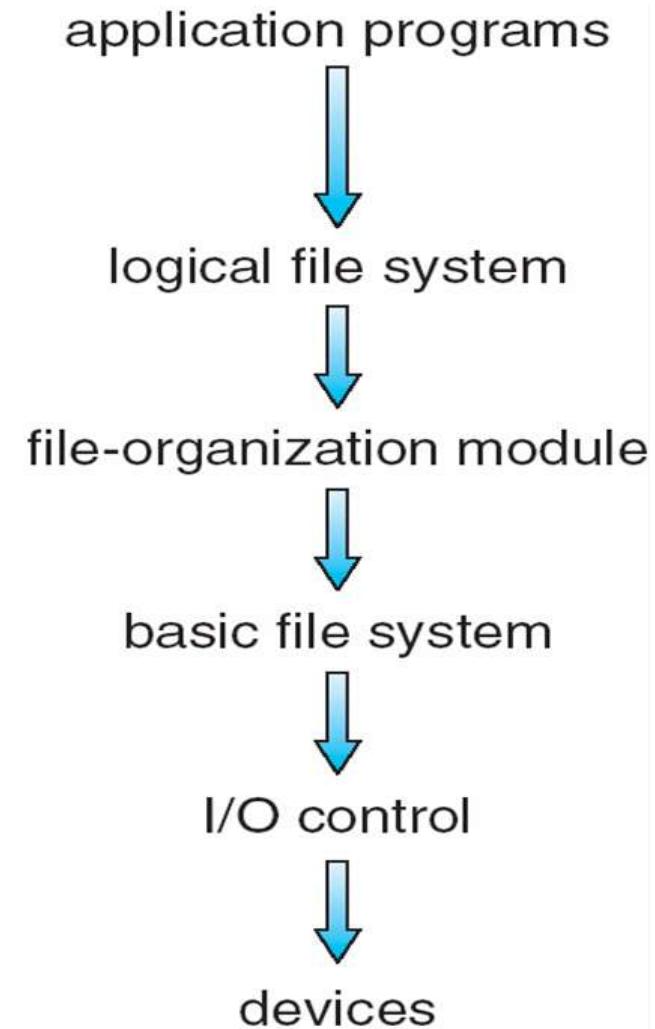
- Consistency semantics specify how multiple users are to access a shared file simultaneously
- Unix file system (UFS) implements:
 - Writes to an open file visible immediately to other users of the same open file
 - One mode of sharing file pointer to allow multiple users to read and write concurrently
- AFS has session semantics

Implementing file systems

File system on disk

- Disk I/O in terms of sectors (512 bytes)
- File system: implementation of acyclic graph using the linear sequence of sectors
- Device driver: available to rest of the OS code to access disk using a block number

File system implementation: layering



File system: Layering

- Device drivers manage I/O devices at the I/O control layer
- Given commands like “read drive1, cylinder 72, track 2, sector 10, into memory location 1060” outputs low-level hardware specific
 - File organization module understands files, logical address, and physical blocks
 - Translates logical block # to physical block #
 - Manages free space, disk allocation

Application programs

```
int main() {  
    char buf[128]; int count;  
    fd = open(...);  
    read(fd, buf, count);  
}
```

OS

Logical file system:

```
sys_read(int fd, char *buf, int count) {  
    file *fp = currproc->fdarray[fd];  
    file_read(fp, ...);  
}
```

File organization module:

```
file_read(file *fp, char *buf, int count) {  
    offset = fp->current-offset;  
    translate offset into blockno;  
    basic_read(blockno, buf, count);  
}
```

Basic File system:

```
basic_read(int blockno, char *buf, ...) {  
    os_buffer *bp;  
    sectorno = calculation on blockno;  
    disk_driver_read(sectorno, bp );  
    move-process-to-wait-queue;  
    copy-data-to-user-buffer(bp, buf);  
}
```

IO Control, Device driver:

```
disk_driver_read(sectorno) {  
    issue instructions to disk controller (ofte  
    to read sectorno    into specific location  
}
```

XV6 does it slightly differently, but following the

Layering advantages

- Layering useful for reducing complexity and redundancy, but adds overhead and can decrease performance
- Logical layers can be implemented by any coding method according to OS designer
- Many file systems, sometimes many within an operating system
- Each with its own format (CD-ROM is ISO 9660;

File system implementation: Different problems to be solved

- What to do at boot time ?**
- How to store directories and files on the partition ?**
- Complex problem. Hierarchy + storage allocation + efficiency + limits on file/directory sizes + links (hard, soft)**
- How to manage list of free sectors/blocks?**
- How to store the summary information about the complete file system**

File system implementation

- We have system calls at the API level, but how do we implement their functions?
- On-disk and in-memory structures. Let's see some of the important ones.
- Boot control block contains info needed by system to boot OS from that volume
- Not always needed. Needed if volume contains OS, usually first block of volume

A typical file control block (inode)

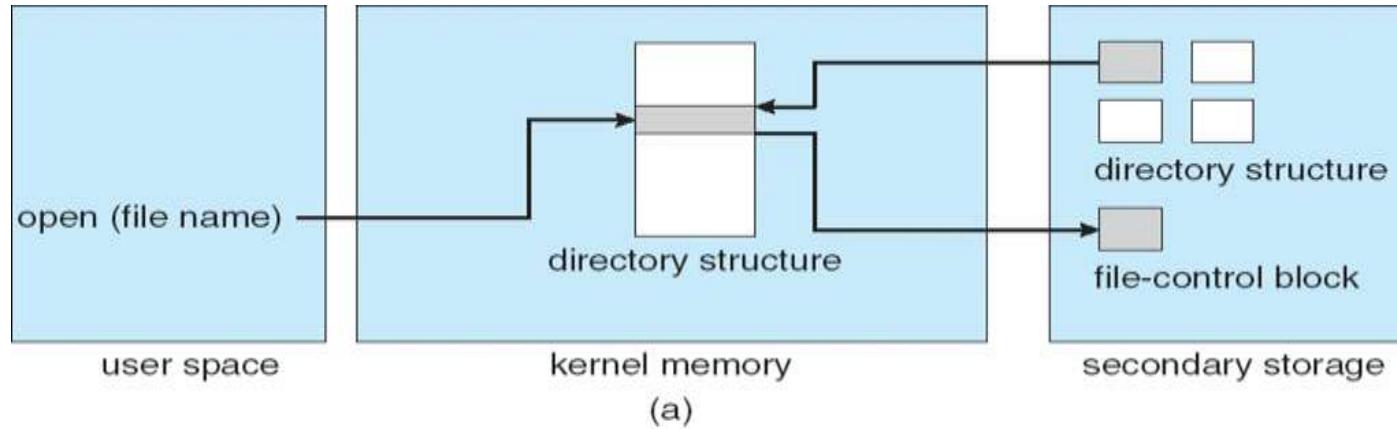
file permissions
file dates (create, access, write)
file owner, group, ACL
file size
file data blocks or pointers to file data blocks

**Why does it NOT contain the
Name of the file ?**

In memory data structures

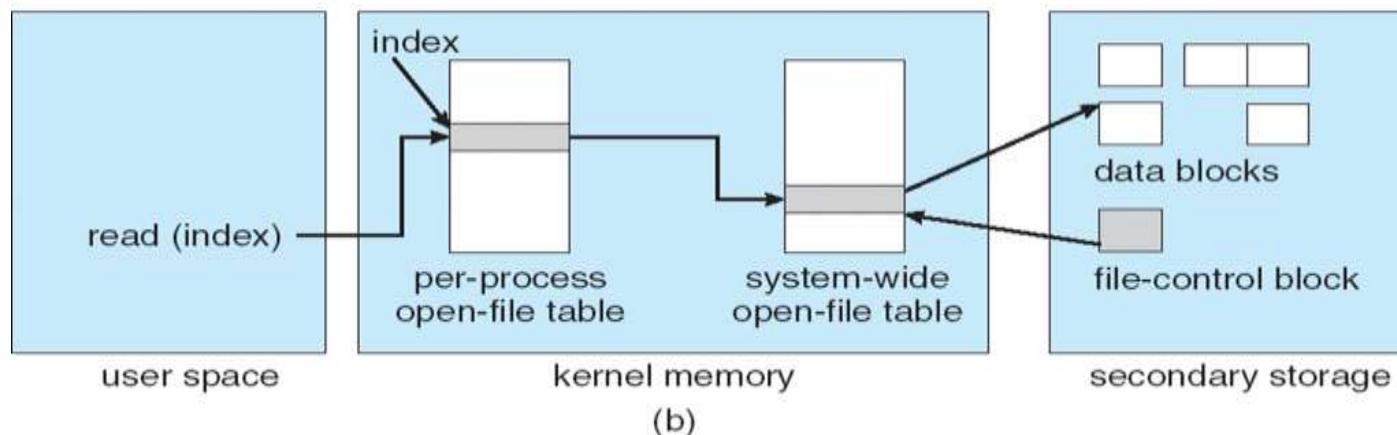
- Mount table
 - storing file system mounts, mount points, file system types
 - See next slide for “file” realated data structures
- Buffers
 - hold data blocks from secondary storage

In memory data structures: for open,read,write, ...



Open returns a file handle for

Data from read eventually comes



At boot time

- Root partition
- Contains the file system hosting OS
- “mounted” at boot time – contains “/”
- Normally can’t be unmounted!
- Check all other partitions
- Specified in */etc/fstab* on Linux
- Check if the data structure on them is consistent

Directory Implementation

- **Problem**

- **Directory contains files and/or subdirectories**
 - Operations required – create files/directories, access files/directories, search for a file (during lookup), etc.
- **Directory needs to give location of each file on disk**

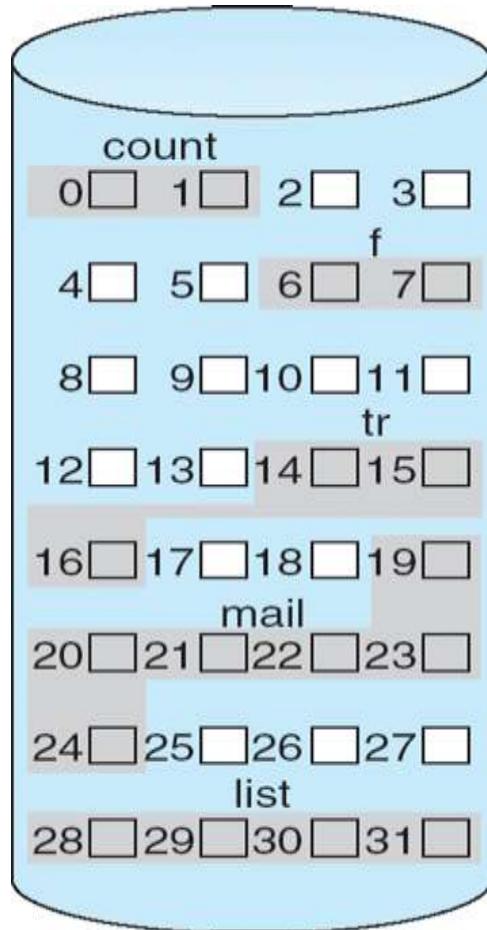
Directory Implementation

- Linear list of file names with pointer to the data blocks
- Simple to program
- Time-consuming to execute
- Linear search time
- Could keep ordered alphabetically via linked list or use B+ tree
- Ext2 improves upon this approach.

Disk space allocation for files

- File contain data and need disk blocks/sectors for storing it
- File system layer does the allocation of blocks on disk to files
- Files need to
 - Be created, expanded, deleted, shrunk, etc.
 - How to accommodate these requirements?

Contiguous Allocation of Disk Space



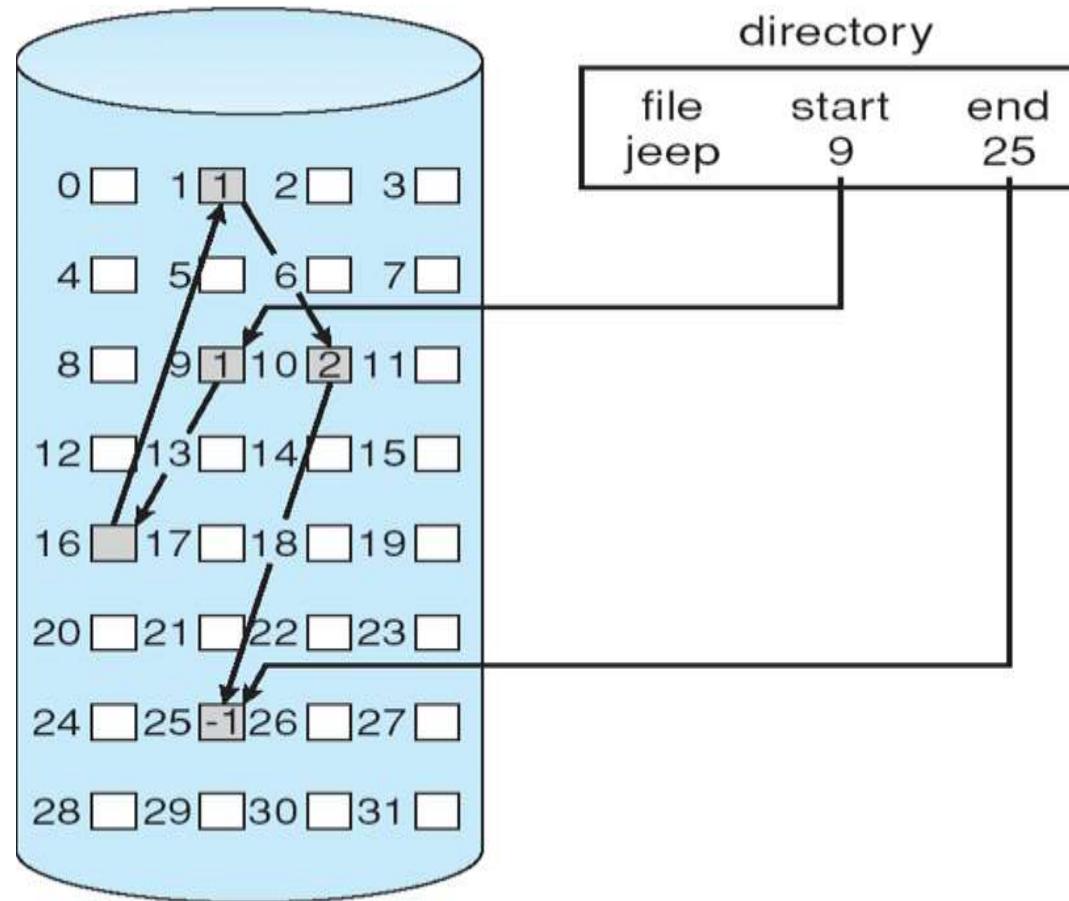
directory

file	start	length
count	0	2
tr	14	3
mail	19	6
list	28	4
f	6	2

Contiguous allocation

- Each file occupies set of contiguous blocks
- Best performance in most cases
- Simple – only starting location (block #) and length (number of blocks) are required
- Problems include finding space for file, knowing file size, external fragmentation, need for compaction off-line (downtime) or on-line

Linked allocation of blocks to a file

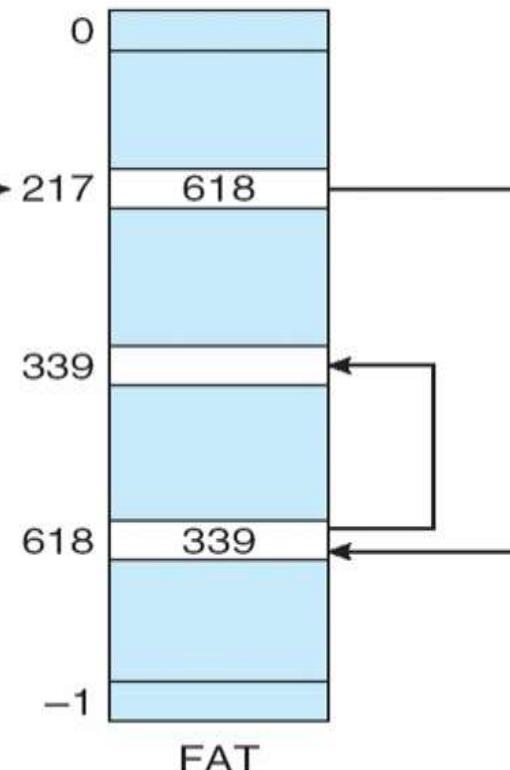
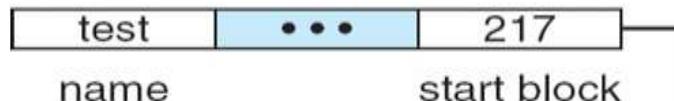


Linked allocation of blocks to a file

- Linked allocation
- Each file a linked list of blocks
- File ends at nil pointer
- No external fragmentation
- Each block contains pointer to next block (i.e.
- Free space management system called when new block needed
- Improve efficiency by clustering blocks into groups but increases internal fragmentation
- Reliability can be a problem

FAT: File Allocation Table

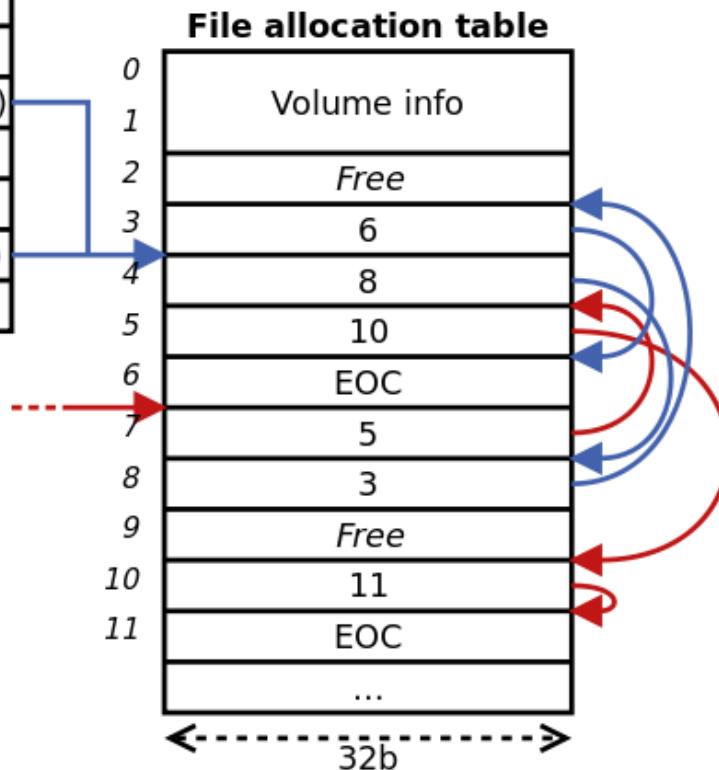
directory entry



- FAT (File Allocation Table), a variation
- Beginning of volume has table, indexed by block number
- Much like a linked list, but faster on disk and cacheable
- New block allocation simple

Directory table entry (32B)

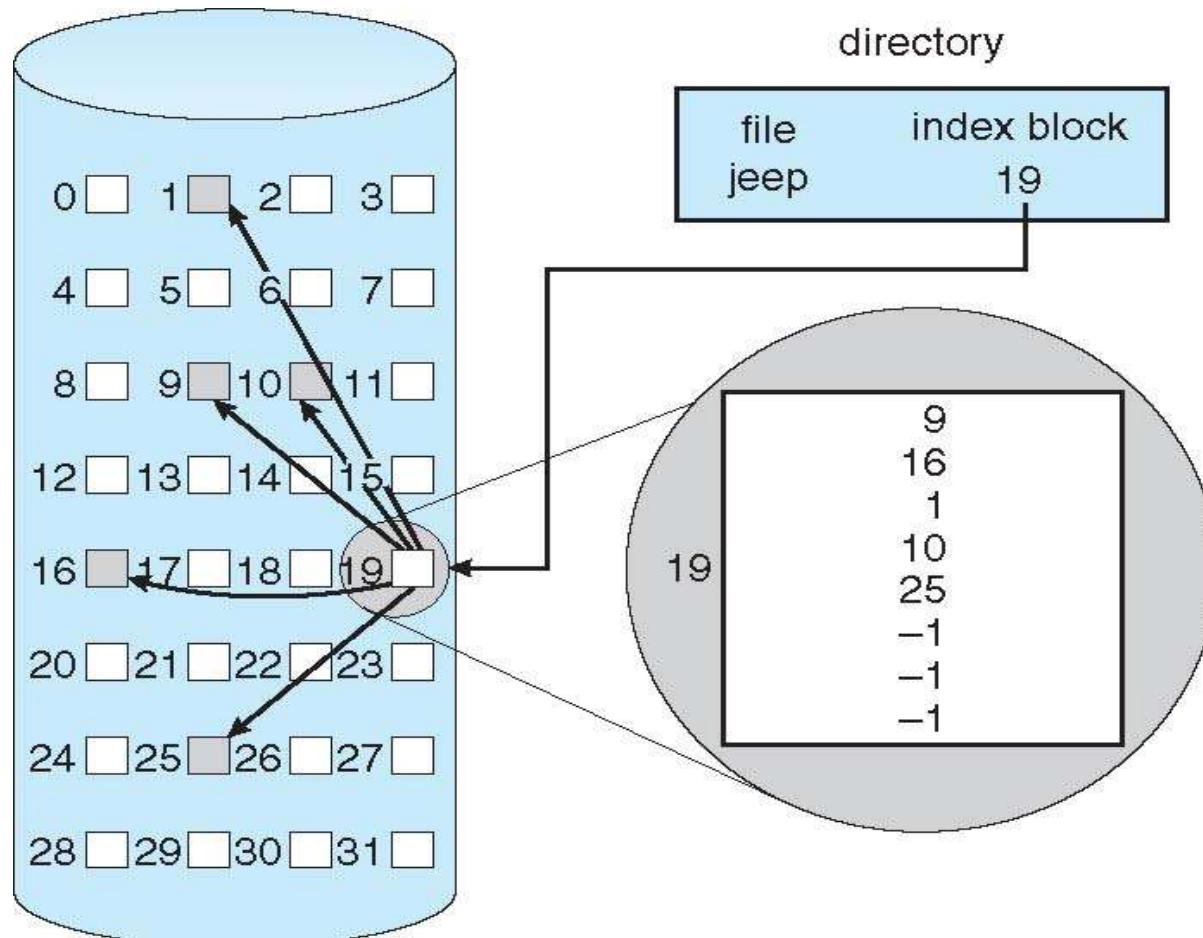
Filename (8B)
Extension (3B)
Attributes (1B)
Reserved (1B)
Create time (3B)
Create date (2B)
Last access date (2B)
First cluster # (MSB, 2B)
Last mod. time (2B)
Last mod. date (2B)
First cluster # (LSB, 2B)
File size (4B)



FAT: File Allocation Table

Variants: FAT8, FAT12, FAT16, FAT32, VFAT, ...

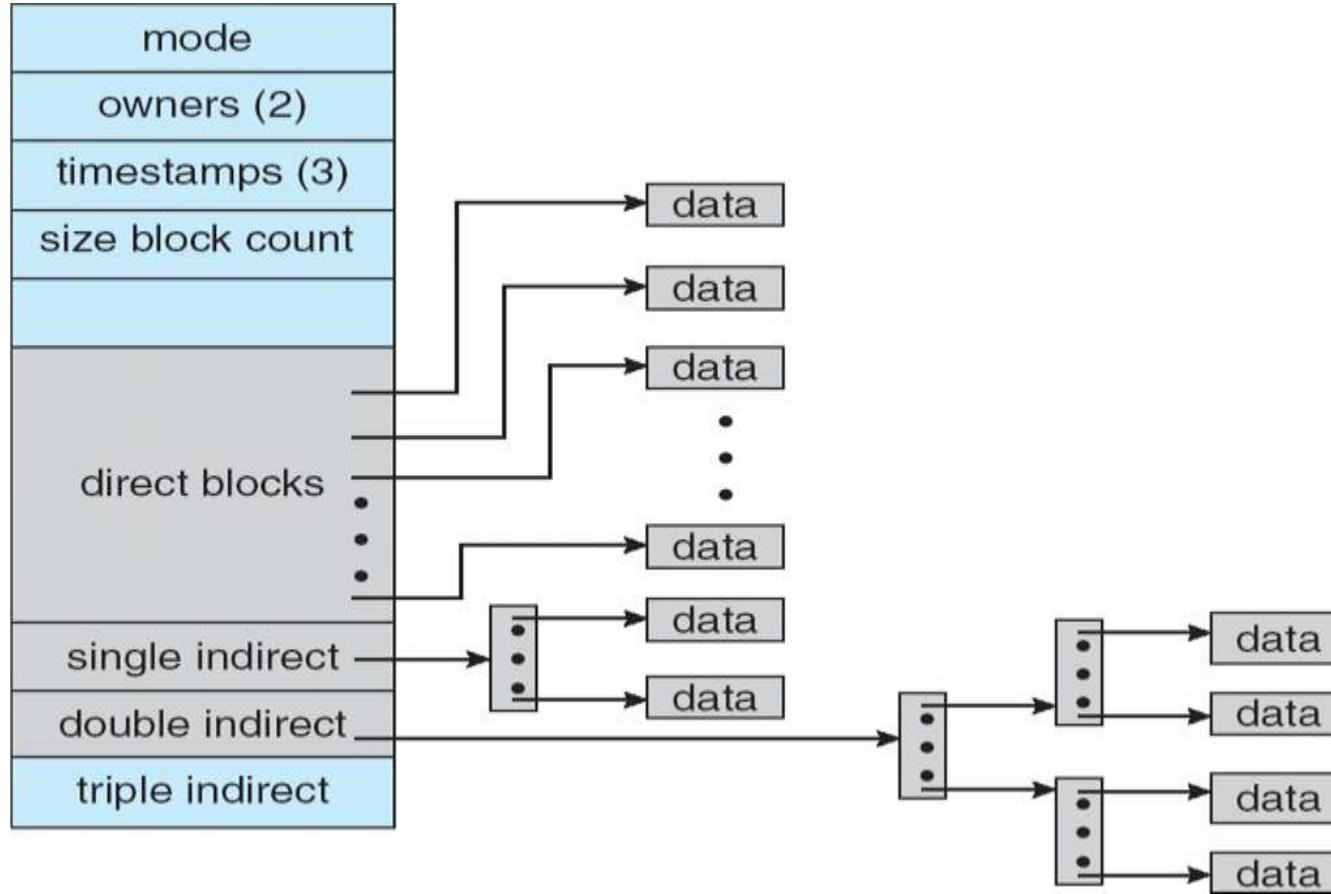
Indexed allocation



Indexed allocation

- Need index table
- Random access
- Dynamic access without external fragmentation,
but have overhead of index block
- Mapping from logical to physical in a file of
maximum size of 256K bytes and block size of 512
bytes. We need only 1 block for index table

Unix UFS: combined scheme for block allocation



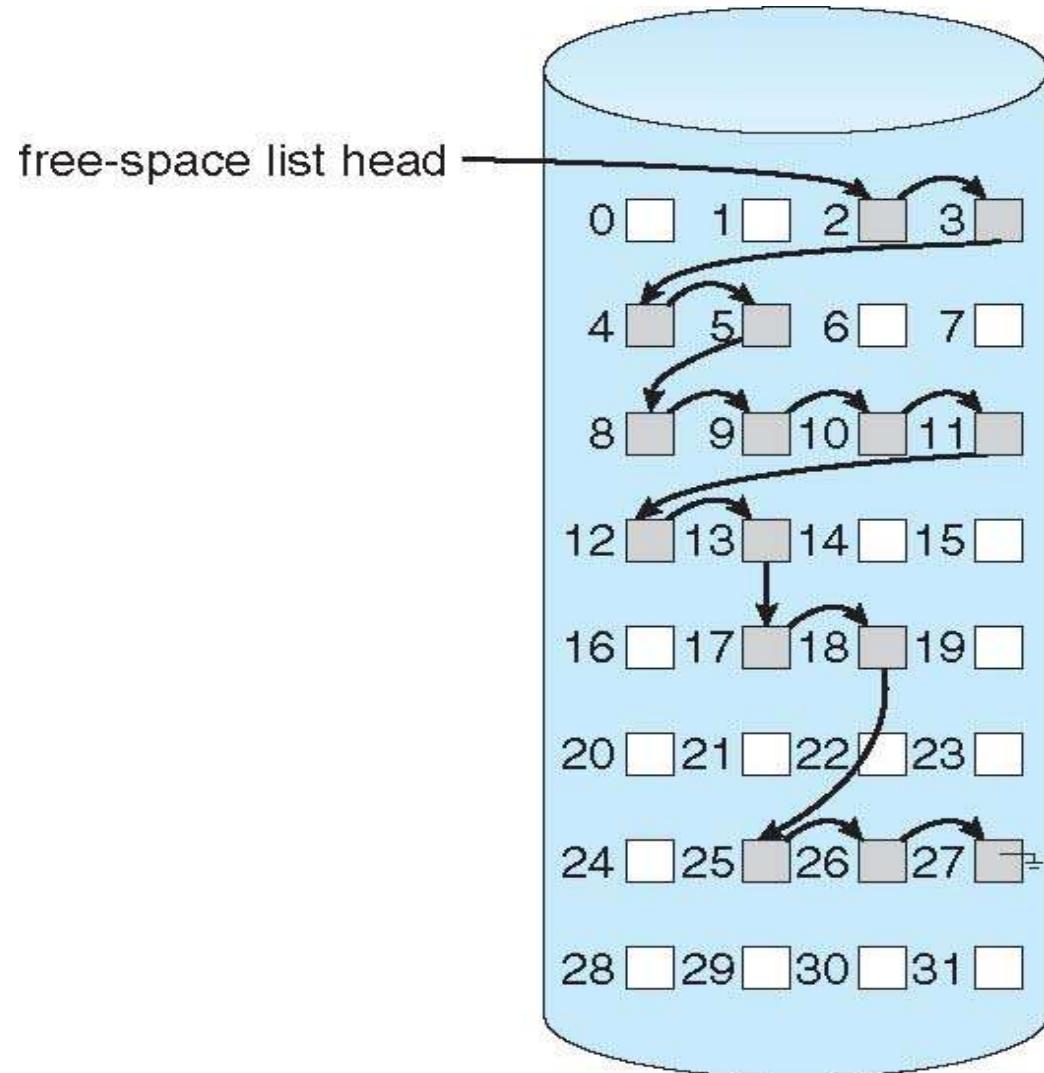
Free Space Management

- ❑ File system maintains free-space list to track available blocks/clusters
- ❑ Bit vector or bit map (n blocks)
- ❑ Or Linked list

Free Space Management: bit vector

- Each block is represented by 1 bit.
- If the block is free, the bit is 1; if the block is allocated, the bit is 0.
- For example, consider a disk where blocks 2, 3, 4, 5, 8, 9, 10, 11, 12, 13, 17, 18, 25, 26, and 27 are free and the rest of the blocks are allocated. The free-space bitmap would be 00111100111110001100000011100000 ...

Free Space Management: Linked list (not in memory, on disk!)

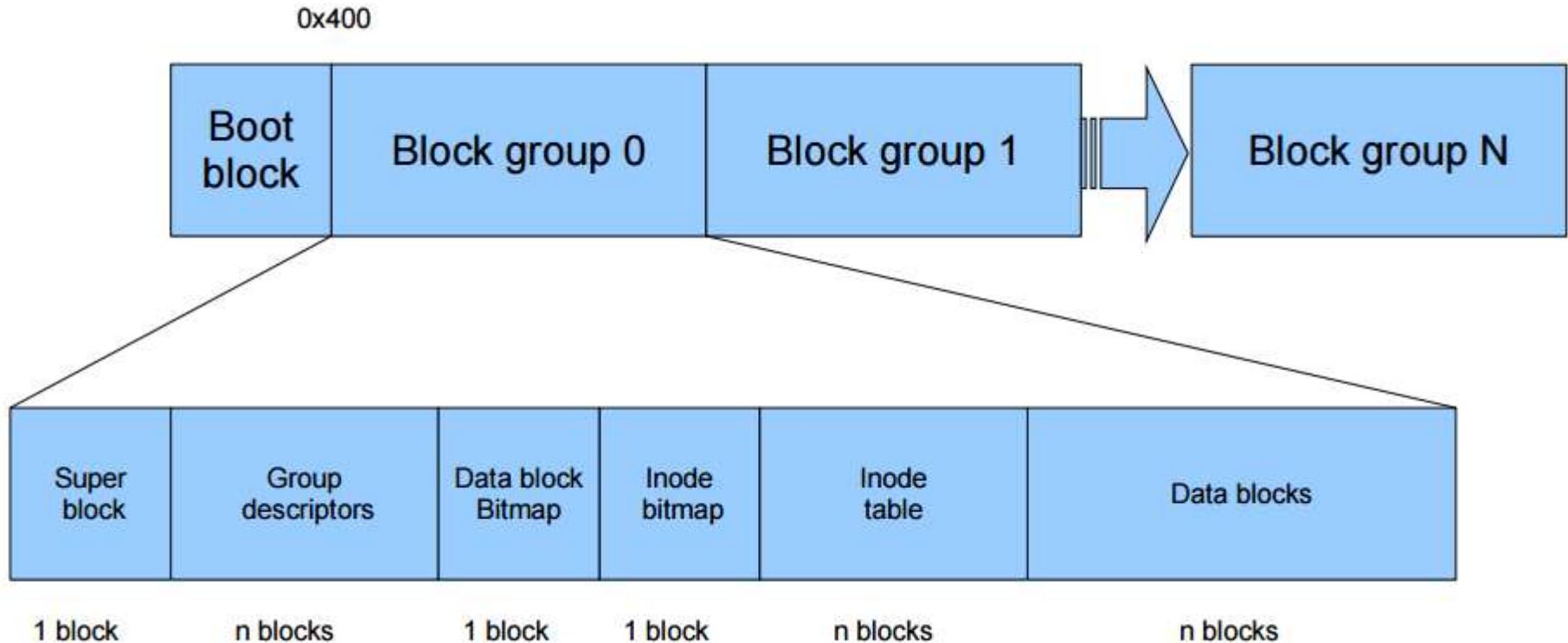


Further improvements on link list method of free-blocks

- Grouping
- Counting
- Space Maps (ZFS)
- Read as homework

Ext2 FS layout

Ext2 FS Layout



```
struct ext2_super_block {
    __le32 s_inodes_count; /* Inodes count */
    __le32 s_blocks_count; /* Blocks count */
    __le32 s_r_blocks_count; /* Reserved blocks count */
    __le32 s_free_blocks_count; /* Free blocks count */
    __le32 s_free_inodes_count; /* Free inodes count */
    __le32 s_first_data_block; /* First Data Block */
    __le32 s_log_block_size; /* Block size */
    __le32 s_log_frag_size; /* Fragment size */
    __le32 s_blocks_per_group; /* # Blocks per group */
    __le32 s_frags_per_group; /* # Fragments per group */
    __le32 s_inodes_per_group; /* # Inodes per group */
    __le32 s_mtime; /* Mount time */
    __le32 s_wtime; /* Write time */
    __le16 s_mnt_count; /* Mount count */
    __le16 s_max_mnt_count; /* Maximal mount count */
    __le16 s_magic; /* Magic signature */
    __le16 s_state; /* File system state */
    __le16 s_errors; /* Behaviour when detecting errors */
```

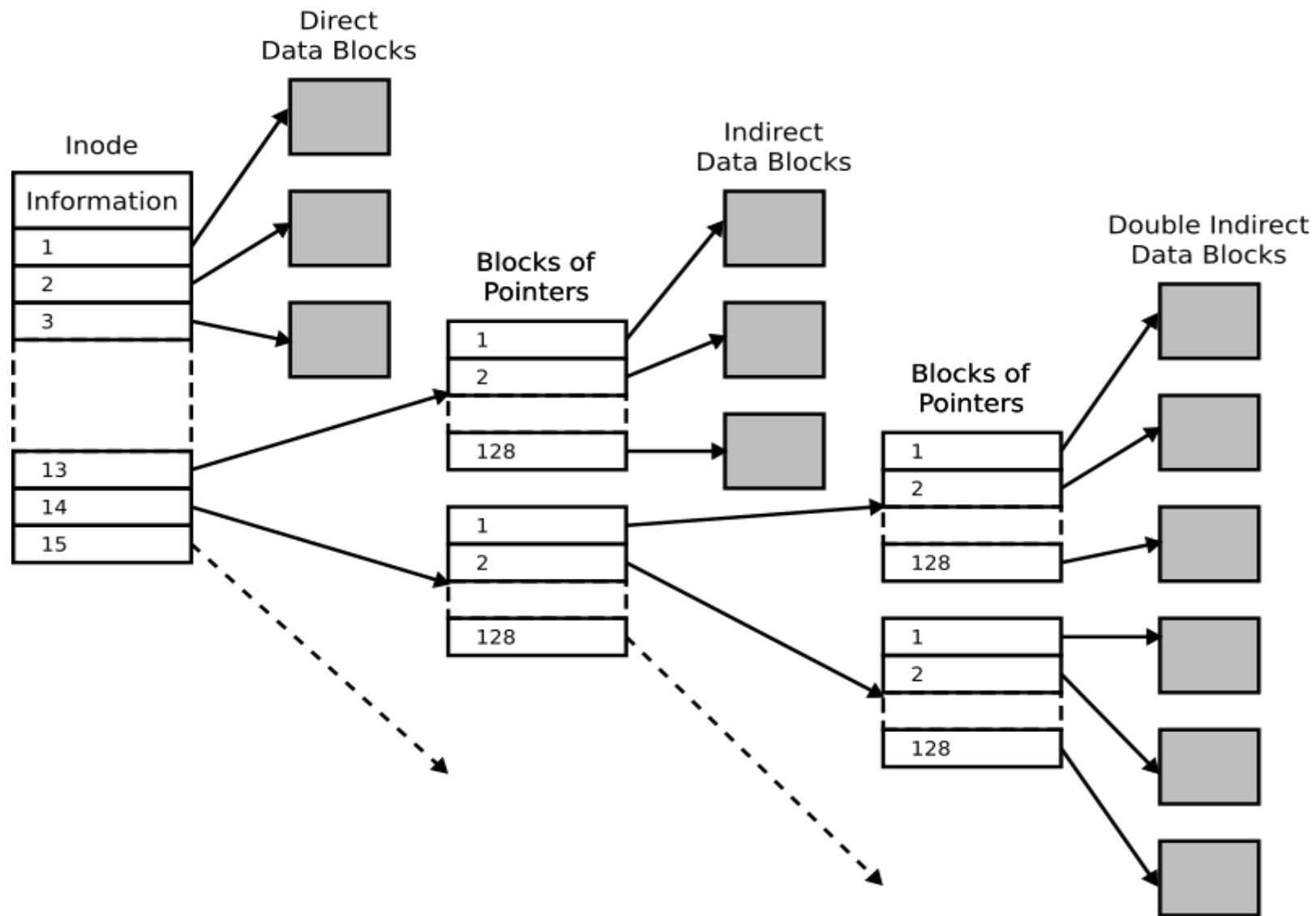
```
struct ext2_super_block {  
...  
    __le16 s_minor_rev_level; /* minor revision level */  
    __le32 s_lastcheck;      /* time of last check */  
    __le32 s_checkinterval; /* max. time between checks */  
    __le32 s_creator_os;    /* OS */  
    __le32 s_rev_level;     /* Revision level */  
    __le16 s_def_resuid;   /* Default uid for reserved blocks */  
    __le16 s_def_resgid;   /* Default gid for reserved blocks */  
    __le32 s_first_ino;    /* First non-reserved inode */  
    __le16 s_inode_size;   /* size of inode structure */  
    __le16 s_block_group_nr; /* block group # of this superblock */  
    __le32 s_feature_compat; /* compatible feature set */  
    __le32 s_feature_incompat; /* incompatible feature set */  
    __le32 s_feature_ro_compat; /* readonly-compatible feature set */  
    __u8  s_uuid[16];       /* 128-bit uuid for volume */  
    char s_volume_name[16]; /* volume name */  
    char s_last_mounted[64]; /* directory where last mounted */  
    __le32 s_algorithm_usage_bitmap; /* For compression */
```

```
struct ext2_super_block {  
...  
    __u8    s_prealloc_blocks; /* Nr of blocks to try to preallocate*/  
    __u8    s_prealloc_dir_blocks; /* Nr to preallocate for dirs */  
    __u16   s_padding1;  
/*  
 * Journaling support valid if EXT3_FEATURE_COMPAT_HAS_JOURNAL set.  
 */  
    __u8    s_journal_uuid[16]; /* uuid of journal superblock */  
    __u32   s_journal_inum;    /* inode number of journal file */  
    __u32   s_journal_dev;    /* device number of journal file */  
    __u32   s_last_orphan;    /* start of list of inodes to delete */  
    __u32   s_hash_seed[4];    /* HTREE hash seed */  
    __u8    s_def_hash_version; /* Default hash version to use */  
    __u8    s_reserved_char_pad;  
    __u16   s_reserved_word_pad;  
    __le32  s_default_mount_opts;  
    __le32  s_first_meta_bg;   /* First metablock block group */  
    __u32   s_reserved[190];    /* Padding to the end of the block */
```

```
struct ext2_group_desc
{
    __le32 bg_block_bitmap;      /* Blocks bitmap block */
    __le32 bg_inode_bitmap;     /* Inodes bitmap block */
    __le32 bg_inode_table;     /* Inodes table block */
    __le16 bg_free_blocks_count; /* Free blocks count */
    __le16 bg_free_inodes_count; /* Free inodes count */
    __le16 bg_used_dirs_count; /* Directories count */
    __le16 bg_pad;
    __le32 bg_reserved[3];
};
```

```
struct ext2_inode {  
    __le16 i_mode;    /* File mode */  
    __le16 i_uid;     /* Low 16 bits of Owner Uid */  
    __le32 i_size;    /* Size in bytes */  
    __le32 i_atime;   /* Access time */  
    __le32 i_ctime;   /* Creation time */  
    __le32 i_mtime;   /* Modification time */  
    __le32 i_dtime;   /* Deletion Time */  
    __le16 i_gid;     /* Low 16 bits of Group Id */  
    __le16 i_links_count; /* Links count */  
    __le32 i_blocks;  /* Blocks count */  
    __le32 i_flags;   /* File flags */
```

Inode in ext2



```
struct ext2_inode {  
    ...  
    union {  
        struct {  
            __le32 l_i_reserved1;  
        } linux1;  
        struct {  
            __le32 h_i_translator;  
        } hurd1;  
        struct {  
            __le32 m_i_reserved1;  
        } masix1;  
    } osd1;          /* OS dependent 1 */  
    __le32 i_block[EXT2_N_BLOCKS];/* Pointers to blocks */  
    __le32 i_generation; /* File version (for NFS) */  
    __le32 i_file_acl; /* File ACL */  
    __le32 i_dir_acl; /* Directory ACL */  
    __le32 i_faddr;   /* Fragment address */
```

```
struct ext2_inode {  
    ...  
    union {  
        struct {  
            __u8 l_i_frag; /* Fragment number */      __u8 l_i_fsize; /* Fragment size */  
            __u16 i_pad1;          __le16 l_i_uid_high; /* these 2 fields */  
            __le16 l_i_gid_high; /* were reserved2[0] */  
            __u32 l_i_reserved2;  
        } linux2;  
        struct {  
            __u8 h_i_frag; /* Fragment number */      __u8 h_i_fsize; /* Fragment size */  
            __le16 h_i_mode_high;      __le16 h_i_uid_high;  
            __le16 h_i_gid_high;  
            __le32 h_i_author;  
        } hurd2;  
        struct {  
            __u8 m_i_frag; /* Fragment number */      __u8 m_i_fsize; /* Fragment size */  
            __u16 m_pad1;          __u32 m_i_reserved2[2];  
        } masix2;  
    } osd2;          /* OS dependent 2 */
```

Ext2 FS Layout: Directory entry

	inode	rec_len	file_type	name_len	name						
0	21	12	1	2	.	\0	\0	\0			
12	22	12	2	2	.	.	\0	\0			
24	53	16	5	2	h	o	m	e	1	\0	\0
40	67	28	3	2	u	s	r	\0			
52	0	16	7	1	o	l	d	f	i	1	e
68	34	12	4	2	s	b	i	n			

Let's see a program to read superblock of an ext2 file system.

Efficiency and Performance (and the risks created while trying to achieve it!)

Efficiency

- Efficiency dependent on:
 - Disk allocation and directory algorithms
 - Types of data kept in file's directory entry
 - Pre-allocation or as-needed allocation of metadata structures
 - Fixed-size or varying-size data structures

Performance

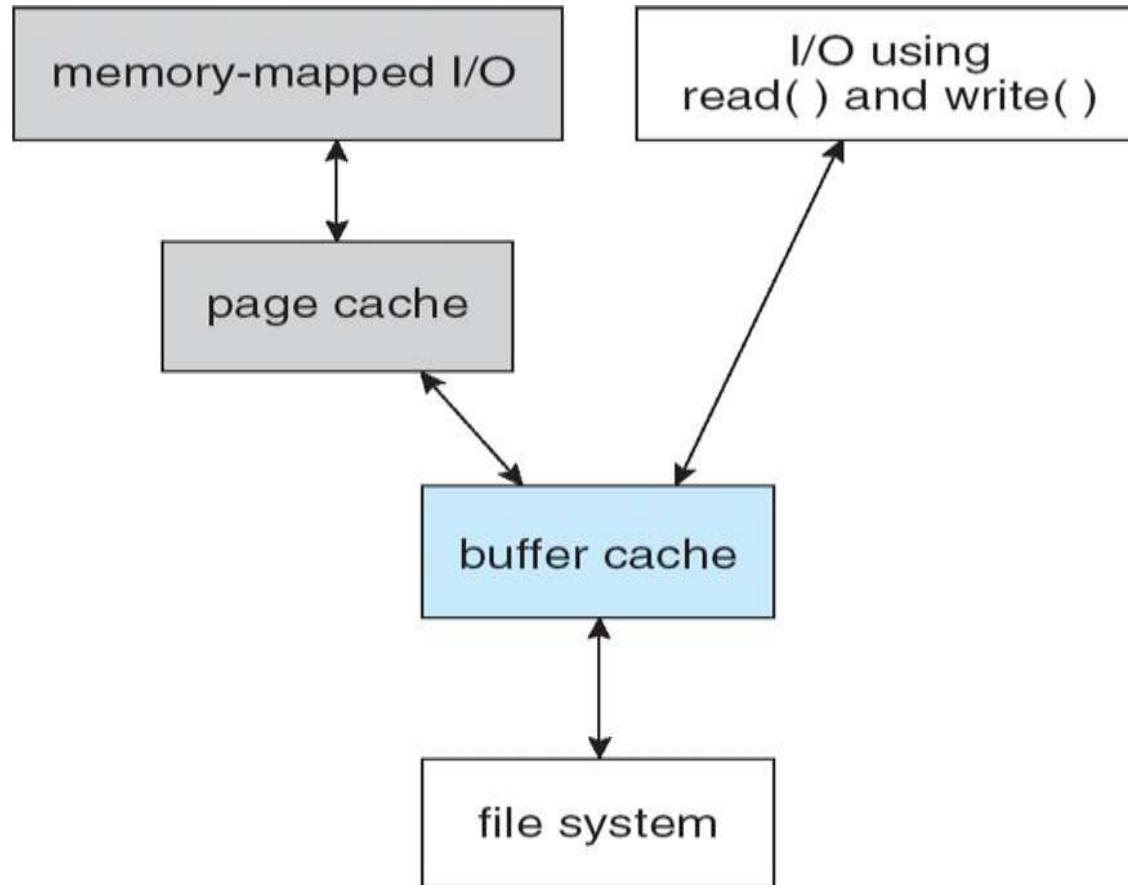
- Keeping data and metadata close together
- Buffer cache – separate section of main memory for frequently used blocks
- Synchronous writes sometimes requested by apps or needed by OS
- No buffering / caching – writes must hit disk before acknowledgement

A synchronous write is more common, bufferable

Page cache

- A page cache caches pages rather than disk blocks using virtual memory techniques and addresses
- Memory-mapped I/O uses a page cache
- Routine I/O through the file system uses the buffer (disk) cache
- This leads to the following figure

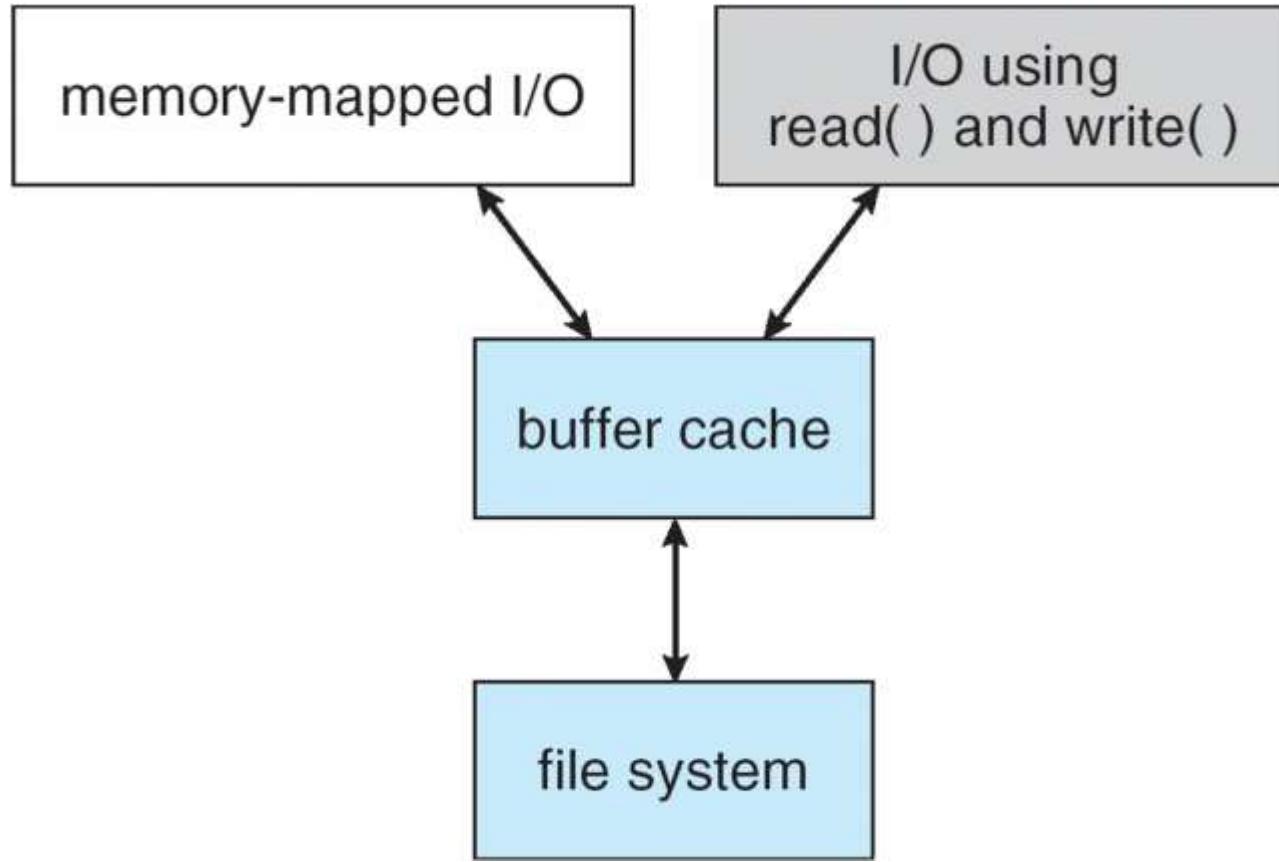
I/O Without a Unified Buffer Cache



Unified buffer cache

- A unified buffer cache uses the same page cache to cache both memory-mapped pages and ordinary file system I/O to avoid double caching
- But which caches get priority, and what replacement algorithms to use?

I/O Using a Unified Buffer Cache



Recovery

- Problem. Consider creating a file on ext2 file system.
- Following on disk data structures will/may get modified
- Directory data block, new directory data block, block bitmap, inode table, inode table bitmap, group descriptor, super block, data blocks for new file, more data block bitmaps, ...
- All cached in memory by OS

Recovery

- Consistency checking – compares data in directory structure with data blocks on disk, and tries to fix inconsistencies
- Can be slow and sometimes fails
- Use system programs to back up data from disk to another storage device (magnetic tape, other magnetic disk, optical)
- Recover lost file or disk by restoring data from

Log structured file systems

- Log structured (or journaling) file systems record each metadata update to the file system as a transaction
- All transactions are written to a log
- A transaction is considered committed once it is written to the log (sequentially)
- Sometimes to a separate device or section of disk
- However, the file system may not yet be updated

Journaling file systems

- ❑ Veritas FS
- ❑ Ext3, Ext4
- ❑ Xv6 file system!

Operating Systems

Sem VI 2023-24

Course Introduction,

Course Plan

Evaluation Scheme

Me

.Abhijit Ashok Meenakshi

.3rd Floor, ENTC EXTN

.9422308125 (Signal, WhatsApp)

Significance

- The identifier course for “Computer Engineering”
 - How is a “functional” (usable, useful) computer system built?
Answer is – essentially with OS
 - The glue that binds



Answers to questions like

- Why is my computer running slow?
- Why does a system “hang” / “crash” ?
- The exact sequence of events in hardware, when I click on “close” button
- The exact sequence of events between pressing of a key, and seeing it on the screen
- How is it possible that multiple applications are “running” “at the same time”, even if you have a

This course is a must for a career in

- Networking

- Security

- Microprocessor Design

- Devops

- Cloud

- Storage, Backup

- Databases

The issue about the name of the course

.The kernel

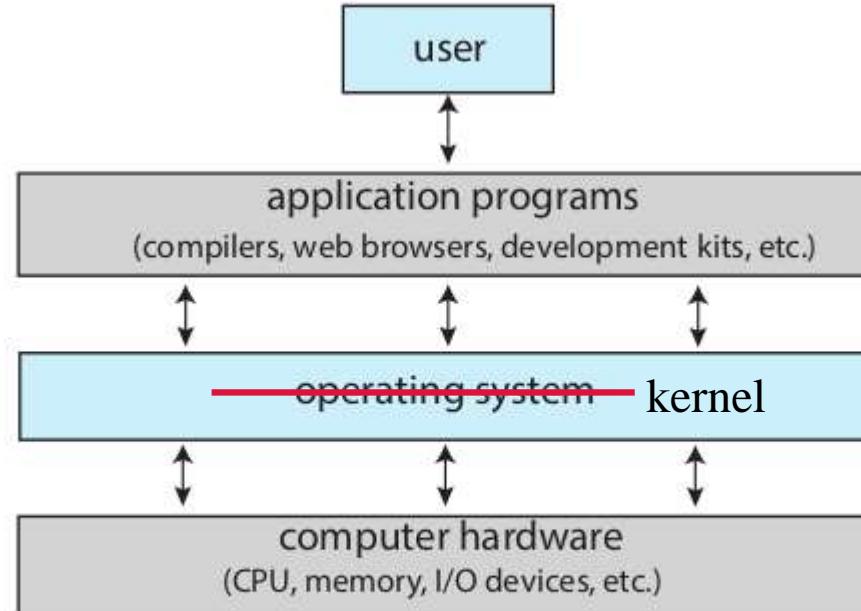


Figure 1.1 Abstract view of the components of a computer system.

OS or kernel?

- Debate on terminology
- What you use in daily life is not kernel, but
 - GUI, Shell, Applications, ...
 - One view: OS = kernel + (GUI, Shell, Libraries, System programs, Minimum Applications,...)
 - Correct, IMO!
- What we study in this course is kernel

Attendance requirement

- .75% : Institute norm

- Absence should be informed

- It will be assumed that you are on campus, except on a declared holiday

Theory course evaluation

- Total Marks: 100
- The total marks for each of the following categories will be converted to proportional maximum marks as follows:
 - Quiz-1: 15 Marks
 - Quiz-2: 15 Marks
 - Anytime quizzes: 10 Marks (may be ON campus)

Lab course evaluation

- Total Marks: 100
- The total marks for each of the following categories will be converted to proportional maximum marks as follows:
 - Mandatory Assignments: 20 Marks
 - Mid Term Exam: 30 Marks
 - Optional Assignments/Project: 50 Marks

Lab Assignments

- They are time consuming!
- You will be busy entire week (10-20 hrs per assignment)!
- Don't hesitate to submit incomplete assignments.
 - No binary (0 or full) marking!
 - Partial marks for partial work done.

Lab Pre-requisites

- GNU/Linux command line, at least 20-30 commands
- Dual boot installation of GNU/Linux
- Basic information about folder structure on GNU/Linux
- Backup your data! You are likely to loose it at least once!

The insatiable curiosity to "look inside" and "do

Textbooks

- Operating System Concepts, by Abraham Silberschatz, P B Galvin, G Gagne
 - 10th edition
- Introduction to Systems Software by Jonathan Misruda
- The C programming Language by Kernighan and Ritchie

On Plagiarism

- .The expected similarity score for each submission is 50%, unless specifically mentioned.
 - Assignments with similarity score below the announced score will be checked.
 - If the score is more than threshold, then teachers will examine the code and decide if it's plagiarism.
 - The decision of teacher will be final on deciding if any small amount of plagiarism is involved.

On Plagiarism

- Penalties
 - First instance: zero in that assignment
 - Second instance: Fail in course
- Following typical excuses will not be entertained
 - I Shared my code by mistake (Don't do mistakes!)
 - I had given access of my Moodle/email/etc account to someone and it was misused (You are

On Plagiarism

- Statistics:
 - 12 students failed OS Lab last semester.
 - Almost all of them because they plagiarised (not because they could not do assignments)
- Let's see how I check it and a list of plagiarisms detected

Course project

- Will be tracked on gitlab repo
- Your date-time-wise commits will be used to determine if you were engaged in doing it on a regular basis
 - It is assumed that you are likely to have plagiarised, if you have not worked on it regularly!
- After 2.5 months of semester, weekly updates will be taken from you about the project work

Course project

- The following are already available with your teacher for plagiarism check:
 - All the source-codes (assignments and projects) of all the students who have done this OS course in COEP !
 - All xv6 projects on the web
 - An uncanny-eye to detect different coding styles

A fair enough judgement about your abilities !

During lab hours

- You are expected
 - To have seen the recorded videos related to the current assignment, and all lectures related to it and ask doubts!
 - To show the work done/being-done on the current assignment and discuss with lab-incharge
 - Make noise discussing concepts in OS

Keep the lab instructor engaged with any

Feedback from last year

End Semester FeedBack Part-I

Sr.No.	Question Title	Ratings	Star Ratings
1	Clarity of expectations of students	3.5	★ ★ ★ ★ ☆
2	Effectiveness of teacher in terms of: (a) Communication skills (b) Use of teaching aids	3.5	★ ★ ★ ★ ☆
3	Effectiveness of teacher in terms of: (a) Technical content (b) course content	3.5	★ ★ ★ ★ ☆
4	Feedback provided on students progress	3.5	★ ★ ★ ★ ☆
5	Has the teacher covered entire syllabus as prescribed by College?	3.6	★ ★ ★ ★ ☆
6	Has the teacher covered relevant topics beyond syllabus	3.5	★ ★ ★ ★ ☆
7	Motivation and inspiration for students to learn	3.5	★ ★ ★ ★ ☆
8	Pace on which contents were covered	3.4	★ ★ ★ ★ ☆
9	Support for the development of students skill (i) Practical demonstration (ii) Hands on training	3.5	★ ★ ★ ★ ☆
10	Willingness to offer help and advice to students.	3.5	★ ★ ★ ★ ☆
	Feedback Out of 25	17.5	★ ★ ★ ★ ☆

End Semester FeedBack Part-I

Sr.No.	Question Title	Ratings	Star Ratings
1	Clarity of expectations of students	3.8	
2	Effectiveness of teacher in terms of: (a) Communication skills (b) Use of teaching aids	3.8	
3	Effectiveness of teacher in terms of: (a) Technical content (b) course content	3.8	
4	Feedback provided on students progress	3.8	
5	Has the teacher covered entire syllabus as prescribed by College?	3.8	
6	Has the teacher covered relevant topics beyond syllabus	3.8	
7	Motivation and inspiration for students to learn	3.8	
8	Pace on which contents were covered	3.8	
9	Support for the development of students skill (i) Practical demonstration (ii) Hands on training	3.8	
10	Willingness to offer help and advice to students.	3.8	
	Feedback Out of 25	18.9	

Feedback from last year

Sr.No	Feed Back Student Comment
1	Teacher really makes you think to create a deep understanding about the subject
2	Its a very detailed course. It has been taught in very deeply. The advanced concepts should be optional for people who are willing to learn it. overall its a very time consuming course and very difficult to understand
3	None
4	good
5	Best teacher in this entire college IMO
6	Keep it up
7	excellent
8	Sir please teach us some subject in the next sem. If you are going to teach an elective please inform us.
9	Appreciate all the hard work put in the course. You are the only true professor who has taught us till now.

Sr.No	Feed Back Student Comment
1	Sir clearly explains the concepts in detail and gives personal attention to the queries of students. Engages the students in different lab tasks and makes the subject interesting.
2	teaching is excellent but over expection from student .

Survey Results

Your level of interest in learning

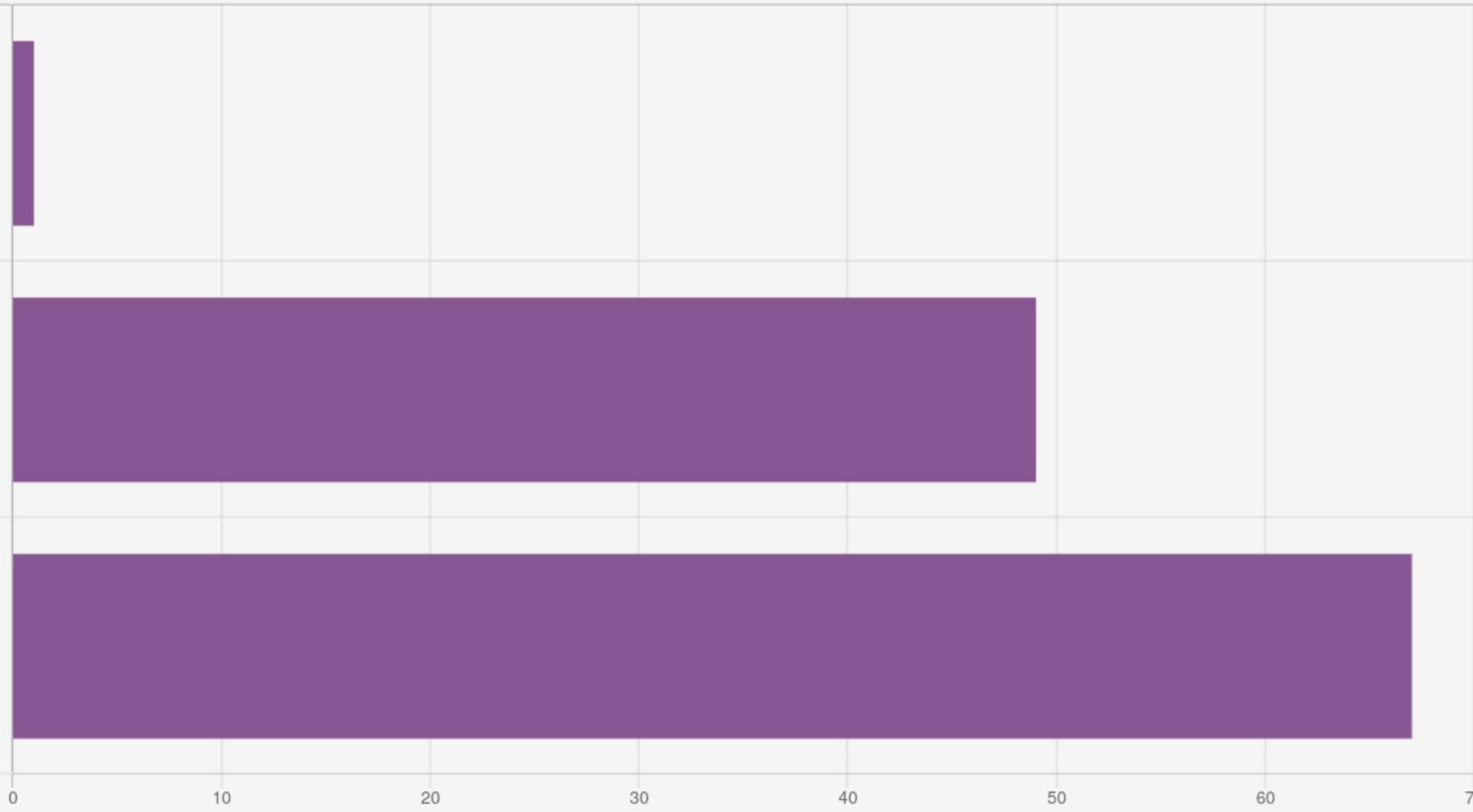
OS/Kernel

Responses

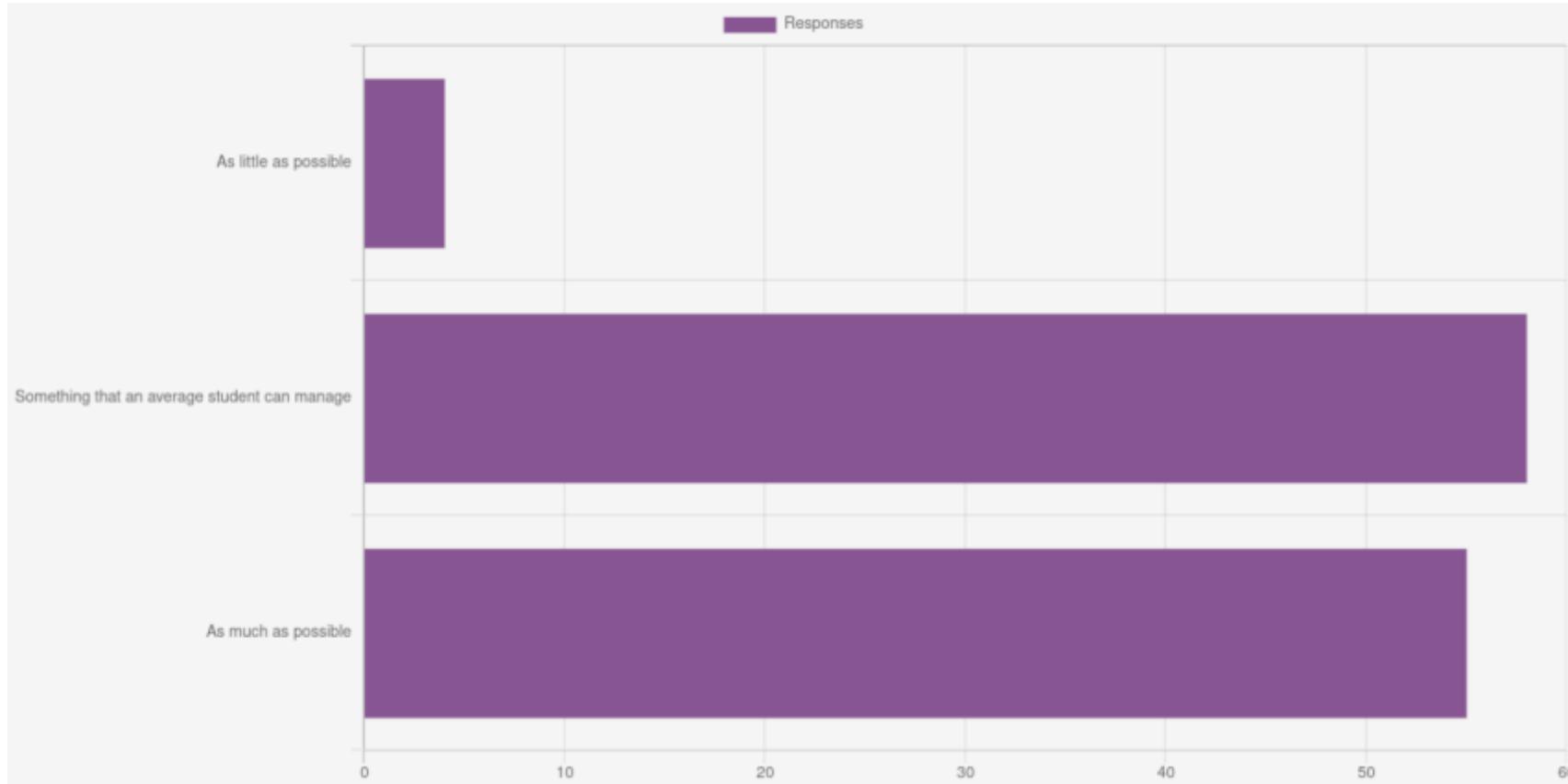
Not interested at all

Just to get basic introduction

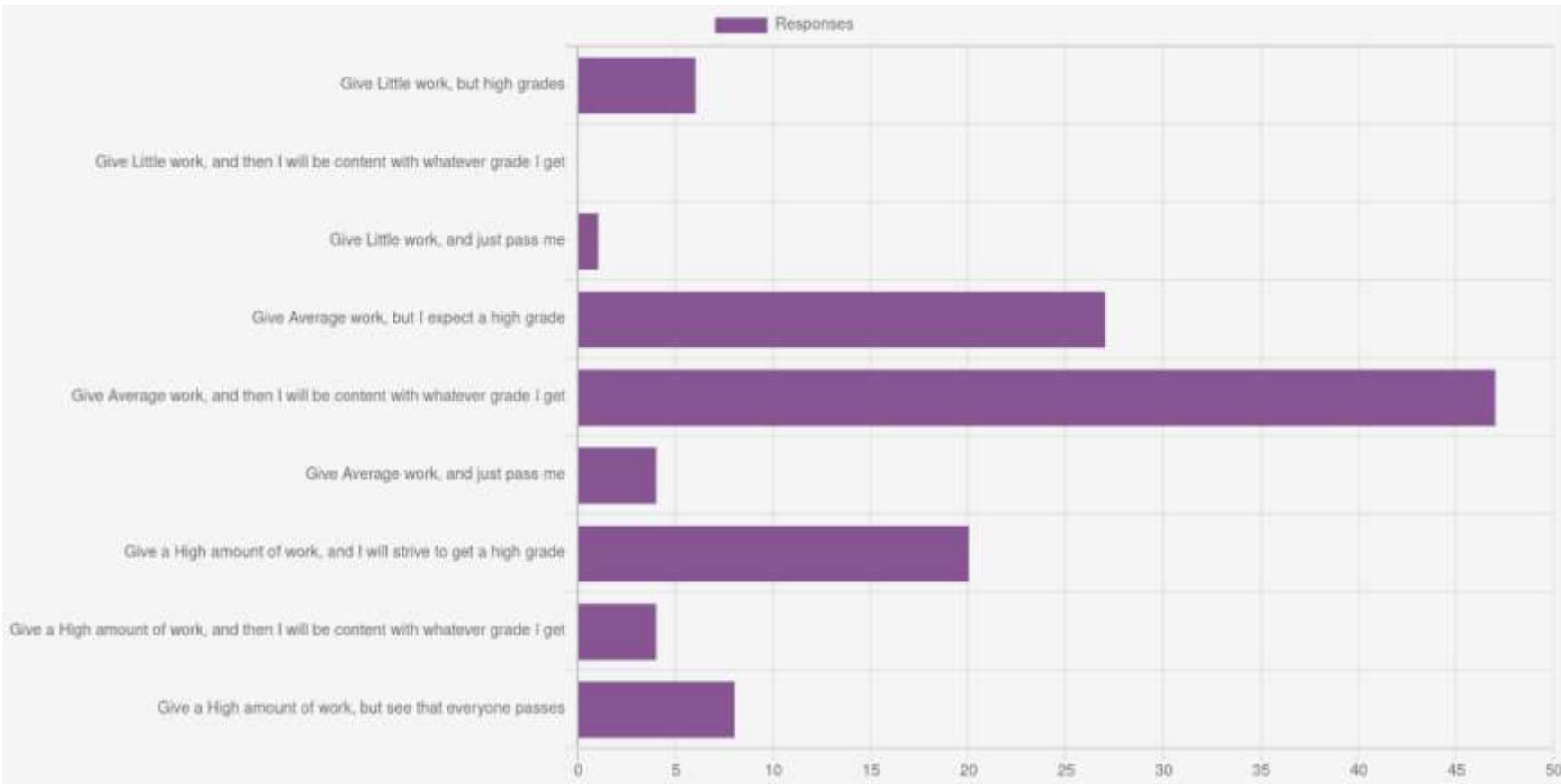
Seriously interested



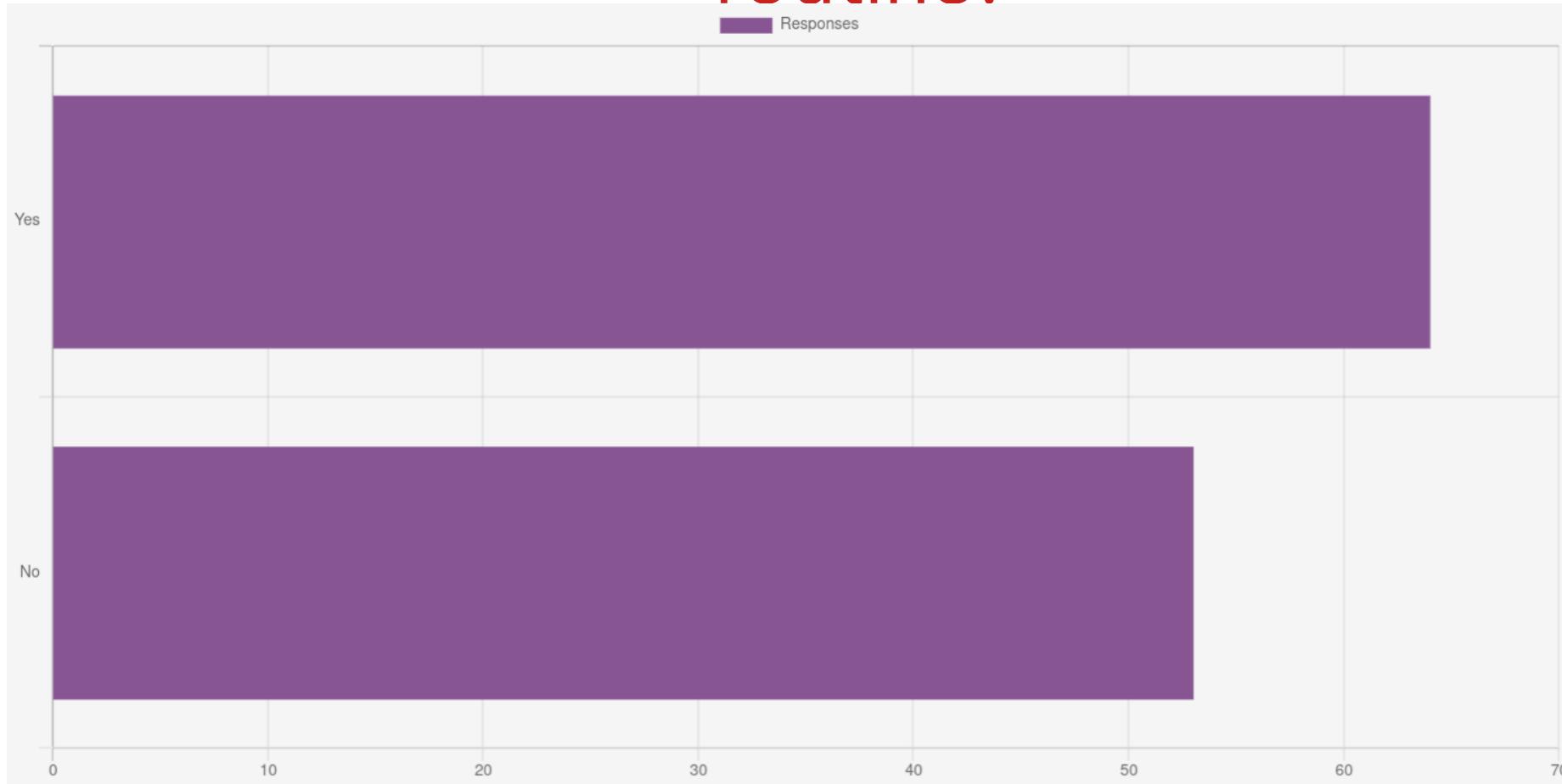
Your expectation about doing "hands-on" work



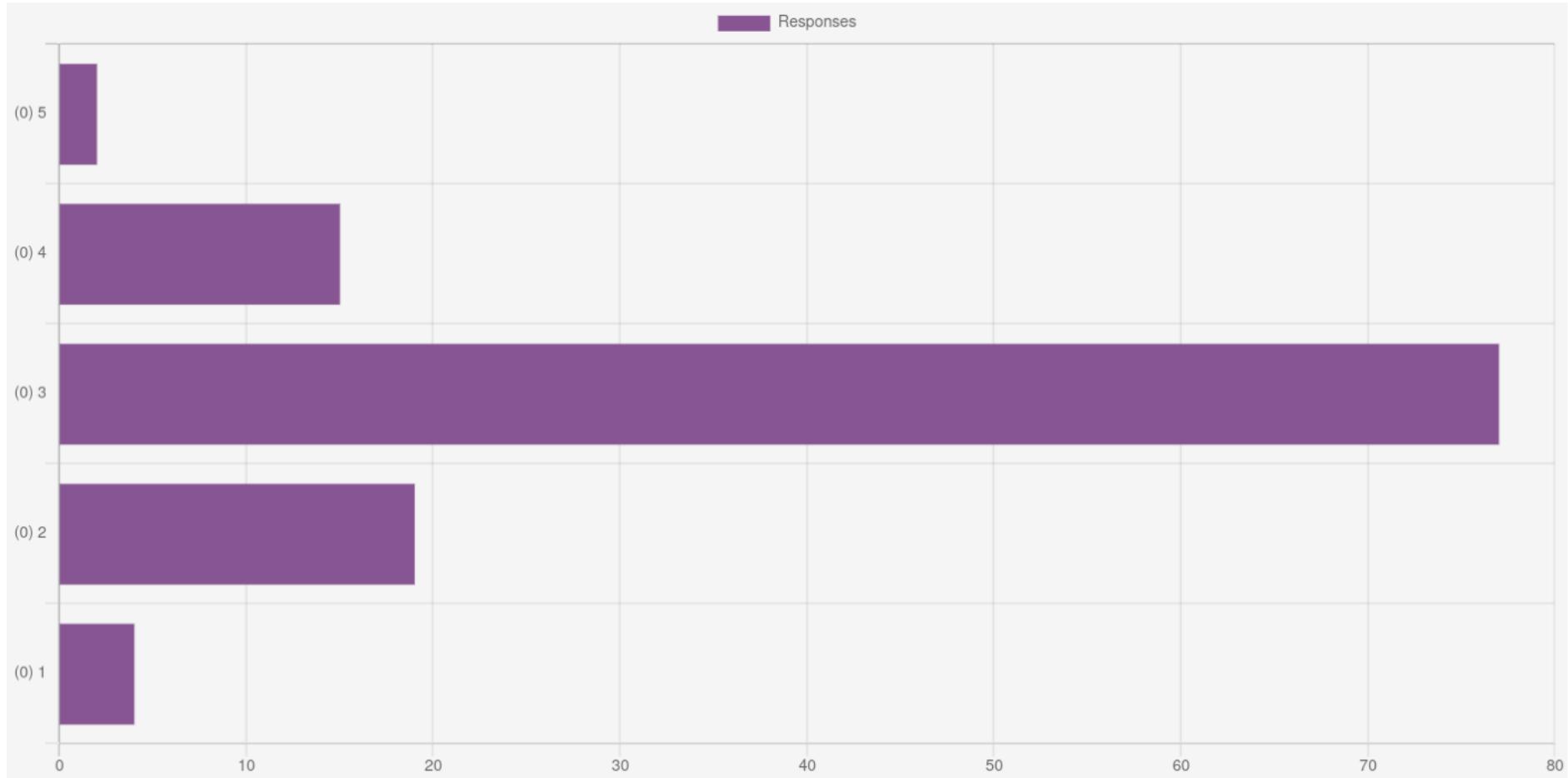
My expectations about Lab "work" (assignments, projects) and "grades":



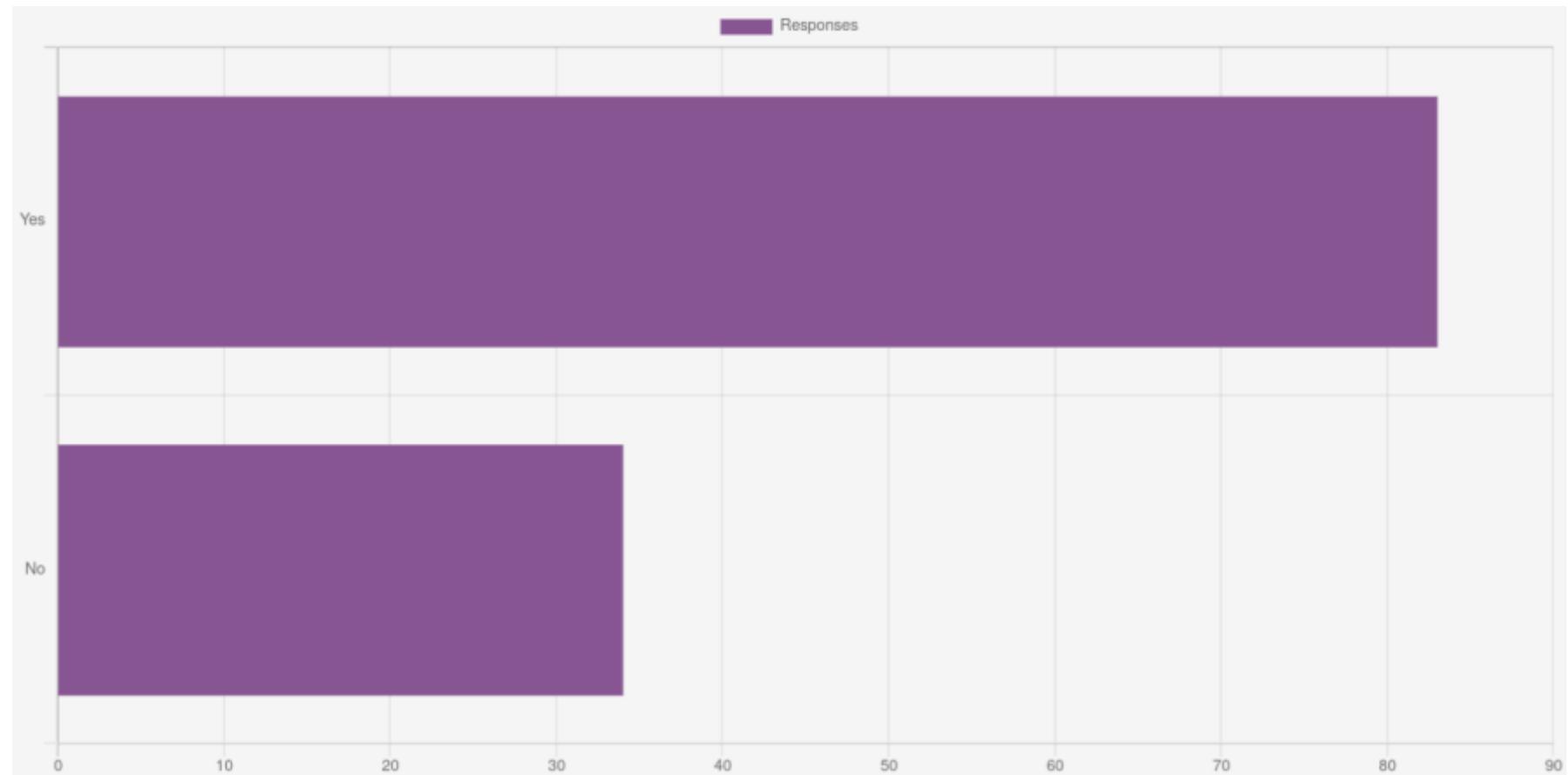
Do you use Linux command line as a routine?



Your self assessment on using Linux command line

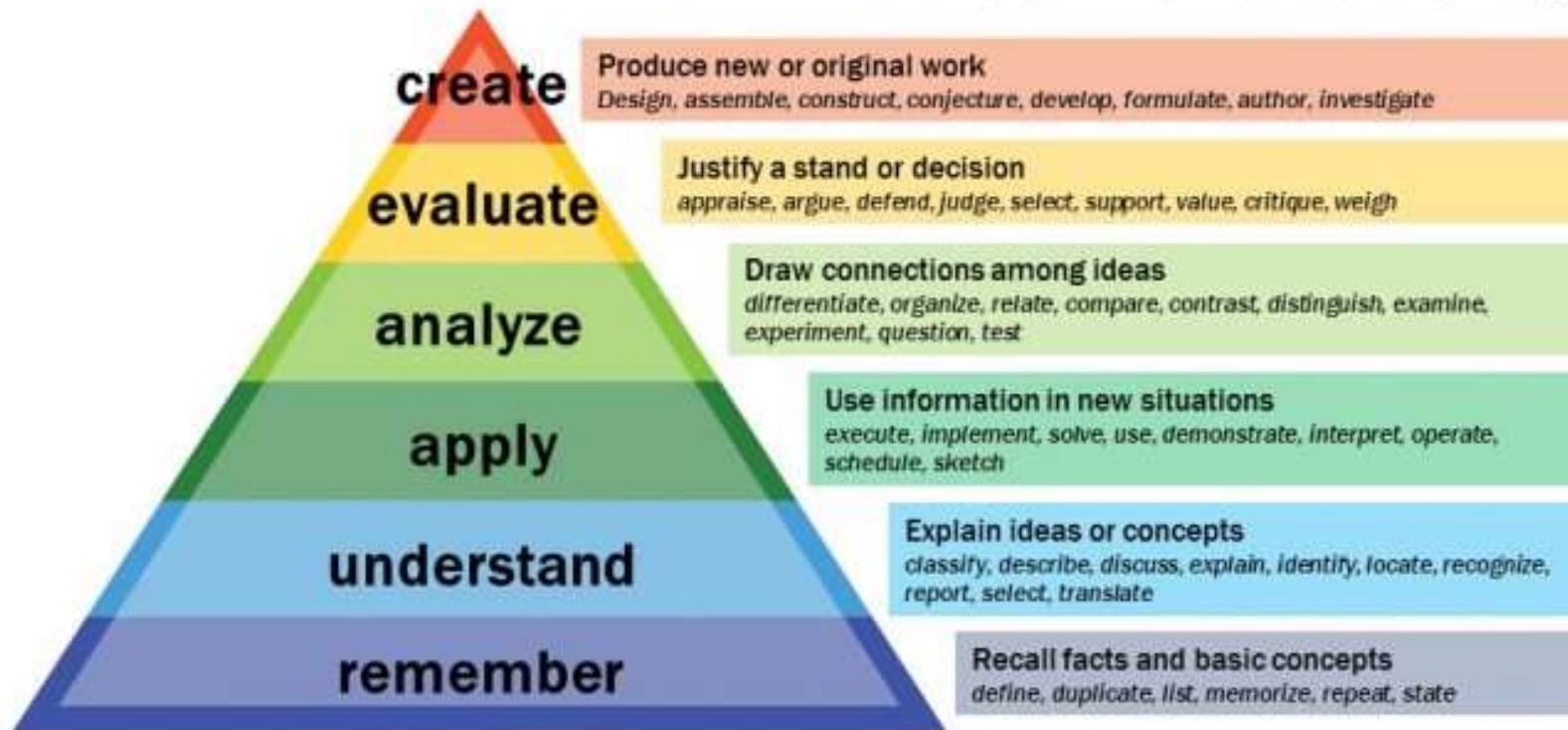


Are you willing to read a textbook?



More on learning

Bloom's Taxonomy



More on learning

- Remember
 - What is graph? What is the shortest path problem?
- Understand
 - Describe Dijkstra's algorithm with one example.
- Apply
 - Given below is a graph. Find the shortest path

More on learning

- Analyze
 - Compare Dijkstra's algorithm with Bellman Ford algorithm and mention instances when Dijkstra's algorithm is preferable
- Evaluate
 - Indian Railways has chosen to use Dijkstra's algorithm for answering queries of customers. Is it a good decision?

Notes on reading xv6 code

Abhijit A. M.

abhijit.comp@coep.ac.in

Credits:

xv6 book by Cox, Kaashoek, Morris

Notes by Prof. Sorav Bansal

Introduction to xv6

Structure of xv6 code

Compiling and executing xv6 code

About xv6

- Unix Like OS
- Multi tasking, Single user
- On x86 processor
- Supports some system calls
- Small code, 7 to 10k
- Meant for learning OS concepts
- No : demand paging, no copy-on-write fork, no shared-memory, fixed size stack for user programs

Use cscope and ctags with VIM

- Go to folder of xv6 code and run

```
cscope -q *. [chS]
```

- Also run

```
ctags *. [chS]
```

- Now download the file

http://cscope.sourceforge.net/cscope_maps.vim as
.cscope_maps.vim in your ~ folder

- And add line "source

~/.cscope_maps.vim" in your ~/.vimrc file

- Read this tutorial

Use call graphs (using doxygen)

- Doxygen – a documentation generator.
- Can also be used to generate “call graphs” of functions
- Download xv6
- Install doxygen on your Ubuntu machine.
- cd to xv6 folder
- Run “doxygen -g doxyconfig”
- This creates the file “doxyconfig”

Use call graphs (using doxygen)

- Create a folder “doxygen”
- Open “doxyconfig” file and make these changes.

PROJECT_NAME

= "XV6"

OUTPUT_DIRECTORY

= ./doxygen

CREATE_SUBDIRS

= YES

EXTRACT_ALL

= YES

EXCLUDE

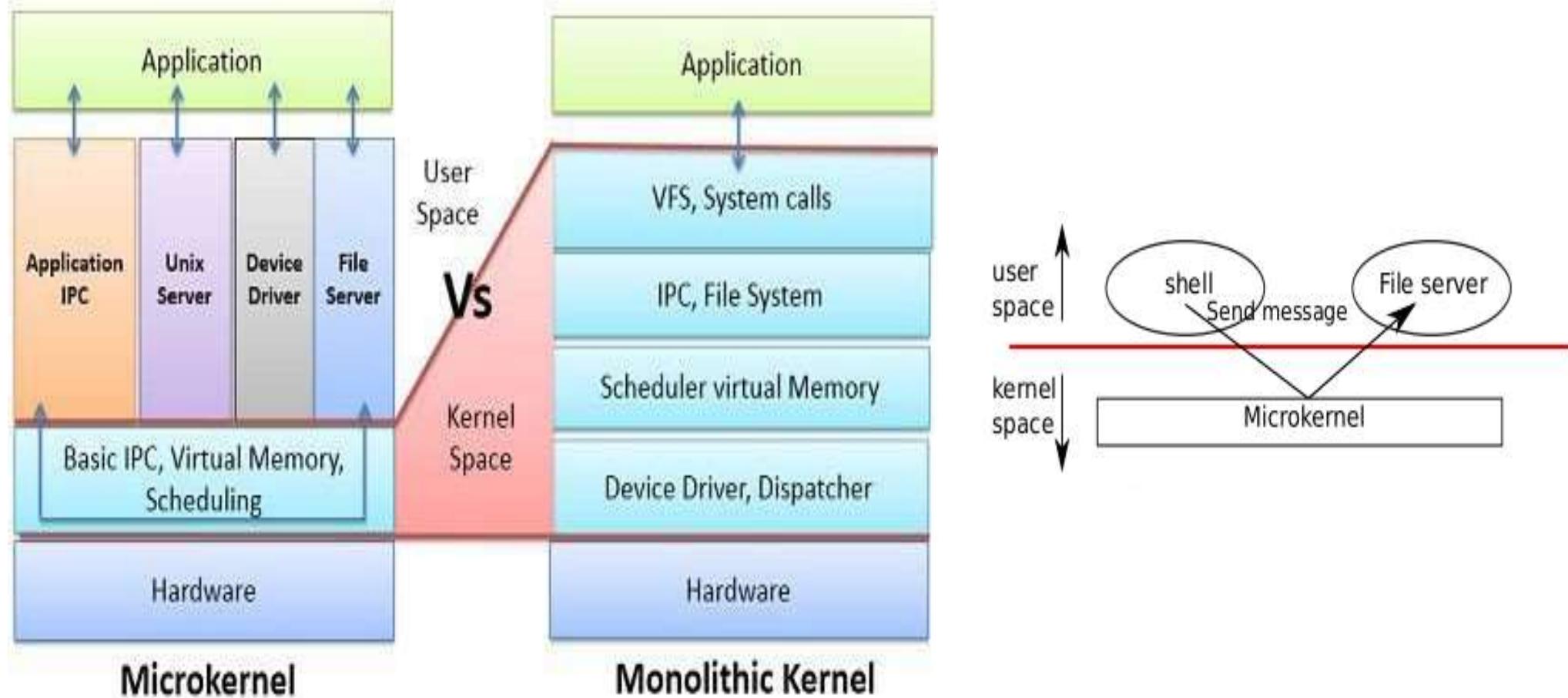
= usertests.c

cat.c yes.c echo.c forktest.c grep.c

init.c kill.c ln.c ls.c mkdir.c rm.c

sh.c stressfs.c wc.c zombie.c

Xv6 follows monolithic kernel approach



qemu

- A virtual machine manager, like Virtualbox
- Qemu provides us
 - BIOS
 - Virtual CPU, RAM, Disk controller, Keyboard controller
 - IOAPIC, LAPIC
- Qemu runs xv6 using this command

```
qemu -serial mon:stdio -drive  
file=fs.img,index=1,media=disk,format  
=raw -drive  
file=xv6.img,index=0,media=disk,forma
```

qemu

□ Understanding qemu command

□ -serial mon:stdio

□ the window of xv6 is also multiplexed in your normal terminal.

□ Run “make qemu”, then Press “Ctrl-a” and “c” in terminal and you get qemu prompt

□ -drive file=fs.img,index=1,media=disk,format=raw

□ Specify the hard disk in “fs.img”, accessible at first slot in IDE(or SATA, etc), as a “disk” , with “raw” format

□ -smp 2

About files in XV6 code

- ❑ **cat.c echo.c forktest.c grep.c
init.c kill.c ln.c ls.c mkdir.c rm.c
sh.c stressfs.c usertests.c wc.c
yes.c zombie.c**

- ❑ User programs for testing xv6

- ❑ **Makefile**

- ❑ To compile the code

- ❑ **dot-bochsrc**

- ❑ For running with emulator bochs

About files in XV6 code

- ❑ `bootasm.S` `entryother.S` `entry.S`
`initcode.S` `swtch.S` `trapasm.S`
`usys.S`

- ❑ Kernel code written in Assembly. Total 373 lines

- ❑ `kernel.ld`

- ❑ Instructions to Linker, for linking the kernel properly

- ❑ `README` `Notes` `LICENSE`

- ❑ Misc files

Using Makefile

- **make qemu**
 - Compile code and run using “qemu” emulator
- **make xv6.pdf**
 - Generate a PDF of xv6 code
- **make mkfs**
 - Create the mkfs program
- **make clean**
 - Remove all intermediary and final build files

Files generated by Makefile

- **.o files**

- Compiled from each .c file

- No need of separate instruction in Makefile to create .o files

- **_%: %.o \$(ULIB)** line is sufficient to build each .o for a _xyz file

Files generated by Makefile

❑asm files

❑Each of them has an equivalent object code file or C file. For example

```
❑bootblock: bootasm.S bootmain.c  
          $(CC) $(CFLAGS) -fno-pic -O  
-nostdinc -I. -c 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
```

Files generated by Makefile

- **_ln, _ls, etc**
- **Executable user programs**
- **Compilation process is explained after few slides**

Files generated by Makefile

- ❑ **xv6.img**

- ❑ Image of xv6 created

- ❑ **xv6.img: bootblock kernel**

- ❑ **dd if=/dev/zero of=xv6.img count=10000**

- ❑ **dd if=bootblock of=xv6.img conv=notrunc**

- ❑ **dd if=kernel of=xv6.img seek=1 conv=notrunc**

Files generated by Makefile

bootblock

```
bootblock: bootasm.S bootmain.c  
          $(CC) $(CFLAGS) -fno-pic -O -  
nostdinc -I. -c 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  
          $(OBJDUMP) -S bootblock.o >  
bootblock.asm
```

Files generated by Makefile

kernel

```
kernel: $(OBJS) entry.o entryother initcode kernel.ld  
        $(LD) $(LDFLAGS) -T kernel.ld  
-o kernel entry.o $(OBJS) -b binary  
initcode entryother  
        $(OBJDUMP) -S kernel >  
kernel.asm  
        $(OBJDUMP) -t kernel | sed  
'1,/SYMBOL TABLE/d; s/ .* //';  
/^$$/d' > kernel.sym
```

Files generated by Makefile

- **fs.img**

- A disk image containing user programs and README

- **fs.img: mkfs README \$ (UPROGS)**

- **./mkfs fs.img README**
 - \$ (UPROGS)**

- **.sym files**

- Symbol tables of different programs

- E.g. for file “kernel”

- **\$ (OBJDUMP) -t kernel | sed**

Size of xv6 C code

- `wc *[ch] | sort -n`
- **10595 34249 278455 total**
- Out of which
- **738 4271 33514 dot-bochssrc**
- **wc cat.c echo.c forktest.c grep.c init.c kill.c ln.c ls.c mkdir.c rm.c sh.c stressfs.c usertests.c wc.c yes.c zombie.c**
- **2849 6864 51993 total**
- So total code is $10595 - 2849 - 738 = 7008$ lines

List of commands to try (in given order)

usertests # Runs lot of tests and takes upto 10 minutes to run

stressfs # opens , reads and writes to files in parallel

ls # out put is filetype, inode number, type

cat README

ls;ls

cat README | grep BUILD

echo hi there

echo hi there | grep hi

List of commands to try (in this order)

echo README | grep Wa

echo README | grep Wa | grep ty # does not work

cat README | grep Wa | grep bl # works

ls > out # takes time!

mkdir test

cd test

ls .. # works from inside test

cd # fails

cd / # works

wc README

rm out

ls . test # listing both directories

ln cat xyz; ls

User Libraries: Used to link user land programs

- Ulib.c
- Strcpy, strcmp, strlen, memset, strchr, stat, atoi, memmove
- Stat uses open()
- Usys.S -> compiles into usys.o
- Assembly code file. Basically converts all calls like open() (e.g. used in ulib.c) into assembly code using “int” instruction.
- Run following command see the last 4 lines in the output

```
objdump -d usys.o
```

- 00000048 <open>:

□	48:	b8 0f 00 00 00	mov	\$0xf,%eax
□	4d:	cd 40	int	\$0x40
□	4f:	c3	ret	

User Libraries: Used to link user land programs

- printf.c
- Code for printf()!
- Interesting to read this code.
- Uses variable number of arguments. Normal technique in C is to use va_args library, but here it uses pointer arithmetic.
- Written using two more functions: printint() and putc() - both call write()
- Where is code for write()?

User Libraries: Used to link user land programs

❑ **umalloc.c**

- ❑ This is an implementation of malloc() and free()
- ❑ Almost same as the one done in “The C Programming Language” by Kernighan and Ritchie
- ❑ Uses sbrk() to get more memory from xv6 kernel

Understanding the build process in more details

□ Run

```
make qemu | tee make-output.txt
```

- You will get all compilation commands in make-output.txt

Compiling user land programs

Normally when you compile a program on Linux

You compile it for the same ‘target’ machine (= CPU + OS)

The compiler itself runs on the same OS

To compile a user land program for xv6, we don’t have a compiler on xv6,

So we compile the programs (using make, cc) on Linux , for xv6

Obviously they can’t link with the standard libraries on Linux

Compiling user land programs

```
ULIB = ulib.o usys.o printf.o umalloc.o

_%: %.o $(ULIB)

    $(LD) $(LDFLAGS) -N -e main -Ttext 0 -o $@ $^
    $(OBJDUMP) -S $@ > $*.asm
    $(OBJDUMP) -t $@ | sed '1,/SYMBOL TABLE/d; s/ .* /'
    /; /^$$/d' > $*.sym
```

\$@ is the name of the file being generated

\$^ is dependencies . i.e. \$(ULIB) and %.o in this case

Compiling user land programs

```
gcc -fno-pic -static -fno-builtin -fno-strict-
aliasing -O2 -Wall -MD -ggdb -m32 -Werror -fno-omit-
frame-pointer -fno-stack-protector -fno-pie -no-pie
-c -o cat.o cat.c
```

```
ld -m      elf_i386 -N -e main -Ttext 0 -o _cat cat.o
ulib.o usys.o printf.o umalloc.o
```

```
objdump -S _cat > cat.asm
```

```
objdump -t _cat | sed '1,/SYMBOL TABLE/d; s/ .* / /;
/^$/d' > cat.sym
```

Compiling user land programs

Mkfs is compiled like a Linux program !

```
gcc -Werror -Wall -o mkfs mkfs.c
```

How to read kernel code ?

- Understand the data structures
- Know each global variable, typedefs, lists, arrays, etc.
- Know the purpose of each of them
 - While reading a code path, e.g. exec()
 - Try to ‘locate’ the key line of code that does major work
 - Initially (but not forever) ignore the ‘error checking’ code
- Keep summarising what you have read

Pre-requisites for reading the code

- Understanding of core concepts of operating systems
- Memory Management, processes, fork-exec, file systems, synchronization, x86 architecture, calling convention , computer organization
- 2 approaches:
 - 1) Read OS basics first, and then start reading xv6 code
 - Good approach, but takes more time !
 - 2) Read some basics, read xv6, repeat

Gives a headstart, but you will always have gaps in

Memory Management Basics

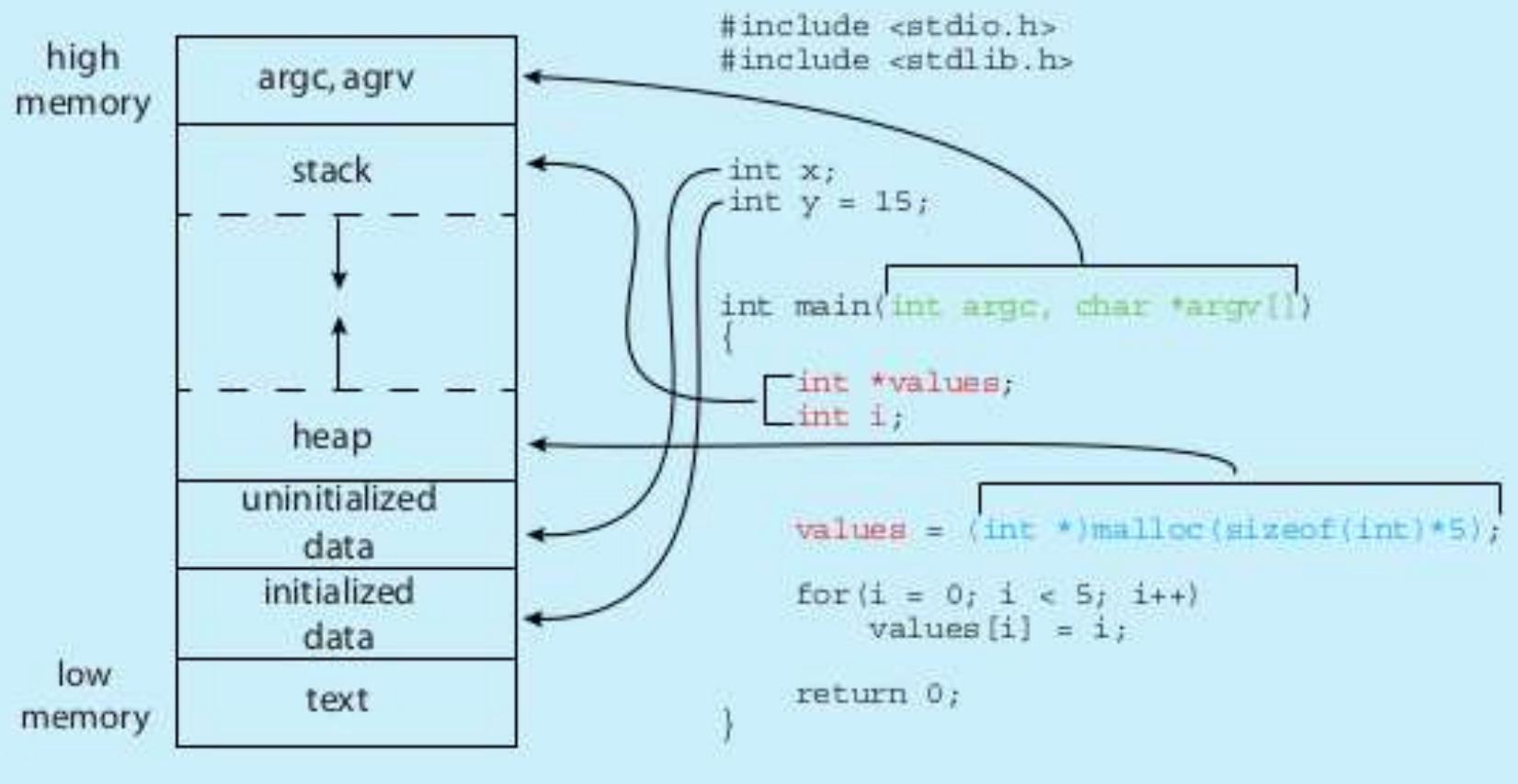
Summary

- 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



This “layout” shows
(a) which parts of a C pr
(b) A typical conceptual

\$ size /bin/ls

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

Terminology

- Text

- The machine code of the program : machine code for functions

- Data

- Initiazlied global variables

- Do not get confused with the generic word “data” in English

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

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()

The sequence

- Do not forget this
 - Machine code is typically generated by compiler
 - This machine code is put in RAM by the Loader (part of kernel) , that is exec() , when it's requested to run that program
 - The CPU's PIPELINE will issue addresses on address bus as seen in the executable file

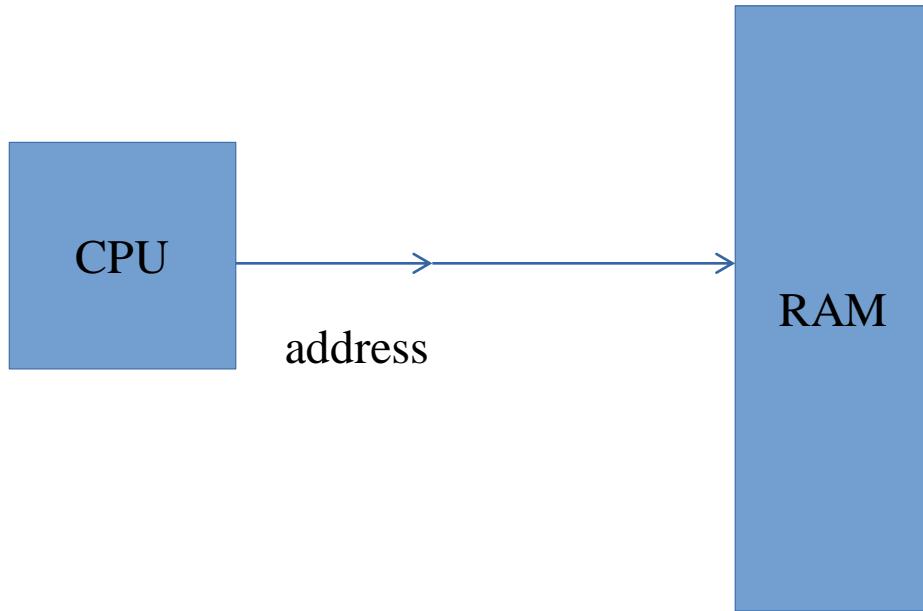
So question arises

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

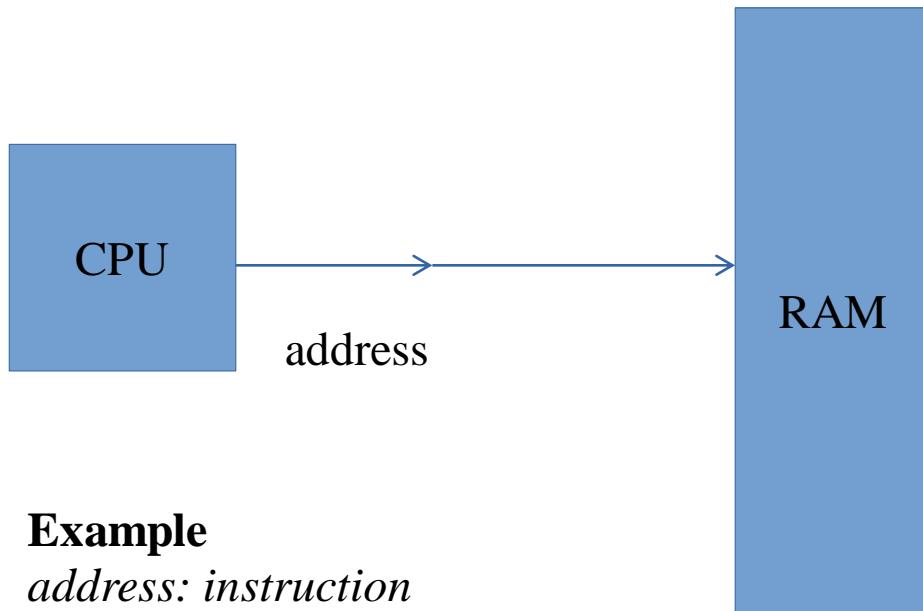
Which binding is actually used, i

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
```

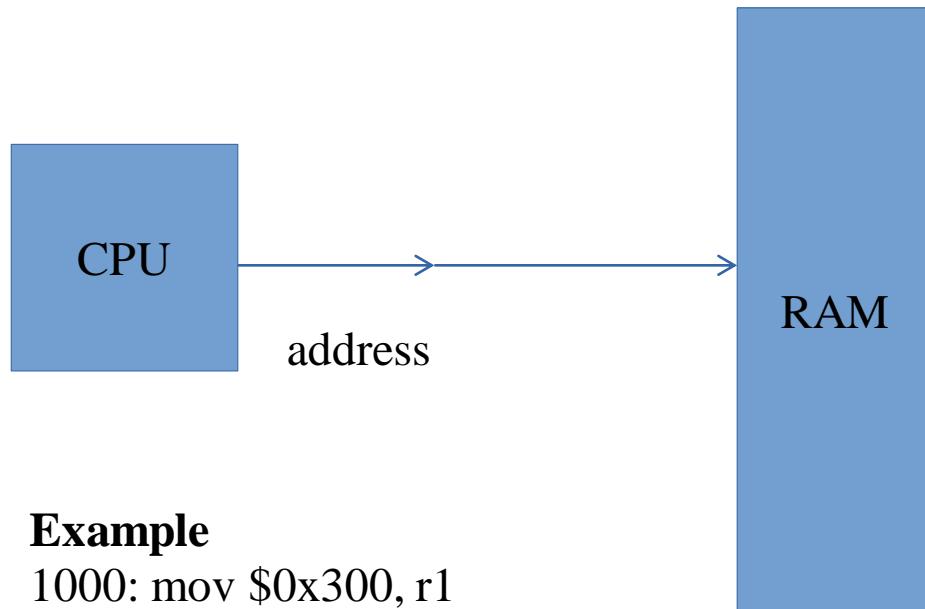
Sequence of addressed sent by CPU: 1000, 0x300, 1004, 1008, 1000, 0x300

- 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

Simplest case

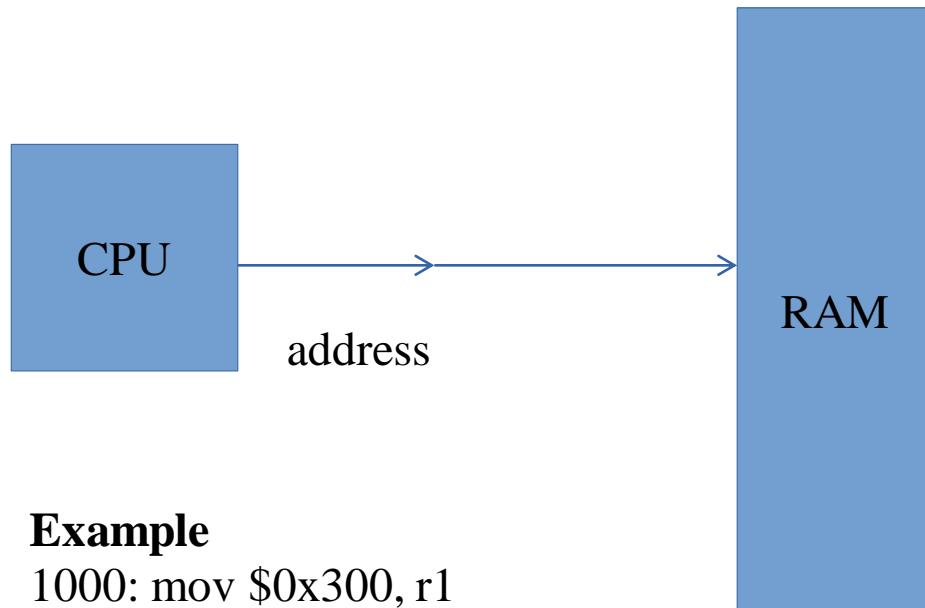


Example

```
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

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

Base/Relocation + Limit scheme

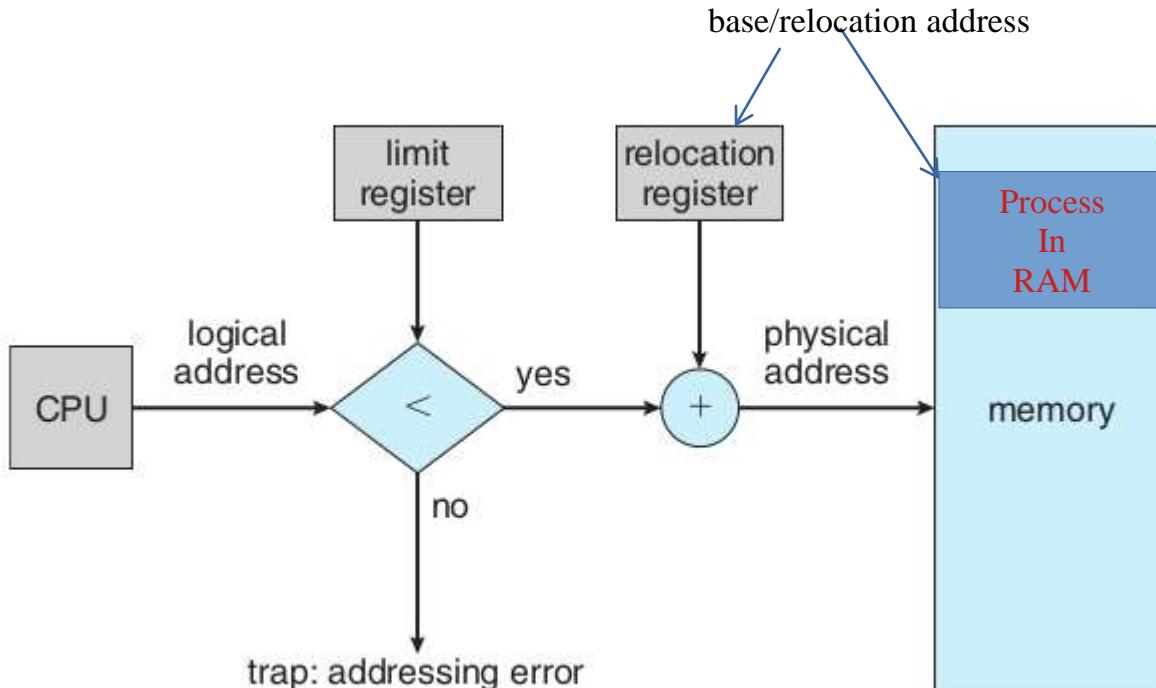


Figure 9.6 Hardware support for relocation and limit registers.

- Base and Limit are two registers inside CPU's Memory Management Unit

- 'base' is added to the address generated by

Memory Management Unit (MMU)

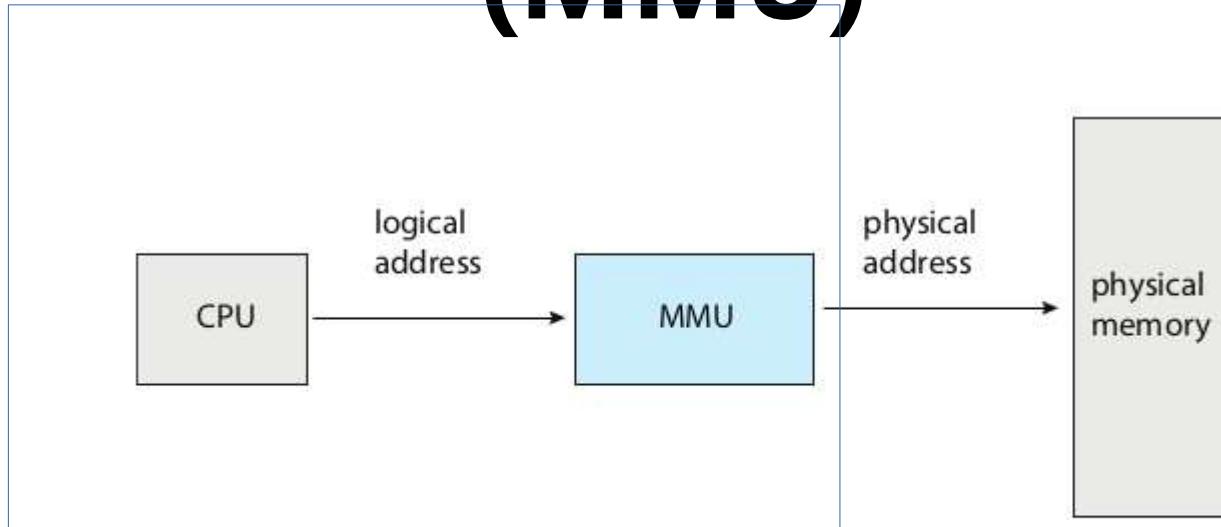
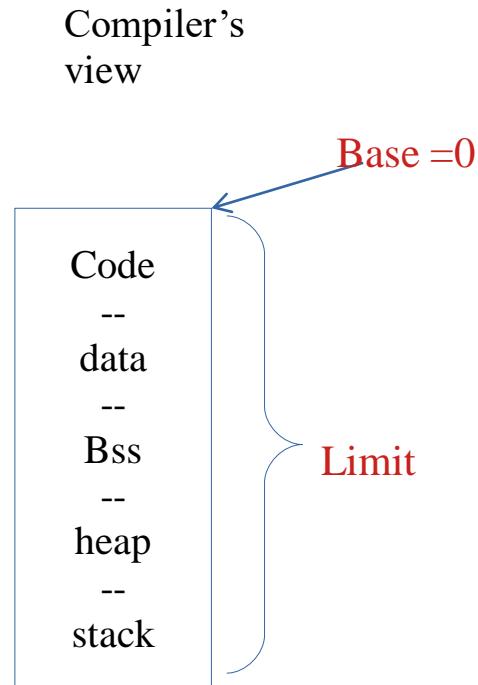
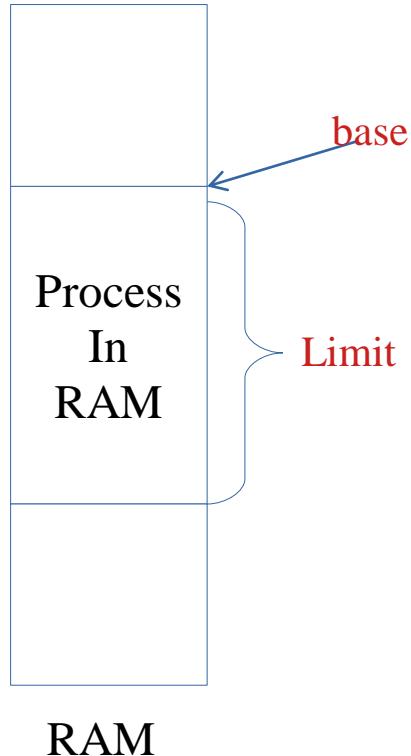


Figure 9.4 Memory management unit (MMU).

- Is part of the CPU chip, acts on every memory address issue by execution unit of the CPU

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

Base/Relocation + Limit scheme

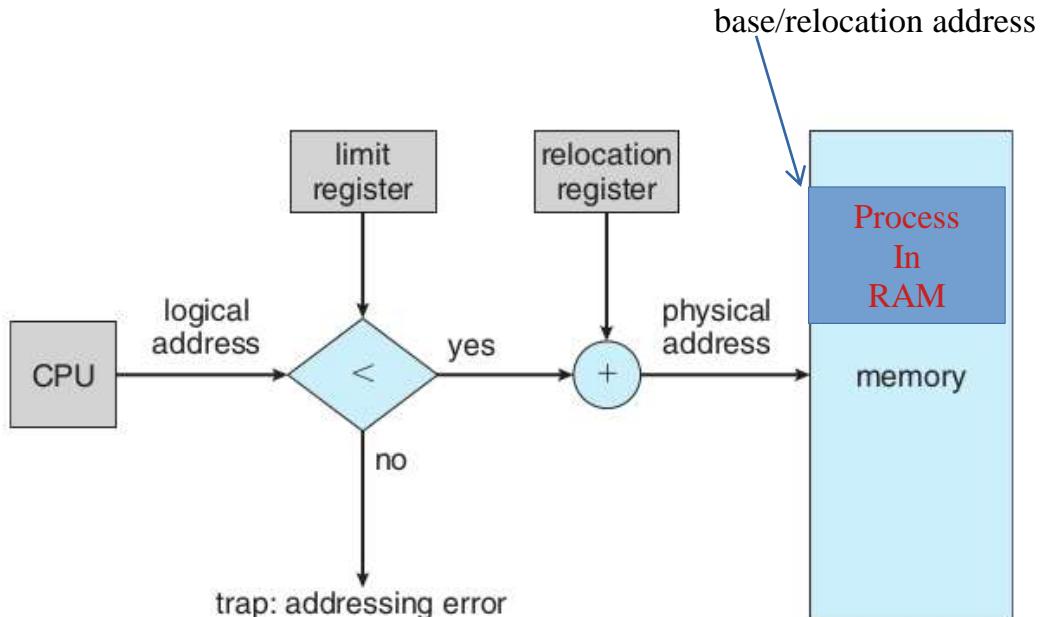


Figure 9.6 Hardware support for relocation and limit registers.

- Loader's job
 - While loading the process in memory
 - must load as one continuous segment
 - Find an empty slot for this!
 - Remember the

Base/Relocation + Limit scheme

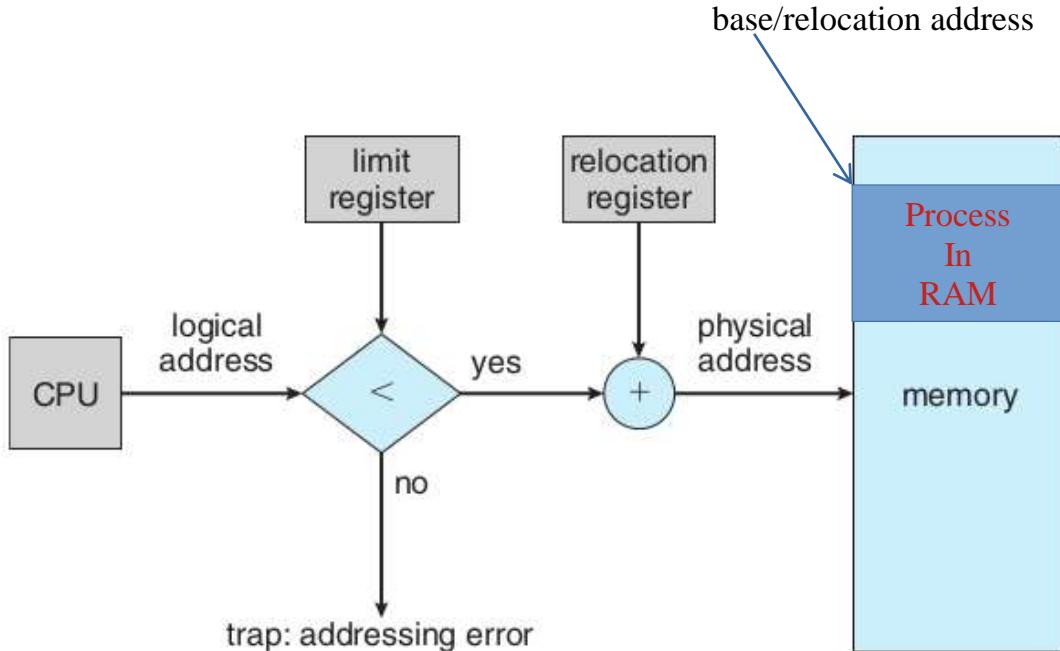
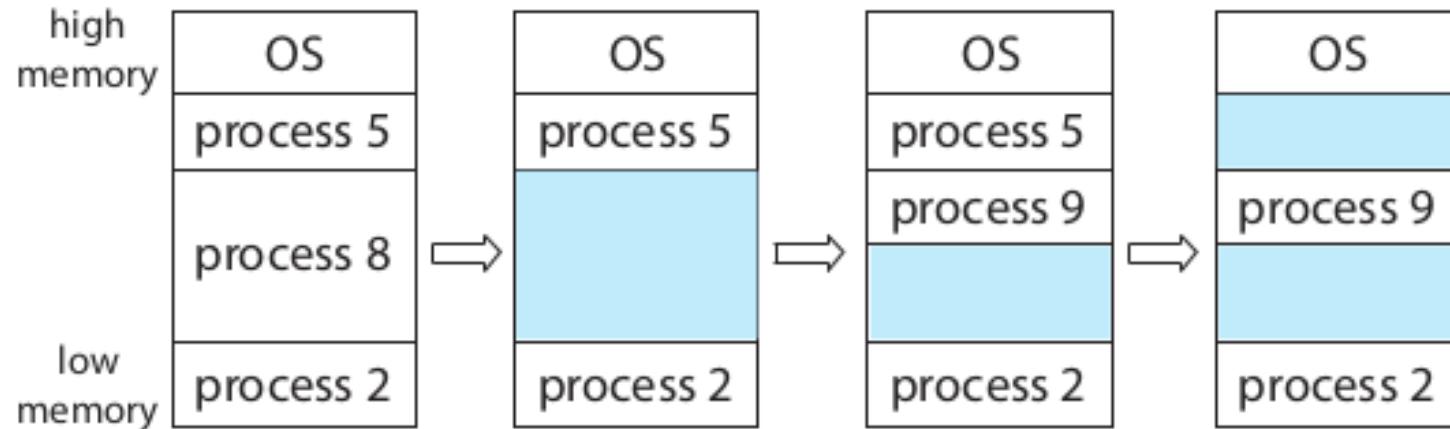


Figure 9.6 Hardware support for relocation and limit registers.

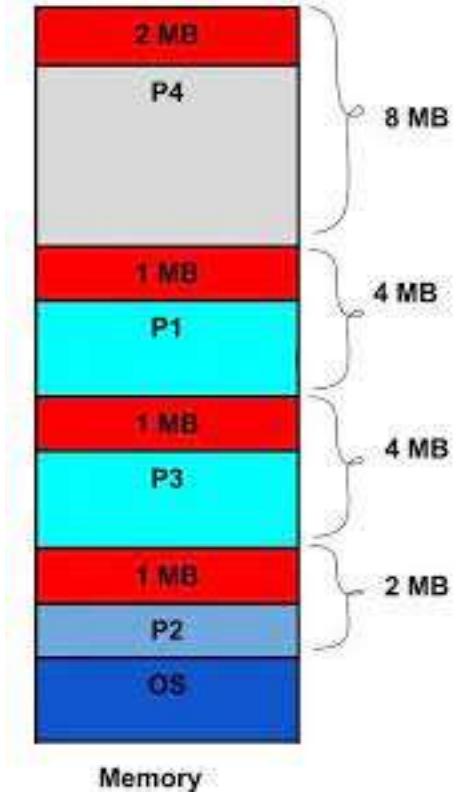
.Combined effect
– “**Relocatable code**” – the process can go anywhere in RAM at the time of loading
– Some memory

Example scenario of memory in base+limit scheme



Process 8 called `fork()` Process 9 forked Process 5 terminated

Continuous memory management and external fragmentation problem



Free chunks: 2 MB, 1MB, 1MB, 1MB

Total 5 MB is available!

Can we create a process of size 3MB?

No! - 3 MB not continuous!

External Fragmentation

- .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!

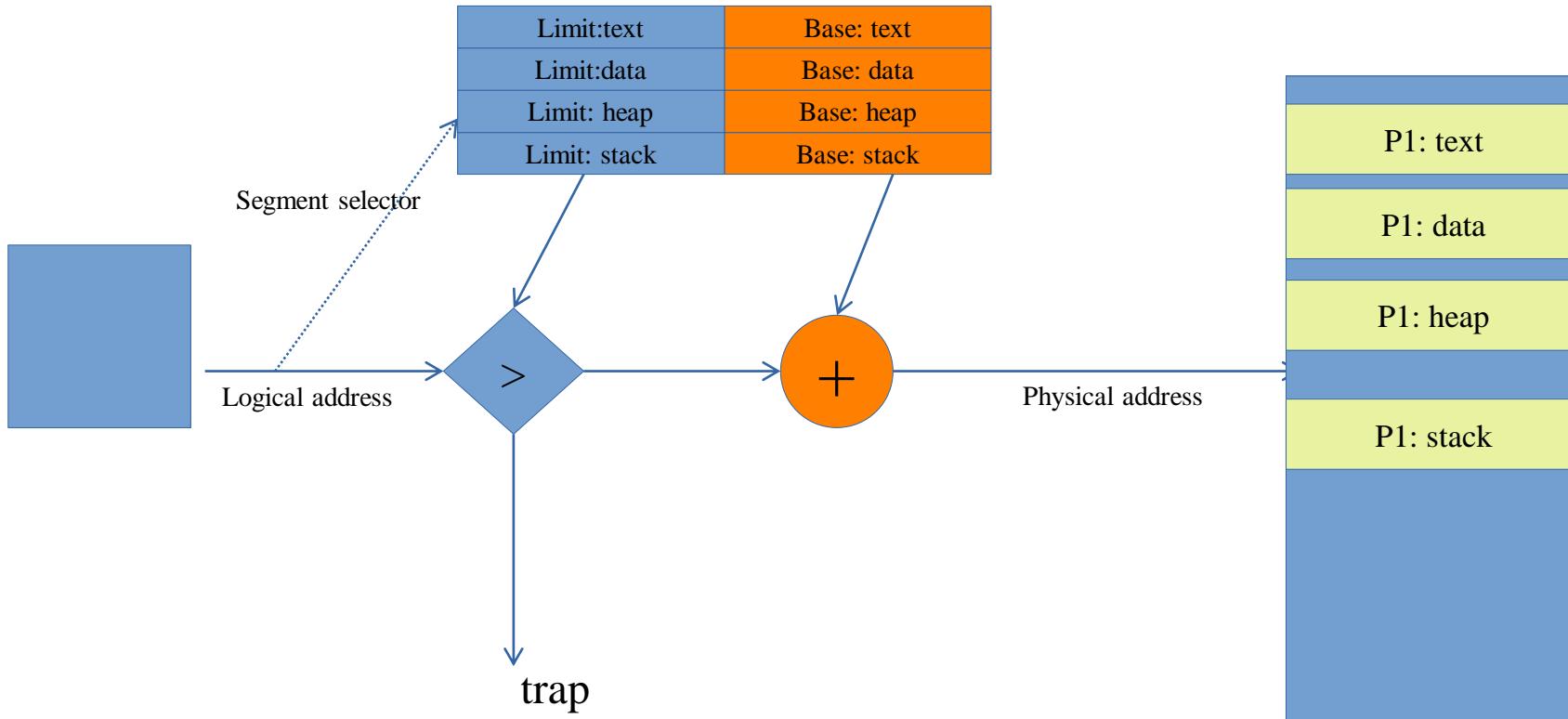
It should be possible to have relocatable code even
with “simplest case”

By doing extra work during “loading”.

How?

(Ans: loader replaces all addresses in the code! That is
quite a lot of work, and challenging too)

Next scheme: Multiple base + limit pairs



Specifying “selector”

- Explicitely
 - Mov ES: 300, \$30
 - Here selector is Extra Segment Register and Logical address (or offset) is 300
- Implicitely, like in x86
 - Mov 300, \$30
 - Here Data Segment register is “implicit” selector

Next scheme: Segmentation

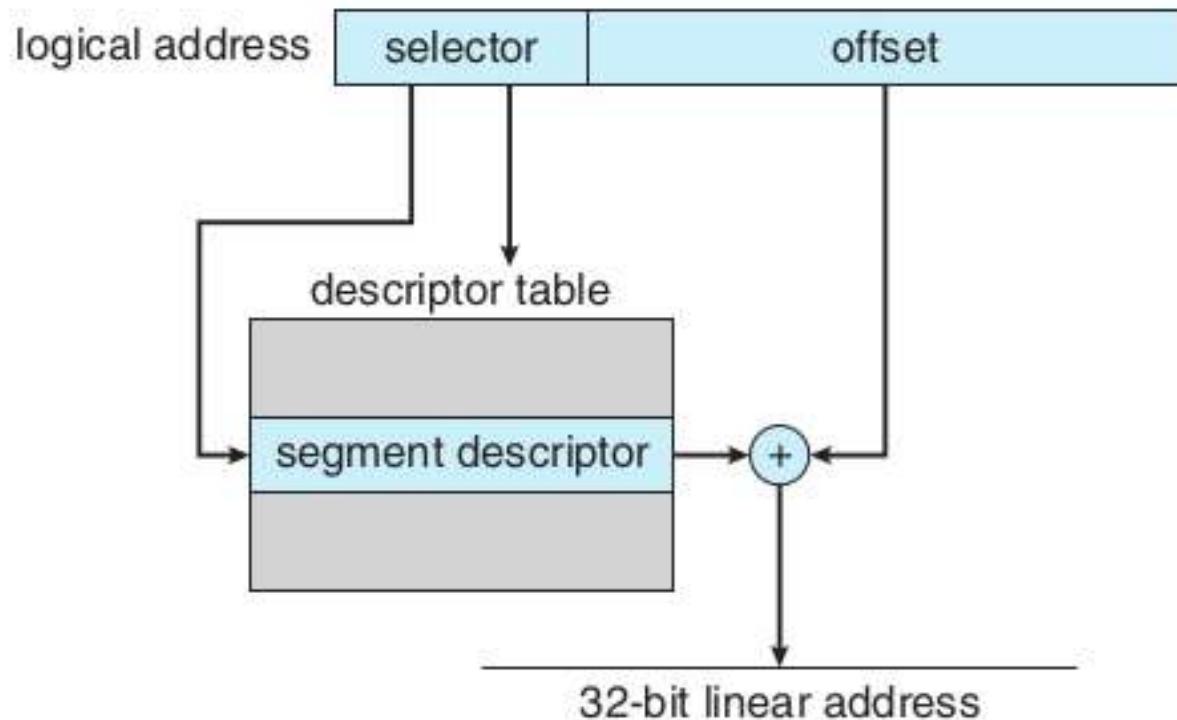
Multiple base + limit pairs

- 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 each segment. And

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

- Base + limit pairs can also be stored in some memory location (not in registers). Then it's called **Segment Table**.
 - Question: how will the cpu know where it's in memory?
 - One CPU register to point to the location of table in memory
- . **Segment Table Base Register (STBR)**
- X86 has two tables. Local Descriptor Table and Global Descriptor Table. Both are segment tables.
- Accordingly it has LDTR and GDTR.

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



Note:

"Selector" here is normally a segment

The machine code only has offset in

The "Descriptor table" is a segment

Figure 9.22 IA-32 segmentation.

Segmentation and External fragmentation

- Does segmentation also suffer from external fragmentation?
 - Yes!

How many segment tables?

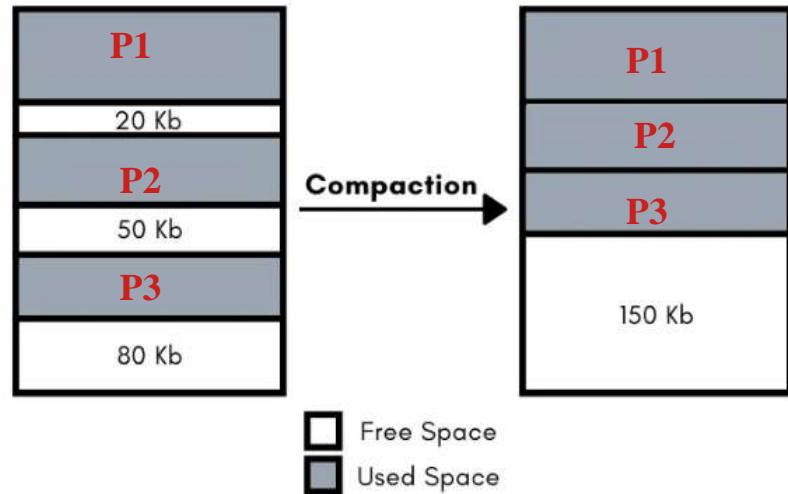
- One per process
- One for the kernel!
 - Remember: kernel code also undergoes translation in MMU!

A point of confusion on “root” user

- “root” user has privileges in accessing files
- When “root” user runs an application, it runs in process-mode, not in kernel-mode!
 - That process can do “more” but , it’s not kernel mode code!
- Kernel is not run by “root” user!
 - Kernel is not run by any user

Solution to external fragmentation

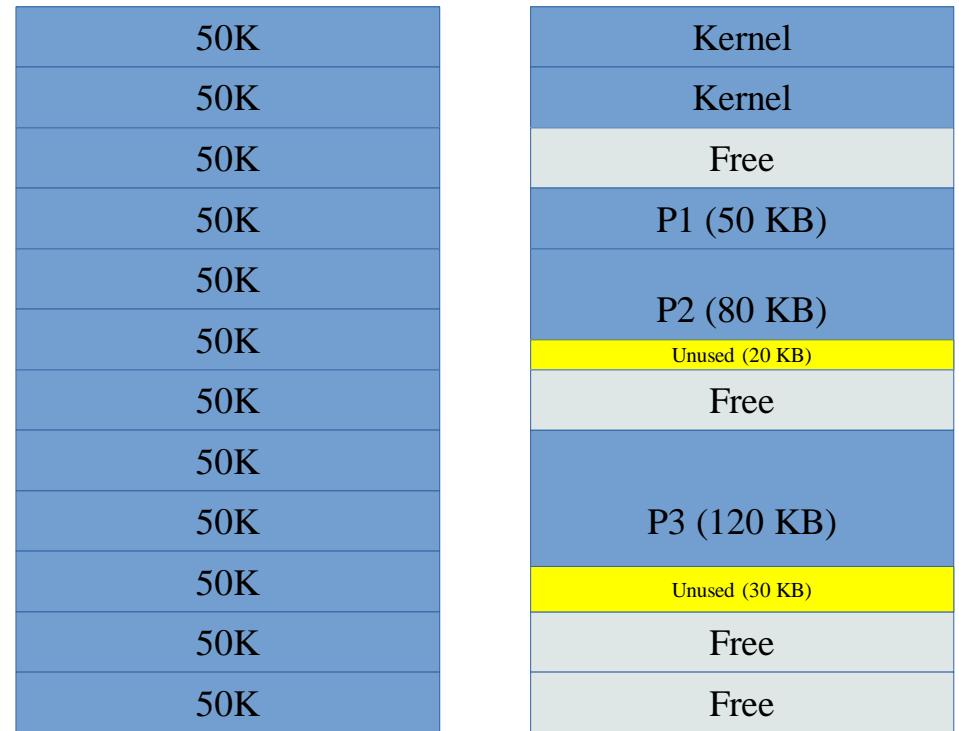
- .Compaction !
- .OS moves the process chunks in memory to make available continuous memory region
- Then it must update the memory



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,



Solving external fragmentation problem

- Process should not be continuous in memory!
 - The trouble is finding a big continuous chunk!
- 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*

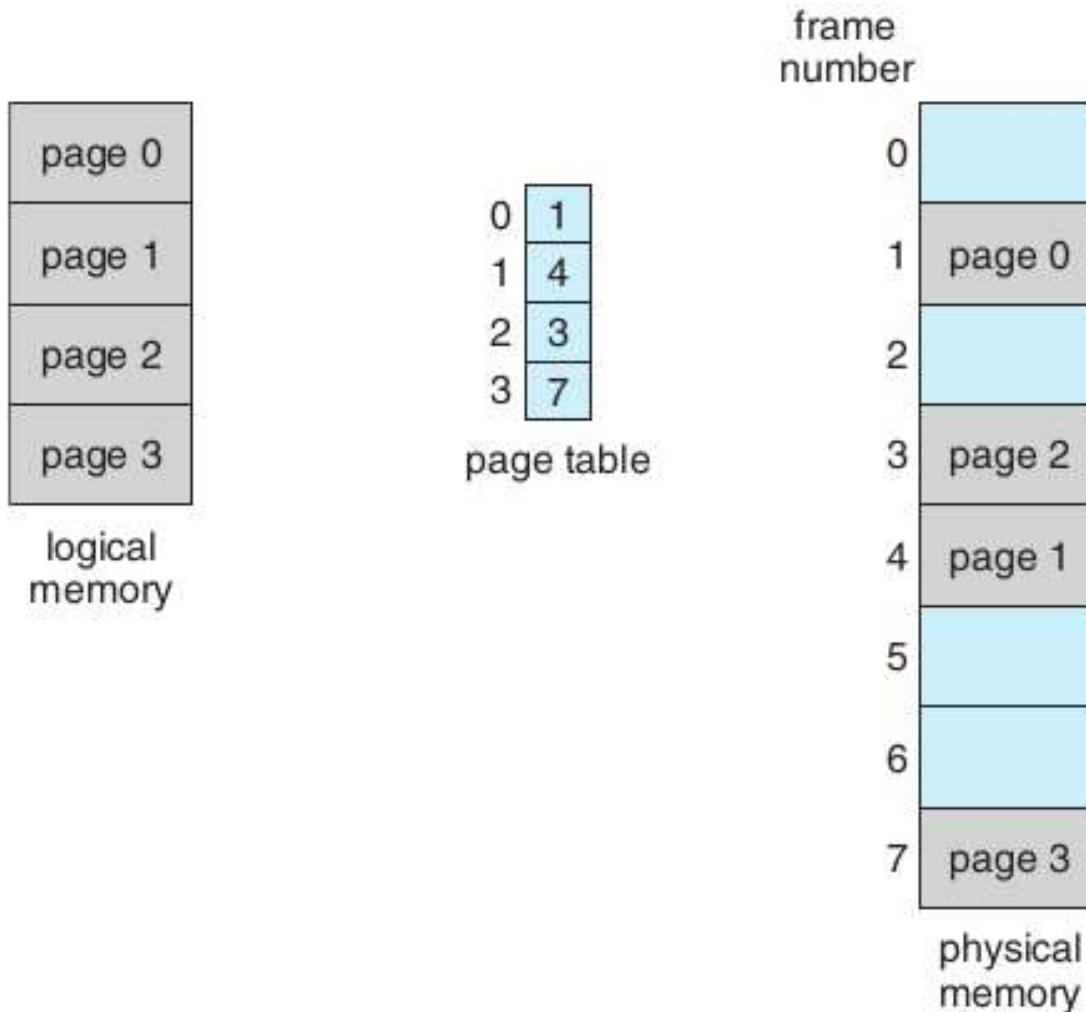


Figure 9.9 Paging model of logical and physical memory.

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.

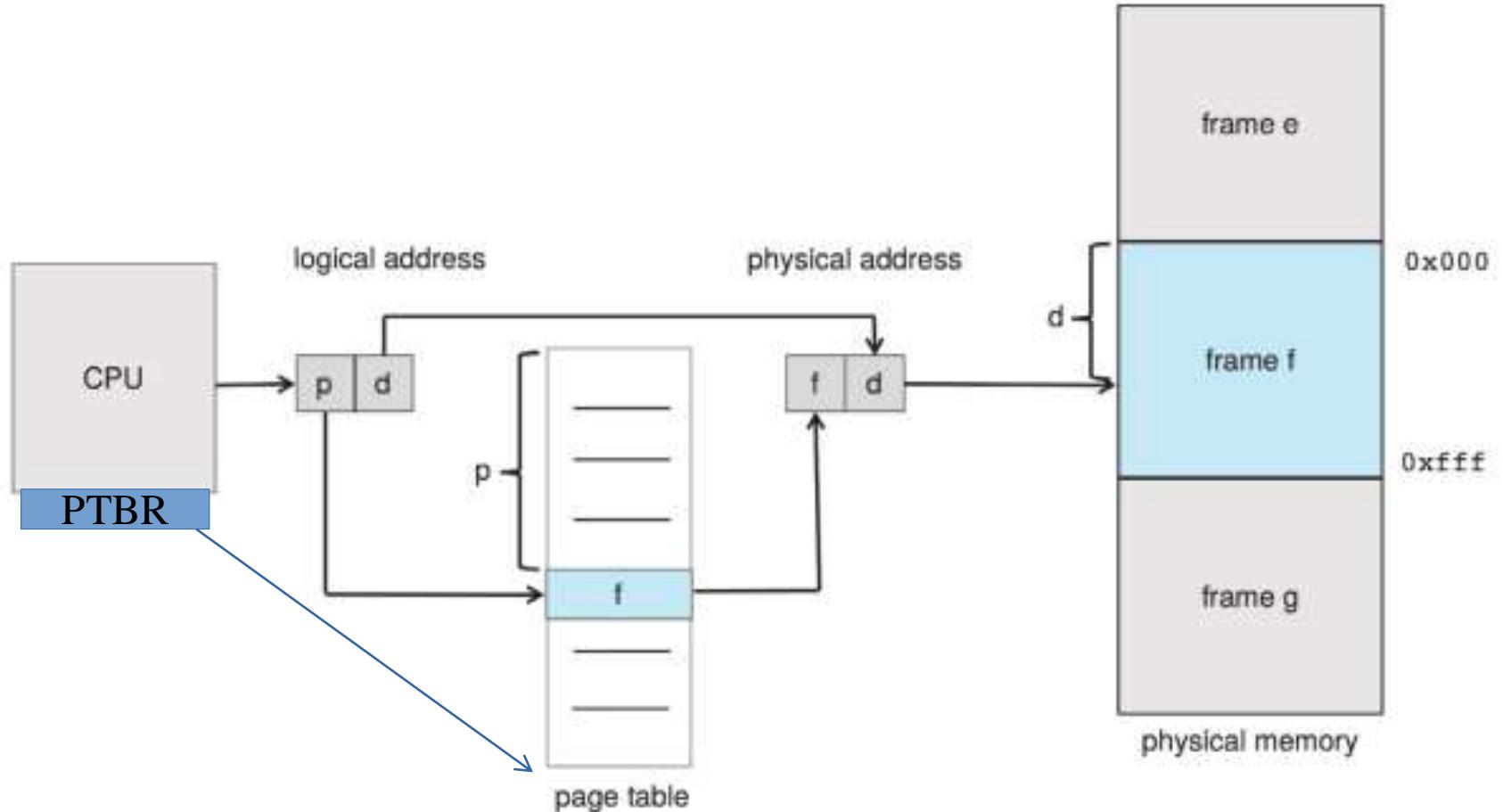


Figure 9.8 Paging hardware.

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 (PTBR) inside CPU will give location of an in memory table called page

Paging

- .Compiler: assume the process to be one continuous chunk of memory (!) . Generate addresses accordingly . Compiler still treats text,data,bss,.. to be separate chunks, but in multiples of page-sizes.
- .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

free-frame list

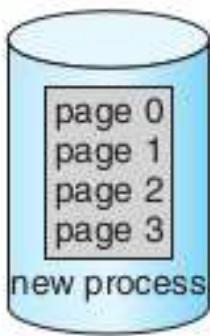
14

13

18

20

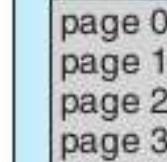
15



(a)

free-frame list

15



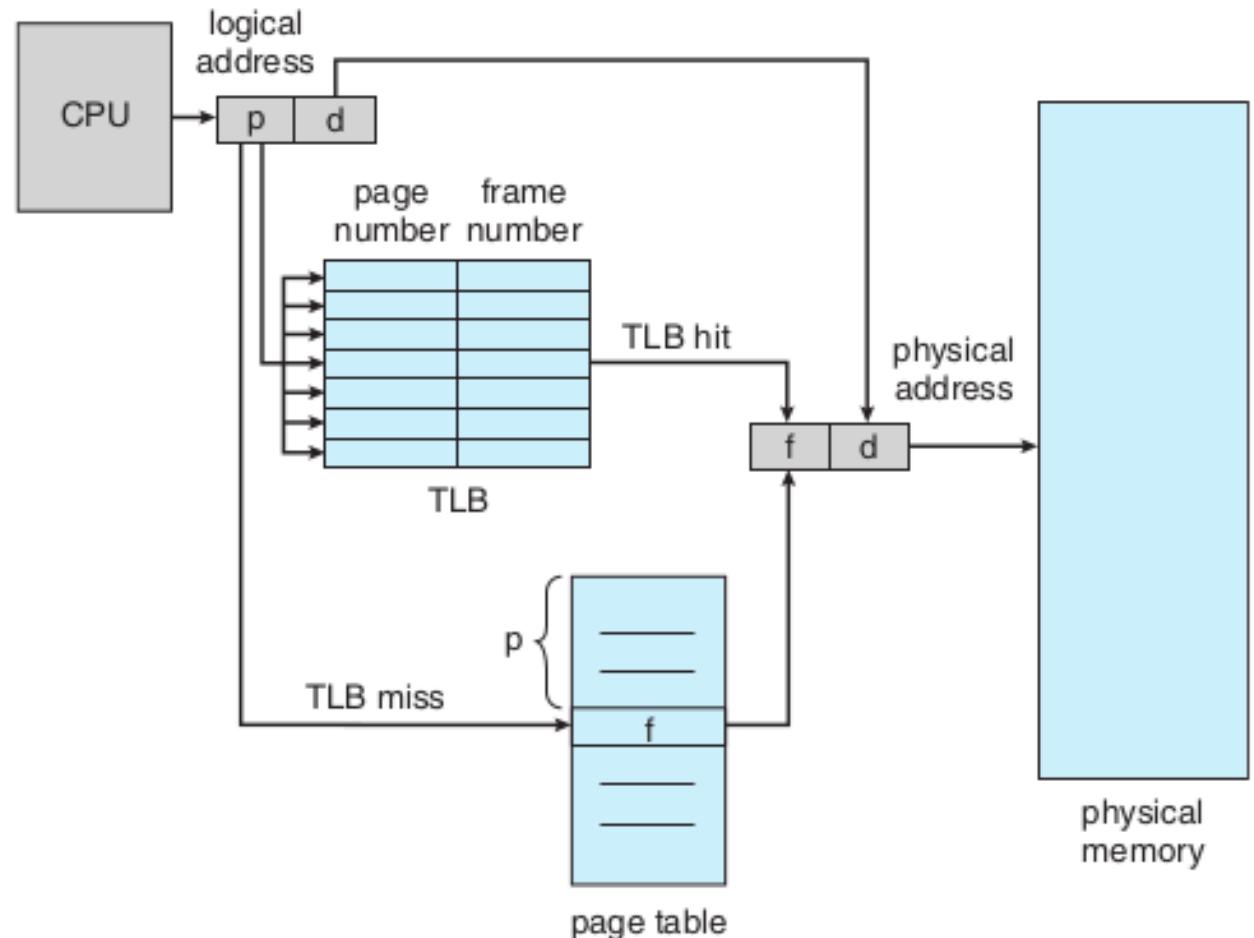
Speeding up paging

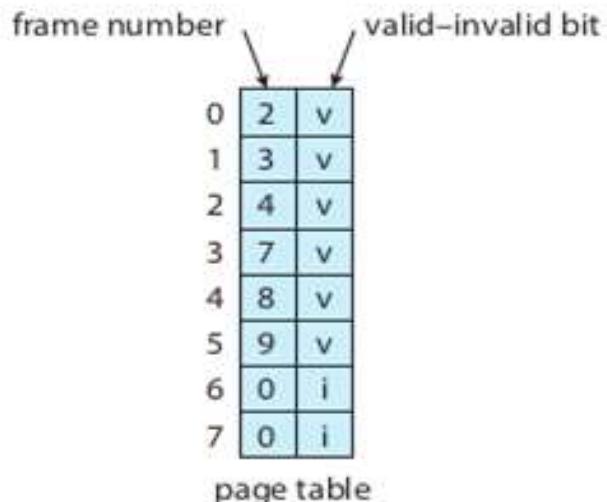
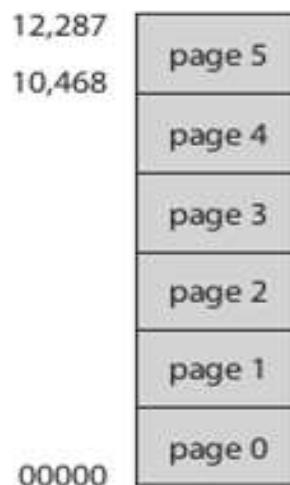
- Translation Lookaside Buffer (TLB)

- Part of CPU hardware

- A cache of Page table entries

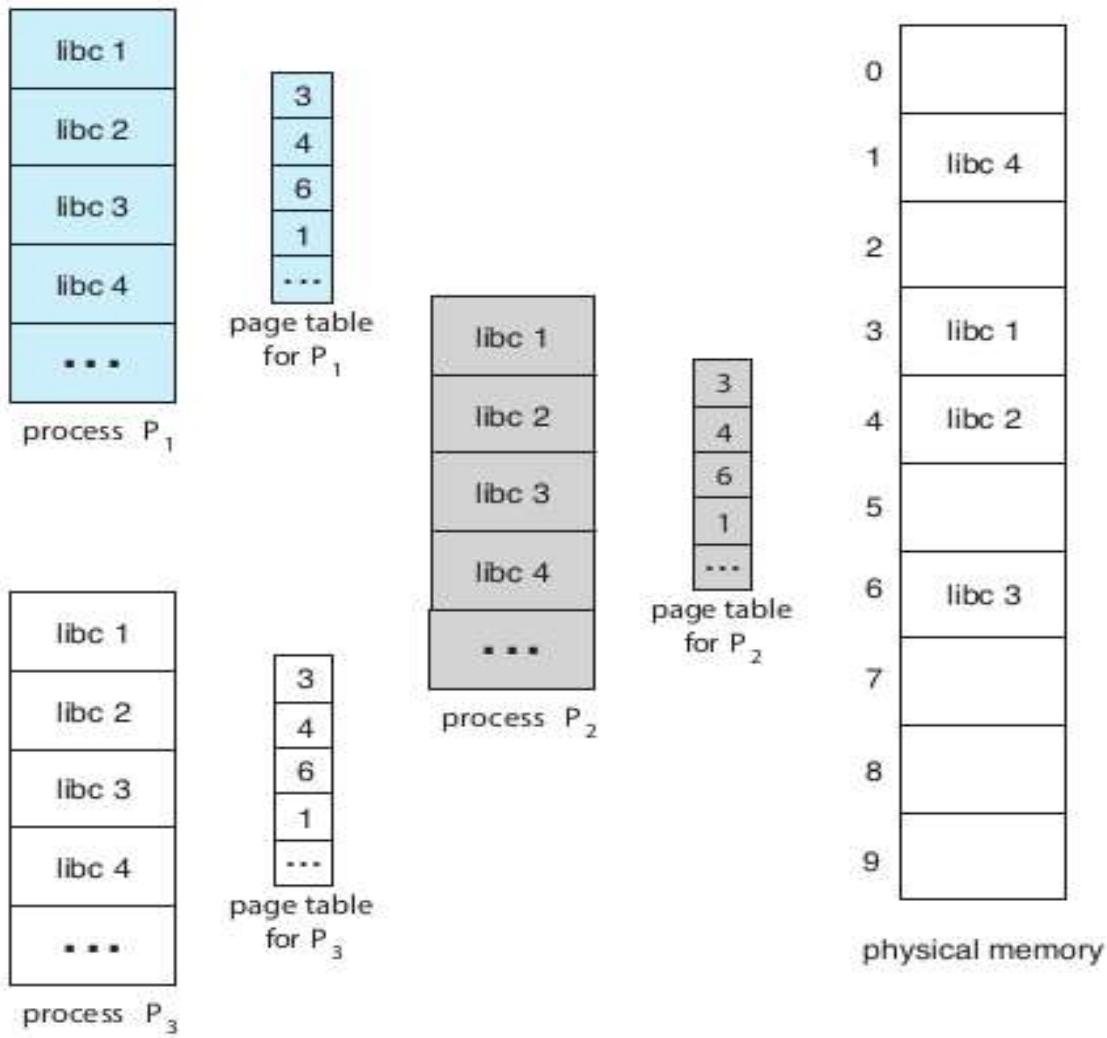
- Searched in parallel for a





Memory protection with paging

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



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

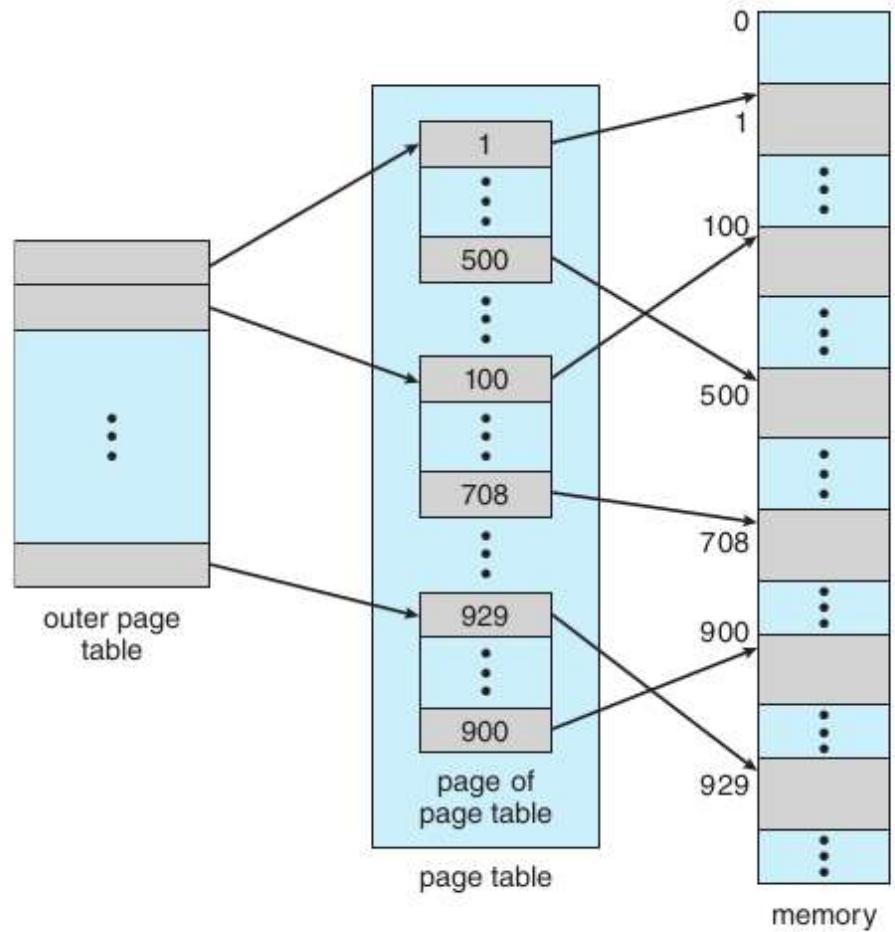
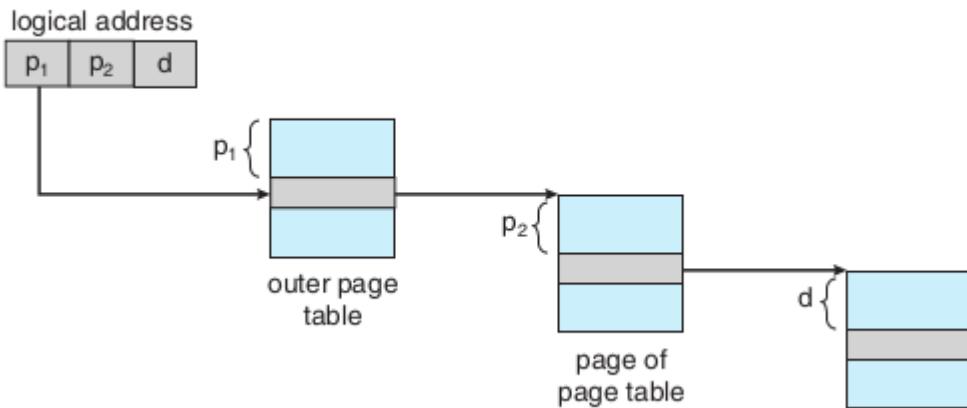
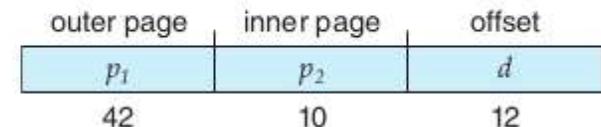


Figure 9.15 A two-level page-table scheme.



X86 memory management

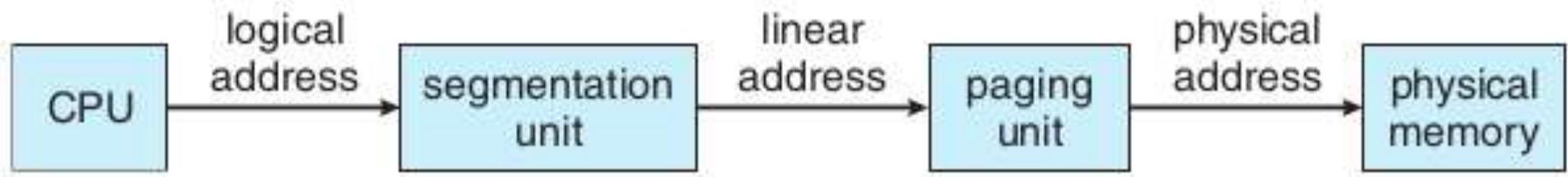


Figure 9.21 Logical to physical address translation in IA-32.

Segmentation in x86

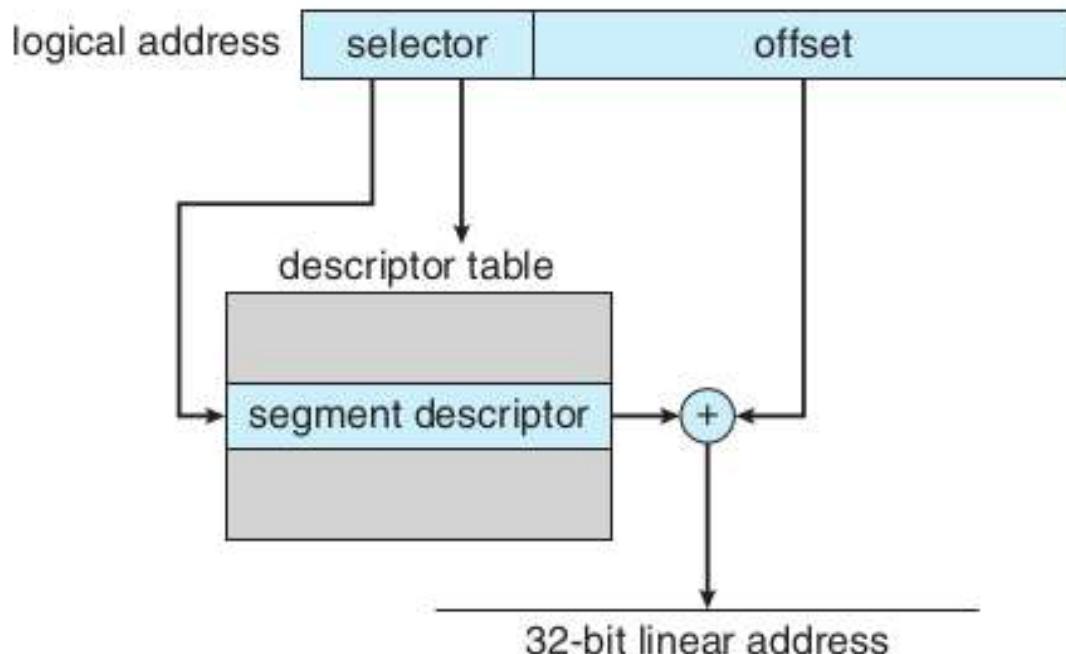


Figure 9.22 IA-32 segmentation.

.The selector is automatically chosen using Code Segment (CS) register, or Data Segment (DS) register depending on which type of

Paging in x86

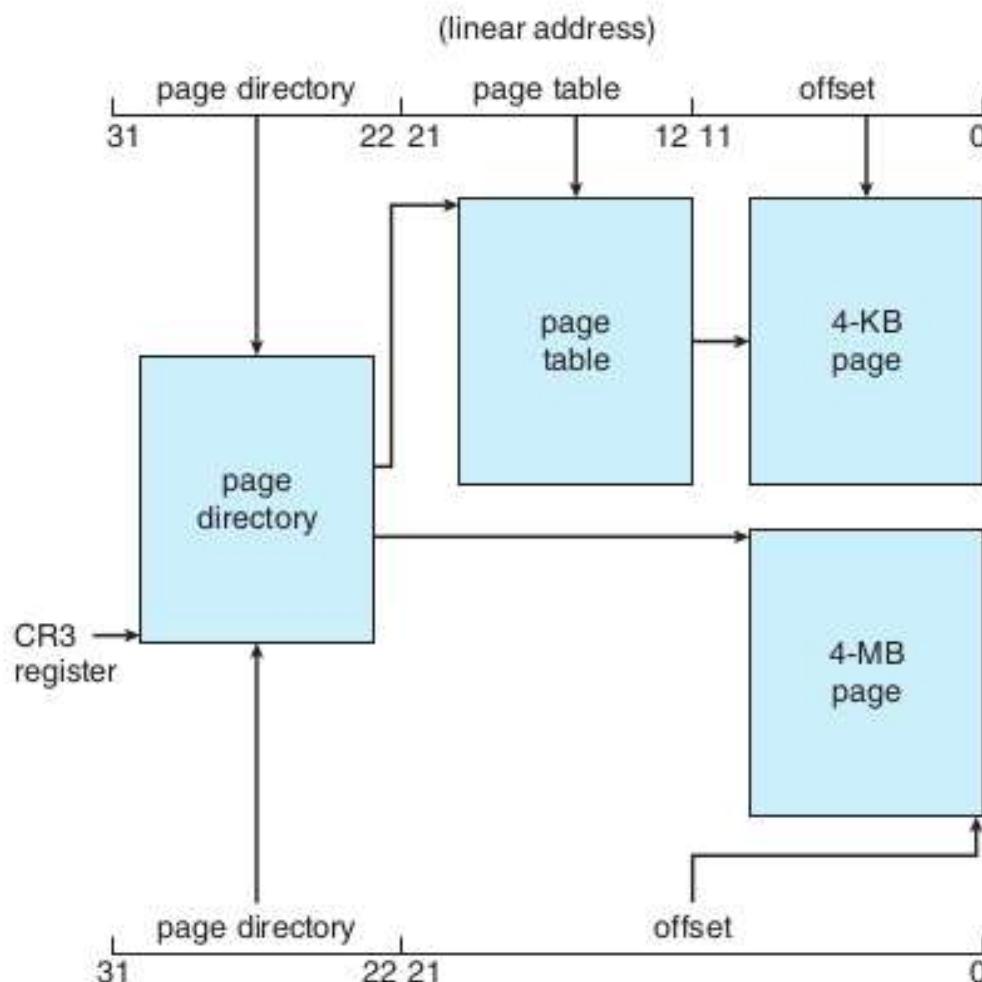
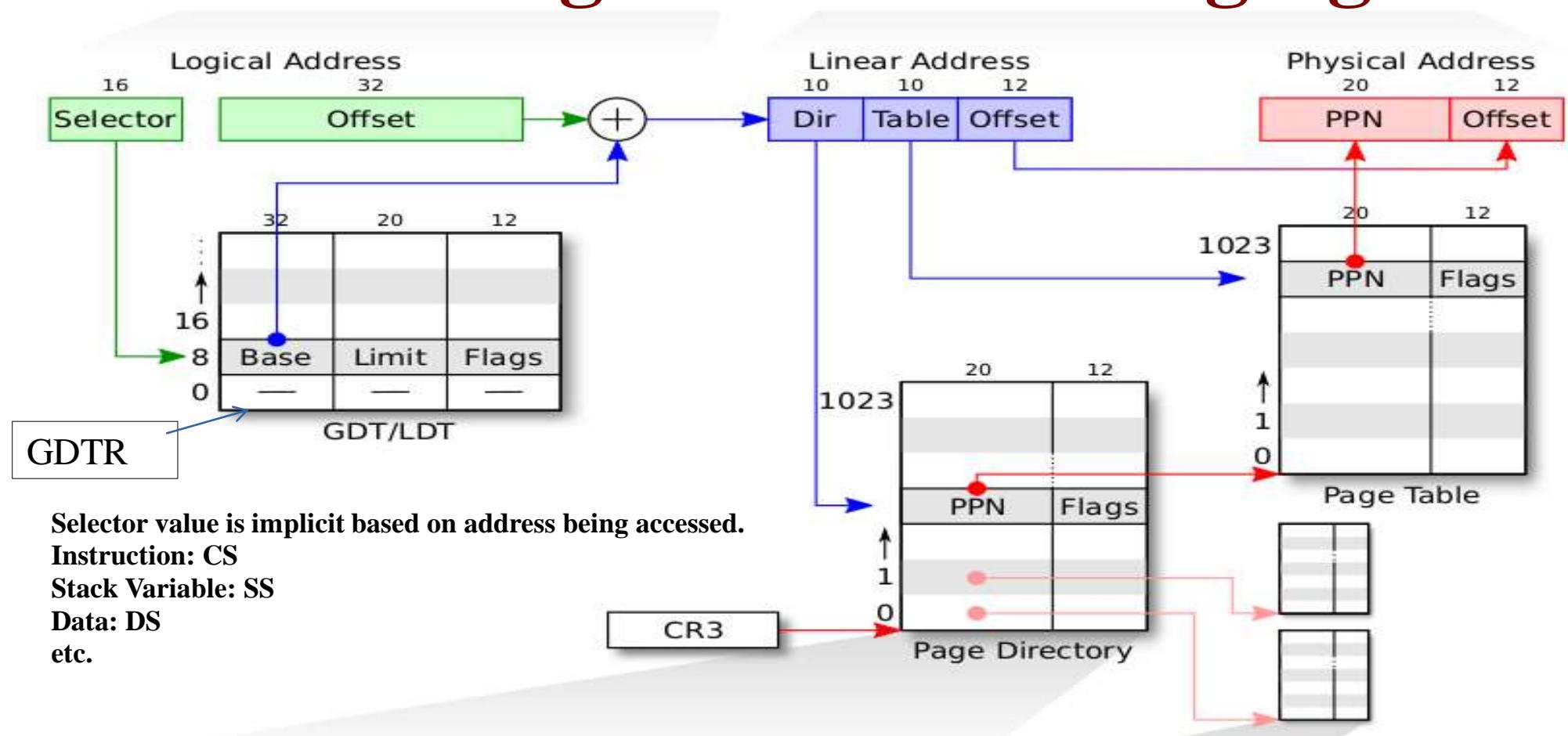


Figure 9.23 Paging in the IA-32 architecture.

- 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

X86 Segmentation + Paging



X86 Segmentation + Paging

- Paging is optional, segmentation compulsory
- Setting flags in Control Registers (CR) enables this
 - Page Table, Page Directory, page - are all size=4k, if 2-level paging is used
 - Makes life simpler for the kernel

Basics of X86 architecture

Abhijit A M

abhijit.comp@coep.ac.in

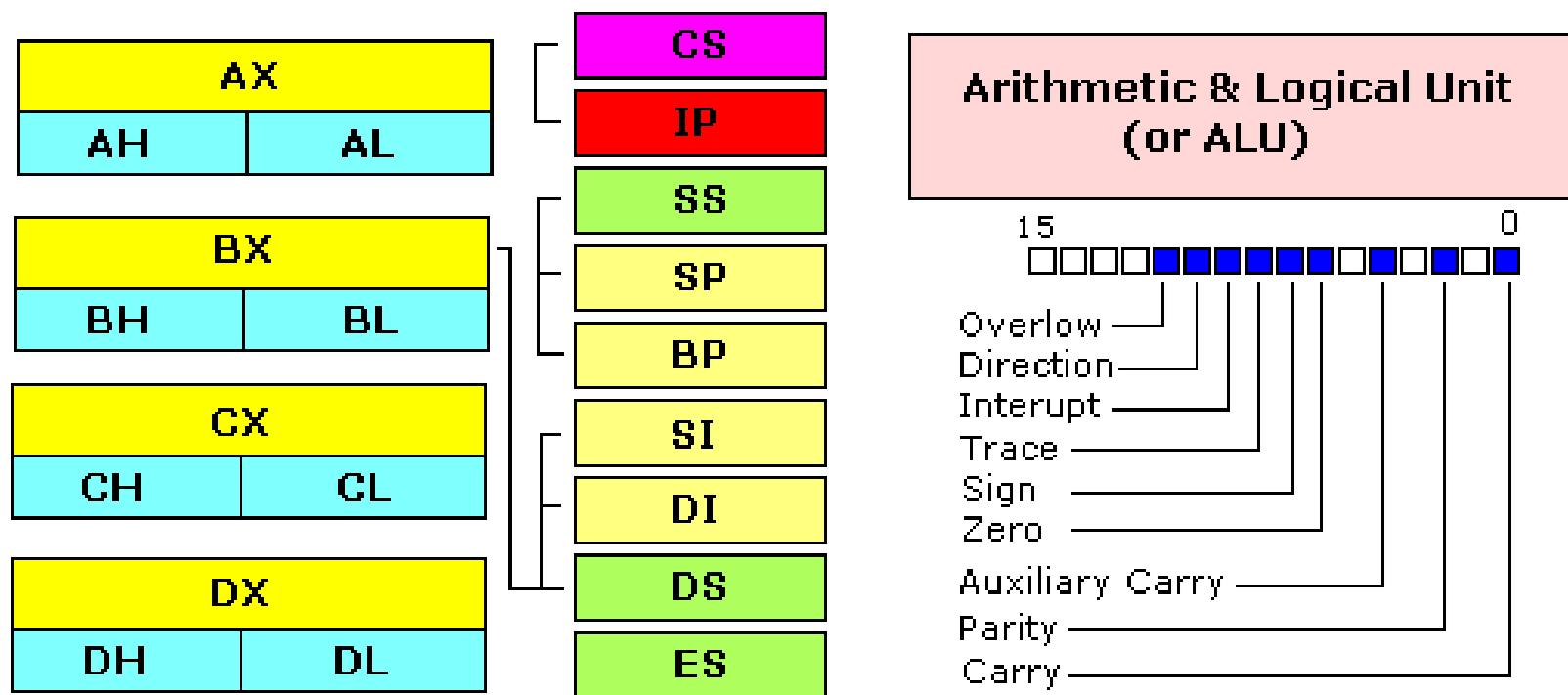
Credits: Notes by Prof. Sorav Bansal, <https://www.felixcloutier.com/x86/>, Intel Documentation

A processor typically has

- integer registers and their execution unit
- floating-point/vector registers and execution unit(s)
- memory management unit (MMU)
- multiprocessor/multicore: local interrupt controller (APIC)
- etc

8086: 16 bit CPU (precursor to x86 family)

Central Processing Unit (or CPU)



8086 Registers

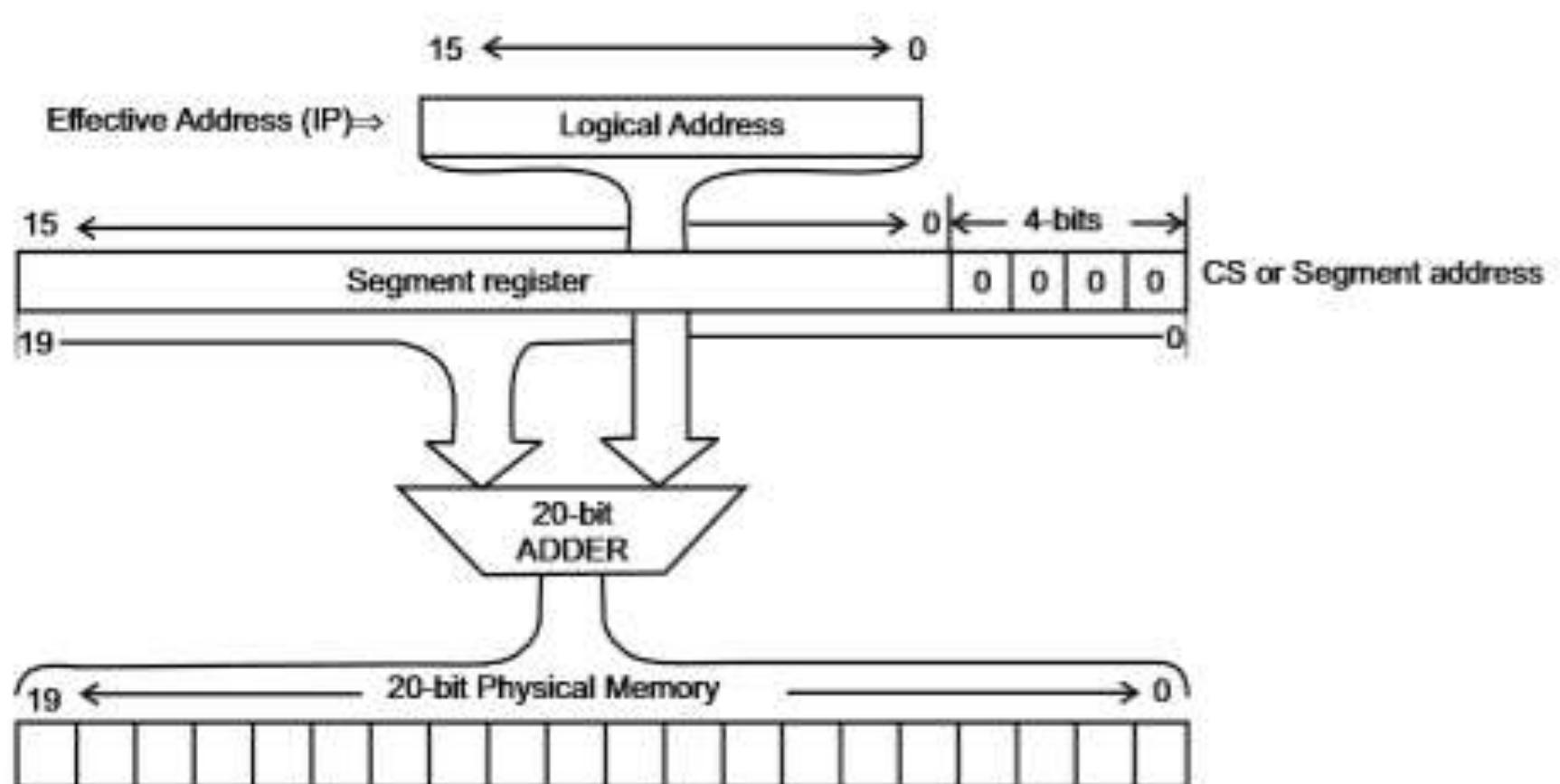
- 16 bit ought to be enough ! (?)
- General purpose registers
 - four 16-bit data registers: AX, BX, CX, DX
 - each in two 8-bit halves, e.g. AH and AL
- Registers for memory addressing
 - SP, BP, SI, DI : 16 bit
 - SP stack pointer, BP base pointer, SI Source Index, DI Destination Index
 - IP instruction Pointer
 - Addressable memory: $2^{16} = 64\text{ kb}$

8086 address extension

- 8086 has 20-bit physical addresses ==> 1 MB AS
- the extra four bits usually come from a 16-bit "segment register":
 - CS - code segment, for fetches via IP
 - SS - stack segment, for load/store via SP and BP
 - DS - data segment, for load/store via other registers
 - ES - another data segment, destination for string operations
- %cs:%ip full address: %cs * 16 + %ip is actual

8086 address extension

- Extending ‘address space’ with the help of MMU



FLAGS register

- FLAGS - various condition codes: whether last arithmetic operation
 - overflowed
 - was positive/negative
 - was [not] zero
 - carry/borrow on add/subtract
 - etc.
- whether interrupts are enabled
- direction of data copy instructions

32 bit 80386

- boots in 16-bit mode, then on running particular instructions switches to 32-bit mode (protected mode)
 - For backward compatibility, INTEL continued this. The CPU starts in 16 bit mode, called real mode
 - If particular instructions are not executed (may be in boot loader) all processors will continue in real mode
- registers are 32 bits wide, called **EAX** rather than **AX**

32 bit 80386

- Still possible to access 16 bit registers using AX or BX
- Specific **coding of machine instructions** to tell whether operands are 16 or 32 bit
- prefixes **0x66/0x67** toggle between 16-bit and 32-bit **operands/addresses** respectively
- in 32-bit mode, MOVW is expressed as **0x66 MOVW**
- the **.code32** in bootasm.S tells assembler to generate **0x66** for e.g. MOVW
- 80386 also changed segments and added paged

Summary of registers in 80386 (32 bit)

□ General registers

- 32 bits : **EAX EBX ECX EDX**
- 16 bits : **AX BX CX DX**
- 8 bits : **AH AL BH BL CH CL DH DL**

Summary of registers in 80386

□ Expected usage of Segment Registers

- CS: Holds the **Code segment** in which your program runs.
- DS : Holds the **Data segment** that your program accesses.
- ES,FS,GS : These are extra segment registers
- SS : Holds the **Stack segment** your program uses.

□ All 16 bit

Summary of registers in 80386

- For a typical register, the corresponding segment is used. Pairs of Indexes & pointers (Segment & Registers)

- CS:EIP

- Code Segment: Index Pointer

- E.g. mov \$32, %eax ==> The code of move instruction uses

- SS:EBP

- Stack Segment: EBP ==> mov (%ebp), %eax ==> for accessing (%ebp) SS: EBP address will be used

- DS:ESI , ES: EDI .

- ESI: Extended Source Index, EDI: Extended

X86 Assembly Code

□ Syntax

- Intel syntax: op dst, src (Intel manuals!)
 - AT&T (gcc/gas) syntax: op src, dst (xv6)
 - uses b, w, l suffix on instructions to specify size of operands
-
- Operands are registers, constant, memory via register, memory via constant

Examples of X86 instructions

AT&T syntax	"C"-ish equivalent	Operands
movl %eax, %edx	edx = eax;	register mode
movl \$0x123, %edx	edx = 0x123;	immediate
movl 0x123, %edx	edx = *(int32_t*)0x123;	direct
movl (%ebx), %edx	edx = *(int32_t*)ebx;	indirect
movl 4(%ebx), %edx	edx = *(int32_t*)(ebx+4)	displaced

Instructions suffix/prefix

`mov %eax, %ebx # 32 bit data`

`movw %ax, %bx # move 16 bit data`

`mov %ax, %bx # ax is 16 bit, so equivalent
to movw`

`mov $123, 0x123 # Ambigious`

`movw $123, 0x123 # correct, move 16 bit data`

Types of Instructions

- **data movement**

- **MOV, PUSH, POP, ...**

- **Arithmetic**

- **TEST, SHL, ADD, AND, ...**

- **i/o**

- **IN, OUT, ...**

- **Control**

- **JMP, JZ, JNZ, CALL, RET**

- **String**

- **REP MOVSB, ...**

- **System**

- **IRET, INT**

Interrupt handling

Privilege levels

- 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 CPL field inside %cs register

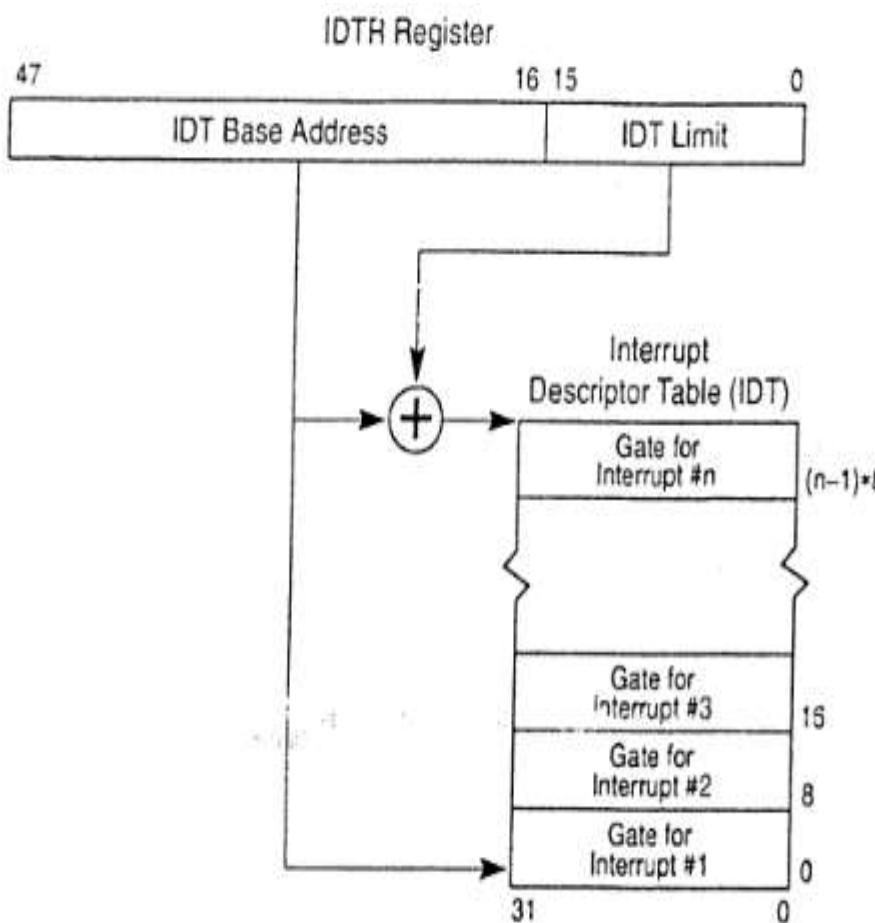
Privilege levels

- 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 is an in memory table. 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

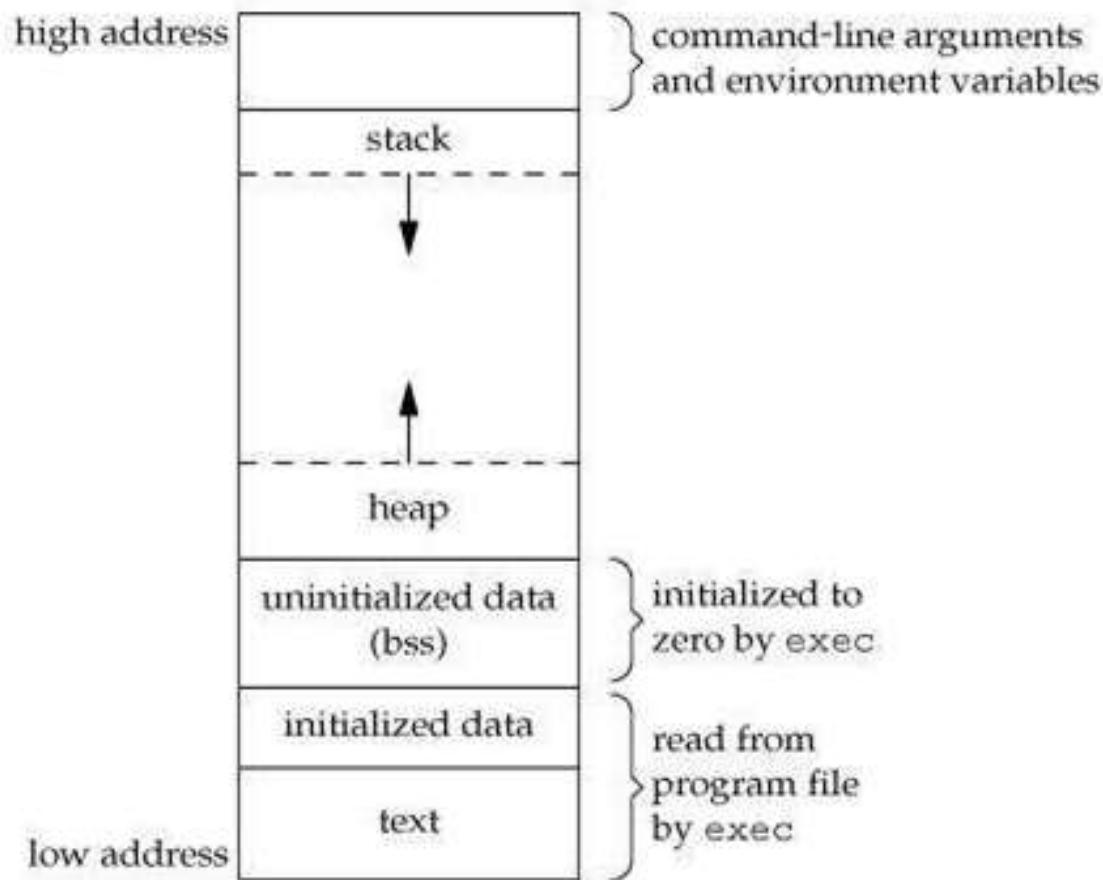
IDTR and IDT



- An entry in IDT is called a “gate”
- IDTR is a CPU register, IDT is in memory table

Memory Management in x86

Memory Layout of a C Program



Text: object code of the program

Data: Global variables + (local/global) Static variables

BSS: Uninitialized data, globals that were not initialized

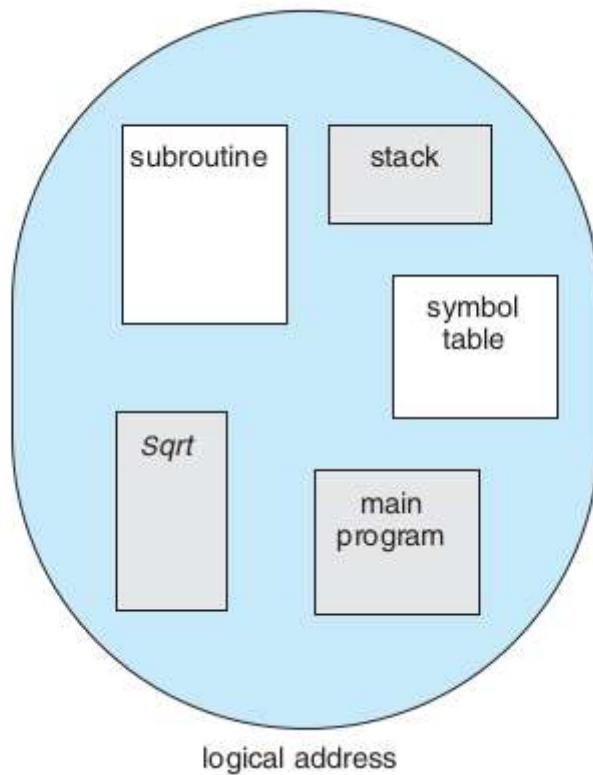
Stack: Region for allocating local variables, function parameters, return values

Heap: Region for use by malloc(), free()

Arguments, Environment variables: Initialized by kernel (during exec)

text, data, bss are also present in the ELF file
stack and heap are not present in ELF file
WHY?

Why the Segment:Offset Pairs In x86 ?



Segmentation
Compiler's view of the program

Compiler generates object code **assuming** that different memory regions corresponding to the program are different “segments” in memory

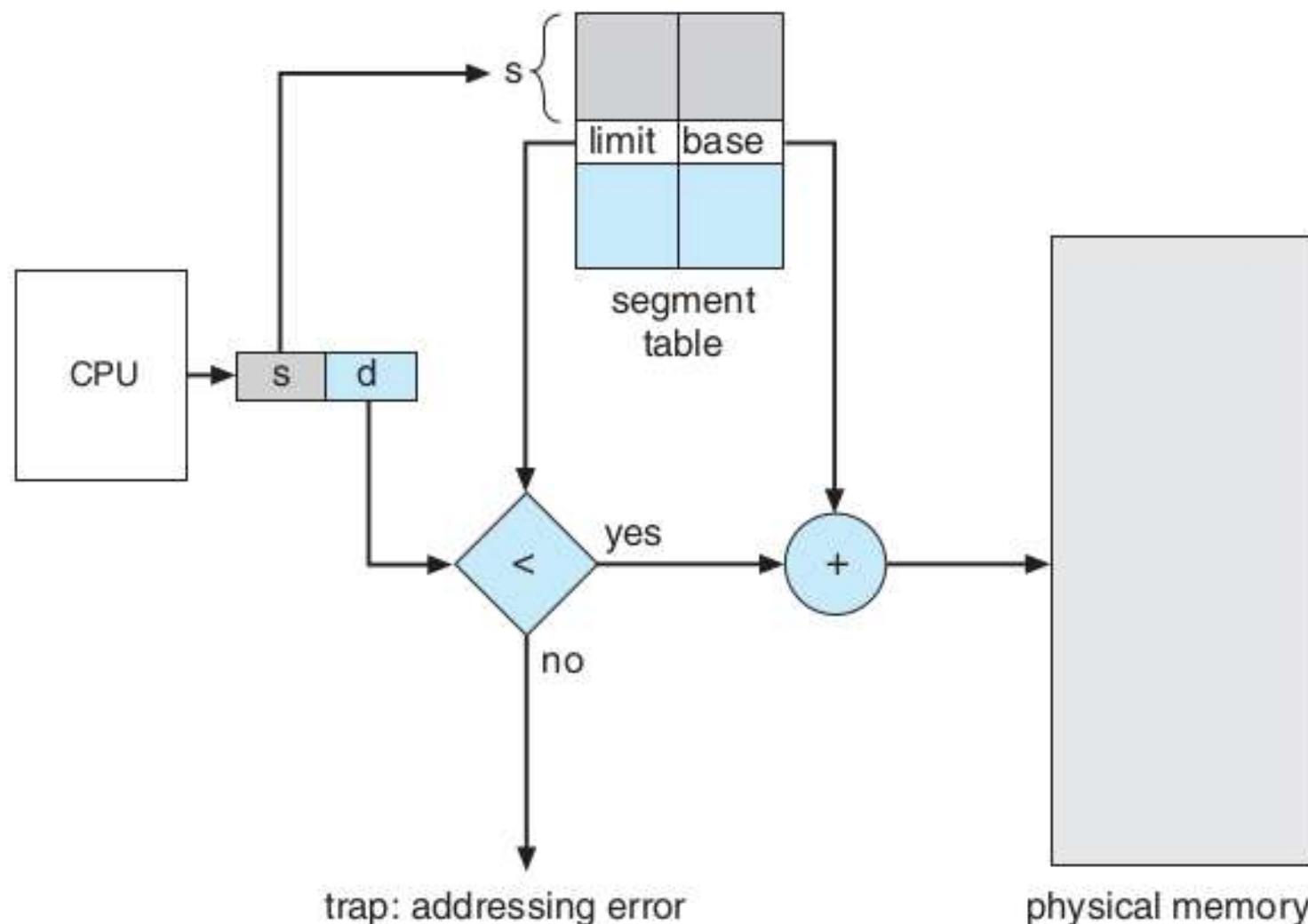
Segment: a continuous chunk

Segment register gives the location of the chunk, index/offset register gives the offset

E.g. in CS:IP CS gives the location of the segment, IP gives the index in it

Segmentation

Address Translation, general idea (not x86)



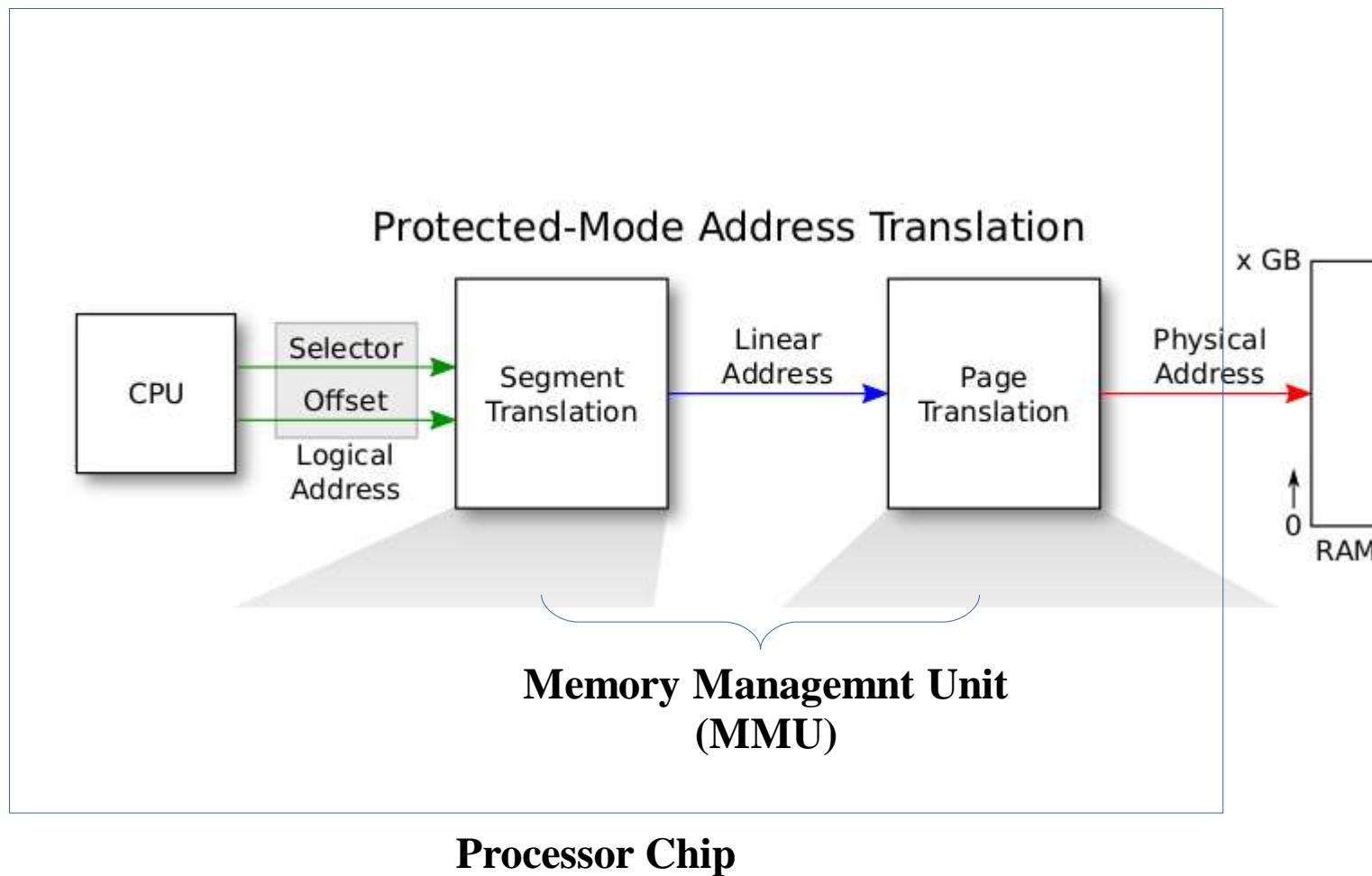
Real mode and protected mode

- Beware: note the same as user mode and kernel mode!
- Backward compatibility was desired by Intel
 - 80286 was 32 bit, 8086 was 16 bit
 - Binary encoding for 80286 was different, registers were 32 bit, etc; also more speed, different memory management hardware, etc.
 - But still they wanted their customers to keep running earlier object code on new processor
 - So they ensured that the processor boots up as if it was 16 bit 8086/8088. REAL MODE!

More on real mode

- addresses in real mode always correspond to real locations in memory.
 - Memory address calculation done by MMU in real mode: segment*16+offset
- no support for memory protection, multitasking, or code privilege levels (user/kernel mode!)
- May use the BIOS routines or OS routines
- DOS like OS were written when PCs were in the era of 8088/real-mode.
 - Single tasking systems

X86 address : protected mode address translation



X86 segmentation

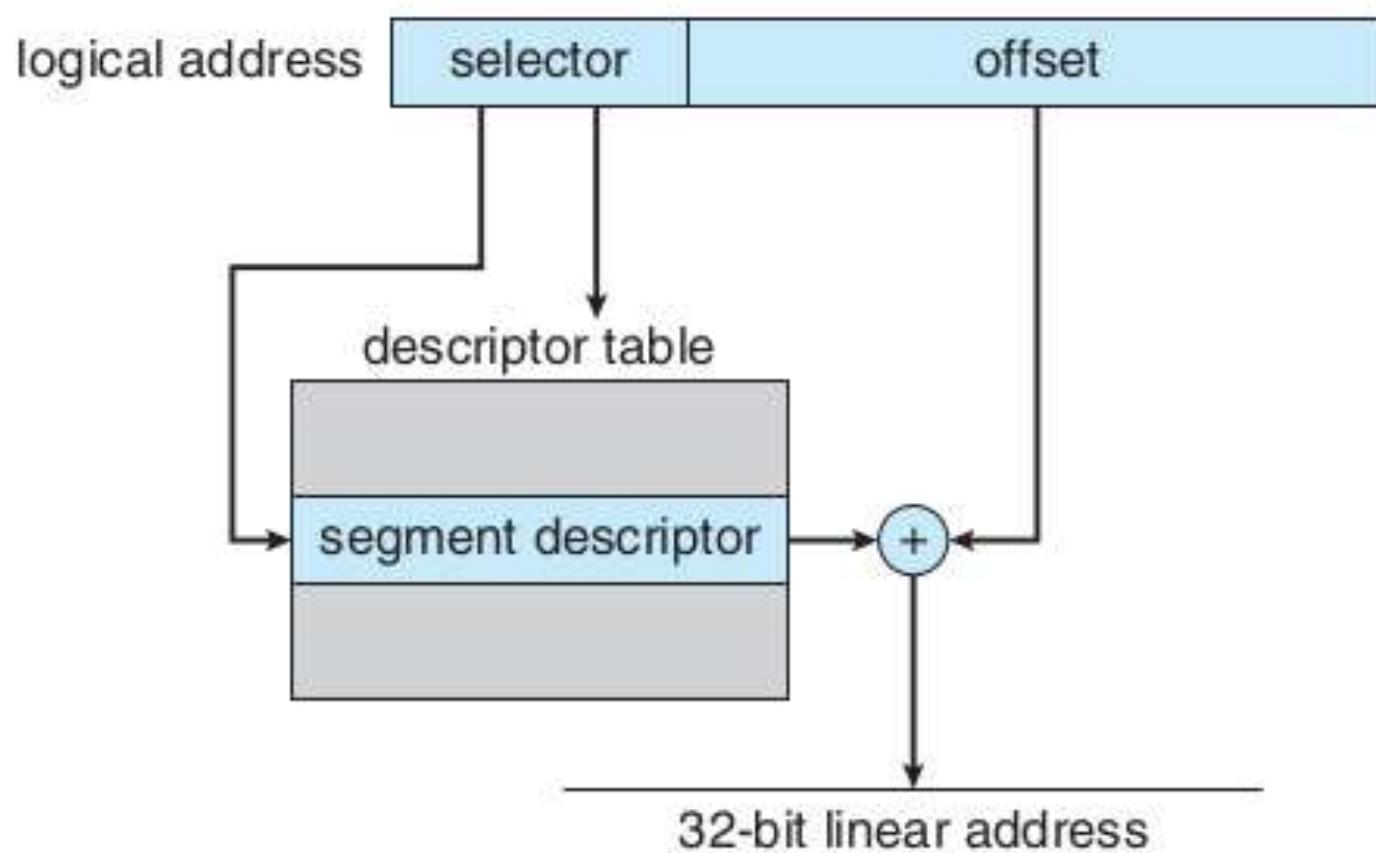


Figure 8.22 IA-32 segmentation.

X86 paging

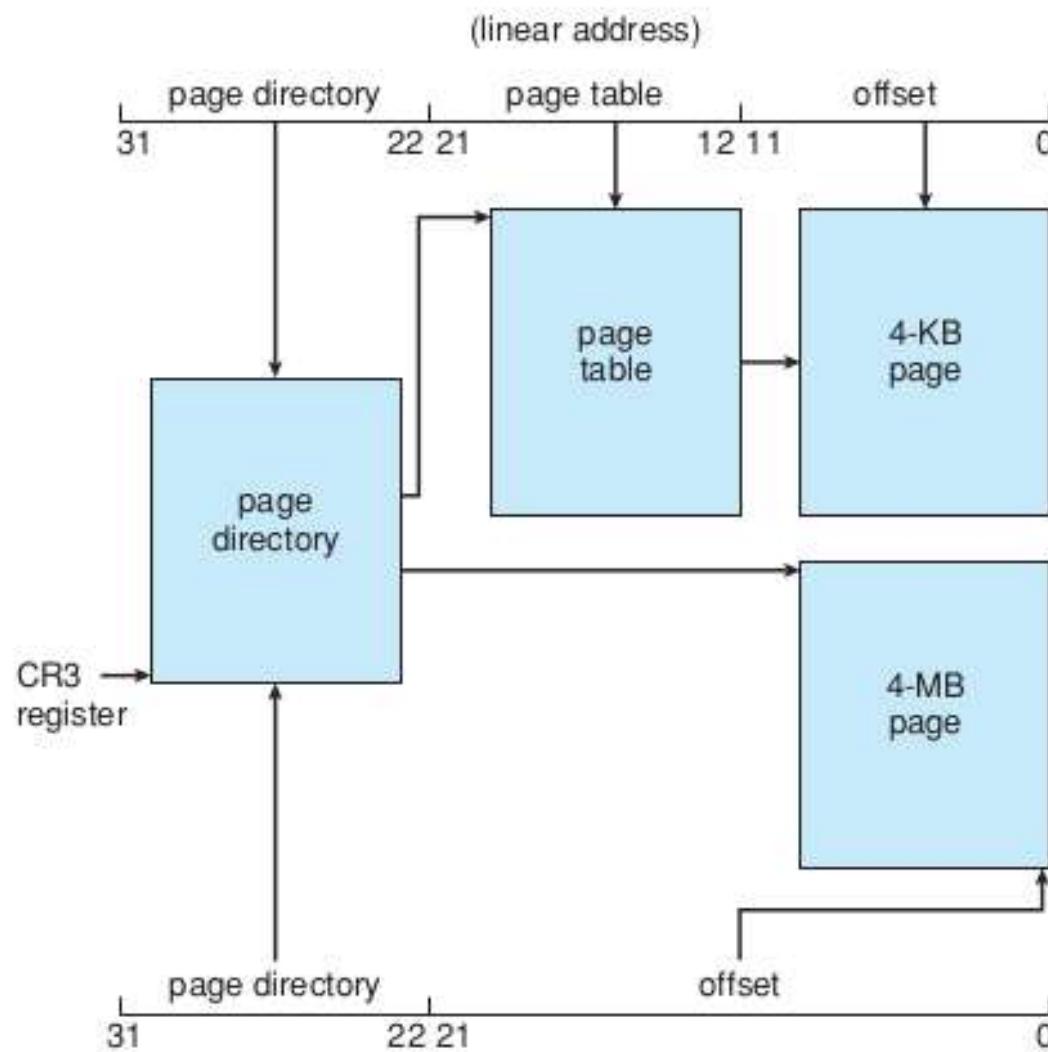
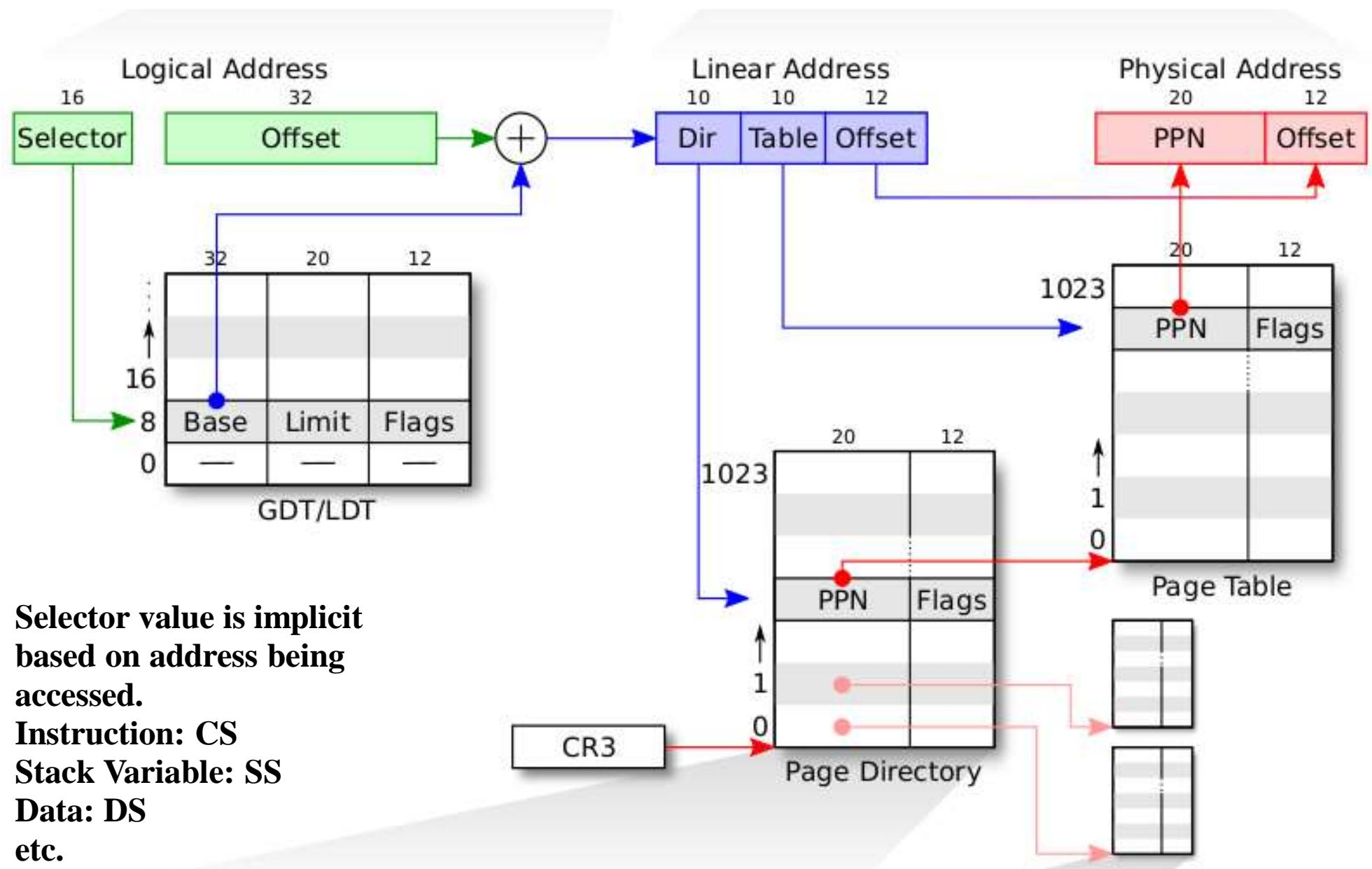
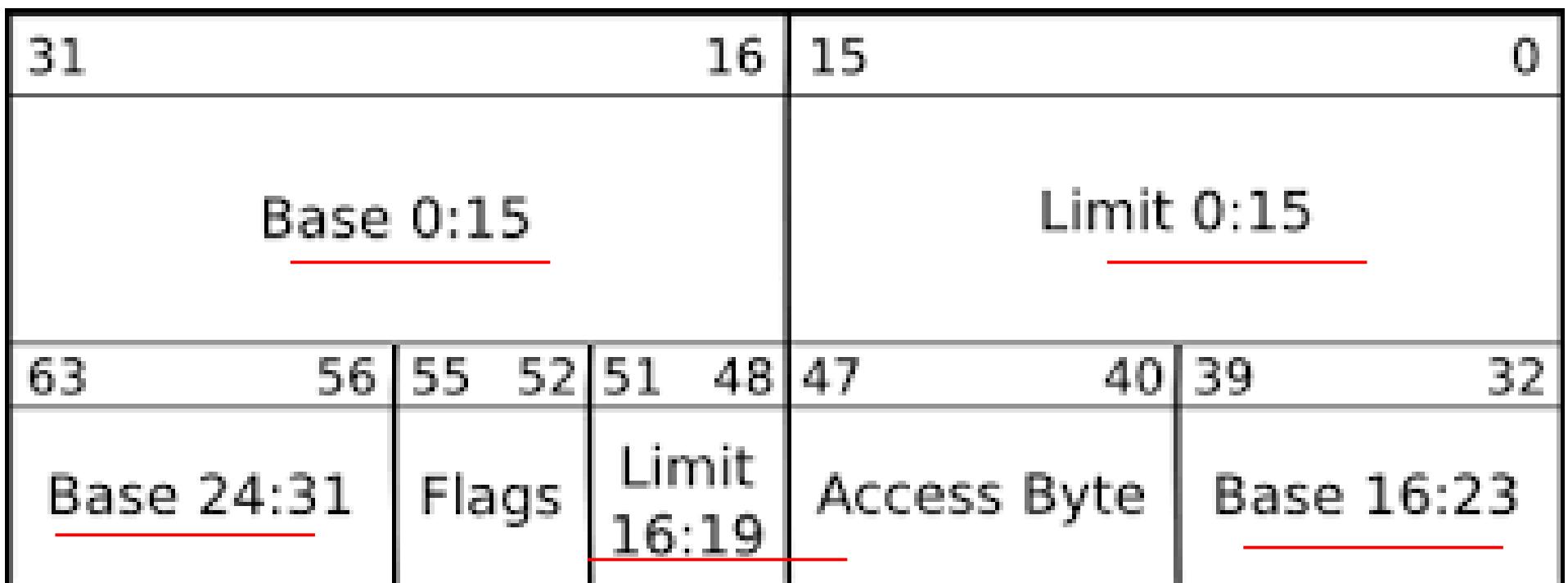


Figure 8.23 Paging in the IA-32 architecture.

Segmentation + Paging



GDT Entry

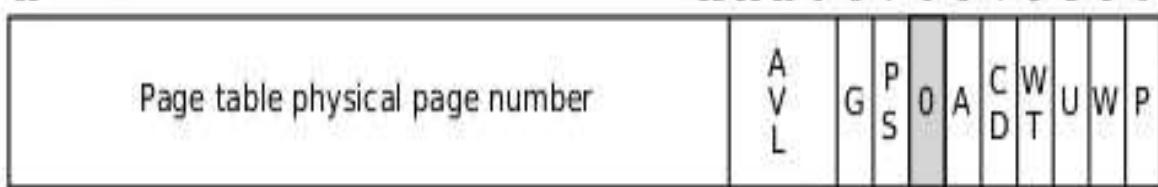


This is an in Memory entry. The diagram shows the format.

Page Directory Entry (PDE)

Page Table Entry (PTE)

31



PDE

P Present

W Writable

U User

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

CD Cache disabled

A Accessed

D Dirty

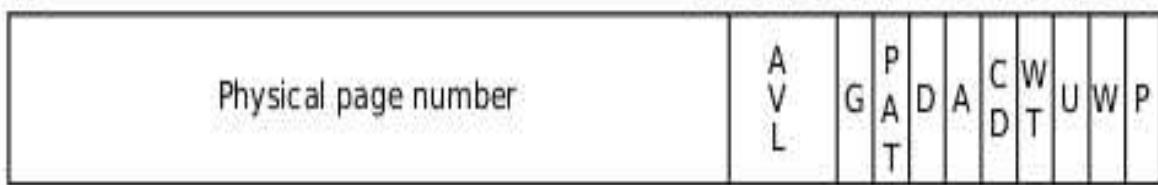
PS Page size (0=4KB, 1=4MB)

PAT Page table attribute index

G Global page

AVL Available for system use

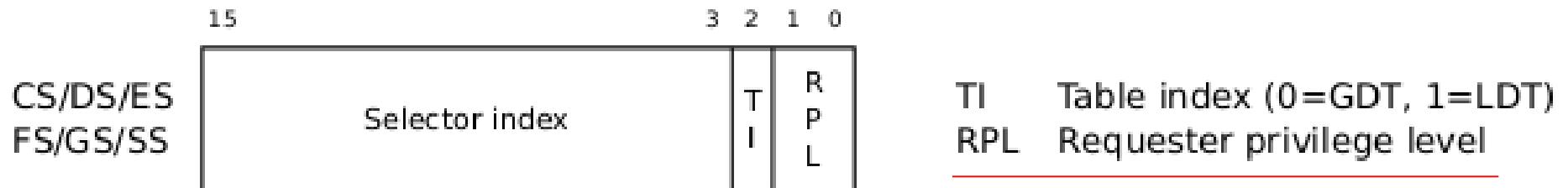
31



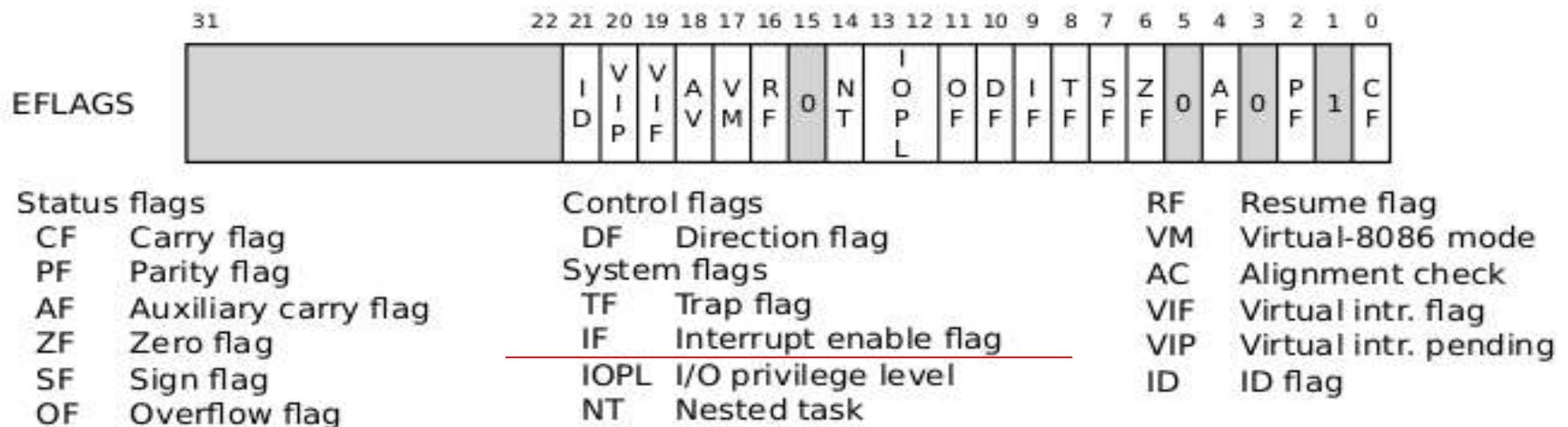
PTE

These are in memory entries. The diagrams show the format of each entry.

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

	31	12 11	5 4 3 2	0
	Page-Directory-Table Base Address		P C D	P W T

PWT Page-level writes transparent

PCT Page-level cache disable

CR4

CR4

31	11	10	9	8	7	6	5	4	3	2	1	0
	O S X M	O S F X	P C G E	P G C E	M A S E	P A S E	P S E	D T S D	T P V I	P V M E		

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

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 [Module 1](#))

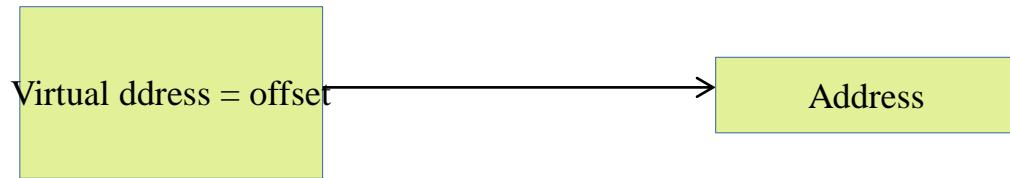
bootloader

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

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.

addr = seg << 4 + 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:

- Seg:off with 16 bit segments can actually address more than 20 bits of memory. After 0x100000 ($=2^{20}$), 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

After this

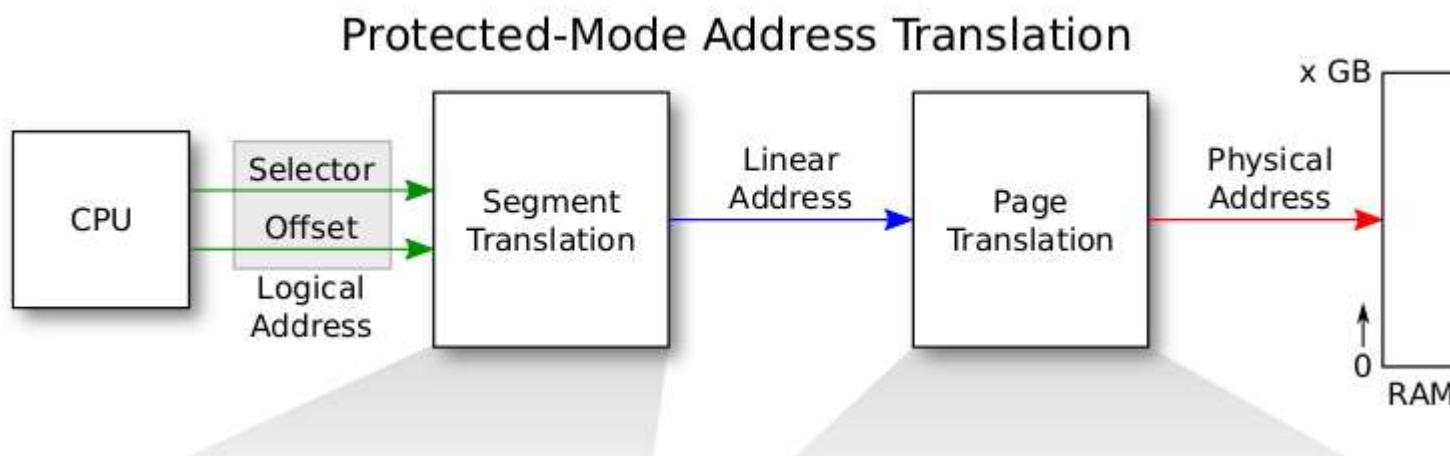
**Some instructions are run
to enter protected mode**

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

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

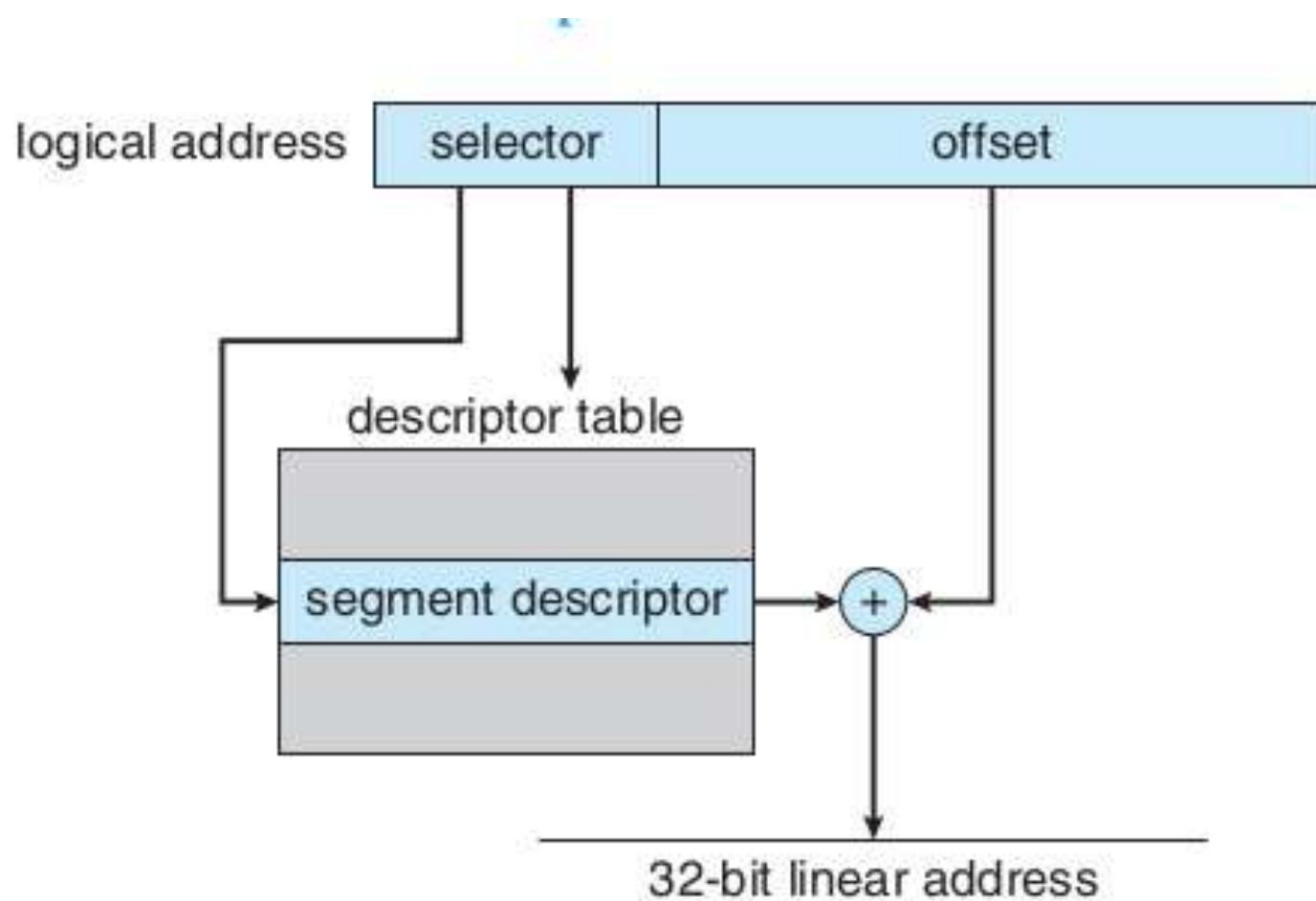


Figure 8.22 IA-32 segmentation.

Paging concept, hierarchical paging

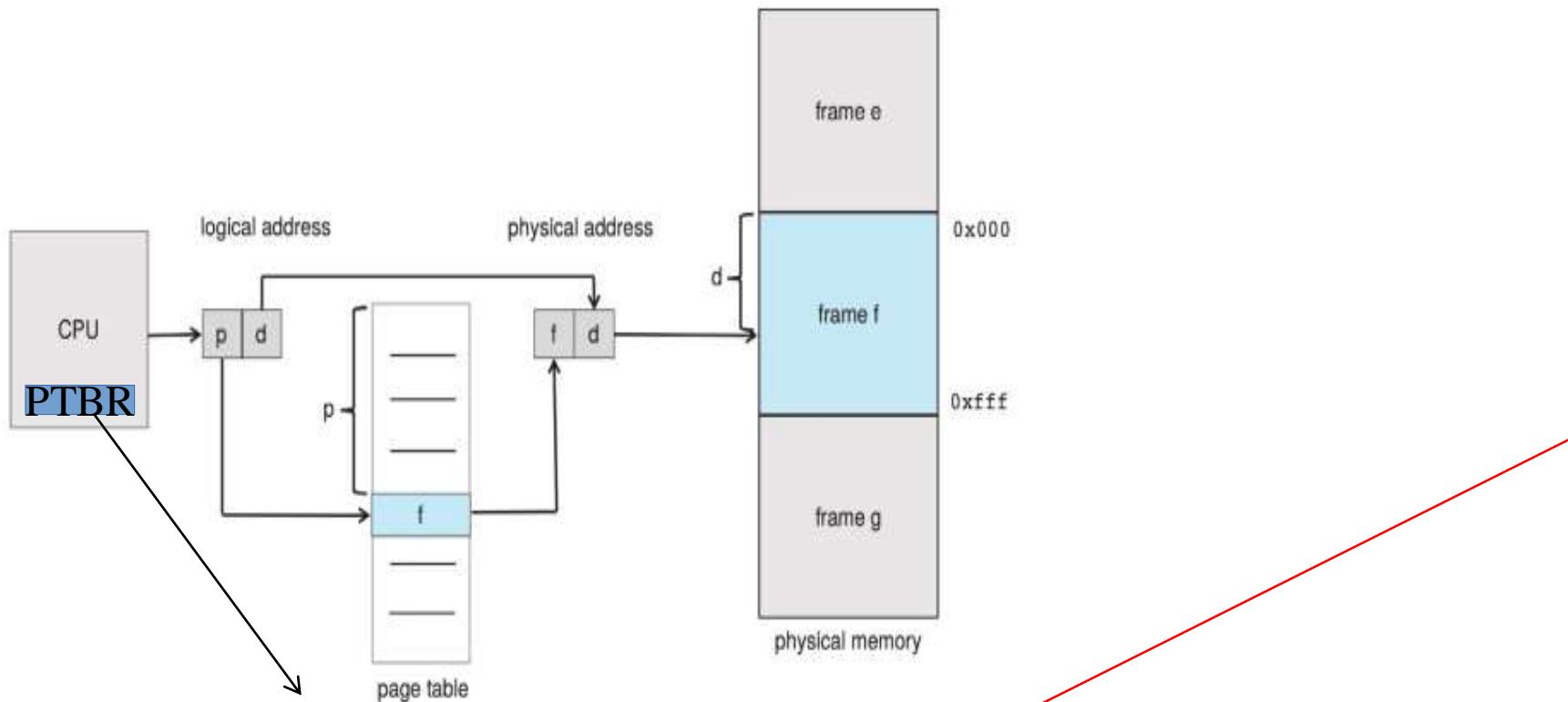


Figure 9.8 Paging hardware.

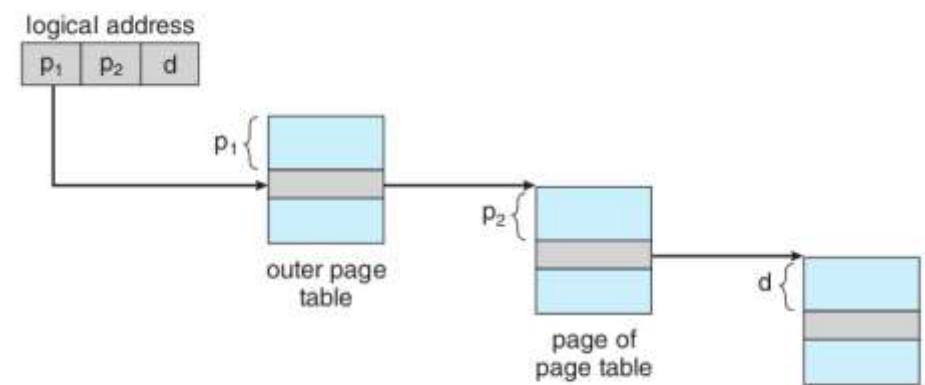


Figure 9.16 Address translation for a two-level 32-bit paging architecture.

X86 paging

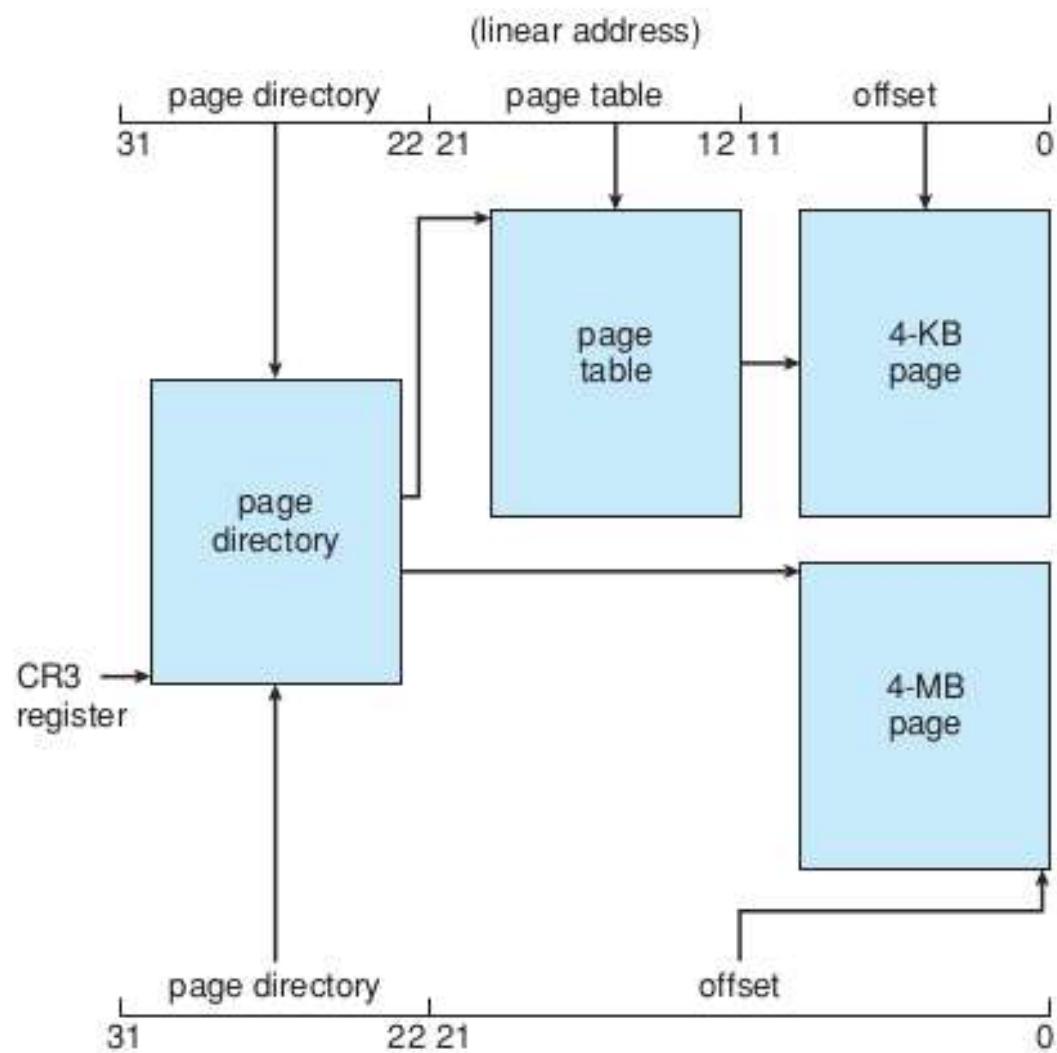


Figure 8.23 Paging in the IA-32 architecture.

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	G	P	S	0	A	C	W	D	T	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

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

PTE

CR3

CR3

	31	12 11	5 4 3 2	0
	Page-Directory-Table Base Address		P C D	P W T

PWT Page-level writes transparent

PCT Page-level cache disable

CR4

CR4

31	11	10	9	8	7	6	5	4	3	2	1	0
	O S X M	O S F X	P C G E	P G C E	M A S E	P A S E	P S E	D T S D	T P V I	P V M E		

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

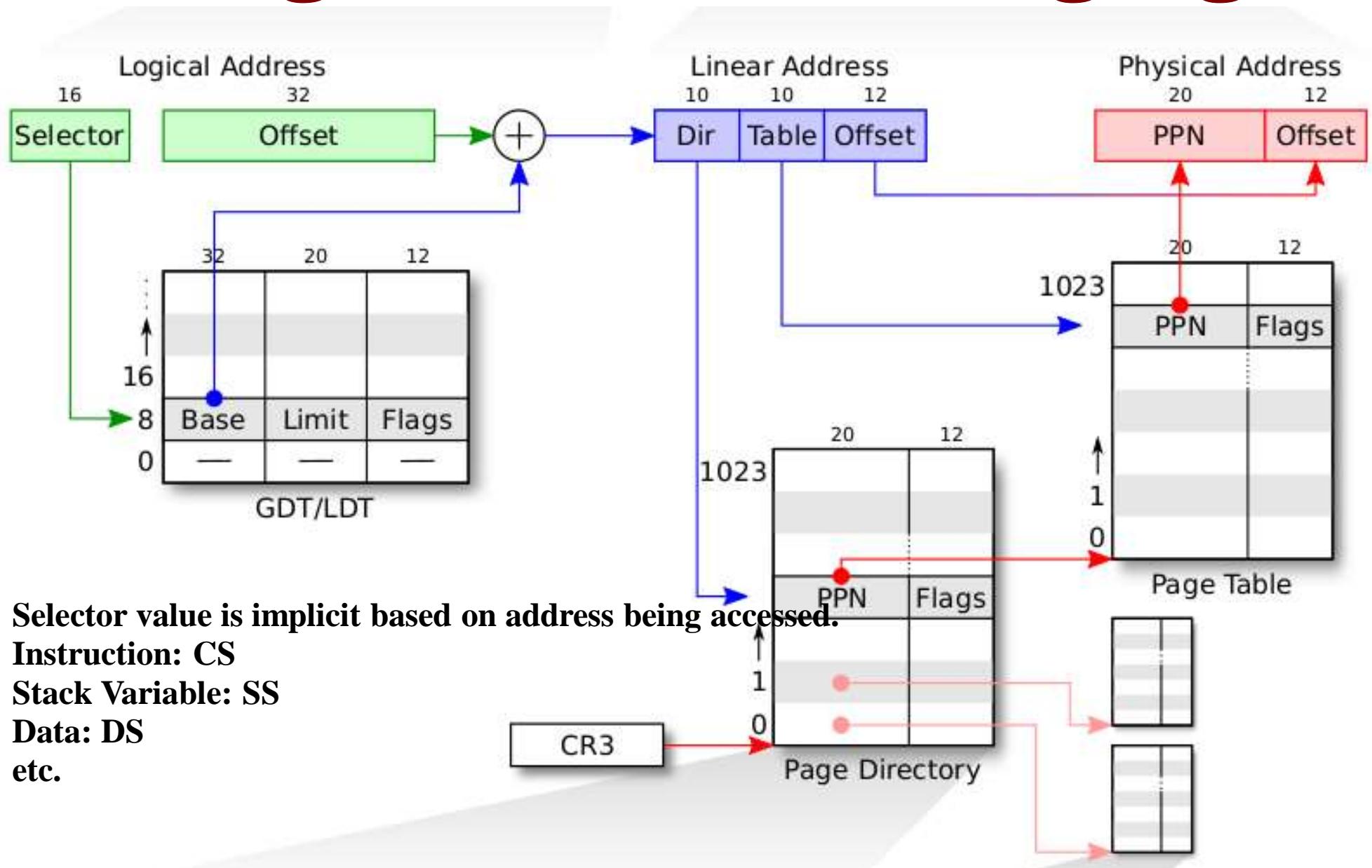
PCE Performance counter enable

OSFXSR OS FXSAVE/FXRSTOR support

OSXMM- OS unmasked exception support

EXCPT

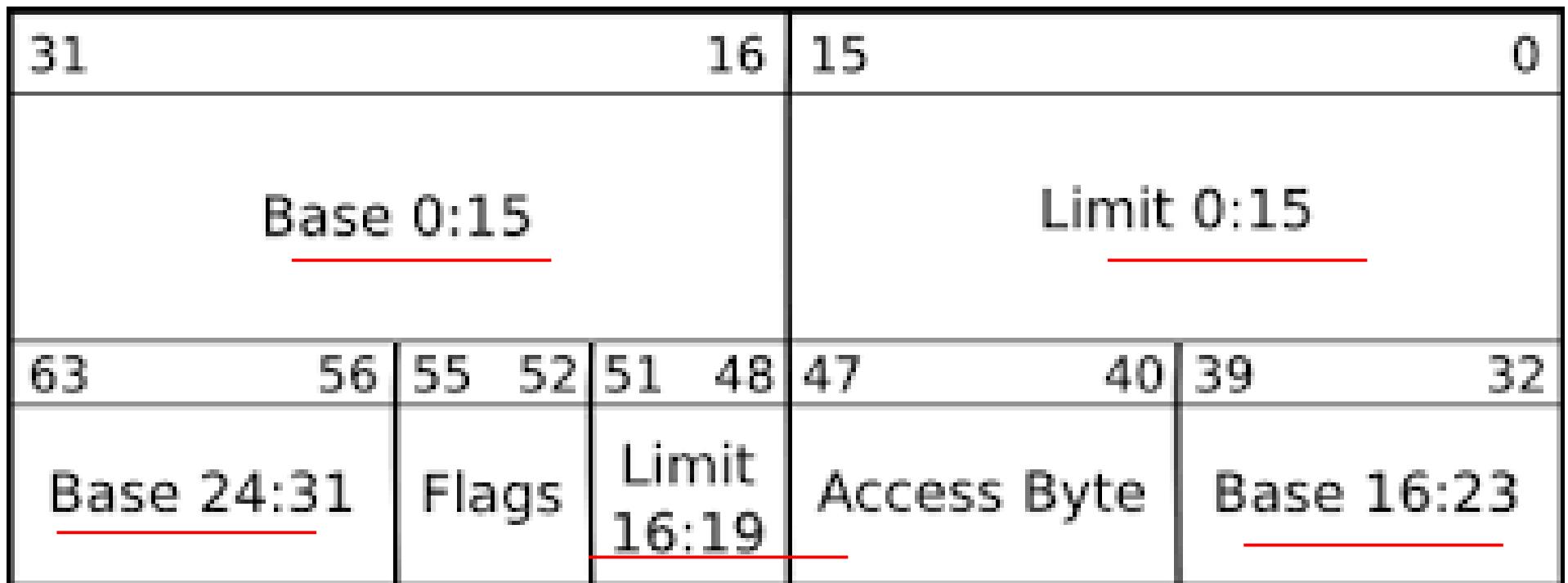
Segmentation + Paging



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

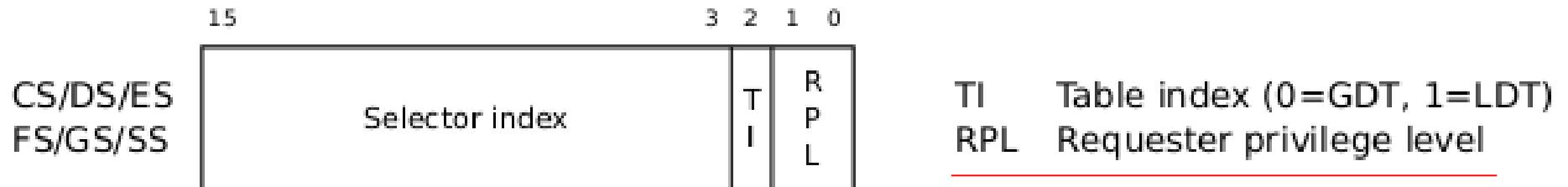
GDT Entry



asm.h

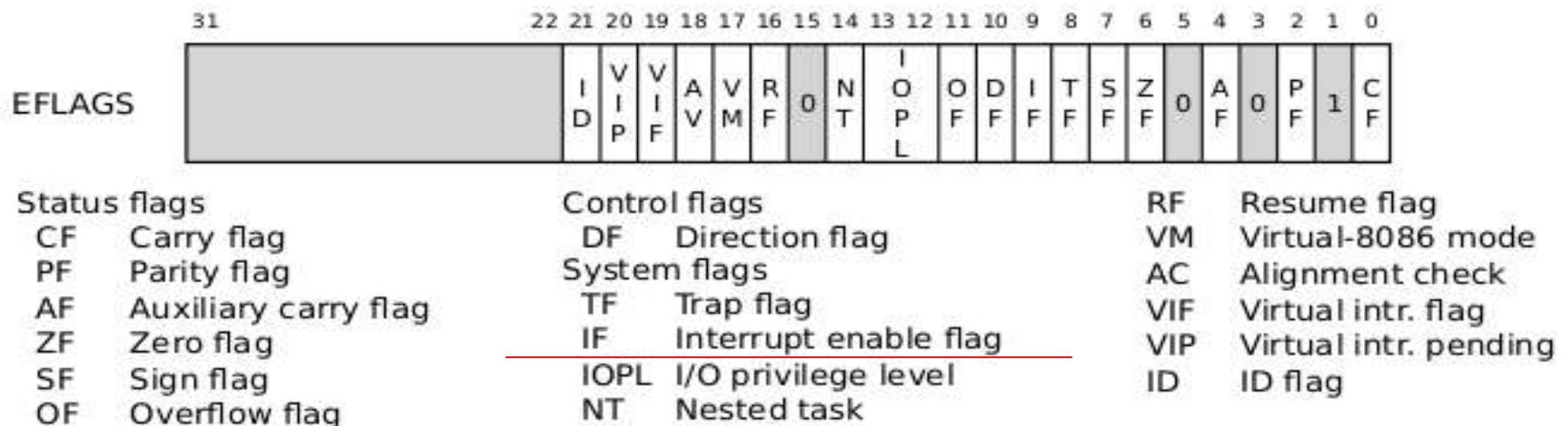
```
#define SEG_ASM(type,base,lim) \
    .word (((lim) >> 12) & 0xffff), ((base) & 0xffff); \
    .byte (((base) >> 16) & 0xff), (0x90 | (type)), \
          (0xC0 | (((lim) >> 28) & 0xf)), (((base) >> 24) & 0xff)
```

Segment selector



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

EFLAGS register



lgdt gdtdesc

...

Bootstrap GDT

.p2align 2 # force 4 byte alignment

gdt:

SEG_NULLASM #
null seg

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

lgdt

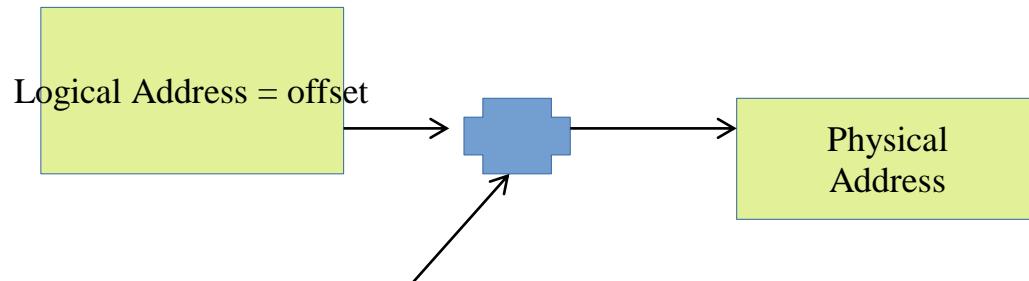
load the processor's (GDT) register with the value gdtdesc which points to the table gdt.

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

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



Base	Limit	Permissions
0	4GB	Write
0	4GB	Read, Execute
0	0	0

GDT

Still
Logical Address = Physical address

But with GDT in picture and
Protected Mode operation

During this time,

Loading kernel from ELF into p

Addresses in “kernel” file trans

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

	31 30 29 28	19 18 17 16 15	6 5 4 3 2 1 0
CR0	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
			NW Not write-through
			CD Cache disable
			PG Paging

PG: Paging enabled or not

WP: Write protection on/off

PE: Protection Enabled --> protected mode.

Complete transition to 32 bit mode

```
ljmp $(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 ljmp.

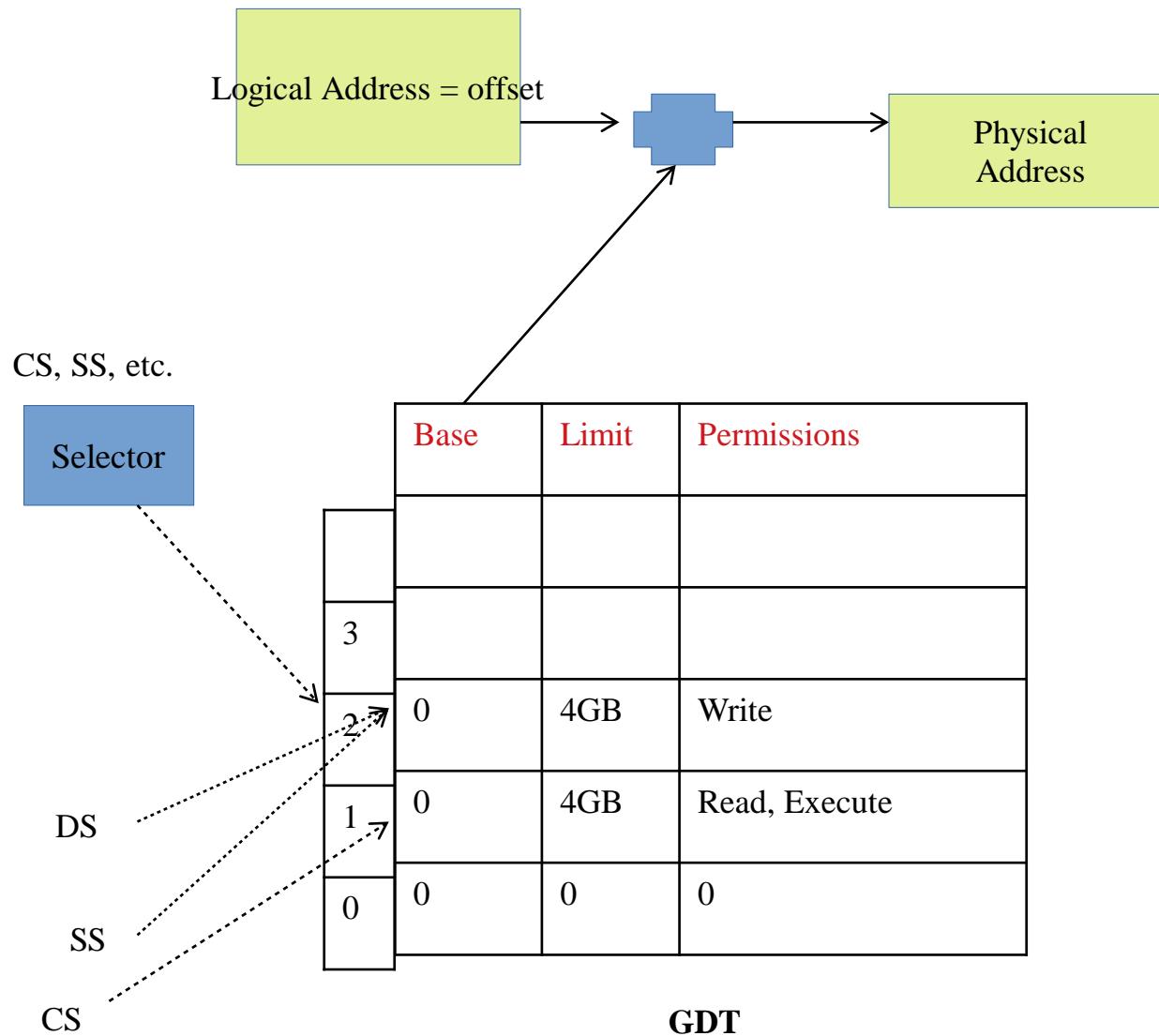
Note: The segment descriptors are set up with no translation (that

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
```

- 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

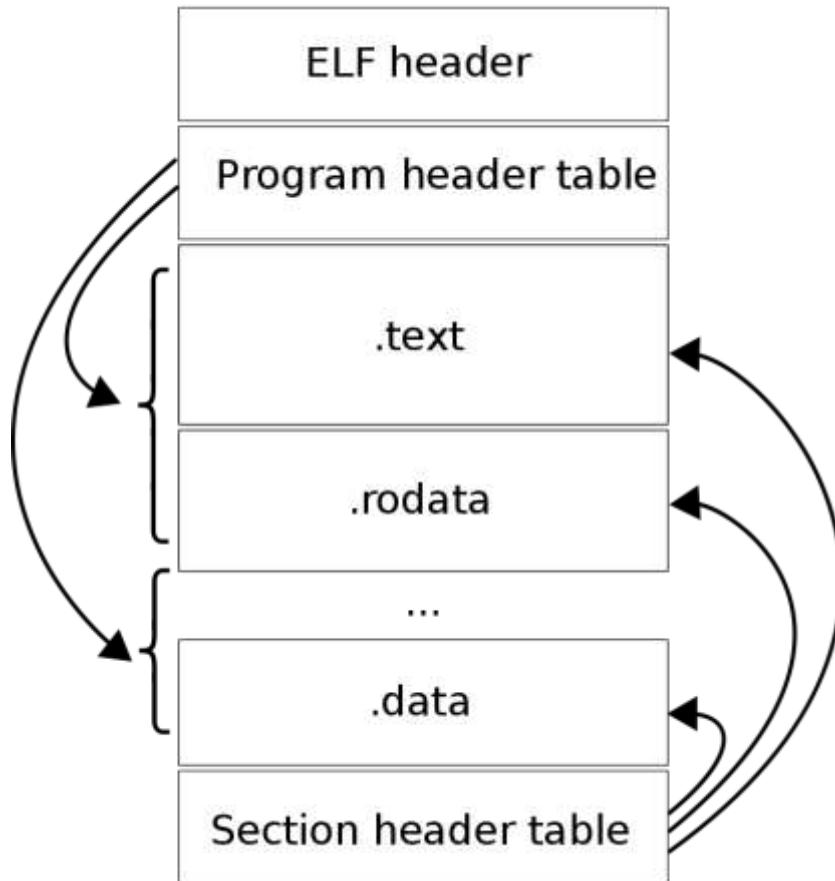
- ❑ Bootmain loads the first 4096 bytes of the ELF binary. It places the in-

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

bootmain()

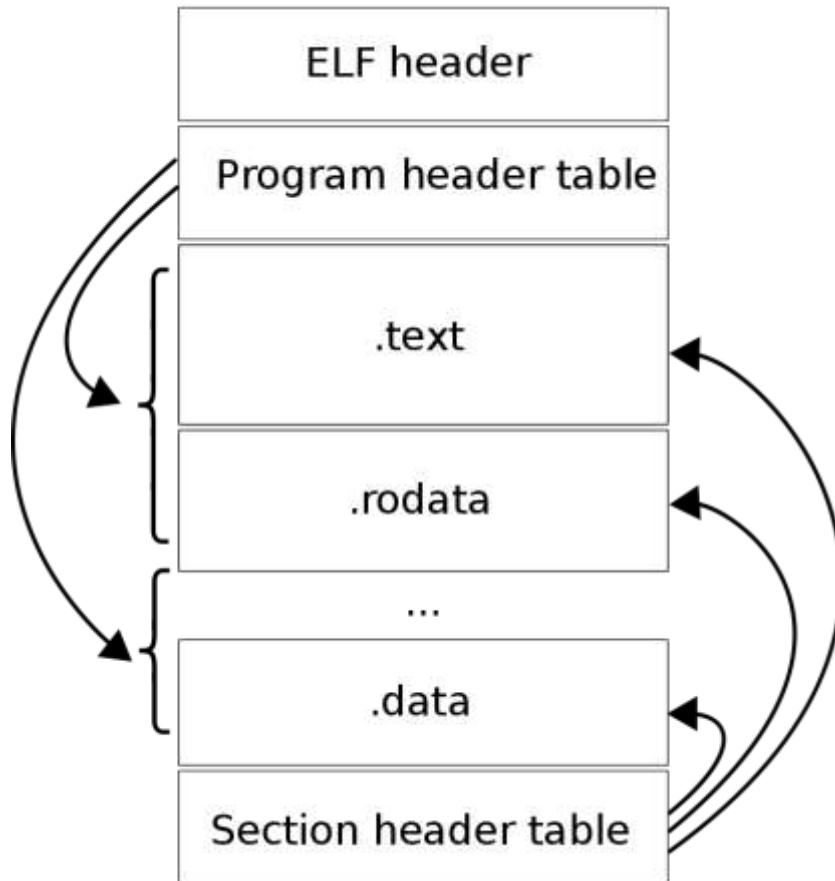
- Check if it's really ELF or not
 - // Is this an ELF executable?
- Next load kernel code from ELF file “kernel” into memory
 - if(elf->magic != ELF_MAGIC)
return; // let
bootasm.S handle error

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;
```

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.
```

kernel: Run ‘objdump -x -a kernel | head -15’ & see this
file format elf32-i386

kernel

architecture: i386, flags 0x00000112:

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

Stack : everything zeroes

Diff between mem
Code to be loaded at KERNBASE + KE

```
// Load each program segment (ignores ph flags).  
Load code from ELF to memory  
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).
```

To understand
further
code

Remember: 4
MB pages
are possible

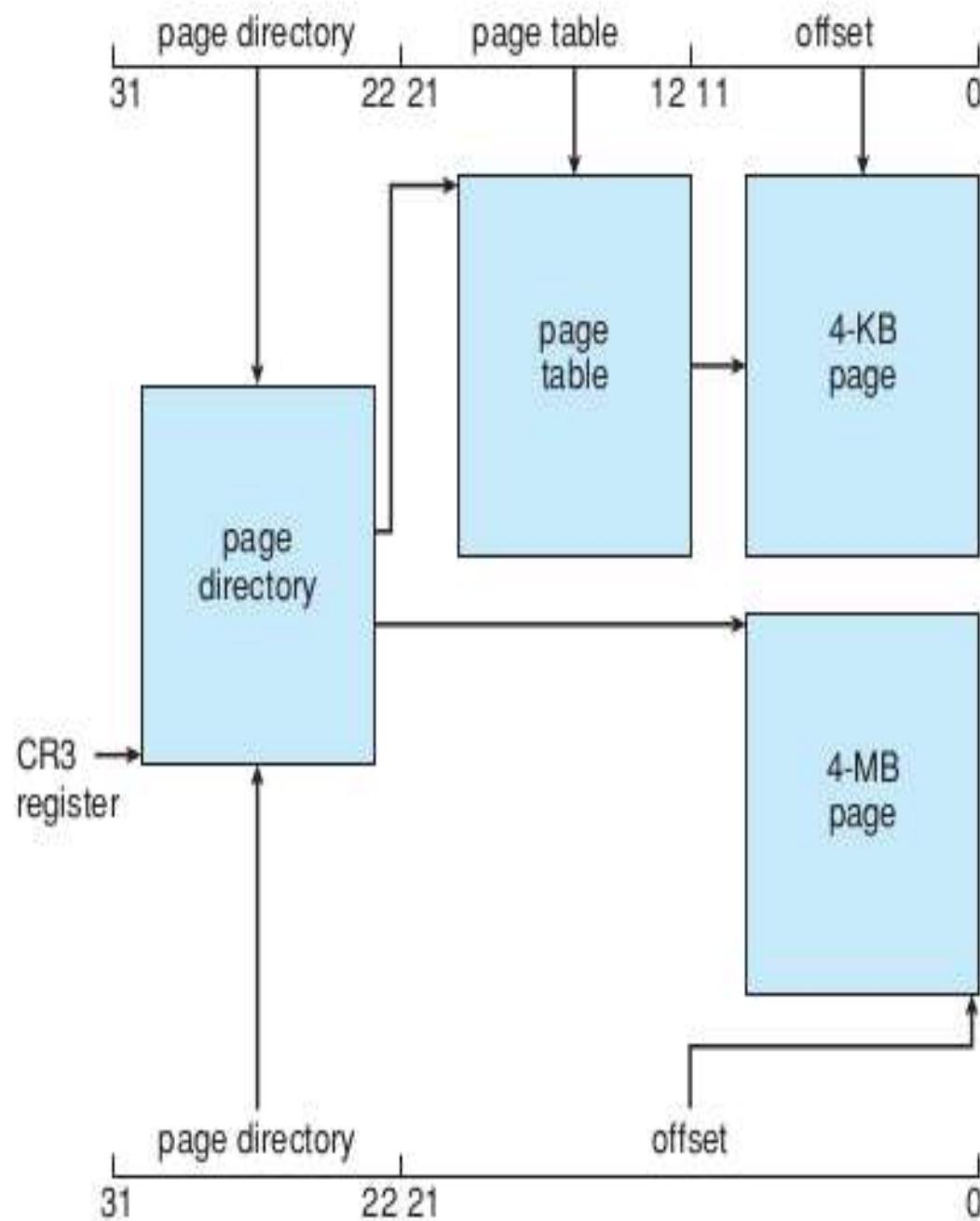
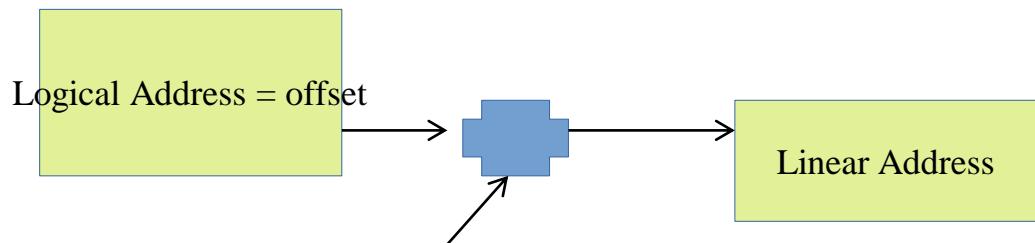


Figure 8.23 Paging in the IA-32 architecture.

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

RAM



CS, SS, etc.

Selector

	Base	Limit	Permissions
3			
2	0	4GB	Write
1	0	4GB	Read, Execute
0	0	0	0

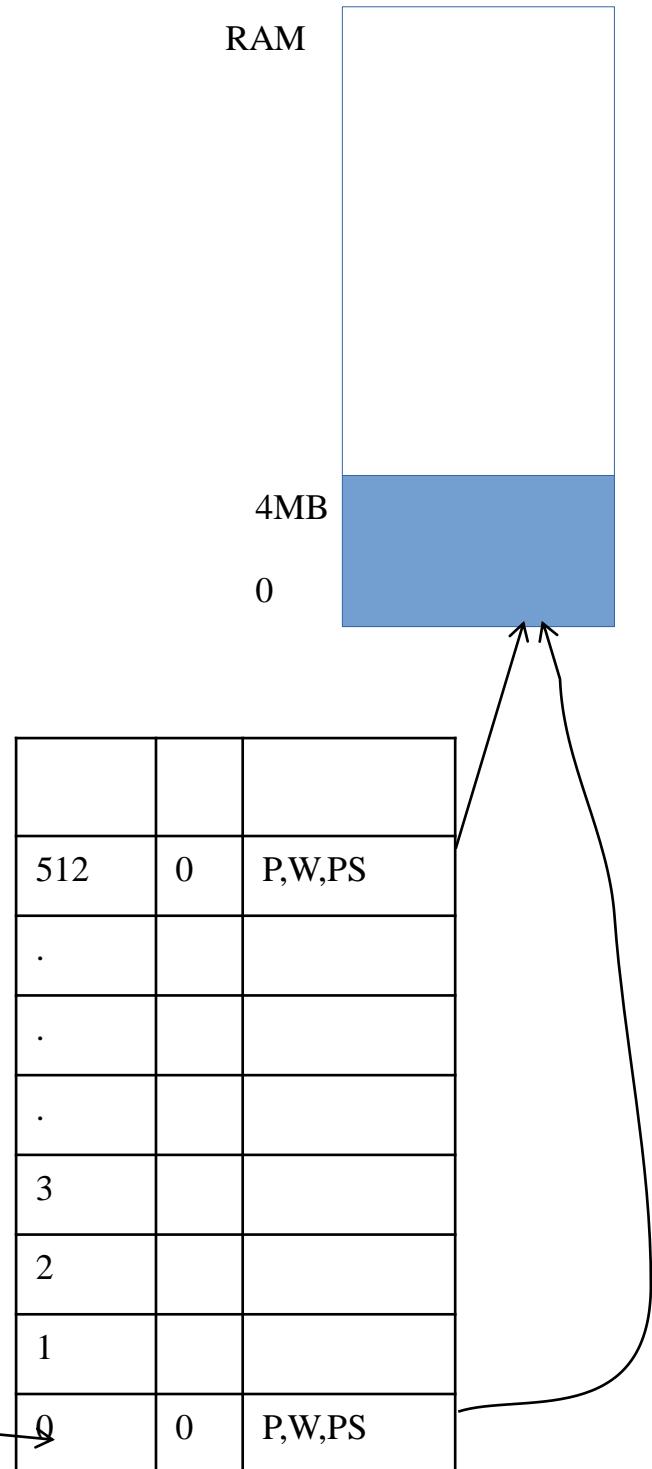
GDT

DS

SS

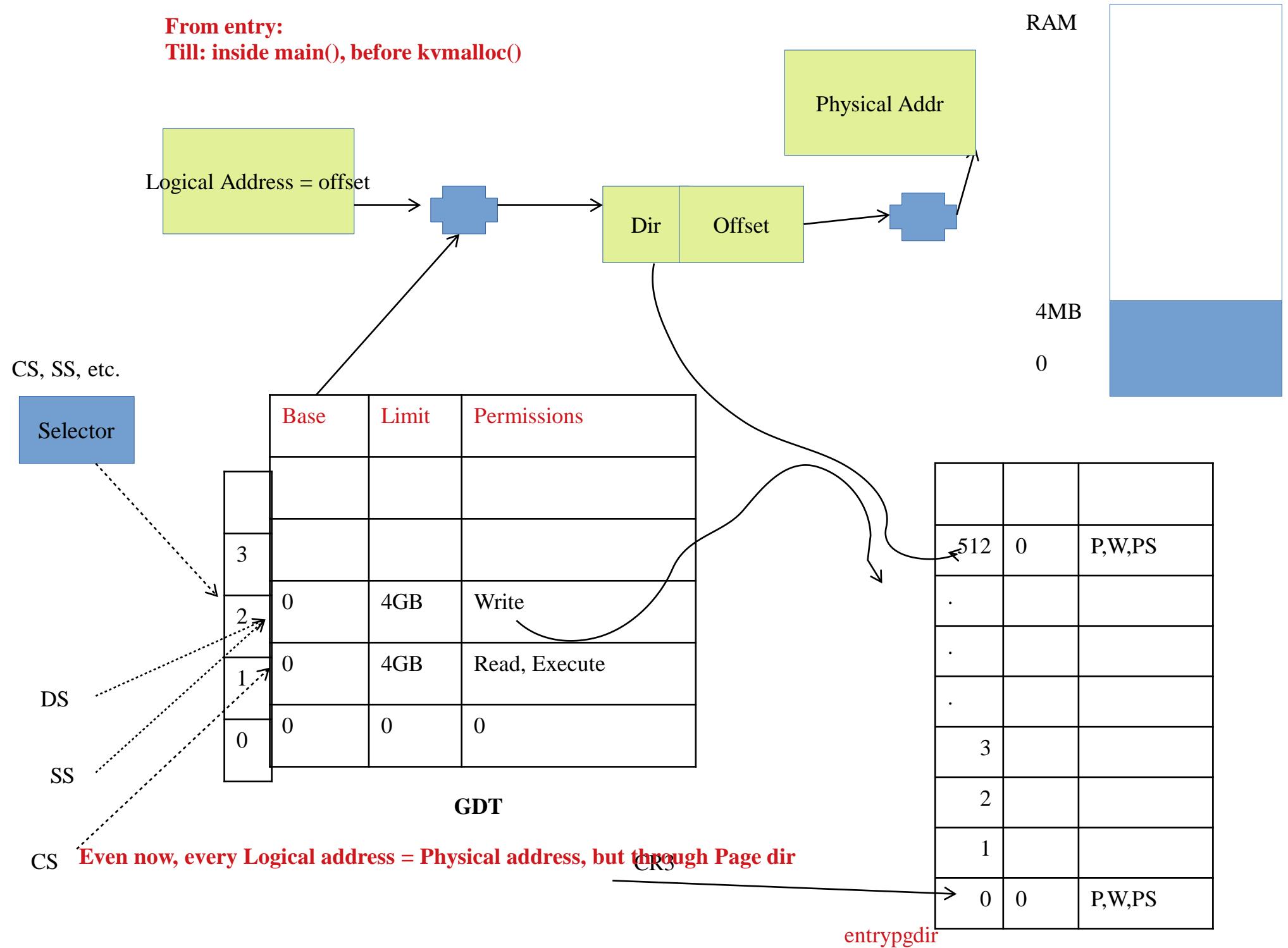
CS

CR3 → entrypgdir



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,  
  
};
```

#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 i

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.

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
```

- # 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

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

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 in xv6 code

Process Table

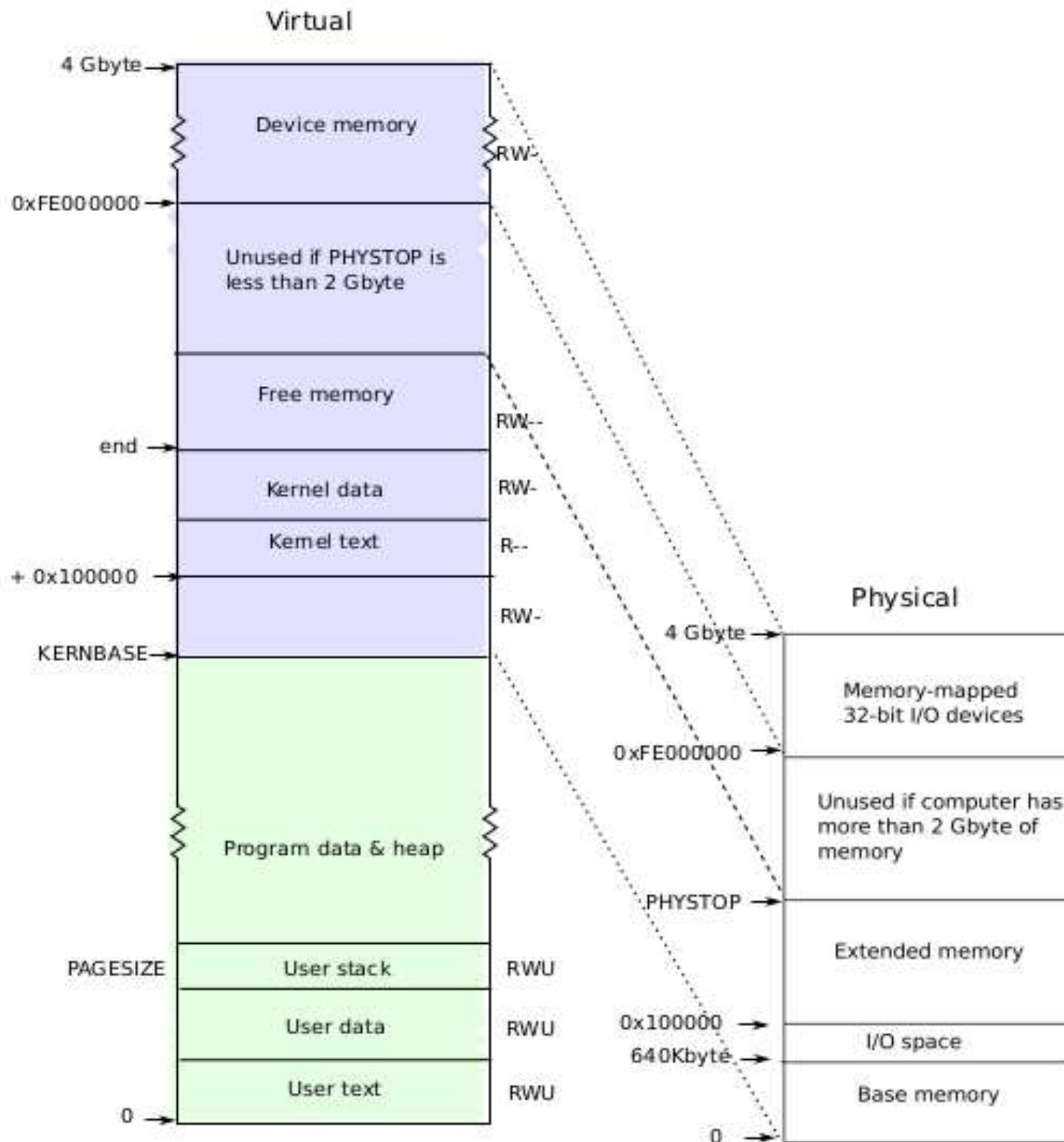
```
struct {  
  
    struct spinlock lock;  
  
    struct proc proc[NPROC];  
  
} ptable;
```

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

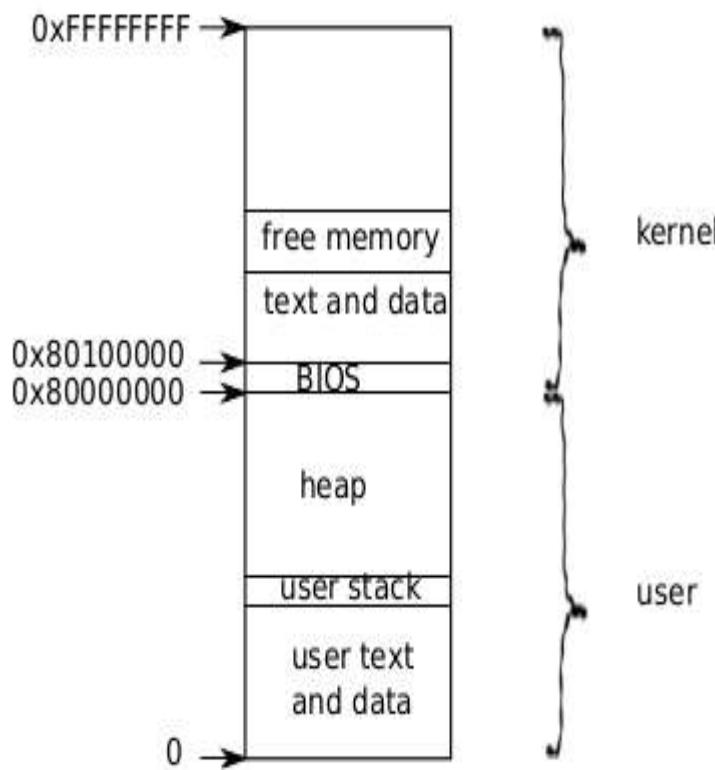
Layout of process's VA space

kv6 schema!

different from Linux

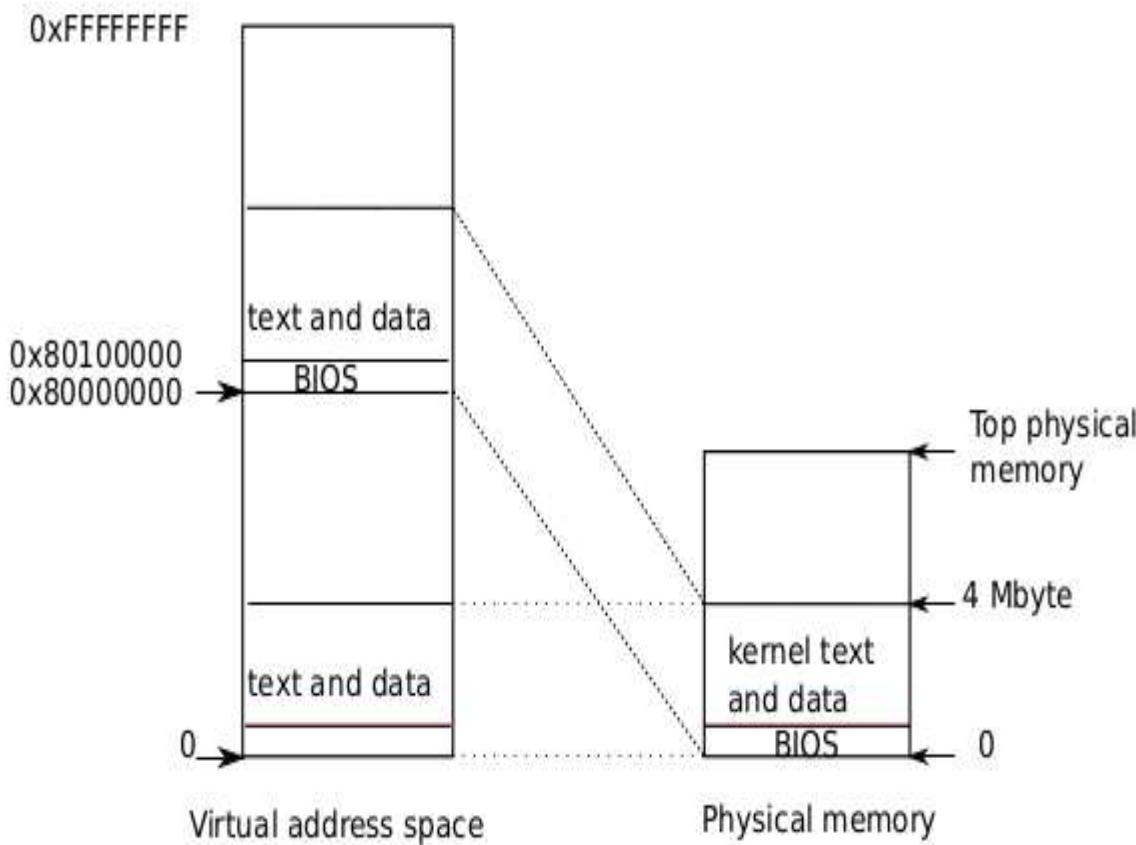


Logical layout of memory for a process



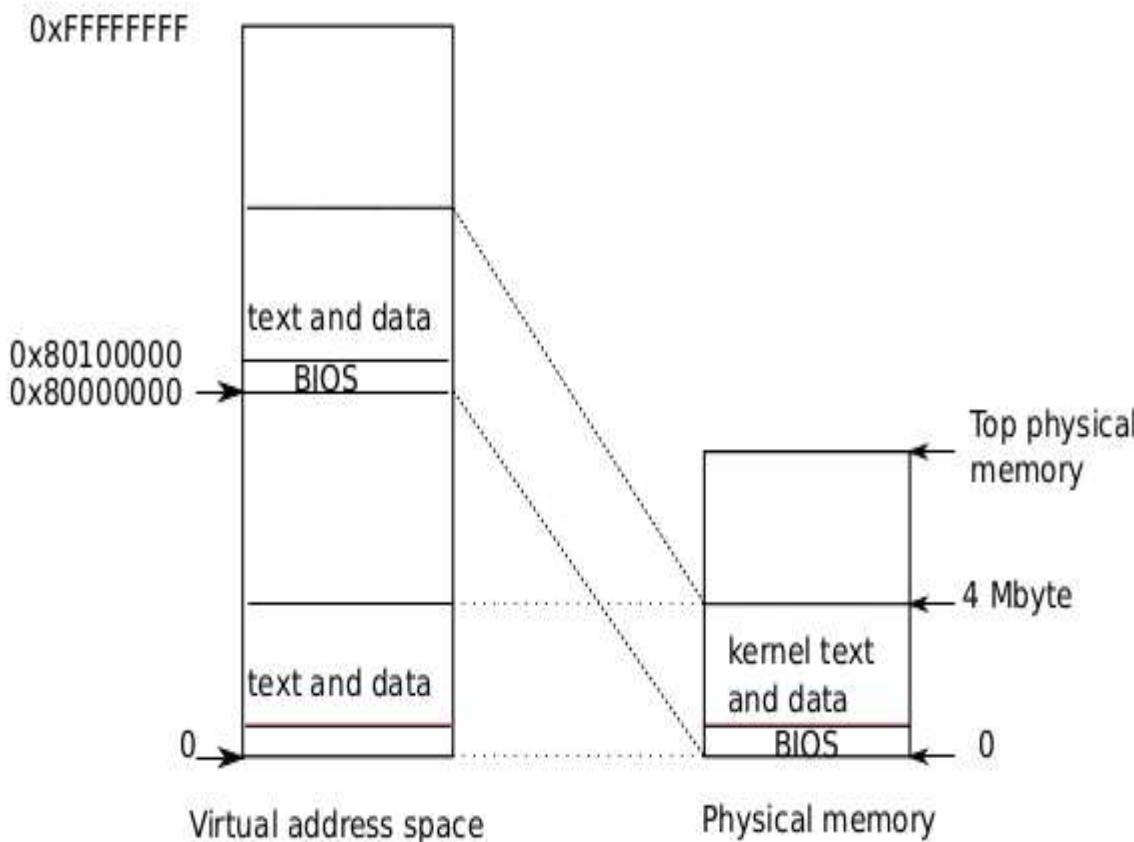
- Address 0: code
- Then globals
- Then stack
- Then heap
- Each process's address space maps kernel's text,

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

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

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

Struct proc

// Per-process state

struct proc {

uint sz;

// Size of process

memory (bytes)

pde_t* pgdir;

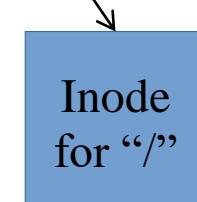
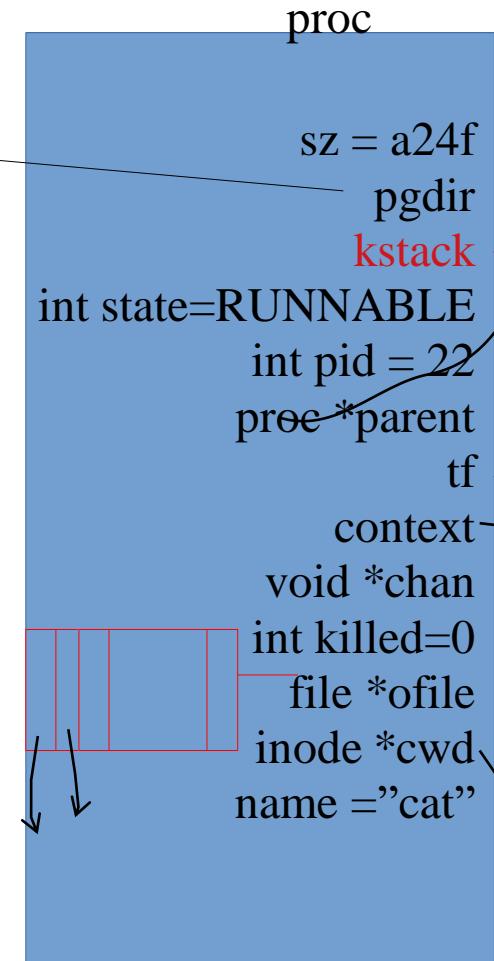
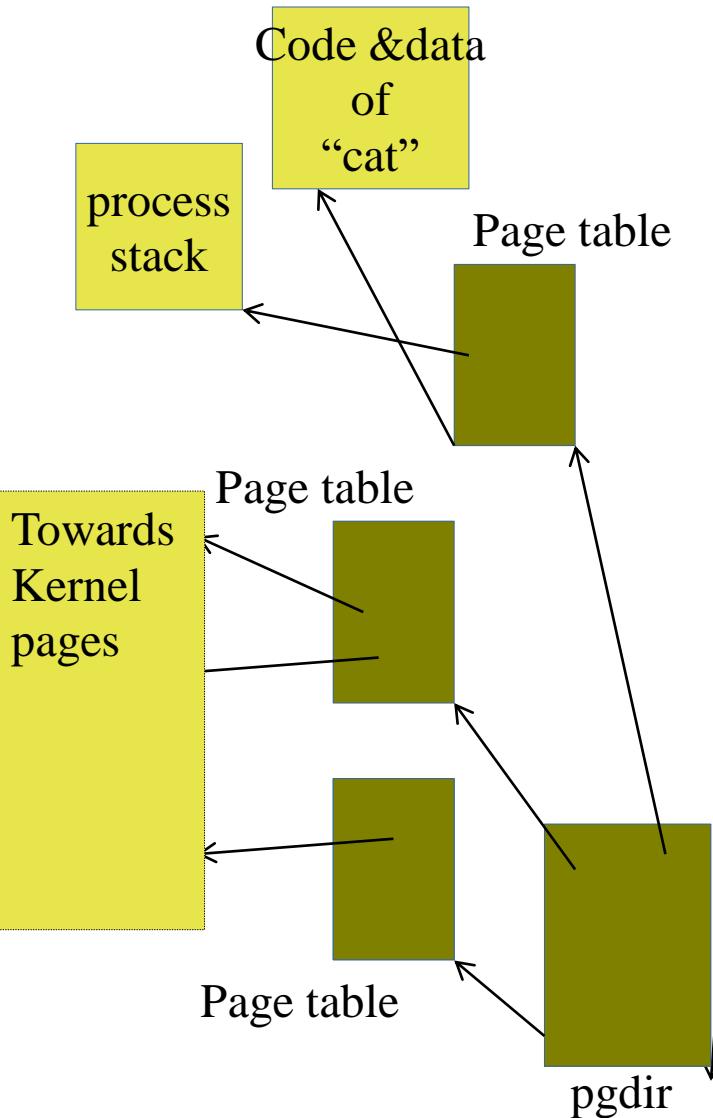
// Page table

```
char *kstack; // Be  
kernel stack for this process
```

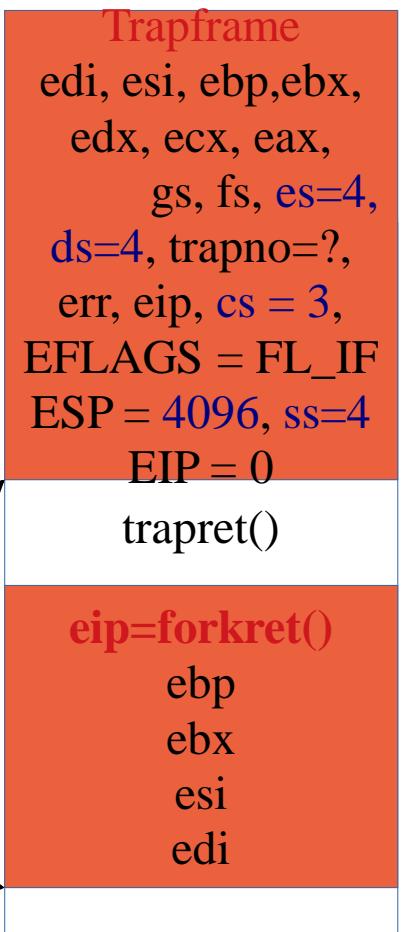
```
enum procstate state;
```

allocated, ready to run, running, wait-

struct proc diagram: Very imp!

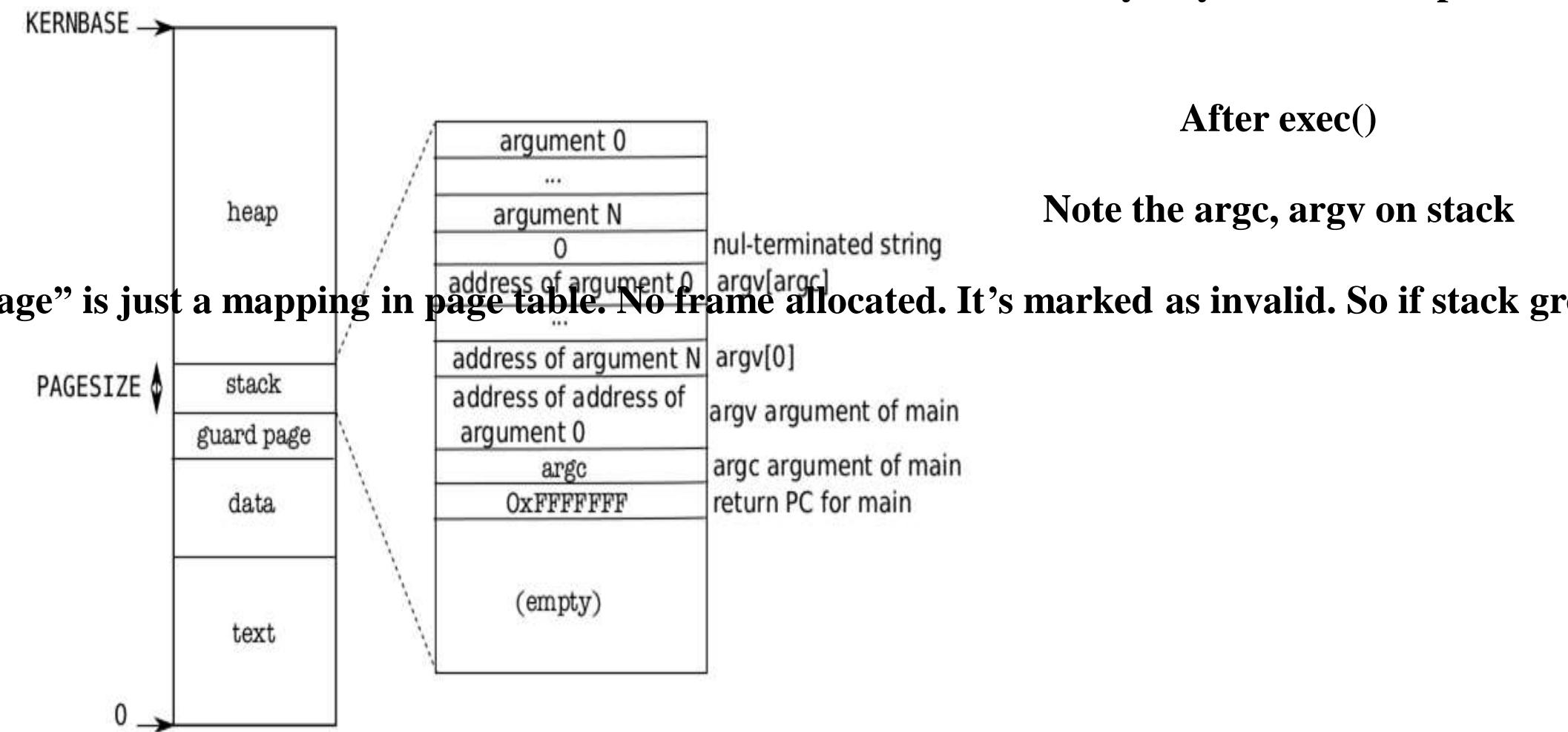


`sz = ELF-code->memsz` (includes data, check "ld -N")
 $+ 2 \times 4096$ (for stack)



Memory Layout of a user process

Memory Layout of a user process

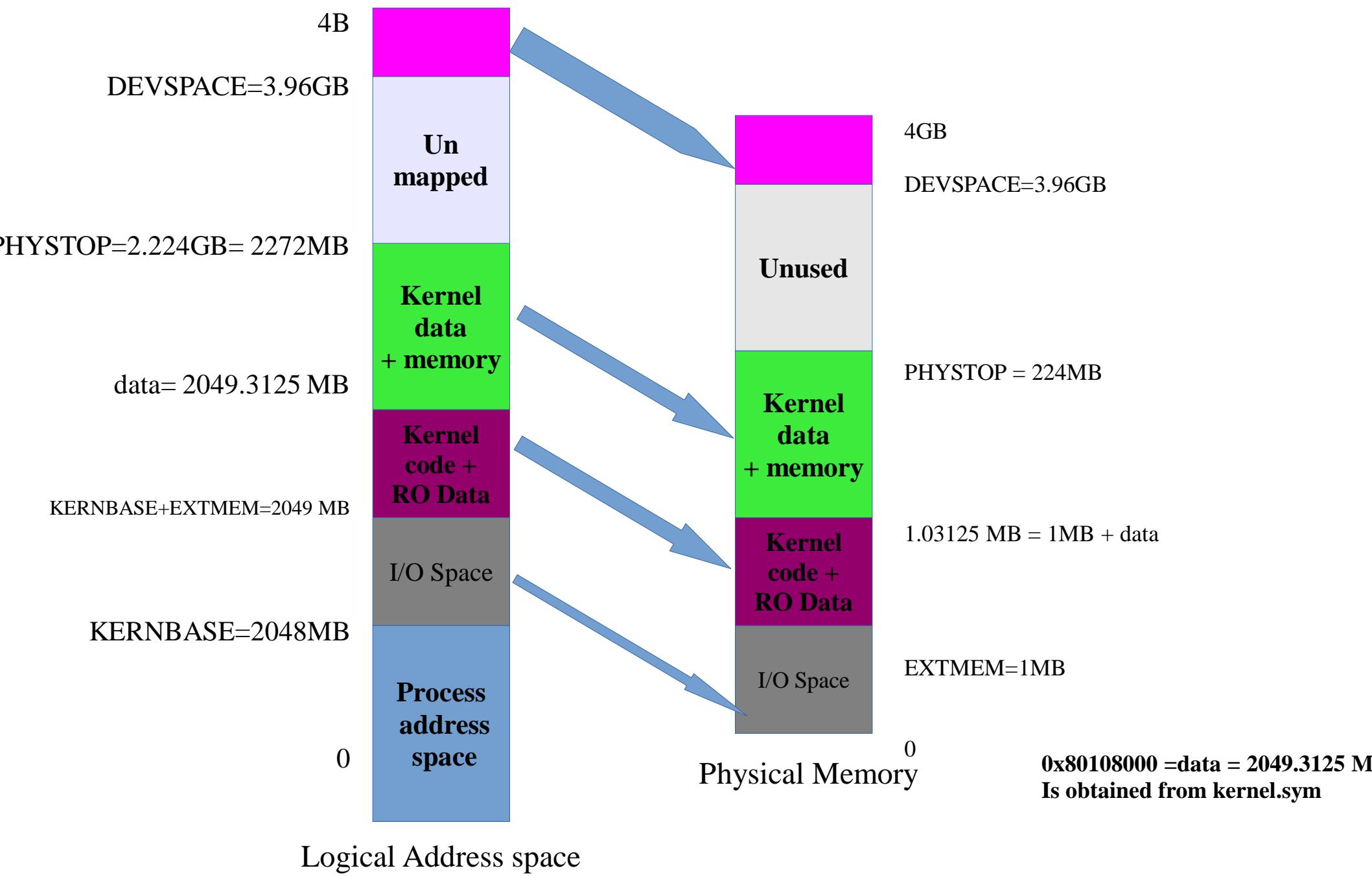


Handling Traps

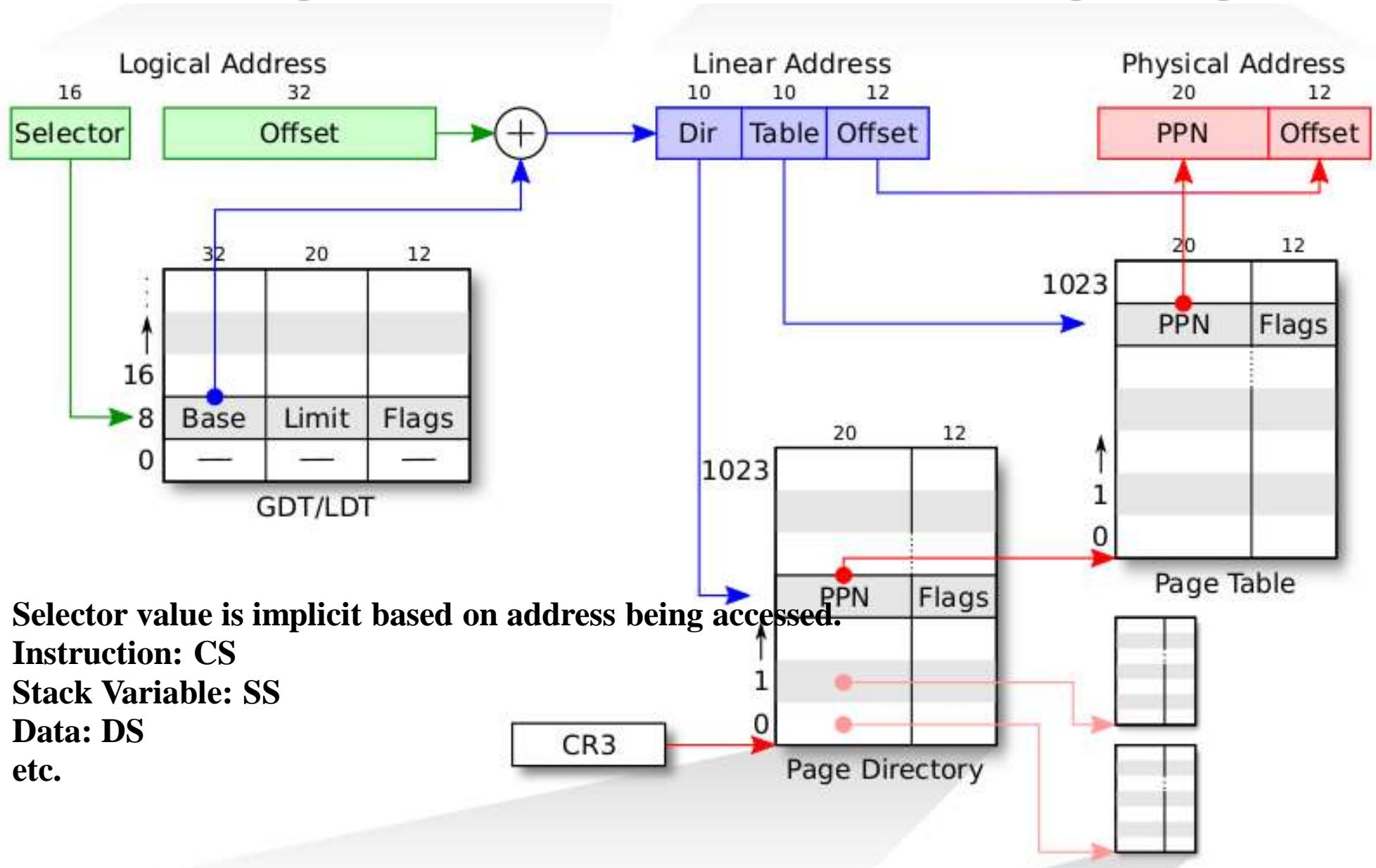
Some basic steps

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

kmap[] mappings done in kvmalloc(). This shows segmentwise, entries are done in page directory and page table for corre

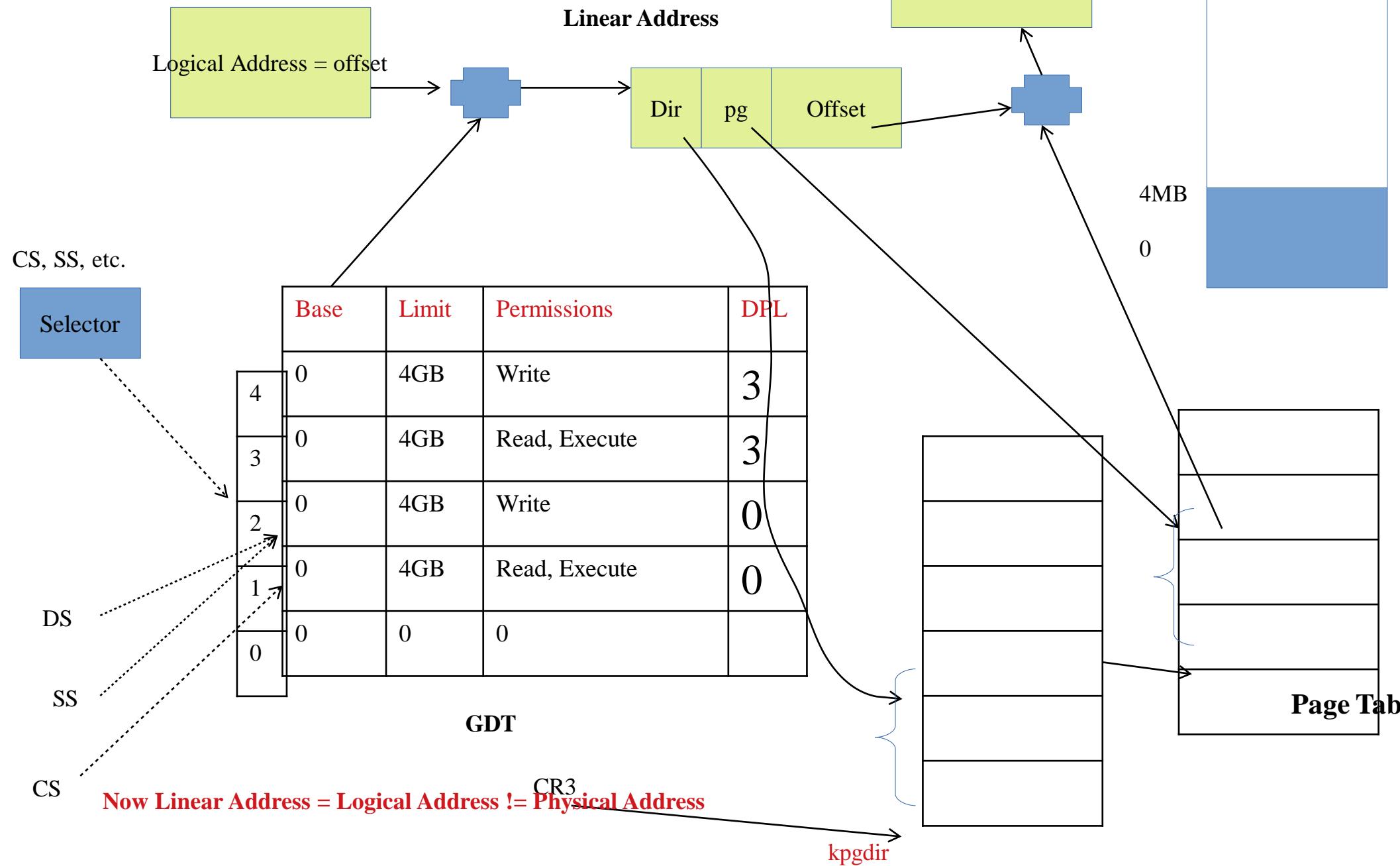


Segmentation + Paging



After seginit() in main().

On the processor where we started booting



Handling traps

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

Handling traps

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

Privilege level

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



TI Table index (0=GDT, 1=LDT)
RPL Requester privilege level

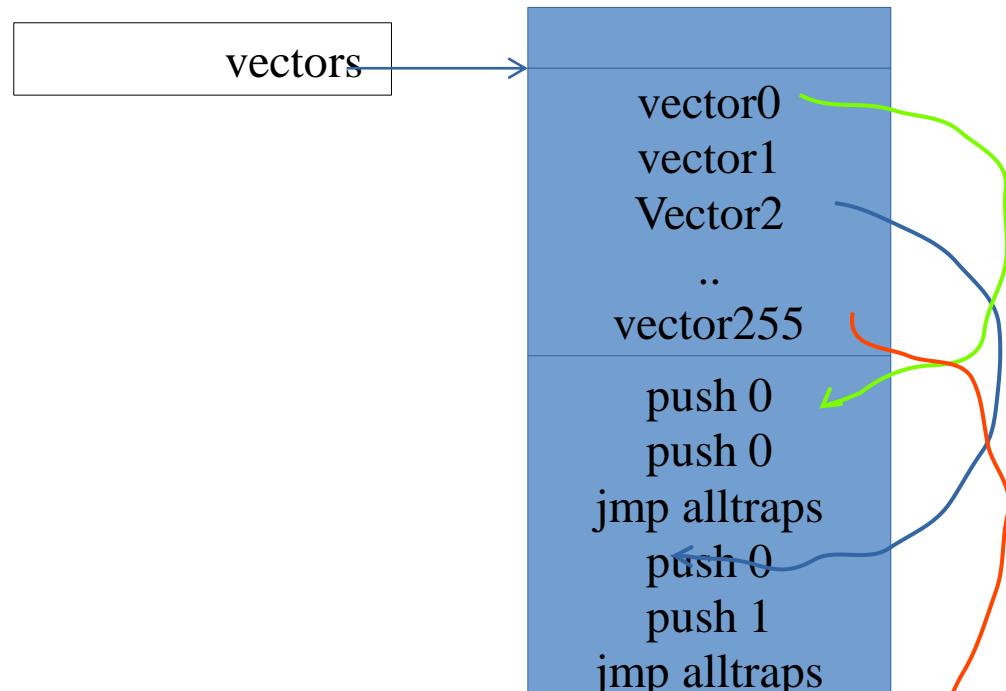
Privilege level

- 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)

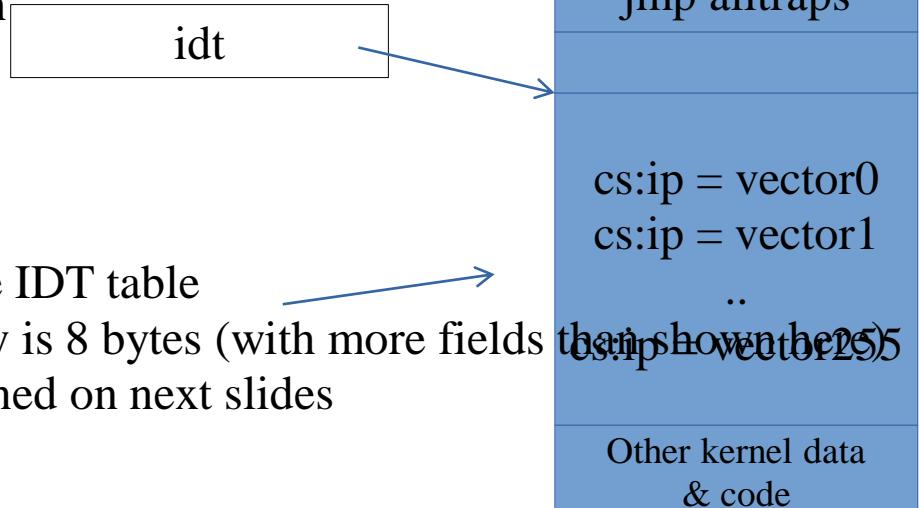
- IDT defines interrupt handlers
- Has 256 entries
 - each giving the %cs and %eip to be used when handling the corresponding interrupt.
- Mapping
 - Interrupts 0-31 are defined for software exceptions, like divide errors or attempts to access invalid memory addresses.
 - Xv6 maps the 32 hardware interrupts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

IDT setup
done by tvinit() function



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

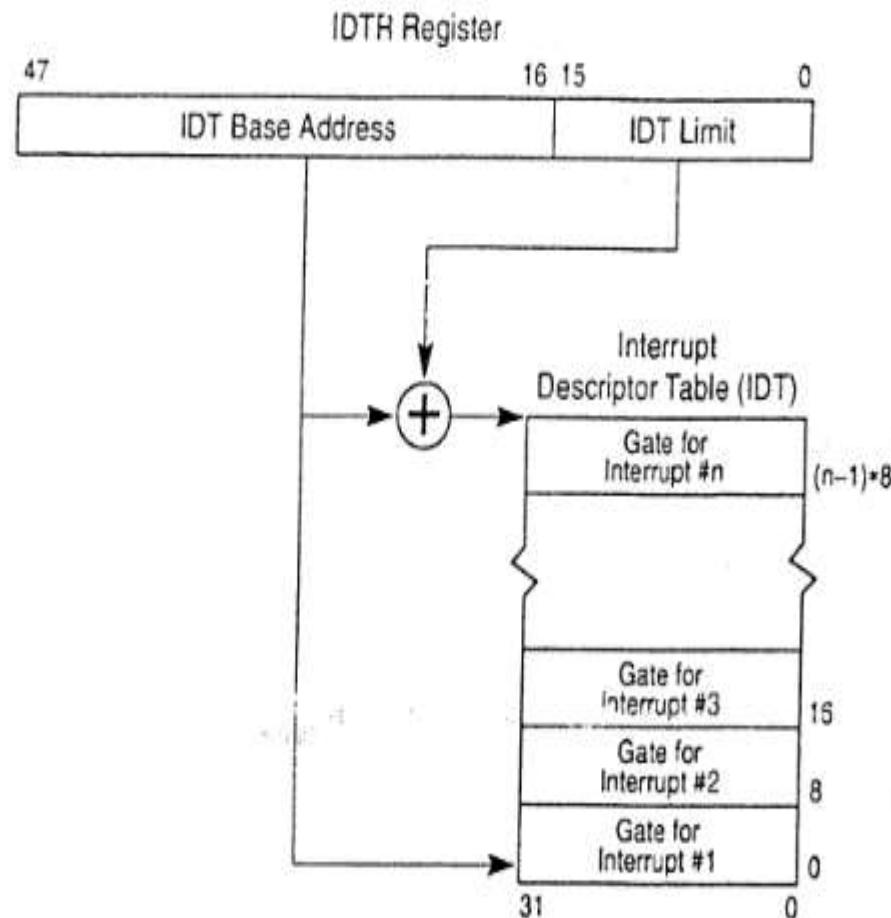
The tvinit() is called during kernel initialization



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



IDTR and IDT



IDT is in RAM

IDTR is in CPU

Interrupt Descriptor Table (IDT) entries (in RAM)

```
// Gate descriptors for interrupts  
and traps
```

```
struct gatedesc {  
    uint off_15_0 : 16; // low 16 bits  
of offset in segment  
    uint cs : 16; // code segment  
selector  
    uint args : 5; // # args, 0 for  
interrupt/trap gates  
    uint rsv1 : 3; // reserved(should
```

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,
```

Setting IDT entries

```
#define SETGATE(gate, istrap, sel,
off, d)          \
{                  \
    (gate).off_15_0 = (uint)(off) & \
0xffff;           \
    (gate).cs = (sel); \
    (gate).args = 0; \
}
```

Setting IDT entries

Vectors.S

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

```
# handlers
```

```
.globl alltraps
```

```
.globl vector0
```

```
vector0:
```

```
    pushl $0
```

```
    li t0
```

trapasm.S

```
#include "mmu.h"  
  
# vectors.S sends all  
traps here.
```

```
.globl alltraps
```

```
alltraps:
```

```
    # Build trap frame.
```

```
    pushl %ds
```

```
    pushl %es
```

```
    pushl %fs
```

How will interrupts be handled?

On int instruction/interrupt the CPU does this:

- Fetch the n'th descriptor from the IDT, where n is the argument of int.
(IDTR->idt[n])
- Check that CPL in %cs is <= DPL, where DPL is the privilege level in the descriptor.
- Temporarily save %esp and %ss in CPU-internal registers, but
 - 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

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
```

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();

trap()

- Argument is trapframe**

- In alltraps**

- Before “call trap”, there was “push %esp” and stack had the trapframe**

- Remember calling convention --> when a function is called, the stack contains the arguments in reverse order (here only 1 arg)**

trap()

- Has a switch
- **switch(tf->trapno)**
- Q: who set this trapno?
- Depending on the type of trap
- Call interrupt handler
 - Timer
 - wakeup(&ticks)
 - IDE: disk interrupt
 - Ideintr()
 - KBD
 - Kbdintr()
 - COM1
 - Uatrintr()
 - If Timer

when trap() returns

❑#Back in alltraps

call trap

addl \$4, %esp

Return falls through
to trapret...

.globl trapret

trapret:

popal

❑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

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

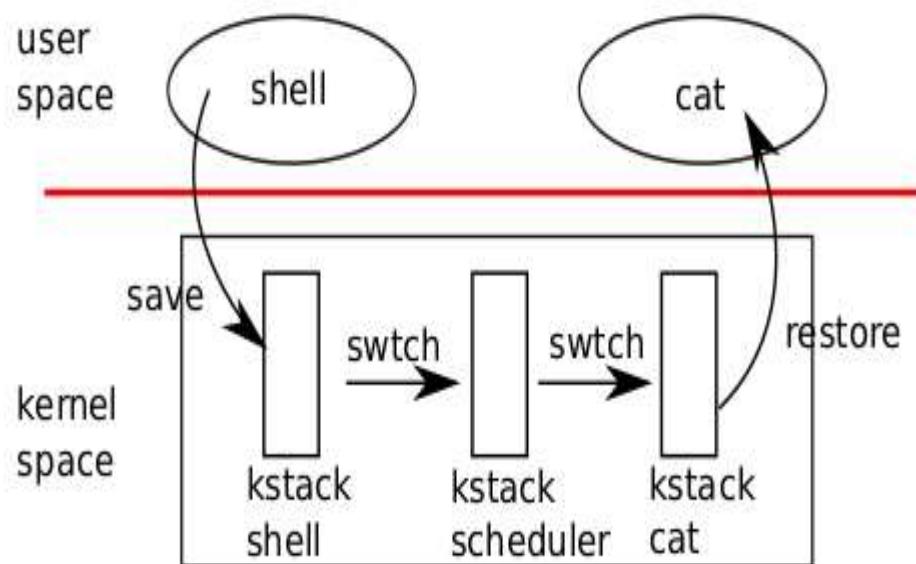
F() { i++; j++; }

❑ P2 was doing

G() { x--; y++; }

❑ P1 will experience a timer interrupt,

4 stacks need to change!



- User stack of process ->
- kernel stack of process
- Switch to kernel stack
- The normal sequence on any

scheduler()

- Enable interrupts
- Find a **RUNNABLE** process. Simple round-robin!
- **c->proc = p**
- **switchuvm(p) : Save TSS and make CR3 to point to new process pagedir**
- **p->state = RUNNING**

swtch

swtch:

movl 4(%esp), %eax

movl 8(%esp), %edx

Save old callee-saved registers

pushl %ebp

pushl %ebx

pushl %esi

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

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

void

sched(void)

{

int intena;

struct proc *p =
myproc();

if(!holding(&ptabl
e.lock))

sched()

□ get current
process

□ Error checking
code (ignore as of
now)

□ get interrupt
enabled status on
current CPU
(ignore as of now)

sched() and schduler()

```
 sched() {
```

...

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

□ after swtch() call in sched(), the control jumps to Y in scheduler

```
    scheduler(void) {
```

...

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

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

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()
- Now the loop in scheduler()
 - calls switchkvm()
 - Then continues to find next process (P2) to run
 - Then calls switchuvm(p): changing the page

File Systems

Abhijit A M

abhijit.comp@coep.ac.in

What we are going to learn

- The operating system interface (system calls, commands/utilities) for accessing files in a file-system
- Design aspects of OS to implement the file system
 - On disk data structure
 - In memory kernel data structures

What is a file?

- A (dumb!) sequence of bytes (typically on a permanent storage:secondary, tertiary) , with
 - A name
 - Permissions
 - Owner
 - Timestamps,
 - Etc.

File types and kernel

- For example, MP4 file
- *vlc* will do a open(...) on the file, and call read(...),
interpret the contents of the file as movie and show movie
- Kernel will simply provide open(...) read(...),
write(...) to access file data
- Meaning of the file contents is known to VLC and
not to kernel!

file type	usual extension	function
executable	exe, com, bin or none	ready-to-run machine-language program
object	obj, o	compiled, machine language, not linked
source code	c, cc, java, pas, asm, a	source code in various languages
batch	bat, sh	commands to the command interpreter
text	txt, doc	textual data, documents
word processor	wp, tex, rtf, doc	various word-processor formats
library	lib, a, so, dll	libraries of routines for programmers
print or view	ps, pdf, jpg	ASCII or binary file in a format for printing or viewing
archive	arc, zip, tar	related files grouped into one file, sometimes compressed, for archiving or storage
multimedia	mpeg, mov, rm, mp3, avi	binary file containing audio or A/V information

What is a file?

- The sequence of bytes can be *interpreted* (*by an application*) to be
 - Just a sequence of bytes
 - E.g. a text file
 - Sequence of records/structures
 - E.g. a file of student records , by database application, etc
 - A complexly organized collection of records and

File attributes

- Run

- \$ ls -l

- on Linux

- To see file listing with different attributes

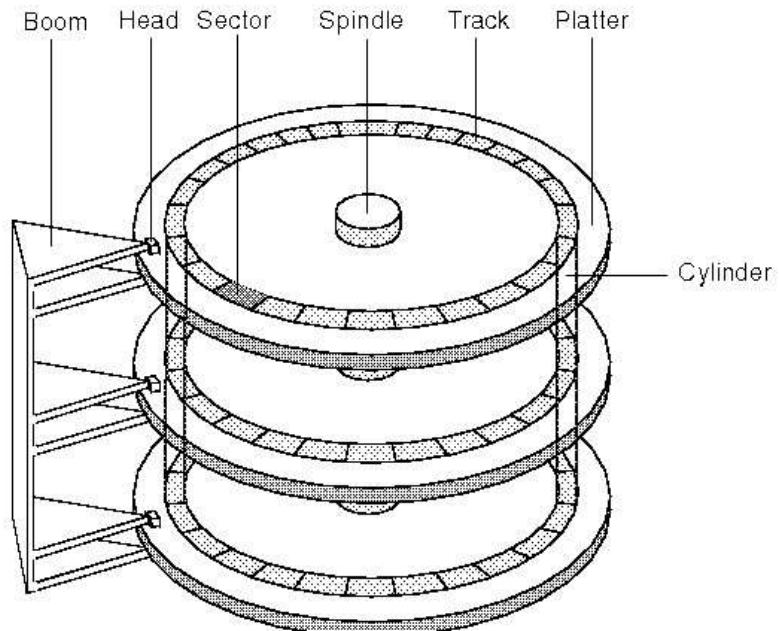
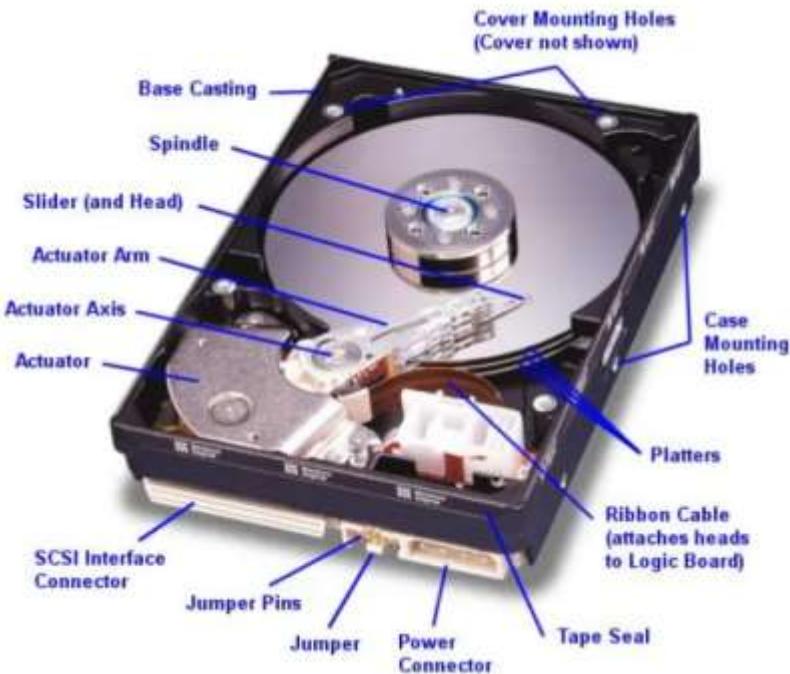
- Different OSes and file-systems provide different sets of file attributes

- Some attributes are common to most, while some

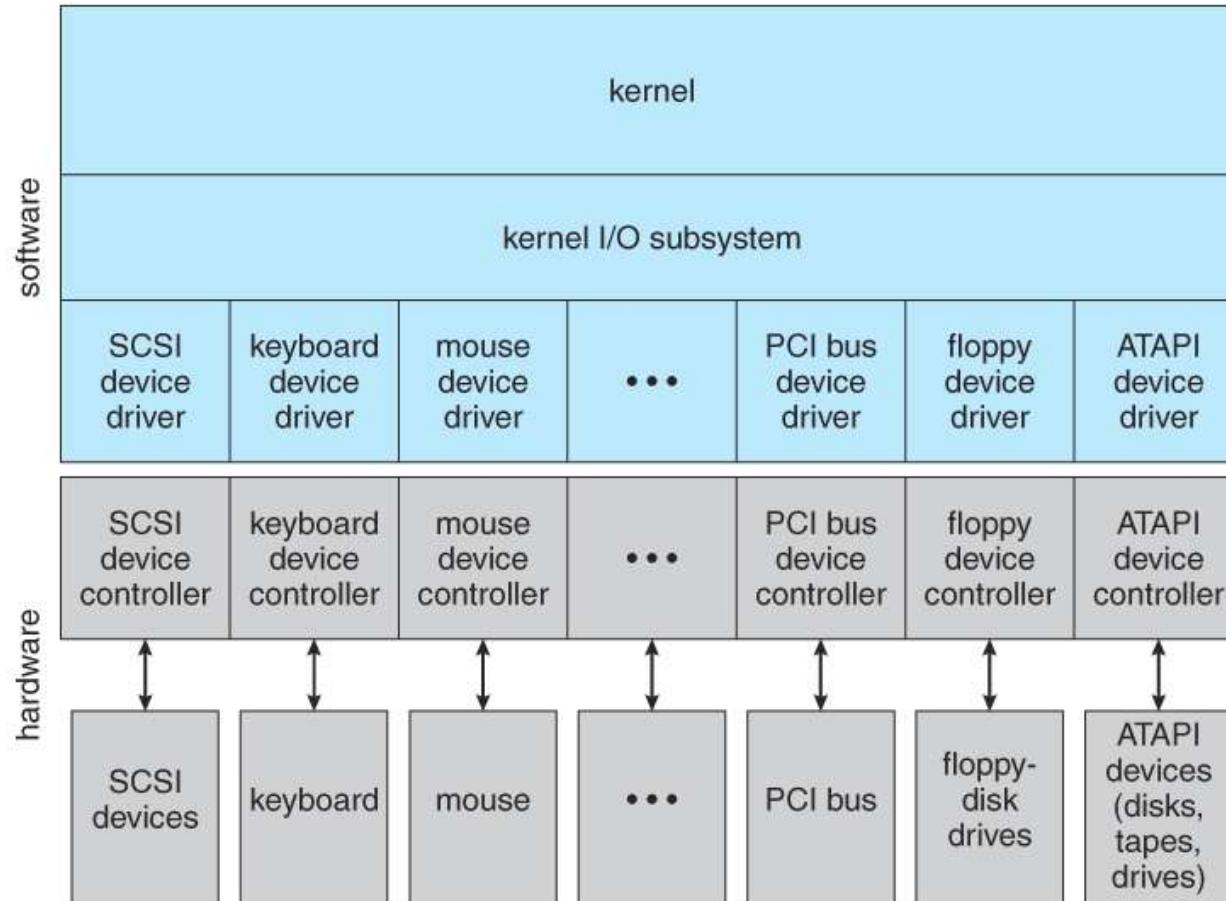
Access methods

- OS system calls may provide two types of access to files
 - Direct Access
 - read n
 - write n
 - position to n
 - Sequential Access
 - read next
 - write next
 - reset
 - no read after

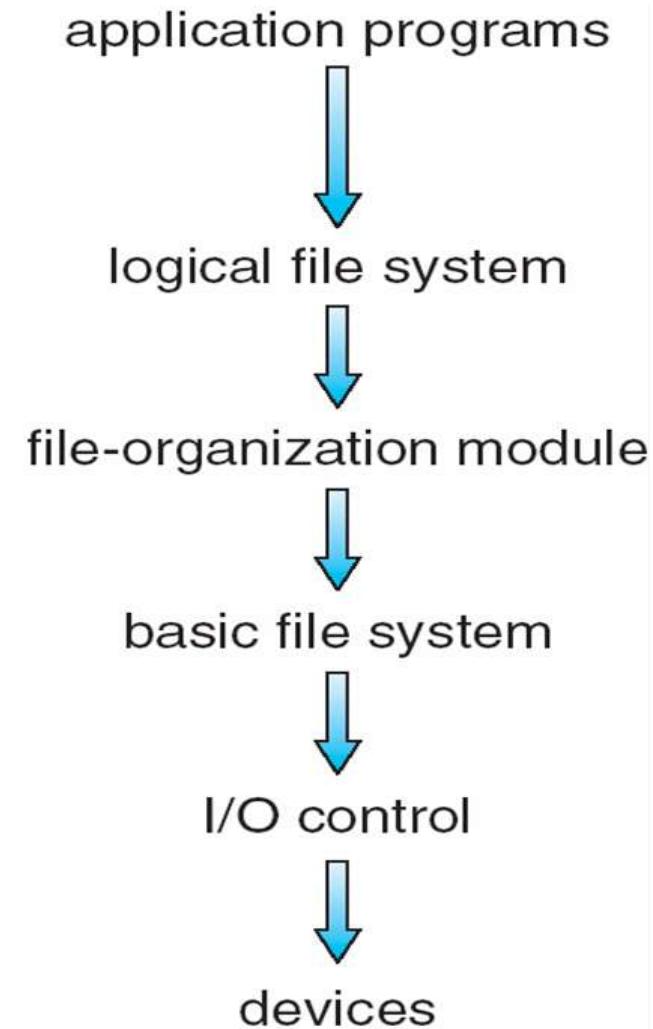
Disk



Device Driver



File system implementation: layering



Application programs

```
int main() {  
    char buf[128]; int count;  
    fd = open(...);  
    read(fd, buf, count);  
}
```

OS

Logical file system:

```
sys_read(int fd, char *buf, int count) {  
    file *fp = currproc->fdarray[fd];  
    file_read(fp, ...);  
}
```

File organization module:

```
file_read(file *fp, char *buf, int count) {  
    offset = fp->current-offset;  
    translate offset into blockno;  
    basic_read(blockno, buf, count);  
}
```

Basic File system:

```
basic_read(int blockno, char *buf, ...) {  
    os_buffer *bp;  
    sectorno = calculation on blockno;  
    disk_driver_read(sectorno, bp );  
    move-process-to-wait-queue;  
    copy-data-to-user-buffer(bp, buf);  
}
```

IO Control, Device driver:

```
disk_driver_read(sectorno) {  
    issue instructions to disk controller (ofte  
    to read sectorno    into specific location  
}
```

XV6 does it slightly differently, but following the

OS's job now

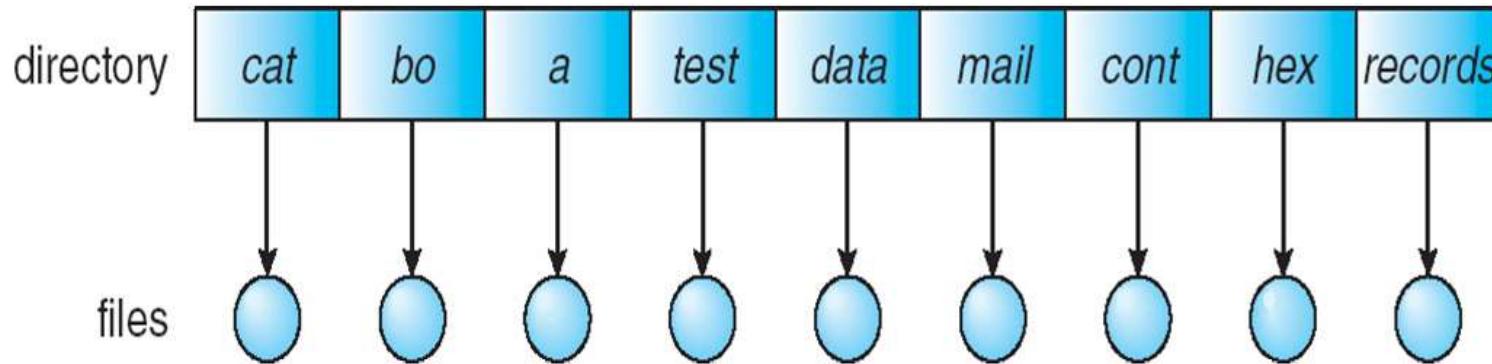
- To implement the logical view of file system as seen by end user
- Using the logical block-based view offered by the device driver

Formatting

- Physical hard disk divided into partitions
- Partitions also known as minidisks, slices
- A raw disk partition is accessible using device driver – but no block contains any data !
- Like an un-initialized array, or sectors/blocks
- Formatting
- Creating an initialized data structure on the partition, so that it can start storing the acyclic

Different types of “layouts”

Single level directory



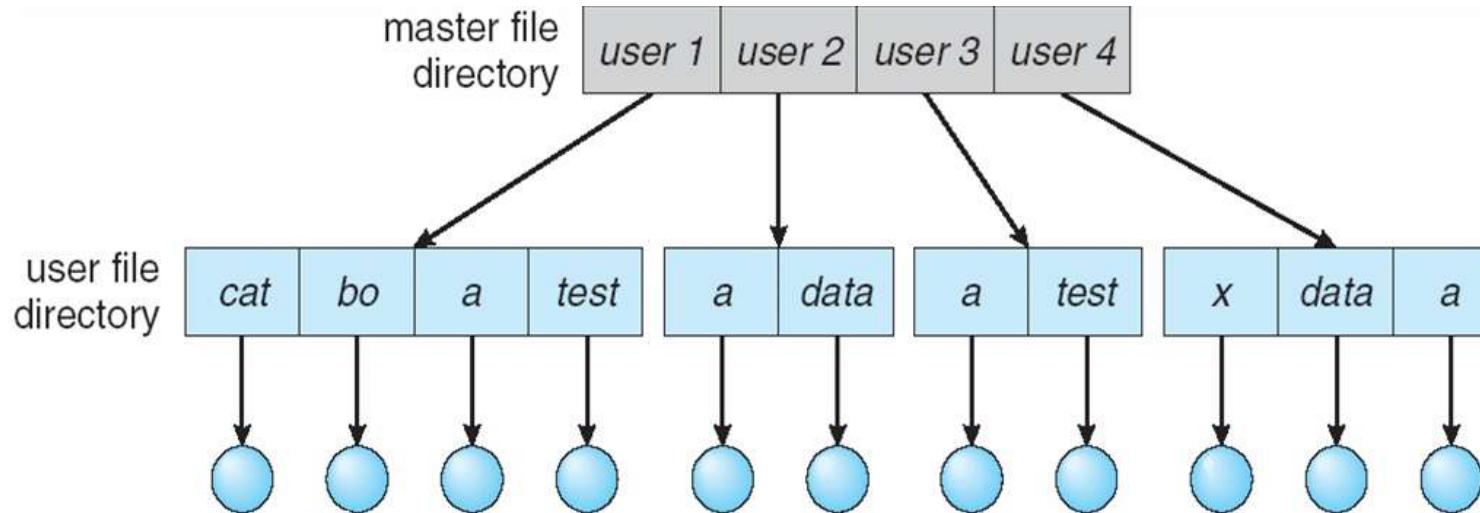
Naming problem

Grouping problem

Example: RT-11, from 1970s <https://en.wikipedia.org/wiki/RT-11>

Different types of “layouts”

Two level directory



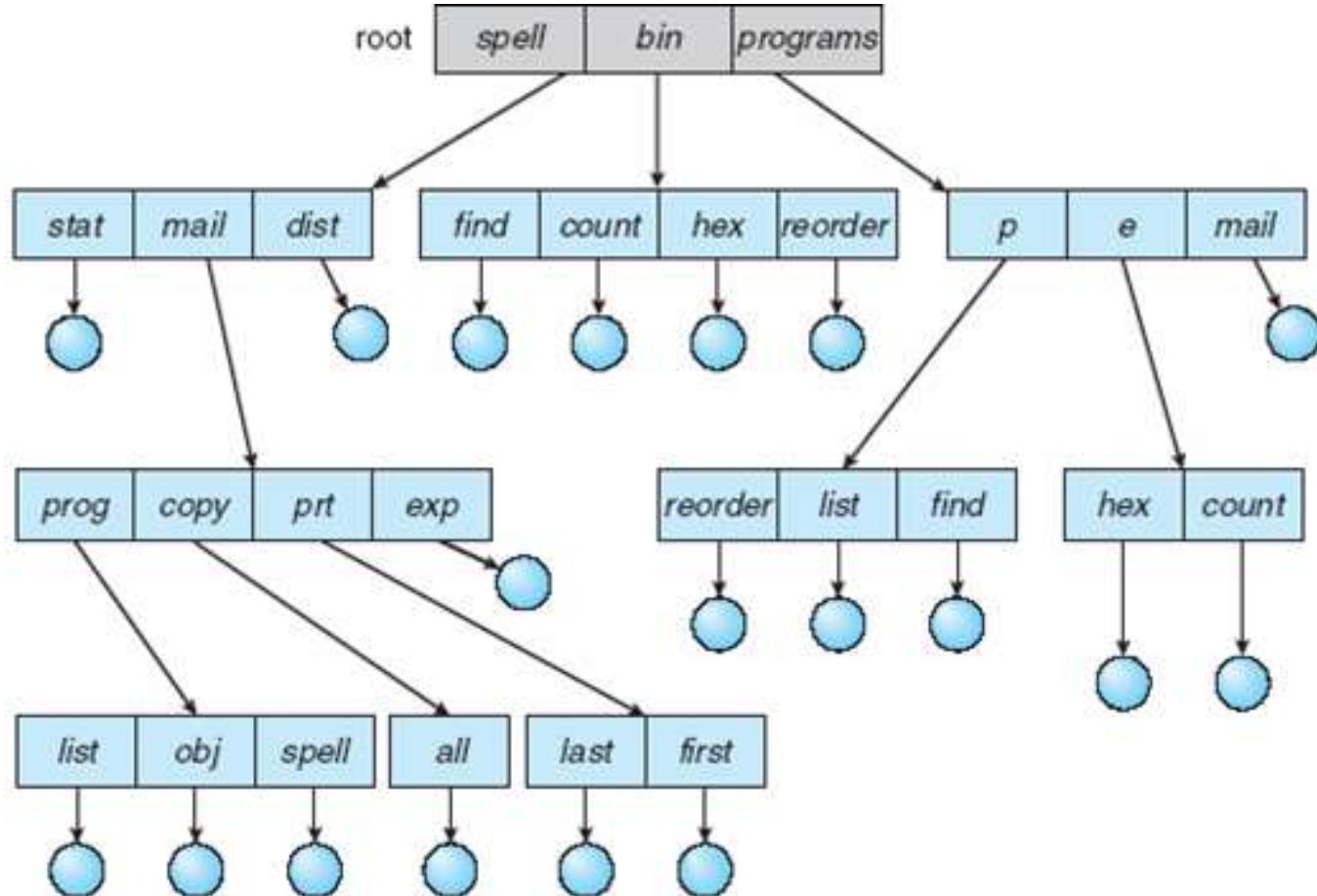
Path name

Can have the same file name for different user

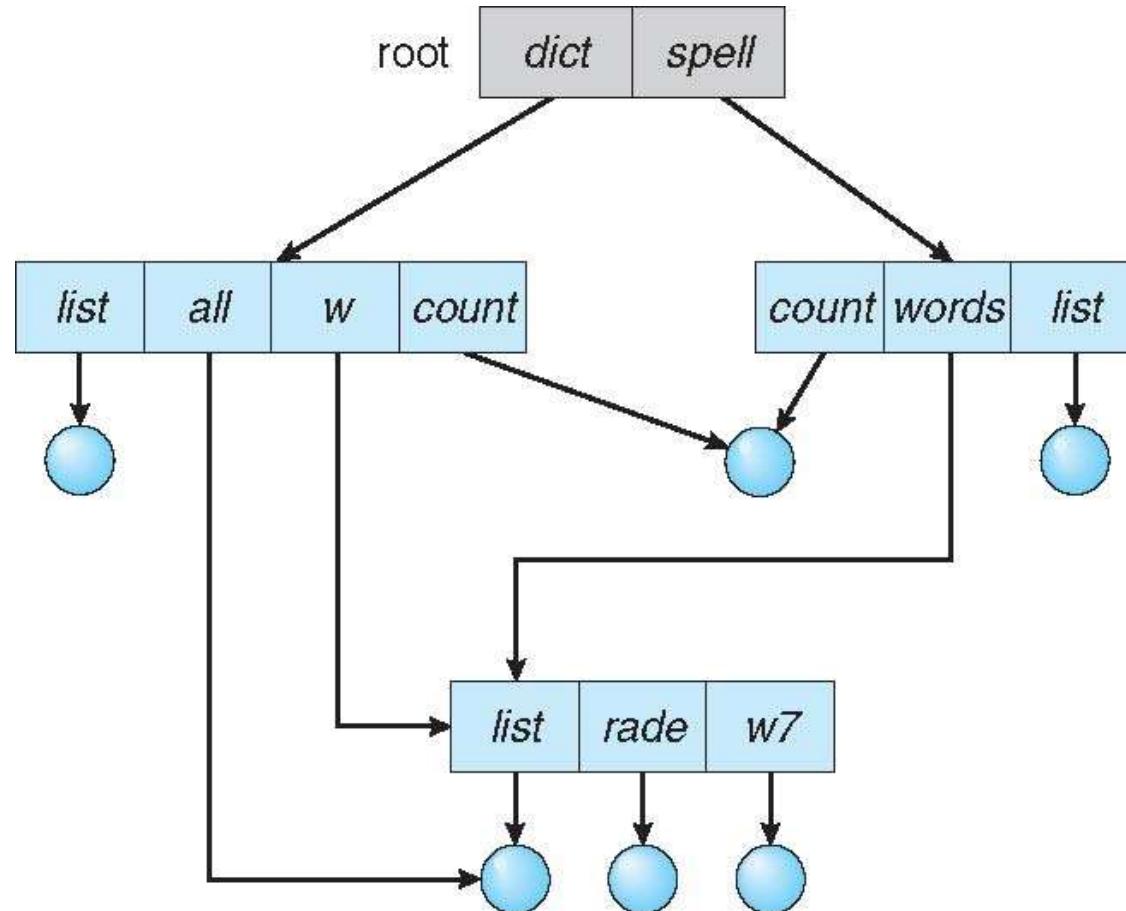
Efficient searching

No grouping capability

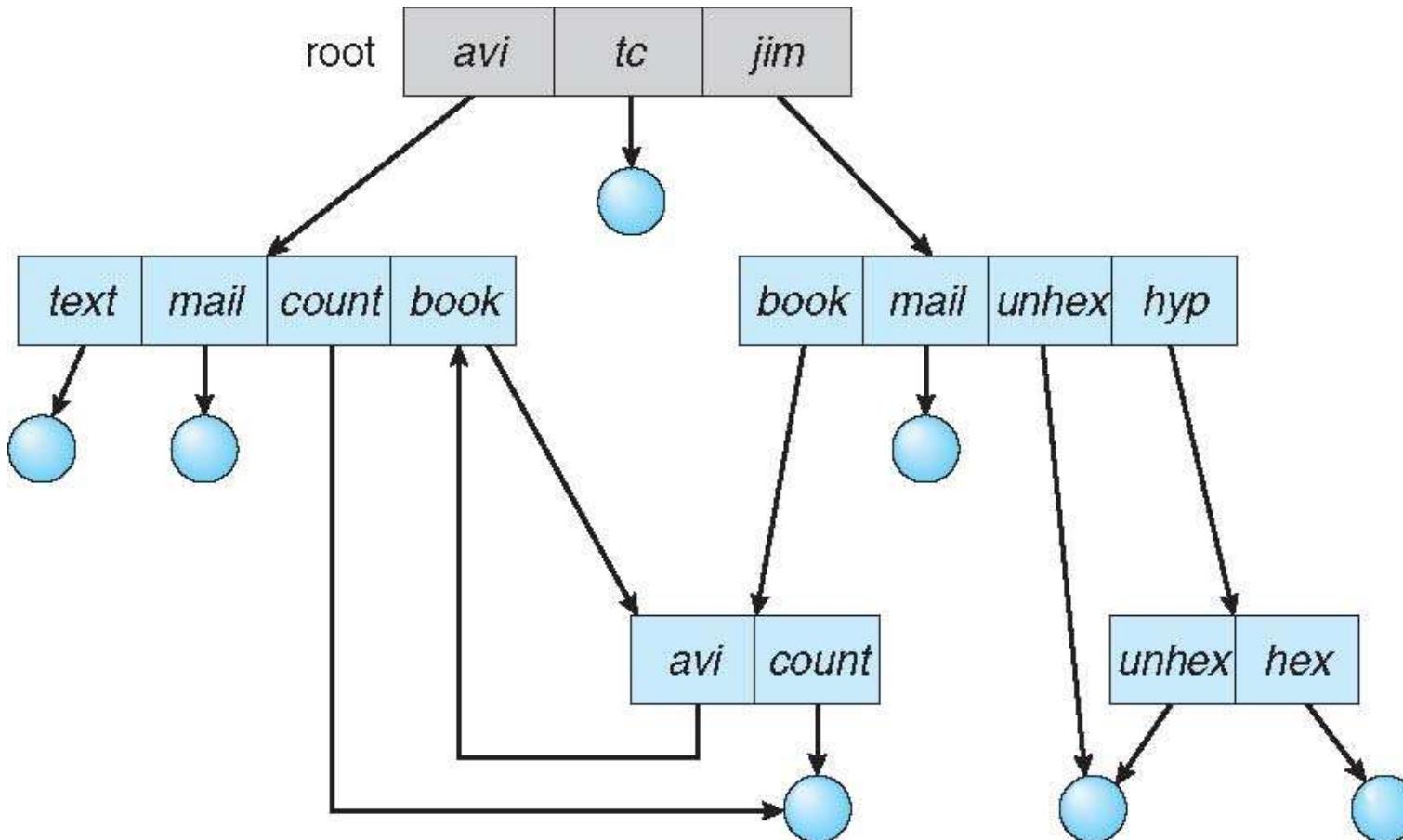
Tree Structured directories



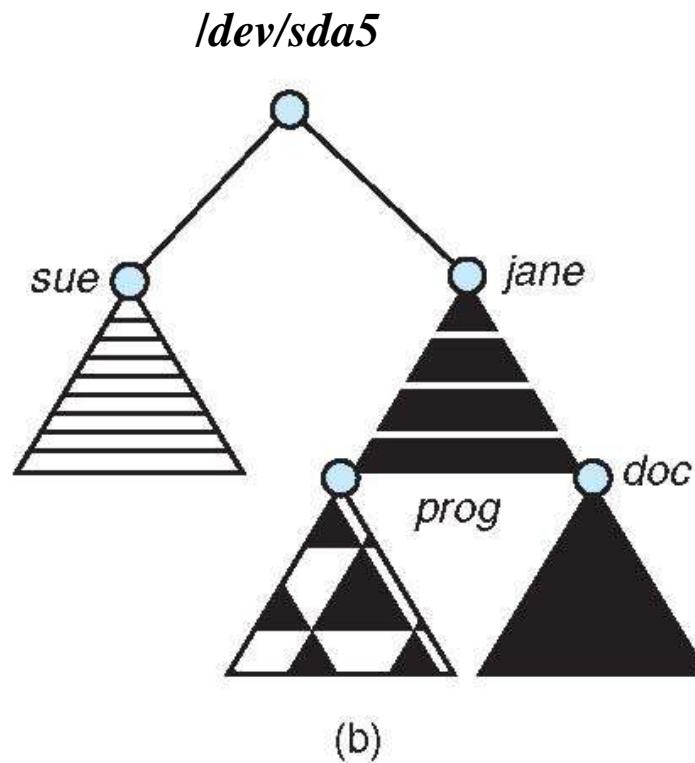
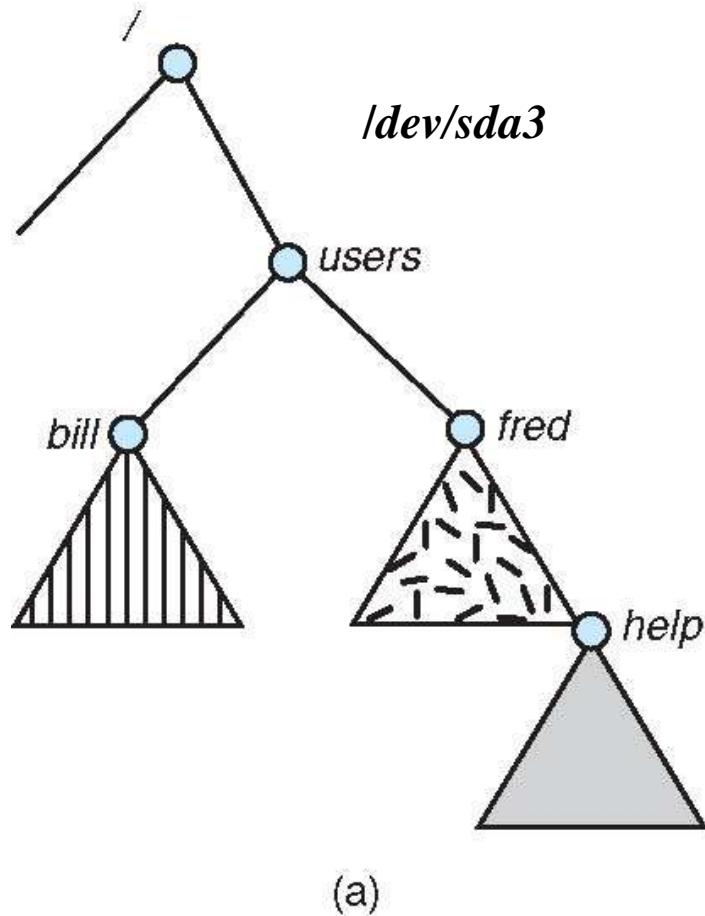
Acyclic Graph Directories



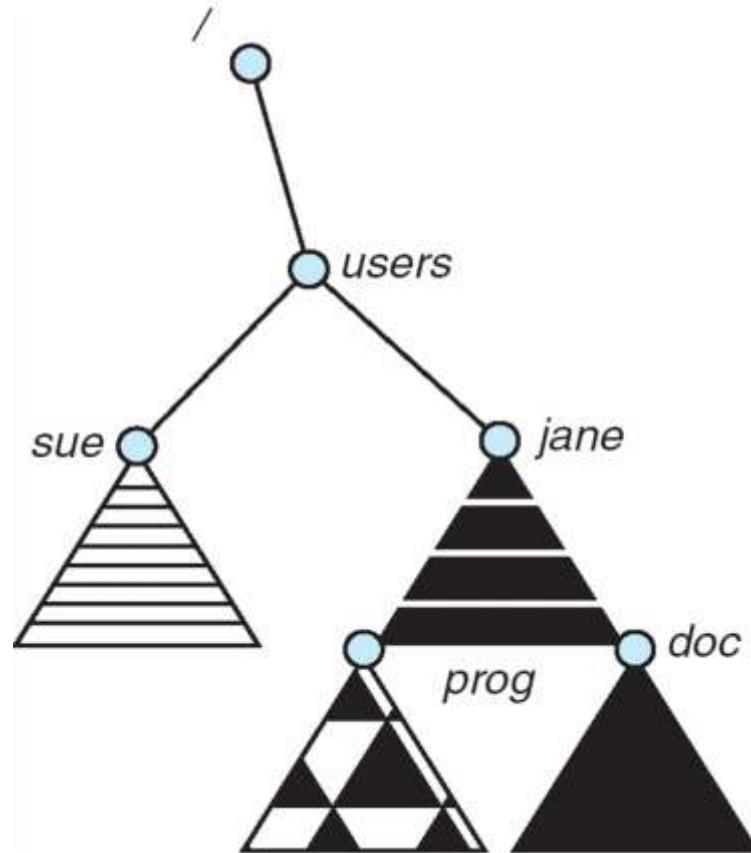
General Graph directory



Mounting of a file system: before



Mounting of a file system: after



`$sudo mount /dev/sda5 /users`

Remote mounting: NFS

- Network file system

- \$ sudo mount 10.2.1.2:/x/y /a/b

- The /x/y partition on 10.2.1.2 will be made available under the folder /a/b on this computer

File sharing semantics

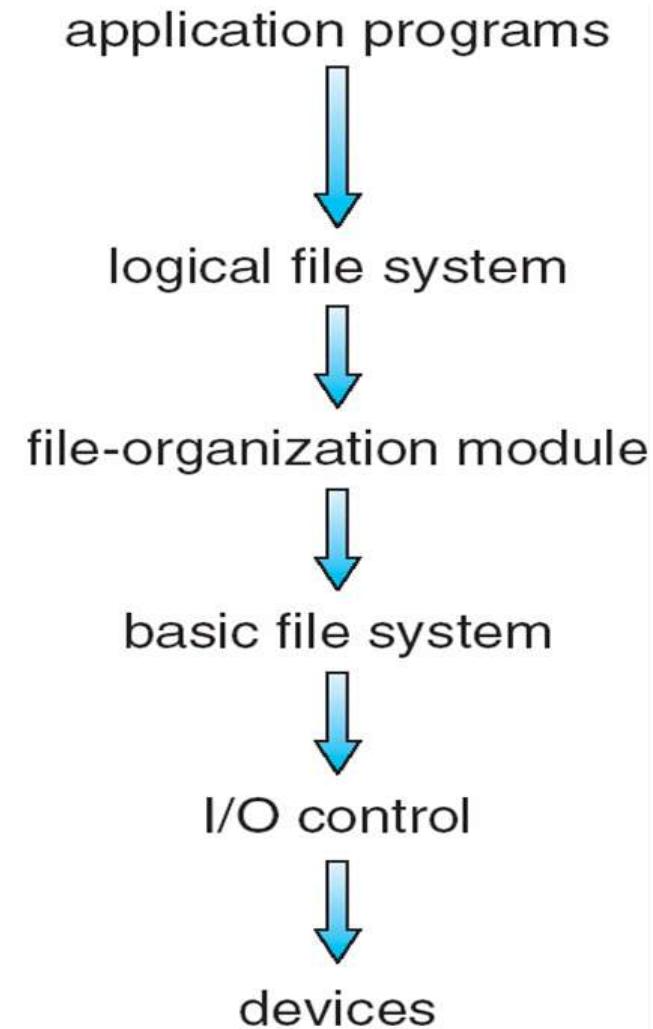
- Consistency semantics specify how multiple users are to access a shared file simultaneously
- Unix file system (UFS) implements:
 - Writes to an open file visible immediately to other users of the same open file
 - One mode of sharing file pointer to allow multiple users to read and write concurrently
- AFS has session semantics

Implementing file systems

File system on disk

- What we know
- Disk I/O in terms of sectors (512 bytes)
- File system: implementation of acyclic graph using the linear sequence of sectors
- Store a acyclic graph into array of “blocks”/“sectors”
- Device driver available: gives sector/block wise access to the disk

File system implementation: layering



File system: Layering

- Device drivers manage I/O devices at the I/O control layer
- Given commands like “read drive1, cylinder 72, track 2, sector 10, into memory location 1060” outputs low-level hardware specific
 - File organization module understands files, logical address, and physical blocks
 - Translates logical block # to physical block #
 - Manages free space, disk allocation

File system implementation: Different problems to be solved

- What to do at boot time, how to locate kernel ?
- How to store directories and files on the partition ?
- Complex problem. Hierarchy + storage allocation + efficiency + limits on file/directory sizes + links (hard, soft)
- How to manage list of free sectors/blocks?
- How to store the summary information about the complete file system : #files, #free-blocks, ...

File system implementation: Different problems to be solved

- About storing a file, how to store**

- Data**

- Attributes**

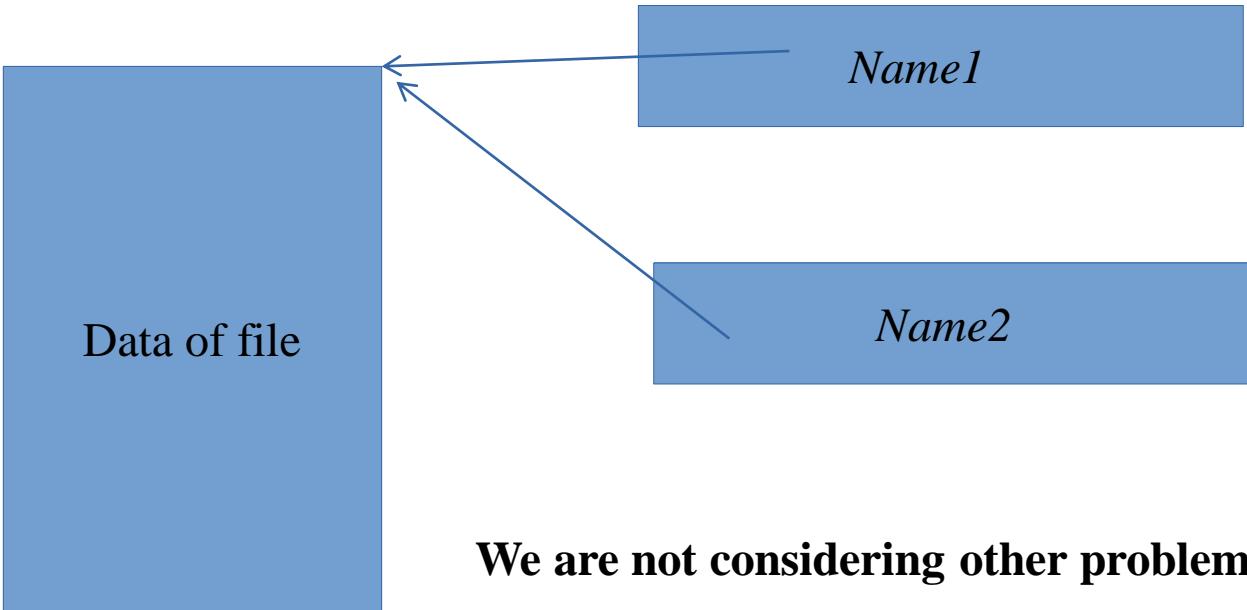
- Name**

- Link count**

The hard link problem

- Need to separate name from data !
- */x/y and /a/b* should be same file. How?
- Both names should refer to same data !
- Data is separated separately from name, and the name gives a “reference” to data
- What about attributes ?
- They go with data! (not with name!)

The hard link problem



We are not considering other problems here , just focussing on hard link p

A typical file control block (inode)

file permissions
file dates (create, access, write)
file owner, group, ACL
file size
file data blocks or pointers to file data blocks

Name is stored separately

Where?

IN data block of directory

In memory data structures

- Mount table

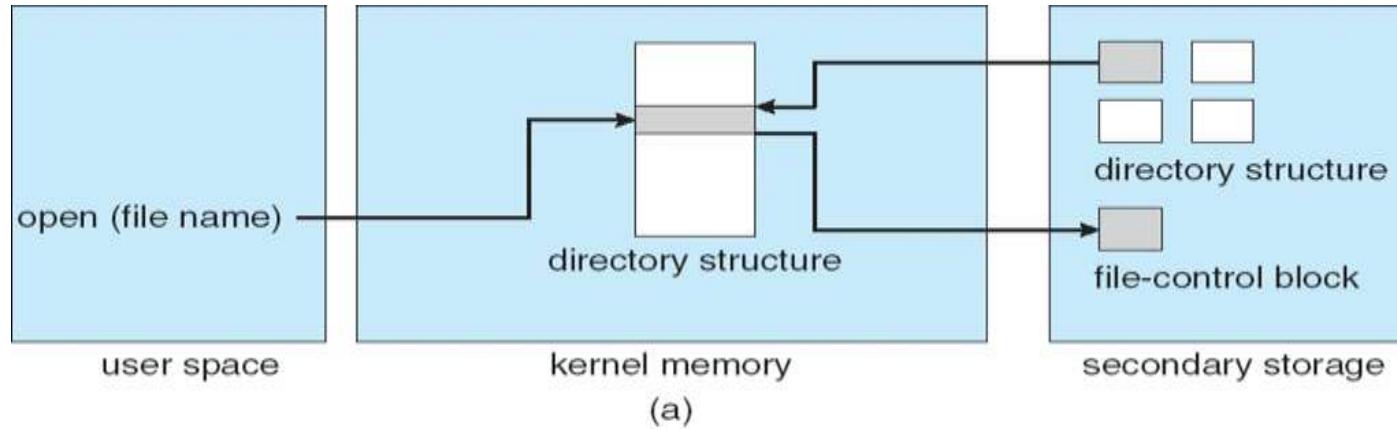
- **storing file system mounts, mount points, file system types**

- See next slide for “file” related data structures

- Buffers

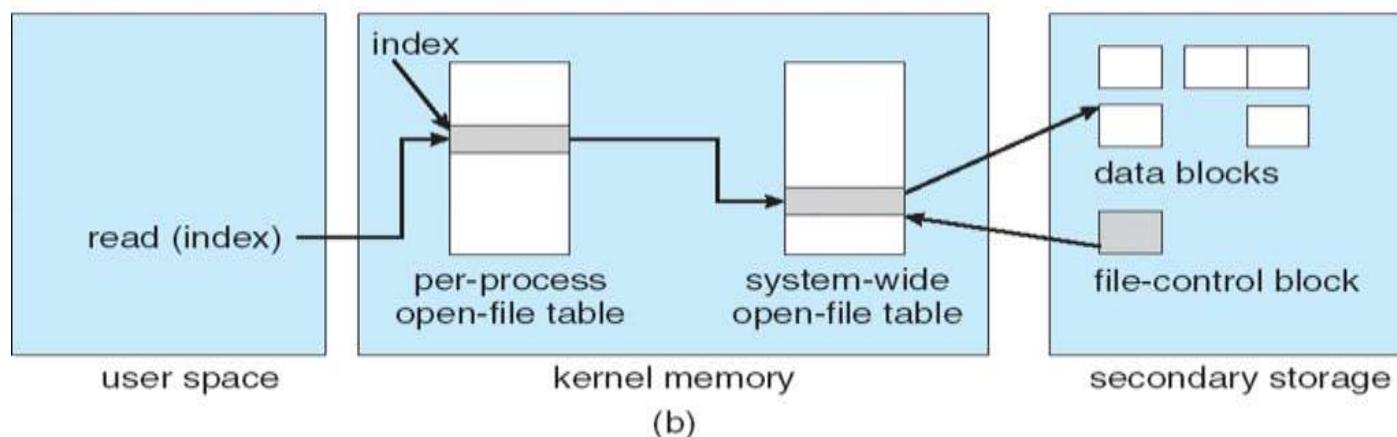
- **hold data blocks from secondary storage**

In memory data structures: for open,read,write, ...



Open returns a file handle for

Data from read eventually co



At boot time

- Root partition
- Contains the file system hosting OS
- “mounted” at boot time – contains “/”
- Normally can’t be unmounted!
- Check all other partitions
- Specified in */etc/fstab* on Linux
- Check if the data structure on them is consistent

Directory Implementation

- **Problem**

- **Directory contains files and/or subdirectories**
 - Operations required – create files/directories, access files/directories, search for a file (during lookup), etc.
- **Directory needs to give location of each file on disk**

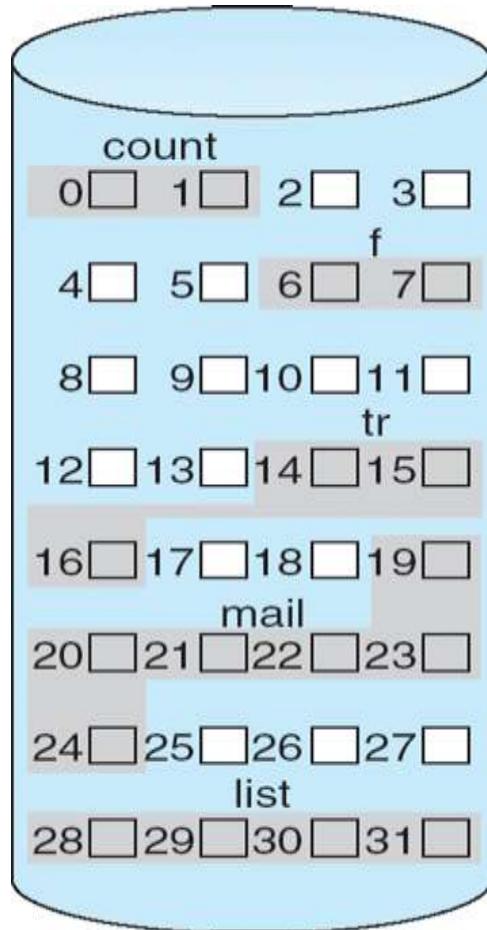
Directory Implementation

- Linear list of file names with pointer to the data blocks
- Simple to program
- Time-consuming to execute
- Linear search time
- Could keep ordered alphabetically via linked list or use B+ tree
- Ext2 improves upon this approach.

Disk space allocation for files

- File contain data and need disk blocks/sectors for storing it
- File system layer does the allocation of blocks on disk to files
- Files need to
 - Be created, expanded, deleted, shrunk, etc.
 - How to accommodate these requirements?

Contiguous Allocation of Disk Space



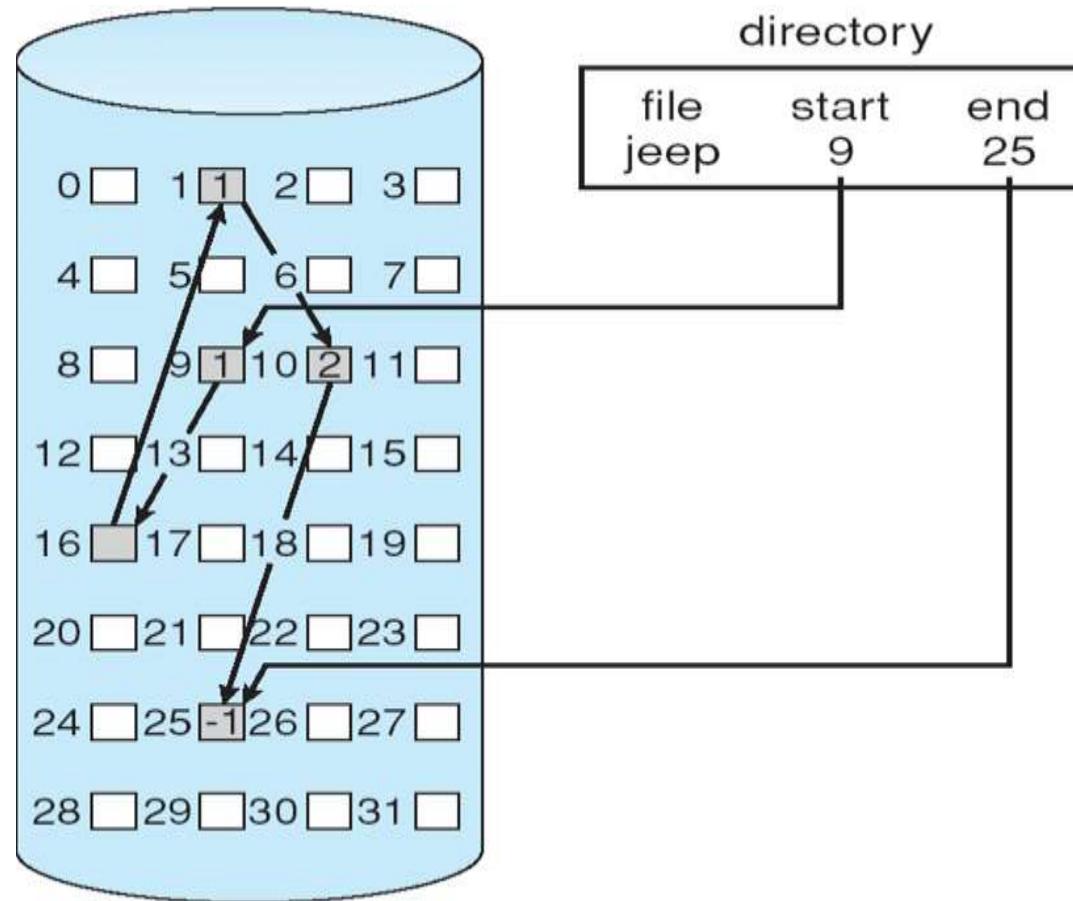
directory

file	start	length
count	0	2
tr	14	3
mail	19	6
list	28	4
f	6	2

Contiguous allocation

- Each file occupies set of contiguous blocks
- Best performance in most cases
- Simple – only starting location (block #) and length (number of blocks) are required
- Problems include finding space for file, knowing file size, external fragmentation, need for compaction off-line (downtime) or on-line

Linked allocation of blocks to a file

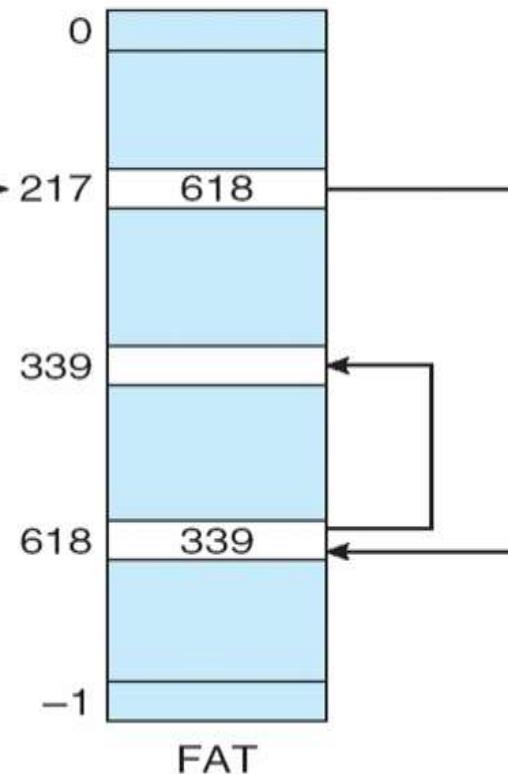
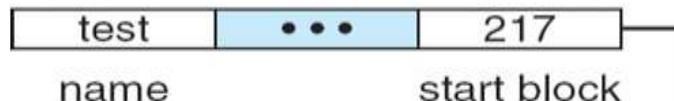


Linked allocation of blocks to a file

- Linked allocation
- Each file a linked list of blocks
- File ends at nil pointer
- No external fragmentation
- Each block contains pointer to next block (i.e.
- Free space management system called when new block needed
- Improve efficiency by clustering blocks into groups but increases internal fragmentation
- Reliability can be a problem

FAT: File Allocation Table

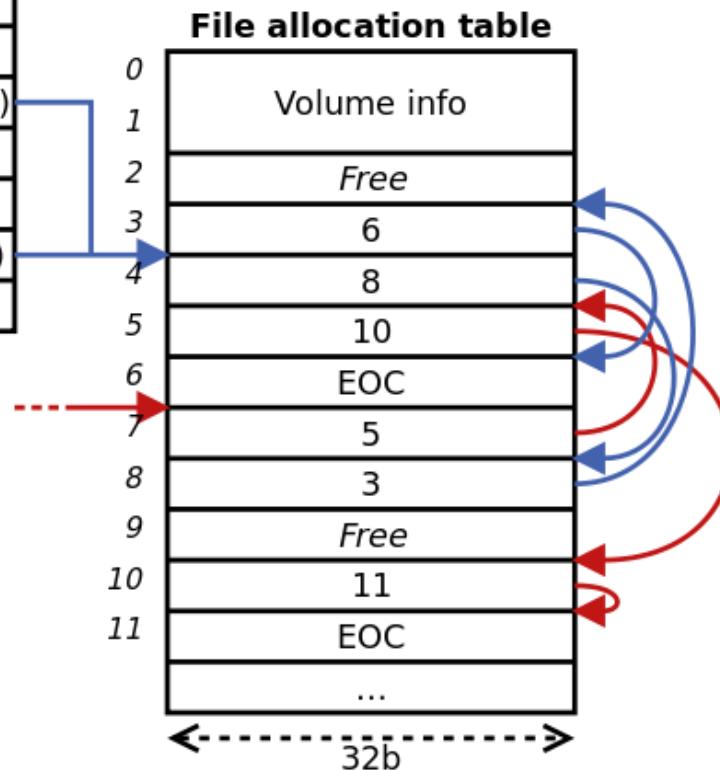
directory entry



- FAT (File Allocation Table), a variation
- Beginning of volume has table, indexed by block number
- Much like a linked list, but faster on disk and cacheable
- New block allocation simple

Directory table entry (32B)

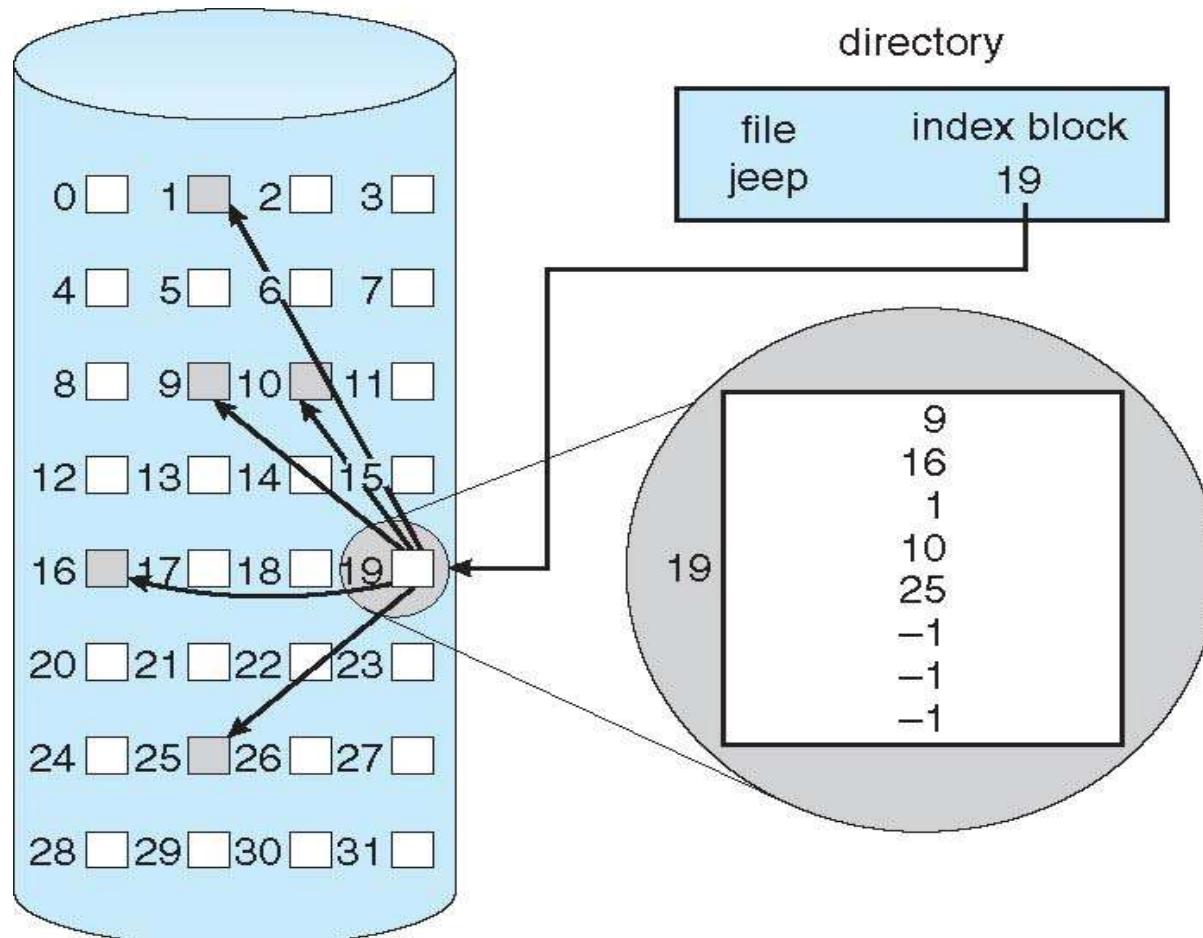
Filename (8B)
Extension (3B)
Attributes (1B)
Reserved (1B)
Create time (3B)
Create date (2B)
Last access date (2B)
First cluster # (MSB, 2B)
Last mod. time (2B)
Last mod. date (2B)
First cluster # (LSB, 2B)
File size (4B)



FAT: File Allocation Table

Variants: FAT8, FAT12, FAT16, FAT32, VFAT, ...

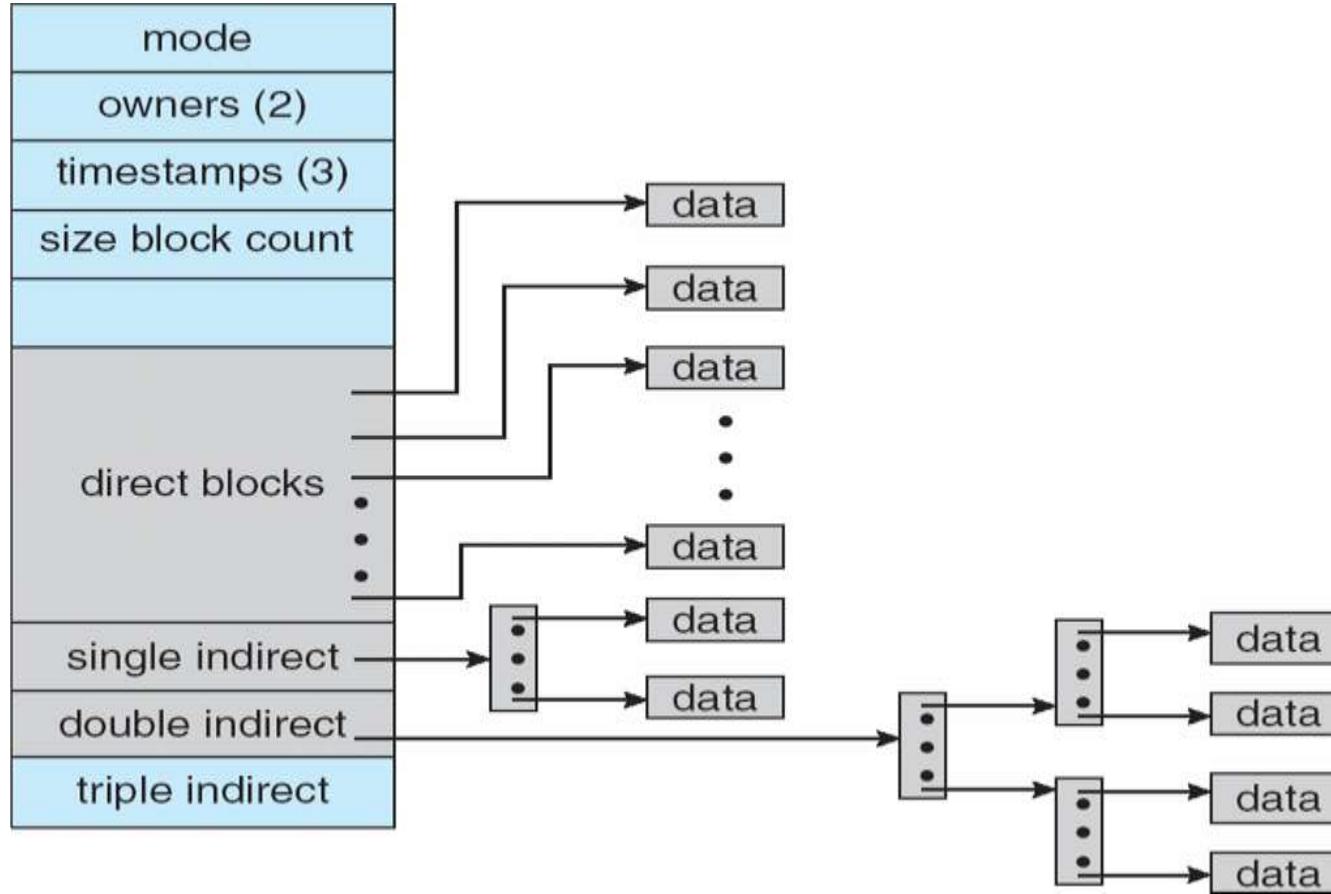
Indexed allocation



Indexed allocation

- Need index table
- Random access
- Dynamic access without external fragmentation,
but have overhead of index block
- Mapping from logical to physical in a file of
maximum size of 256K bytes and block size of 512
bytes. We need only 1 block for index table

Unix UFS: combined scheme for block allocation



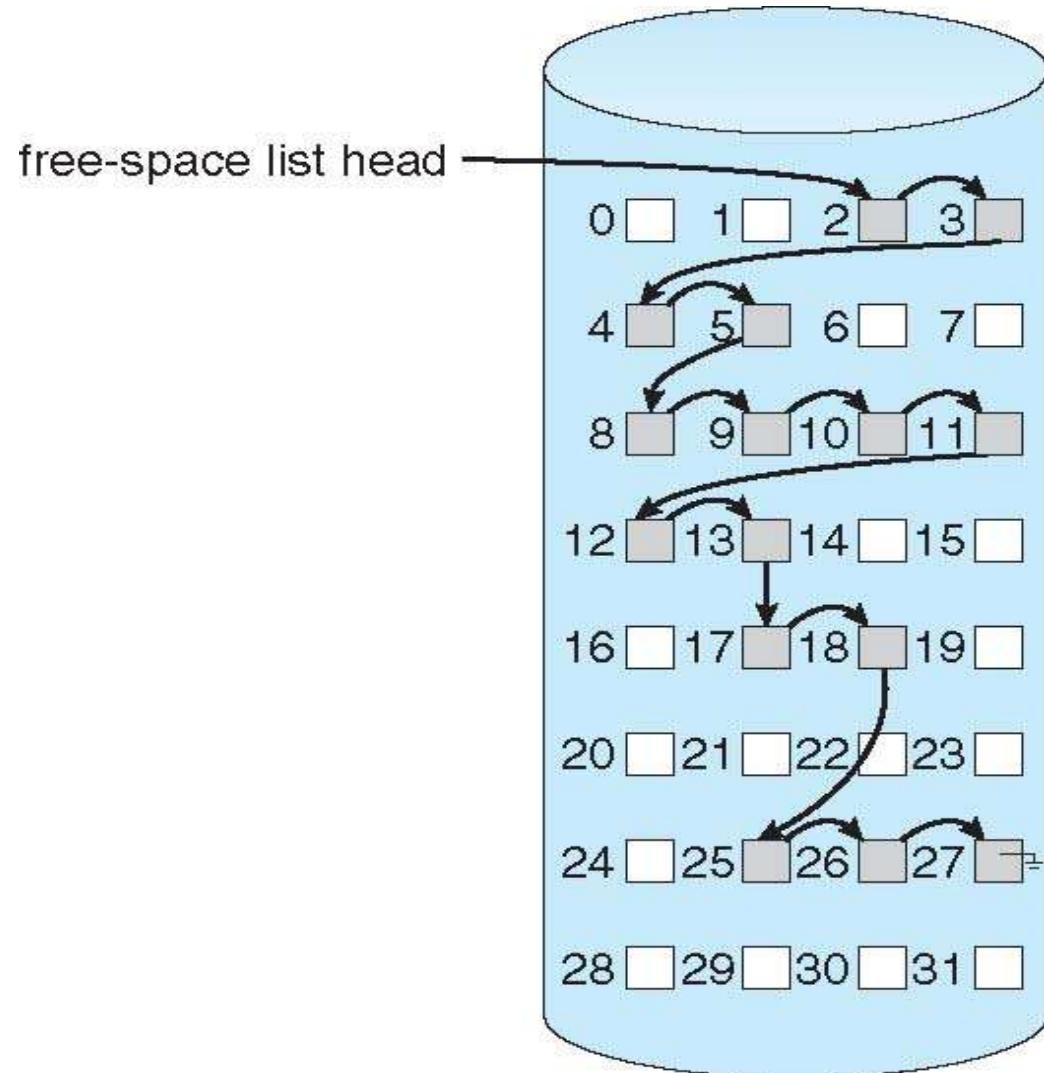
Free Space Management

- ❑ File system maintains free-space list to track available blocks/clusters
- ❑ Bit vector or bit map (n blocks)
- ❑ Or Linked list

Free Space Management: bit vector

- Each block is represented by 1 bit.
- If the block is free, the bit is 1; if the block is allocated, the bit is 0.
- For example, consider a disk where blocks 2, 3, 4, 5, 8, 9, 10, 11, 12, 13, 17 18, 25, 26, and 27 are free and the rest of the blocks are allocated. The free-space bitmap would be
 - 0011100111110001100000011100000 ...

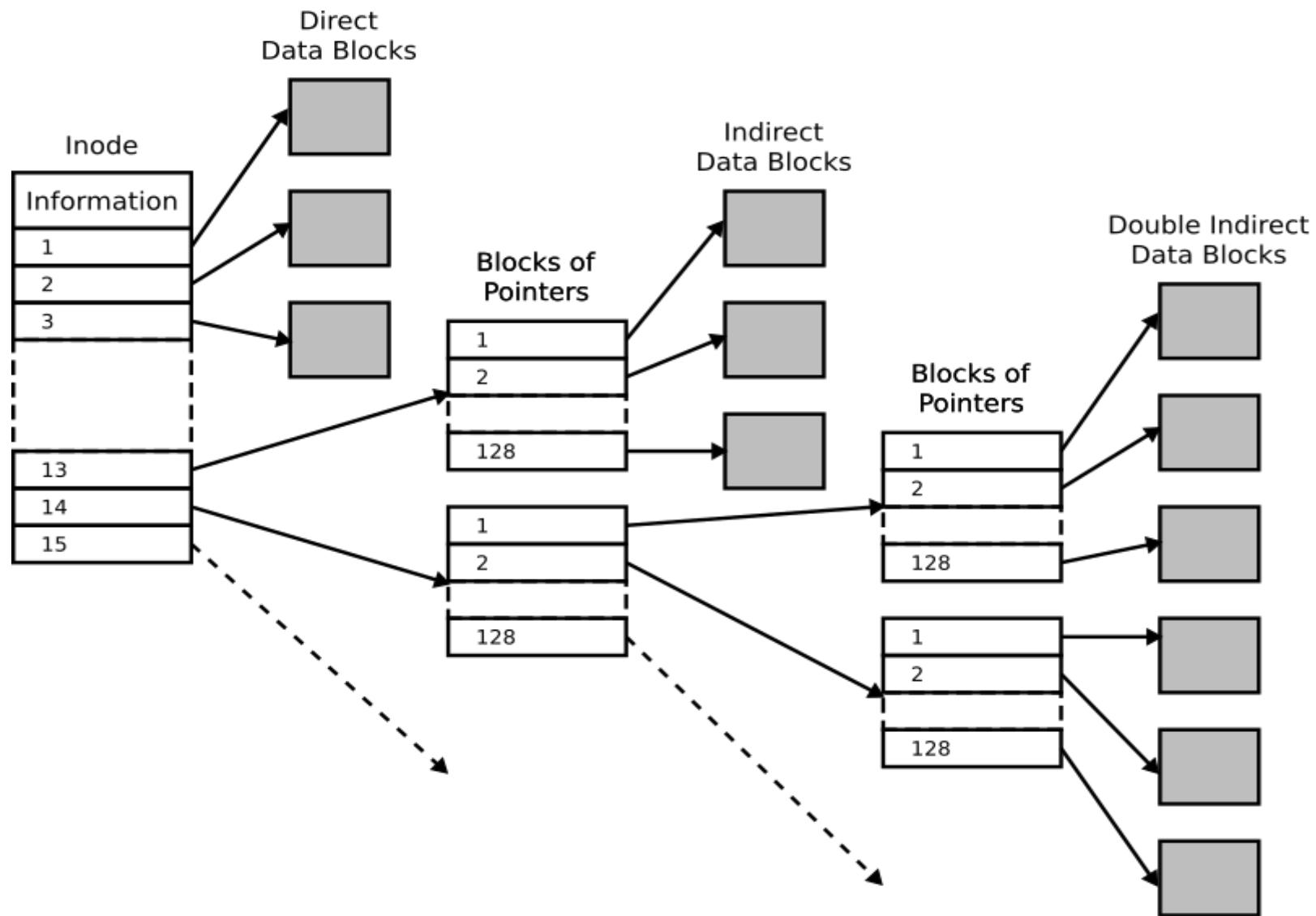
Free Space Management: Linked list (not in memory, on disk!)



Ext2 FS layout

```
struct ext2_inode {  
    __le16 i_mode;    /* File mode */  
    __le16 i_uid;     /* Low 16 bits of Owner Uid */  
    __le32 i_size;    /* Size in bytes */  
    __le32 i_atime;   /* Access time */  
    __le32 i_ctime;   /* Creation time */  
    __le32 i_mtime;   /* Modification time */  
    __le32 i_dtime;   /* Deletion Time */  
    __le16 i_gid;     /* Low 16 bits of Group Id */  
    __le16 i_links_count; /* Links count */  
    __le32 i_blocks;  /* Blocks count */  
    __le32 i_flags;   /* File flags */
```

Inode in ext2



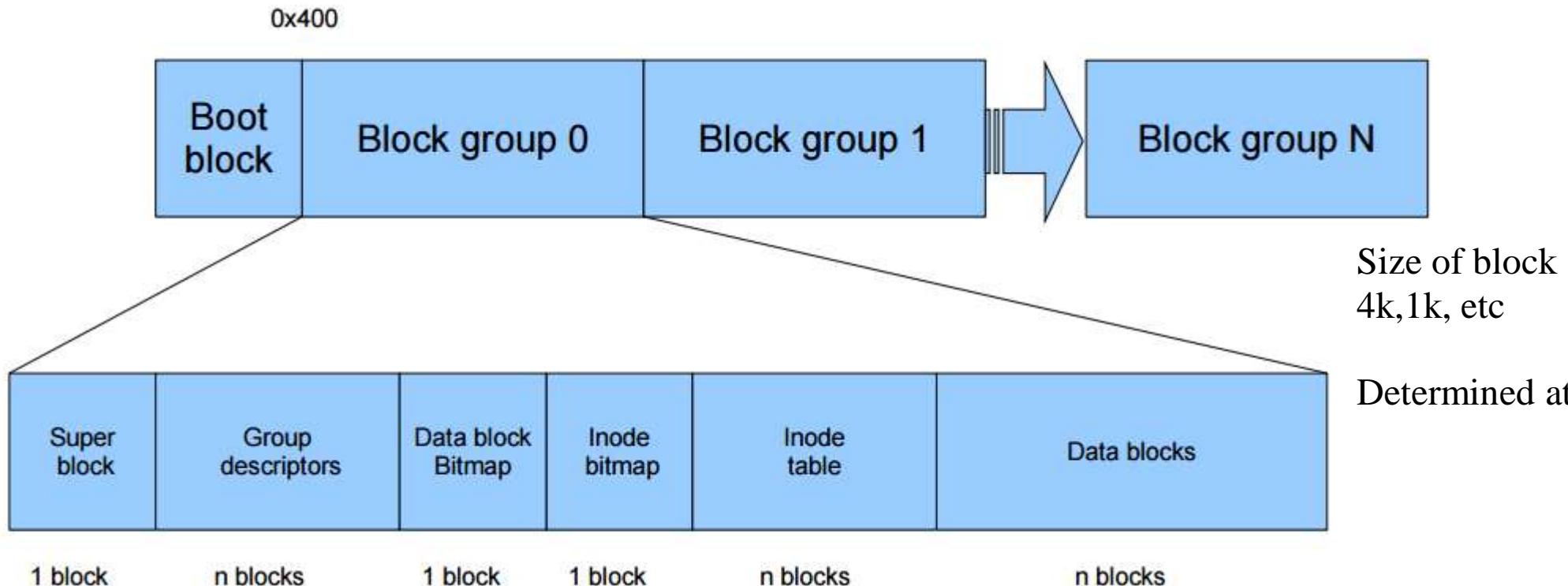
```
struct ext2_inode {  
    ...  
    union {  
        struct {  
            __le32 l_i_reserved1;  
        } linux1;  
        struct {  
            __le32 h_i_translator;  
        } hurd1;  
        struct {  
            __le32 m_i_reserved1;  
        } masix1;  
    } osd1;          /* OS dependent 1 */  
    __le32 i_block[EXT2_N_BLOCKS];/* Pointers to blocks */  
    __le32 i_generation; /* File version (for NFS) */  
    __le32 i_file_acl; /* File ACL */  
    __le32 i_dir_acl; /* Directory ACL */  
    __le32 i_faddr;   /* Fragment address */
```

```
struct ext2_inode {  
    ...  
    union {  
        struct {  
            __u8 l_i_frag; /* Fragment number */      __u8 l_i_fsize; /* Fragment size */  
            __u16 i_pad1;          __le16 l_i_uid_high; /* these 2 fields */  
            __le16 l_i_gid_high; /* were reserved2[0] */  
            __u32 l_i_reserved2;  
        } linux2;  
        struct {  
            __u8 h_i_frag; /* Fragment number */      __u8 h_i_fsize; /* Fragment size */  
            __le16 h_i_mode_high;      __le16 h_i_uid_high;  
            __le16 h_i_gid_high;  
            __le32 h_i_author;  
        } hurd2;  
        struct {  
            __u8 m_i_frag; /* Fragment number */      __u8 m_i_fsize; /* Fragment size */  
            __u16 m_pad1;          __u32 m_i_reserved2[2];  
        } masix2;  
    } osd2;      /* OS dependent 2 */
```

Ext2 FS Layout: Entries in directory's data blocks

	inode	rec_len	file_type	name_len	name				
0	21	12	1	2	.	\0	\0	\0	
12	22	12	2	2	.	.	\0	\0	
24	53	16	5	2	h	o	m	e	1 \0 \0 \0 \0
40	67	28	3	2	u	s	r	\0	
52	0	16	7	1	o	l	d	f	i 1 e \0
68	34	12	4	2	s	b	i	n	

Ext2 FS Layout



Calculations done by “mkfs” like this

- Block size = 4KB (specified to mkfs)
- Number of total blocks = size of partition / 4KB
- How to get size of partition ?
- $4\text{KB} = 4 * 1024 * 8 = 32768 \text{ bits}$
- Data Block Bitmap, Inode Bitmap are always one block
- So
- size of a group is 32,768 Blocks
- #groups = #blocks-in-partition / 32,768

```
struct ext2_super_block {
    __le32 s_inodes_count; /* Inodes count */
    __le32 s_blocks_count; /* Blocks count */
    __le32 s_r_blocks_count; /* Reserved blocks count */
    __le32 s_free_blocks_count; /* Free blocks count */
    __le32 s_free_inodes_count; /* Free inodes count */
    __le32 s_first_data_block; /* First Data Block */
    __le32 s_log_block_size; /* Block size */
    __le32 s_log_frag_size; /* Fragment size */
    __le32 s_blocks_per_group; /* # Blocks per group */
    __le32 s_frags_per_group; /* # Fragments per group */
    __le32 s_inodes_per_group; /* # Inodes per group */
    __le32 s_mtime; /* Mount time */
    __le32 s_wtime; /* Write time */
    __le16 s_mnt_count; /* Mount count */
    __le16 s_max_mnt_count; /* Maximal mount count */
    __le16 s_magic; /* Magic signature */
    __le16 s_state; /* File system state */
    __le16 s_errors; /* Behaviour when detecting errors */
```

```
struct ext2_super_block {  
...  
    __le16 s_minor_rev_level; /* minor revision level */  
    __le32 s_lastcheck;      /* time of last check */  
    __le32 s_checkinterval; /* max. time between checks */  
    __le32 s_creator_os;    /* OS */  
    __le32 s_rev_level;     /* Revision level */  
    __le16 s_def_resuid;   /* Default uid for reserved blocks */  
    __le16 s_def_resgid;   /* Default gid for reserved blocks */  
    __le32 s_first_ino;    /* First non-reserved inode */  
    __le16 s_inode_size;   /* size of inode structure */  
    __le16 s_block_group_nr; /* block group # of this superblock */  
    __le32 s_feature_compat; /* compatible feature set */  
    __le32 s_feature_incompat; /* incompatible feature set */  
    __le32 s_feature_ro_compat; /* readonly-compatible feature set */  
    __u8 s_uuid[16]; /* 128-bit uuid for volume */  
    char s_volume_name[16]; /* volume name */  
    char s_last_mounted[64]; /* directory where last mounted */  
    __le32 s_algorithm_usage_bitmap; /* For compression */
```

```
struct ext2_super_block {  
...  
    __u8    s_prealloc_blocks; /* Nr of blocks to try to preallocate*/  
    __u8    s_prealloc_dir_blocks; /* Nr to preallocate for dirs */  
    __u16   s_padding1;  
/*  
 * Journaling support valid if EXT3_FEATURE_COMPAT_HAS_JOURNAL set.  
 */  
    __u8    s_journal_uuid[16]; /* uuid of journal superblock */  
    __u32   s_journal_inum;    /* inode number of journal file */  
    __u32   s_journal_dev;    /* device number of journal file */  
    __u32   s_last_orphan;    /* start of list of inodes to delete */  
    __u32   s_hash_seed[4];    /* HTREE hash seed */  
    __u8    s_def_hash_version; /* Default hash version to use */  
    __u8    s_reserved_char_pad;  
    __u16   s_reserved_word_pad;  
    __le32  s_default_mount_opts;  
    __le32  s_first_meta_bg;   /* First metablock block group */  
    __u32   s_reserved[190];   /* Padding to the end of the block */
```

```
struct ext2_group_desc
{
    __le32 bg_block_bitmap;      /* Blocks bitmap block */
    __le32 bg_inode_bitmap;     /* Inodes bitmap block */
    __le32 bg_inode_table;     /* Inodes table block */
    __le16 bg_free_blocks_count; /* Free blocks count */
    __le16 bg_free_inodes_count; /* Free inodes count */
    __le16 bg_used_dirs_count; /* Directories count */
    __le16 bg_pad;
    __le32 bg_reserved[3];
};
```

Traversal / path-name resolution

//resolving /a/b

Let's see a program to read superblock of an ext2 file system.

Synchronization

My formulation

- OS = data structures + synchronization
- Synchronization problems make writing OS code challenging
- Demand exceptional coding skills

Race problem

```
long c = 0, c1 = 0, c2 = 0,    int main() {  
run = 1;  
  
void *thread1(void *arg)  
{  
    while(run == 1) {  
        c++;  
        c1++;  
    }  
}  
    pthread_t th1, th2;  
    pthread_create(&th1,  
NULL, thread1, NULL);  
    pthread_create(&th2,  
NULL, thread2, NULL);  
    //fprintf(stdout,  
    "Ending main\n");  
    sleep(2);  
    run = 0;
```

Race problem

- On earlier slide
- Value of **c** should be equal to $c1 + c2$, but it is not!
- Why?
- There is a “race” between **thread1** and **thread2** for updating the variable **c**
- **thread1** and **thread2** may get scheduled in any order and *interrupted* any point in time
- The changes to **c** are not atomic!
- What does that mean?

Race problem

- C++, when converted to assembly code, could be

- mov c, r1**

- add r1, 1**

- mov r1, c**

- Now following sequence of instructions is possible among thread1 and thread2

- thread1: mov c, r1**

- thread2: mov c, r1**

- thread1: add r1, 1**

- thread1: mov r1, c**

Races: reasons

- **Interruptible kernel**
- If entry to kernel code does not disable interrupts, then modifications to any kernel data structure can be left incomplete
- This introduces concurrency
- Multiprocessor systems
- On SMP systems: memory is shared, kernel and process code run on all processors
- Same variable can be updated parallelly (not concurrently)

What about non-interruptible kernel or

Critical Section problem

```
do {  
    entry section  
    critical section  
    exit section  
    remainder section  
} while (TRUE);
```

Figure 6.1 General structure of a typical process P .

Critical Section Problem

- Consider system of n processes {p₀, p₁, ... p_{n-1}}
- Each process has critical section segment of code
- Process may be changing common variables, updating table, writing file, etc
- When one process in critical section, no other may be in its critical section
- Critical section problem is to design protocol to solve this
- Each process must ask permission to enter critical section in entry section, may follow critical section with exit section

Expected solution characteristics

□ 1. Mutual Exclusion

- If process P_i is executing in its critical section, then no other processes can be executing in their critical sections

□ 2. Progress

- If no process is executing in its critical section and there exist some processes that wish to enter their critical section, then the selection of the processes that will enter the critical section next cannot be postponed indefinitely

□ 3. Bounded Waiting

suggested solution - 1

```
int flag = 1;
```

```
void *thread1(void *arg)
```

```
{
```

```
    while(run == 1) {
```

```
        while(flag == 0)
```

```
        ;
```

```
        flag = 0;
```

```
        c++;
```

```
        flag = 1;
```

- What's wrong here?

- Assumes that

- while(flag ==) ; flag = 0

- will be atomic

suggested solution - 2

```
int flag = 0;  
  
void *thread1(void *arg) {  
    while(run == 1) {  
        if(flag)  
            c++;  
        else  
            continue;  
        c1++;  
    }  
}  
  
void *thread2(void *arg) {  
    while(run == 1) {  
        if(!flag)  
            c++;  
        else  
            continue;  
        c2++;  
        flag = 1;  
    }  
}
```

Peterson's solution

- Two process solution
- Assume that the LOAD and STORE instructions are atomic; that is, cannot be interrupted
- The two processes share two variables:
`int turn;`
`Boolean flag[2]`
- The variable `turn` indicates whose turn it is to enter the critical section
- The flag array is used to indicate if a process is ready to enter the critical section. `flag[i] = true`

Peterson's solution

```
do {  
    flag[i] = TRUE;  
    turn = j;  
    while (flag[j] && turn == j)  
        ;  
    critical section  
    flag[i] = FALSE;  
    remainder section  
} while (TRUE);
```

- Provable that
 - Mutual exclusion is preserved
 - Progress requirement is satisfied
 - Bounded-waiting requirement is met

Hardware solution – the one actually implemented

- Many systems provide hardware support for critical section code**
- Uniprocessors – could disable interrupts**
- Currently running code would execute without preemption**
- Generally too inefficient on multiprocessor systems**
- Operating systems using this not broadly scalable**
- Modern machines provide special atomic hardware instructions**
- Atomic = non-interruptable**

Solution using test-and-set

```
lock = false; //global  
  
do {  
    while ( TestAndSet (&lock ) )  
        ; // do nothing  
    // critical section  
    lock = FALSE;  
    // remainder section  
} while (TRUE);
```

Definition:

```
boolean TestAndSet (boolean  
*target)  
{  
    boolean rv = *target;  
    *target = TRUE;  
    return rv;  
}
```

Solution using swap

```
lock = false; //global  
  
do {  
    key = true  
    while ( key == true))  
        swap(&lock, &key)  
        // critical section  
    lock = FALSE;  
    // remainder section  
} while (TRUE);
```

Spinlock

- A lock implemented to do ‘busy-wait’
- Using instructions like T&S or Swap

- As shown on earlier slides

- **spinlock(int *lock){**
 While(test-and-set(lock))

;

- **}**

- **spinunlock(lock *lock) {**
 ***lock = false;**

Bounded wait M.E. with T&S

```
do {
    waiting[i] = TRUE;
    key = TRUE;
    while (waiting[i] && key)
        key = TestAndSet(&lock);
    waiting[i] = FALSE;
        // critical section
    j = (i + 1) % n;
    while ((j != i) && !waiting[j])
        j = (j + 1) % n;
    if (j == i)
        lock = FALSE;
    else
        waiting[j] = FALSE;
        // remainder section
} while (TRUE);
```

sleep-locks

- Spin locks result in busy-wait
- CPU cycles wasted by waiting processes/threads
- Solution – threads keep waiting for the lock to be available
 - Move thread to wait queue
 - The thread holding the lock will wake up one of them

Sleep locks/mutexes

```
//ignore syntactical issues
typedef struct mutex {
    int islocked;
    int spinlock;
    waitqueue q;
}mutex;
wait(mutex *m) {
    spinlock(m->spinlock);
    while(m->islocked)
        Block(mutex *m,
              spinlock *sl) {
            currprocess->state =
                WAITING
            move current process to
            m->q
            spinunlock(sl);
            Sched();
            spinlock(sl);
        }
}
```

Some thumb-rules of spinlocks

- ❑ Never block a process holding a spinlock !

- ❑ Typical code:

```
while(condition)
```

```
{ Spin-unlock()
```

```
Schedule()
```

```
Spin-lock()
```

```
}
```

- ❑ Hold a spin lock for only a short duration of time

- ❑ Spinlocks are preferable on multiprocessor systems

Locks in xv6 code

struct spinlock

// Mutual exclusion lock.

struct spinlock {

 uint locked; // Is the lock held?

// For debugging:

 char *name; // Name of lock.

 struct cpu *cpu; // The cpu holding the lock.

 uint pcs[10]; // The call stack (an array of
 program counters)

spinlocks in xv6 code

```
struct {  
    struct spinlock lock;  
    struct buf buf[NBUF];  
    struct buf head;  
} beache;  
  
struct {  
    struct spinlock lock;  
    struct file file[NFILE];  
} ftable;  
  
struct {  
    static struct spinlock  
    idelock;  
  
    struct {  
        struct spinlock lock;  
        int use_lock;  
        struct run *freelist;  
    } kmem;  
  
    struct log {  
        struct spinlock lock;  
        ...}  
}
```

```
static inline uint  
xchg(volatile uint *addr,  
      uint newval)  
{  
    uint result;  
    // The + in "+m"  
    // denotes a read-modify-  
    // write operand.  
    asm volatile("lock;  
xchgl %0, %1" :  
               "+m" (*addr),  
               "=a" (result) :  
               "1" (newval) :
```

Spinlock in xv6

```
void acquire(struct  
spinlock *lk)  
{  
    pushcli(); // disable  
    // interrupts to avoid  
    // deadlock.  
    // The xchg is atomic.  
    while(xchg(&lk->locked, 1) != 0)  
        ;
```

Void acquire(struct spinlock

***lk)**

{

**pushcli(); // disable
interrupts to avoid deadlock.**

if(holding(lk))

panic("acquire");

.....

void pushcli(void)

{

int eflags;

spinlocks

- Pushcli() - disable
interrupts on that
processor**

- One after another many
acquire() can be called
on different spinlocks**

- Keep a count of them in
mycpu()->ncli**

void

release(struct spinlock
*lk)

{

...

 asm volatile("movl \$0,
%0" : "+m" (lk->locked)
:);

 popcli();

}

.

Void popcli(void)

spinlocks

- Popcli()

- Restore interrupts if last popcli() call restores ncli to 0 & interrupts were enabled before pushcli() was called

spinlocks

- Always disable interrupts while acquiring spinlock
- Suppose **iderw** held the **idelock** and then got interrupted to run **ideintr**.
- **Ideintr** would try to lock **idelock**, see it was held, and wait for it to be released.
- In this situation, **idelock** will never be released
- Deadlock
- General OS rule: if a spin-lock is used by an interrupt handler, a processor must never hold that lock with interrupts enabled

sleeplocks

- Sleeplocks don't spin. They move a process to a wait-queue if the lock can't be acquired
- XV6 approach to “wait-queues”
- Any memory address serves as a “wait channel”
 - The sleep() and wakeup() functions just use that address as a ‘condition’
 - There are no per condition process queues! Just one global queue of processes used for scheduling, sleep, wakeup etc. --> Linear search everytime !
 - costly, but simple

void

**sleep(void *chan, struct
spinlock *lk)**

{

**struct proc *p =
myproc();**

....

**if(lk != &ptable.lock){
 acquire(&ptable.lock);**

release(lk);

}

p->chan = chan;

sleep()

**□ At call must hold lock
on the resource on which
you are going to sleep**

**□ since you are going to
change p-> values & call
sched(), hold ptable.lock
if not held**

**□ p->chan = given address
remembers on which
condition the process is
waiting**

Calls to sleep() : examples of “chan” (output from cscope)

0 console.c consoleread
251 sleep(&input.r, &cons.lock);

2 ide.c iderw 169
sleep(b, &idelock);

3 log.c begin_op 131
sleep(&log, &log.lock);

6 pipe.c piperead 111
sleep(&p->nread, &p->lock);

7 proc.c wait
317 sleep(curproc,
&ptable.lock);

8 sleeplock.c
acquiresleep 28 sleep(lk,
&lk->lk);

9 sysproc.c
sys_sleep 74
sleep(&ticks,
&tickslock);

```
void wakeup(void *chan)
{
    acquire(&ptable.lock);
    wakeup1(chan);
    release(&ptable.lock);
}

static void wakeup1(void
*chan)
{
    struct proc *p;
```

Wakeup()

- Acquire ptable.lock since you are going to change ptable and p-> values
- just linear search in process table for a process where p->chan is given address
- Make it runnable

sleeplock

```
// Long-term locks for processes

struct sleeplock {
    uint locked;          // Is the lock held?
    struct spinlock sl; // spinlock protecting this sleep
    lock

// For debugging:
    char *name;          // Name of lock.
    int pid;             // Process holding lock
```

Sleeplock acquire and release

void

acquiresleep(struct
sleeplock *lk)

{

acquire(&lk->lk);

while (lk->locked) {

/ Abhijit: interrupts
are not disabled in sleep
!*/*

sleep(lk, &lk->lk);

void

releasesleep(struct
sleeplock *lk)

{

acquire(&lk->lk);

 lk->locked = 0;

 lk->pid = 0;

wakeup(lk);

release(&lk->lk);

Where are sleeplocks used?

- struct buf
 - waiting for I/O on this buffer
- struct inode
 - waiting for I/o to this inode

□ Just two !

Sleeplocks issues

- sleep-locks support yielding the processor during their critical sections.
- This property poses a design challenge:
 - if thread T1 holds lock L1 and has yielded the processor (waiting for some other condition),
 - and thread T2 wishes to acquire L1,
 - we have to ensure that T1 can execute
 - while T2 is waiting so that T1 can release L1.
- T2 can't use the spin-lock acquire function here: it spins with interrupts turned off, and that would prevent T1 from running.

More needs of synchronization

- Not only critical section problems
- Run processes in a particular order
- Allow multiple processes read access, but only one process write access
- Etc.



Semaphore

- Synchronization tool that does not require busy waiting
- Semaphore S – integer variable
- Two standard operations modify S: wait() and signal()
- Originally called P() and V()
- Less complicated
 - Can only be accessed via two indivisible (atomic) operations
 - `wait (S) {`
 - `while S <= 0`
 - ; // no-op
 - `S--;`
 - }
 - `signal (S) {`

Semaphore for synchronization

- **Counting** semaphore – integer value can range over an unrestricted domain
- **Binary** semaphore – integer value can range only between 0 and 1; can be simpler to implement

Also known as **mutex locks**

- Can implement a counting semaphore **S** as a binary semaphore
- Provides mutual exclusion

- **Semaphore mutex;** // initialized to 1
- **do {**
- **wait (mutex);**
- **// Critical Section**
- **signal (mutex);**
- **// remainder section**

Different uses of semaphores

For mutual exclusion

/*During initialization*/

semaphore sem;

initsem (&sem, 1);

/* On each use*/

P (&sem);

Use resource;

V (&sem);

Event-wait

/* During initialization */

semaphore event;

initsem (&event, 0); /* probably at boot time */

/* Code executed by thread that must wait on event */

P (&event); /* Blocks if event has not occurred */

/* Event has occurred */

V (&event); /* So that another thread may wake up */

Control countable resources

/* During initialization */

semaphore counter;

initsem (&counter, resourceCount);

/* Code executed to use the resource */

P (&counter); /* Blocks until resource is available */

Use resource; /* Guaranteed to be available now */

V (&counter); /* Release the resource */

Semaphore implementation

```
Wait(sem *s) {  
    while(s <=0)  
        block(); // could be  
        ";"  
    s--;  
}  
  
signal(sem *s) {  
    s++;  
}
```

- Left side – expected behaviour
- Both the wait and signal should be atomic.
- This is the semantics of the semaphore.

Semaphore implementation? - 1

```
struct semaphore {  
    int val;  
    spinlock sl;  
};  
  
sem_init(semaphore *s,  
int initval) {  
    s->val = initval;  
    s->sl = 0;  
}
```

```
signal(semaphore *s) {  
    spinlock(*(s->sl));  
    (s->val)++;  
    spinunlock(*(s->sl));  
}
```

- suppose 2 processes
trying wait.

val = 1;

Th1: spinlock

Th2: spinlock

Semaphore implementation? - 2

```
struct semaphore {  
    int val;  
    spinlock sl;  
};  
  
sem_init(semaphore *s,  
int initval) {  
    s->val = initval;  
    s->sl = 0;  
}  
  
wait(semaphore *s) {  
    spinlock(&(s->sl));  
    while(s->val <=0) {  
        spinunlock(&(s->sl));  
        spinlock(&(s->sl));  
    }  
    (s->val)--;  
    spinunlock(&(s->sl));  
}
```

Semaphore implementation? - 3, idea

```
struct semaphore {  
    int val;  
    spinlock sl;  
};  
  
sem_init(semaphore *s,  
int initval) {  
    s->val = initval;  
    s->sl = 0;  
}  
  
wait(semaphore *s) {  
    spinlock(&(s->sl));  
    while(s->val <=0) {  
        Block();  
    }  
    (s->val)--;  
    spinunlock(&(s->sl));  
}  
  
signal(seamphore *s) {
```

Semaphore implementation? - 3a

```
struct semaphore {  
    int val;  
    spinlock sl;  
    list l;  
};  
  
sem_init(semaphore *s,  
        int initval) {  
    s->val = initval;  
    s->sl = 0;  
}  
  
wait(semaphore *s) {  
    spinlock(&(s->sl));  
    while(s->val <=0) {  
        spinunlock(&(s->sl));  
        block(s);  
    }  
    (s->val)--;  
    spinunlock(&(s->sl));  
}
```

Semaphore implementation? - 3b

```
struct semaphore {  
    int val;  
    spinlock sl;  
    list l;  
};  
  
sem_init(semaphore *s,  
        int initval) {  
    s->val = initval;  
    s->sl = 0;  
  
    wait(semaphore *s) {  
        spinlock(&(s->sl));  
        while(s->val <= 0) {  
            block(s);  
        }  
        (s->val)--;  
        spinunlock(&(s->sl));  
    }  
  
    signal(semaphore *s) {
```

Semaphore implementation? - 3c

```
struct semaphore {  
    int val;  
    spinlock sl;  
    list l;  
};  
  
sem_init(semaphore *s,  
        int initval) {  
    s->val = initval;  
    s->sl = 0;
```

```
wait(semaphore *s) {  
    spinlock(&(s->sl)); // A  
    while(s->val <=0) {  
        block(s);  
        spinlock(&(s->sl)); // B  
    }  
    (s->val)--;  
    spinunlock(&(s->sl));
```

Semaphore Implementation

- Must guarantee that no two processes can execute `wait()` and `signal()` on the same semaphore at the same time
- Thus, implementation becomes the critical section problem where the wait and signal code are placed in the critical section
- Could now have busy waiting in critical section implementation
- But implementation code is short
- Little busy waiting if critical section rarely occupied

Semaphore in Linux

```
struct semaphore {  
    raw_spinlock_t    lock;  
    unsigned int      count;  
    struct list_head  wait_list;  
};  
  
static noinline void __sched  
__down(struct semaphore *sem)  
{  
    __down_common(sem,  
    TASK_UNINTERRUPTIBLE,  
    MAX_SCHEDULE_TIMEOUT);  
}  
  
void down(struct semaphore *sem)  
{  
    unsigned long flags;  
  
    raw_spin_lock_irqsave(&sem->lock, flags);  
    if (likely(sem->count > 0))  
        sem->count--;  
    else  
        __down(sem);  
    raw_spin_unlock_irqrestore(&sem->lock, flags);  
}
```

Semaphore in Linux

```
static inline int __sched
__down_common(struct
semaphore *sem, long state, long
timeout)
```

```
{
```

```
    struct task_struct *task =
current;
```

```
    struct semaphore_waiter
waiter;
```

```
    list_add_tail(&waiter.list,
&sem->wait_list);
```

```
    waiter.task = task;
```

```
    waiter.up = false;
```

```
    for (;;) {
        if (signal_pending_state(state,
task))
            goto interrupted;
        if (unlikely(timeout <= 0))
            goto timed_out;
        __set_task_state(task, state);
        raw_spin_unlock_irq(&sem->lock);
        timeout =
schedule_timeout(timeout);
        raw_spin_lock_irq(&sem->lock);
        if (waiter.up)
```

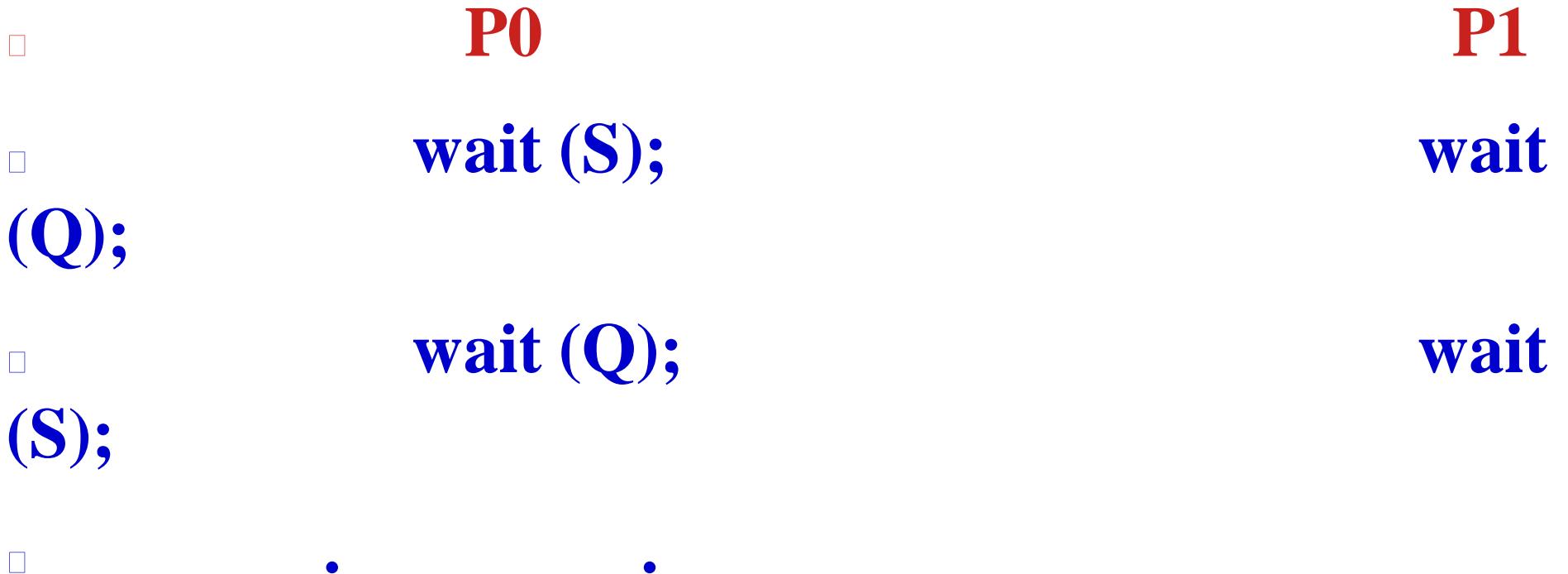
Drawbacks of semaphores

- Need to be implemented using lower level primitives like spinlocks
- Context-switch is involved in blocking and signaling – time consuming
- Can not be used for a short critical section

Deadlocks

Deadlock

- ❑ two or more processes are waiting indefinitely for an event that can be caused by only one of the waiting processes
- ❑ Let S and Q be two semaphores initialized to 1



Example of deadlock

- Let's see the pthreads program : **deadlock.c**
- Same program as on earlier slide, but with **pthread_mutex_lock();**

Non-deadlock, but similar situations

- Starvation – indefinite blocking
- A process may never be removed from the semaphore queue in which it is suspended

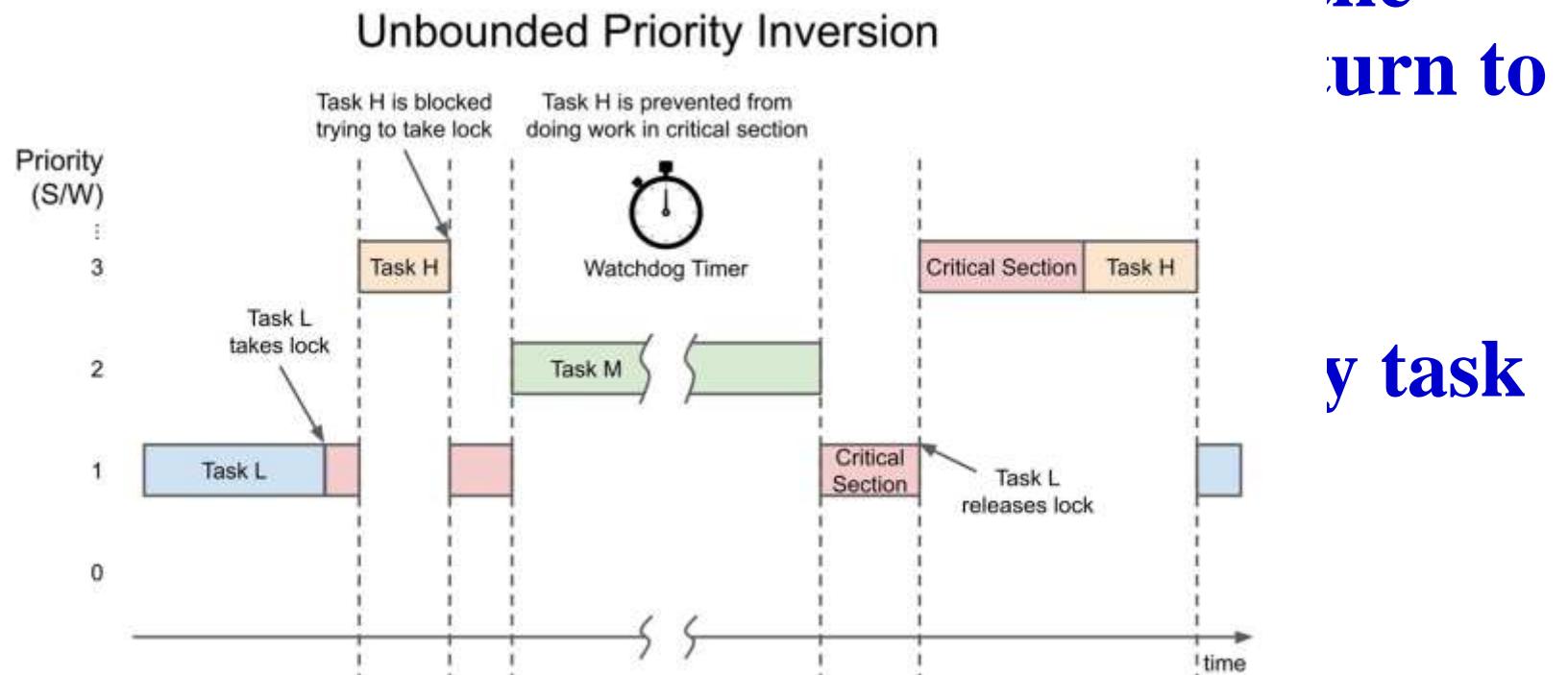
Non-deadlock, but similar situations

□ Priority Inversion

□ Scheduling problem when lower-priority process holds a lock needed by higher-priority process (so it can not pre-empt lower priority process), and a medium priority process (that does not need the lock) runs to higher priority

lock) p
higher

□ Solved
tempo
to high



Livelock

- Similar to deadlock, but processes keep doing ‘useless work’
- E.g. two people meet in a corridor opposite each other
 - Both move to left at same time
 - Then both move to right at same time
 - Keep Repeating!
- No process able to progress, but each doing ‘some work’ (not sleeping/waiting), state keeps changing

Livelock example

```
#include <stdio.h>
#include <pthread.h>

struct person {
    int otherid;
    int otherHungry;
    int myid;
};

int main() {
    pthread_t th1, th2;
    /* thread two runs in this
       function */
    int spoonWith = 1;
    void *eat(void *param)
    {
        int eaten = 0;
        struct person person=
        *(struct person *)param;
        while (!eaten) {
            if (spoonWith == myid) {
                spoonWith = otherid;
                eaten = 1;
            }
            else if (spoonWith == otherid) {
                spoonWith = myid;
                eaten = 1;
            }
            else {
                sleep(1);
            }
        }
    }
}
```

More on deadlocks

- ❑ Under which conditions they can occur?
- ❑ How can deadlocks be avoided/prevented?
- ❑ How can a system recover if there is a deadlock ?

System model for understanding deadlocks

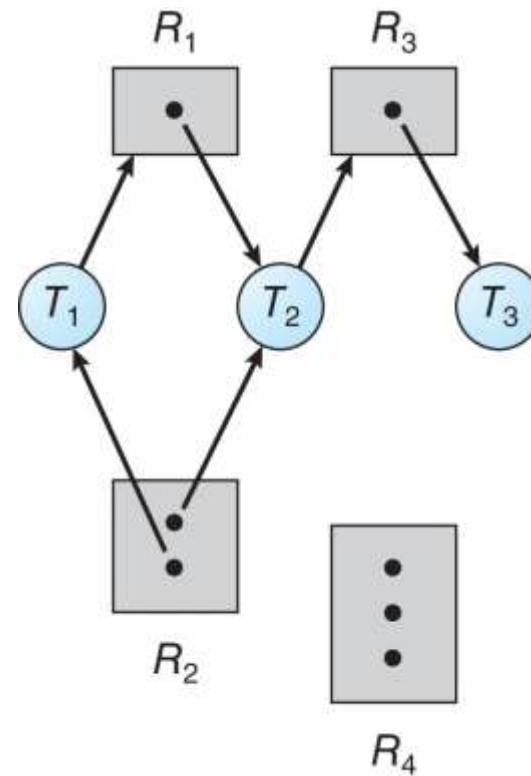
- System consists of resources
- Resource types R_1, R_2, \dots, R_m
- CPU cycles, memory space, I/O devices
- Resource: Most typically a lock, synchronization primitive
- Each resource type R_i has W_i instances.
- Each process utilizes a resource as follows:
 - request
 - use
 - release

Deadlock characterisation

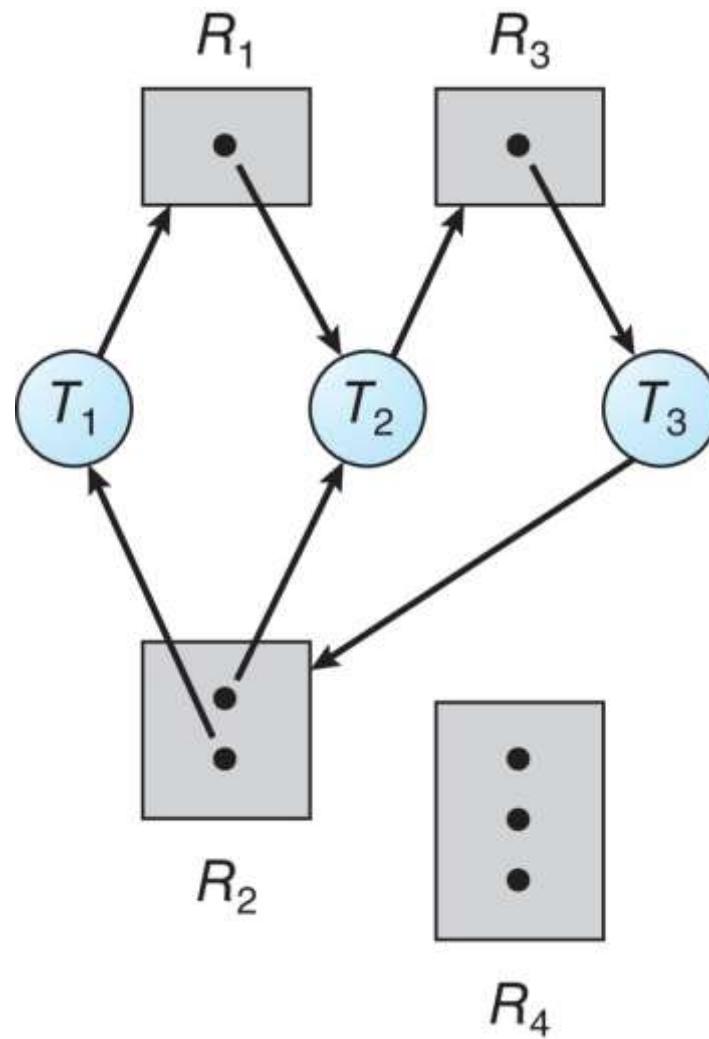
- Deadlock is possible only if ALL of these conditions are TRUE at the same time
 - Mutual exclusion: only one process at a time can use a resource
 - Hold and wait: a process holding at least one resource is waiting to acquire additional resources held by other processes
 - No preemption: a resource can be released only voluntarily by the process holding it, after that process has completed its task
 - Circular wait: there exists a set {P0, P1, ..., Pn} of

Resource Allocation Graph Example

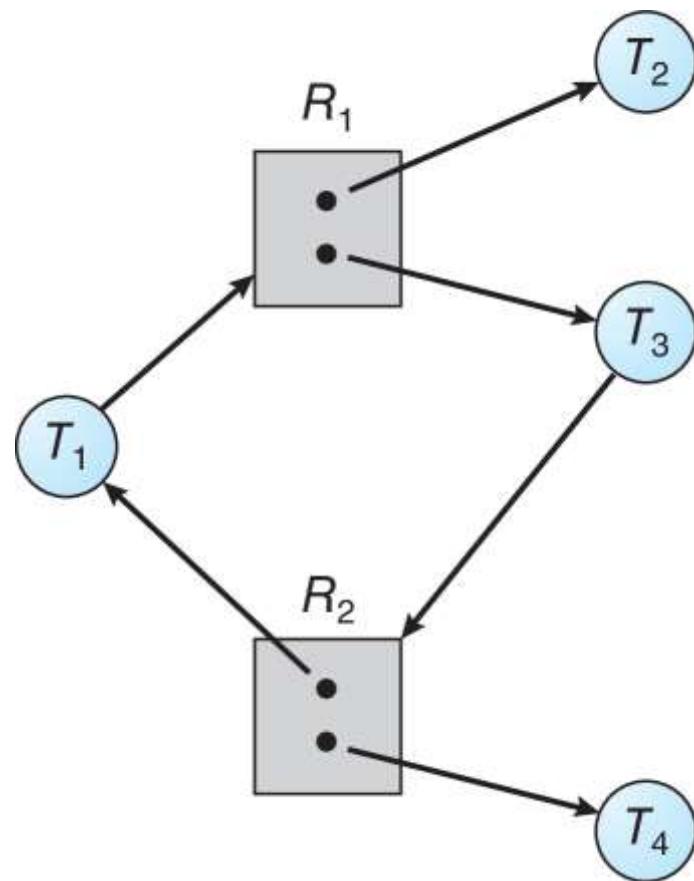
- One instance of R1
- Two instances of R2
- One instance of R3
- Three instances of R4
- T1 holds one instance of R2 and is waiting for an instance of R1
- T2 holds one instance of R1, one instance of R2, and is waiting for an instance of R3



Resource Allocation Graph with a Deadlock



Graph with a Cycle But no Deadlock



Basic Facts

- If graph contains no cycles -> no deadlock
- If graph contains a cycle :
 - if only one instance per resource type, then deadlock
 - if several instances per resource type, possibility of deadlock

Methods for Handling Deadlocks

- Ensure that the system will never enter a deadlock state:
 - 1) Deadlock prevention
 - 2) Deadlock avoidance
 - 3) Allow the system to enter a deadlock state and then recover
 - 4) Ignore the problem and pretend that deadlocks never occur in the system.

(1) Deadlock Prevention

- Invalidate one of the four necessary conditions for deadlock:
 - Mutual Exclusion – not required for sharable resources (e.g., read-only files); must hold for non-sharable resources
 - Hold and Wait – must guarantee that whenever a process requests a resource, it does not hold any other resources
 - Require process to request and be allocated all its resources before it begins execution, or allow process to request resources only when the process

(1) Deadlock Prevention (Cont.)

- ❑ **No Preemption:**

- ❑ If a process that is holding some resources requests another resource that cannot be immediately allocated to it, then all resources currently being held are released

- ❑ Preempted resources are added to the list of resources for which the process is waiting

- ❑ Process will be restarted only when it can regain its old resources, as well as the new ones that it is requesting

- ❑ **Circular Wait:**

(1) Deadlock prevention: Circular Wait

- Invalidating the circular wait condition is most common.

- Simply assign each resource (i.e., mutex locks) a unique number.

- Resources must be acquired in order.

- If:
first_mutex is mapped to order 1

```
/* thread_one runs in this function */
void *do_work_one(void *param)
{
    pthread_mutex_lock(&first_mutex);
    pthread_mutex_lock(&second_mutex);
    /**
     * Do some work
     */
    pthread_mutex_unlock(&second_mutex);
    pthread_mutex_unlock(&first_mutex);

    pthread_exit(0);
}

/* thread_two runs in this function */
void *do_work_two(void *param)
{
    pthread_mutex_lock(&second_mutex);
    pthread_mutex_lock(&first_mutex);
    /**
     * Do some work
     */
    pthread_mutex_unlock(&first_mutex);
    pthread_mutex_unlock(&second_mutex);

    pthread_exit(0);
}
```

(1) Preventing deadlock: cyclic wait

- Locking hierarchy : Highly preferred technique in kernels
- Decide an ordering among all ‘locks’
- Ensure that on ALL code paths in the kernel, the locks are obtained in the decided order!
- Poses coding challenges!
- A key differentiating factor in kernels
 - Do not look at only the current lock being taken, look at all the locks the code may be holding at any given point in code!

(1) Prevention in Xv6: Lock Ordering

- lock on the directory, a lock on the new file's inode, a lock on a disk block buffer, idelock, and ptable.lock.

(2) Deadlock avoidance

- Requires that the system has some additional a priori information available
- Processes declare resources they want, BEFORE-hand
- Resources are always allocated by an ALLOCATOR algorithm
- It can predict if a deadlock can happen

(2) Deadlock avoidance

□ Please see: concept of safe states, unsafe states, Banker's algorithm

(3) Deadlock detection and recovery

- How to detect a deadlock in the system?
- The Resource-Allocation Graph is a graph. Need an algorithm to detect cycle in a graph.
- How to recover?
- Abort all processes or abort one by one?
- Which processes to abort?
- Priority ?
- Time spent since forked()?
- Resources used?
- Resources needed?

“Condition”

Synchronization Tool

What is condition variable?

- A variable with a sleep queue
- Threads can sleep on it, and wake-up all remaining

Struct condition {

 Proc *next

 Proc *prev

 Spinlock *lock

}

Different variables of this type can be used as different ‘conditions’

Code for condition variables

//Spinlock s is held
before calling wait

```
void wait (condition *c,  
spinlock_t *s)
```

```
(  
    spin_lock (&c->listLock);
```

add self to the linked
list;

```
    spin_unlock (&c-
```

```
void do_signal (condition  
*c)
```

```
/*Wakeup one thread  
waiting on the condition*/
```

```
{
```

```
    spin_lock (&c->listLock);
```

remove one thread from
linked list, if it is
nonempty;

Semaphore implementation using condition variables?

□ Is this possible?

□ Can we try it?

```
typedef struct semaphore {
```

```
    //something
```

```
    condition c;
```

```
}semaphore;
```

□ Now write code for semaphore P() and V()

Classical Synchronization Problems

Bounded-Buffer Problem

- Producer and consumer processes
- N buffers, each can hold one item
- Producer produces ‘items’ to be consumed by consumer , in the bounded buffer
- Consumer should wait if there are no items
- Producer should wait if the ‘bounded buffer’ is full

Bounded-Buffer Problem: solution with semaphores

- **Semaphore mutex initialized to the value 1**
- **Semaphore full initialized to the value 0**
- **Semaphore empty initialized to the value N**

Bounded-buffer problem

The structure of the producer process

```
do {  
    // produce an item in nextp  
    wait (empty);  
    wait (mutex);  
    // add the item to the buffer  
    signal (mutex);
```

The structure of the Consumer process

```
do {  
    wait (full);  
    wait (mutex);  
    // remove an item from  
    // buffer to nextc  
    signal (mutex);
```

Bounded buffer problem

- ❑ Example : pipe()
- ❑ Let's see code of pipe in xv6 – a solution using sleeplocks

Readers-Writers problem

- A data set is shared among a number of concurrent processes
- Readers – only read the data set; they do not perform any updates
- Writers – can both read and write
- Problem – allow multiple readers to read at the same time
- Only one single writer can access the shared data at the same time
- Several variations of how readers and writers are treated – all involve priorities

The structure of a writer process

```
do {  
    wait (wrt) ;  
    // writing is performed  
    signal (wrt) ;  
} while (TRUE);
```

Readers-Writers problem

```
The structure of a reader process  
do {  
    wait (mutex) ;  
    readcount ++ ;  
    if (readcount == 1)  
        wait (wrt) ;  
    signal (mutex)  
    // reading is performed  
    wait (mutex) ;  
    readcount -- ;  
    if (readcount == 0)  
        signal (wrt) ;  
    signal (mutex) ;  
} while (TRUE);
```

Readers-Writers Problem Variations

- First variation – no reader kept waiting unless writer has permission to use shared object
- Second variation – once writer is ready, it performs write asap
- Both may have starvation leading to even more variations
- Problem is solved on some systems by kernel providing reader-writer locks

Reader-write lock

- A lock with following operations on it
 - **Lockshared()**
 - **Unlockshared()**
 - **LockExcl()**
 - **UnlockExcl()**
- Possible additions
 - **Downgrade()** -> from excl to shared
 - **Upgrade()** -> from shared to excl

Code for reader-writer locks

```
struct rwlock {  
    int nActive; /* num of  
active readers, or -1 if a  
writer is active */  
  
    int nPendingReads;  
    int nPendingWrites;  
    spinlock_t sl;  
  
    condition canRead;  
    condition canWrite;
```

```
void lockShared (struct  
rwlock *r)  
(  
    spin_lock (&r->sl);  
    r->nPendingReads++;  
    if (r->nPendingWrites  
    > 0)  
        wait (&r->canRead,  
&r->sl ); /*don't starve  
writers */
```

Code for reader-writer locks

```
void unlockShared  
(struct rwlock *r)
```

```
{
```

```
    spin_lock (&r->sl);
```

```
    r->nActive--;
```

```
    if (r->nActive == 0) {
```

```
        spin_unlock (&r->sl);
```

```
        do signal (&r->canWrite);
```

```
void lockExclusive  
(struct rwlock *r)
```

```
(
```

```
    spin_lock (&r->sl);
```

```
    r->nPendingWrltes++;
```

```
    while (r->nActive)
```

```
        wait (&r->canWrite,  
&r->sl);
```

```
    r->nPendingWrites--;
```

Code for reader-writer locks

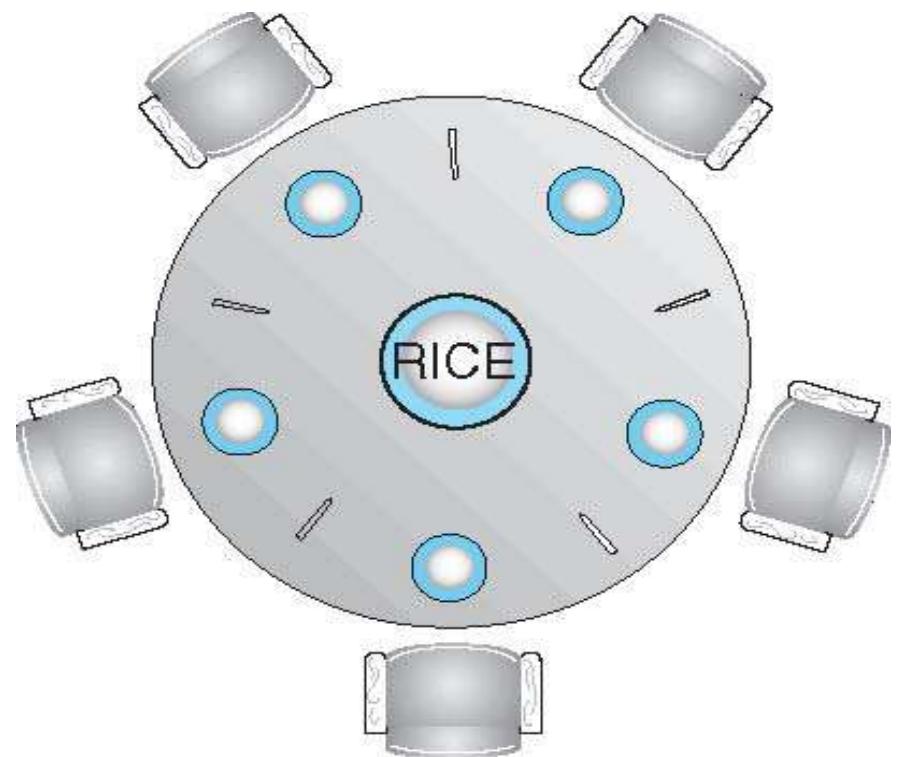
```
void unlockExclusive  
(struct rwlock *r){  
  
    boolean t  
    wakeReaders;  
  
    spin_lock (&r->sl);  
  
    r->nActive = 0;  
  
    wakeReaders = (r->nPendingReads != 0);  
  
    spin_unlock (&r->sl);  
  
    if (wakeReaders)
```

Try writing code for
downgrade and upgrade

Try writing a reader-
writer lock using
semaphores!

Dining-Philosophers Problem

- Philosophers spend their lives thinking and eating
- Don't interact with their neighbors, occasionally try to pick up 2 chopsticks (one at a time) to eat from bowl
- Need both to eat, then release both when done
- In the case of 5 philosophers



Dining philosophers: One solution

The structure of Philosopher i:

```
do {  
    wait ( chopstick[i] );  
    wait ( chopStick[ (i + 1) % 5] );  
    // eat  
    signal ( chopstick[i] );  
    signal (chopstick[ (i + 1) % 5] );  
    // think  
} while (TRUE);
```

Dining philosophers: Possible approaches

- Allow at most four philosophers to be sitting simultaneously at the table.
- Allow a philosopher to pick up her chopsticks only if both chopsticks are available
 - to do this, she must pick them up in a critical section
- Use an asymmetric solution
 - that is, an odd-numbered philosopher picks up first her left chopstick and then her right chopstick
 - whereas an even-numbered philosopher picks up her right chopstick and then her left chopstick

Other solutions to dining philosopher's problem

- Using higher level synchronization primitives like ‘monitors’

Practical Problems

Lost Wakeup problem

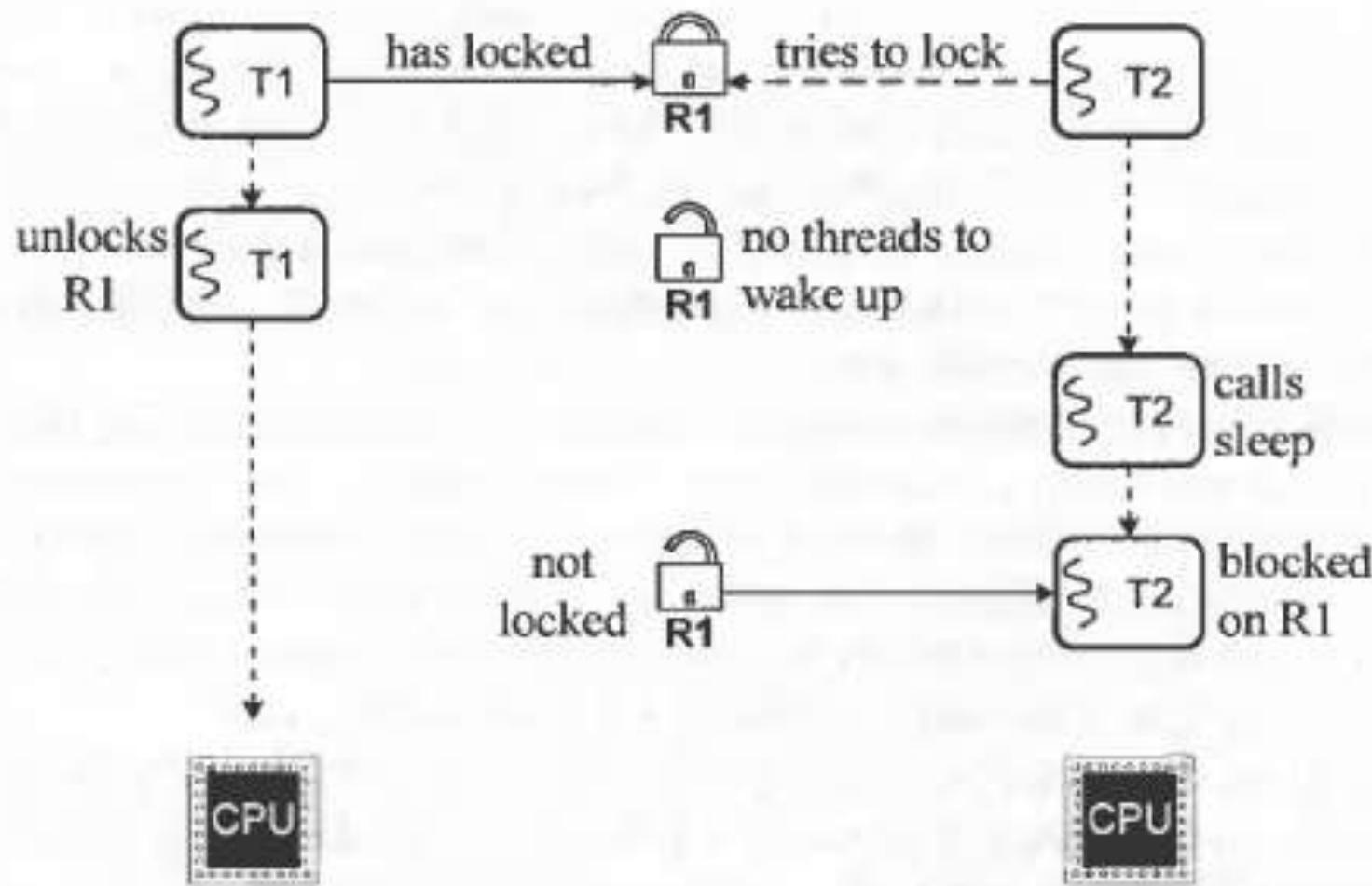


Figure 7-6. The lost wakeup problem.

Lost Wakeup problem

- The sleep/wakeup mechanism does not function correctly on a multiprocessor.
- Consider a potential race:
 - Thread T1 has locked a resource R1.
 - Thread T2, running on another processor, tries to acquire the resource, and finds it locked.
 - T2 calls sleep() to wait for the resource.
 - Between the time T2 finds the resource locked and the time it calls sleep(), T1 frees the resource and proceeds to wake up all threads blocked on it.
 - Since T2 has not yet been put on the sleep queue, it

Thundering herd problem

- Thundering Herd problem
- On a multiprocessor, if several threads were locked the resource
- Waking them all may cause them to be simultaneously scheduled on different processors
- and they would all fight for the same resource again.
- Starvation
- Even if only one thread was blocked on the resource, there is still a time delay between its waking up and actually running.

Case Studies

Linux Synchronization

- Prior to kernel Version 2.6, disables interrupts to implement short critical sections
- Version 2.6 and later, fully preemptive
- Linux provides:
 - semaphores
 - spinlocks
 - reader-writer versions of both
- Atomic integers
 - On single-cpu system, spinlocks replaced by enabling and disabling kernel preemption

Linux Synchronization

□ Atomic variables

atomic_t is the type for atomic integer

□ Consider the variables

atomic_t counter;

int value;

<i>Atomic Operation</i>	<i>Effect</i>
atomic_set(&counter,5);	counter = 5
atomic_add(10,&counter);	counter = counter + 10
atomic_sub(4,&counter);	counter = counter - 4
atomic_inc(&counter);	counter = counter + 1
value = atomic_read(&counter);	value = 12

Pthreads synchronization

- ❑ Pthreads API is OS-independent

- ❑ It provides:

- ❑ mutex locks

- ❑ condition variables

- ❑ Non-portable extensions include:

- ❑ read-write locks

- ❑ spinlocks

Synchronization issues in xv6 kernel

Difference approaches

- Pros and Cons of locks
- Locks ensure serialization
- Locks consume time !
- Solution – 1
 - One big kernel lock
 - Too inefficient
- Solution – 2
 - One lock per variable
 - Often un-necessary, many data structures get manipulated in once place, one lock for all of them

Three types of code

- System calls code
- Can it be interruptible?
 - If yes, when?
- Interrupt handler code
 - Disable interrupts during interrupt handling or not?
 - Deadlock with iderw ! - already seen
- Process's user code
 - Ignore. Not concerned with it now.

Interrupts enabling/disabling in xv6

- Holding every spinlock disables interrupts!
- System call code or Interrupt handler code won't be interrupted if
 - The code path followed took at least once spinlock !
 - Interrupts disabled only on that processor!
- Acquire calls pushcli() before xchg()
- Release calls popcli() after xchg()

Memory ordering

- Compiler may generate machine code for out-of-order execution !
 - Processor pipelines can also do the same!
 - This often improves performance
 - Compiler may reorder 4 after 6 --> Trouble!
 - Solution: Memory
- Consider this
- 1) `l = malloc(sizeof *l);`
 - 2) `l->data = data;`
 - 3) `acquire(&listlock);`
 - 4) `l->next = list;`
 - 5) `list = l;`
 - 6) `release(&listlock);`

Lost Wakeup?

- Do we have this problem in xv6?
- Let's analyze again!
- The race in `acquiresleep()`'s call to `sleep()` and `releasesleep()`
- T1 holding lock, T2 willing to acquire lock
- Both running on different processor
- Or both running on same processor
- What happens in both scenarios?
- Introduce a T3 and T4 on each of two different processors. Now how does the scenario change?

Code of sleep()

```
if(lk != &ptable.lock){  
    acquire(&ptable.lock);  
    release(lk);  
}
```

- Why this check?
- Deadlock otherwise!
- Check: wait() calls with ptable.lock held!

Exercise question : 1

Sleep has to check lk != &ptable.lock to avoid a deadlock

Suppose the special case were eliminated by replacing

```
if(lk != &ptable.lock){  
    acquire(&ptable.lock);  
    release(lk);  
}
```

bget() problem

- bget() panics if no free buffers!
- Quite bad
- Should sleep !
- But that will introduce many deadlock problems.
Which ones ?

iget() and ilock()

- **iget() does no hold lock on inode**
- **Ilock() does**
- **Why this separation?**
- **Performance? If you want only “read” the inode, then why lock it?**
- **What if iget() returned the inode locked?**

Interesting cases in namex()

```
while((path =  
skipelem(path, name)) != 0){  
    ilock(ip);  
    if(ip->type != T_DIR){  
        iunlockput(ip);  
        return 0;  
    }  
    if(nameiparent &&  
*path == '\0'){  
        if((next = dirlookup(ip,  
name, 0)) == 0){  
            iunlockput(ip);  
            ip  
        }  
        iunlockput(ip);  
        ip  
    }  
    --> only after obtaining
```

Xv6

Interesting case of holding and releasing
ptable.lock in scheduling

One process acquires, another releases!

Giving up CPU

- A process that wants to give up the CPU
 - must acquire the process table lock `ptable.lock`
 - release any other locks it is holding
 - update its own state (`proc->state`),
 - and then call `sched()`
 - Yield follows this convention, as do sleep and exit
 - Lock held by one process P1, will be released another process P2 that starts running after `sched()`
 - remember P2 returns either in `yield()` or `sleep()`
- In both, the first thing done is releasing `ptable.lock`

Interesting race if ptable.lock is not held

- Suppose P1 calls yield()
- Suppose yield() does not take ptable.lock
- Remember yield() is for a process to give up CPU
- Yield sets process state of P1 to RUNNABLE
- Before yield's sched() calls swtch()
- Another processor runs scheduler() and runs P1 on that processor
- Now we have P1 running on both processors!
- P1 in yield taking ptable.lock prevents this

Homework

- Read the version-11 textbook of xv6
- Solve the exercises!

Operating Systems

Sem VI 2023-24

Course Introduction,

Course Plan

Evaluation Scheme

Me

.Abhijit Ashok Meenakshi

.3rd Floor, ENTC EXTN

.9422308125 (Signal, WhatsApp)

Significance

- The identifier course for “Computer Engineering”

- How is a “functional” (usable, useful) computer system built?
Answer is – essentially with OS

- The glue that binds



Answers to questions like

- Why is my computer running slow?
- Why does a system “hang” / “crash” ?
- The exact sequence of events in hardware, when I click on “close” button
- The exact sequence of events between pressing of a key, and seeing it on the screen
- How is it possible that multiple applications are “running” “at the same time”, even if you have a

This course is a must for a career in

- Networking

- Security

- Microprocessor Design

- Devops

- Cloud

- Storage, Backup

- Databases

The issue about the name of the course

.The kernel

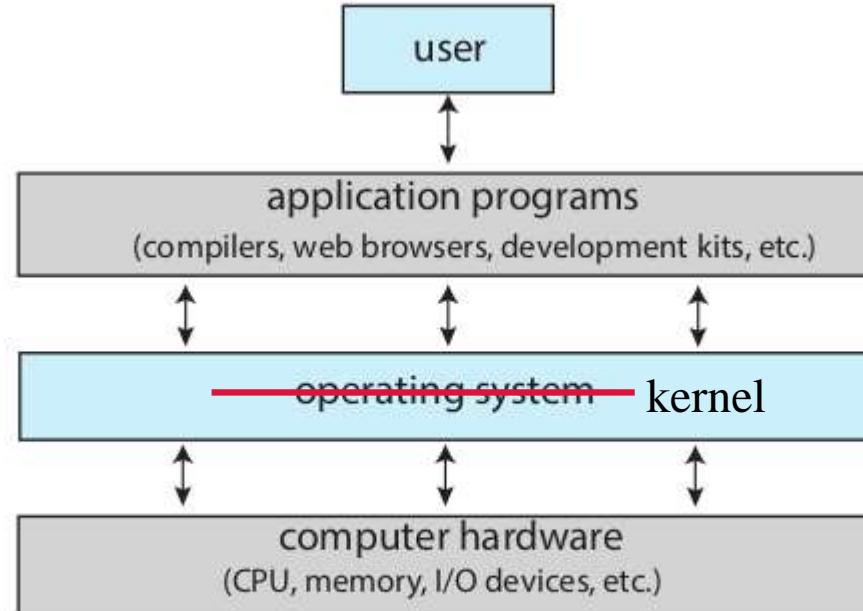


Figure 1.1 Abstract view of the components of a computer system.

OS or kernel?

- Debate on terminology
- What you use in daily life is not kernel, but
 - GUI, Shell, Applications, ...
 - One view: OS = kernel + (GUI, Shell, Libraries, System programs, Minimum Applications,...)
 - Correct, IMO!
- What we study in this course is kernel

Attendance requirement

- .75% : Institute norm

- Absence should be informed

- It will be assumed that you are on campus, except on a declared holiday

Theory course evaluation

- Total Marks: 100
- The total marks for each of the following categories will be converted to proportional maximum marks as follows:
 - Quiz-1: 15 Marks
 - Quiz-2: 15 Marks
 - Anytime quizzes: 10 Marks (may be ON campus)

Lab course evaluation

- Total Marks: 100
- The total marks for each of the following categories will be converted to proportional maximum marks as follows:
 - Mandatory Assignments: 20 Marks
 - Mid Term Exam: 30 Marks
 - Optional Assignments/Project: 50 Marks

Lab Assignments

- They are time consuming!
- You will be busy entire week (10-20 hrs per assignment)!
- Don't hesitate to submit incomplete assignments.
 - No binary (0 or full) marking!
 - Partial marks for partial work done.

Lab Pre-requisites

- GNU/Linux command line, at least 20-30 commands
- Dual boot installation of GNU/Linux
- Basic information about folder structure on GNU/Linux
- Backup your data! You are likely to loose it at least once!

The insatiable curiosity to "look inside" and "do

Textbooks

- Operating System Concepts, by Abraham Silberschatz, P B Galvin, G Gagne
 - 10th edition
- Introduction to Systems Software by Jonathan Misruda
- The C programming Language by Kernighan and Ritchie

On Plagiarism

- .The expected similarity score for each submission is 50%, unless specifically mentioned.
 - Assignments with similarity score below the announced score will be checked.
 - If the score is more than threshold, then teachers will examine the code and decide if it's plagiarism.
 - The decision of teacher will be final on deciding if any small amount of plagiarism is involved.

On Plagiarism

- Penalties
 - First instance: zero in that assignment
 - Second instance: Fail in course
- Following typical excuses will not be entertained
 - I Shared my code by mistake (Don't do mistakes!)
 - I had given access of my Moodle/email/etc account to someone and it was misused (You are

On Plagiarism

- Statistics:
 - 12 students failed OS Lab last semester.
 - Almost all of them because they plagiarised (not because they could not do assignments)
- Let's see how I check it and a list of plagiarisms detected

Course project

- Will be tracked on gitlab repo
- Your date-time-wise commits will be used to determine if you were engaged in doing it on a regular basis
 - It is assumed that you are likely to have plagiarised, if you have not worked on it regularly!
- After 2.5 months of semester, weekly updates will be taken from you about the project work

Course project

- The following are already available with your teacher for plagiarism check:
 - All the source-codes (assignments and projects) of all the students who have done this OS course in COEP !
 - All xv6 projects on the web
 - An uncanny-eye to detect different coding styles

A fair enough judgement about your abilities !

During lab hours

- You are expected
 - To have seen the recorded videos related to the current assignment, and all lectures related to it and ask doubts!
 - To show the work done/being-done on the current assignment and discuss with lab-incharge
 - Make noise discussing concepts in OS

Keep the lab instructor engaged with any

Feedback from last year

End Semester FeedBack Part-I

Sr.No.	Question Title	Ratings	Star Ratings
1	Clarity of expectations of students	3.5	★ ★ ★ ★ ☆
2	Effectiveness of teacher in terms of: (a) Communication skills (b) Use of teaching aids	3.5	★ ★ ★ ★ ☆
3	Effectiveness of teacher in terms of: (a) Technical content (b) course content	3.5	★ ★ ★ ★ ☆
4	Feedback provided on students progress	3.5	★ ★ ★ ★ ☆
5	Has the teacher covered entire syllabus as prescribed by College?	3.6	★ ★ ★ ★ ☆
6	Has the teacher covered relevant topics beyond syllabus	3.5	★ ★ ★ ★ ☆
7	Motivation and inspiration for students to learn	3.5	★ ★ ★ ★ ☆
8	Pace on which contents were covered	3.4	★ ★ ★ ★ ☆
9	Support for the development of students skill (i) Practical demonstration (ii) Hands on training	3.5	★ ★ ★ ★ ☆
10	Willingness to offer help and advice to students.	3.5	★ ★ ★ ★ ☆
	Feedback Out of 25	17.5	★ ★ ★ ★ ☆

End Semester FeedBack Part-I

Sr.No.	Question Title	Ratings	Star Ratings
1	Clarity of expectations of students	3.8	
2	Effectiveness of teacher in terms of: (a) Communication skills (b) Use of teaching aids	3.8	
3	Effectiveness of teacher in terms of: (a) Technical content (b) course content	3.8	
4	Feedback provided on students progress	3.8	
5	Has the teacher covered entire syllabus as prescribed by College?	3.8	
6	Has the teacher covered relevant topics beyond syllabus	3.8	
7	Motivation and inspiration for students to learn	3.8	
8	Pace on which contents were covered	3.8	
9	Support for the development of students skill (i) Practical demonstration (ii) Hands on training	3.8	
10	Willingness to offer help and advice to students.	3.8	
	Feedback Out of 25	18.9	

Feedback from last year

Sr.No	Feed Back Student Comment
1	Teacher really makes you think to create a deep understanding about the subject
2	Its a very detailed course. It has been taught in very deeply. The advanced concepts should be optional for people who are willing to learn it. overall its a very time consuming course and very difficult to understand
3	None
4	good
5	Best teacher in this entire college IMO
6	Keep it up
7	excellent
8	Sir please teach us some subject in the next sem. If you are going to teach an elective please inform us.
9	Appreciate all the hard work put in the course. You are the only true professor who has taught us till now.

Sr.No	Feed Back Student Comment
1	Sir clearly explains the concepts in detail and gives personal attention to the queries of students. Engages the students in different lab tasks and makes the subject interesting.
2	teaching is excellent but over expection from student .

Survey Results

Your level of interest in learning

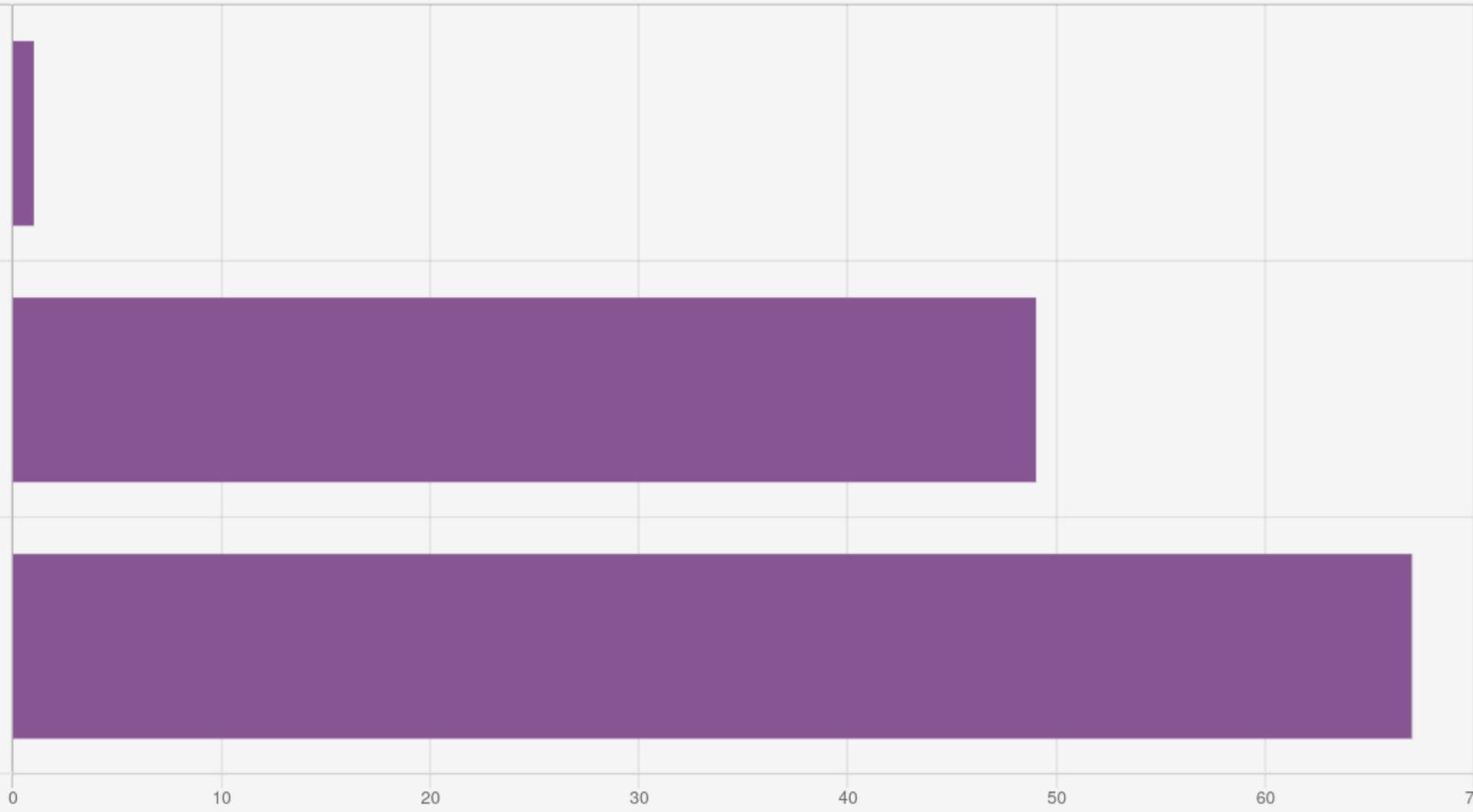
OS/Kernel

Responses

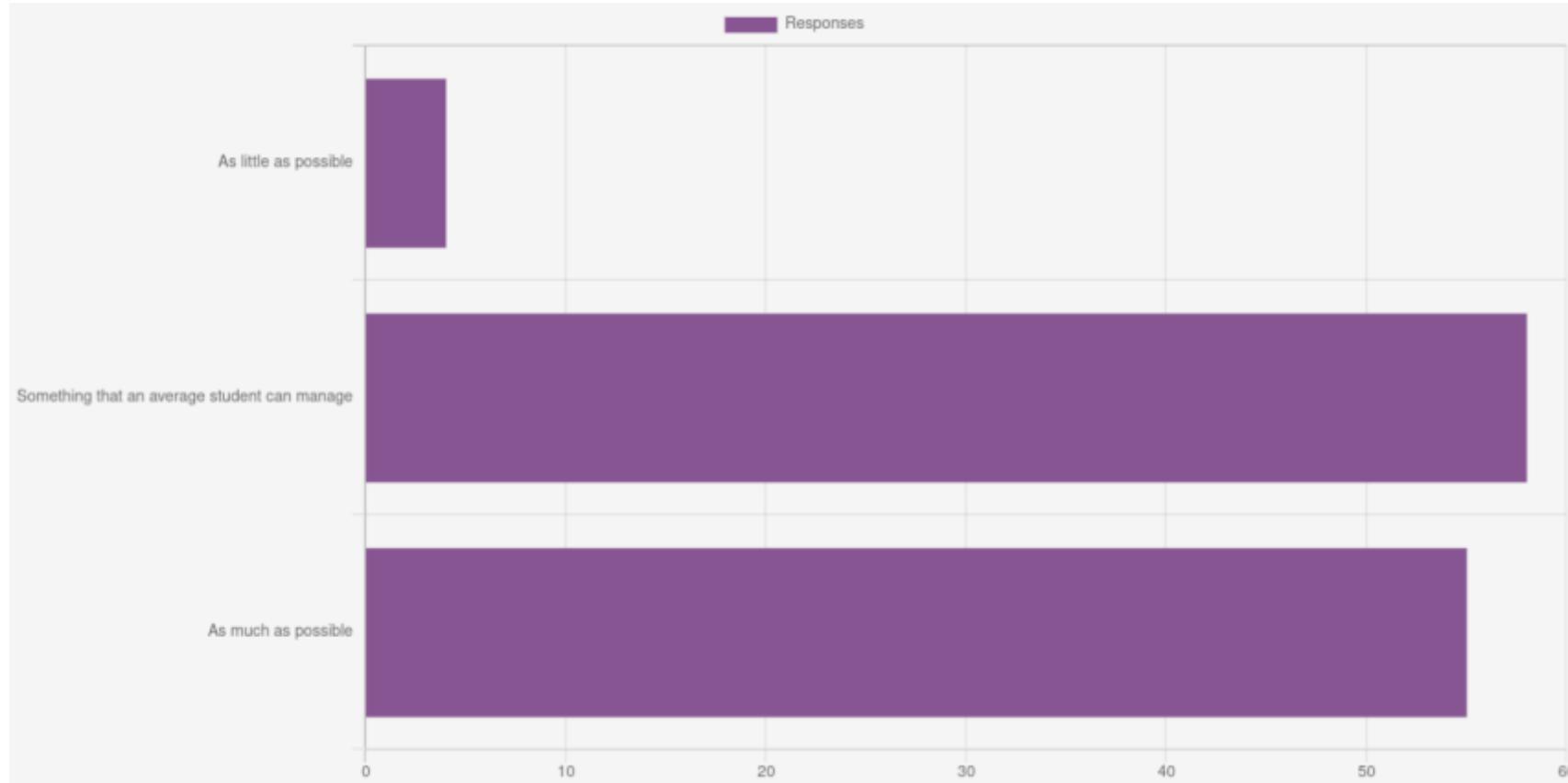
Not interested at all

Just to get basic introduction

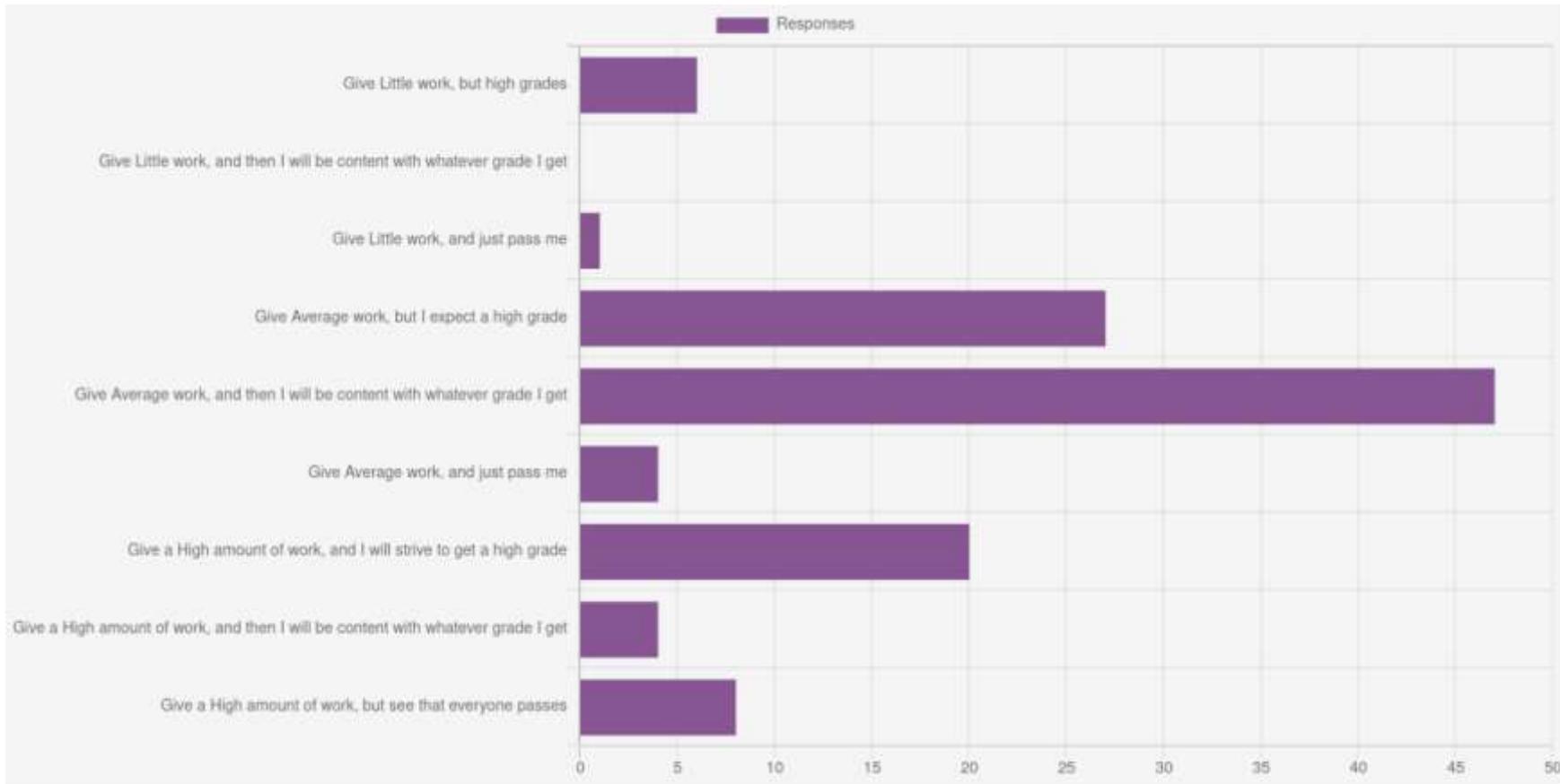
Seriously interested



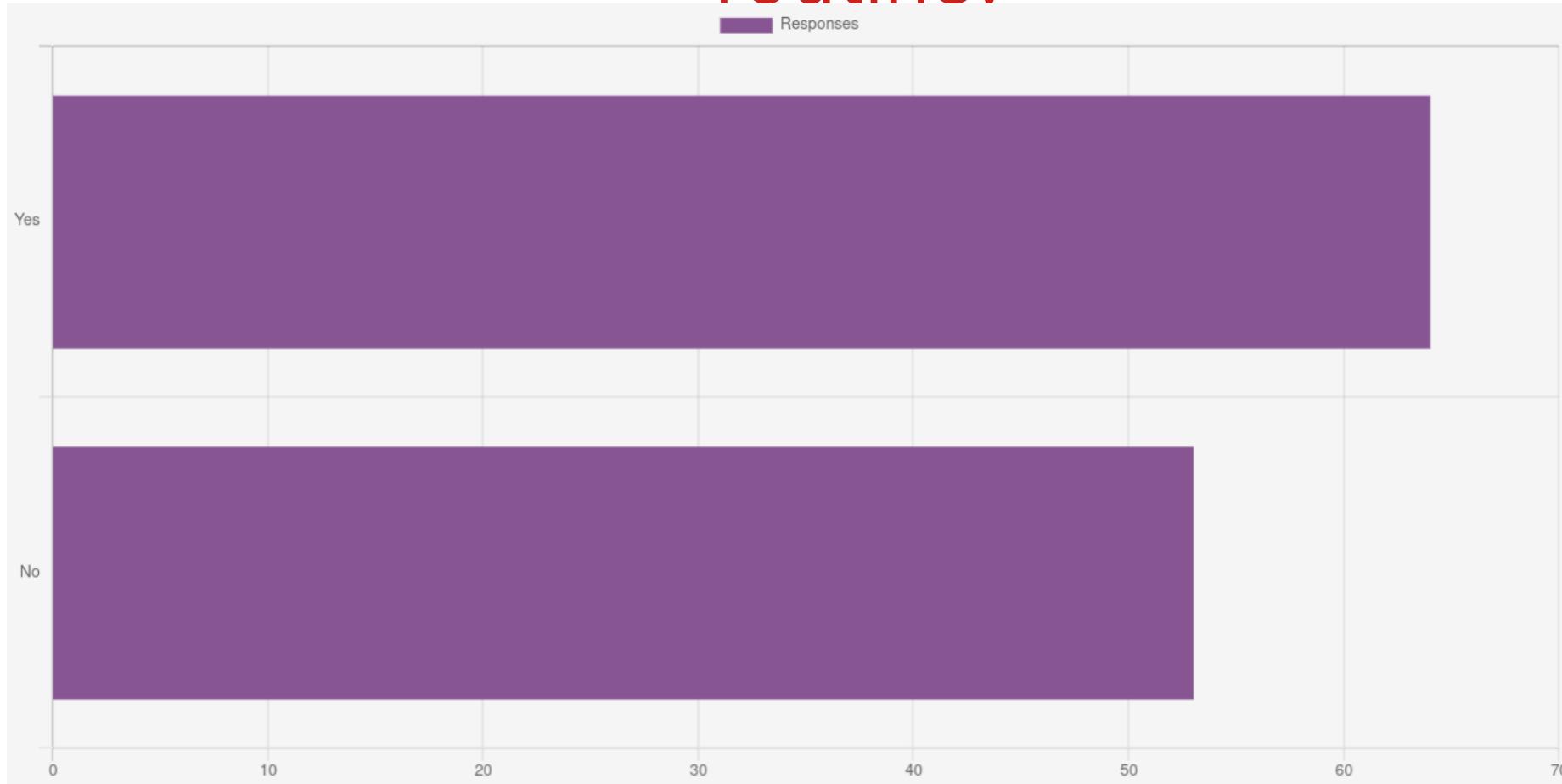
Your expectation about doing "hands-on" work



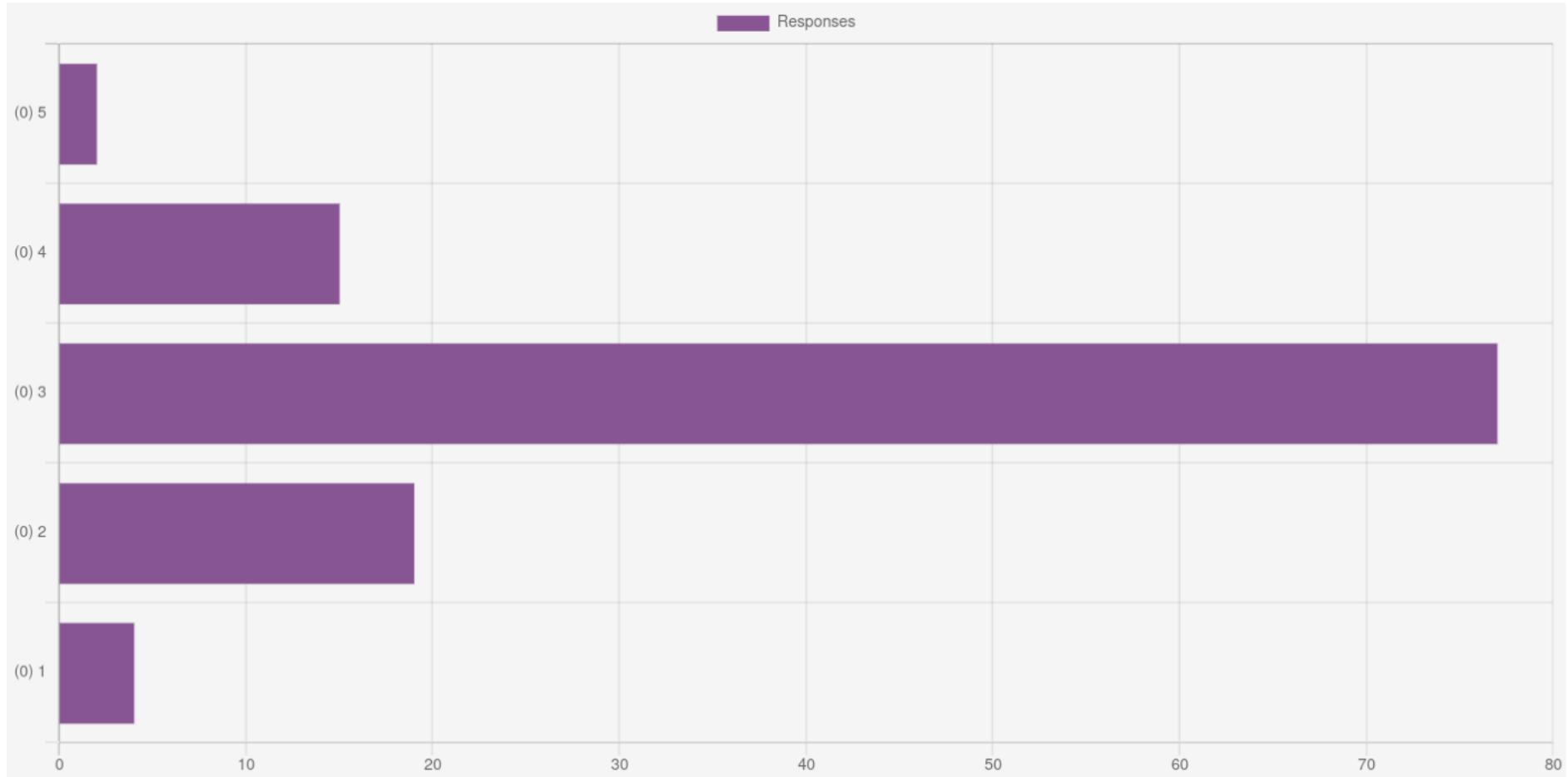
My expectations about Lab "work" (assignments, projects) and "grades":



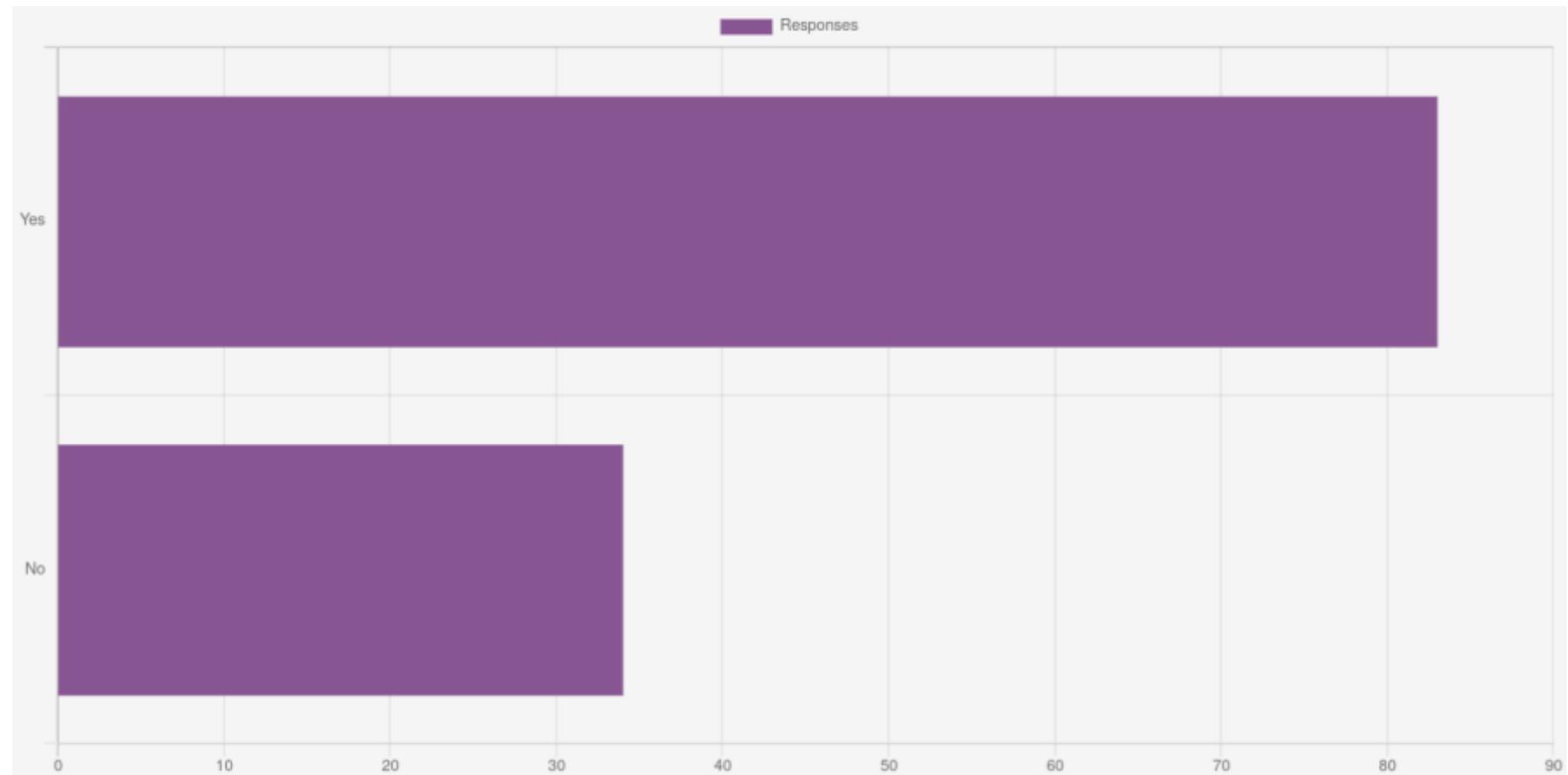
Do you use Linux command line as a routine?



Your self assessment on using Linux command line

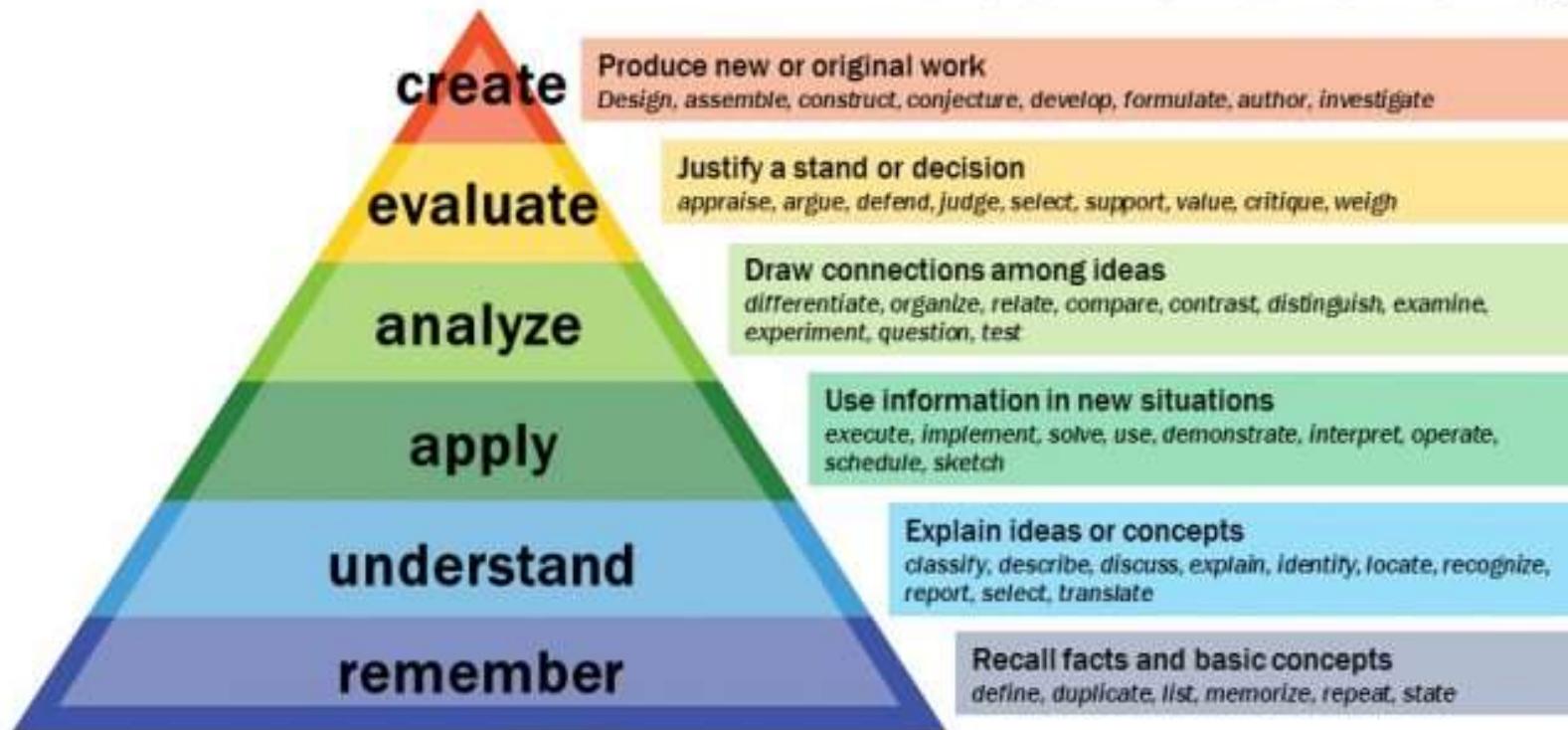


Are you willing to read a textbook?



More on learning

Bloom's Taxonomy



More on learning

- Remember
 - What is graph? What is the shortest path problem?
- Understand
 - Describe Djkstra's algorithm with one example.
- Apply
 - Given below is a graph. Find the shortest path

More on learning

- Analyze
 - Compare Dijkstra's algorithm with Bellman Ford algorithm and mention instances when Dijkstra's algorithm is preferable
- Evaluate
 - Indian Railways has chosen to use Dijkstra's algorithm for answering queries of customers. Is it a good decision?

Introduction to Applications, Files, Linux commands and Basics of System Calls

Abhijit A. M.

abhijit.comp@coep.ac.in

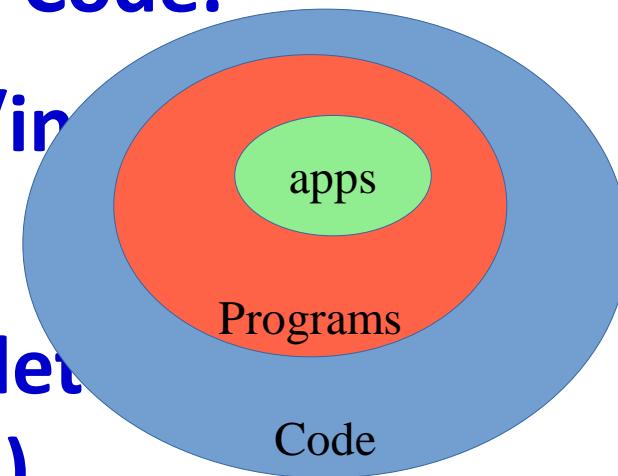
Why GNU/Linux ?

Few Key Concepts

- What is a file?
- File is a “dumb” sequence of bytes lying on the hard disk (or SSD or CD or PD, etc)
- It has a name, owner, size, etc.
- Does not do anything on its own ! Just stays there !

Few Key Concepts

- What is an application?
- Application is a program that runs in an “environment” created by the operating system, under the control of the operating system
- Hence also called “User Applications”
- Words: Program, Application, Code.
- Code: Any piece of complete/incomplete program in a programming language
- Program: Any piece of “complete/incomplete” program (device drivers, applications, ...)



Few Key Concepts

- ***Files don't open themselves***
- ***Always some application/program open()s a file.***
- ***Files don't display themselves***
- ***A file is displayed by the program which opens it.***
Each program has it's own way of handling files
- ***It's possible NOT TO HAVE an application to display/show a file***

Few Key Concepts

- Programs don't run themselves
- You click on a program, or run a command --> equivalent to request to Operating System to run it.
The OS runs your program
- Users (humans) request OS to run programs, using Graphical or Command line interface
- and programs open files

Path names

- Tree like directory structure
 - Root directory called /
 - Programs need to identify files using path names
 - A absolute/complete path name for a file.
□ /home/student/a.c
 - Relative path names, . and .. notation
- concept: every running program has a *current working directory*
- current directory

A command

- Name of an executable file
- For example: 'ls' is actually “/bin/ls”
- Command takes arguments
- E.g. ls /tmp/
- Command takes options
- E.g. ls -a

A command

- Command can take both arguments and options

- E.g. ls -a /tmp/

- Options and arguments are basically argv[] of the main() of that program

```
int main(int argc, char *argv[]) {  
    int i;  
  
    for(i = 0; i < argc; i++)  
        printf("%s\n", argv[i]);  
  
}  
  
$ ./a.out hi hello 123 /a/b/c.c ./m/a.c
```

Basic Navigation Commands

- **pwd**
- **ls**

- **ls -l**
- **ls -l /tmp/**
- **ls -l /home/student/Desktop**
- **ls -l ./Desktop**

- **ls -a**
- **\ls -F**

- **cd**

- **cd /tmp/**
- **cd**
- **cd /home/student/Desktop**

- **notation:** ~

- **cd ~**

Map these commands to navigation using a graphical file browser

Before the command line, the concept of Shell and System calls

□ System Call

- *Applications* often need to tasks involving hardware
 - Reading input, printing on screen, reading from network, etc.
 - They are not permitted to do it *directly* and compelled to do it using functionality given by OS
 - How is this done? We'll learn in later few lectures.
 - This functionality is called “system calls”

Before the command line, the concept of Shell and System calls

□ System Call

□ A function from OS code

□ Does specific operations with hardware (e.g. reading from keyboard, writing to screen, *opening* a file from disk, etc.)

□ Applications can't access hardware directly, they have to request the OS using system calls

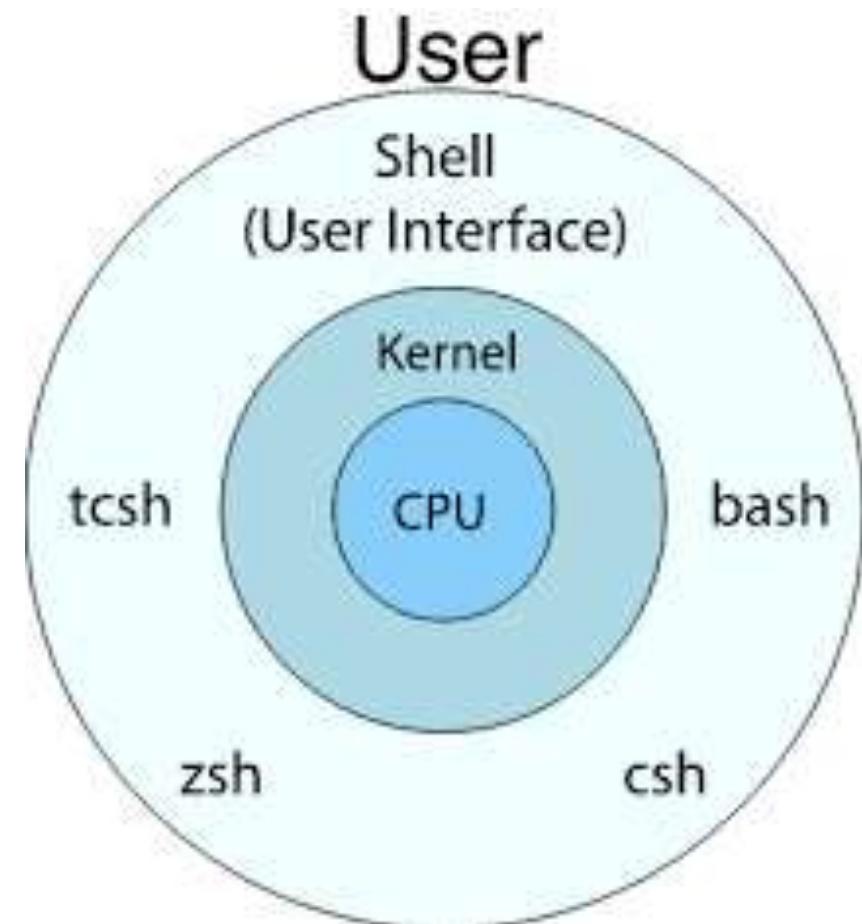
□ Examples

□ `open("/x/y", ..)`

□ `read(fd, &a, ...);`

The Shell

- **Shell = Cover**
- **Covers some of the Operating System's “System Calls” (mainly fork+exec) for the Applications**
- **Talks with Users and Applications and does some talk with OS**



Not a very accurate diagram !

The Shell

Shell waits for user's input

Requests the OS to run a program which the user has
asked to run

Again waits for user's input

GUI is a Shell !

Let's Understand fork() and exec()

```
#include <unistd.h>

int main() {
    fork();
    printf("hi\n");
    return 0;
}
```

```
#include <unistd.h>

int main() {
    printf("hi\n");
    execl("/bin/ls", "ls",
          NULL);
    printf("bye\n");
    return 0;
}
```

A simple shell

```
#include <stdio.h>
#include <unistd.h>
int main() {
    char string[128];
    int pid;
    while(1) {
        printf("prompt>");
        scanf("%s", string);
        pid = fork();
        if(pid == 0) {
            execl(string, string, NULL);
        } else {
            wait(0);
        }
    }
}
```

Users on Linux

- Root and others

- root

- superuser, can do (almost) everything

- Uid = 0

- Other users

- Uid != 0

- UID, GID understood by kernel

- Groups

- Set of users is a group

File Permissions on Linux

- 3 sets of 3 permission
- Octal notation: Read = 4, Write = 2, Execute = 1
- 644 means
 - Read-Write for owner, Read for Group, Read for others
 - chmod command, used to change permissions, uses these notations
 - It calls the chmod() system call
 - Permissions are for processes started by the user, but in common language often we say “permissions”

File Permissions on Linux

-rw-r--r-- 1 abhijit abhijit 1183744 May 16 12:48 01_linux_basics.ppt

-rw-r--r-- 1 abhijit abhijit 341736 May 17 10:39 Debian Family Tree.svg

drwxr-xr-x 2 abhijit abhijit 4096 May 17 11:16 fork-exec

-rw-r--r-- 1 abhijit abhijit 7831341 May 11 12:13 foss.odp

3 sets of 3 permissions

3 sets = user (owner), group,
others

3 permissions = read, write,
execute

Owner

size

name

last-modification

hard link count

File Permissions on Linux

- **r on a file : can read the file**
 - **open(... O_RDONLY) works**
- **w on a file: can modify the file**
 - **open(... O_WRONLY) works**
- **x on a file: can ask the os to run the file as an executable program**
 - **exec(...) works**
- **r on a directory: can do 'ls'**
 - **w on a directory: can add/remove files from that directory (cannot write to files)**

Access rights examples

`---rw-r--r--`

Readable and writable for file owner (actually a process started by the owner!), only readable for others

`---rW-r-----`

Readable and writable for file owner, only readable for users belonging to the file group.

`drwx-----`

Directory only accessible by its owner

`-----r-x`

File executable by others but neither by your friends nor by yourself. Nice protections for a trap...

Permissions: more !

- Setuid/setgid bit

```
$ ls -l /usr/bin/passwd
```

```
-rwsr-xr-x 1 root root 68208 Nov 29 17:23 /usr/bin/passwd
```

- How to set the s bit?

```
▫ chmod u+s <filename>
```

- What does this mean?

▫ Any user can run this process, but the process itself runs as if run by the owner of the file

- passwd runs as if run by “root” even if you run it

Man Pages (self study)

■ Manpage

- \$ man ls
- \$ man 2 mkdir
- \$ man man
- \$ man -k mkdir

■ Manpage sections

- 1 User-level cmds and apps
- /bin/mkdir
- 2 System calls
- int mkdir(const char *, ...);
- 3 Library calls
- int printf(const char *, ...);

□ 4 Device drivers and network protocols

■ /dev/tty

□ 5 Standard file formats

■ /etc/hosts

□ 6 Games and demos

■ /usr/games/fortune

□ 7 Misc. files and docs

■ man 7 locale

□ 8 System admin. Cmds

■ /sbin/reboot

GNU / Linux filesystem structure

Not imposed by the system. Can vary from one system to the other, even between two GNU/Linux installations!

/	Root directory
/bin/	Basic, essential system commands
/boot/	Kernel images, initrd and configuration files
files	
/dev/	Files representing devices <i>/dev/hda: first IDE hard disk</i>
/etc/	System and application configuration files
/home/	User directories
/lib/	Basic system shared libraries

GNU / Linux filesystem structure (self study)

/lost+found

/media

/media/cdrom

/mnt/

filesystems

/opt/

instead

/proc/

/proc/version ...

/root/

/sbin/

/sys/

power, etc.)

Corrupt files the system tried to recover

**Mount points for removable media:
/media/usbdisk,**

Mount points for temporarily mounted

**Specific tools installed by the sysadmin
/usr/local/ often used**

Access to system information

/proc/cpuinfo,

root user home directory

Administrator-only commands

**System and device controls
(cpu frequency, device**

GNU / Linux filesystem structure (self study)

/tmp/

/usr/

system)

Temporary files

Regular user tools (not essential to the

/usr/bin/, /usr/lib/,

/usr/sbin...

Specific software installed by the sysadmin

(often preferred to /opt/)

/var/

Data used by the system or system servers

/var/log/, /var/spool/mail
mail), /var/spool/lpd

(incoming

(print jobs)...

Files: cut, copy, paste, remove, (self study)

□ **cat <filenames>**

□ **cat /etc/passwd**

□ **cat fork.c**

□ **cat <filename1>
<filename2>**

□ **cp <source> <target>**

□ **cp a.c b.c**

□ **cp a.c /tmp/**

□ **cp a.c /tmp/b.c**

□ **mv <source> <target>**

□ **mv a.c b.c**

□ **mv a.c /tmp/**

□ **mv a.c /tmp/b.c**

□ **rm <filename>**

□ **rm a.c**

□ **rm a.c b.c c.c**

□ **rm -r /tmp/a**

□ **mkdir**

□ **mkdir /tmp/a /tmp/b**

□ **rmdir**

Useful Commands (self study)

- echo
- echo hi
- echo hi there
- echo “hi there”
- j=5; echo \$j
- sort
- sort
- sort < /etc/passwd
- firefox
- grep
- grep bash /etc/passwd
- grep -i display /etc/passwd
- egrep -i 'a|b' /etc/passwd
- less <filename>
- head <filename>
- head -5 <filename>
- tail -10 <filename>

Useful Commands (self study)

□ **alias**

alias ll='ls -l'

□ **tar**

tar cvf folder.tar folder

□ **gzip**

gzip a.c

□ **touch**

touch xy.txt

touch a.c

□ **strings**

strings a.out

□ **adduser**

sudo adduser test

□ **su**

su administrator

Useful Commands (self study)

□ **df**

df -h

□ **du**

du -hs .

□ **bc**

□ **time**

□ **date**

□ **diff**

□ **wc**

□ **dd**

Network Related Commands (self study)

□ **ifconfig**

□ **ssh**

□ **scp**

□ **telnet**

□ **ping**

□ **w**

□ **last**

□ **whoami**

Unix job control

- Start a background process:

- gedit a.c &

- gedit

- hit ctrl-z*

- bg**

- Where did it go?

- jobs

- ps

- Terminate the job: kill it

- kill %*jobid*

- kill *pid*

- Bring it back into the foreground

- fg %1

Configuration Files

- Most applications have configuration files in TEXT format

- Most of them are in `/etc`

- `/etc/passwd` and `/etc/shadow`

- Text files containing user accounts

- `/etc/resolv.conf`

- DNS configuration

- `/etc/network/interfaces`

- Network configuration

- `/etc/hosts`

- Local database of Hostname-IP mappings

- `/etc/apache2/apache2.conf`

- Apache webserver configuration

~/.bashrc file (self study)

□ **~/.bashrc**

Shell script read each time a bash shell is started

- You can use this file to define
 - Your default environment variables (PATH, EDITOR...).
 - Your aliases.
 - Your prompt (see the bash manual for details).
 - A greeting message.
 - Also **~/.bash_history**

Mounting

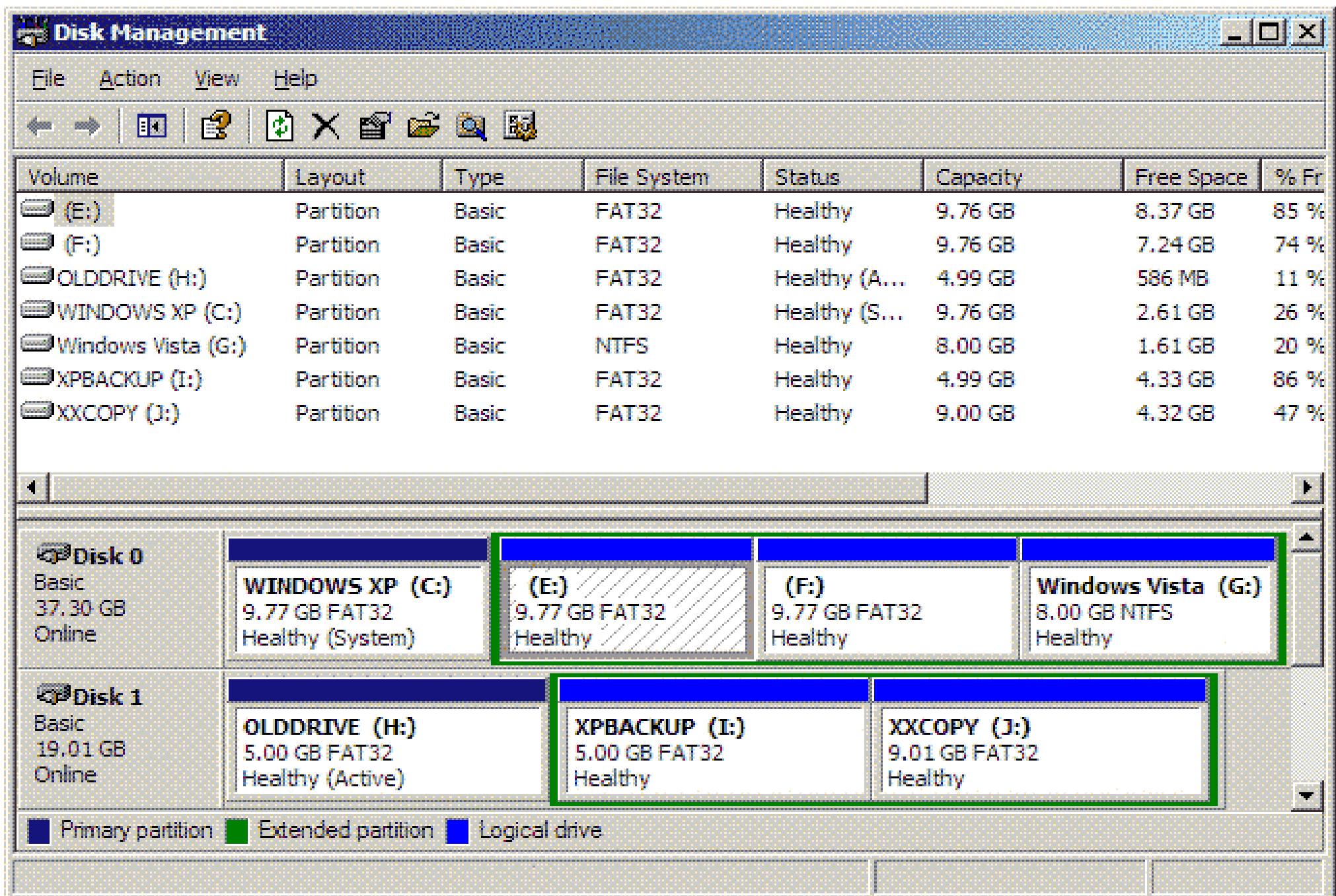
Partition

- What is C:\ , D:\, E:\ etc on your computer ?
- “Drive” is the popular term
- Typically one of them represents a CD/DVD RW
- What do the others represent ?
- They are “partitions” of your “hard disk”

Partition

- Your hard disk is one contiguous chunk of storage
- Lot of times we need to “logically separate” our storage
- Partition is a “logical division” of the storage
- Every “drive” is a partition
- A logical chunk of storage is partition
- Hard disk partitions (C:, D:), CD-ROM, Pen drive, ...

Partitions



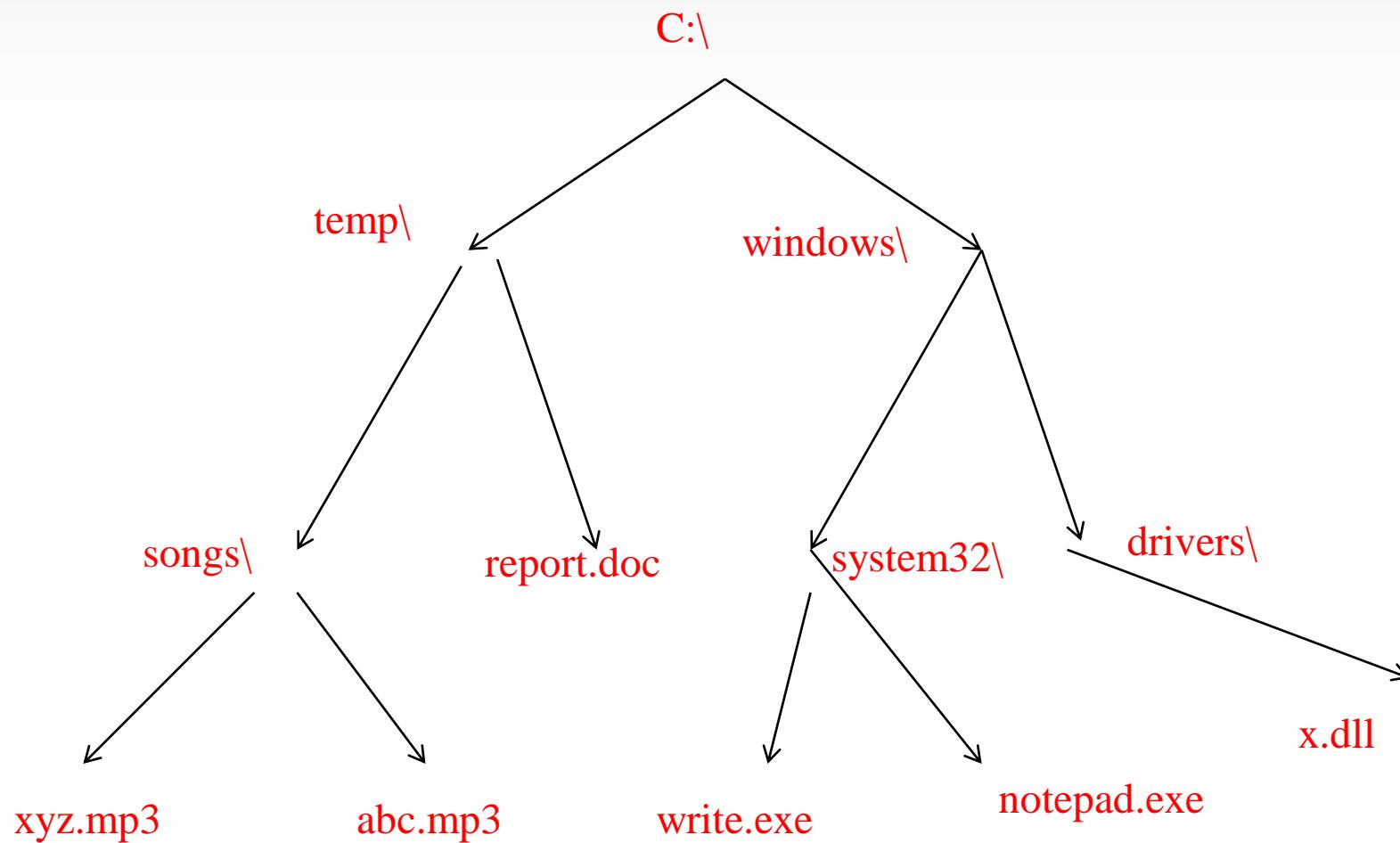
Managing partitions and hard drives

- System → Administration → Disk Utility
- Use **gparted** or **fdisk** to partition drives on Linux
- Hard drive partition names on Linux
 - `/dev/sda` → Entire hard drive
 - `/dev/sda1`, `/dev/sda2`, `/dev/sda3`, Different partitions of the hard drive
- Each partition has a *type* – ext4, ext3, ntfs, fat32, etc.
- Formatting: creating an empty layout on disk, layout capable of storing the tree of files/folders

Windows Namespace

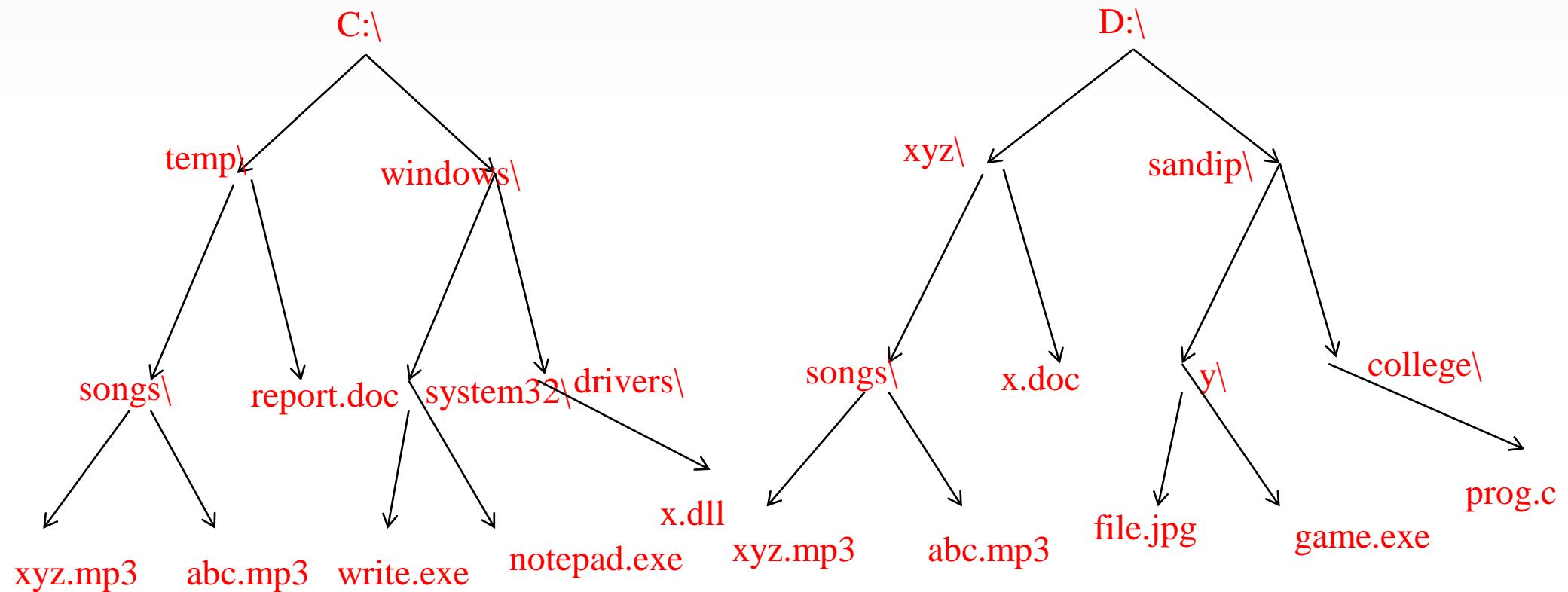
c:\temp\songs\xyz.mp3

- Root is C:\ or D:\ etc
- Separator is also “\”



Windows Namespace

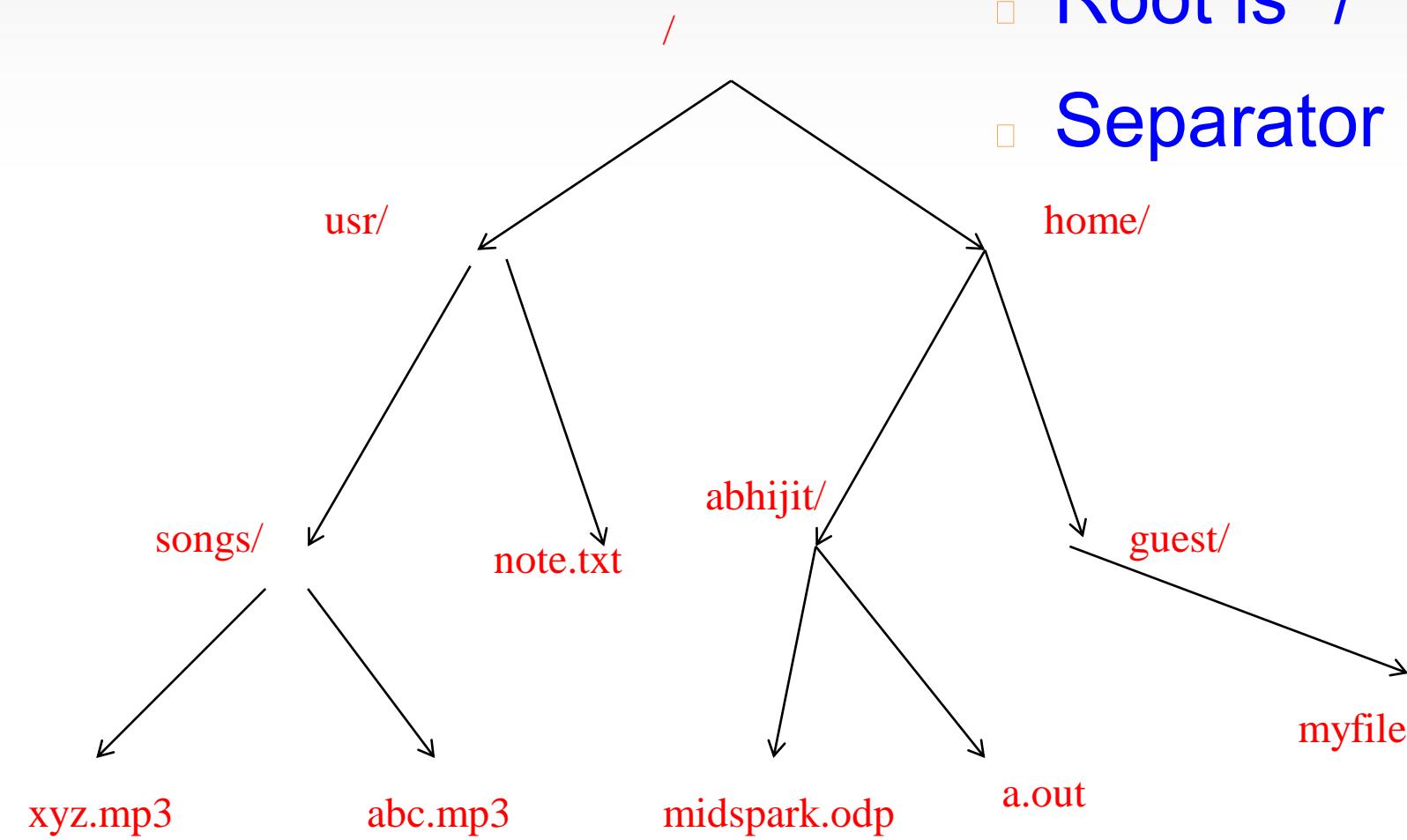
- C:\ D:\ Are partitions of the disk drive
- Typical convention: C: contains programs, D: contains data
 - One “tree” per partition
 - Together they make a “forest”



Linux Namespace: On a partition

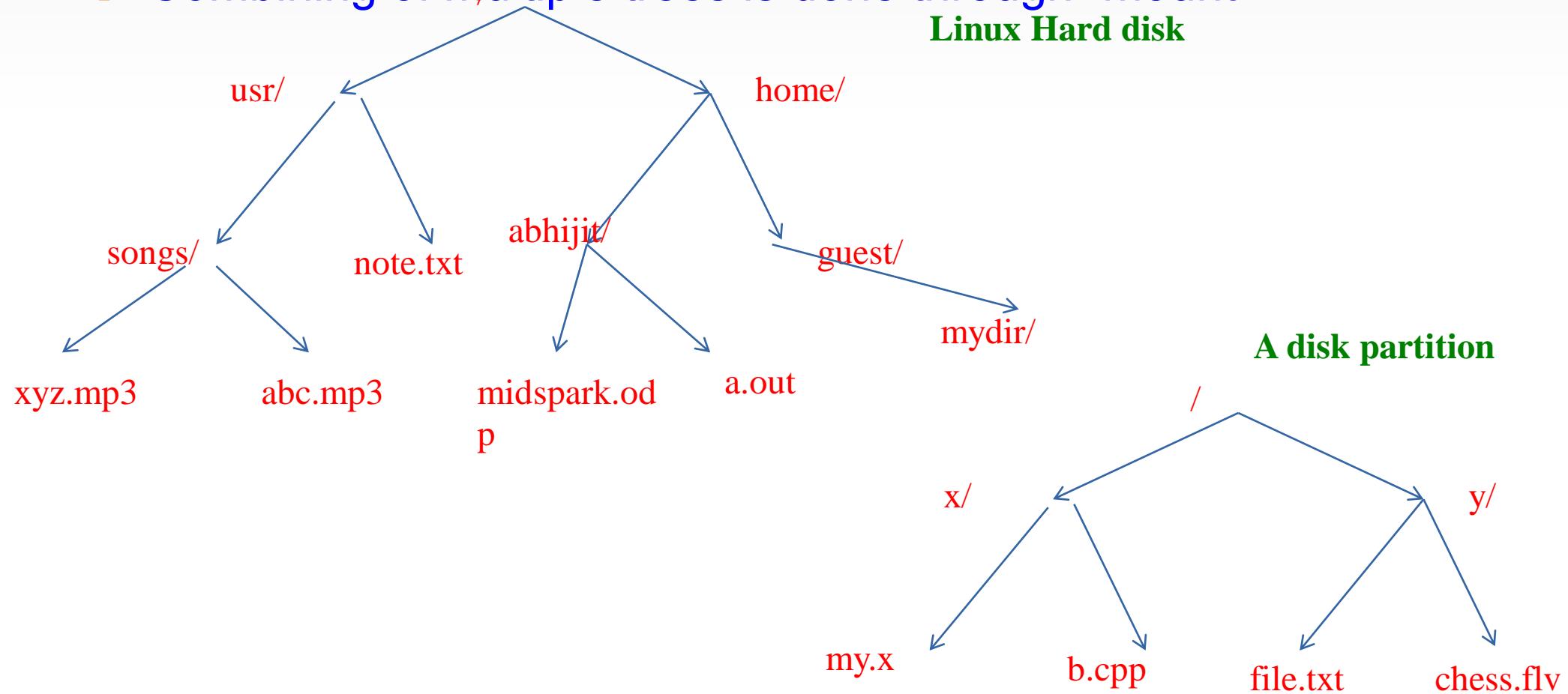
/usr/songs/xyz.mp3

- On every partition:
- Root is “/”
- Separator is also “/”



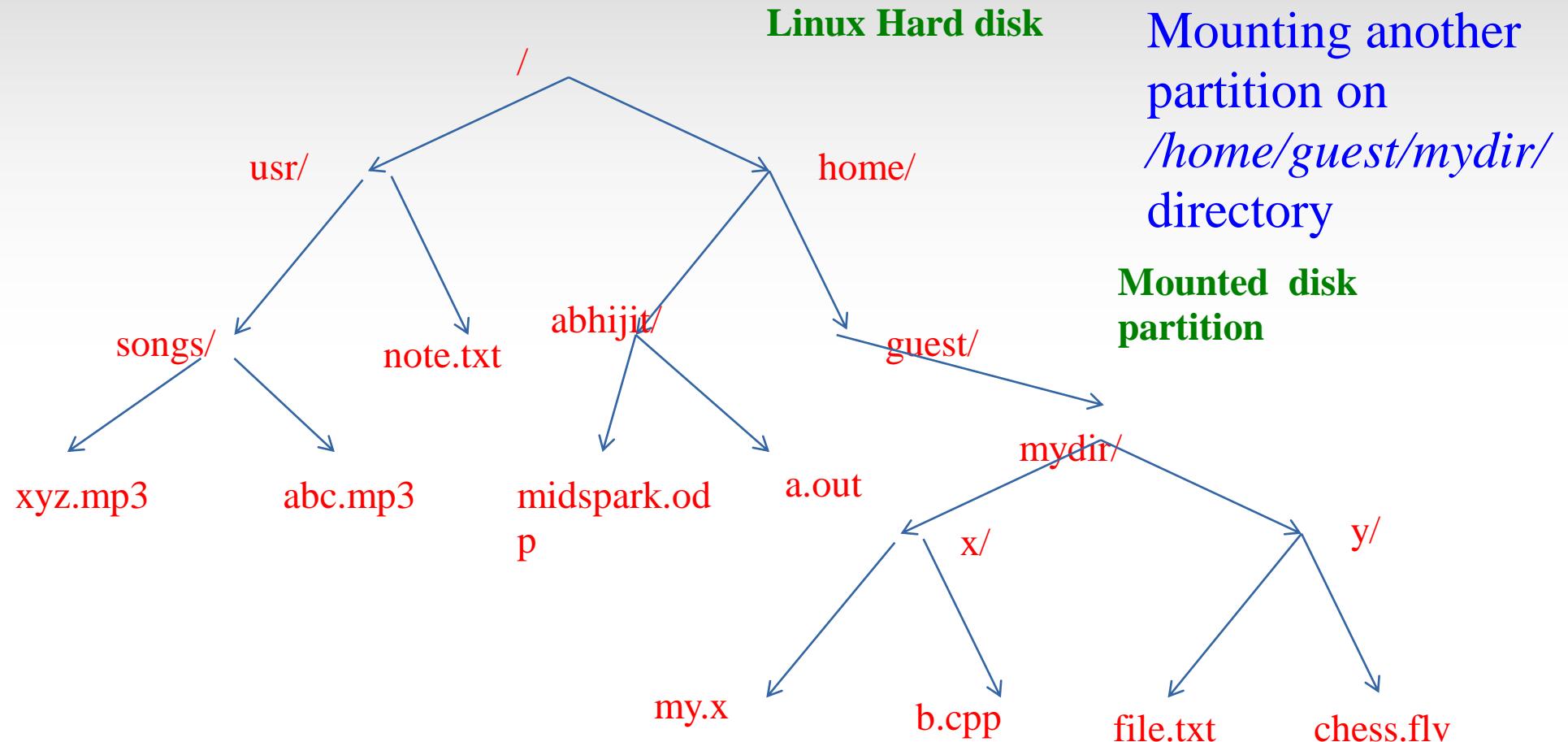
Linux namespace: Mount

- Linux namespace is a single “tree” and not a “forest” like Windows
- Combining of multiple trees is done through “mount”



Linux namespace

Mounting a partition



/home/guest/mydir/x/b.cpp → way to access
the file on the other disk partition

Mounting across network!

Using Network File System (NFS)

`sudo apt install nfs-common`

`$ sudo mount 172.16.1.75:/mnt/data /myfolder`

Files that are not regular/directory

Special devices (1)

Device files with a special behavior or contents

`/dev/null`

The data sink! Discards all data written to this file.

Useful to get rid of unwanted output, typically log information:

```
mplayer black_adder_4th.avi &> /dev/null
```

`/dev/zero`

Reads from this file always return \0 characters

Useful to create a file filled with zeros:

```
dd if=/dev/zero of=disk.img bs=1k count=2048
```

See `man null` or `man zero` for details

Special devices (2)

`/dev/random`

Returns random bytes when read. Mainly used by cryptographic programs. Uses interrupts from some device drivers as sources of true randomness (“entropy”).

Reads can be blocked until enough entropy is gathered.

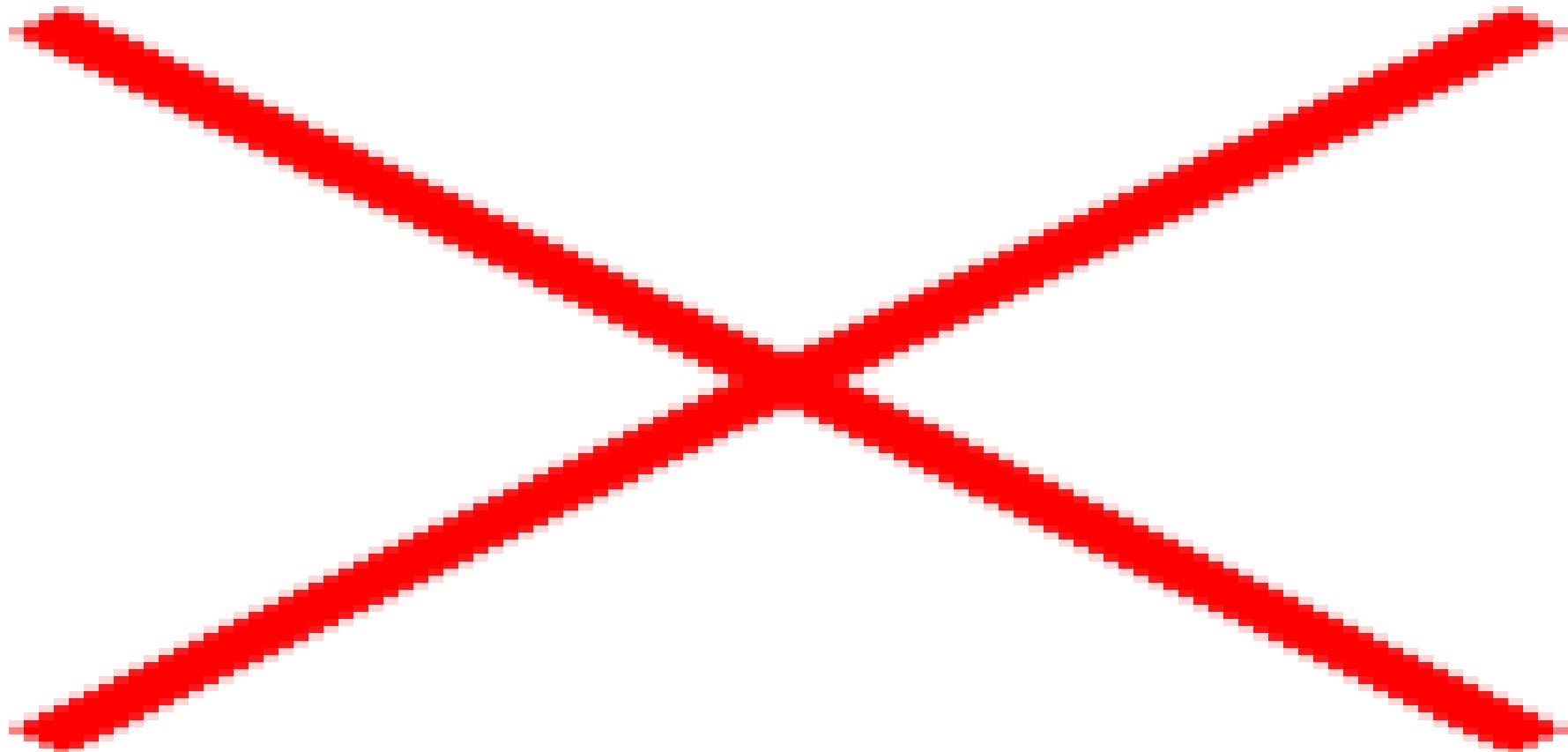
`/dev/urandom`

For programs for which pseudo random numbers are fine. Always generates random bytes, even if not enough entropy is available (in which case it is possible, though still difficult, to predict future byte sequences from past ones).

See man `random` for details.

Files names and inodes

Hard Links Vs Soft Links



Creating “links”

□ Hard link

```
$ touch m
```

```
$ ls -l m
```

```
-rw-rw-r-- 1 abhijit abhijit 0 Jan  5 16:18 m
```

```
$ ln m mm
```

```
$ ls -l m mm
```

```
-rw-rw-r-- 2 abhijit abhijit 0 Jan  5 16:18 m
```

```
-rw-rw-r-- 2 abhijit abhijit 0 Jan  5 16:18 mm
```

```
$ ln mm mmm
```

Event Driven Kernel, Multi-tasking OS

Abhijit A. M.

abhijit.comp@coep.ac.in

(C) Abhijit A.M.

Available under Creative Commons Attribution-ShareAlike License V3.0+

Building an “OS”

- E.g. Debian

- A collection of thousands of applications, libraries, system programs, ... and Linux kernel !
- Linux kernel is at the heart, but heart is not a human without the body !
- Job of “Debian Developers”
 - Collect the source code of all things you want
 - Create an “Environment” for compiling things : bootstrap challenge
 - Compile everything

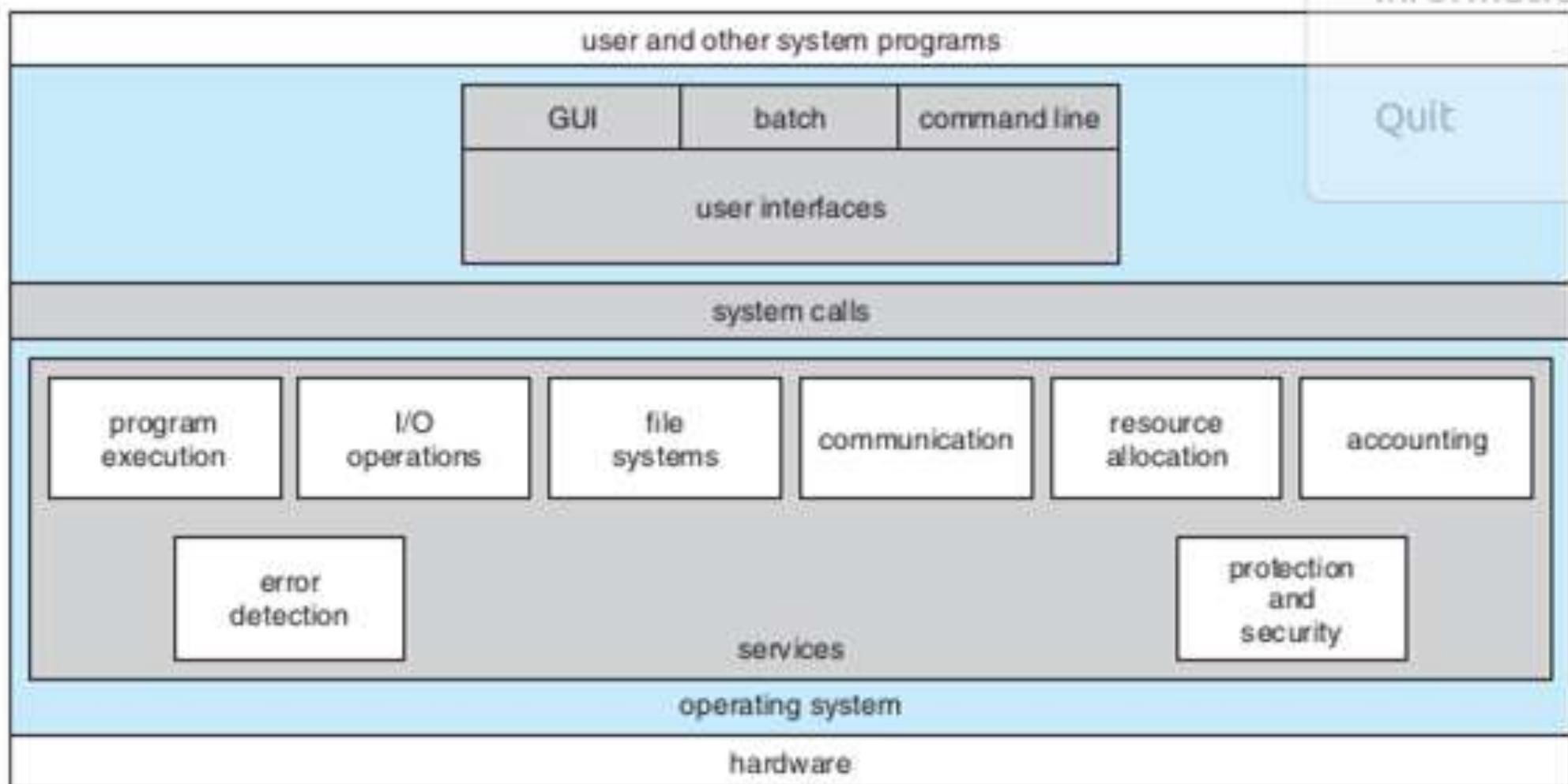


Figure 2.1 A view of operating system services.

Components of a computer system

Abstract view

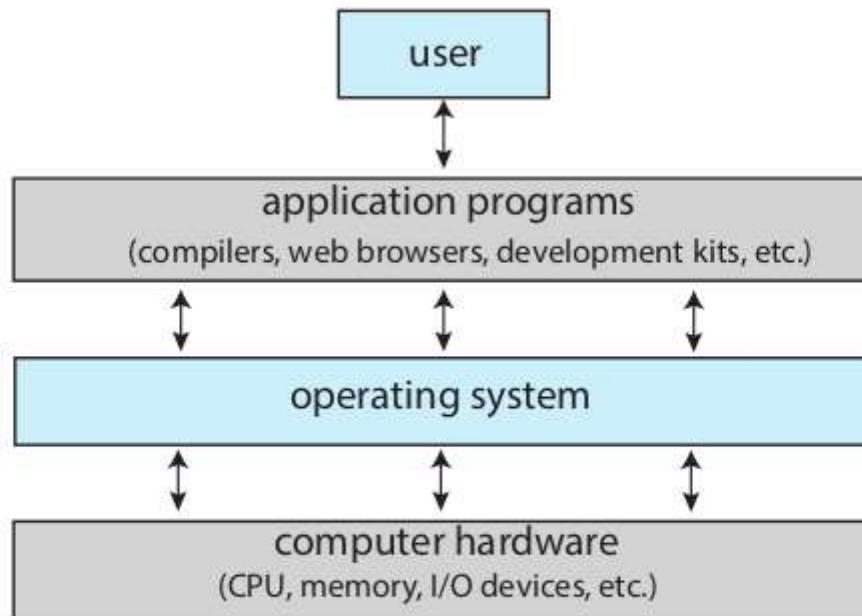
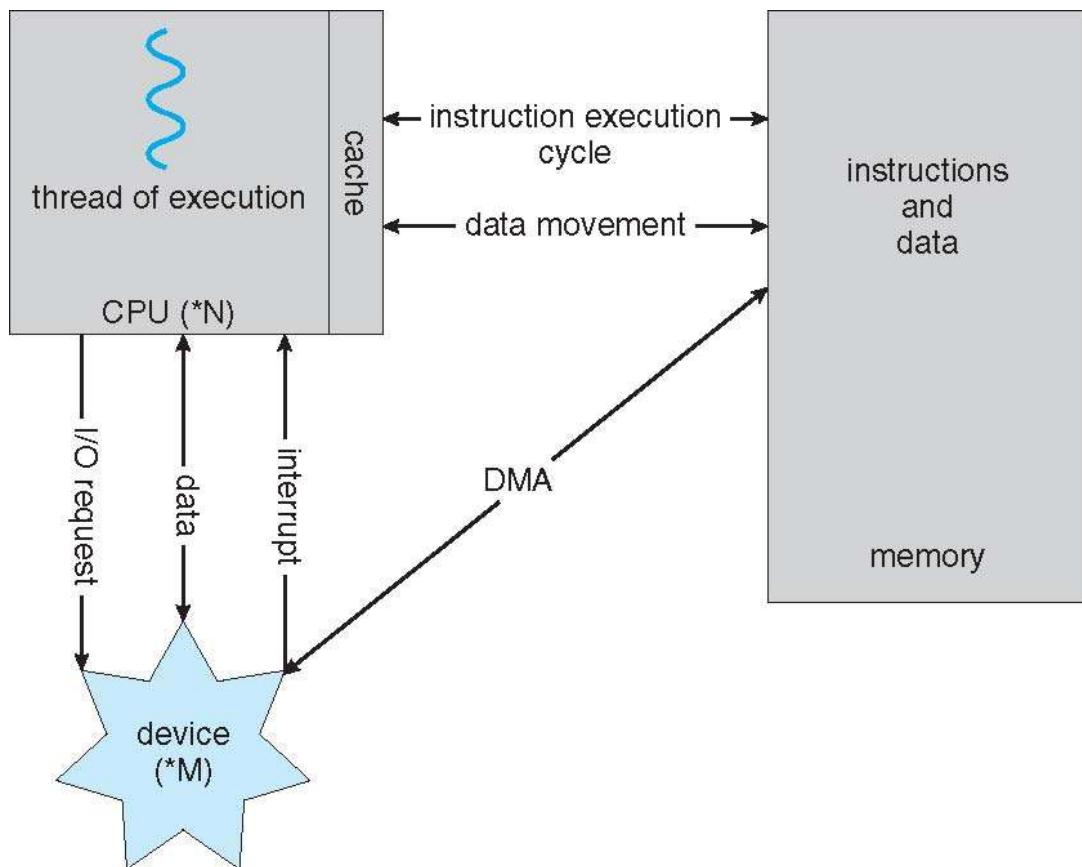


Figure 1.1 Abstract view of the components of a computer system.

Von Neumann architecture



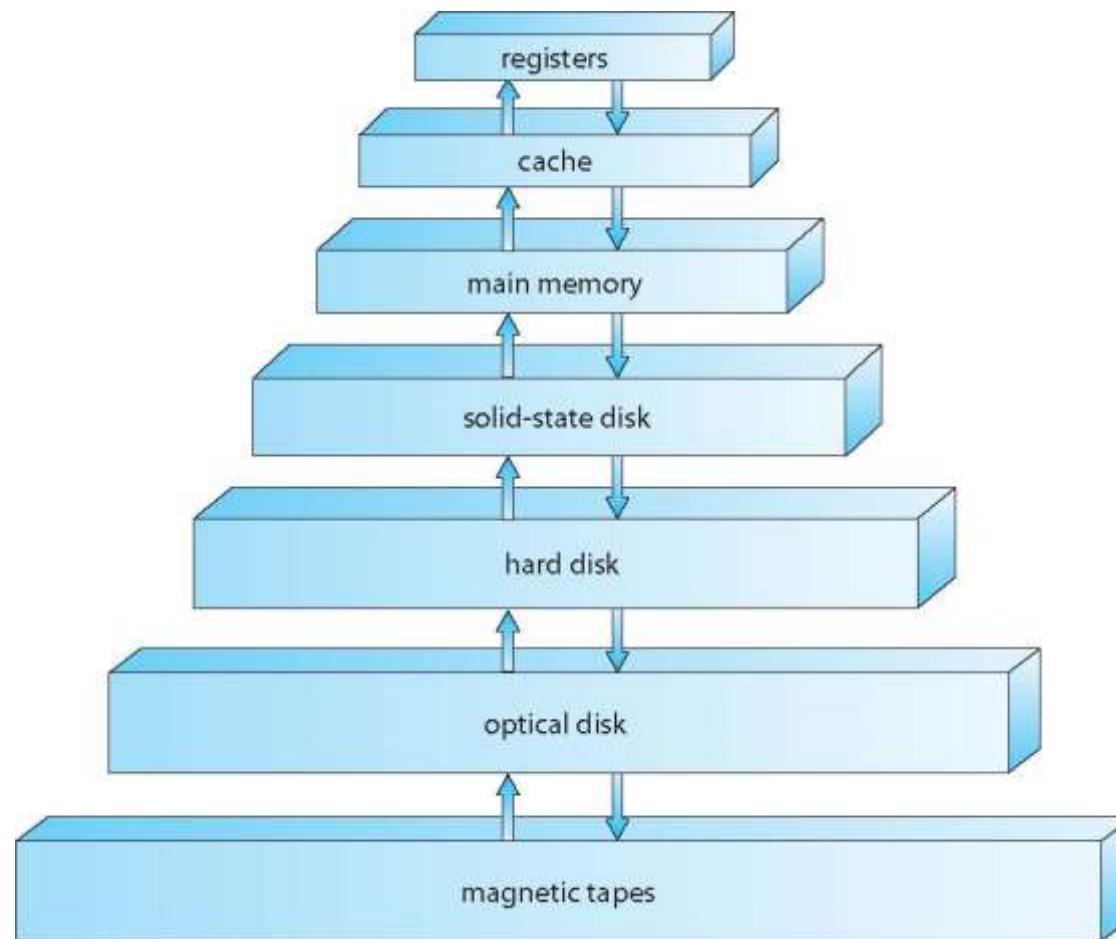
- Processor
 - Fetch
 - Decode
 - Execute
 - Repeat

A von Neumann architecture

A key to “understanding”

- Everything happens on processor
- Processor is always running some instruction
- We should be able to tell possible execution sequences on processor

Memory hierarchy



Memory hierarchy

System Calls

- Services provided by operating system to applications
 - Essentially available to applications by calling the particular software interrupt application
 - All system calls essentially involve the “INT 0x80” on x86 processors + Linux
 - Different arguments specified in EAX register inform the kernel about different system calls
 - The C library has wrapper functions for each of the system calls

Types of System Calls

□ File System Related

- Open(), read(), write(), close(), etc.

□ Processes Related

- Fork(), exec(), ...

□ Memory management related

- Mmap(), shm_open(), ...

□ Device Management

□ Information maintainance – time,date

https://linuxhint.com/list_of_linux_syscalls/

□ Communication between processes (IPC)

EXAMPLES OF WINDOWS AND UNIX SYSTEM CALLS

	Windows	Unix
Process Control	CreateProcess() ExitProcess() WaitForSingleObject()	fork() exit() wait()
File Manipulation	CreateFile() ReadFile() WriteFile() CloseHandle()	open() read() write() close()
Device Manipulation	SetConsoleMode() ReadConsole() WriteConsole()	ioctl() read() write()
Information Maintenance	GetCurrentProcessID() SetTimer() Sleep()	getpid() alarm() sleep()
Communication	CreatePipe() CreateFileMapping() MapViewOfFile()	pipe() shm_open() mmap()
Protection	SetFileSecurity() InitializeSecurityDescriptor() SetSecurityDescriptorGroup()	chmod() umask() chown()

```
int main() {  
    int a = 2;  
    printf("hi\n");  
}
```

C Library

```
int printf("void *a, ...){  
...  
    write(1, a, ...);  
}
```

Code schematic
-----user-kernel-mode-boundary-----

//OS code
int sys_write(int fd, char
*, int len) {

**figure out location on
disk**

**where to do the write
and**
carry out the

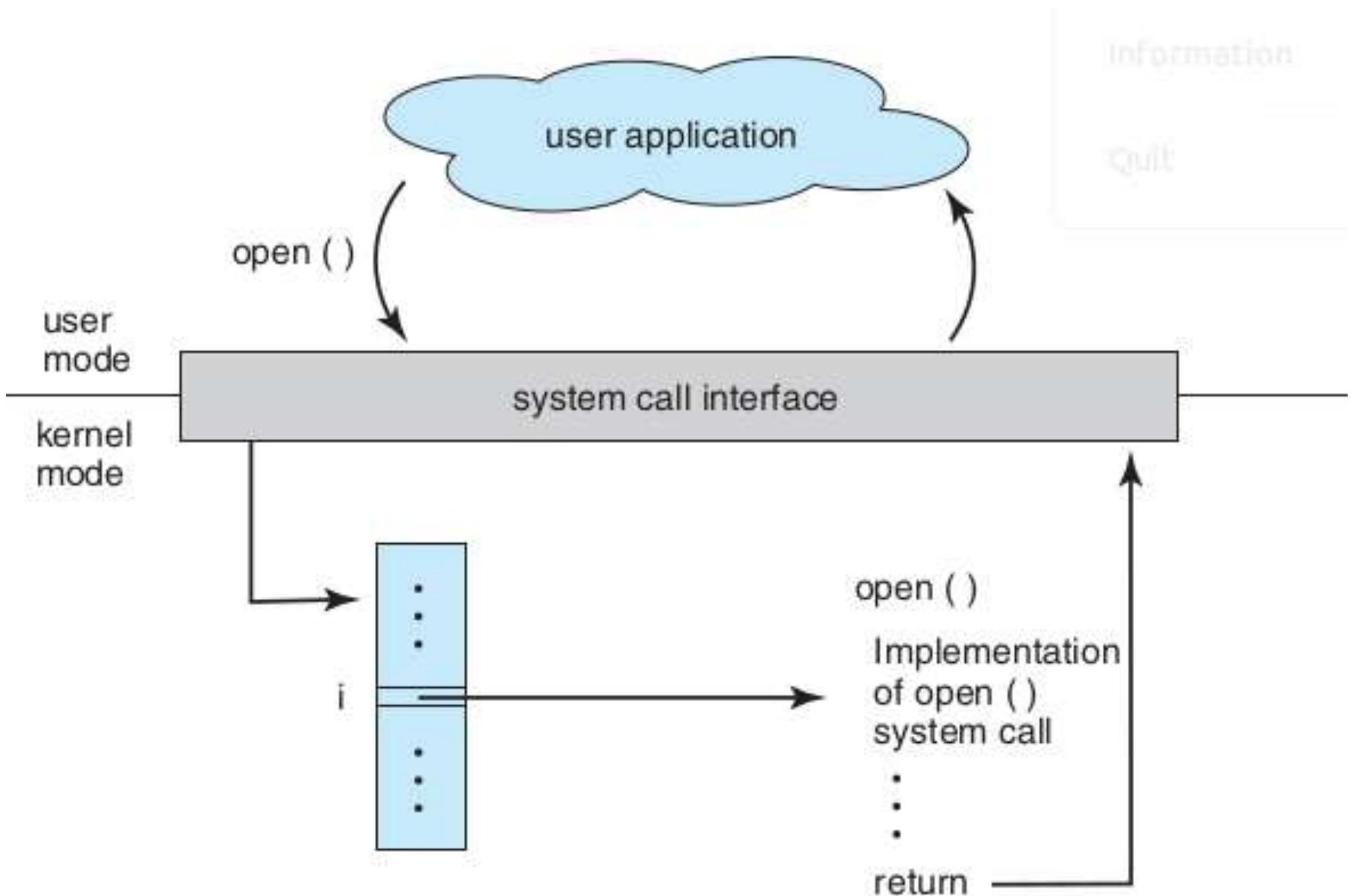


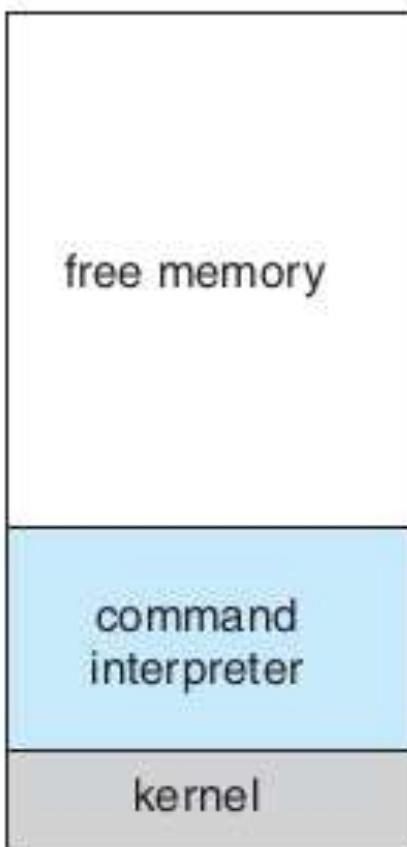
Figure 2.6 The handling of a user application invoking the `open()` system call.

Process

- A program in execution
- Exists in RAM
- Scheduled by OS
 - In a timesharing system, intermittantly scheduled by allocating a time quantum, e.g. 20 microseconds
- The “ps” command on Linux

Process in RAM

- Memory is required to store the following components of a process
 - Code
 - Global variables (data)
 - Stack (stores local variables of functions)
 - Heap (stores malloced memory)
 - Shared libraries (e.g. code of printf, etc)
 - Few other things, may be



(a)

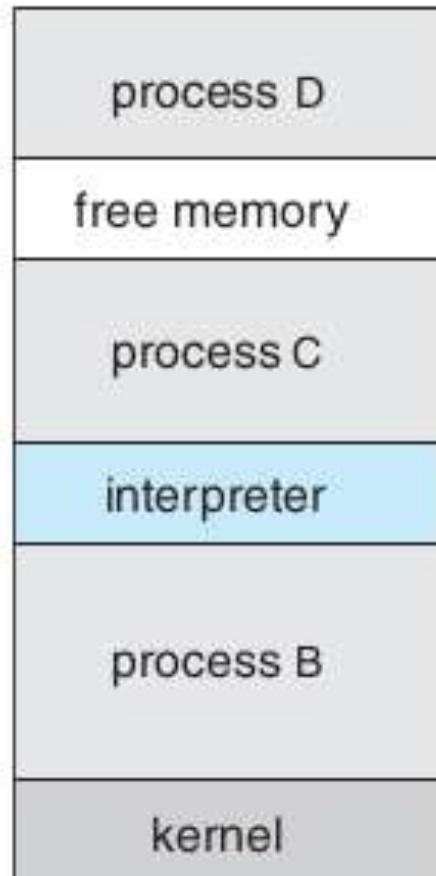


(b)

Figure 2.9 MS-DOS execution. (a) At system startup. (b) Running a program.

MS-DOS: a single tasking operating system

Only one program in RAM at a time, and only one program can run at a time



A multi tasking system
With multiple programs loaded in memory
Along with kernel

(A very simplified conceptual diagram. Things are more complex in reality)

fork()

- A running process creates it's duplicate!
- After call to fork() is over
 - Two processes are running
 - Identical
 - The calling function returns in two places!
 - Caller is called parent, and the new process is called child
 - PID is returned to parent and 0 to child

exec()

- Variants: execvp(), execl(), etc.
- Takes the path name of an executable as an argument
- Overwrites the existing process using the code provided in the executable
- The original process is OVER ! Vanished!
- The new program starts running overwritting the existing process!

Let's Understand fork() and exec()

```
#include <unistd.h>

int main() {
    fork();
    printf("hi\n");
    return 0;
}
```

```
#include <unistd.h>

int main() {
    printf("hi\n");
    execl("/bin/ls", "ls", NULL);
    printf("bye\n");
    return 0;
}
```

A simple shell

```
#include <stdio.h>
#include <unistd.h>
int main() {
    char string[128];
    int pid;
    while(1) {
        printf("prompt>");
        scanf("%s", string);
        pid = fork();
        if(pid == 0) {
            execl(string, string, NULL);
        } else {
            wait(0);
        }
    }
}
```

Shell using fork and exec

□ Demo

- The only way a process can be created on Unix/Linux is using `fork()` + `exec()`
- All processes that you see were started by some other process using `fork()` + `exec()` , except the initial “*init*” process
- *When you click on “firefox” icon, the user-interface program does a `fork()` + `exec()` to start firefox; same with a command line shell program*
- The “bash” shell you have been using is nothing but a command line interface application.

The boot process

□ BIOS/UEFI

- The firmware. Runs “Automatically”.
- CPU is hardwired to start running instructions stored at a fixed address.
- The Motherboard manufacturers, ensure that the BIOS/UEFI is at his address

□ kernel

- The Boot Loader shows the choice of running an OS, user selects a choice
 - The Boot loader loads the code of kernel from disk into RAM
 - PC changed to kernel code
- ## □ Kernel initializes

Event Driven kernel

Multi-tasking, Multi-programming

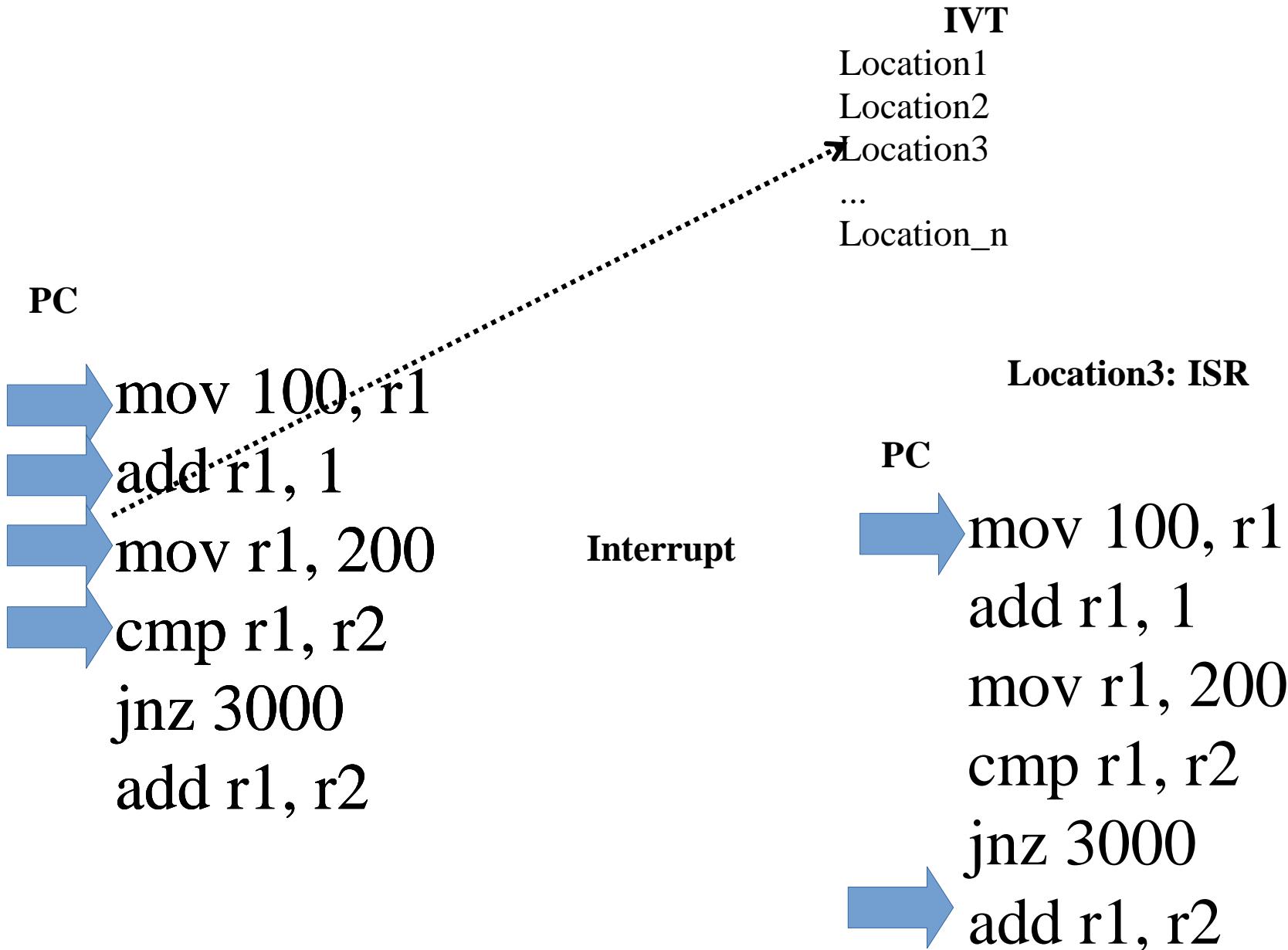
Understanding hardware interrupts

- Hardware devices (keyboard, mouse, hard disk, etc) can raise “hardware interrupts”
- Basically create an electrical signal on some connection to CPU (/bus)
- This is notified to CPU (in hardware)
- Now CPU’s normal execution is interrupted!
 - What’s the normal execution?
 - CPU will not continue doing the fetch, decode, execute, change PC cycle !
 - What happens then?

Understanding hardware interrupts

□ On Hardware interrupt

- The PC changes to a location pre-determined by CPU manufacturers!
- Now CPU resumes normal execution
 - What's normal?
 - Same thing: Fetch, Decode, Execute, Change PC, repeat!
 - But...
- But what's there at this pre-determined address?



Hardware interrupts and OS

When OS starts running initially

- It copies relevant parts of it's own code at all possible memory addresses that a hardware interrupt can lead to!**
- If there is an IVT in the hardware, OS sets up the IVT also!**
- Intelligent, isn't' it?**

Now what?

- Whenever there is a hardware interrupt – what will happen?**
- The PC will change to predetermined location**

Key points

- Understand the interplay of hardware features + OS code + clever combination of the two to achieve OS control over hardware
- Most features of computer systems / operating systems are derived from hardware features
 - We will keep learning this throughout the course
 - Hardware support is needed for many OS features

Time Shared CPU

- Timesharing happens after the OS has been loaded and Desktop environment is running
- The OS and different application programs keep executing on the CPU alternatively (more about this later)
 - The CPU is time-shared between different applications and OS itself
- How is this done?
 - kernel sets up the “timer register”

Multiprogramming

□ Program

- Just a binary (machine code) file lying on the hard drive. E.g. /bin/ls
- Does not do anything!

□ Process

- A program that is executing
- Must exist in RAM before it executes. Exec() does this.
- One program can run as multiple processes. What does that mean?

Multiprogramming

□ Multiprogramming

- A system where multiple processes(!) exist at the same time in the RAM
- But only one runs at a time!
 - Because there is only one CPU

□ Multi tasking

- Time sharing between multiple processes in a multi-programming system
- Timer interrupt used to achieve this.

Question

□ Select the correct one

- 1) A multiprogramming system is not necessarily multitasking
- 2) A multitasking system is not necessarily multiprogramming

Events , that interrupt CPU's functioning

□ Three types of “traps” : Events that make the CPU run code at a pre-defined address (given by IVT)

1) Hardware interrupts

2) Software interrupt instructions (trap)

E.g. instruction “INT”

3) Exceptions

e.g. a machine instruction that does division by zero

Illegal instruction, etc.

Some are called “faults”, e.g. “page fault”,

Multi tasking requirements

- Two processes should not be
 - Able to steal time of each other
 - See data/code of each other
 - Modify data/code of each other
 - Etc.
- The OS ensures all these things. How?
 - To be seen later.

**But the OS is “always” “running”
“in the background”
Isn’t it?**

Absolutely No!

**Let's understand
What kind of
Hardware, OS interplay
makes
Multitasking possible**

Two types of CPU instructions and two modes of CPU operation

- CPU instructions can be divided into two types

- Normal instructions

- mov, jmp, add, etc.

- Privileged instructions

- Normally related to hardware devices

- E.g.

- IN, OUT # write to I/O memory locations

- INTR # software interrupt, etc.

Two types of CPU instructions and two modes of CPU operation

- CPUs have a mode bit (can be 0 or 1)**
- The two values of the mode bit are called: User mode and Kernel mode**
- If the bit is in user mode**
 - Only the normal instructions can be executed by CPU**
- If the bit is in kernel mode**
 - Both the normal and privileged instructions can be executed by CPU**
- If the CPU is “made” to execute privileged**

Two types of CPU instructions and two modes of CPU operation

- The operating system code runs in kernel mode.**
 - How? Wait for that!**
- The application code runs in user mode**
 - How? Wait !**
 - So application code can not run privileged hardware instructions**
- Transition from user mode to kernel mode and vice-versa**
 - Special instruction called “software interrupt” instructions**

Software interrupt instruction

- E.g. INT on x86 processors

- Does two things at the same time!

- Changes mode from user mode to kernel mode in CPU
- + Jumps to a pre-defined location! Basically changes PC to a pre-defined value.
 - Close to the way a hardware interrupt works. Isn't it?
- Why two things together?
- What's there are the pre-defined location?

Software interrupt instruction

- What's the use of these type of instructions?
 - An application code running INT 0x80 on x86 will now cause
 - Change of mode
 - Jump into OS code
 - Effectively a request by application code to OS to do a particular task!
 - E.g. read from keyboard or write to screen !
 - OS providing hardware services to applications !

```
int main() {  
    int a = 2;  
    printf("hi\n");  
}
```

C Library

```
int printf("void *a, ...){  
...  
    write(1, a, ...);  
}
```

Code schematic

-----user-kernel-mode-boundary-----

//OS code

```
int sys_write(int fd, char  
*, int len) {
```

figure out location on
disk

where to do the write
and

carry out the

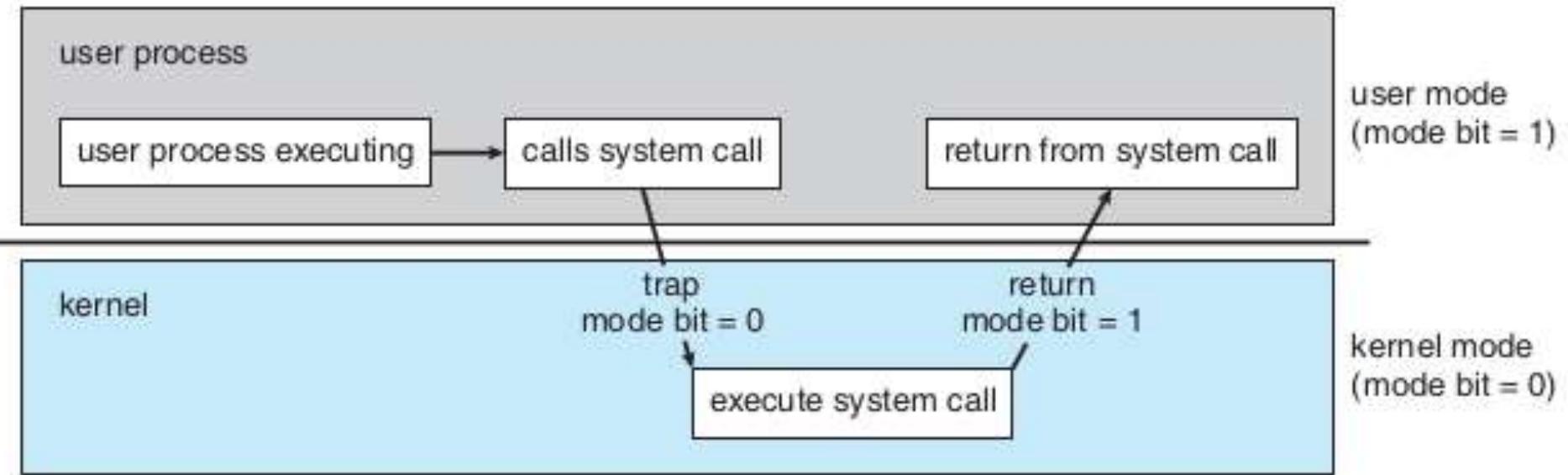


Figure 1.10 Transition from user to kernel mode.

Software interrupt instruction

- How does application code run INT instruction?
 - C library functions like printf(), scanf() which do I/O requests contain the INT instruction!
 - Control flow
 - Application code -> printf -> INT -> OS code -> back to printf code -> back to application code

Example: C program

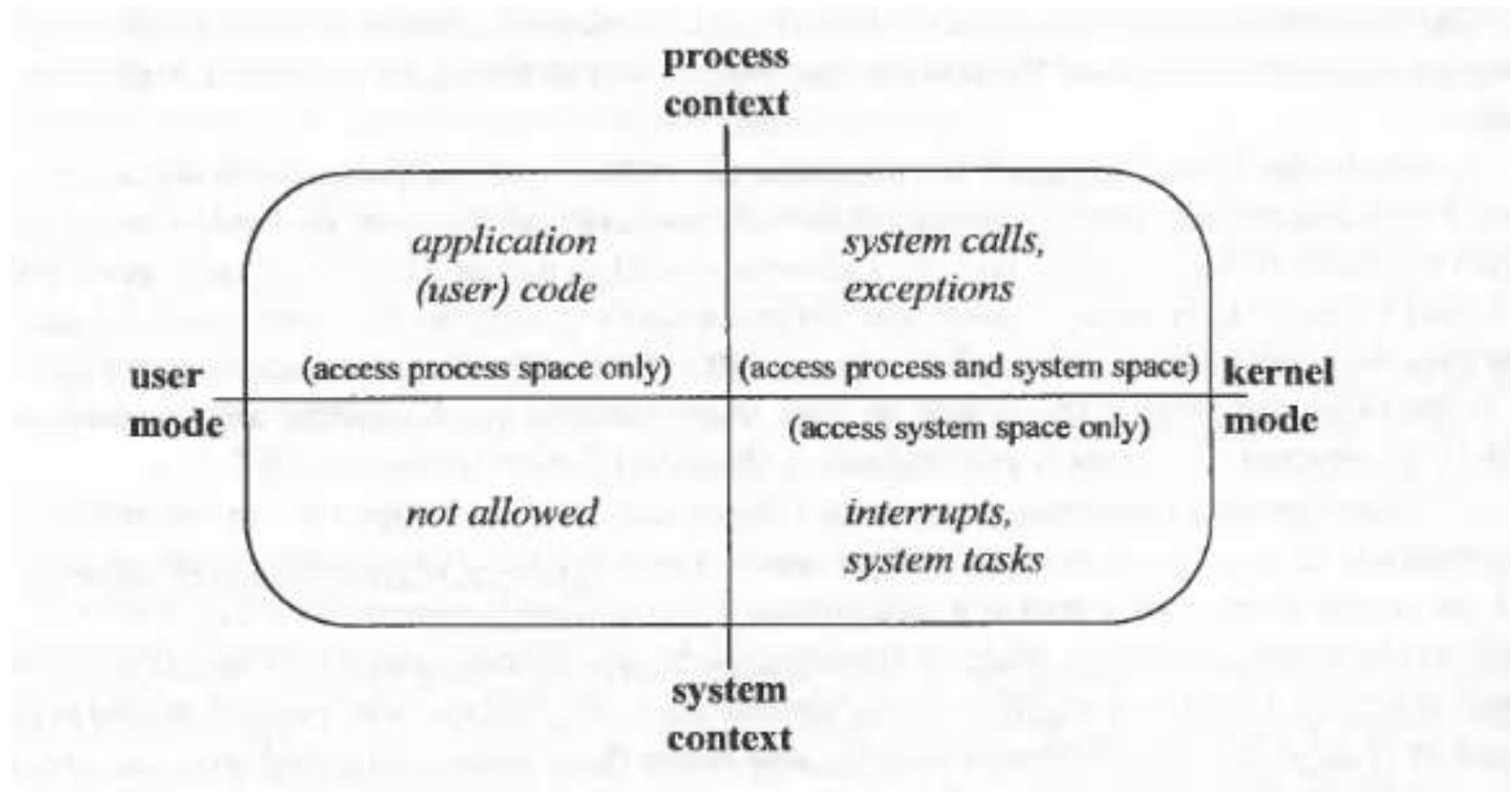
```
int main() {  
    int i, j, k;  
    k = 20;  
    scanf("%d", &i); // This jumps into OS and  
    returns back  
    j = k + i;  
    printf("%d\n", j); // This jumps into OS and  
    returns back  
    return 0;
```

Interrupt driven OS code

- OS code is sitting in memory , and runs intermittantly . When?
 - On a software or hardware interrupt or exception!
 - Event/Interrupt driven OS!
 - Hardware interrupts and exceptions occur asynchronously (un-predictedly) while software interrupts are caused by application code

What runs on the processor ?

- 4 possibilities.



Some system calls related to files

Abhijit A M

abhijit.comp@coep.ac.in

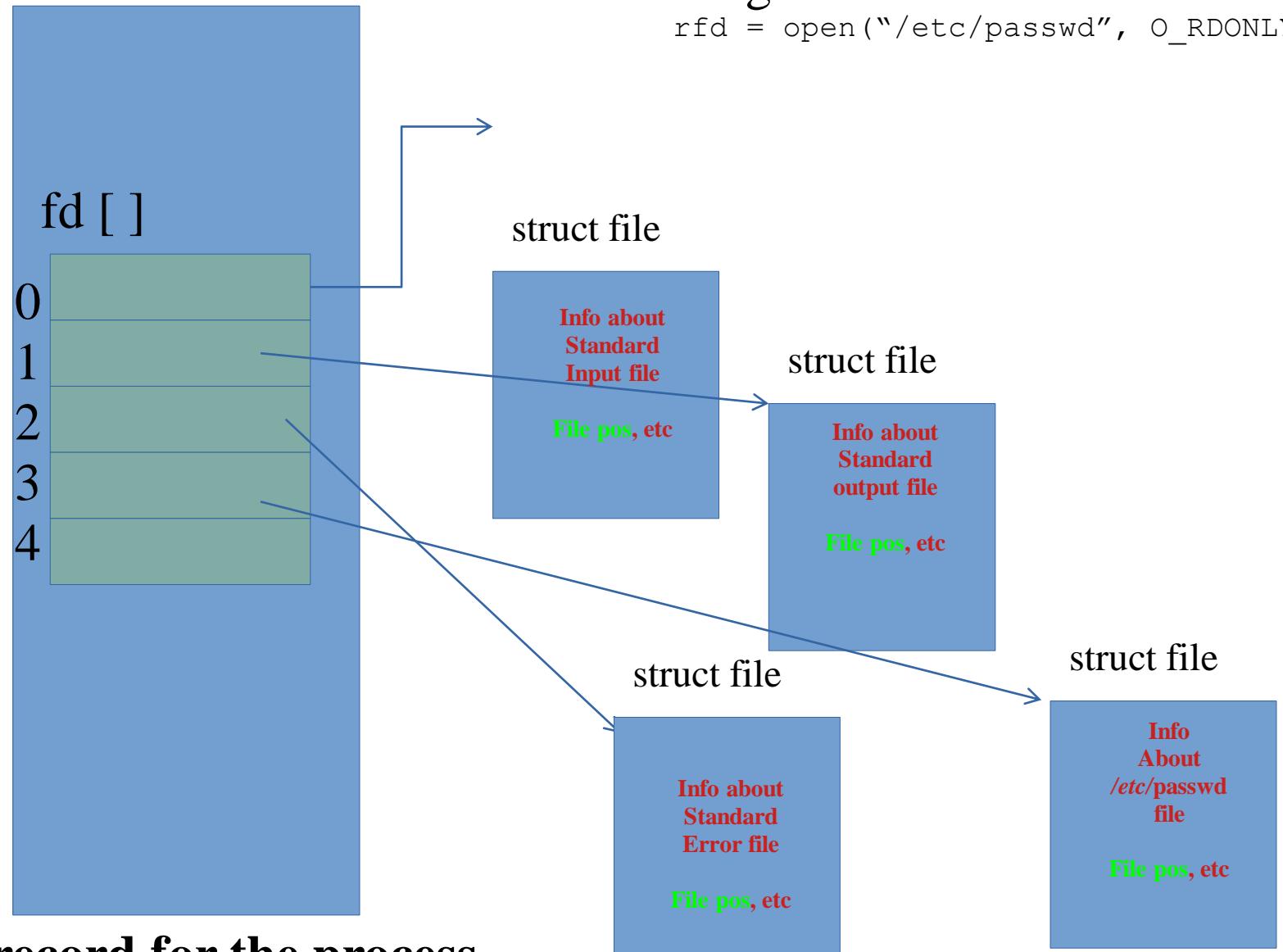
System Calls

- Kernel provided functions
- Run in Kernel mode
- Essentially invoked through the Software Interrupt instruction (“INT”)
 - INT -> Lookup in IVT -> jump to kernel code

List of open files

Program did

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



In kernel, record for the process

File system related system calls

- To get permission to “access” the file
 - open()
- To read sequentially from a file that has been open()
 - read()
- To write sequentially to a file that has been open()
 - write()
- To change “file position” anywhere
 - lseek()
- To release access to the file

```
int main(int argc, char *argv[]) {
```

```
    int fd;
```

Example: cat program

```
    char ch;
```

```
    fd = open(argv[1], O_RDONLY);
```

```
    if(fd == -1) {
```

```
        perror("mycat: ");
```

```
        exit(errno);
```

```
}
```

```
    while(read(fd, &ch, 1))
```

```
        putchar(ch);
```

```
    return 0;
```

```
}
```

```
fd = open(argv[1], O_RDONLY);  
if(fd == -1) {  
    perror("open failed:");  
    return errno;  
}  
  
fdw = open(argv[2], O_WRONLY | O_CREAT, S_IRUSR);  
if(fdw == -1) {  
    perror("open failed:");  
    return errno;  
}  
  
while(read(fd, &ch, 1))
```

Standard file descriptors

- `stdin (0), stdout(1), stderr(2)`
- Already open when a process begins
- Can be closed!
- `stdin`
 - Read from keyboard
- `stdout, stderr`
 - Write to screen, but two different

Standard file descriptors

.Stdin(0)

ch= getchar();

is equivalent to

read(0, &ch, 1);

.Stdout(1)

printf("hello")

is equivalent to

.Stderr(2)

fprintf(stderr,
‘hello’)

is equivalent to

write(2, “hello”,
5);

Redirection

- .Output redirection

```
close(1);
```

```
fd = open(..., O_WRONLY);
```

- .Input redirection

```
close(0);
```

```
fd = open(..., O_RDONLY);
```

dup()

- Duplicates a file descriptor
 - Essentially the “struct file *” in the kernel fdarray is copied !

•Example

```
fd = open(..., O_RDONLY);
```

```
close(0);
```

```
dup(fd);
```

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

Process Control Block

- 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



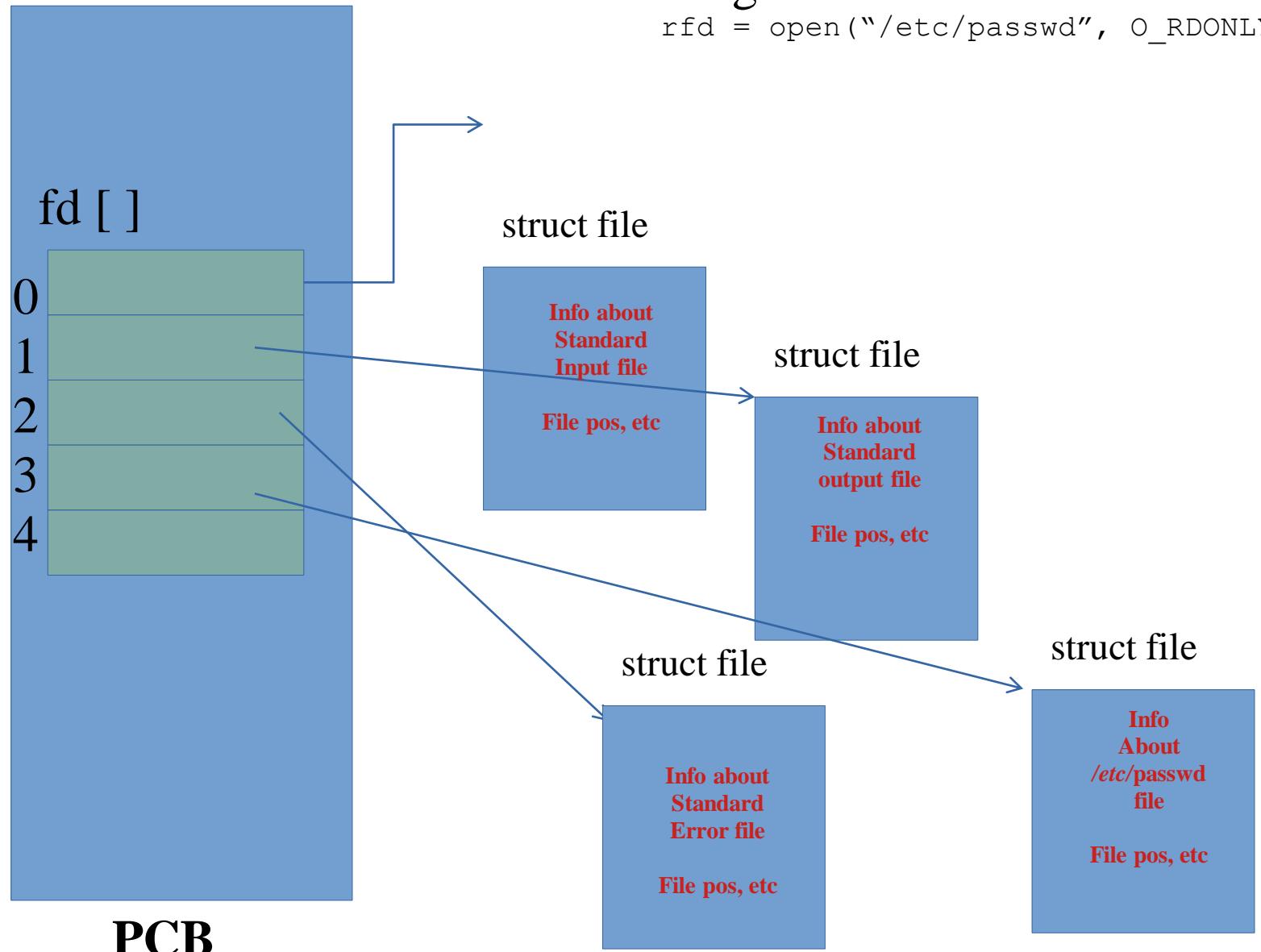
Figure 3.3 Process control block (PCB).

- Process ID (PID)
- Process State
- Program counter
- Registers (copy)
- Memory limits of the process
- Accounting information
- I/O status

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

```

// 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[NOFILE]; // Open files
    struct inode *cwd; // Current directory
    char name[16];    // Process name (debugging)
};

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

```

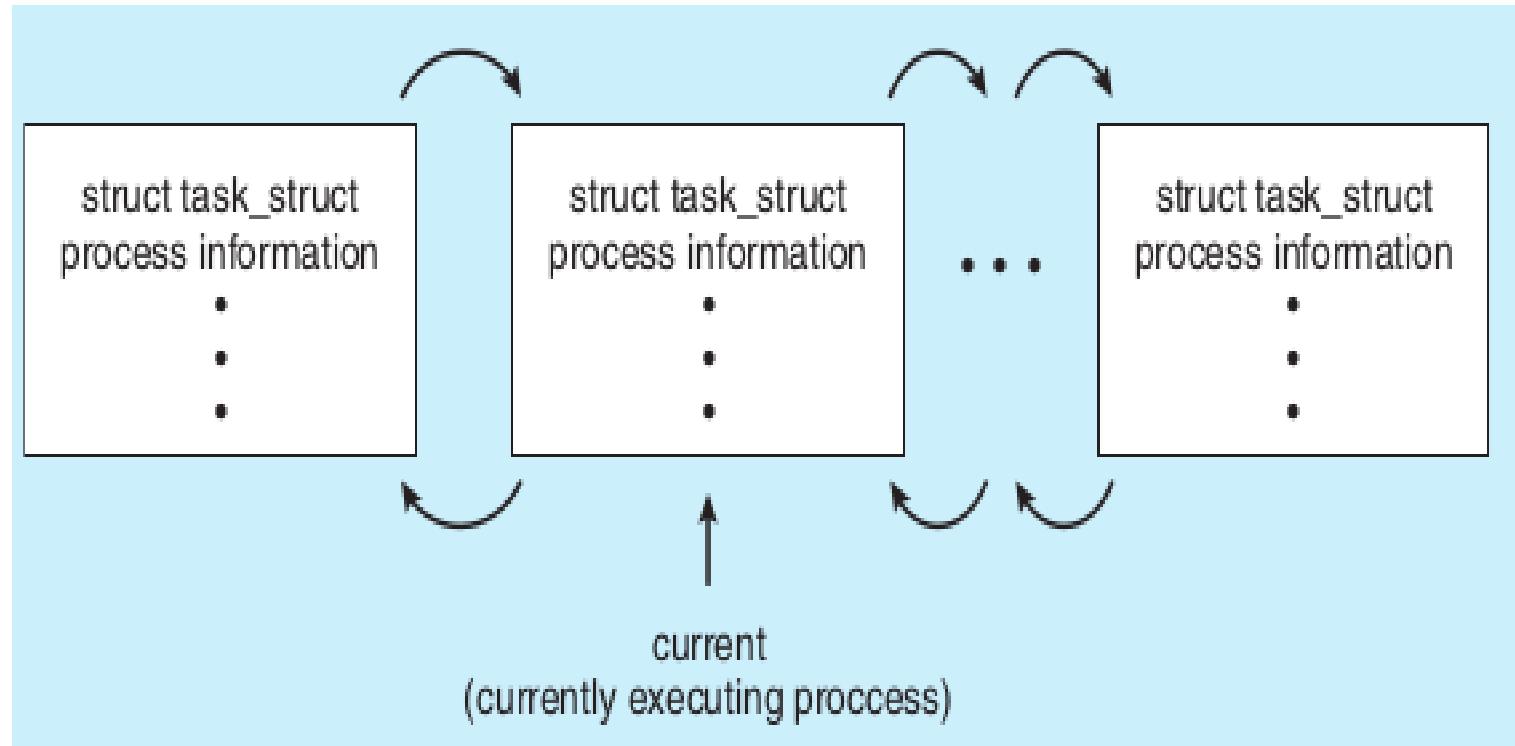
```

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

```

Process Queues/Lists inside OS

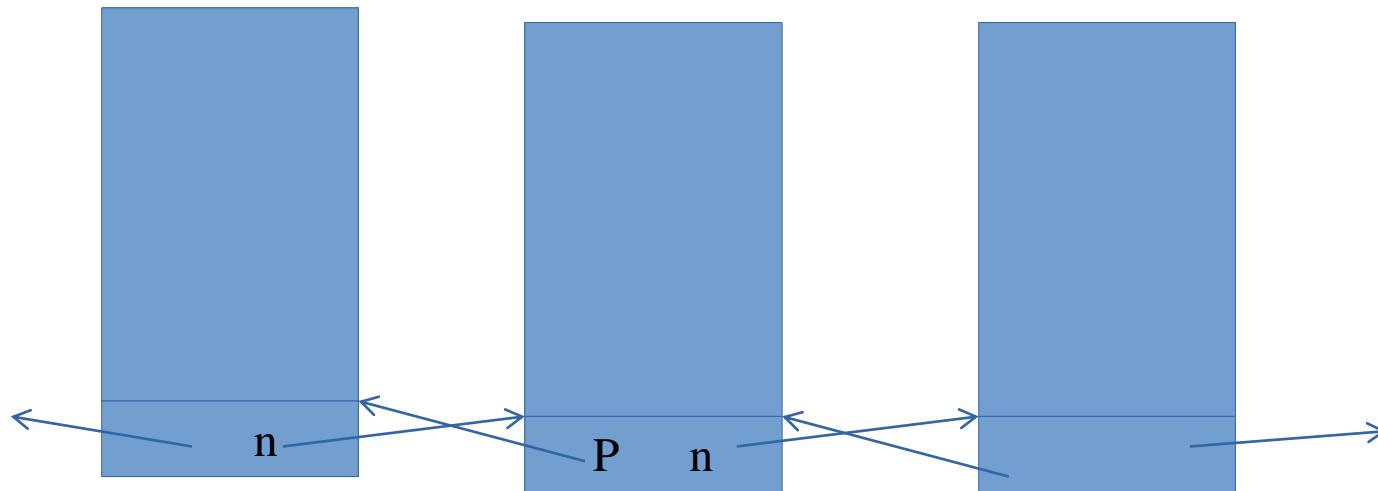
- Different types of queues/lists can be maintained by OS for the processes
 - A queue of processes which need to be scheduled
 - A queue of processes which have requested input/output to a device and hence need to be put on hold/wait
 - List of processes currently running on multiple CPUs



// 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;  
};
```



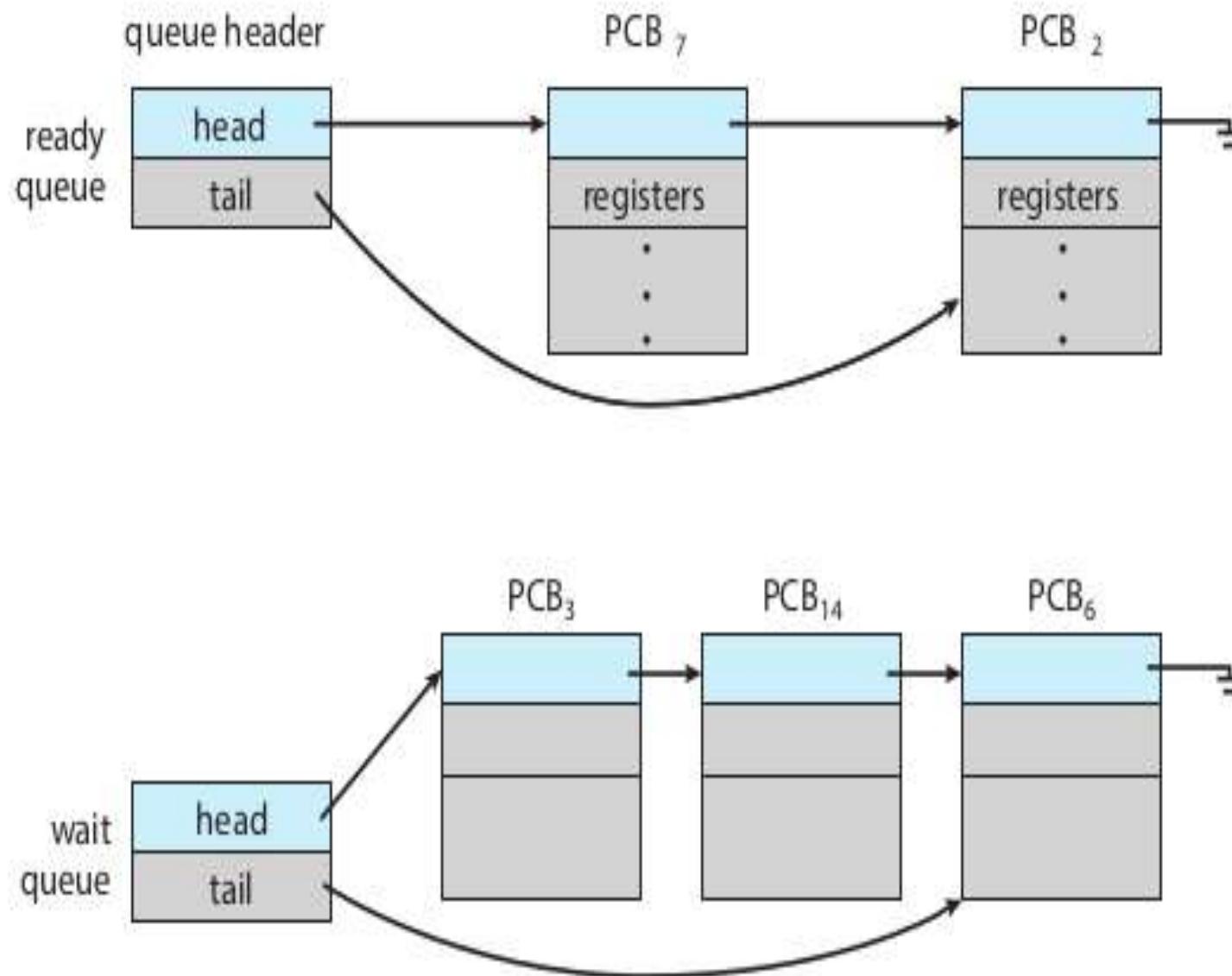


Figure 3.4 The ready queue and wait queues.

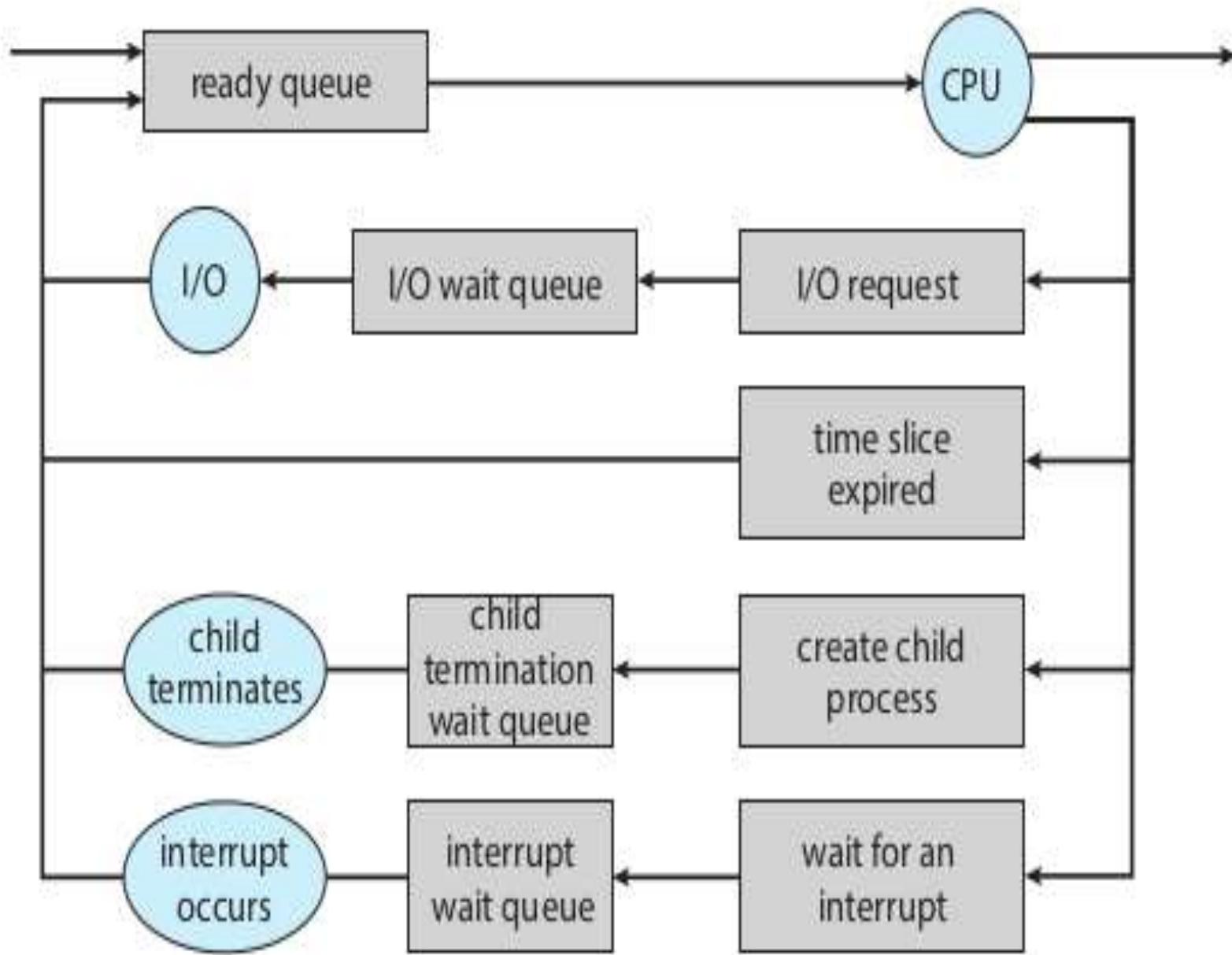


Figure 3.5 Queueing-diagram representation of process scheduling.

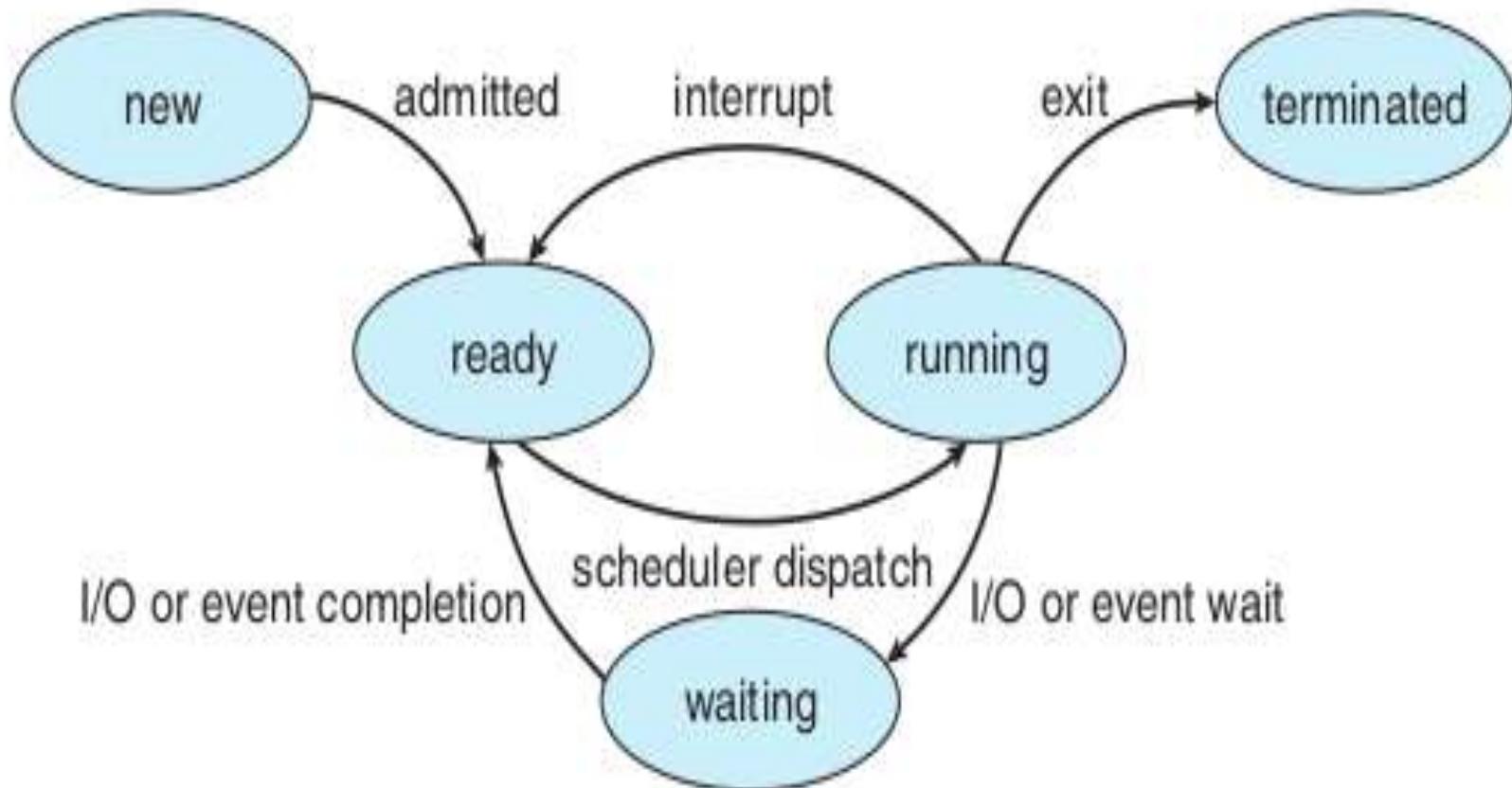


Figure 3.2 Diagram of process state.

Conceptual diagram

See state in output of
ps axu
(BSD style options, witho

On Linux

“Giving up” CPU by a process or blocking

```
int main() {  
    i = j + k;  
    scanf("%d", &k);  
}  
  
int scanf(char *x,  
...) {  
    ...
```

OS Syscall

```
sys_read(int fd,  
char *buf, int len) {  
    file f = current-  
        >fdarray[fd];  
    int offset = f-  
        >position;  
    ...
```

“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

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

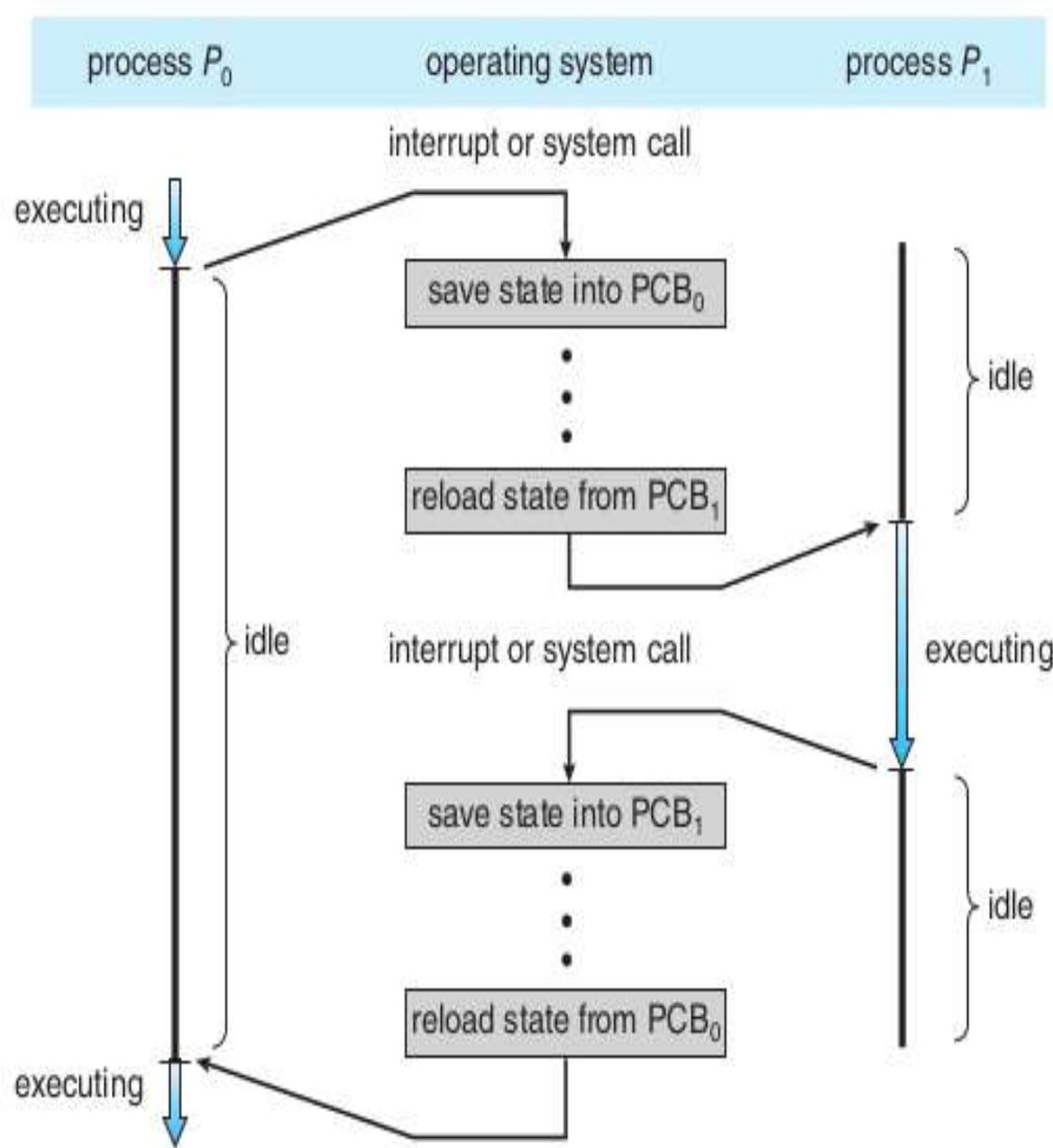


Figure 3.6 Diagram showing context switch from process to process.

Peculiarity of context switch

- When a process is running, the function calls work in LIFO fashion
 - Made possible due to calling convention (a protocol for using processor stack for passing parameters and returning values, used by compilers to generate machine code)
- When an interrupt occurs

Compilation, Linking, Loading

Abhijit A M

Review of last few lectures

Boot sequence: BIOS, boot-loader, kernel

- Boot sequence: Process world

- kernel->init -> many forks+execs() ->

- Hardware interrupts, system calls, exceptions

- Event driven kernel

- System calls

- Fork exec open read

What are compiler, assembler, linker and loader, and C library

System Programs/Utilities

Most essential to make a kernel really usable

Standard C Library

- A collection of some of the most frequently needed functions for C programs
 - `scanf`, `printf`, `getchar`, system-call wrappers (`open`, `read`, `fork`, `exec`, etc.), ...
- An machine/object code file containing the machine code of all these functions
 - Not a source code! Neither a header file. More later.

Compiler

- application program, which converts one (programming) language to another

- Most typically compilers convert a high level language like C, C++, etc. to Machine code language

- E.g. GCC /usr/bin/gcc

- Usage: e.g.



Assembler

- application program, converts assembly code into machine code

- What is assembly language?

- Human readable machine code language.

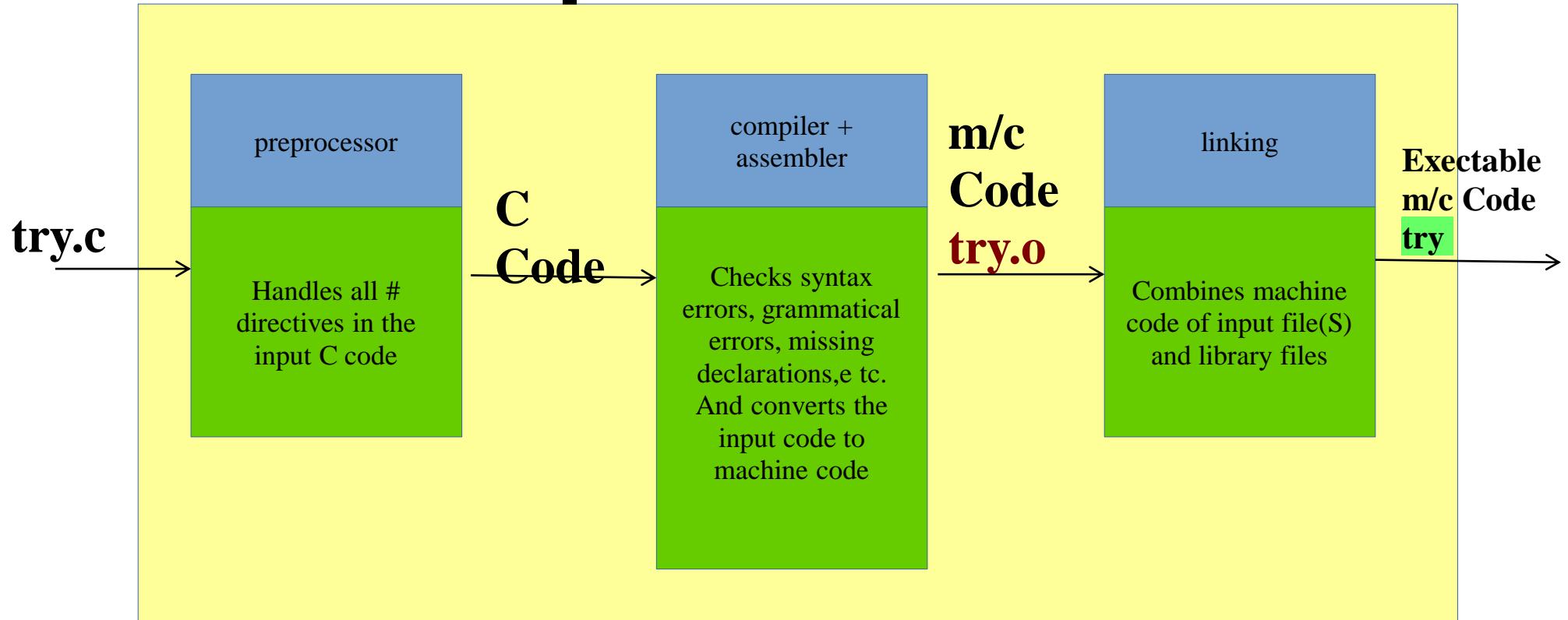
- E.g. x86 assembly code

- mov 50, r1

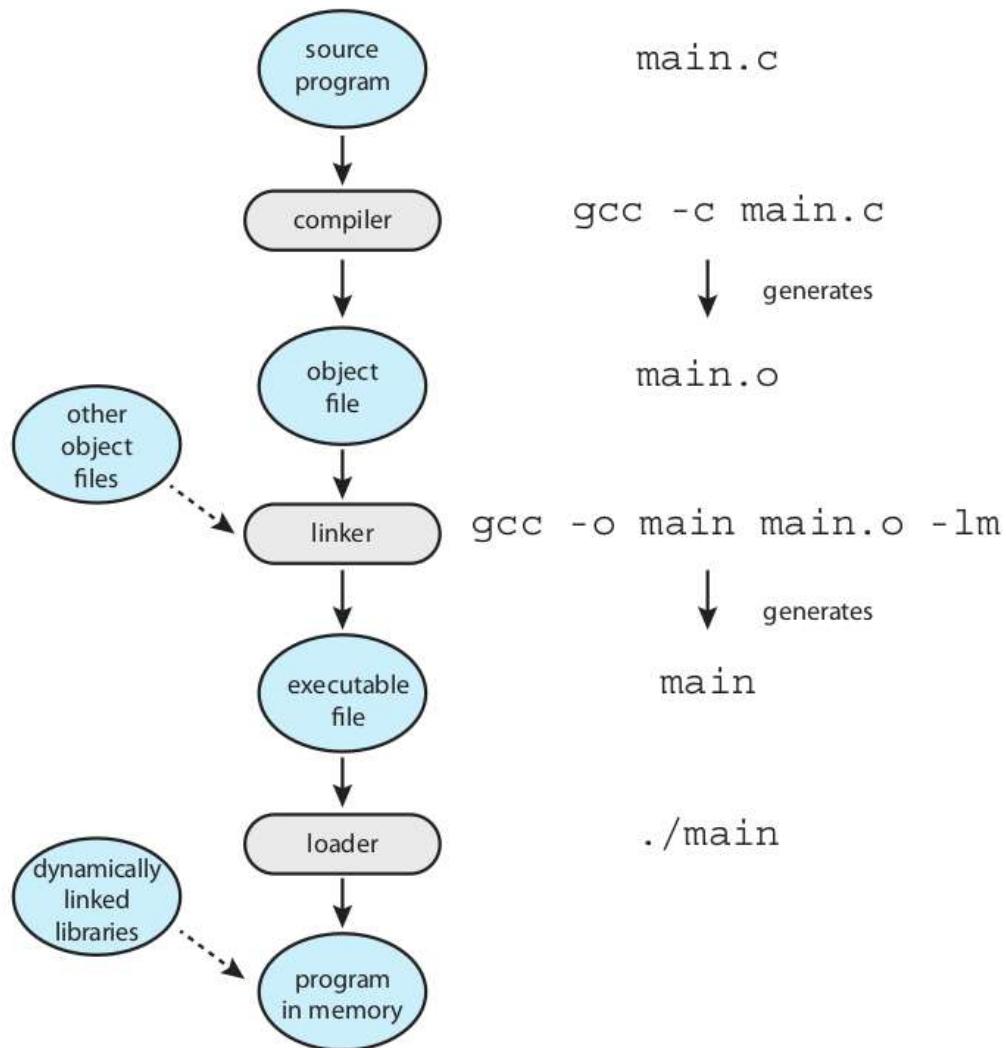
- add 10, r1



Compilation Process



gcc



.From the
textbook

Figure 2.11 The role of the linker and loader.

Example

try.c

```
#include <stdio.h>
#define MAX 30
int f(int, int);
int main() {
    int i, j, k;
    scanf("%d%d", &i, &j);
    k = f(i, j) + MAX;
    printf("%d\n", k);
    return 0;
}
```

f.c

```
int g(int);
#define ADD(a, b) (a + b)
int f(int m, int n) {
    return ADD(m,n) + g(10);
}
```

g.c

```
int g(int x) {
    return x + 10;
}
```

Try these commands, observe the output/errors/warnings, and try to und

```
$ gcc try.c
$ gcc -c try.c
$ gcc -c f.c
$ gcc -c g.c
$ gcc try.o f.o g.o -o try
$ gcc -E try.c
$ gcc -E f.c
```

More about the steps

• Pre-processor

– `#define ABC XYZ`

• cut ABC and paste XYZ

– `# include <stdio.h>`

• copy-paste the file stdio.h

• There is no CODE in stdio.h, only typedefs, #includes, #define, #ifdef, etc.

• Linking

– Normally links with the standard C-library by default

– To link with other libraries, use the -l option of gcc

Using gcc itself to understand the process

- Run only the preprocessor

- cc -E test.c

- Shows the output on the screen

- Run only till compilation (no linking)

- cc -c test.c

- Generates the "test.o" file , runs compilation + assembler

- gcc -S main.c

- One step before machine code generation, stops at assembly code

- Combine multiple .o files (only linking part)

- cc test.o main.o -o myprogram

Linking steps

- Linker is an application program

- On linux, it's the "ld" program

- E.g. you can run commands like \$ ld a.o b.o -o c.o

- Normally you have to specify some options to ld to get a proper executable file.

- When you run gcc

- \$ cc main.o f.o g.o -o try

- the CC will internally invoke "ld". ld does the job of linking

Linking steps

.The resultant file "try" here, will contain the codes of all the functions and linkages also.

.What is linking?

– "connecting" the call of a function with the code of the function.

.What happens with the code of printf()?

– The linker or CC will automatically pick up code from the libc.so.6 file for the functions.

Executable file format

- An executable file needs to execute in an environment created by OS and on a particular processor
 - Contains machine code + other information for OS
 - Need for a structured-way of storing machine code in it
- ELF : The format on Linux.
 - Try this
 - \$ file /bin/ls
 - \$ file /usr/lib/x86_64-linux-gnu/libc-2.31.so

Exec() and ELF

- When you run a program
 - \$./try
 - Essentially there will be a fork() and exec("./try", ...)
 - So the kernel has to read the file "./try" and understand it.
 - So each kernel will demand its own object code file format.
 - Hence ELF, EXE, etc. Formats
- ELF is used not only for executable (complete machine code) programs, but also for partially compiled files e.g. main.o and library files like libc.so.6
- What is a.out?
 - "a.out" was the name of a format used on earlier Unixes.
 - It so happened that the early compiler-writers also created

Utilities to play with object code files

.objdump

-\$ objdump -D -x /bin/ls

-Shows all disassembled machine instructions and "headers"

.hexdump

-\$ hexdump /bin/ls

-Just shows the file in hexadecimal

readelf

.ar

-To create a "statically linked" library file

-\$ ar -crs libmine.a one.o two.o
.Gcc to create shared library

-\$ gcc hello.o -shared -o libhello.so

.To see how gcc invokes as, ld, etc; do this

-\$ gcc -v hello.c -o hello

Linker, Loader, Link-Loader

- Linker or linkage-editor or link-editor

- The “ld” program. Does linking.

- Loader

- The exec(). It loads an executable in the memory.

- Link-Loader

- Often the linker is called link-loader in literature.
Because where were days when the linker and

Static, dynamic / linking, loading

- Both linking and loading can be
 - Static or dynamic
 - More about this when we learn memory management
- An important fundamental:
 - memory management features of processor, memory management architecture of kernel, executable/object code file format, output of

Cross-compiler

- .Compiler on system-A, but generate object-code file for system-B (target system)
 - E.g. compile on Ubuntu, but create an EXE for windows
- .Normally used when there is no compiler available on target system
 - see gcc -m option

Calling Convention

Abhijit A M

The need for calling convention

An essential task of the compiler

Generates object code (file) for given source code (file)

Processors provide simple features

Registers, machine instructions (add, mov, jmp, call, etc.),
imp registers like stack-pointer, etc; ability to do
byte/word sized operations

No notion of data types, functions, variables, etc.

But languages like C provide high level features

Data types, variables, functions, recursion, etc

The need for calling convention

**Examples of some of the challenges before the compiler
“call” + “ret” does not make a C-function call!**

**A “call” instruction in processor simply does this
Pushes IP(that is PC) on stack + Jumps to given address
This is not like calling a C-function !**

Unsolved problem: How to handle parameters, return value?

Processor does not understand data types!

Although it has instructions for byte, word sized data and

Compiler and Machine code generation

Example, code inside a function

```
int a, b, c;
```

```
c = a + b;
```

What kind of code is generated by compiler for this?

sub 12, <esp> #normally local variables are located on stack, make space

mov <location of a in memory>, r1 #location is on stack, e,g. -4(esp)

mov <location of b in memory>, r2

Compiler and Machine code generation

Across function calls

```
int f(int m, n) {  
    int x = m, y = n;  
    return g(x, y);  
}
```

```
int x(int a) {  
    return g (a, a+ 1);  
}  
  
int g(int p, int q) {
```

**g() may be called from f() or
from x()**

**Sequence of function calls
can NOT be predicted by
compiler**

**Compiler has to generate
machine code for each
function assuming nothing
about the caller**

Compiler and Machine code generation

Machine code generation for functions

Mapping C language features to existing machine code instructions.

Typical examples

a =100 ; ==> mov instruction

a = b+ c; ==> mov, add instructions

while(a < 5) { j++; } ==> mov, cmp, jlt, add, etc. Instruction

Where are the local variables in memory?

The only way to store them is on a stack.

Function calls

LIFO

Last in First Out

Must need a “stack” like feature to implement them

Processor Stack

Processors provide a stack pointer

%esp on x86

Instructions like push and pop are provided by hardware

They automatically increment/decrement the stack
pointer. On x86 stack grows downwards (subtract from

Function calls

System stack, compilers, Languages

Processor provides us with esp (stack) and push/pop instructions.

The esp pointer is initialized to a proper value at the time of fork-exec by the OS for each process. Then process runs.

Knowing the above, compilers go ahead with generating machine code using the esp.

This means, Langauges like C which provide for function calls, and recursion can only run on processors which

Activation Record

Local Vars + parameters + return address

When functions call each other

One activation record is built on stack for each function call

On function return, the record is destroyed

On x86

ebp and esp pointers are used to denote the activation record.

How? We will see soon. You may start exploring with “gcc

X86 instructions

leave

Equivalent to

**mov %ebp, %esp # esp = push %eip
ebp**

pop %ebp

ret

Equivalent to

pop %ecx

jmp %ecx

call x

Equivalent to

**push %eip
jmp x**

X86 instructions

endbr64

Normally a NOP

Let's see some examples now

Let's compile using

gcc -S

See code and understand

simple.c and simple.s

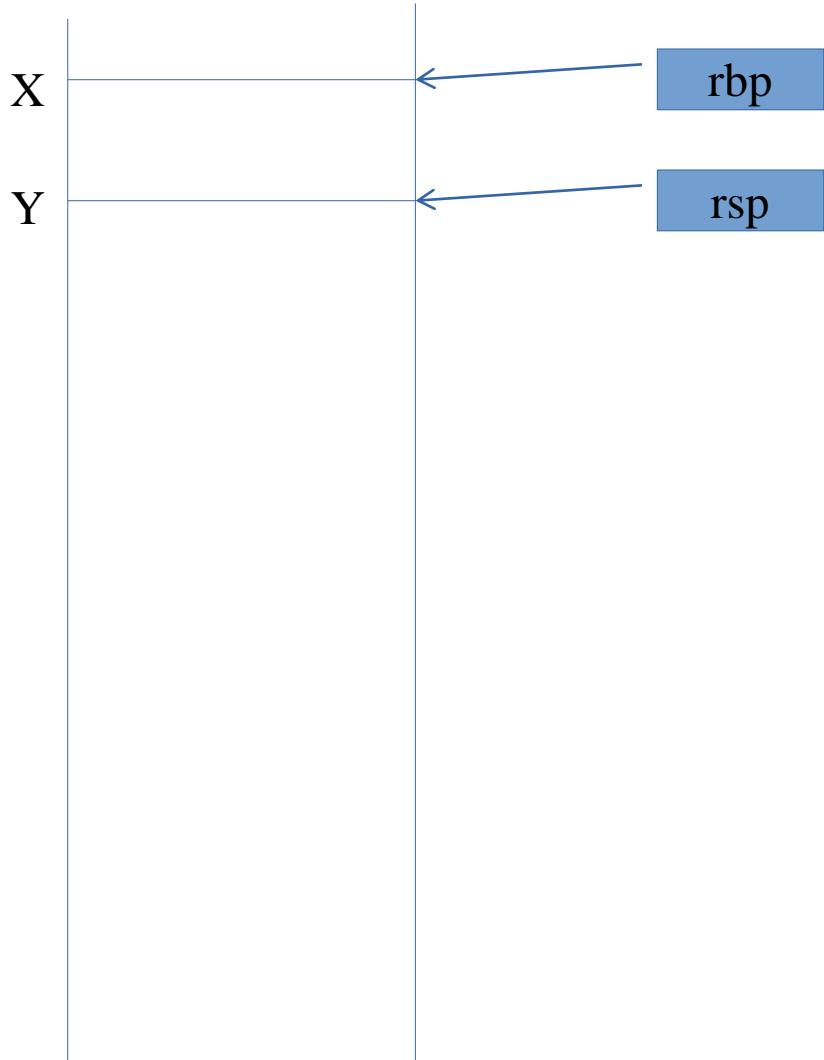
```
int f(int x) {
    int y;
    y = x + 3;
    return y;
}
int main() {
    int a = 20, b = 30;
    b = f(a);
    return b;
}
```

main:

```
    endbr64
    pushq %rbp
    movq %rsp, %rbp
    subq $16, %rsp
    movl $20, -8(%rbp)
    movl $30, -4(%rbp)
    movl -8(%rbp), %eax
    movl %eax, %edi
    call f
    movl %eax, -4(%rbp)
    movl -4(%rbp), %eax
    leave
    ret
```

f:

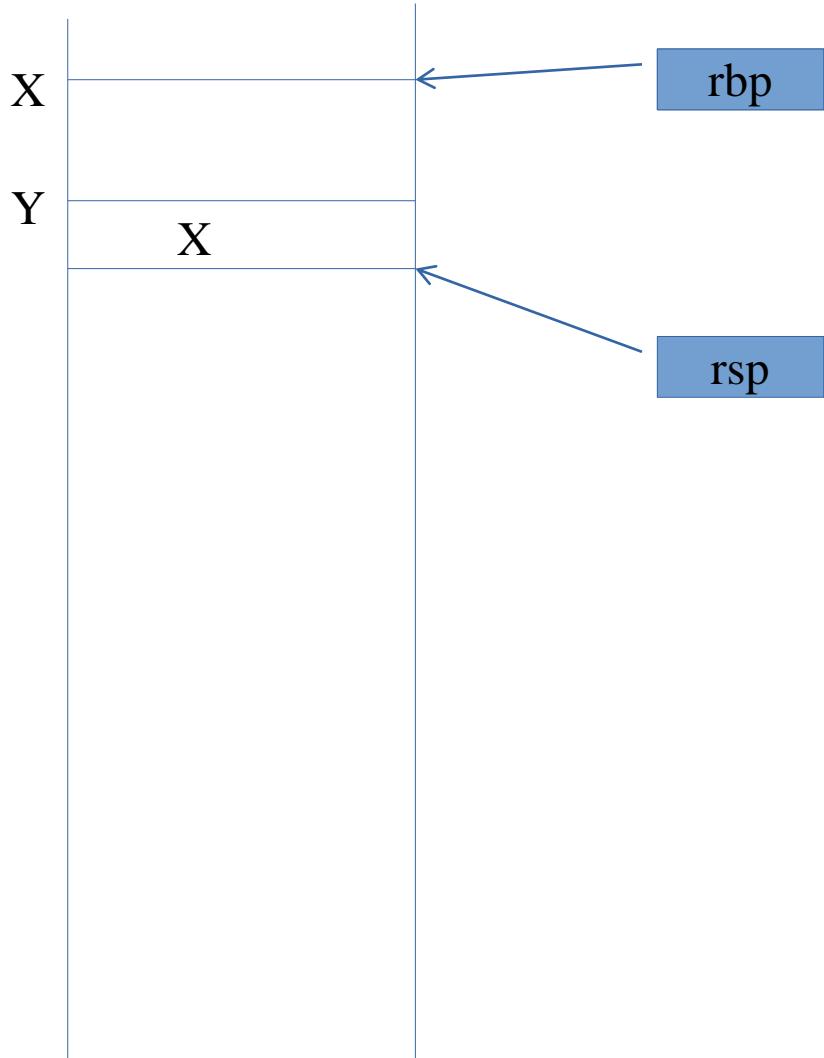
```
    endbr64
    pushq %rbp
    movq %rsp, %rbp
    movl %edi, -20(%rbp)
    movl -20(%rbp), %eax
    addl $3, %eax
    movl %eax, -4(%rbp)
    movl -4(%rbp), %eax
    popq %rbp
    ret
```



main:

```
endbr64
pushq %rbp
movq %rsp, %rbp
subq $16, %rsp
movl $20, -8(%rbp)
movl $30, -4(%rbp)
movl -8(%rbp), %eax
movl %eax, %edi
call f
movl %eax, -4(%rbp)
movl -4(%rbp), %eax
leave
ret
```

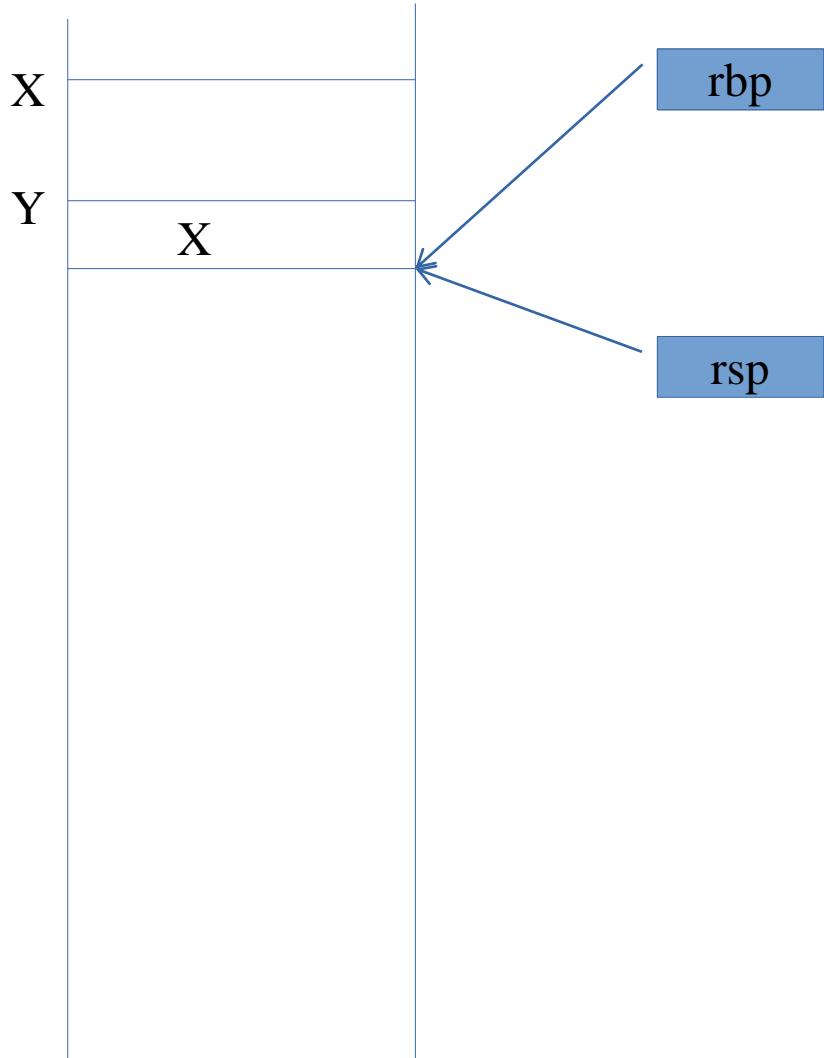
```
int main() {
    int a = 20, b = 30;
    b = f(a);
    return b;
}
```



main:

```
endbr64  
pushq %rbp  
movq %rsp, %rbp  
subq $16, %rsp  
movl $20, -8(%rbp)  
movl $30, -4(%rbp)  
movl -8(%rbp), %eax  
movl %eax, %edi  
call f  
movl %eax, -4(%rbp)  
movl -4(%rbp), %eax  
leave  
ret
```

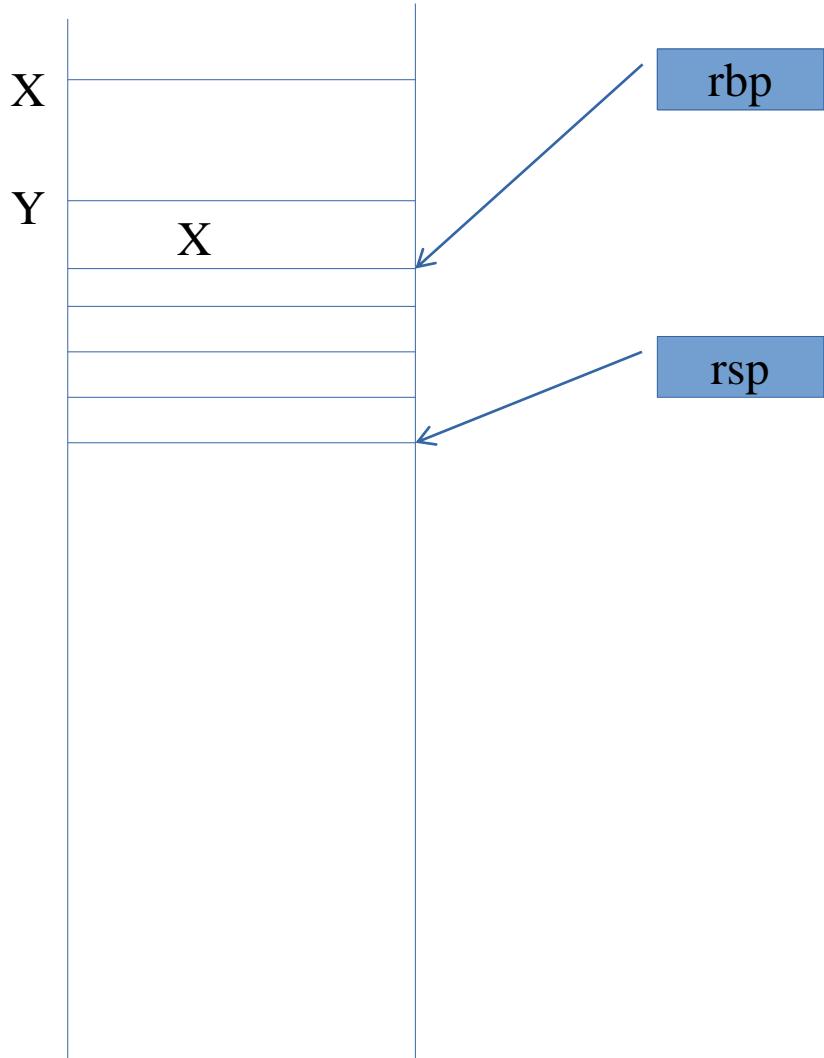
```
int main() {  
    int a = 20, b = 30;  
    b = f(a);  
    return b;  
}
```



main:

```
endbr64
pushq %rbp
movq %rsp, %rbp
subq $16, %rsp
movl $20, -8(%rbp)
movl $30, -4(%rbp)
movl -8(%rbp), %eax
movl %eax, %edi
call f
movl %eax, -4(%rbp)
movl -4(%rbp), %eax
leave
ret
```

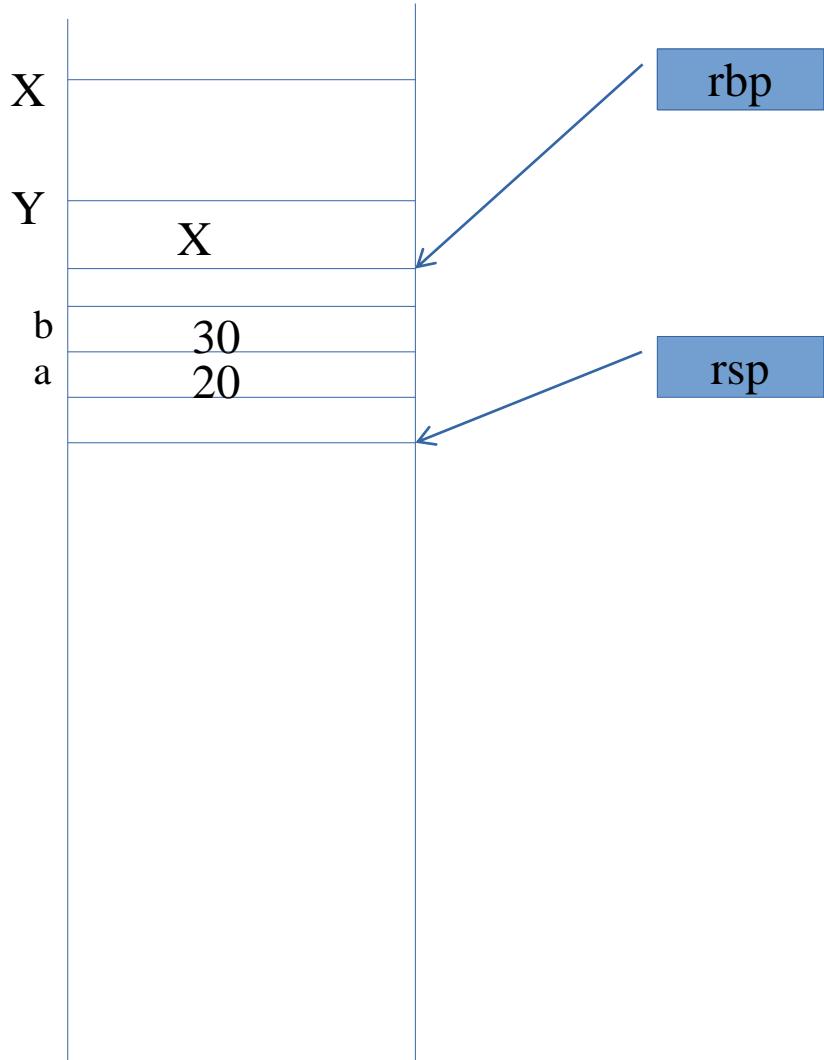
```
int main() {
    int a = 20, b = 30;
    b = f(a);
    return b;
}
```



main:

```
endbr64  
pushq %rbp  
movq %rsp, %rbp  
subq $16, %rsp  
movl $20, -8(%rbp)  
movl $30, -4(%rbp)  
movl -8(%rbp), %eax  
movl %eax, %edi  
call f  
movl %eax, -4(%rbp)  
movl -4(%rbp), %eax  
leave  
ret
```

```
int main() {  
    int a = 20, b = 30;  
    b = f(a);  
    return b;  
}
```



main:

```

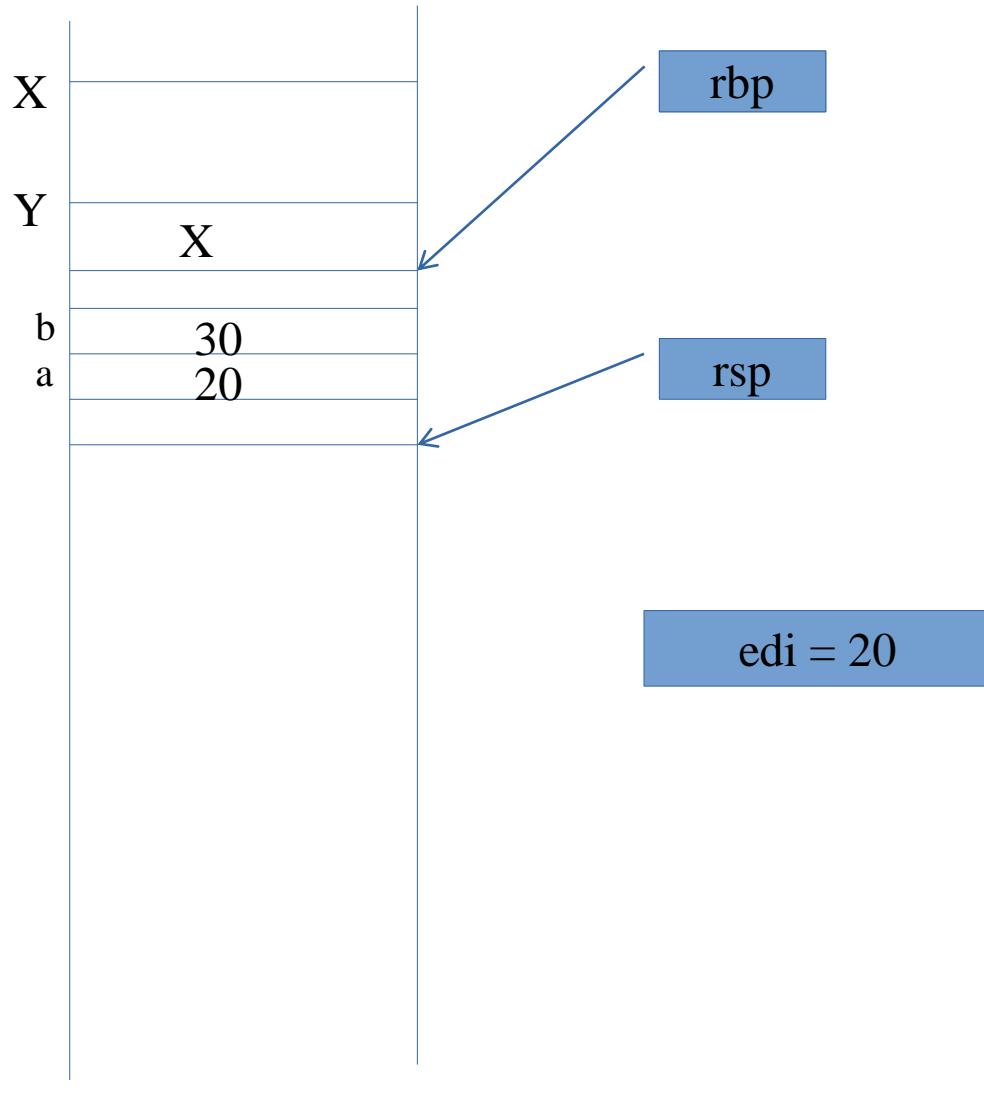
endbr64
pushq %rbp
movq %rsp, %rbp
subq $16, %rsp
movl $20, -8(%rbp)
movl $30, -4(%rbp)
movl -8(%rbp), %eax
movl %eax, %edi
call f
movl %eax, -4(%rbp)
movl -4(%rbp), %eax
leave
ret

```

```

int main() {
    int a = 20, b = 30;
    b = f(a);
    return b;
}

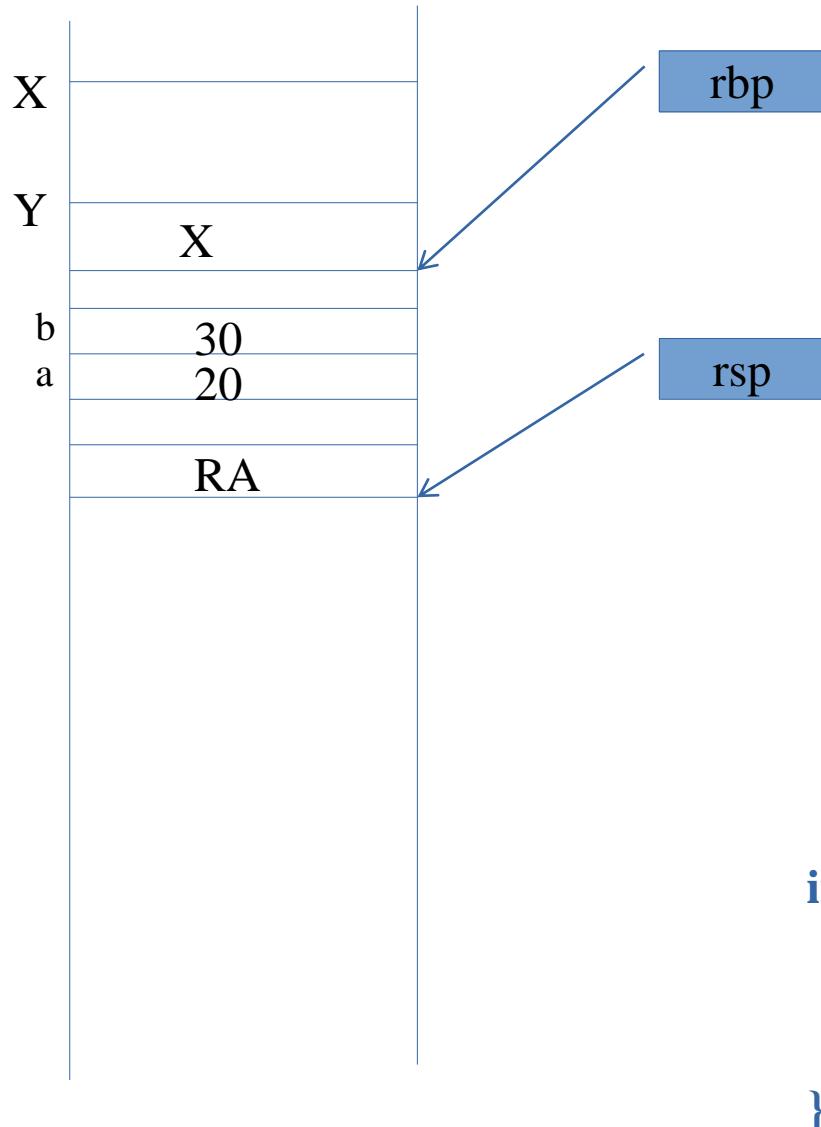
```



main:

```
endbr64  
pushq %rbp  
movq %rsp, %rbp  
subq $16, %rsp  
movl $20, -8(%rbp)  
movl $30, -4(%rbp)  
movl -8(%rbp), %eax  
movl %eax, %edi  
call f  
movl %eax, -4(%rbp)  
movl -4(%rbp), %eax  
leave  
ret
```

```
int main() {  
    int a = 20, b = 30;  
    b = f(a);  
    return b;  
}
```



```

main:
endbr64
pushq %rbp
movq %rsp, %rbp
subq $16, %rsp
movl $20, -8(%rbp)
movl $30, -4(%rbp)
movl -8(%rbp), %eax
movl %eax, %edi
call f
movl %eax, -4(%rbp)
movl -4(%rbp), %eax
Leave
ret

```

```

int f(int x) {
    int y;
    y = x + 3;
    return y;
}

```

```

f:
endbr64
pushq %rbp
movq %rsp, %rbp
movl %edi, -20(%rbp)
movl -20(%rbp), %eax
addl $3, %eax
movl %eax, -4(%rbp)
movl -4(%rbp), %eax
Rpopq %rbp
ret

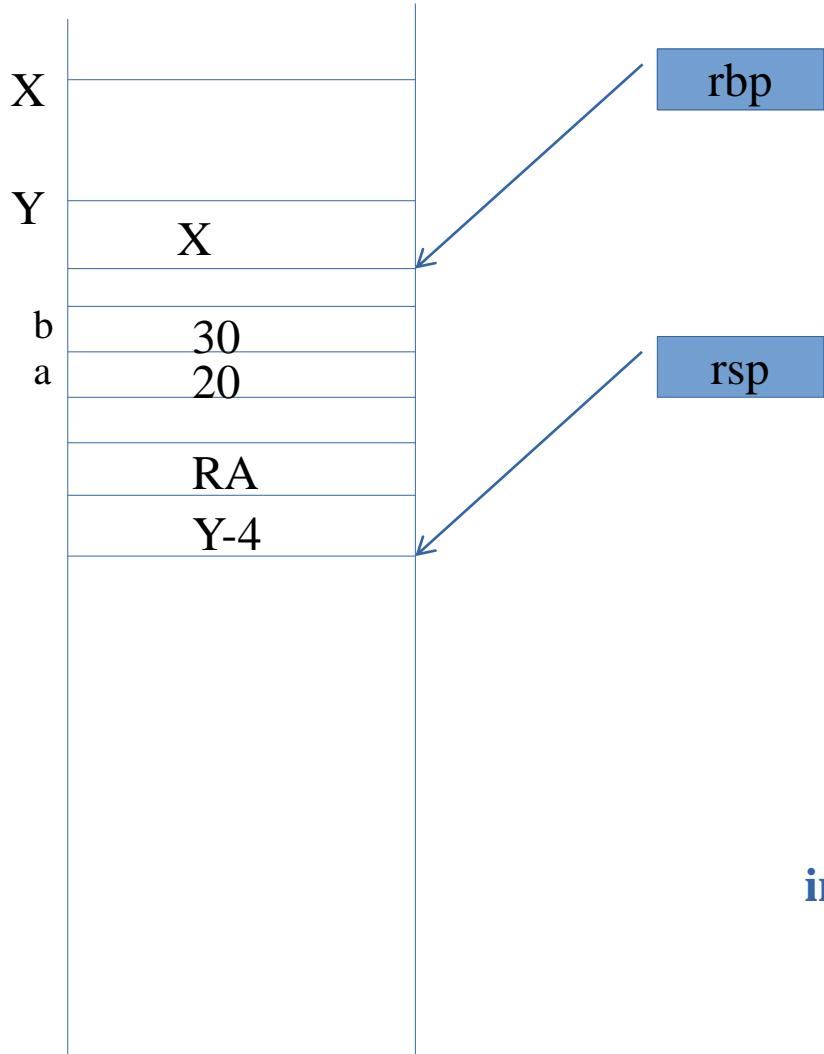
```

edi = 20

```

int main() {
    int a = 20, b = 30;
    b = f(a);
    return b;
}

```



```
main:
endbr64
pushq %rbp
movq %rsp, %rbp
subq $16, %rsp
movl $20, -8(%rbp)
movl $30, -4(%rbp)
movl -8(%rbp), %eax
movl %eax, %edi
call f
movl %eax, -4(%rbp)
movl -4(%rbp), %eax
Leave
ret
```

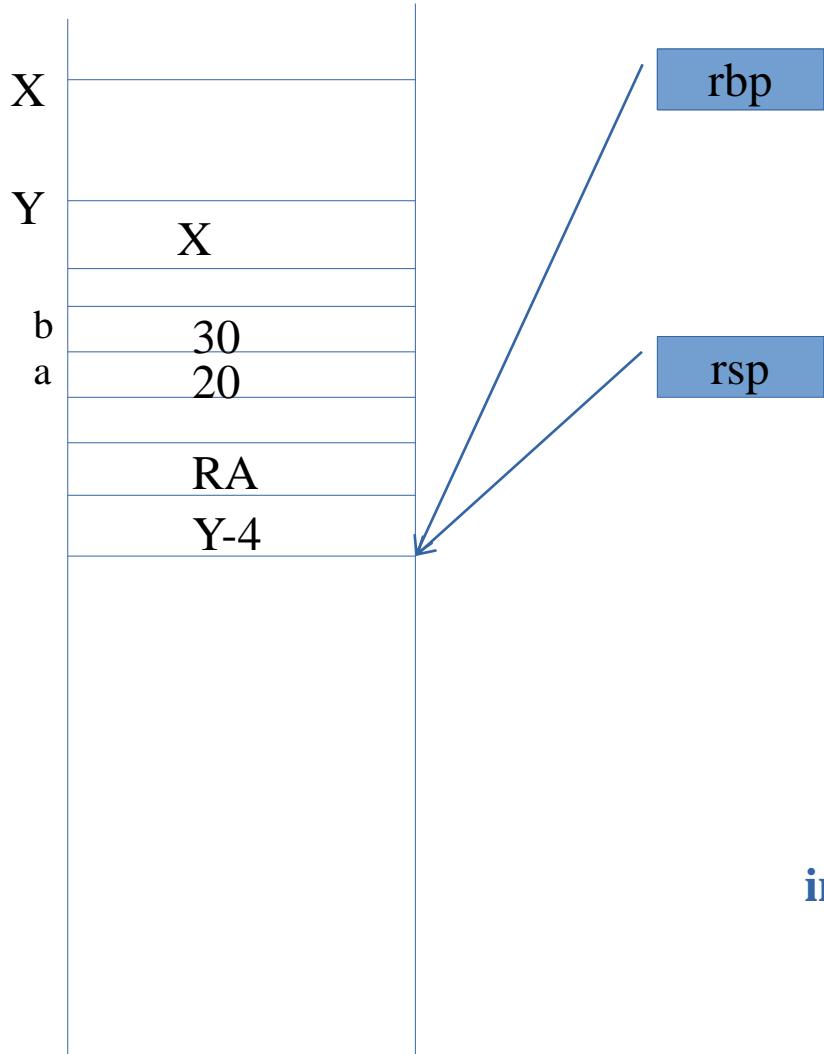
`f:`

```
endbr64
pushq %rbp
movq %rsp, %rbp
movl %edi, -20(%rbp)
movl -20(%rbp), %eax
addl $3, %eax
movl %eax, -4(%rbp)
movl -4(%rbp), %eax
Rpopq %rbp
ret
```

`edi = 20`

```
int f(int x) {
    int y;
    y = x + 3;
    return y;
}
```

```
int main() {
    int a = 20, b = 30;
    b = f(a);
    return b;
}
```



```
main:
endbr64
pushq %rbp
movq %rsp, %rbp
subq $16, %rsp
movl $20, -8(%rbp)
movl $30, -4(%rbp)
movl -8(%rbp), %eax
movl %eax, %edi
call f
movl %eax, -4(%rbp)
movl -4(%rbp), %eax
Leave
ret
```

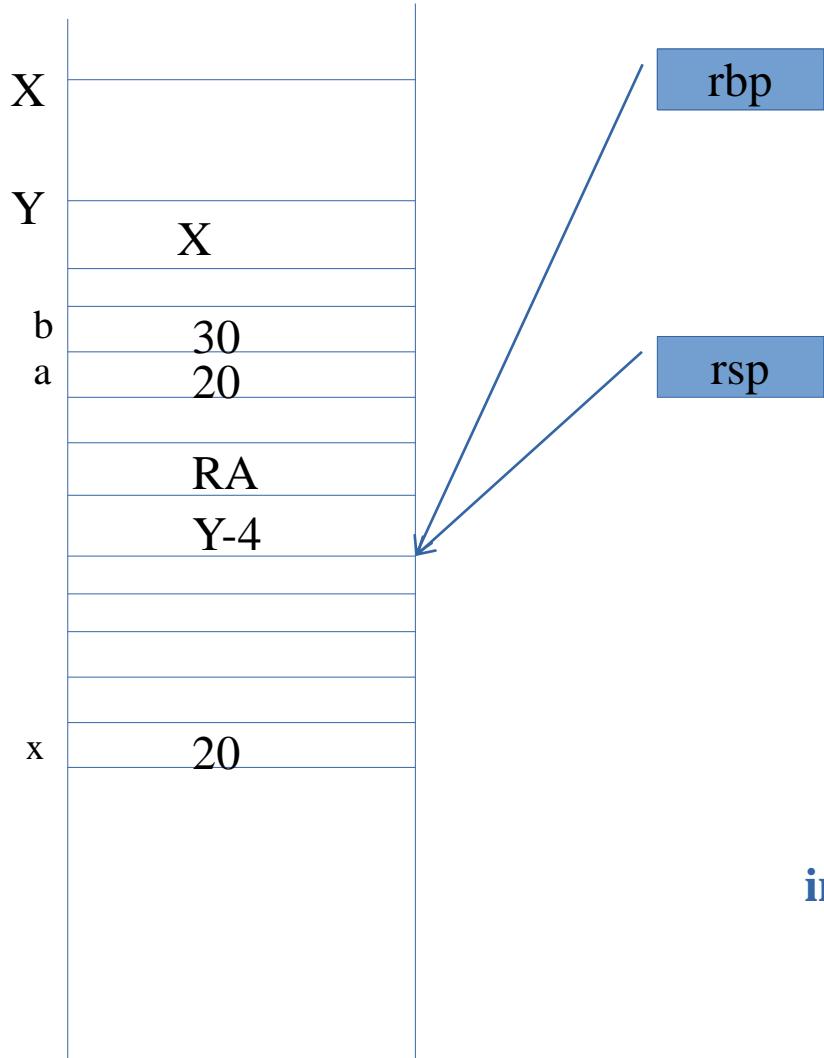
```
int f(int x) {
    int y;
    y = x + 3;
    return y;
}
```

f:

```
endbr64
pushq %rbp
movq %rsp, %rbp
movl %edi, -20(%rbp)
movl -20(%rbp), %eax
addl $3, %eax
movl %eax, -4(%rbp)
movl -4(%rbp), %eax
Rpopq %rbp
ret
```

edi = 20

```
int main() {
    int a = 20, b = 30;
    b = f(a);
    return b;
}
```



```
int f(int x) {
    int y;
    y = x + 3;
    return y;
}
```

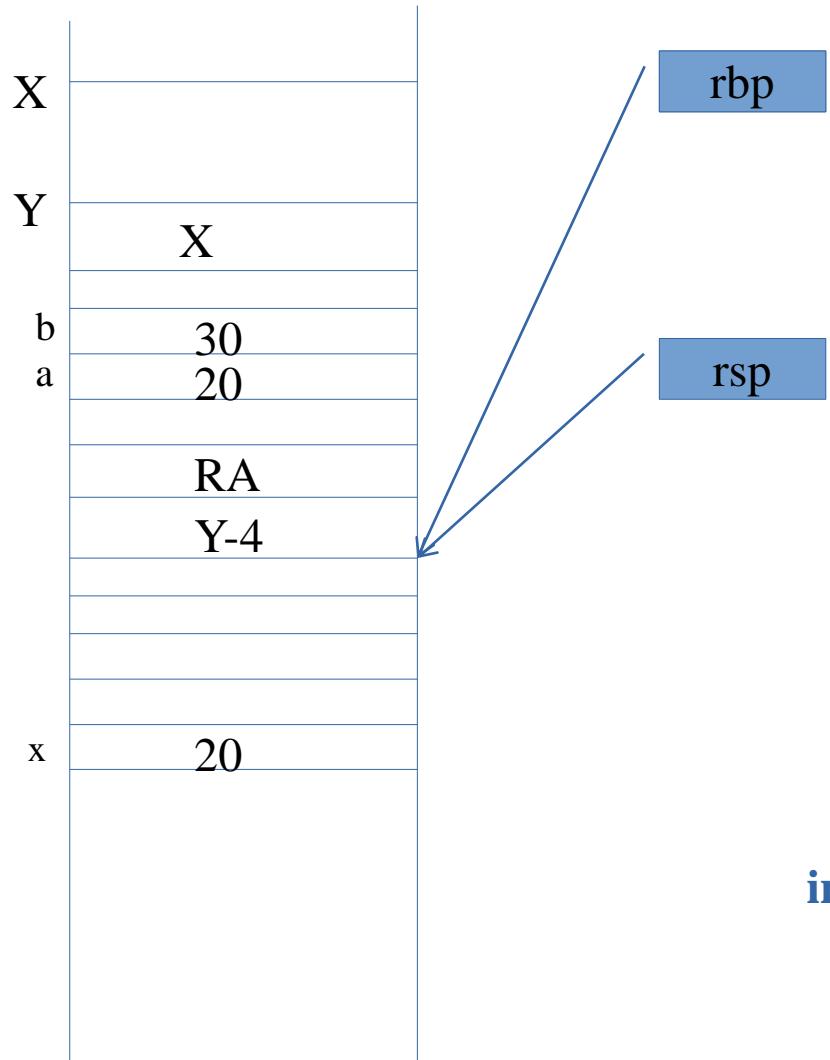
```
main:
endbr64
pushq %rbp
movq %rsp, %rbp
subq $16, %rsp
movl $20, -8(%rbp)
movl $30, -4(%rbp)
movl -8(%rbp), %eax
movl %eax, %edi
call f
movl %eax, -4(%rbp)
movl -4(%rbp), %eax
Leave
ret
```

f:

```
endbr64
pushq %rbp
movq %rsp, %rbp
movl %edi, -20(%rbp)
movl -20(%rbp), %eax
addl $3, %eax
movl %eax, -4(%rbp)
movl -4(%rbp), %eax
Rpopq %rbp
ret
```

edi = 20

```
int main() {
    int a = 20, b = 30;
    b = f(a);
    return b;
}
```



```
int f(int x) {
    int y;
    y = x + 3;
    return y;
}
```

```
main:
endbr64
pushq %rbp
movq %rsp, %rbp
subq $16, %rsp
movl $20, -8(%rbp)
movl $30, -4(%rbp)
movl -8(%rbp), %eax
movl %eax, %edi
call f
movl %eax, -4(%rbp)
movl -4(%rbp), %eax
Leave
ret
```

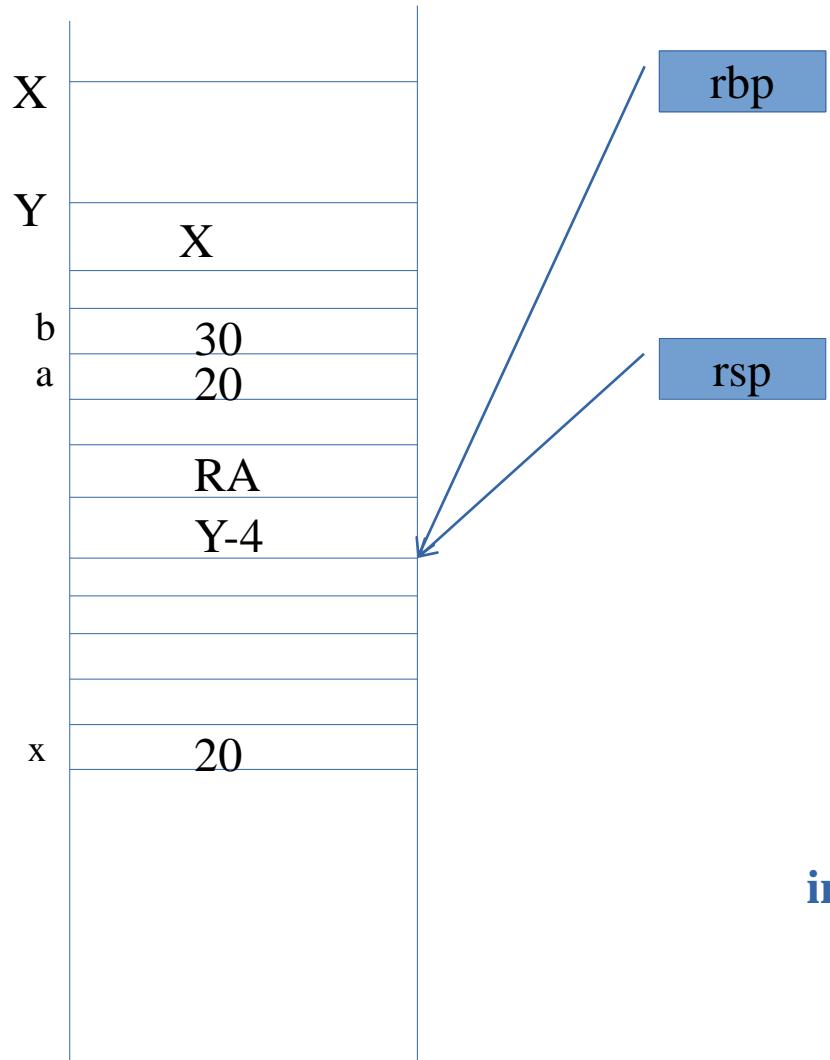
f:

```
endbr64
pushq %rbp
movq %rsp, %rbp
movl %edi, -20(%rbp)
movl -20(%rbp), %eax
addl $3, %eax
movl %eax, -4(%rbp)
movl -4(%rbp), %eax
RApopq %rbp
ret
```

edi = 20

eax = 20

```
int main() {
    int a = 20, b = 30;
    b = f(a);
    return b;
}
```



```
int f(int x) {
    int y;
    y = x + 3;
    return y;
}
```

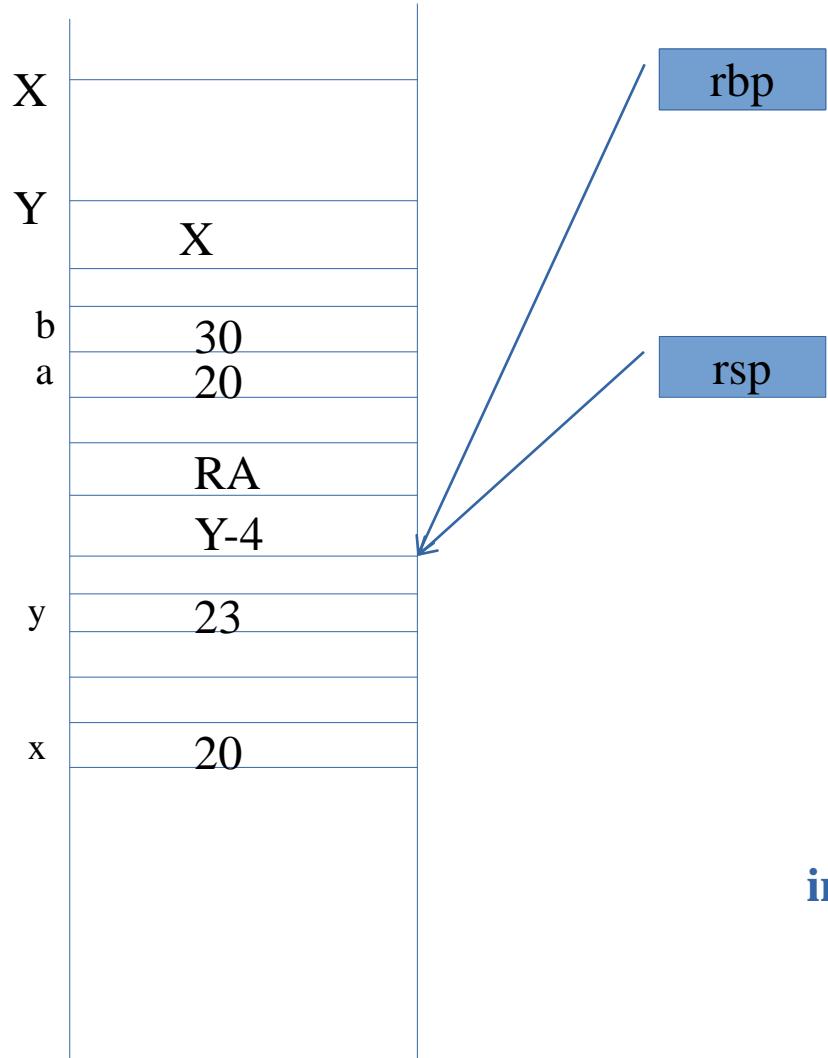
```
main:
endbr64
pushq %rbp
movq %rsp, %rbp
subq $16, %rsp
movl $20, -8(%rbp)
movl $30, -4(%rbp)
movl -8(%rbp), %eax
movl %eax, %edi
call f
movl %eax, -4(%rbp)
movl -4(%rbp), %eax
Leave
ret
```

```
f:
endbr64
pushq %rbp
movq %rsp, %rbp
movl %edi, -20(%rbp)
movl -20(%rbp), %eax
addl $3, %eax
movl %eax, -4(%rbp)
movl -4(%rbp), %eax
RApopq %rbp
ret
```

edi = 20

eax = 23

```
int main() {
    int a = 20, b = 30;
    b = f(a);
    return b;
}
```



```
int f(int x) {
    int y;
    y = x + 3;
    return y;
}
```

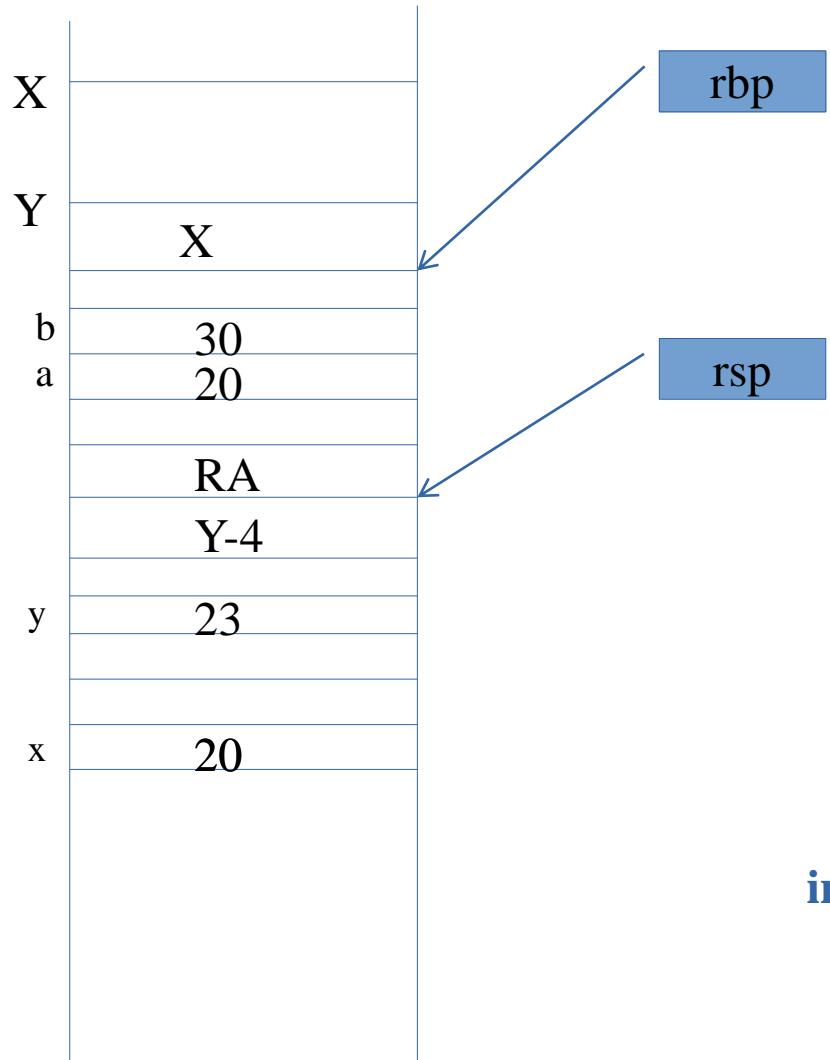
```
main:
endbr64
pushq %rbp
movq %rsp, %rbp
subq $16, %rsp
movl $20, -8(%rbp)
movl $30, -4(%rbp)
movl -8(%rbp), %eax
movl %eax, %edi
call f
movl %eax, -4(%rbp)
movl -4(%rbp), %eax
Leave
ret
```

```
f:
endbr64
pushq %rbp
movq %rsp, %rbp
movl %edi, -20(%rbp)
movl -20(%rbp), %eax
addl $3, %eax
movl %eax, -4(%rbp)
movl -4(%rbp), %eax
RApopq %rbp
ret
```

edi = 20

eax = 23

```
int main() {
    int a = 20, b = 30;
    b = f(a);
    return b;
}
```



```
int f(int x) {
    int y;
    y = x + 3;
    return y;
}
```

```
main:
endbr64
pushq %rbp
movq %rsp, %rbp
subq $16, %rsp
movl $20, -8(%rbp)
movl $30, -4(%rbp)
movl -8(%rbp), %eax
movl %eax, %edi
call f
movl %eax, -4(%rbp)
movl -4(%rbp), %eax
Leave
ret
```

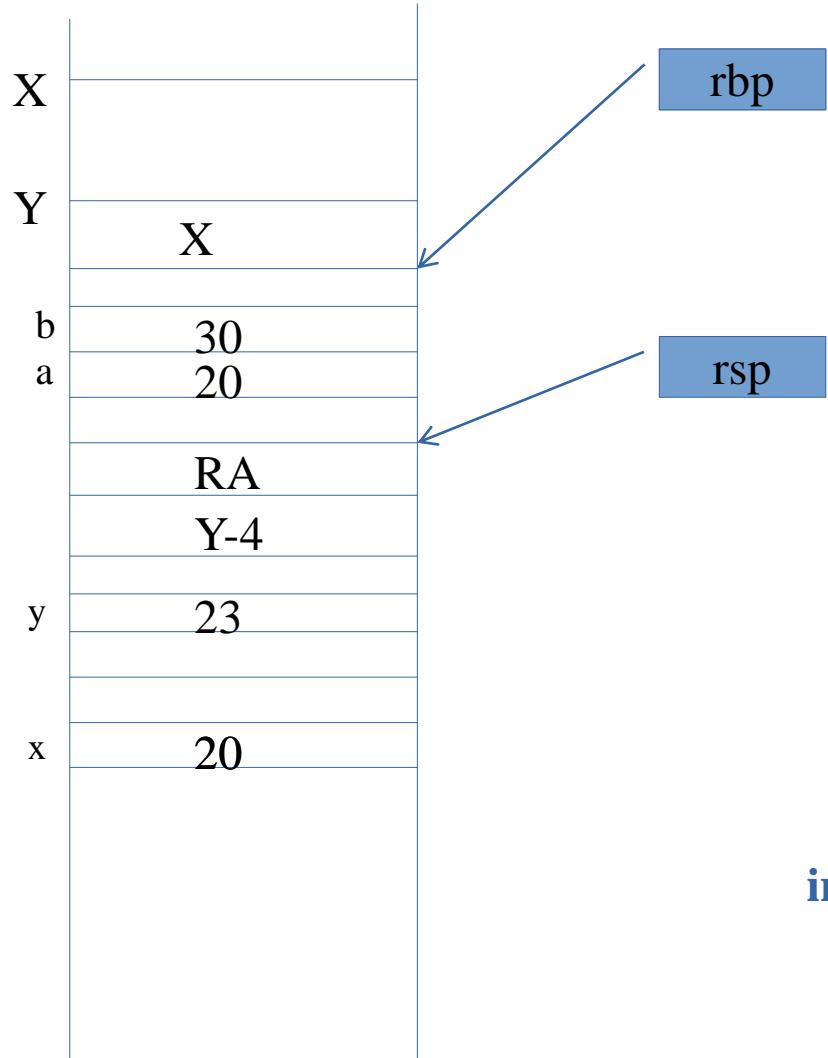
`f:`

```
endbr64
pushq %rbp
movq %rsp, %rbp
movl %edi, -20(%rbp)
movl -20(%rbp), %eax
addl $3, %eax
movl %eax, -4(%rbp)
movl -4(%rbp), %eax
RApopq %rbp
ret
```

`edi = 20`

`Eax = 23`

```
int main() {
    int a = 20, b = 30;
    b = f(a);
    return b;
}
```



```
main:
endbr64
pushq %rbp
movq %rsp, %rbp
subq $16, %rsp
movl $20, -8(%rbp)
movl $30, -4(%rbp)
movl -8(%rbp), %eax
movl %eax, %edi
call f
movl %eax, -4(%rbp)
movl -4(%rbp), %eax
Leave
ret
```

```
int f(int x) {
    int y;
    y = x + 3;
    return y;
}
```

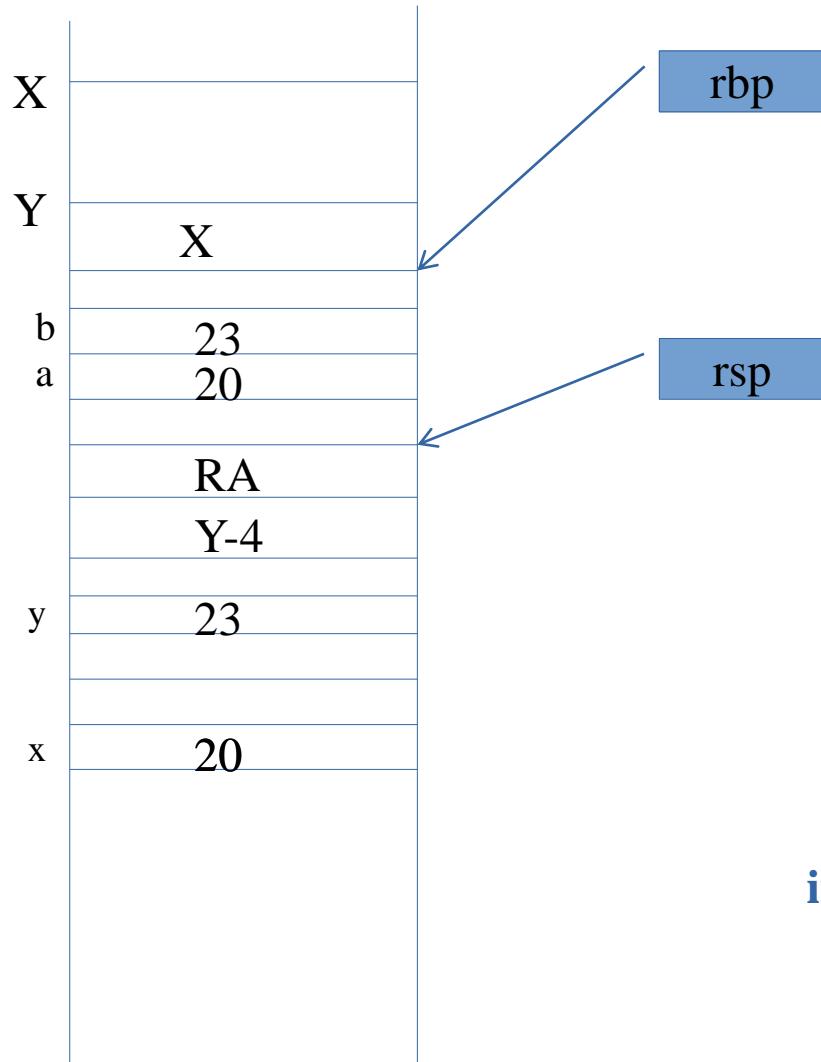
```
f:
endbr64
pushq %rbp
movq %rsp, %rbp
movl %edi, -20(%rbp)
movl -20(%rbp), %eax
addl $3, %eax
movl %eax, -4(%rbp)
movl -4(%rbp), %eax
RApopq %rbp
ret
```

edi = 20

eax = 23

```
int main() {
    int a = 20, b = 30;
    b = f(a);
    return b;
}
```

eip = RA



```
main:  
endbr64  
pushq %rbp  
movq %rsp, %rbp  
subq $16, %rsp  
movl $20, -8(%rbp)  
movl $30, -4(%rbp)  
movl -8(%rbp), %eax  
movl %eax, %edi  
call f  
movl %eax, -4(%rbp)  
movl -4(%rbp), %eax  
leave  
ret
```

```
int f(int x) {  
    int y;  
    y = x + 3;  
    return y;  
}
```

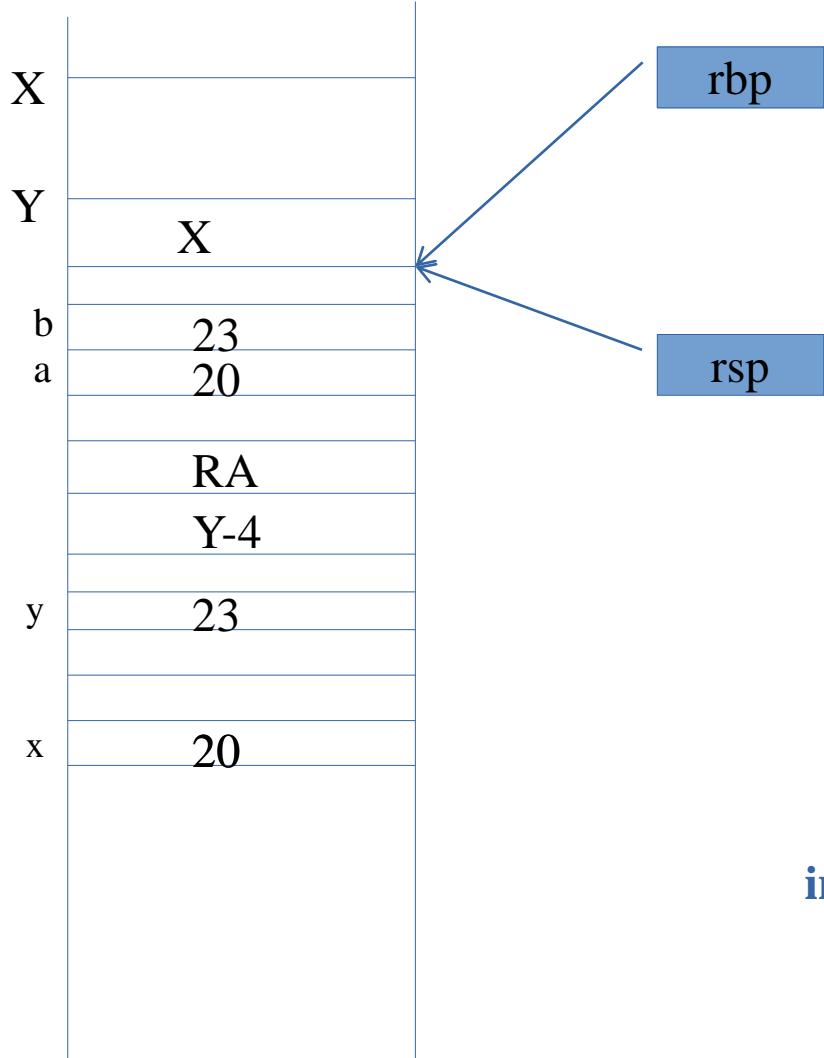
```
f:  
endbr64  
pushq %rbp  
movq %rsp, %rbp  
movl %edi, -20(%rbp)  
movl -20(%rbp), %eax  
addl $3, %eax  
movl %eax, -4(%rbp)  
movl -4(%rbp), %eax  
RApopq %rbp  
ret
```

edi = 20

eax = 23

```
int main() {  
    int a = 20, b = 30;  
    b = f(a);  
    return b;  
}
```

eip = RA



main:

endbr64

```

pushq %rbp
movq %rsp, %rbp
subq $16, %rsp
movl $20, -8(%rbp)
movl $30, -4(%rbp)
movl -8(%rbp), %eax
movl %eax, %edi
call f
movl %eax, -4(%rbp)
movl -4(%rbp), %eax
leave
# mov rbp rsp; pop rbp
ret

```

```

int f(int x) {
    int y;
    y = x + 3;
    return y;
}

```

int main() {

```

int a = 20, b = 30;
b = f(a);
return b;
}

```

f:

```

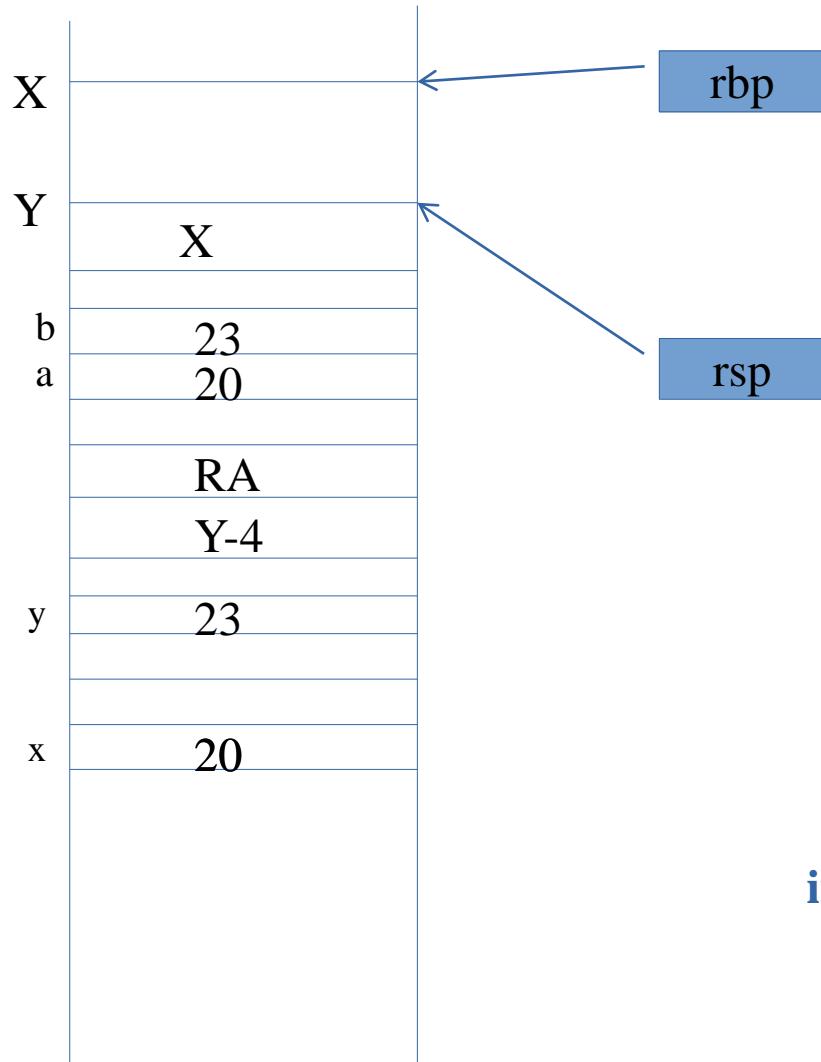
endbr64
pushq %rbp
movq %rsp, %rbp
movl %edi, -20(%rbp)
movl -20(%rbp), %eax
addl $3, %eax
movl %eax, -4(%rbp)
movl -4(%rbp), %eax
popq %rbp
ret

```

edi = 20

eax = 23

eip = RA



```

main:
endbr64
pushq %rbp
movq %rsp, %rbp
subq $16, %rsp
movl $20, -8(%rbp)
movl $30, -4(%rbp)
movl -8(%rbp), %eax
movl %eax, %edi
call f
movl %eax, -4(%rbp)
movl -4(%rbp), %eax
leave
# mov ebp esp; pop ebp
ret

```

```

int f(int x) {
    int y;
    y = x + 3;
    return y;
}

```

```

f:
endbr64
pushq %rbp
movq %rsp, %rbp
movl %edi, -20(%rbp)
movl -20(%rbp), %eax
addl $3, %eax
movl %eax, -4(%rbp)
movl -4(%rbp), %eax
popq %rbp
ret

```

edi = 20

eax = 23

```

int main() {
    int a = 20, b = 30;
    b = f(a);
    return b;
}

```

eip = RA

Further on calling convention

This was a simple program

The parameter was passed in a register!

What if there were many parameters?

CPUs have different numbers of registers.

More parameters, more functions demand a more sophisticated convention

May be slightly different on different processors, or 32-bit, 64-bit variants also.

Caller save and Callee save registers

Local variables

Are visible only within the function

Recursion: different copies of variables

Stored on “stack”

Registers

Are only one copy

Are within the CPU

Local Variables & Registers conflict

Caller save and Callee save registers

Caller Save registers

Which registers need to be saved by caller function . They can be used by the callee function!

The caller function will push them (if already in use, otherwise no need) on the stack

Callee save registers

Will be pushed on to the stack by called (callee) function

How to return values?

On the stack itself – then caller will have to pop

X86 convention – caller, callee saved 32 bit

The caller-saved registers are EAX, ECX, EDX.

The callee-saved registers are EBX, EDI, and ESI

Activation record looks like this

F() called g()

**Parameters-i refers to
parameters passed by f() to
g()**

**Local variable is a variable
in g()**

**Return address is the
location in f() where call
should go back**

X86 caller and callee rules(32 bit)

Caller rules on call

Push caller saved registers on stack

Push parameters on the stack – in reverse order. Why?

Substract esp, copy data

call f() // push + jmp

Caller rules on return

return value is in eax

remove parameters from stack : Add to esp.

Restore caller saved registers (if any)

X86 caller and callee rules

Callee rules on call

1) **push ebp**

mov ebp, esp

ebp(+/-offset) normally used to locate local vars and parameters on stack

ebp holds a copy of esp

Ebp is pushed so that it can be later popped while returnig

<https://www.cs.virginia.edu/~evans/cs216/guides/x86.html>

2) **Allocate local variables**

X86 caller and callee rules

Callee rules on return

- 1) Leave return value in eax**
- 2) Restore callee saved registers**
- 3) Deallocate local variables**
- 4) restore the ebp**
- 5) return**

32 bit vs 64 bit calling convention

Registers are used for passing parameters in 64 bit , to a large extent

Upto 6 parameters

More parameters pushed on stack

See

<https://aaronbloomfield.github.io/pdr/book/x86-64bit-ccc-chapter.pdf>

Beware

**When you read assembly code generated using
gcc -S**

You will find

More complex instructions

But they will essentially follow the convention mentioned

Comparison

	MIPS	x86
Arguments:	First 4 in %a0-%a3, remainder on stack	Generally all on stack
Return values:	%v0-%v1	%eax
Caller-saved registers:	%t0-%t9	%eax, %ecx, & %edx
Callee-saved registers:	%s0-%s9	Usually none

Figure 6.2: A comparison of the calling conventions of MIPS and x86

From the textbook by Misru

simple3.c and simple3.s

```
int f(int x1, int x2, int x3, int x4, int x5, int x6, int x7, int x8, int x9, int x10) {  
    int h;  
    h = x1 + x2 + x3 + x4 + x5 + x6 + x7 + x8 + x9 + x10 + 3;  
    return h;  
}  
int main() {  
    int a1 = 10, a2 = 20, a3 = 30, a4 = 40, a5 = 50, a6 = 60, a7 = 70, a8 = 80, a9 = 90, a10 = 100;  
    int b;  
    b = f(a1, a2, a3, a4, a5, a6, a7, a8, a9, a10);  
    return b;  
}
```

simple3.c and simple3.s

main:

endbr64	movl -24(%rbp), %r9d	
pushq %rbp	movl -28(%rbp), %r8d	movl %eax, %edi
movq %rsp, %rbp	movl -32(%rbp), %ecx	call f
subq \$48, %rsp	movl -36(%rbp), %edx	addq \$32, %rsp
movl \$10, -44(%rbp)	movl -40(%rbp), %esi	movl %eax, -4(%rbp)
movl \$20, -40(%rbp)	movl -44(%rbp), %eax	movl -4(%rbp), %eax
movl \$30, -36(%rbp)	movl -8(%rbp), %edi	leave
movl \$40, -32(%rbp)	pushq %rdi	ret
movl \$50, -28(%rbp)	movl -12(%rbp), %edi	
movl \$60, -24(%rbp)	pushq %rdi	
movl \$70, -20(%rbp)	movl -16(%rbp), %edi	
movl \$80, -16(%rbp)	pushq %rdi	
movl \$90, -12(%rbp)	movl -20(%rbp), %edi	
movl \$100, -8(%rbp)	pushq %rdi	

simple3.c and simple3.s

f:

endbr64

```
pushq %rbp
movq %rsp, %rbp
movl %edi, -20(%rbp)
movl %esi, -24(%rbp)
movl %edx, -28(%rbp)
movl %ecx, -32(%rbp)
movl %r8d, -36(%rbp)
movl %r9d, -40(%rbp)
movl -20(%rbp), %edx
movl -24(%rbp), %eax
addl %eax, %edx
movl -28(%rbp), %eax
addl %eax, %edx
movl -32(%rbp), %eax
```

```
addl %eax, %edx
movl -36(%rbp), %eax
addl %eax, %edx
movl -40(%rbp), %eax
addl %eax, %edx
movl 16(%rbp), %eax
addl %eax, %edx
movl 24(%rbp), %eax
addl %eax, %edx
movl 32(%rbp), %eax
addl %eax, %edx
```

```
movl 40(%rbp), %eax
addl %edx, %eax
addl $3, %eax
movl %eax, -4(%rbp)
movl -4(%rbp), %eax
popq %rbp
ret
```

Let's see a demo of how the stack is built and destroyed during function calls, on a Linux machine using GCC.

Consider this C code

```
int mult(int a, int b) {  
    int c, d = 20, e = 30, f;  
    f = add(d, e);  
    c = a * b + f;  
    return c;  
}  
  
int add(int x, int y) {  
    int z;  
    z = x + y;  
    return z;  
}
```

Translated to assembly as:

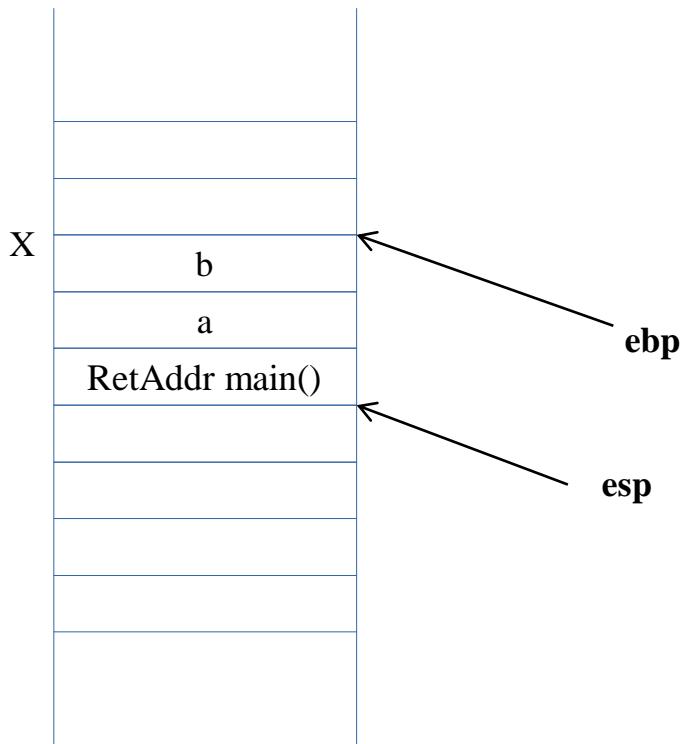
add:

```
pushl %ebp  
movl %esp, %ebp  
subl $16, %esp  
movl 8(%ebp), %edx  
movl 12(%ebp), %eax  
addl %edx, %eax  
movl %eax, -4(%ebp)  
movl -4(%ebp), %eax  
leave  
ret
```

mult:

```
pushl %ebp  
movl %esp, %ebp  
subl $24, %esp  
movl $20, -24(%ebp)  
movl $30, -20(%ebp)  
subl $8, %esp  
pushl -20(%ebp)  
pushl -24(%ebp)  
call add  
addl $16, %esp  
movl %eax, -16(%ebp)  
movl 8(%ebp), %eax  
imull 12(%ebp), %eax  
movl %eax, %edx  
movl -16(%ebp), %eax  
addl %edx, %eax  
movl %eax, -12(%ebp)  
movl -12(%ebp), %eax  
leave  
ret
```

Stack
↓



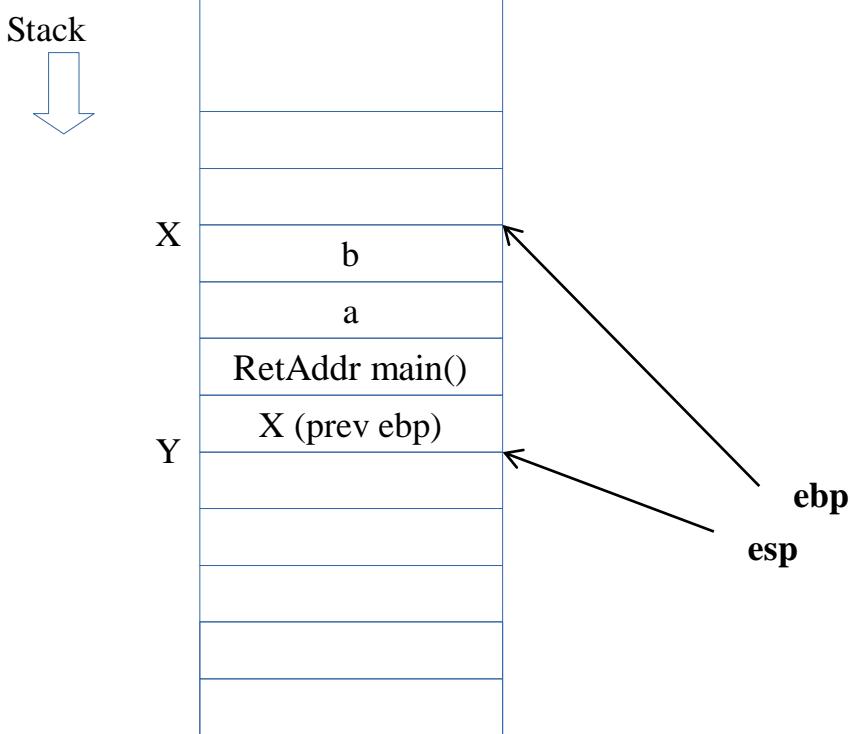
/ Control is here */*

```
int mult(int a, int b) {  
    int c, d = 20, e = 30, f;  
    f = add(d, e);  
    c = a * b + f;  
    return c;  
}  
int add(int x, int y) {  
    int z;  
    z = x + y;  
    return z;  
}
```

** Control is here */*

mult:

```
pushl %ebp  
movl %esp, %ebp  
subl $24, %esp  
movl $20, -24(%ebp)  
movl $30, -20(%ebp)  
subl $8, %esp  
pushl -20(%ebp)  
pushl -24(%ebp)  
call add
```



```

int mult(int a, int b) {
    int c, d = 20, e = 30, f;
    f = add(d, e);
    c = a * b + f;
    return c;
}
int add(int x, int y) {
    int z;
    z = x + y;
    return z;
}

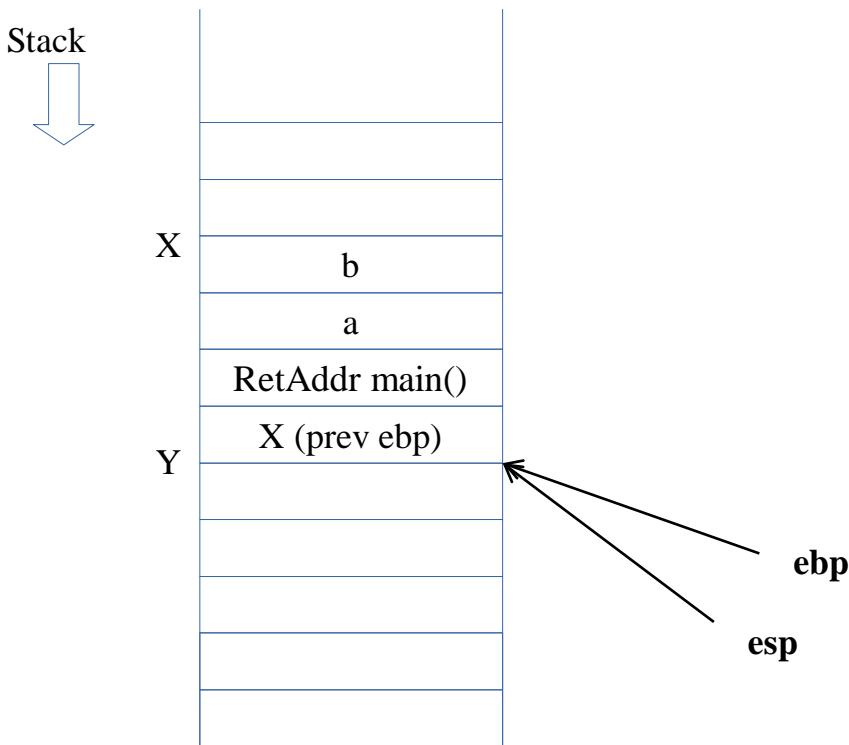
```

mult:

```

pushl %ebp
movl %esp, %ebp
subl $24, %esp
movl $20, -24(%ebp)
movl $30, -20(%ebp)
subl $8, %esp
pushl -20(%ebp)
pushl -24(%ebp)
call add

```



```

int mult(int a, int b) {
    int c, d = 20, e = 30, f;
    f = add(d, e);
    c = a * b + f;
    return c;
}
int add(int x, int y) {
    int z;
    z = x + y;
    return z;
}

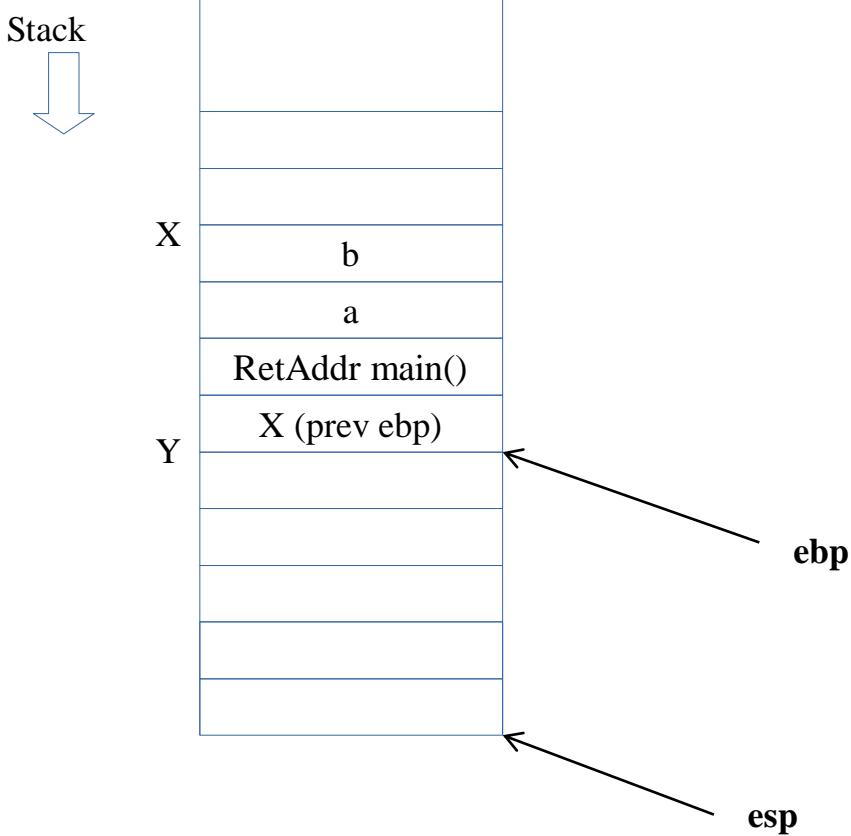
```

mult:

```

pushl %ebp
movl %esp, %ebp
subl $24, %esp
movl $20, -24(%ebp)
movl $30, -20(%ebp)
subl $8, %esp
pushl -20(%ebp)
pushl -24(%ebp)
call add

```



```

int mult(int a, int b) {
    int c, d = 20, e = 30, f;
    f = add(d, e);
    c = a * b + f;
    return c;
}
int add(int x, int y) {
    int z;
    z = x + y;
    return z;
}

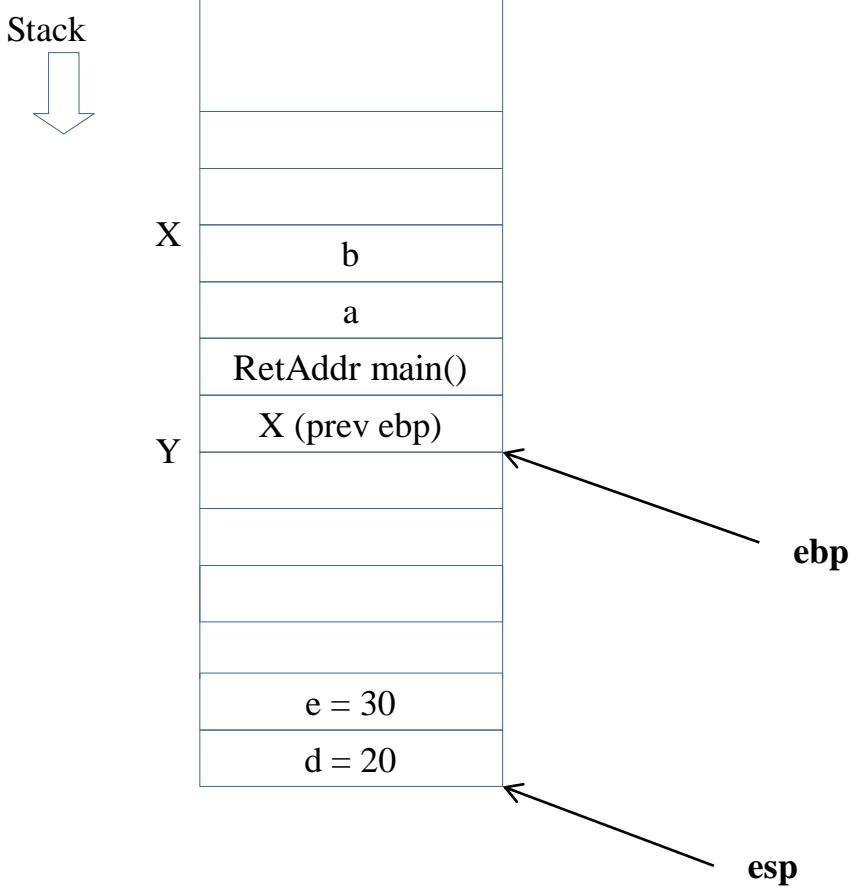
```

mult:

```

pushl %ebp
movl %esp, %ebp
subl $24, %esp
movl $20, -24(%ebp)
movl $30, -20(%ebp)
subl $8, %esp
pushl -20(%ebp)
pushl -24(%ebp)
call add

```



```

int mult(int a, int b) {
    int c, d = 20, e = 30, f;
    f = add(d, e);
    c = a * b + f;
    return c;
}
int add(int x, int y) {
    int z;
    z = x + y;
    return z;
}

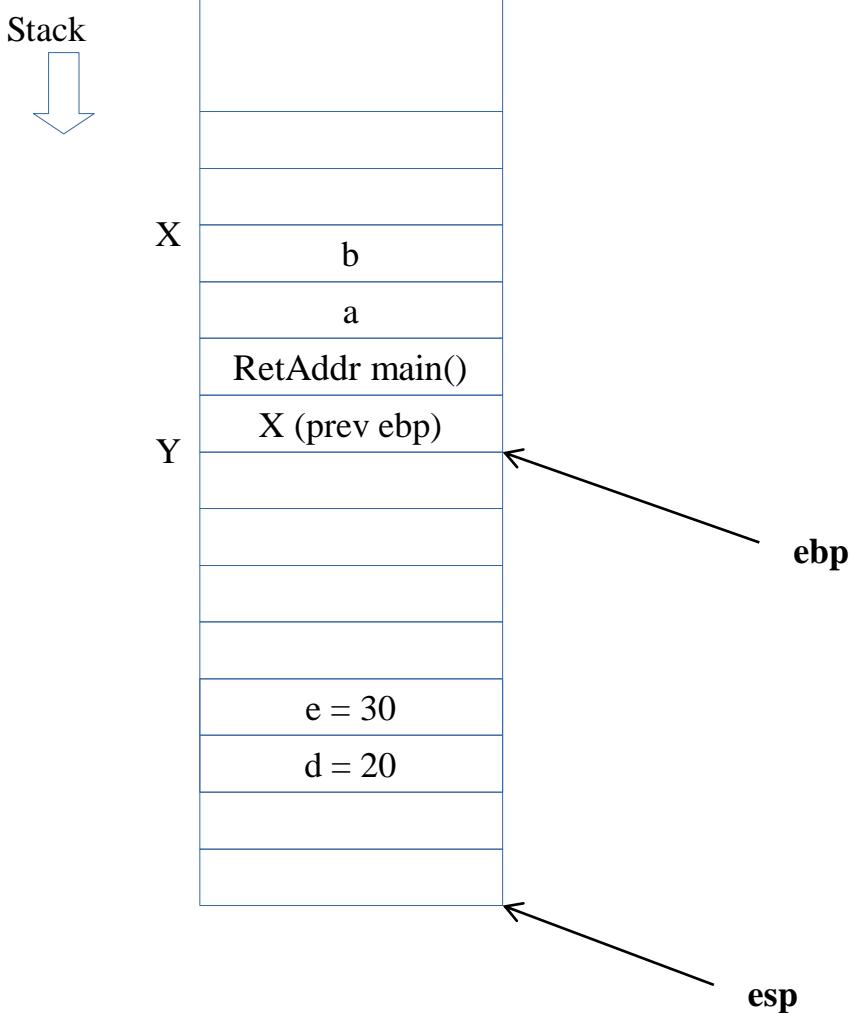
```

mult:

```

pushl %ebp
movl %esp, %ebp
subl $24, %esp
movl $20, -24(%ebp)
movl $30, -20(%ebp)
subl $8, %esp
pushl -20(%ebp)
pushl -24(%ebp)
call add

```



```

int mult(int a, int b) {
    int c, d = 20, e = 30, f;
    f = add(d, e);
    c = a * b + f;
    return c;
}
int add(int x, int y) {
    int z;
    z = x + y;
    return z;
}

```

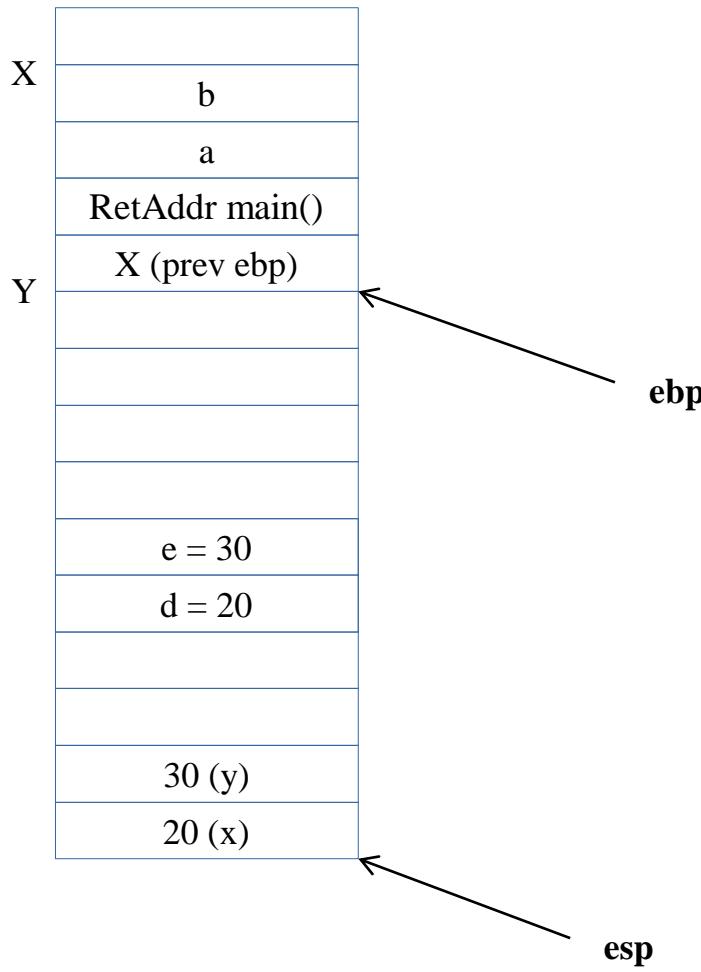
mult:

```

pushl %ebp
movl %esp, %ebp
subl $24, %esp
movl $20, -24(%ebp)
movl $30, -20(%ebp)
subl $8, %esp
pushl -20(%ebp)
pushl -24(%ebp)
call add

```

Stack

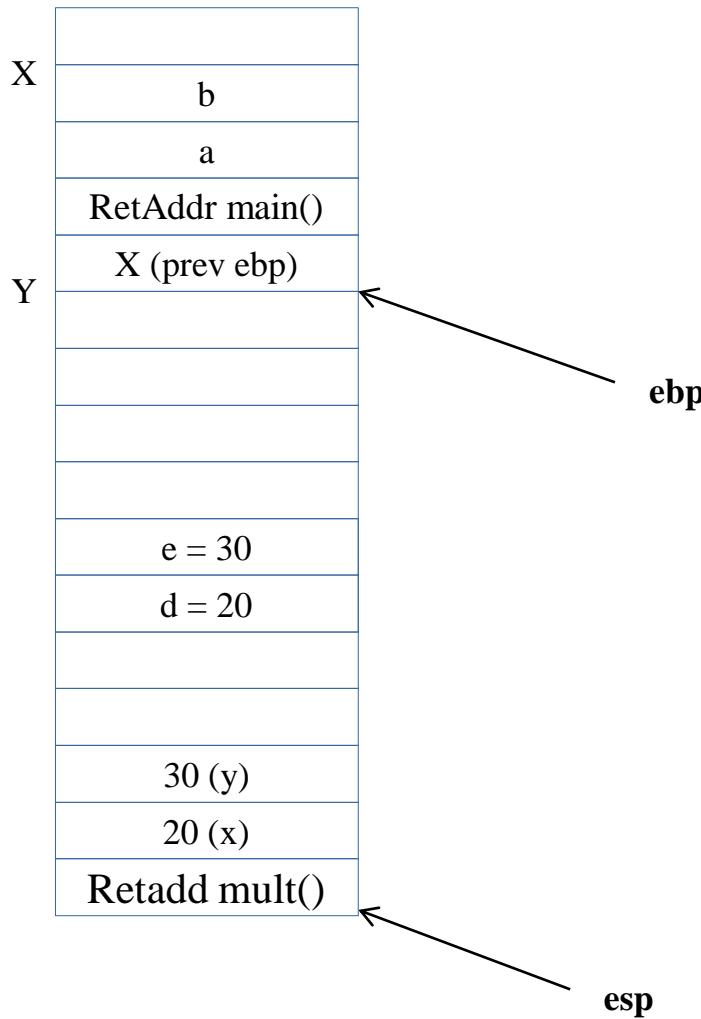


```
int mult(int a, int b) {  
    int c, d = 20, e = 30, f;  
    f = add(d, e);  
    c = a * b + f;  
    return c;  
}  
int add(int x, int y) {  
    int z;  
    z = x + y;  
    return z;  
}
```

mult:

```
pushl %ebp  
movl %esp, %ebp  
subl $24, %esp  
movl $20, -24(%ebp)  
movl $30, -20(%ebp)  
subl $8, %esp  
pushl -20(%ebp)  
pushl -24(%ebp)  
call add
```

Stack



```
int mult(int a, int b) {  
    int c, d = 20, e = 30, f;  
    f = add(d, e);  
    c = a * b + f;  
    return c;  
}  
int add(int x, int y) {  
    int z;  
    z = x + y;  
    return z;  
}
```

mult:

```
pushl %ebp  
movl %esp, %ebp  
subl $24, %esp  
movl $20, -24(%ebp)  
movl $30, -20(%ebp)  
subl $8, %esp  
pushl -20(%ebp)  
pushl -24(%ebp)  
call add
```

Stack



X	b
	a
	RetAddr main()
Y	X (prev ebp)
e = 30	
d = 20	
30 (y)	
20 (x)	
Retadd mult()	
Y(prev ebp)	

ebp

esp

```
int mult(int a, int b) {  
    int c, d = 20, e = 30, f;  
    f = add(d, e);  
    c = a * b + f;  
    return c;  
}
```

```
int add(int x, int y) {  
    int z;  
    z = x + y;  
    return z;  
}
```

add:

```
pushl %ebp  
movl %esp, %ebp  
subl $16, %esp  
movl 8(%ebp), %edx  
movl 12(%ebp), %eax  
addl %edx, %eax  
movl %eax, -4(%ebp)  
movl -4(%ebp), %eax  
leave  
ret
```

Stack



X	b
	a
	RetAddr main()
X (prev ebp)	
Y	
e = 30	
d = 20	
30 (y)	
20 (x)	
Retadd mult()	
Y(prev ebp)	

edx = 20
eax = 30
eax = eax + edx = 50

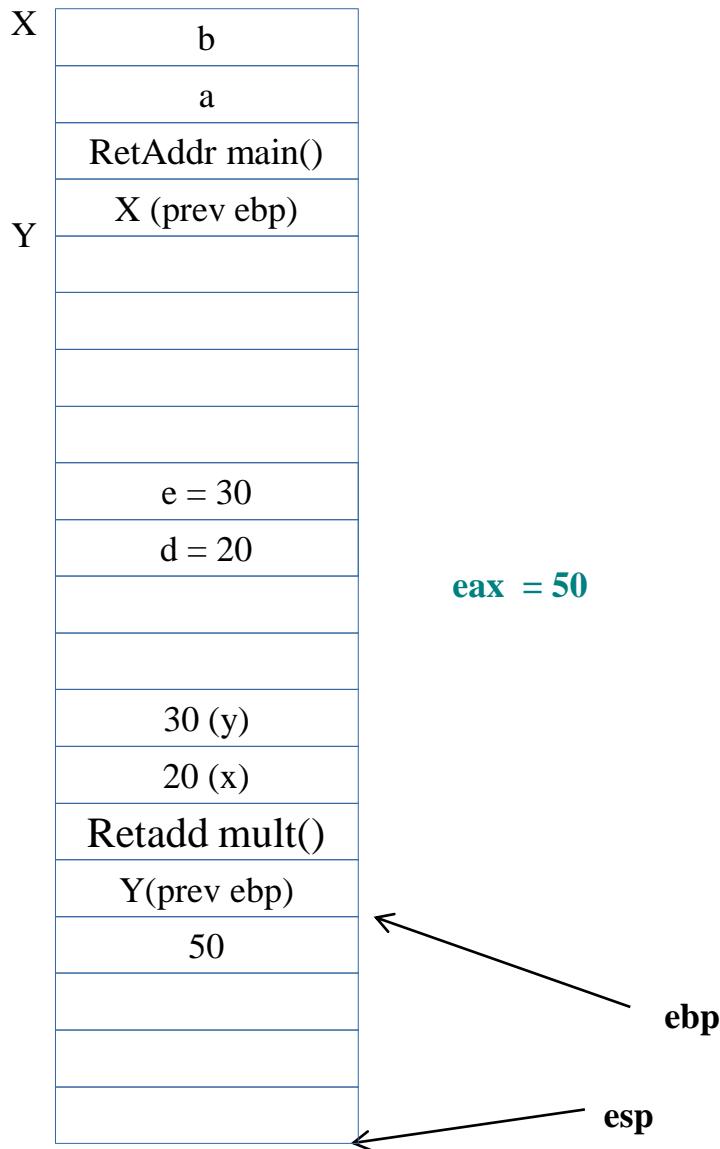
ebp
esp

```
int mult(int a, int b) {  
    int c, d = 20, e = 30, f;  
    f = add(d, e);  
    c = a * b + f;  
    return c;  
}  
int add(int x, int y) {  
    int z;  
    z = x + y;  
    return z;  
}
```

add:

```
pushl %ebp  
movl %esp, %ebp  
subl $16, %esp  
movl 8(%ebp), %edx  
movl 12(%ebp), %eax  
addl %edx, %eax  
movl %eax, -4(%ebp)  
movl -4(%ebp), %eax  
leave  
ret
```

Stack



```
int mult(int a, int b) {  
    int c, d = 20, e = 30, f;  
    f = add(d, e);  
    c = a * b + f;  
    return c;  
}
```

```
int add(int x, int y) {  
    int z;  
    z = x + y;  
    return z;  
}
```

add:

```
pushl %ebp  
movl %esp, %ebp  
subl $16, %esp  
movl 8(%ebp), %edx  
movl 12(%ebp), %eax  
addl %edx, %eax  
movl %eax, -4(%ebp)  
movl -4(%ebp), %eax  
leave  
ret
```

Some redundant code generated here.
Before “leave”. Result is in eax

Stack



X	b
	a
	RetAddr main()
Y	X (prev ebp)
e = 30	
d = 20	
30 (y)	
20 (x)	
Retadd mult()	
Y(prev ebp)	
50	

leave: step 1

eax = 50

esp

ebp

```
int mult(int a, int b) {  
    int c, d = 20, e = 30, f;  
    f = add(d, e);  
    c = a * b + f;  
    return c;  
}
```

```
int add(int x, int y) {  
    int z;  
    z = x + y;  
    return z;  
}
```

add:

```
pushl %ebp  
movl %esp, %ebp  
subl $16, %esp  
movl 8(%ebp), %edx  
movl 12(%ebp), %eax  
addl %edx, %eax  
movl %eax, -4(%ebp)  
movl -4(%ebp), %eax
```

**leave # # Set ESP to EBP, then pop
ret**

Stack



X	b
	a
	RetAddr main()
Y	X (prev ebp)
e = 30	
d = 20	
30 (y)	
20 (x)	
Retadd mult()	
Y(prev ebp)	
50	

leave: step 2

eax = 50

esp

ebp

```
int mult(int a, int b) {  
    int c, d = 20, e = 30, f;  
    f = add(d, e);  
    c = a * b + f;  
    return c;  
}
```

```
int add(int x, int y) {  
    int z;  
    z = x + y;  
    return z;  
}
```

add:

```
pushl %ebp  
movl %esp, %ebp  
subl $16, %esp  
movl 8(%ebp), %edx  
movl 12(%ebp), %eax  
addl %edx, %eax  
movl %eax, -4(%ebp)  
movl -4(%ebp), %eax
```

**leave # # Set ESP to EBP, then pop
ret**

Stack



X	b
	a
	RetAddr main()
Y	X (prev ebp)
	e = 30
	d = 20
	30 (y)
	20 (x)
	Retaddr mult()
	Y(prev ebp)
	50

eax = 50

esp

ebp

```
int mult(int a, int b) {  
    int c, d = 20, e = 30, f;  
    f = add(d, e); // here  
    c = a * b + f;  
    return c;  
}
```

```
int add(int x, int y) {  
    int z;  
    z = x + y;  
    return z;  
}
```

add:

```
pushl %ebp  
movl %esp, %ebp  
subl $16, %esp  
movl 8(%ebp), %edx  
movl 12(%ebp), %eax  
addl %edx, %eax  
movl %eax, -4(%ebp)  
movl -4(%ebp), %eax
```

leave # # Set ESP to EBP, then pop
ret

Stack



X	b
	a
	RetAddr main()
Y	X (prev ebp)
e = 30	
d = 20	
30 (y)	
20 (x)	
Retadd mult()	
Y(prev ebp)	
50	

eax = 50

ebp

esp

```
int mult(int a, int b) {  
    int c, d = 20, e = 30, f;  
    f = add(d, e); // here  
    c = a * b + f;  
    return c;  
}  
int add(int x, int y) {  
    int z;  
    z = x + y;  
    return z;  
}
```

Mult:

....

```
call add  
addl $16, %esp  
movl %eax, -16(%ebp)  
movl 8(%ebp), %eax  
imull 12(%ebp), %eax  
movl %eax, %edx  
movl -16(%ebp), %eax  
addl %edx, %eax  
movl %eax, -12(%ebp)  
movl -12(%ebp), %eax  
leave  
ret
```

Stack



X	b
	a
	RetAddr main()
Y	X (prev ebp)
	f = 50 (eax)
	e = 30
	d = 20
	30 (y)
	20 (x)
	Retadd mult()
	Y(prev ebp)
	50

eax = 50

esp

ebp

```
int mult(int a, int b) {  
    int c, d = 20, e = 30, f;  
    f = add(d, e);  
    c = a * b + f;  
    return c;  
}
```

```
int add(int x, int y) {  
    int z;  
    z = x + y;  
    return z;  
}
```

Mult:

```
....  
    call    add  
    addl   $16, %esp  
movl  %eax, -16(%ebp)  
    movl   8(%ebp), %eax  
    imull  12(%ebp), %eax  
    movl   %eax, %edx  
    movl   -16(%ebp), %eax  
    addl   %edx, %eax  
    movl   %eax, -12(%ebp)  
    movl   -12(%ebp), %eax  
    leave  
    ret
```

Stack



X	b
	a
	RetAddr main()
Y	X (prev ebp)
	f = 50
	e = 30
	d = 20
	30 (y)
	20 (x)
	Retadd mult()
	Y(prev ebp)
	50

eax = a
eax = eax * b
edx = eax
eax = f
eax = edx + eax
*// eax = a*b + f*

ebp

esp

```
int mult(int a, int b) {  
    int c, d = 20, e = 30, f;  
    f = add(d, e);  
    c = a * b + f;  
    return c;  
}  
int add(int x, int y) {  
    int z;  
    z = x + y;  
    return z;  
}
```

Mult:

....

call add
addl \$16, %esp
movl %eax, -16(%ebp)
movl 8(%ebp), %eax
imull 12(%ebp), %eax
movl %eax, %edx
movl -16(%ebp), %eax
addl %edx, %eax
movl %eax, -12(%ebp)
movl -12(%ebp), %eax
leave
ret

Stack



X	b
	a
	RetAddr main()
Y	X (prev ebp)
	c = eax
	f = 50
	e = 30
	d = 20
	30 (y)
	20 (x)
	Retadd mult()
	Y(prev ebp)
	50

// eax = a*b + f

ebp

esp

```
int mult(int a, int b) {  
    int c, d = 20, e = 30, f;  
    f = add(d, e);  
    c = a * b + f;  
    return c;  
}
```

```
int add(int x, int y) {  
    int z;  
    z = x + y;  
    return z;  
}
```

Mult:

....

```
call add  
addl $16, %esp  
movl %eax, -16(%ebp)  
movl 8(%ebp), %eax  
imull 12(%ebp), %eax  
movl %eax, %edx  
movl -16(%ebp), %eax  
addl %edx, %eax  
movl %eax, -12(%ebp)  
movl -12(%ebp), %eax  
leave  
ret
```

Again some redundant code

Stack



X	b
	a
	RetAddr main()
Y	X (prev ebp)
	c = eax
	f = 50
	e = 30
	d = 20
	30 (y)
	20 (x)
	Retadd mult()
	Y(prev ebp)
	50

After leave
// eax = a*b + f

ebp
esp

```
int mult(int a, int b) {  
    int c, d = 20, e = 30, f;  
    f = add(d, e);  
    c = a * b + f;  
    return c;  
}
```

```
int add(int x, int y) {  
    int z;  
    z = x + y;  
    return z;  
}
```

Mult:

....

```
call add  
addl $16, %esp  
movl %eax, -16(%ebp)  
movl 8(%ebp), %eax  
imull 12(%ebp), %eax  
movl %eax, %edx  
movl -16(%ebp), %eax  
addl %edx, %eax  
movl %eax, -12(%ebp)  
movl -12(%ebp), %eax  
leave  
ret
```

Lessons

- Calling function (caller)
 - Pushes arguments on stack , copies values
 - On call
 - Return IP is pushed
- Initially in called function (callee)
 - Old ebp is pushed
 - ebp = stack
 - Stack is decremented to make space for local variables

Lessons

- Before Return
- Ensure that result is in ‘eax’
- On Return
 - stack = ebp
 - Pop ebp (ebp = old ebp)
 - On ‘ret’
 - Pop ‘return IP’ and go back in old function

Lessons

- This was a demonstration for a
- User program, compiled with GCC, On Linux
- Followed the conventions we discussed earlier
- Applicable to
- C programs which work using LIFO function calls
- Compiler can't be used to generate code using this mechanism for
- Functions like fork(), exec(), scheduler(), etc.
- Boot code of OS

File Systems

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

What we are going to learn

- The operating system interface (system calls, commands/utilities) for accessing files in a file-system
- Design aspects of OS to implement the file system
 - On disk data structure
 - In memory kernel data structures

What is a file?

- A (dumb!) sequence of bytes (typically on a permanent storage:secondary, tertiary) , with
 - A name
 - Permissions
 - Owner
 - Timestamps,
 - Etc.
- Types: Text files, binary files (one

File types and kernel

- **For example, MP4 file**
 - vlc will do a open(...) on the file, and call read(...), **interpret** the contents of the file as movie and show movie
 - Kernel will simply provide open(...), read(...), write(...) to access file data
 - Meaning of the file contents is known to VLC and not to kernel!

file type	usual extension	function
executable	exe, com, bin or none	ready-to-run machine-language program
object	obj, o	compiled, machine language, not linked
source code	c, cc, java, pas, asm, a	source code in various languages
batch	bat, sh	commands to the command interpreter
text	txt, doc	textual data, documents
word processor	wp, tex, rtf, doc	various word-processor formats
library	lib, a, so, dll	libraries of routines for programmers
print or view	ps, pdf, jpg	ASCII or binary file in a format for printing or viewing
archive	arc, zip, tar	related files grouped into one file, sometimes compressed, for archiving or storage
multimedia	mpeg, mov, rm, mp3, avi	binary file containing audio or A/V information

What is a file?

- The sequence of bytes can be *interpreted* (*by an application*) to be
 - Just a sequence of bytes
 - E.g. a text file
 - Sequence of records/structures
 - E.g. a file of student records , by database application, etc
 - A complexly organized, collection of records and bytes

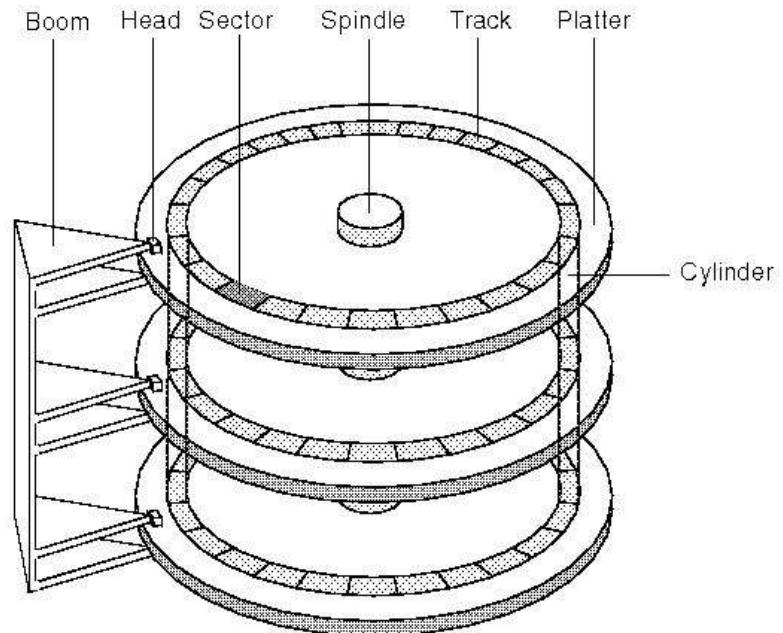
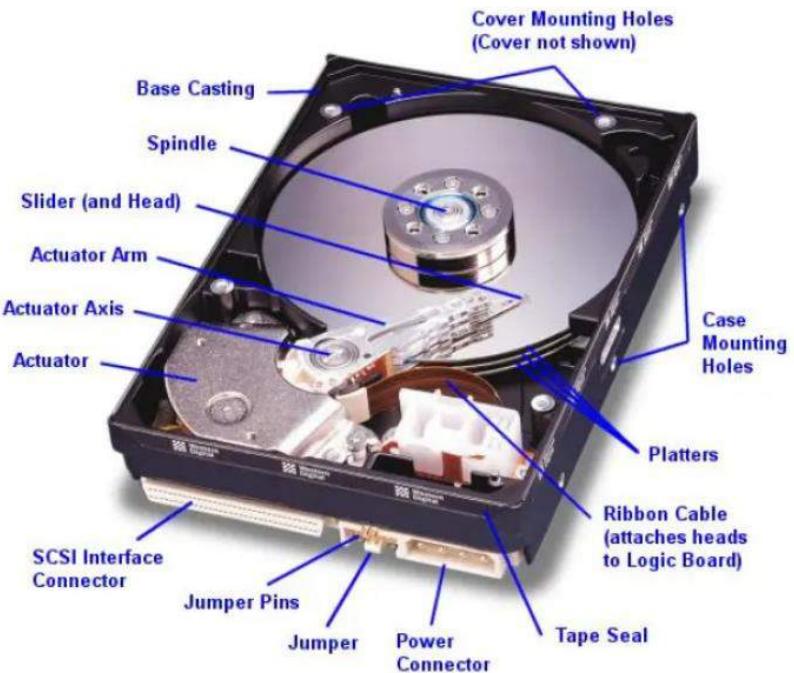
File attributes

- Run
 - \$ ls -l
- on Linux
- To see file listing with different attributes
- Different OSes and file-systems provide different sets of file attributes

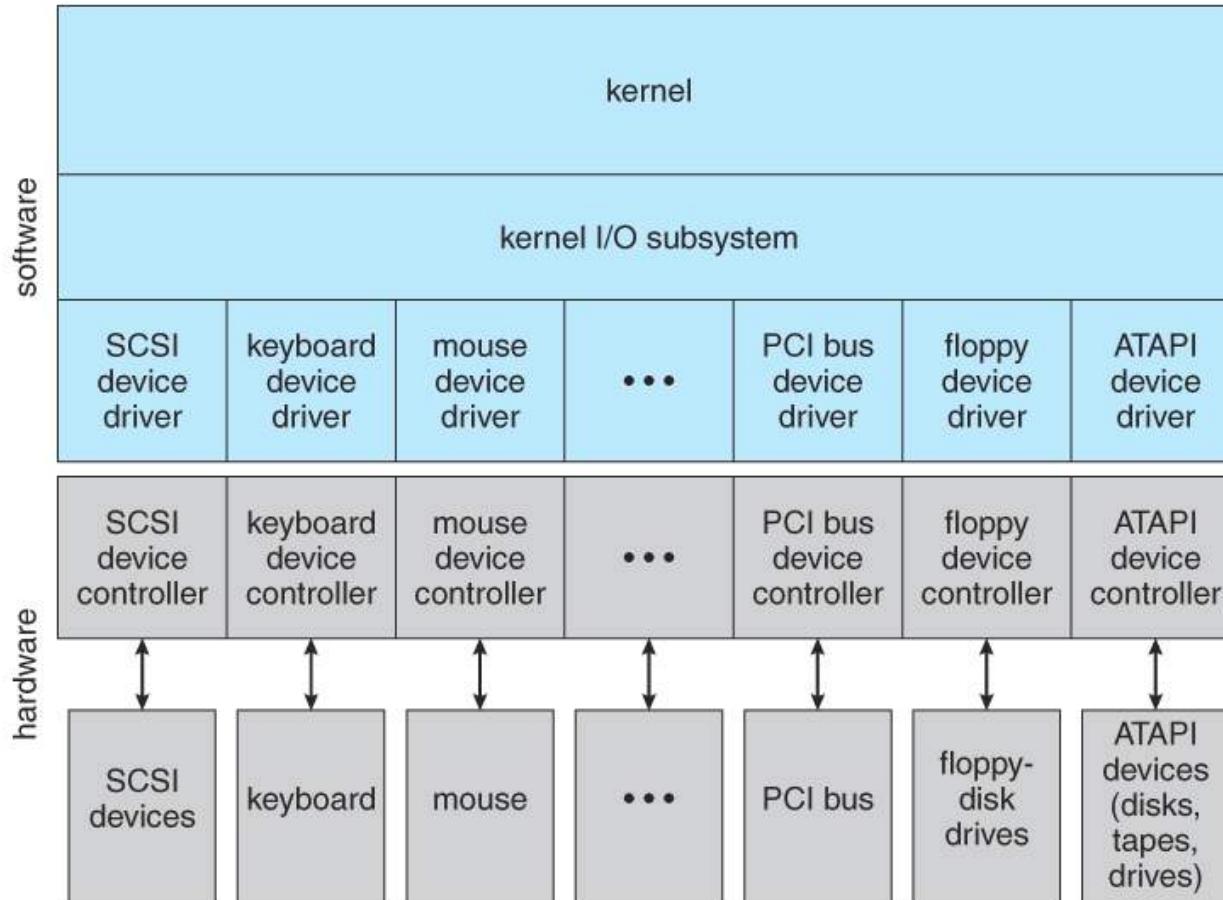
Access methods

- OS system calls may provide two types of access to files
 - Sequential Access
 - read next
 - write next
 - reset
 - no read after last write
 - Direct Access
 - read n
 - write n
 - position to n
 - read next
 - write next
 - rewrite n
 - n = relative block number

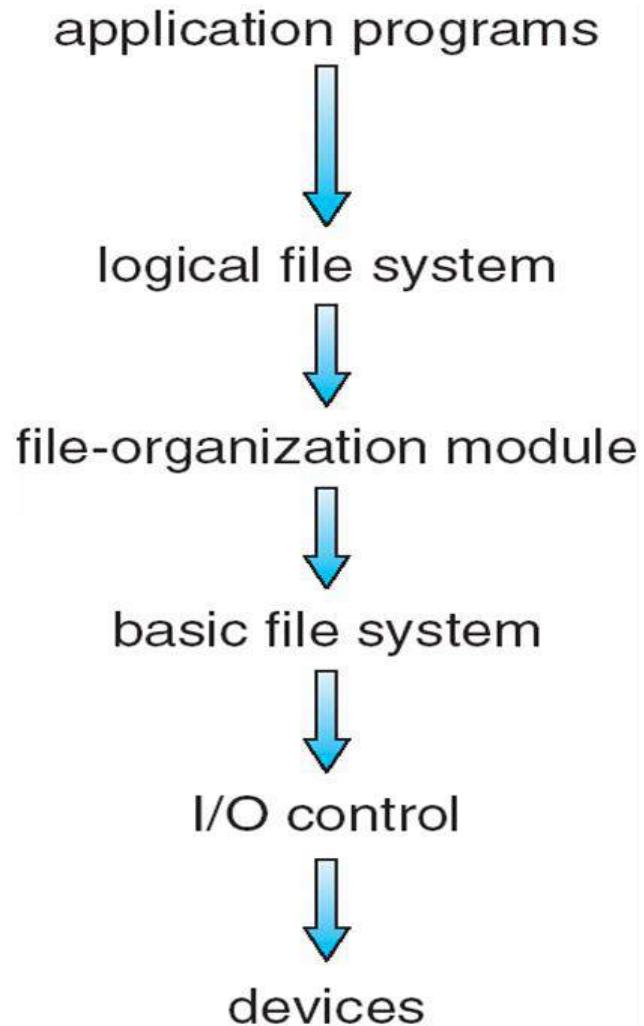
Disk



Device Driver



File system implementation: layering



Application programs

```
int main() {  
    char buf[128]; int count;  
    fd = open(...);  
    read(fd, buf, count);  
}
```

OS

Logical file system:

```
sys_read(int fd, char *buf, int count) {  
    file *fp = currproc->fdarray[fd];  
    file_read(fp, ...);  
}
```

File organization module:

```
file_read(file *fp, char *buf, int count) {  
    offset = fp->current-offset;  
    translate offset into blockno;  
    basic_read(blockno, buf, count);
```

Basic File system:

```
basic_read(int blockno, char *buf, ...) {  
    os_buffer *bp;  
    sectorno = calculation on blockno;  
    disk_driver_read(sectorno, bp );  
    move-process-to-wait-queue;  
    copy-data-to-user-buffer(bp, buf);  
}
```

IO Control, Device driver:

```
disk_driver_read(sectorno) {  
    issue instructions to disk controller  
    (often assembly code)  
    to read sectornointo specific location;  
}
```

XV6 does it slightly differently, but following the layering principle!

OS's job now

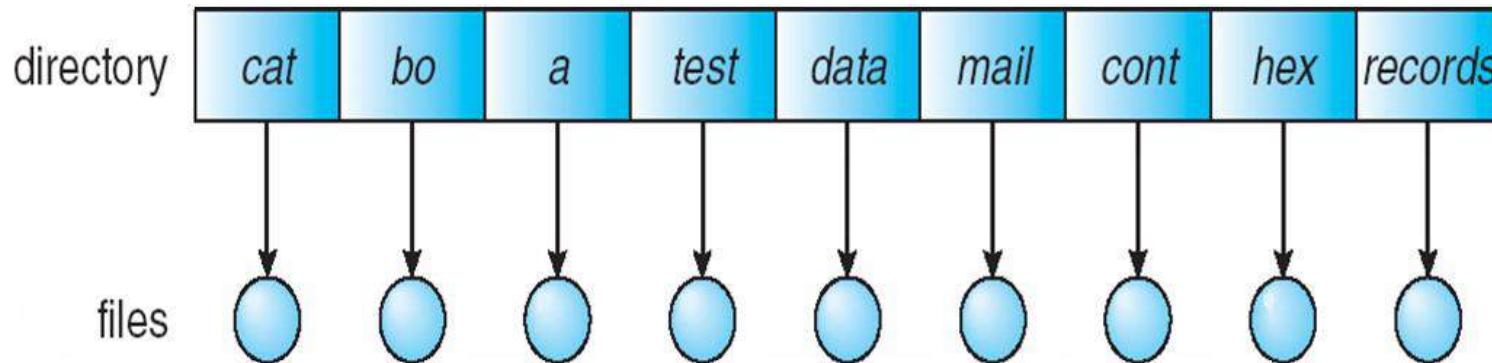
- To implement the logical view of file system as seen by end user
- Using the logical block-based view offered by the device driver

Formatting

- **Physical hard disk divided into partitions**
 - Partitions also known as minidisks, slices
- **A raw disk partition is accessible using device driver – but no block contains any data !**
 - Like an un-initialized array, or sectors/blocks
- **Formatting**
 - Creating an initialized data structure on the

Different types of “layouts”

Single level directory



Naming problem

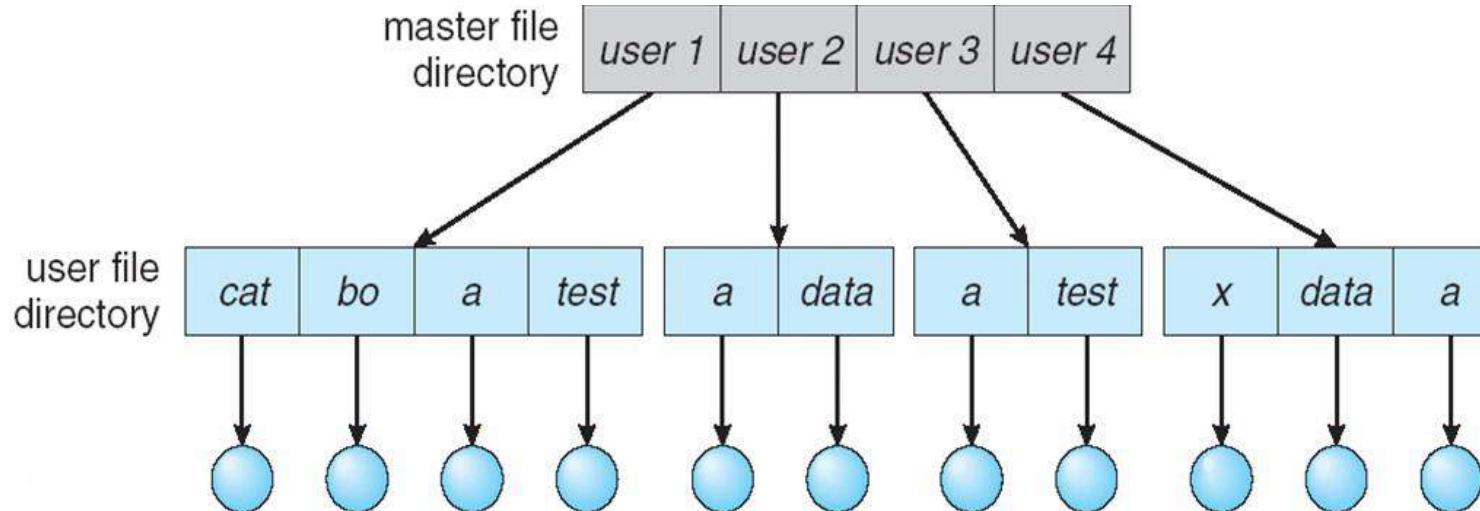
Grouping problem

Example: RT-11, from 1970s

<https://en.wikipedia.org/wiki/RT-11>

Different types of “layouts”

Two level directory



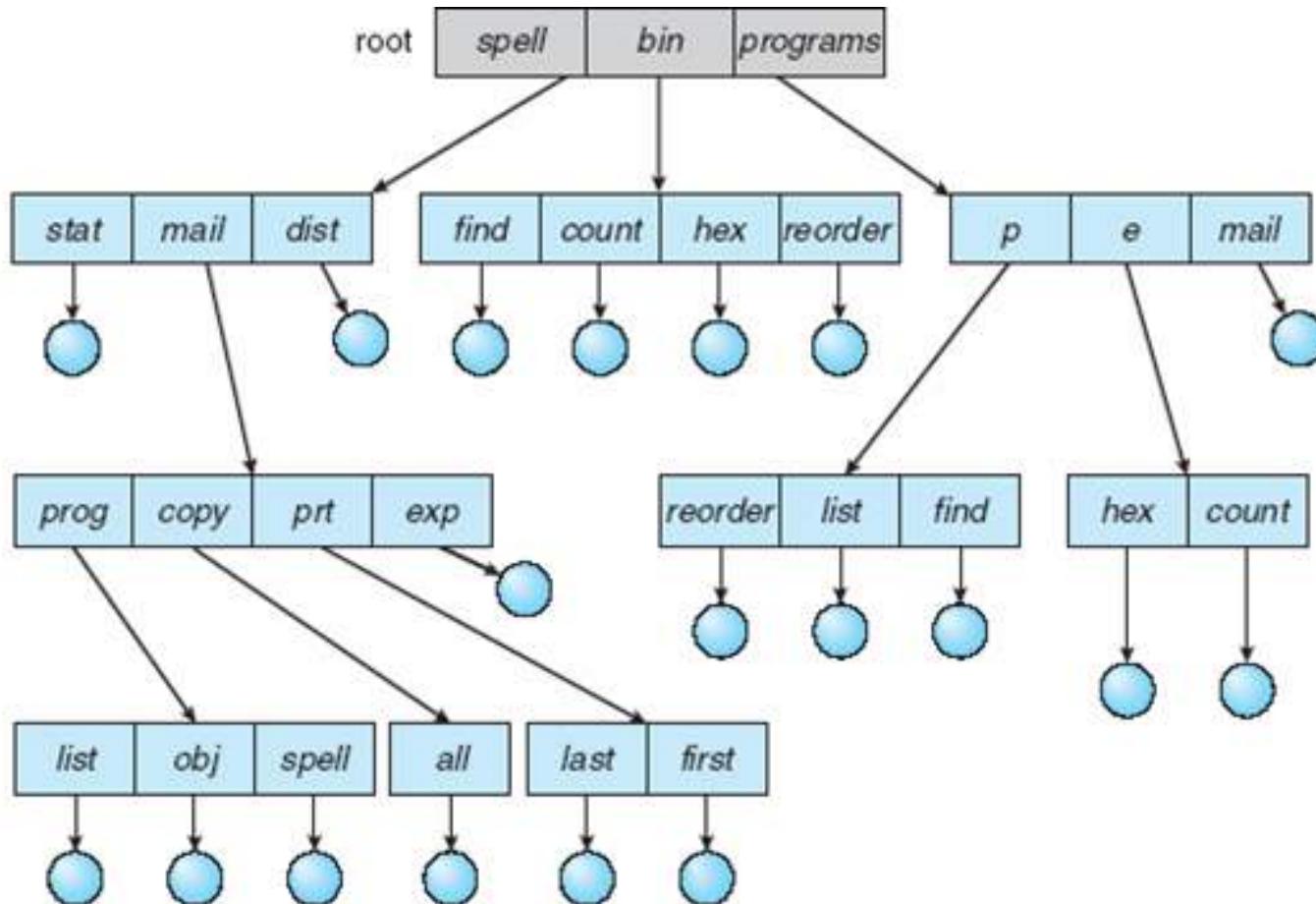
Path name

Can have the same file name for different user

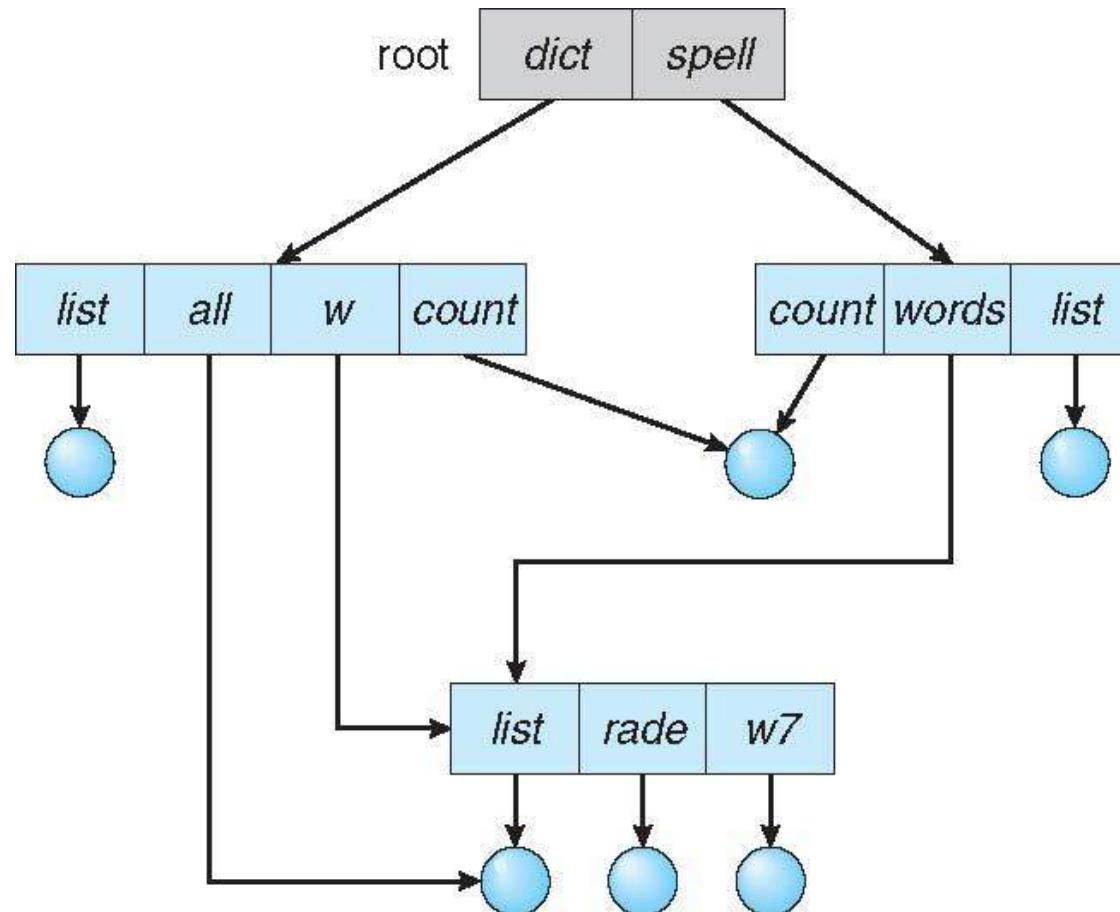
Efficient searching

No grouping capability

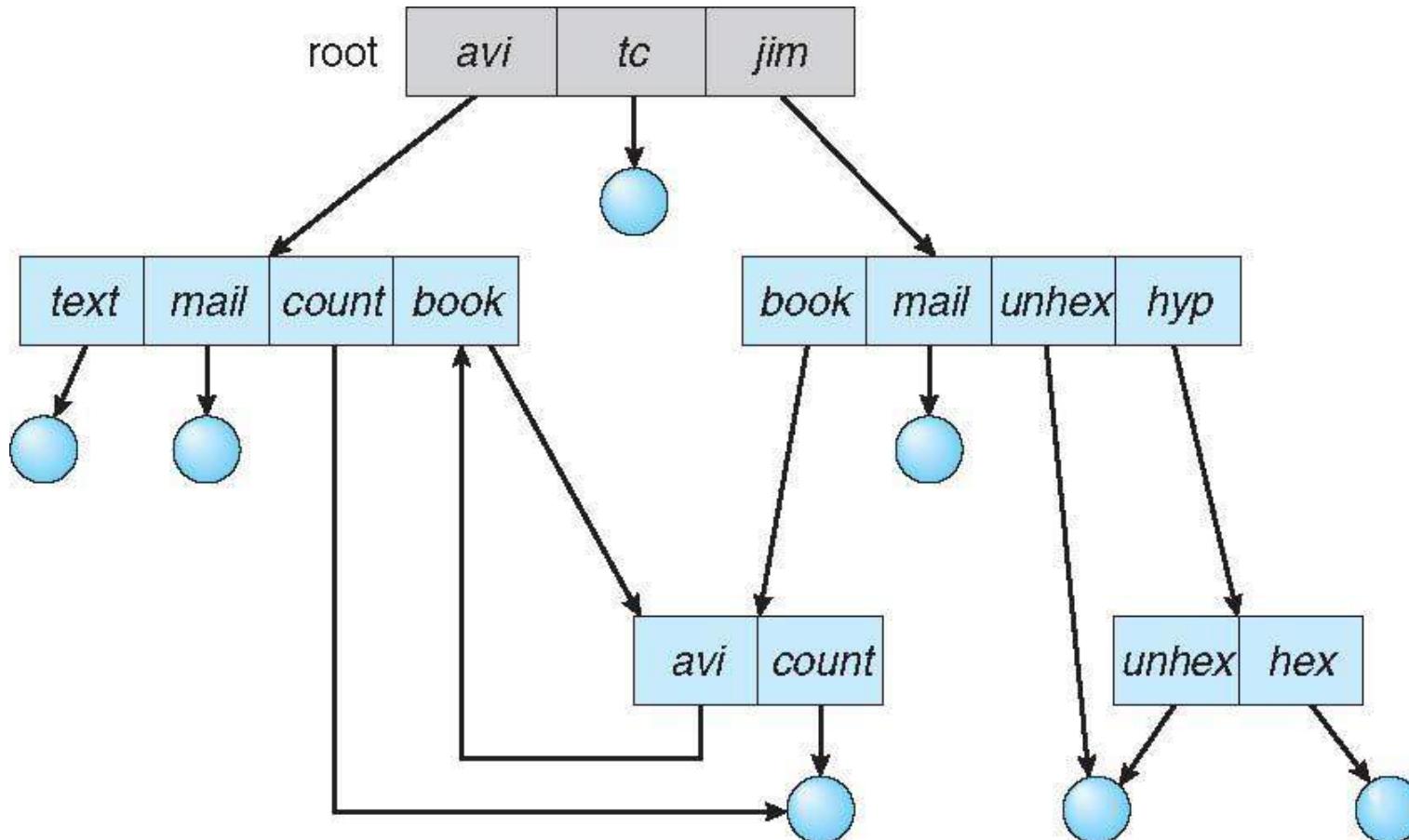
Tree Structured directories



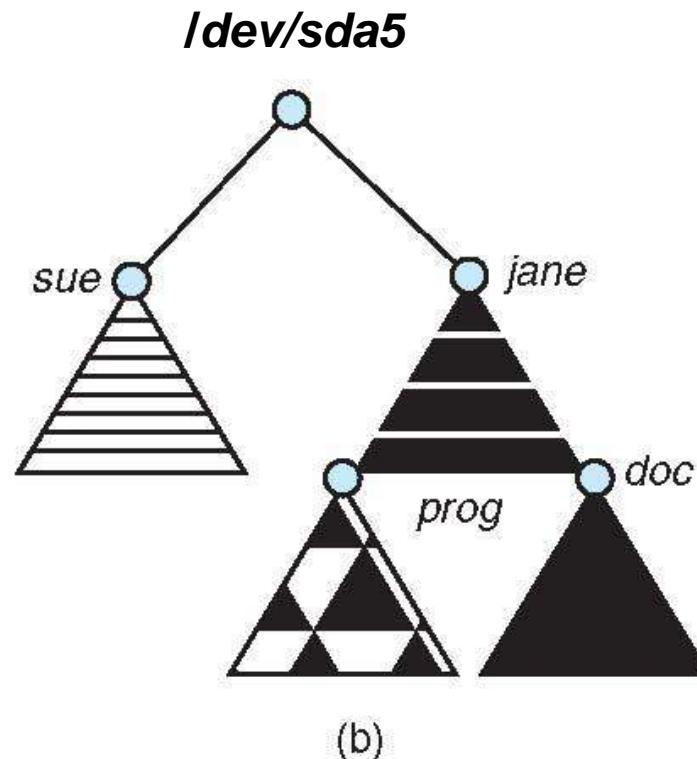
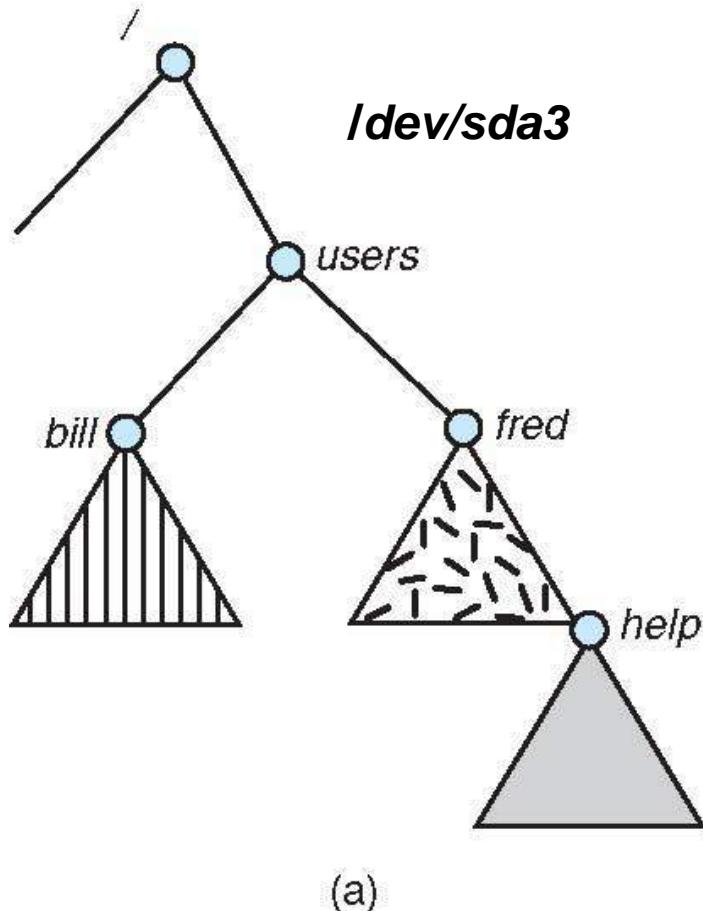
Acyclic Graph Directories



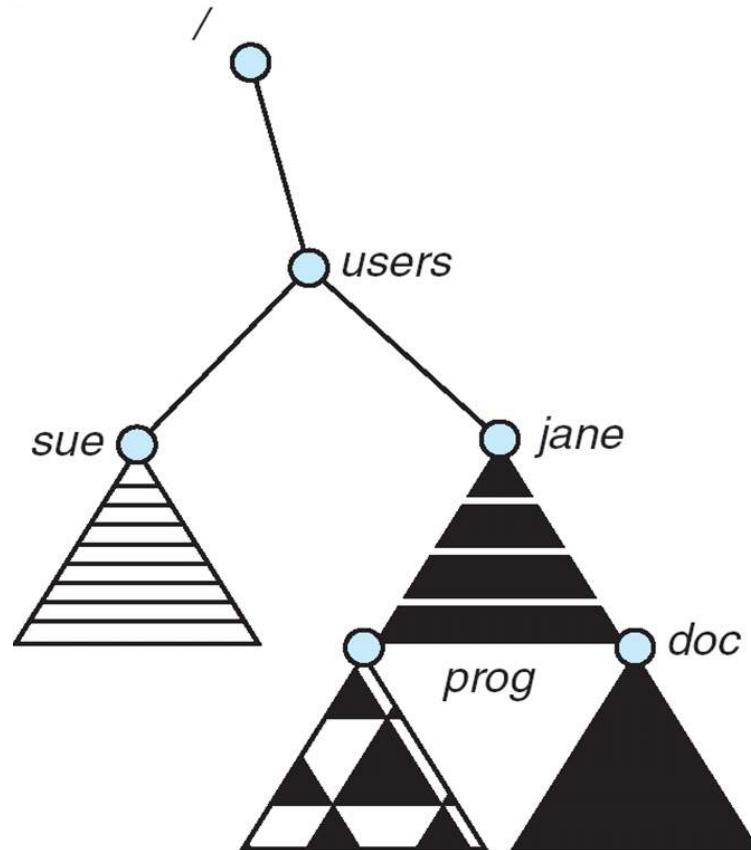
General Graph directory



Mounting of a file system: before



Mounting of a file system: after



```
$sudo mount /dev/sda5 /users
```

Remote mounting: NFS

- **Network file system**
- **\$ sudo mount 10.2.1.2:/x/y /a/b**
 - The /x/y partition on 10.2.1.2 will be made available under the folder /a/b on this computer

File sharing semantics

- **Consistency semantics specify how multiple users are to access a shared file simultaneously**
- **Unix file system (UFS) implements:**
 - Writes to an open file visible immediately to other users of the same open file
 - One mode of sharing file pointer to allow multiple users to read and write concurrently

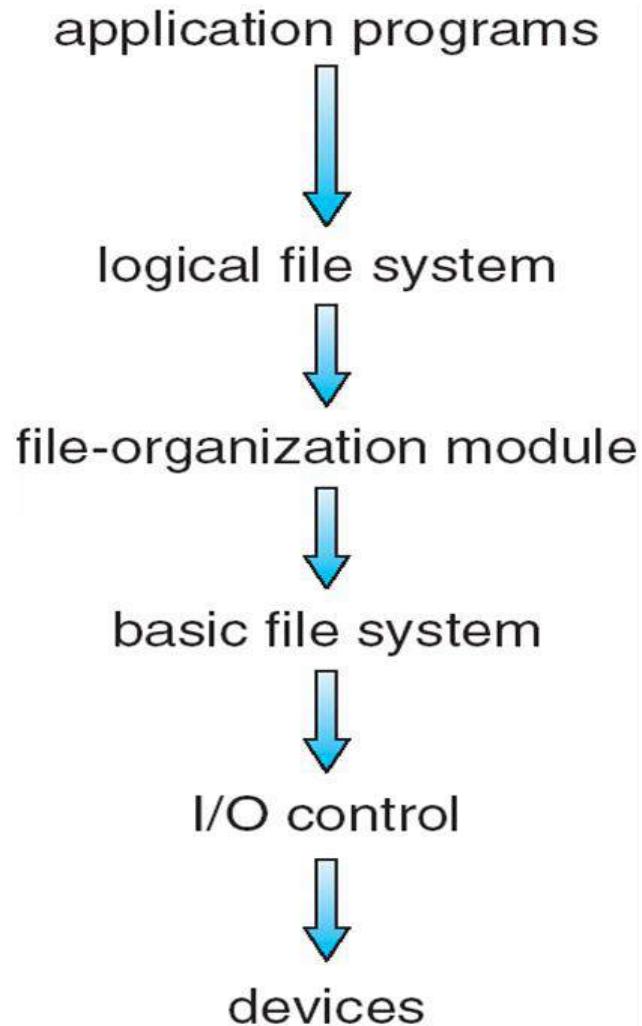
Implementing file systems

File system on disk

□ What we know

- Disk I/O in terms of sectors (512 bytes)
- File system: implementation of acyclic graph using the linear sequence of sectors
 - Store a acyclic graph into array of “blocks”/“sectors”
- Device driver available: gives sector/block wise access to the disk

File system implementation: layering



File system: Layering

- **Device drivers manage I/O devices at the I/O control layer**
 - Given commands like “read drive1, cylinder 72, track 2, sector 10, into memory location 1060” outputs low-level data.
- **File organization module understands files, logical address, and physical blocks**
 - Translates logical block # to physical block #
 - Manages free space.

File system implementation: Different problems to be solved

- What to do at boot time, how to locate kernel ?**
- How to store directories and files on the partition ?**
 - Complex problem. Hierarchy + storage allocation + efficiency + limits on file/directory sizes + links (hard, soft)
- How to manage list of free sectors/blocks?**

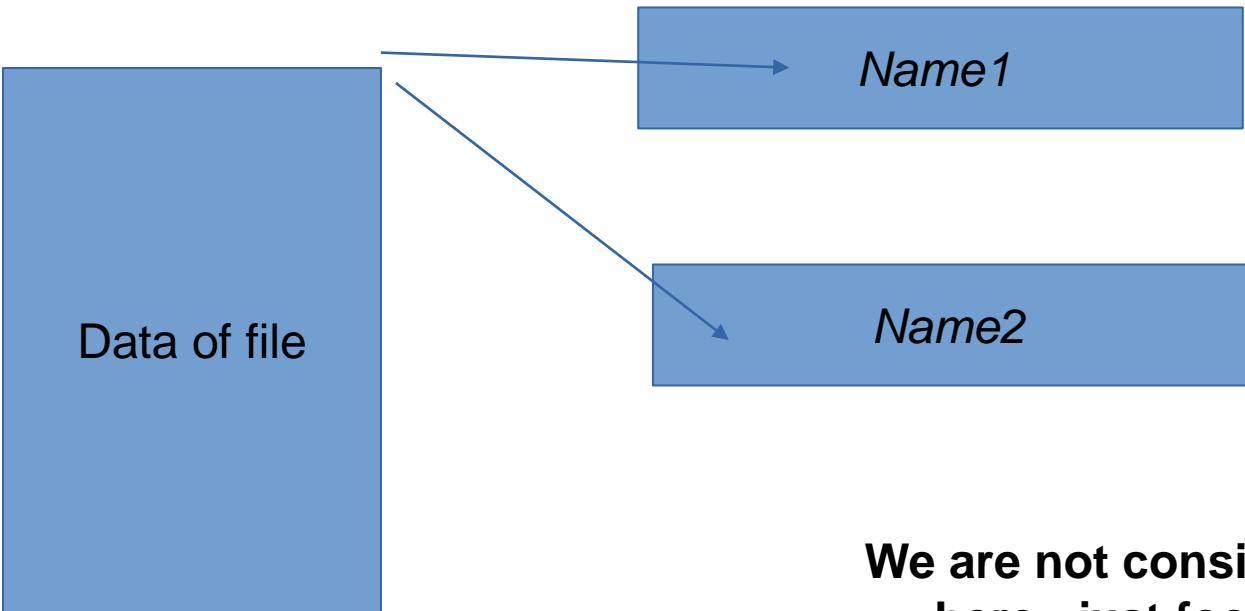
File system implementation: Different problems to be solved

- **About storing a file, how to store**
 - Data
 - Attributes
 - Name
 - Link count

The hard link problem

- **Need to separate name from data !**
 - */x/y and /a/b* should be same file. How?
 - Both names should refer to same data !
 - Data is separated separately from name, and the name gives a “reference” to data
- **What about attributes ?**
 - They go with data! (not with name!)
- **So solution was: indirection !**

The hard link problem



**We are not considering other problems
here , just focussing on hard link
problem**

A typical file control block (inode)

file permissions

**Name is stored
separately**

file dates (create, access, write)

Where?

file owner, group, ACL

**IN data block of
directory**

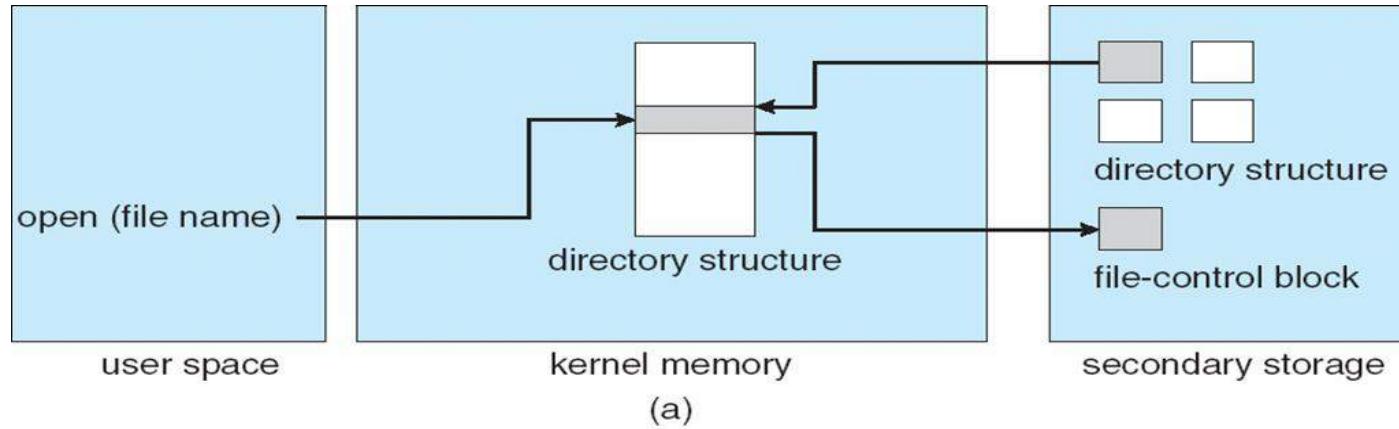
file size

file data blocks or pointers to file data blocks

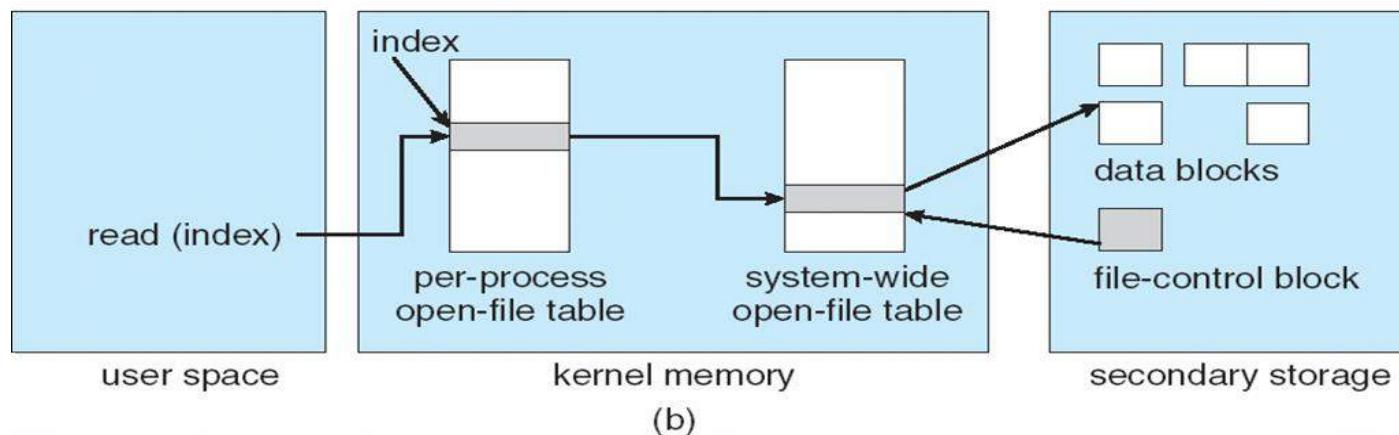
In memory data structures

- **Mount table**
 - storing file system mounts, mount points, file system types
- **See next slide for “file” related data structures**
- **Buffers**
 - hold data blocks from secondary storage

In memory data structures: for open,read,write, ...



Open returns a file handle for subsequent use



Data from read eventually copied to specified user process memory address

At boot time

- **Root partition**
 - Contains the file system hosting OS
 - “mounted” at boot time – contains “/”
 - Normally can’t be unmounted!
- **Check all other partitions**
 - Specified in /etc/fstab on Linux
 - Check if the data structure on them is consistent
 - Consistent != perfect/accurate/complete

Directory Implementation

□ Problem

- Directory contains files and/or subdirectories
- Operations required – create files/directories, access files/directories, search for a file (during lookup), etc.
- Directory needs to give location of each file on disk

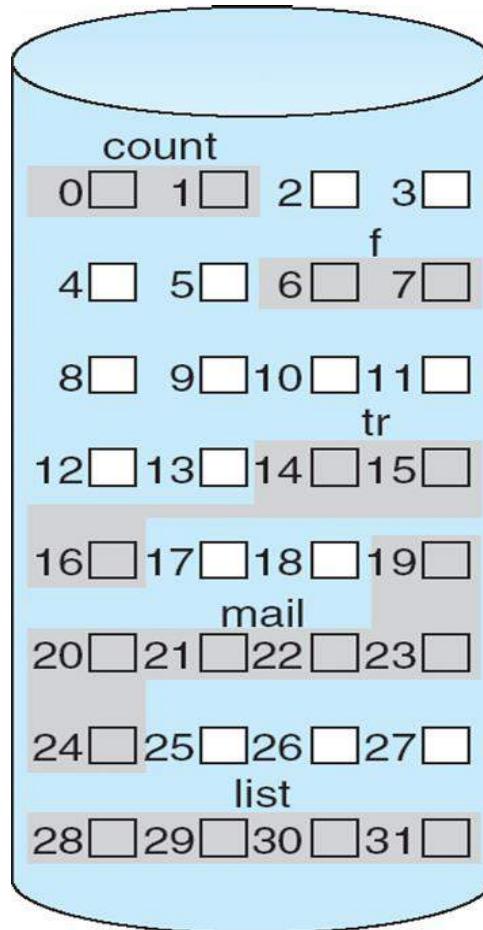
Directory Implementation

- **Linear list of file names with pointer to the data blocks**
 - Simple to program
 - Time-consuming to execute
 - Linear search time
 - Could keep ordered alphabetically via linked list or use B+ tree
 - Ext2 improves upon this approach.

Disk space allocation for files

- **File contain data and need disk blocks/sectors for storing it**
- **File system layer does the allocation of blocks on disk to files**
- **Files need to**
 - Be created, expanded, deleted, shrunk, etc.
 - How to accommodate these requirements?

Contiguous Allocation of Disk Space



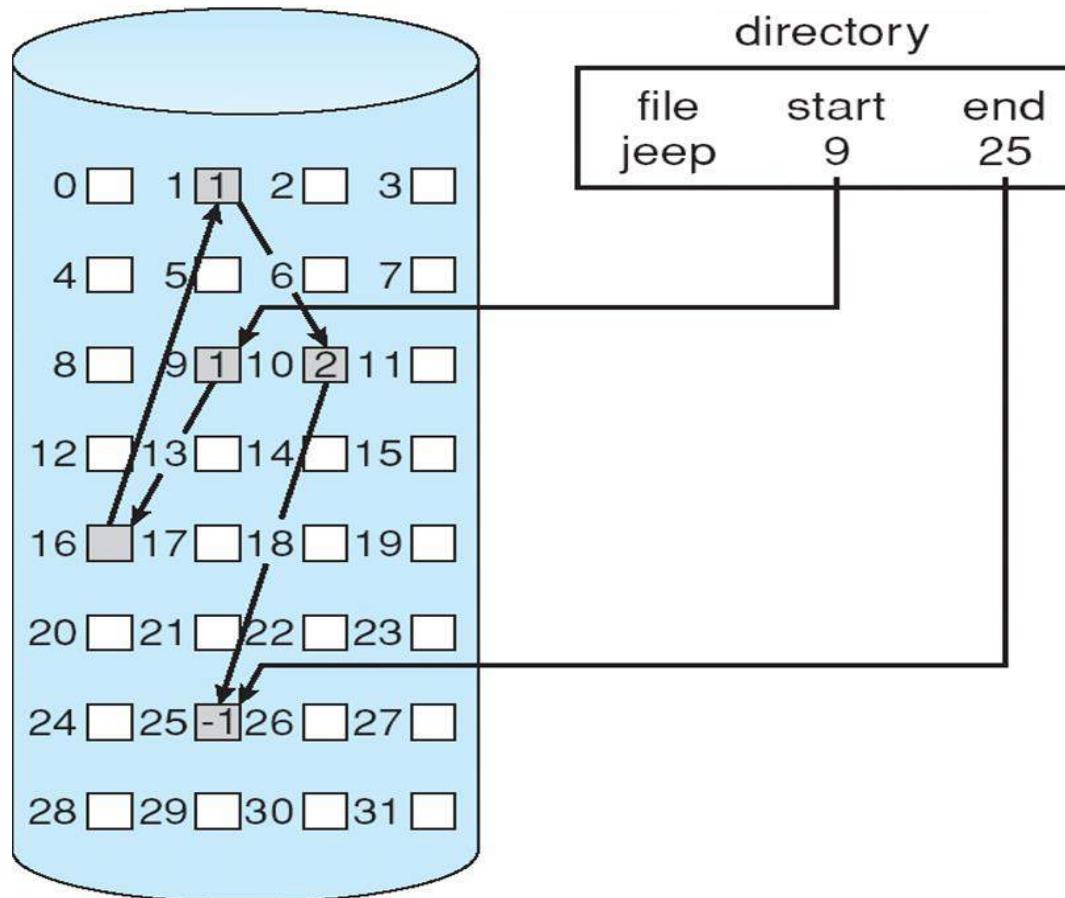
directory

file	start	length
count	0	2
tr	14	3
mail	19	6
list	28	4
f	6	2

Contiguous allocation

- **Each file occupies set of contiguous blocks**
- **Best performance in most cases**
- **Simple – only starting location (block #) and length (number of blocks) are required**
- **Problems include finding space for file, knowing file size, external fragmentation, need for compaction off-line (downtime) or on-line**

Linked allocation of blocks to a file



Linked allocation of blocks to a file

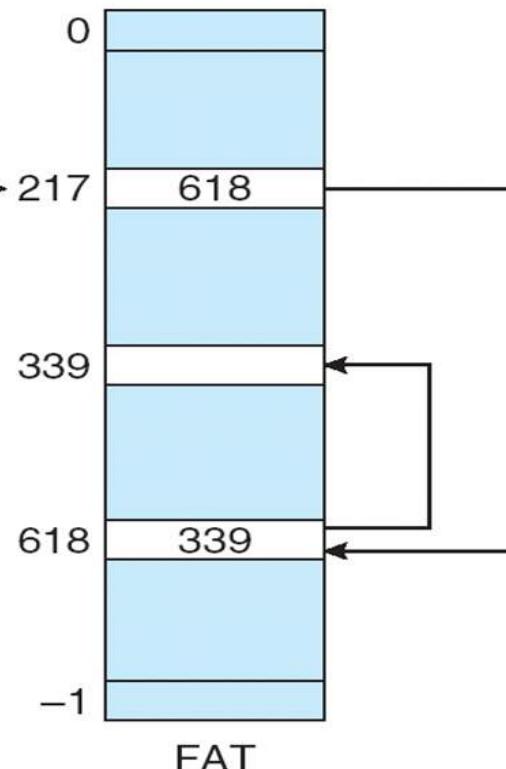
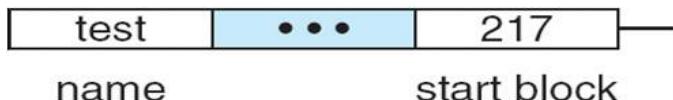
□ **Linked allocation**

- Each file a linked list of blocks
- File ends at nil pointer
- No external fragmentation
- Each block contains pointer to next block (i.e. data + pointer to

- Free space management system called when new block needed
- Improve efficiency by clustering blocks into groups but increases internal fragmentation
- Reliability can be a problem

FAT: File Allocation Table

directory entry



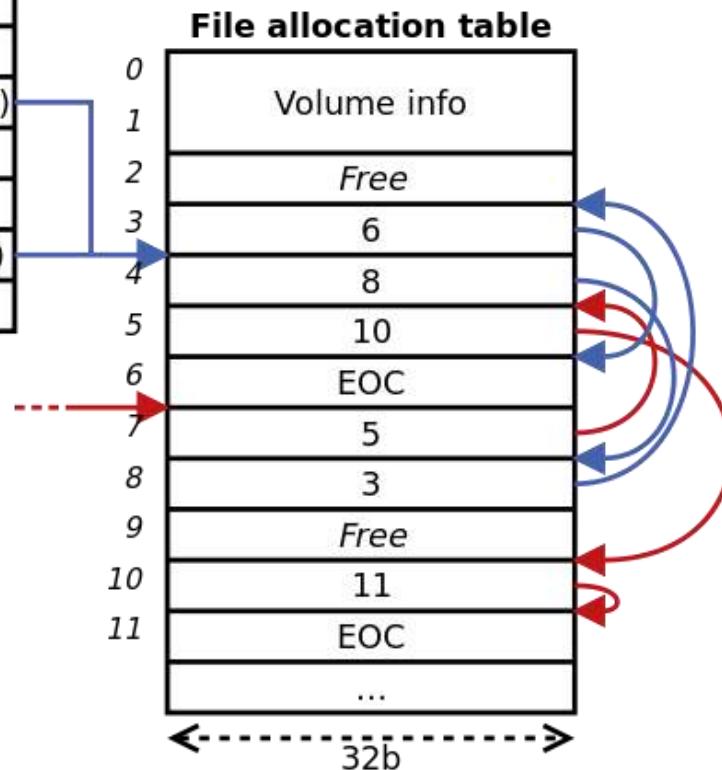
- FAT (File Allocation Table), a variation
 - Beginning of volume has table, indexed by block number
 - Much like a linked list, but faster on disk and cacheable
 - New block allocation simple

FAT: File Allocation Table

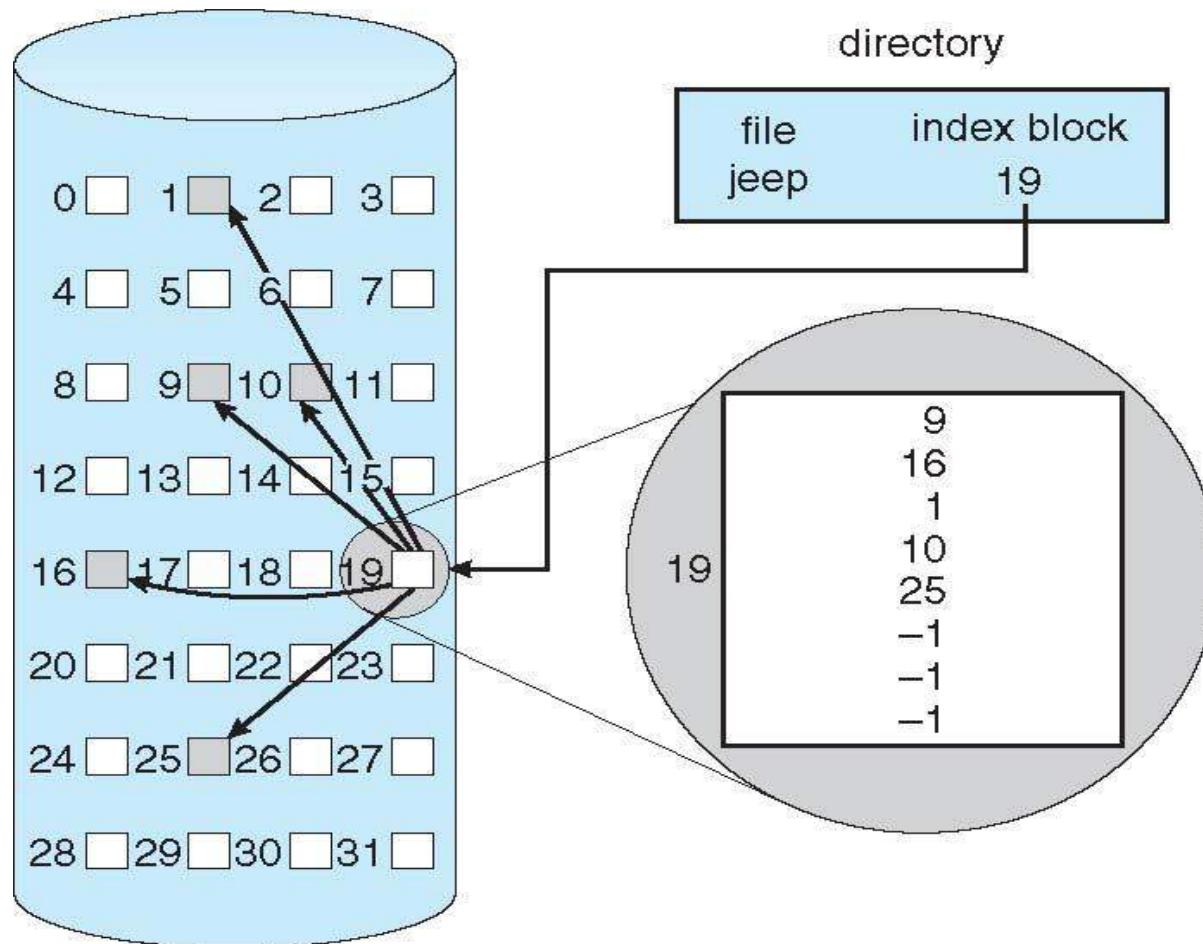
Variants: FAT8,
FAT12, FAT16,
FAT32, VFAT, ...

Directory table entry (32B)

Filename (8B)
Extension (3B)
Attributes (1B)
Reserved (1B)
Create time (3B)
Create date (2B)
Last access date (2B)
First cluster # (MSB, 2B)
Last mod. time (2B)
Last mod. date (2B)
First cluster # (LSB, 2B)
File size (4B)



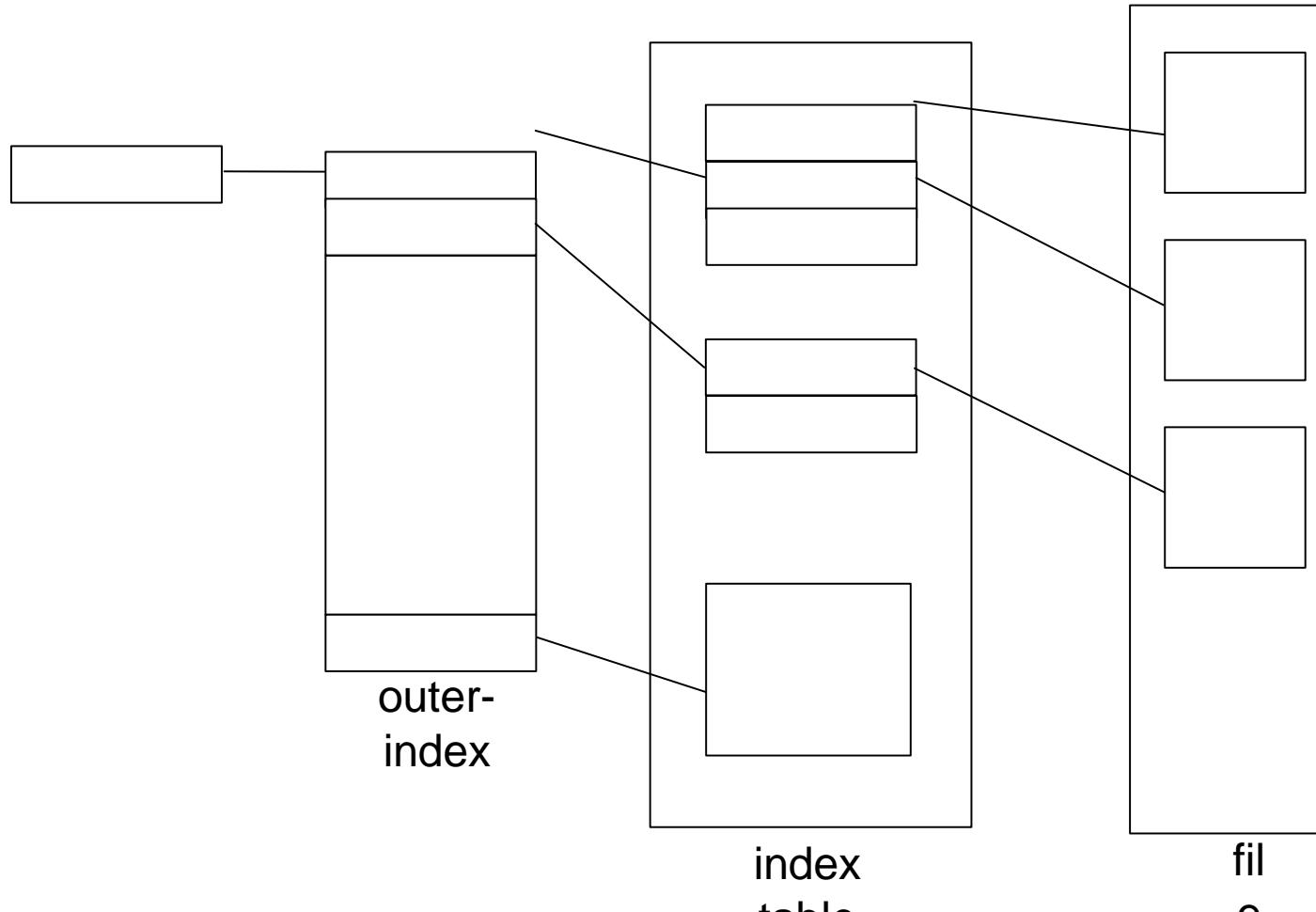
Indexed allocation



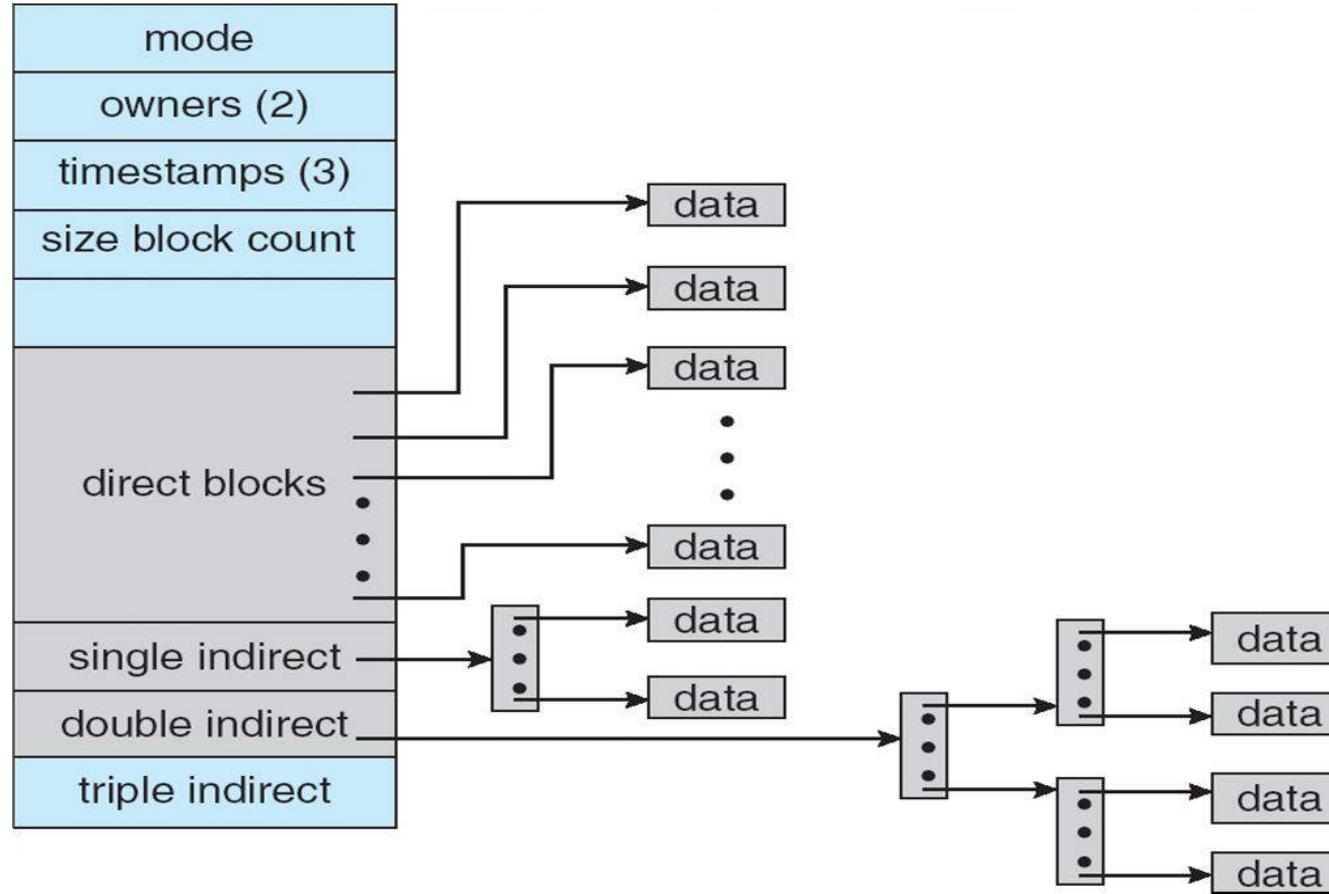
Indexed allocation

- Need index table
- Random access
- Dynamic access without external fragmentation, but have overhead of index block
- Mapping from logical to physical in a file of maximum size of 256K bytes and block size of 512 bytes. We need only 1 block for index

Multi level indexing



Unix UFS: combined scheme for block allocation



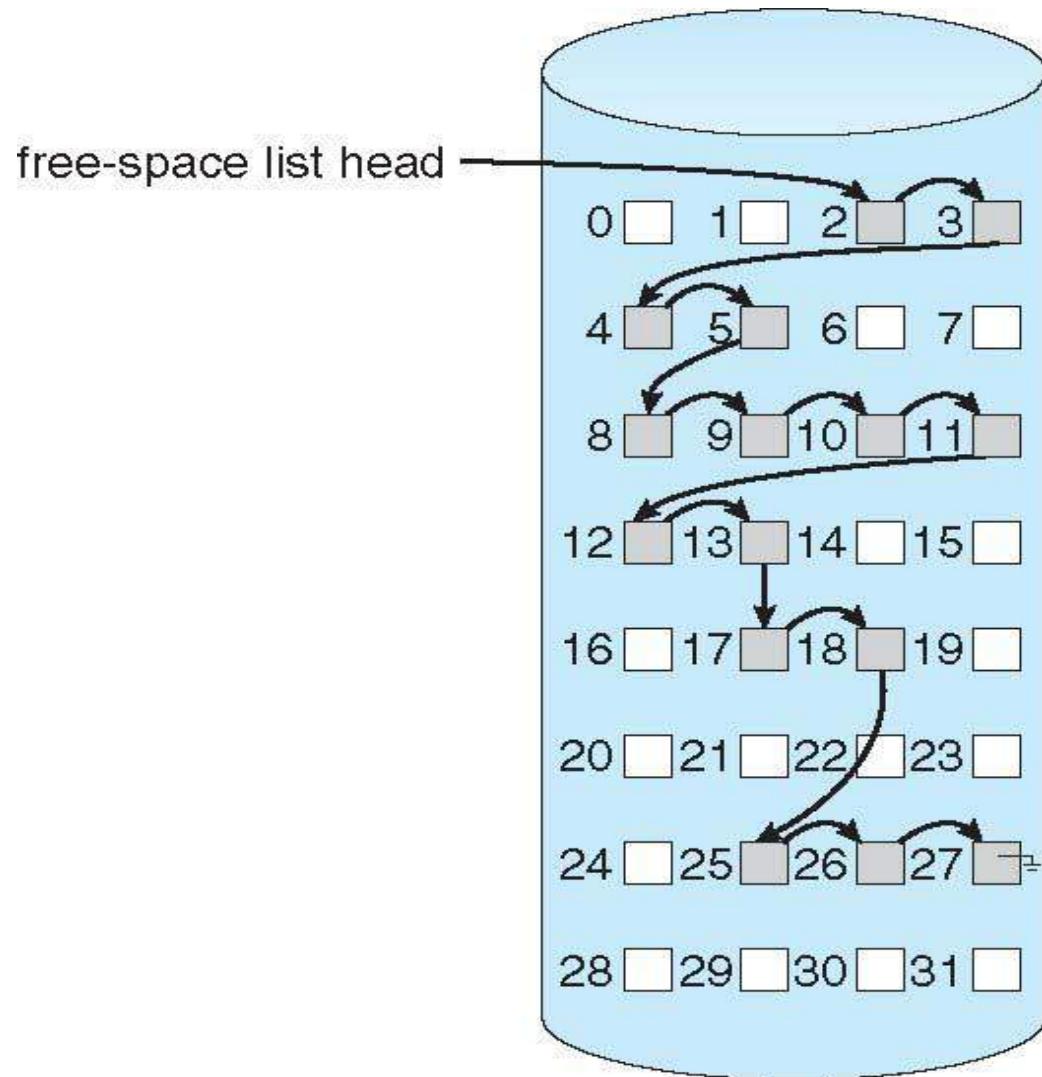
Free Space Management

- **File system maintains free-space list to track available blocks/clusters**
 - Bit vector or bit map (n blocks)
 - Or Linked list

Free Space Management: bit vector

- **Each block is represented by 1 bit.**
- **If the block is free, the bit is 1; if the block is allocated, the bit is 0.**
 - For example, consider a disk where blocks 2, 3, 4, 5, 8, 9, 10, 11, 12, 13, 17 18, 25, 26, and 27 are free and the rest of the blocks are allocated. The free-space bitmap would be
 - 00111100111110001100000011100000 ...

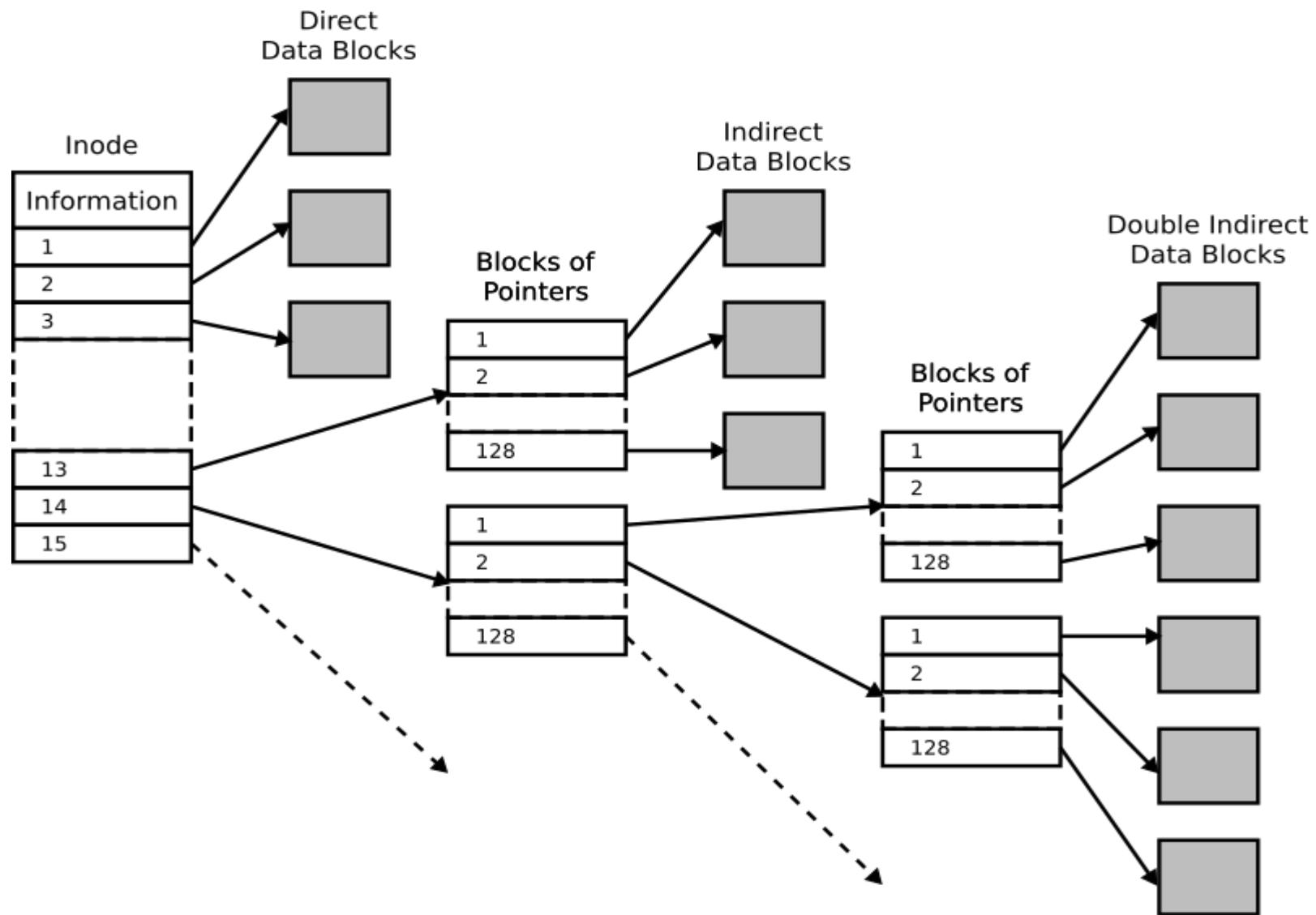
Free Space Management: Linked list (not in memory, on disk!)



Ext2 FS layout

```
struct ext2_inode {  
    __le16 i_mode; /* File mode */  
    __le16 i_uid; /* Low 16 bits of Owner Uid */  
    __le32 i_size; /* Size in bytes */  
    __le32 i_atime; /* Access time */  
    __le32 i_ctime; /* Creation time */  
    __le32 i_mtime; /* Modification time */  
    __le32 i_dtime; /* Deletion Time */  
    __le16 i_gid; /* Low 16 bits of Group Id */  
    __le16 i_links_count; /* Links count */  
    __le32 i_blocks; /* Blocks count */  
    __le32 i_flags; /* File flags */
```

Inode in ext2



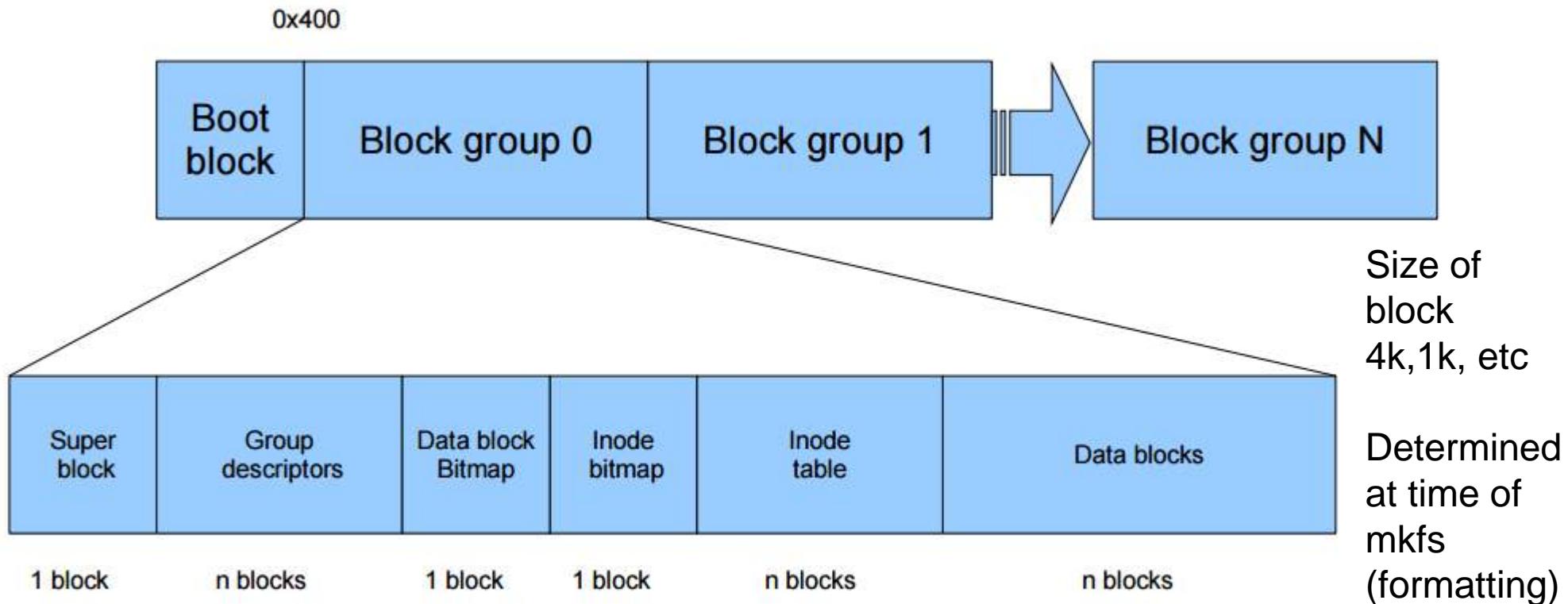
```
struct ext2_inode {  
...  
union {  
    struct {  
        __le32 l_i_reserved1;  
    } linux1;  
    struct {  
        __le32 h_i_translator;  
    } hurd1;  
    struct {  
        __le32 m_i_reserved1;  
    } masix1;  
} osd1; /* OS dependent 1 */  
__le32 i_block[EXT2_N_BLOCKS];/* Pointers to blocks */  
__le32 i_generation; /* File version (for NFS) */  
__le32 i_file_acl; /* File ACL */  
__le32 i_dir_acl; /* Directory ACL */  
__le32 i_faddr; /* Fragment address */
```

```
struct ext2_inode {  
...  
union {  
struct {  
__u8 l_i_frag; /* Fragment number */ __u8 l_i_fsize; /* Fragment size */  
__u16 l_i_pad1; __le16 l_i_uid_high; /* these 2 fields */  
__le16 l_i_gid_high; /* were reserved2[0] */  
__u32 l_i_reserved2;  
} linux2;  
struct {  
__u8 h_i_frag; /* Fragment number */ __u8 h_i_fsize; /* Fragment size */  
__le16 h_i_mode_high; __le16 h_i_uid_high;  
__le16 h_i_gid_high;  
__le32 h_i_author;  
} hurd2;  
struct {  
__u8 m_i_frag; /* Fragment number */ __u8 m_i_fsize; /* Fragment size */  
__u16 m_pad1; __u32 m_i_reserved2[2];  
} masix2;  
} osd2; /* OS dependent 2 */
```

Ext2 FS Layout: Entries in directory's data blocks

	inode	rec_len	file_type	name_len	name				
0	21	12	1	2	.	\0	\0	\0	
12	22	12	2	2	.	.	\0	\0	
24	53	16	5	2	h	o	m	e	1 \0 \0 \0 \0
40	67	28	3	2	u	s	r	\0	
52	0	16	7	1	o	l	d	f	i 1 e \0
68	34	12	4	2	s	b	i	n	

Ext2 FS Layout



Calculations done by “mkfs” like this

- **Block size = 4KB (specified to mkfs)**
- **Number of total blocks = size of partition / 4KB**
 - How to get size of partition ?
- **$4KB = 4 * 1024 * 8 = 32768$ bits**
- **Data Block Bitmap, Inode Bitmap are always one block**
- **So**
 - size of a group is 32,768 Blocks
 - #groups = #blocks-in-partition / 32,768

```
struct ext2_super_block {
    __le32 s_inodes_count; /* Inodes count */
    __le32 s_blocks_count; /* Blocks count */
    __le32 s_r_blocks_count; /* Reserved blocks count */
    __le32 s_free_blocks_count; /* Free blocks count */
    __le32 s_free_inodes_count; /* Free inodes count */
    __le32 s_first_data_block; /* First Data Block */
    __le32 s_log_block_size; /* Block size */
    __le32 s_log_frag_size; /* Fragment size */
    __le32 s_blocks_per_group; /* # Blocks per group */
    __le32 s_frags_per_group; /* # Fragments per group */
    __le32 s_inodes_per_group; /* # Inodes per group */
    __le32 s_mtime; /* Mount time */
    __le32 s_wtime; /* Write time */
    __le16 s_mnt_count; /* Mount count */
    __le16 s_max_mnt_count; /* Maximal mount count */
    __le16 s_magic; /* Magic signature */
    __le16 s_state; /* File system state */
    __le16 s_errors; /* Behaviour when detecting errors */
```

```
struct ext2_super_block {  
...  
    __le16 s_minor_rev_level; /* minor revision level */  
    __le32 s_lastcheck; /* time of last check */  
    __le32 s_checkinterval; /* max. time between checks */  
    __le32 s_creator_os; /* OS */  
    __le32 s_rev_level; /* Revision level */  
    __le16 s_def_resuid; /* Default uid for reserved blocks */  
    __le16 s_def_resgid; /* Default gid for reserved blocks */  
    __le32 s_first_ino; /* First non-reserved inode */  
    __le16 s_inode_size; /* size of inode structure */  
    __le16 s_block_group_nr; /* block group # of this superblock */  
    __le32 s_feature_compat; /* compatible feature set */  
    __le32 s_feature_incompat; /* incompatible feature set */  
    __le32 s_feature_ro_compat; /* readonly-compatible feature set */  
    __u8 s_uuid[16]; /* 128-bit uuid for volume */  
    char s_volume_name[16]; /* volume name */  
    char s_last_mounted[64]; /* directory where last mounted */  
    __le32 s_algorithm_usage_bitmap; /* For compression */
```

```
struct ext2_super_block {  
...  
    __u8 s_prealloc_blocks; /* Nr of blocks to try to preallocate */  
    __u8 s_prealloc_dir_blocks; /* Nr to preallocate for dirs */  
    __u16 s_padding1;  
/*  
 * Journaling support valid if EXT3_FEATURE_COMPAT_HAS_JOURNAL set.  
 */  
    __u8 s_journal_uuid[16]; /* uuid of journal superblock */  
    __u32 s_journal_inum; /* inode number of journal file */  
    __u32 s_journal_dev; /* device number of journal file */  
    __u32 s_last_orphan; /* start of list of inodes to delete */  
    __u32 s_hash_seed[4]; /* HTREE hash seed */  
    __u8 s_def_hash_version; /* Default hash version to use */  
    __u8 s_reserved_char_pad;  
    __u16 s_reserved_word_pad;  
    __le32 s_default_mount_opts;  
    __le32 s_first_meta_bg; /* First metablock block group */  
    __u32 s_reserved[190]; /* Padding to the end of the block */
```

```
struct ext2_group_desc
{
    __le32 bg_block_bitmap; /* Blocks bitmap block */
    __le32 bg_inode_bitmap; /* Inodes bitmap block */
    __le32 bg_inode_table; /* Inodes table block */
    __le16 bg_free_blocks_count; /* Free blocks count */
    __le16 bg_free_inodes_count; /* Free inodes count */
    __le16 bg_used_dirs_count; /* Directories count */
    __le16 bg_pad;
    __le32 bg_reserved[3];
};
```

Traversal / path-name resolution

//resolving /a/b

S -

Let's see a program to read superblock of an ext2
file system.

Processes in xv6 code

Process Table

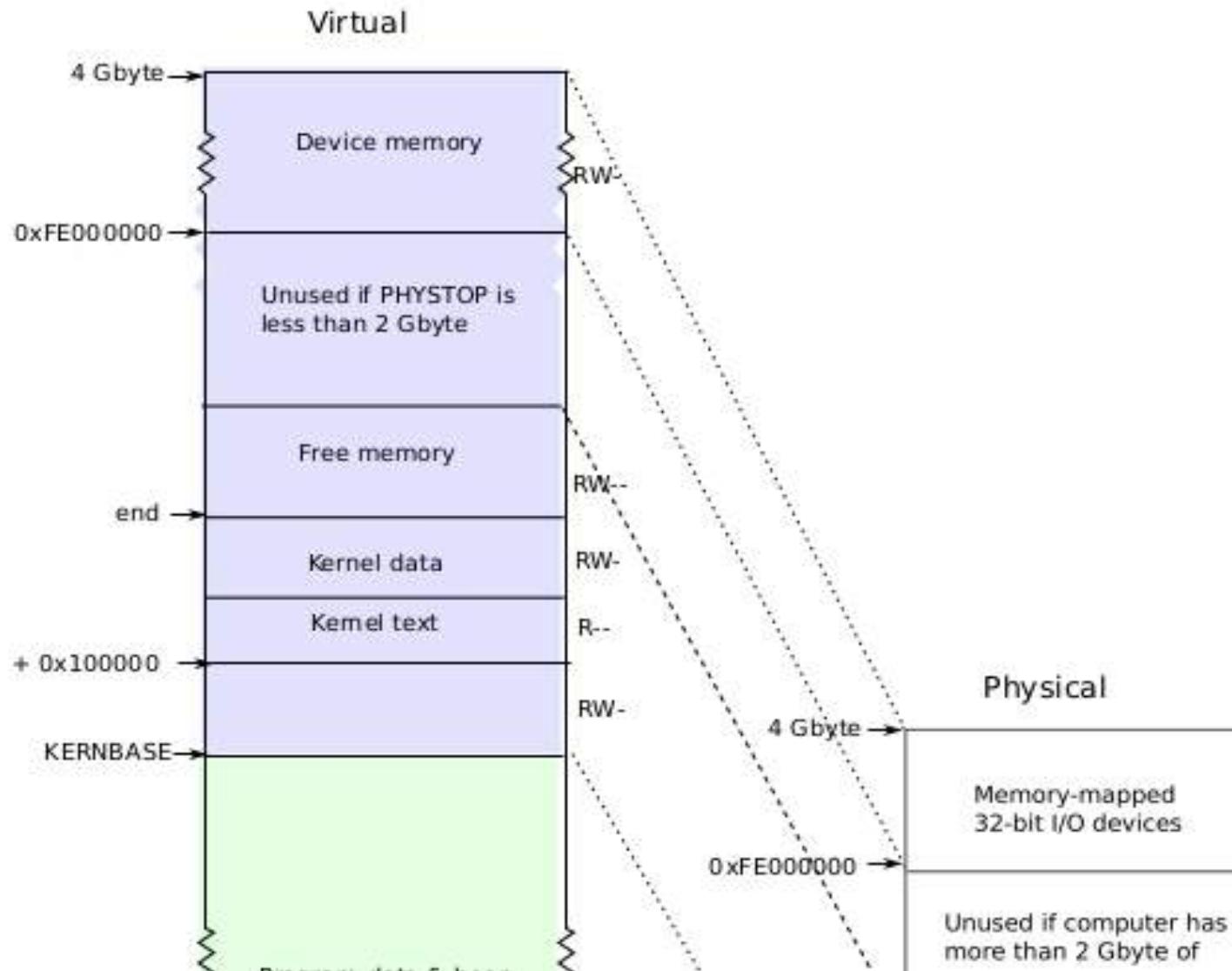
```
struct {  
    struct spinlock lock;  
    struct proc proc[NPROC];  
} ptable;
```

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

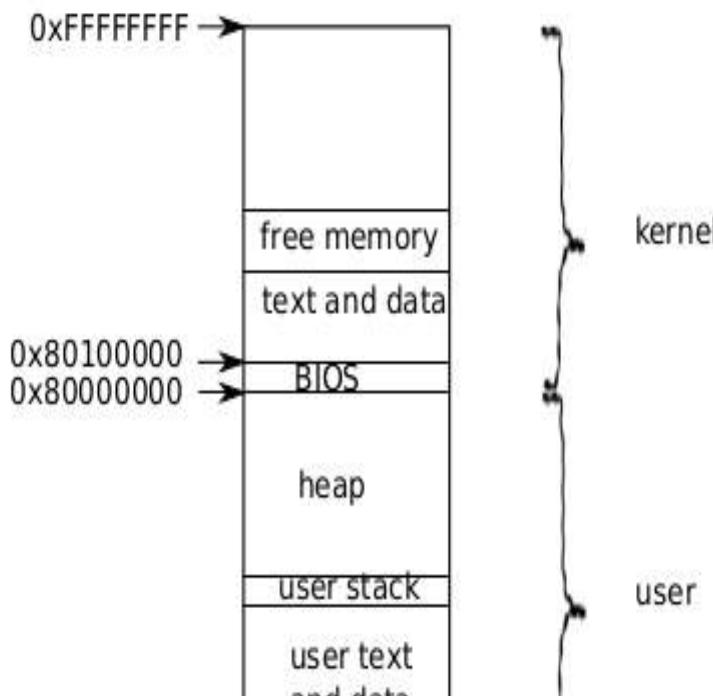
Layout of process's VA space

xv6
schema!

different
from Linux

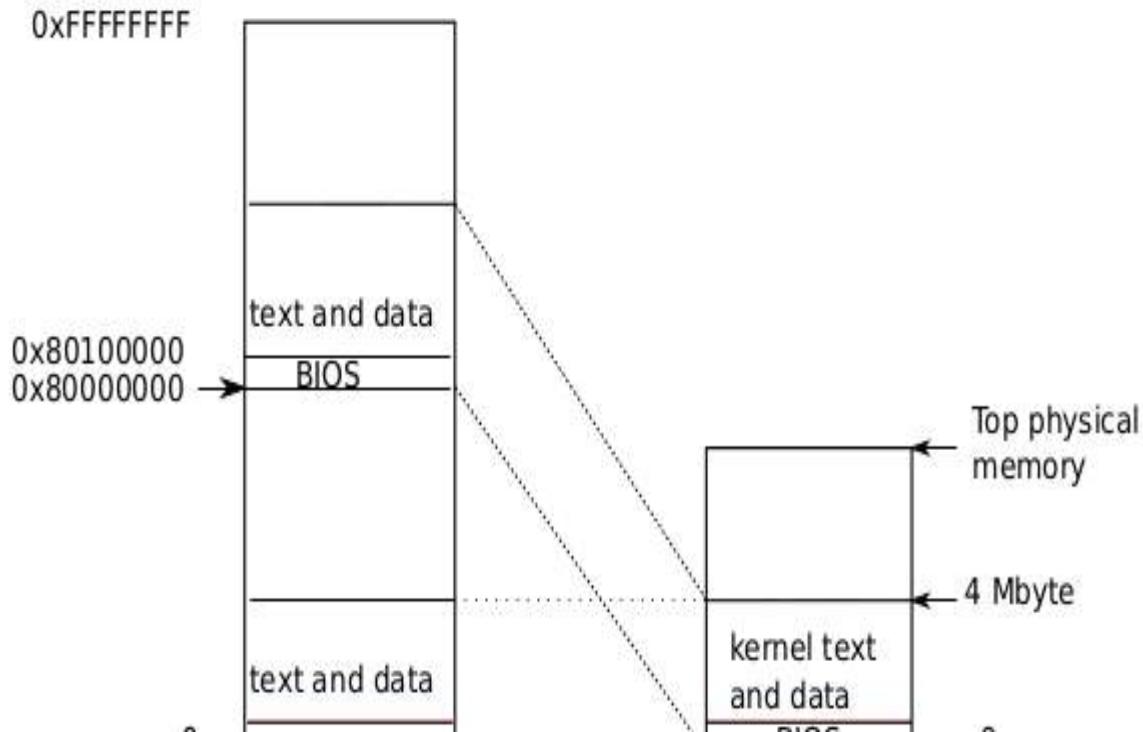


Logical layout of memory for a process



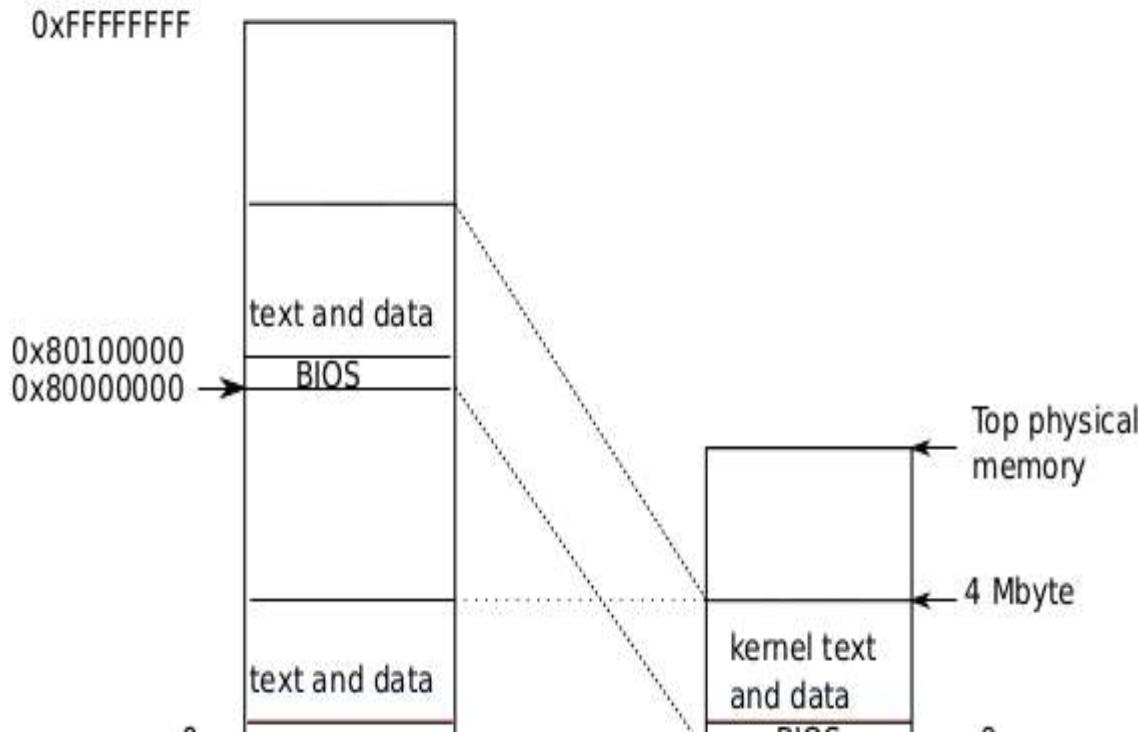
- **Address 0: code**
- **Then globals**
- **Then stack**
- **Then heap**

Kernel mappings in user address space actual location of kernel



- Kernel is loaded at **0x100000 physical address**
- PA 0 to

Kernel mappings in user address space actual location of kernel



□ Kernel is
not
loaded at
the PA
0x8000000
0 because

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 class!

Struct proc

// Per-process state

struct proc {

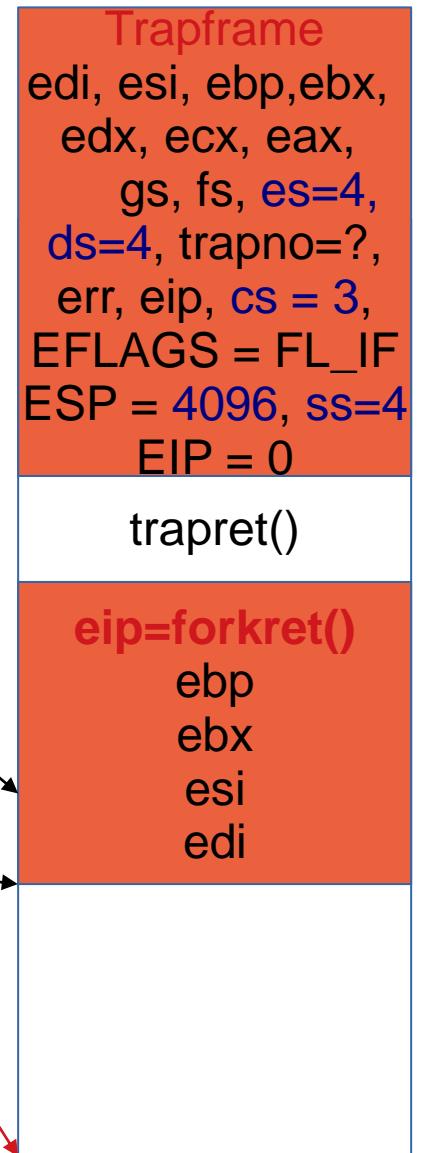
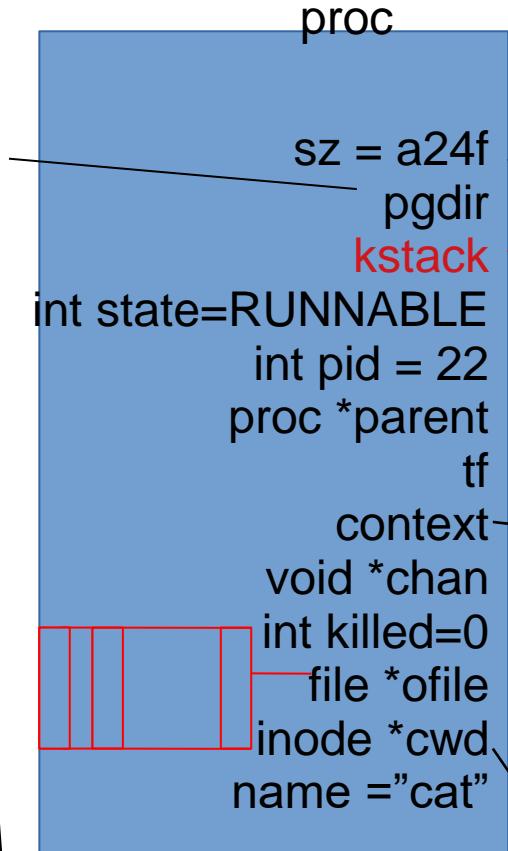
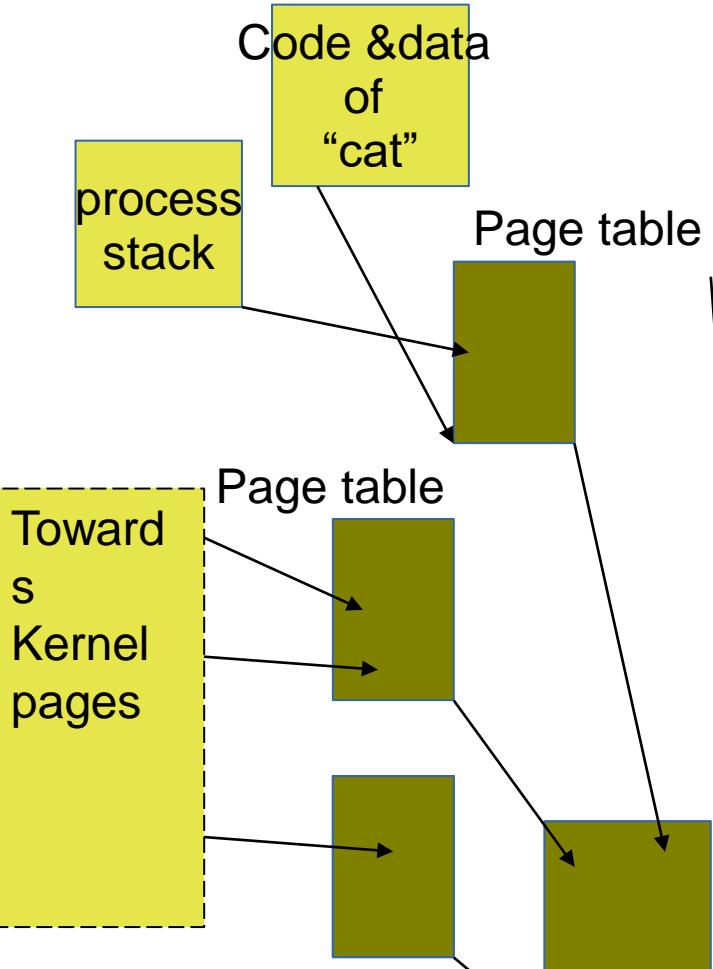
 uint sz; // Size of process memory
 // (bytes)

 pde_t* pgdir; // Page table

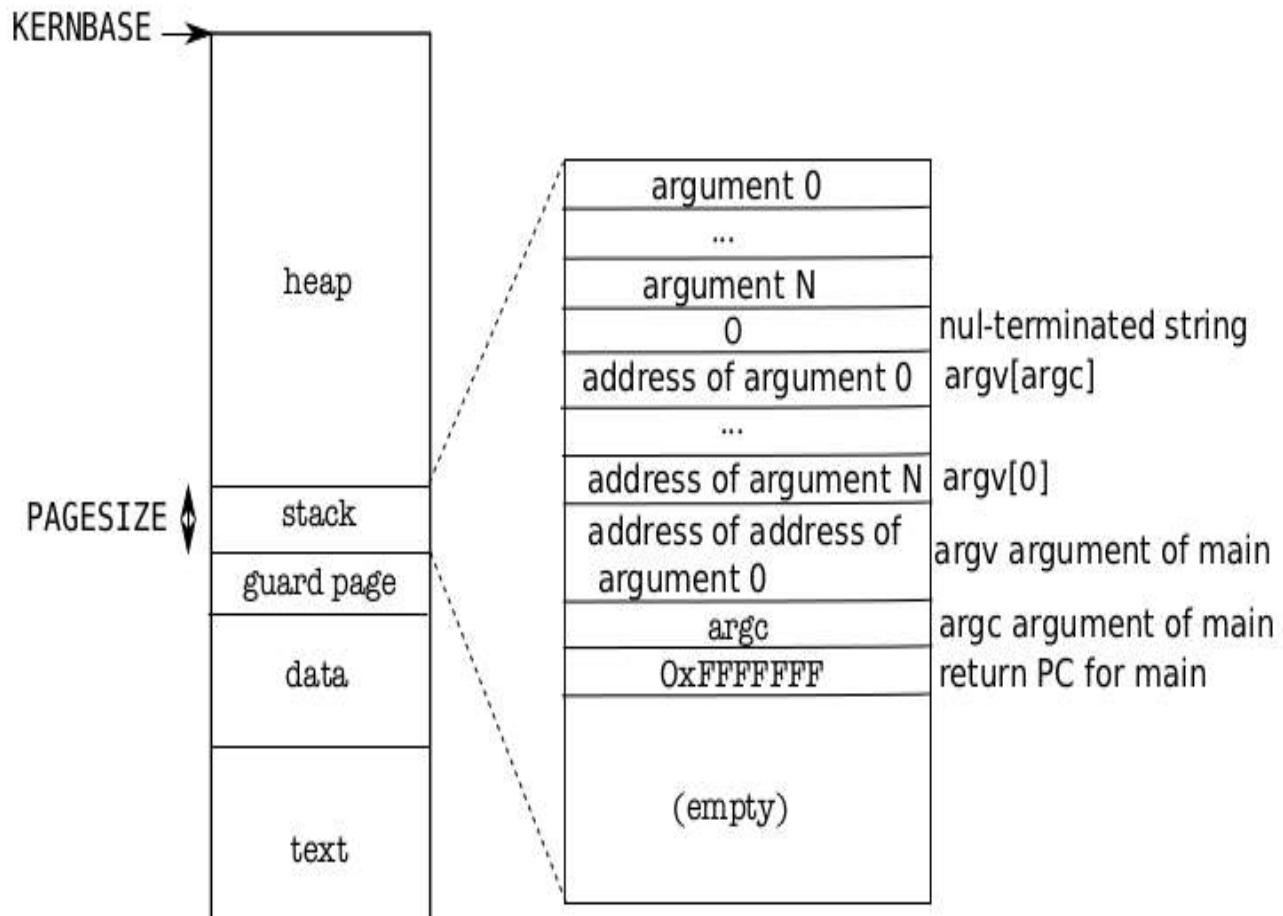
 char *kstack; // Bottom of kernel stack

};

struct proc diagram: Very imp!



Memory Layout of a user process



Memory Layout of a user process

After exec()

Note the argc, argv on stack

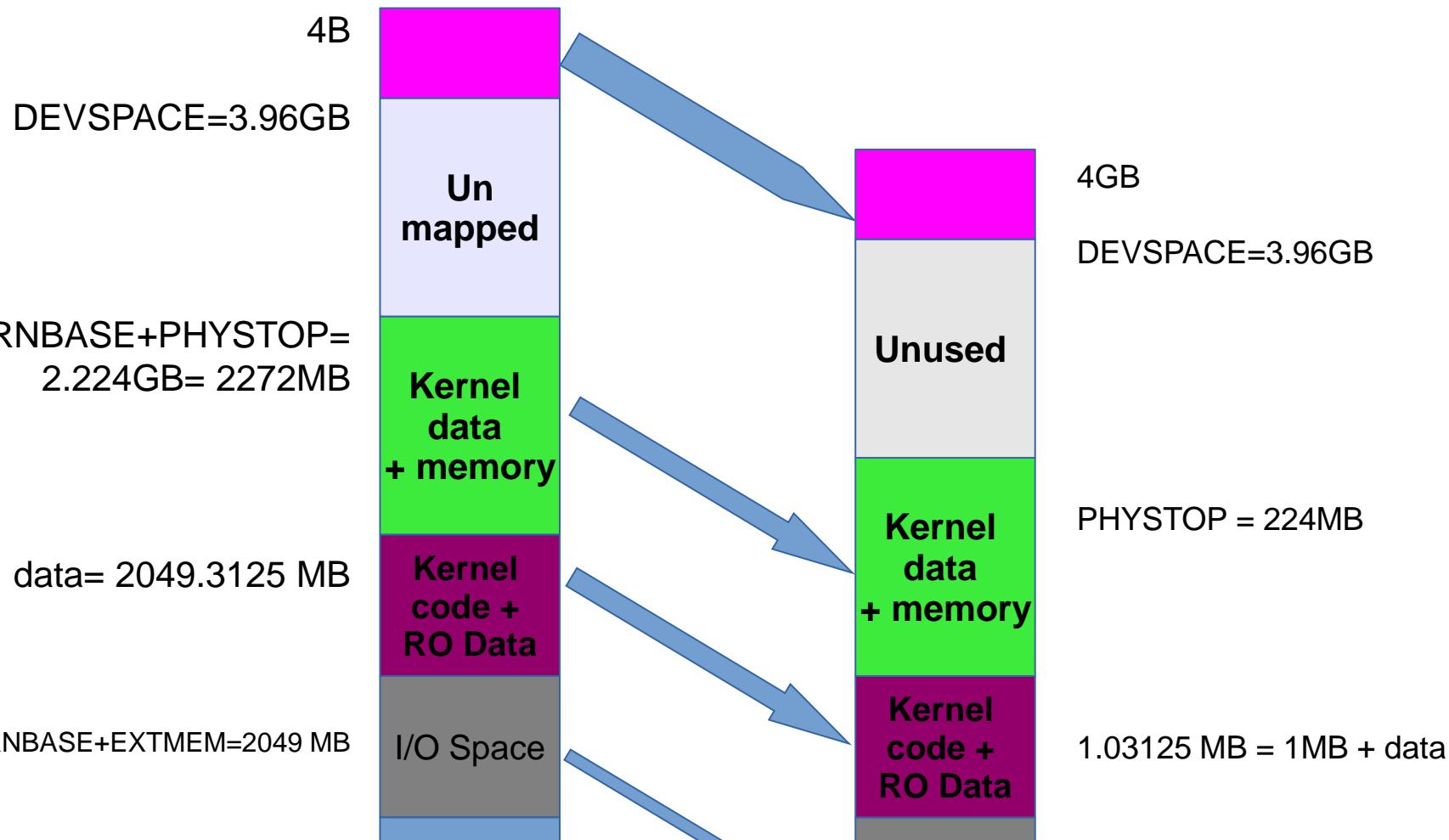
The “guard page” is just a mapping in page table. No frame allocated. It’s marked as invalid. So if stack grows (due to many function calls)

Handling Traps

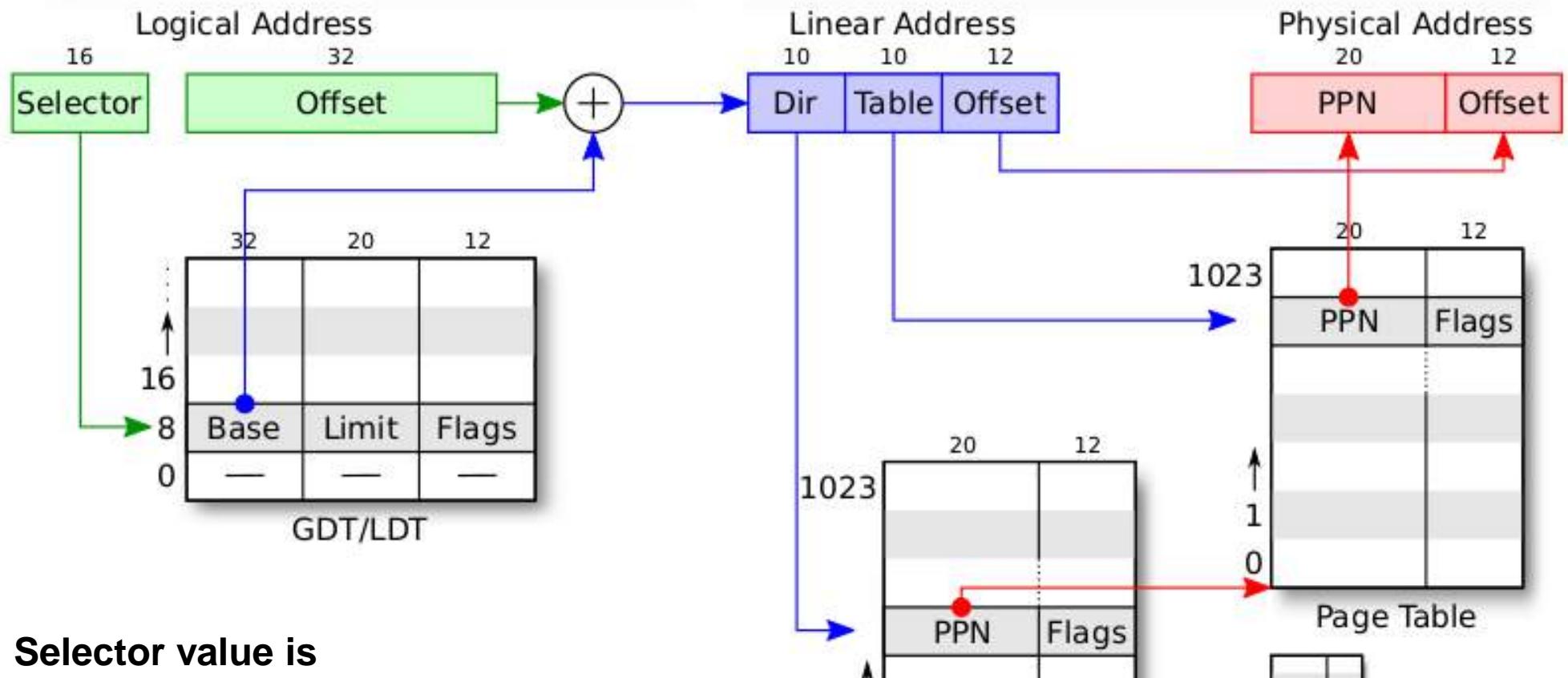
Some basic steps

- **Xv6.img is created by “make”**
 - Contains bootsector, kernel code, kernel data
- **QEMU boots using xv6.img**
 - First it runs bootloader
 - Kernel running..
 - Kernel calls main() of kernel (NOT a C application!) & Initializes:
 - memory management data structures
 - process data structures
 - file handling data structures

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

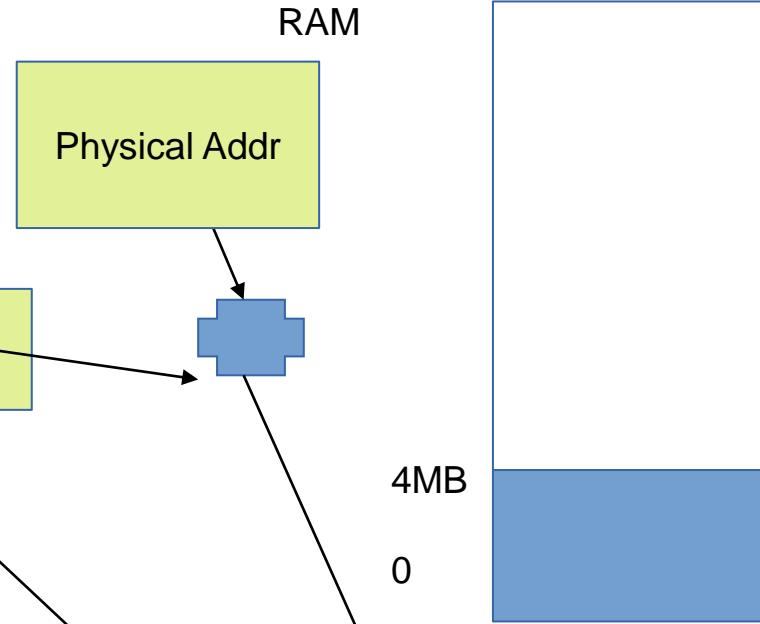
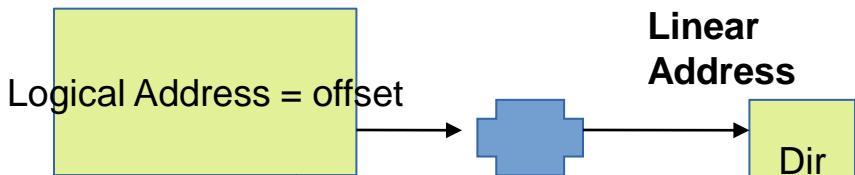


Segmentation + Paging



After seginit() in main().

On the processor where we started booting



CS, SS, etc.



	Base	Limit	Permissions	DPL
4	0	4GB	Write	3
3	0	4GB	Read, Execute	3
2	0	4GB	Write	0
1	0	4GB	Read, Execute	0
0	0	0	0	0

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

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

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

Privilege level

- Changes automatically on

- “int” instruction

- hardware interrupt

- exception

- Changes back on

- iret

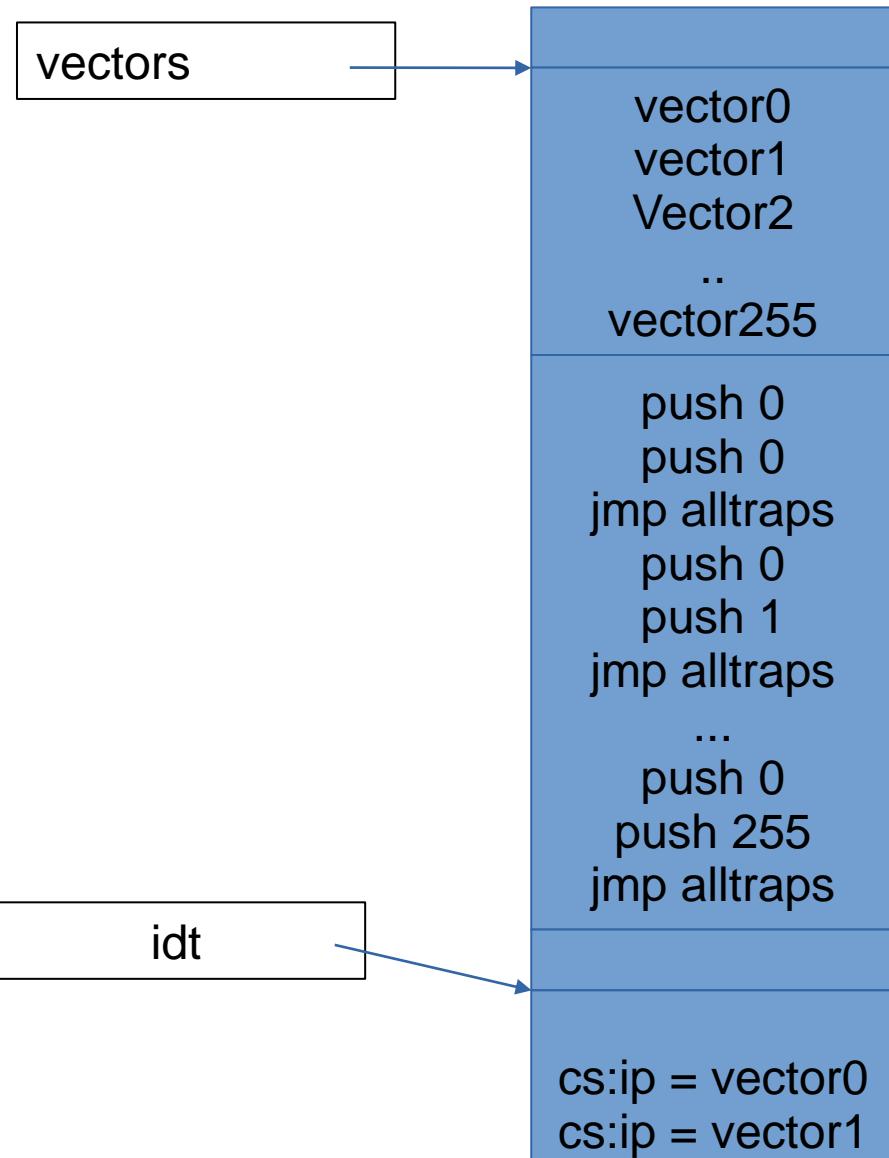
Interrupt Descriptor Table (IDT)

- **IDT defines interrupt handlers**
- **Has 256 entries**
 - each giving the %cs and %eip to be used when handling the corresponding interrupt.
- **Mapping**
 - Interrupts 0-31 are defined for software exceptions, like divide errors or attempts to access invalid

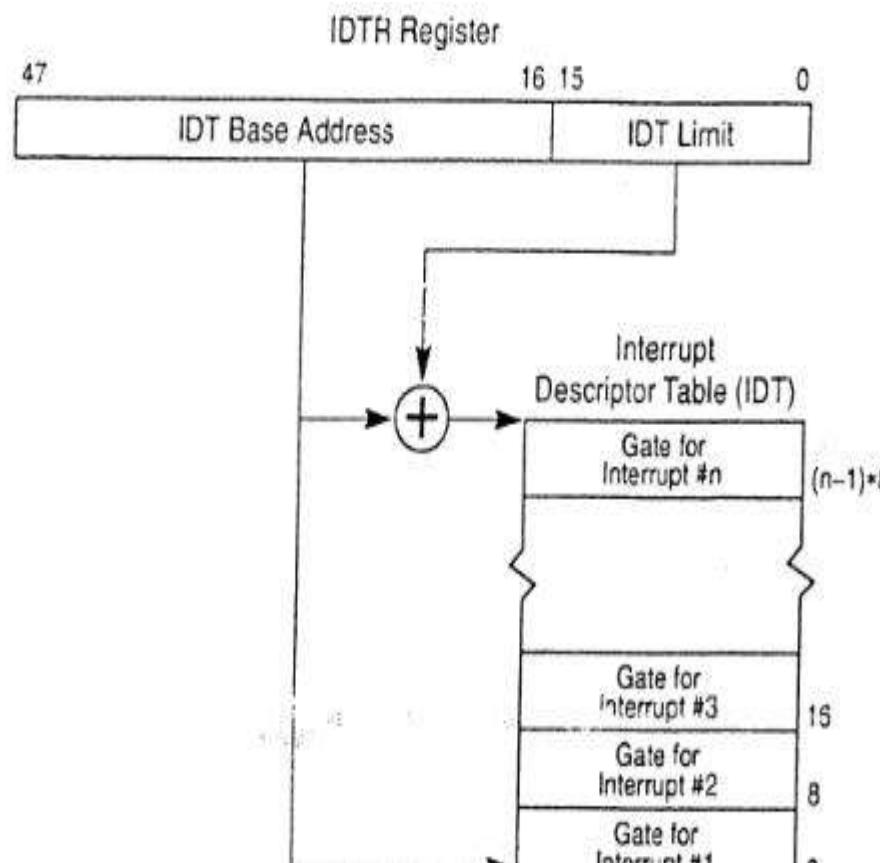
IDT setup
done by tvinit() function

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

The tvinit() is called during
kernel initialization



IDTR and IDT



IDT is in RAM

IDTR is in CPU

Interrupt Descriptor Table (IDT) entries (in RAM)

```
// Gate descriptors for interrupts  
and traps
```

```
struct gatedesc {  
  
    uint off_15_0 : 16; // low 16 bits of  
    offset in segment  
  
    uint cs : 16; // code segment  
    selector
```

Setting IDT entries

```
void  
tvinit(void)  
{  
int i;  
for(i = 0; i < 256; i++)  
SETGATE(idt[i], 0, SEG_KCODE<<3,
```

Setting IDT entries

```
#define SETGATE(gate, istrap, sel,  
off, d)  \  
{ \  
    (gate).off_15_0 = (uint)(off) &  
0xffff; \  
    (gate).cs = (sel); \  
    (gate).args = 0; \  
}
```

Setting IDT entries

Vectors.S

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

```
# handlers
```

```
.globl alltraps  
globl vector0
```

trapasm.S

```
#include "mmu.h"  
  
# vectors.S sends all  
traps here.
```

```
.globl alltraps
```

```
alltraps:
```

```
# Build trap frame.
```

How will interrupts be handled?

On int instruction/interrupt the CPU does this:

- Fetch the n'th descriptor from the IDT, where n is the argument of int. (`IDTR->idt[n]`)
- Check that CPL in `%cs` is \leq DPL, where DPL is the
- Push `%ss`. // optional
- Push `%esp`. // optional (also changes ss,esp using TSS)
- Push `%eflags`.
- Push `%cs`.

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

Build trap frame.

pushl %ds

pushl %es

pushl %fs

pushl %gs

**pushal // push all gen
purpose regs**

Set up data
segments

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

```
void  
trap(struct trapframe  
*tf)
```

```
{
```

```
if(tf->trapno ==  
T_SYSCALL){
```

```
if(myproc()->killed)
```

```
exit();
```

```
myproc()->tf = tf;
```

trap()

- Argument is trapframe
- In all traps
 - Before “call trap”, there was “push %esp” and stack had the trapframe
 - Remember calling

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()

when trap() returns

- #Back in alltraps

call trap

addl \$4, %esp

Return falls through
to trapret...

- 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

- xor ax, ax
add al, al

XV6 bootloader

Abhijit A. M.

abhijit.comp@coep.ac.in

Credits:

xv6 book by Cox, Kaashoek, Morris

Notes by Prof. Soray Bansal

Boot-process

- **Bootloader itself**

- gets 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

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,**

bootloader

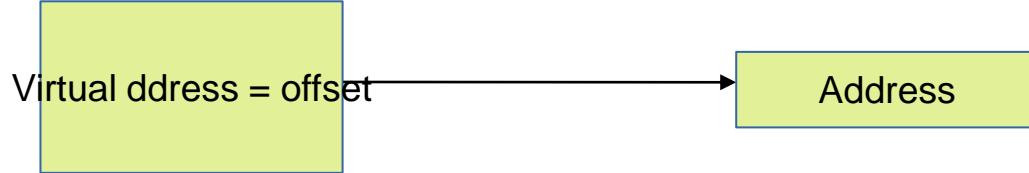
- BIOS Runs (automatically)
- Loads boot sector into RAM at 0x7c00
- Starts executing that code
 - How to make sure that your bootloader is loaded at 0x7c00 ?
 - Makefile has

bootblock: bootblock.S bootmain.c

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
```

- Seg:off with 16 bit segments can actually address more than 20 bits of memory. After 0x100000 ($=2^{20}$), 8086 wrapped addresses to 0.
- 80286 introduced 21st bit of address. But older software

After this

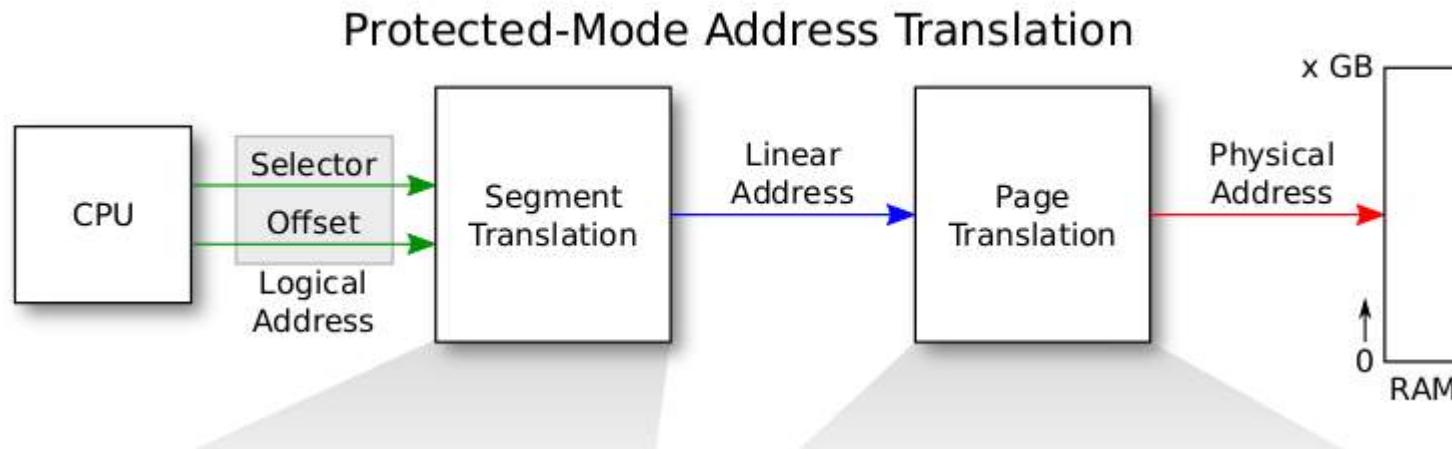
**Some instructions are run
to enter protected mode**

And further code runs in protected mode

Real mode Vs protected mode

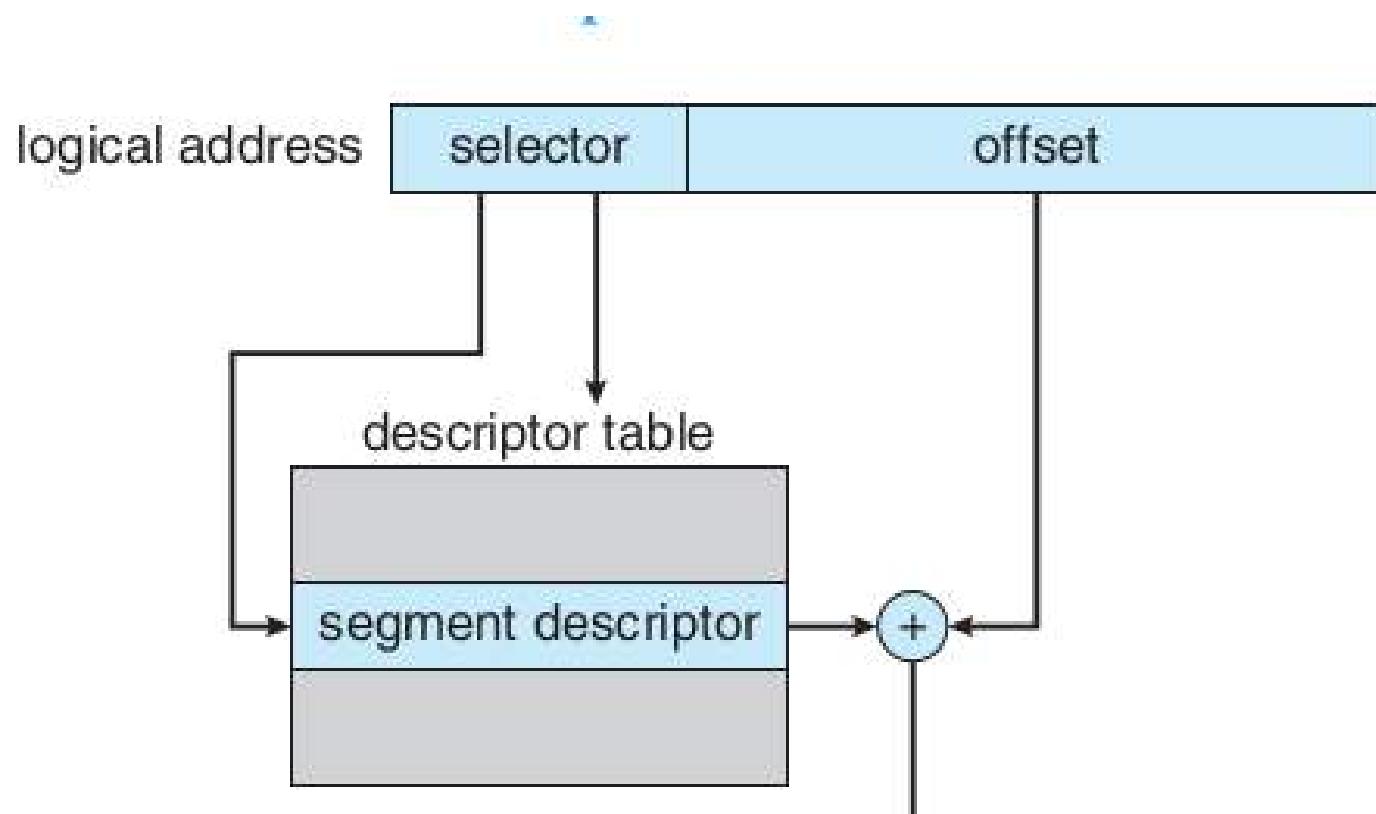
- **Real mode 16 bit registers**
- **Protected mode**
 - Enables segmentation + Paging both
 - No longer seg*16+offset calculations
 - Segment registers is index into segment descriptor table.
But segment:offset pairs continue
 - `mov %esp, $32` # SS will be used with esp
 - More in next few slides

X86 address : protected mode address translation



Both Segmentation and Paging are used in x86

X86 segmentation



Paging concept, hierarchical paging

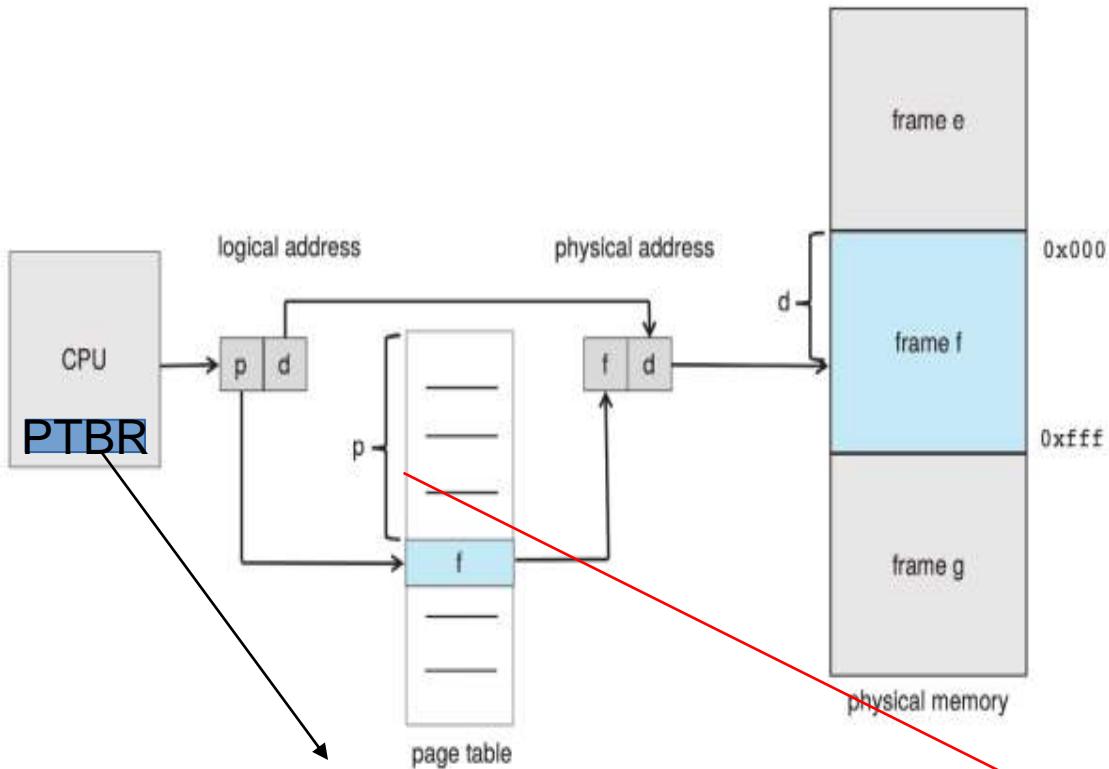
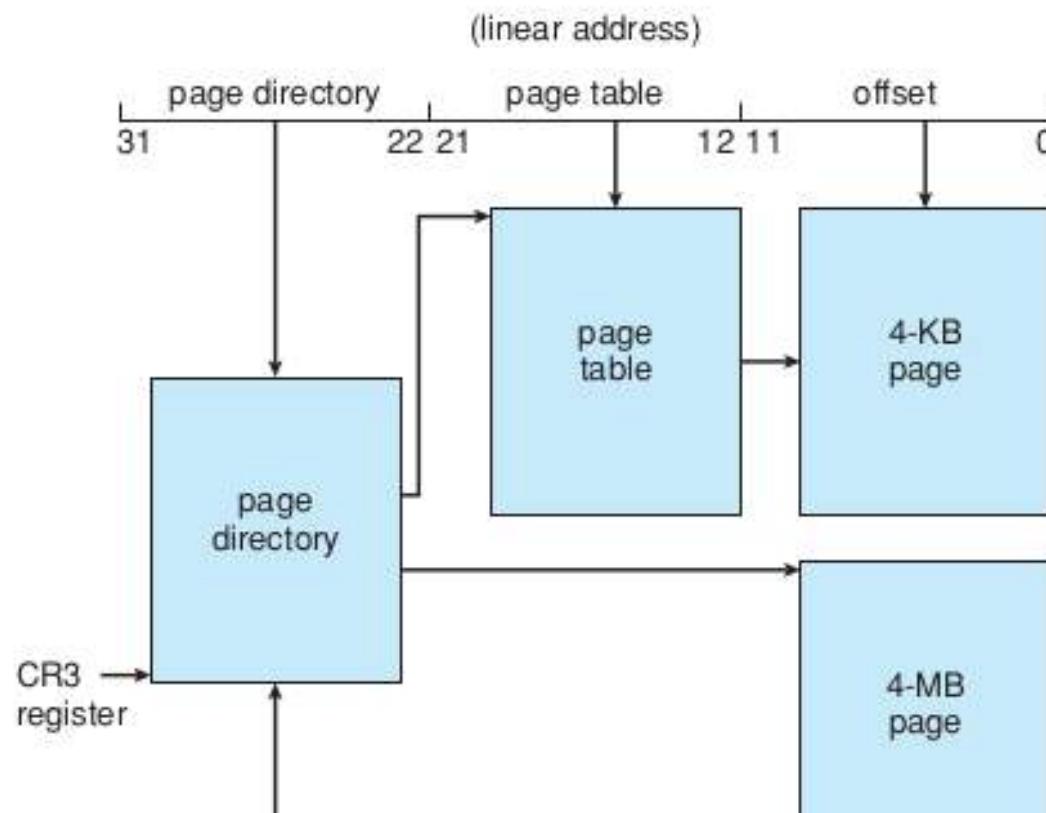


Figure 9.8 Paging hardware.

logical address
p₁ p₂ d

X86 paging



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 A	C D	W T	U D	W T	P U				

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)

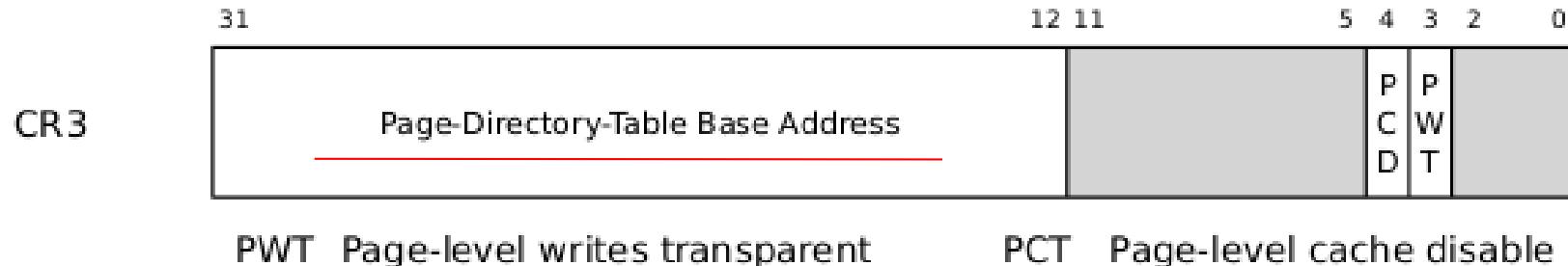
PAT Page table attribute index

G Global page

31

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

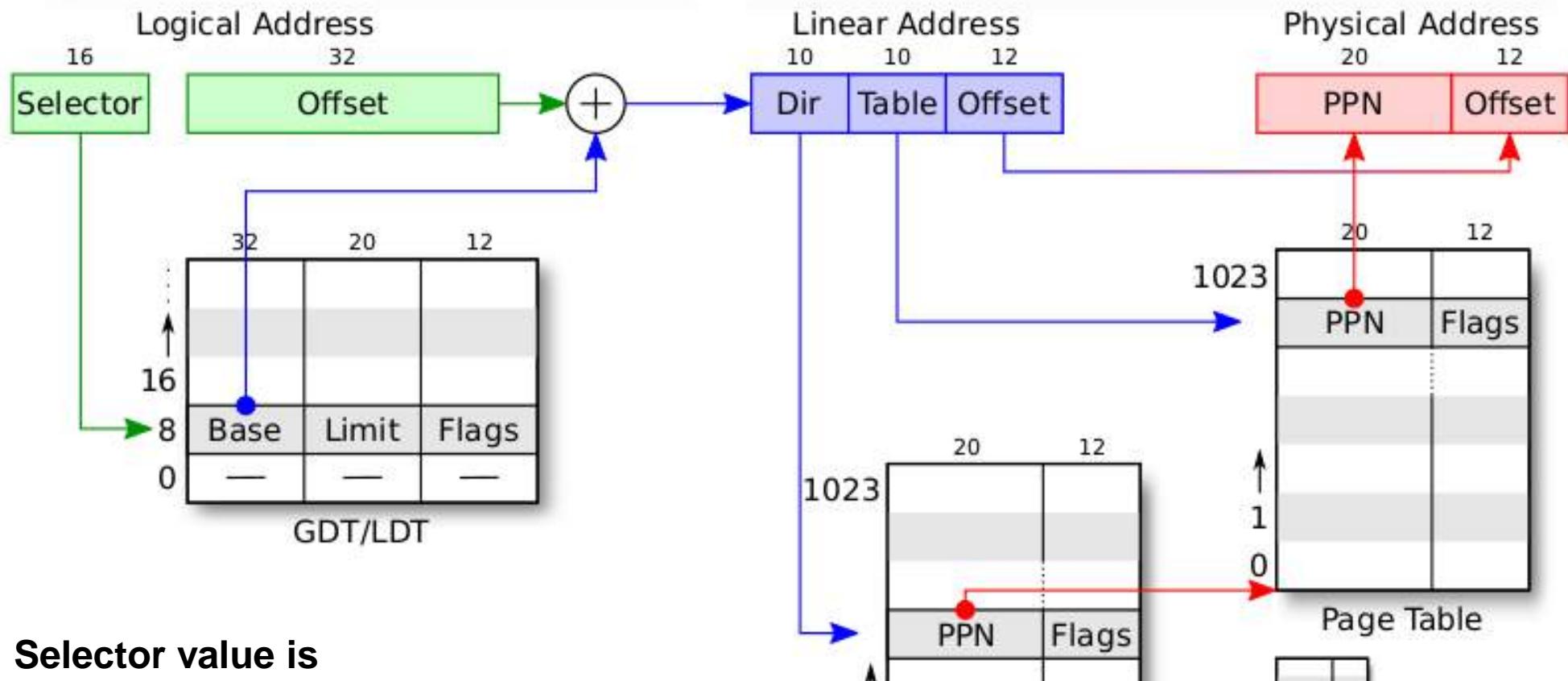
CR3



CR4



Segmentation + Paging



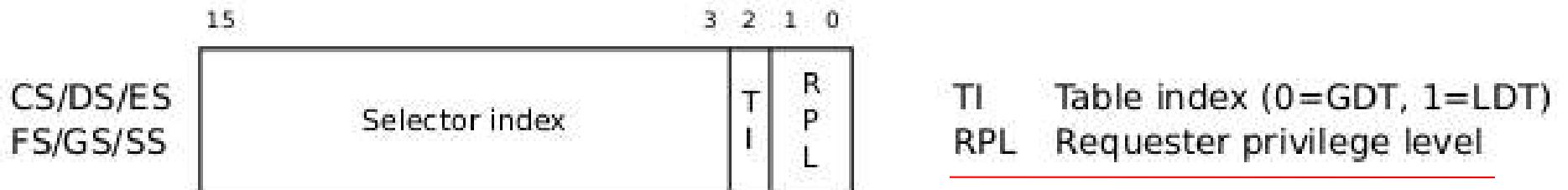
Segmentation + Paging setup of xv6

- 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

GDT Entry

31	16	15	0		
Base 0:15 <hr/>		Limit 0:15 <hr/>			
63	56	55	52		
51	48	47	40	39	32
Base 24:31 <hr/>	Flags	Limit 16:19 <hr/>	Access Byte	Base 16:23 <hr/>	

Segment selector



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

EFLAGS register



lgdt gdtdesc

...

Bootstrap GDT

```
.p2align 2 # force 4  
byte alignment
```

gdt:

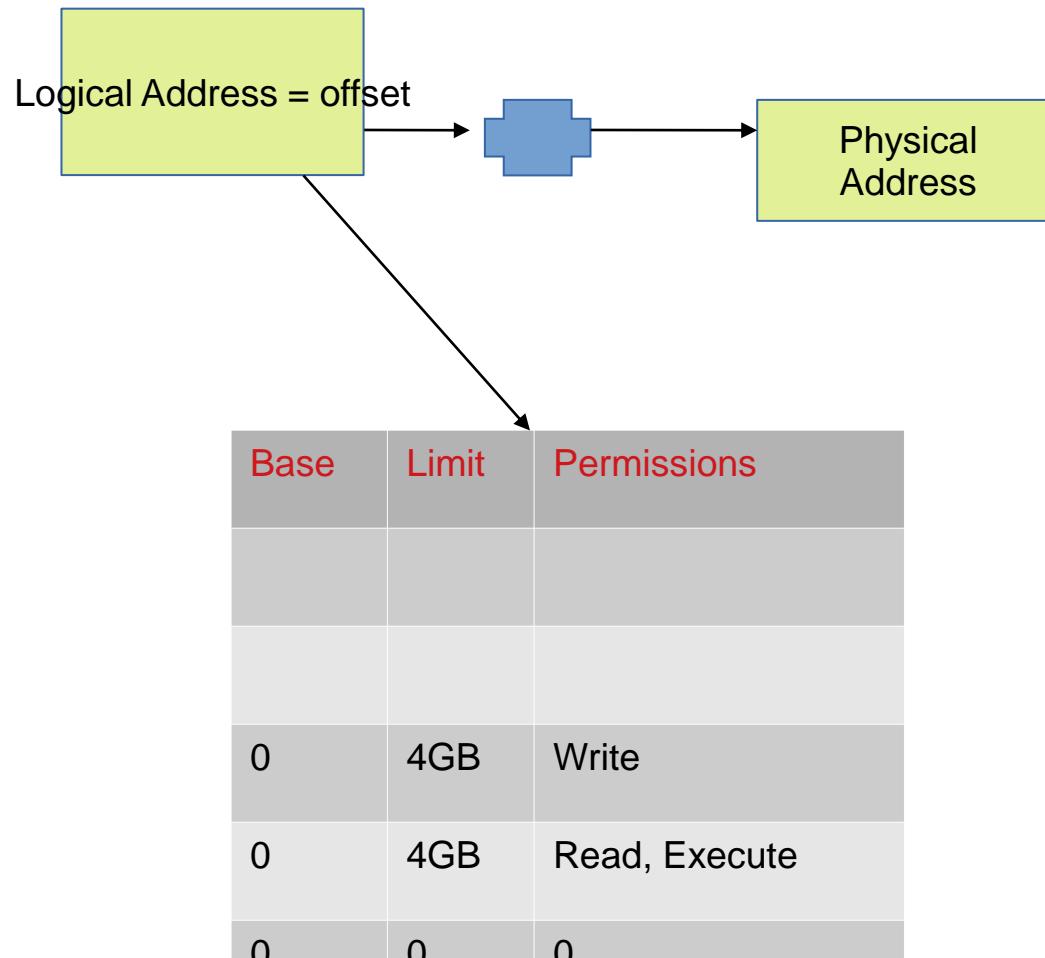
```
SEG_NULLASM # null  
seg
```

```
SEG ASM(STA X|STA
```

lgdt

- 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

**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
MP	Monitor coprocessor	NE	Numeric error
EM	Emulation	WP	Write protect
TS	Task switched	AM	Alignment mask
			NW Not write-through
			CD Cache disable
			PG Paging

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).

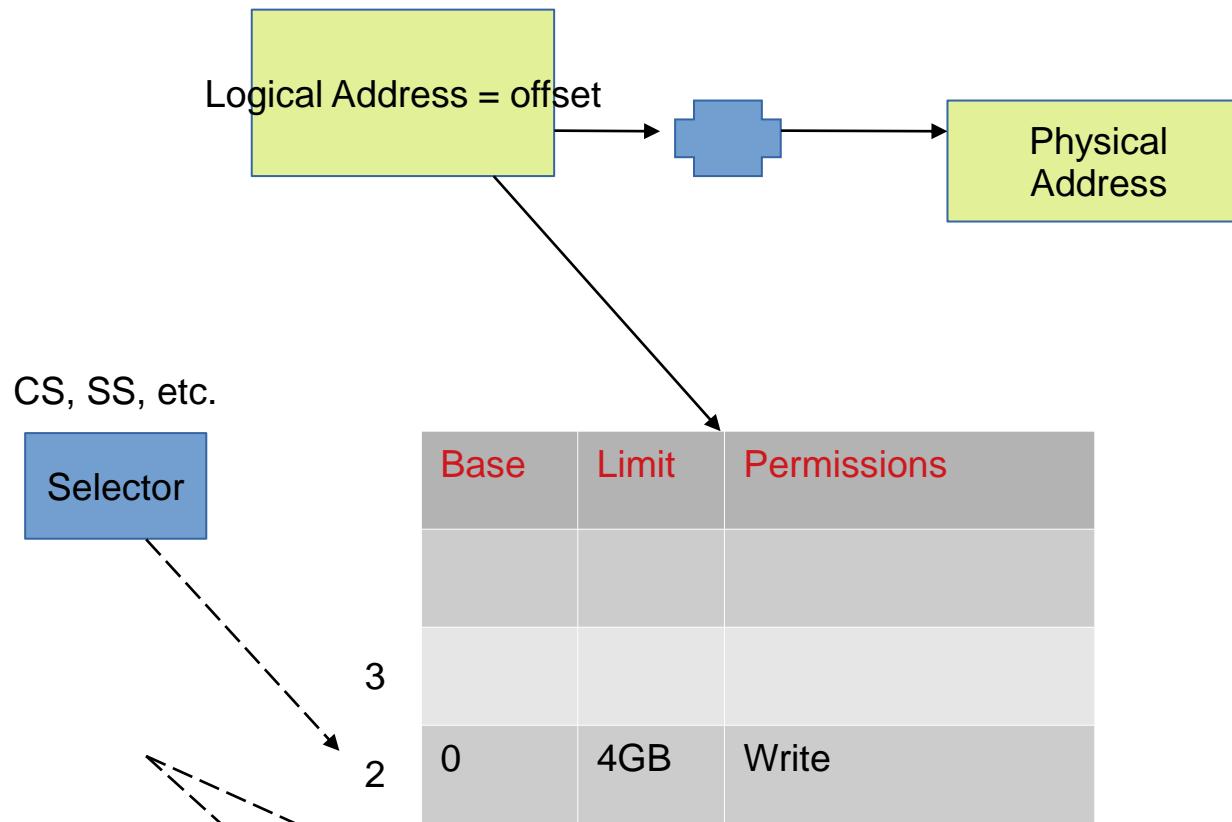
Note that ‘start32’ is the address of the

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
```

- 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

Setup now

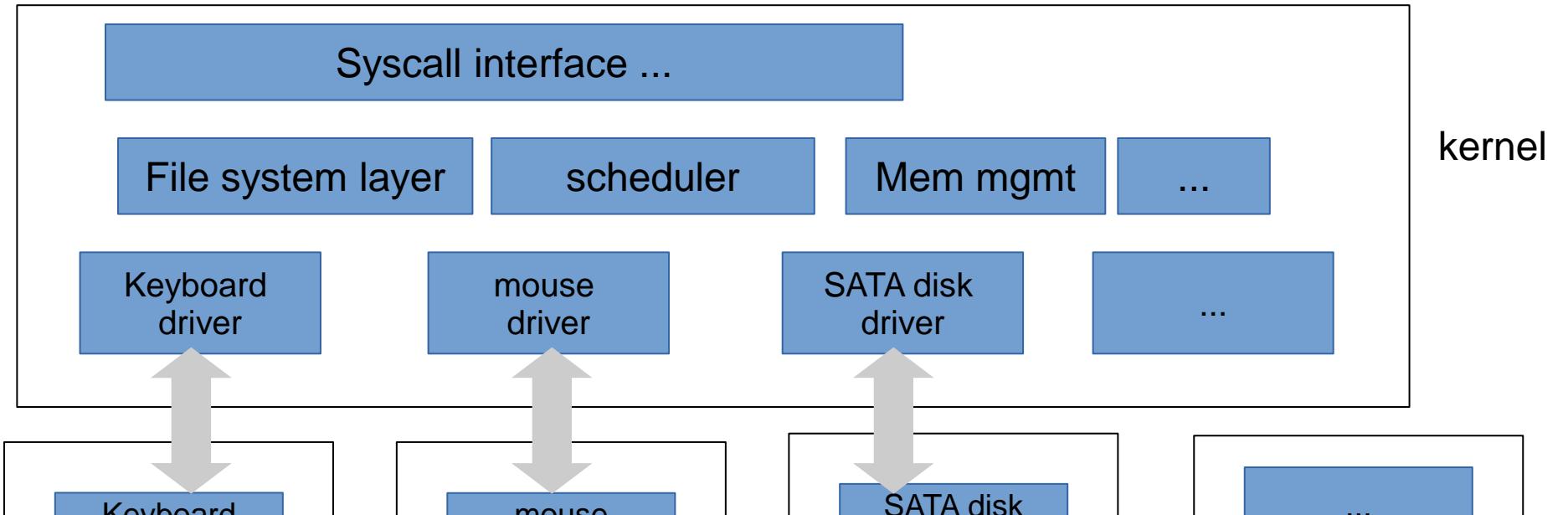


bootmain(): already in memory, as part of ‘bootblock’

- **bootmain.c , expects to find a copy of the kernel executable on the disk starting at the second sector (sector = 1).**
 - Why?

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

Drivers and Controllers

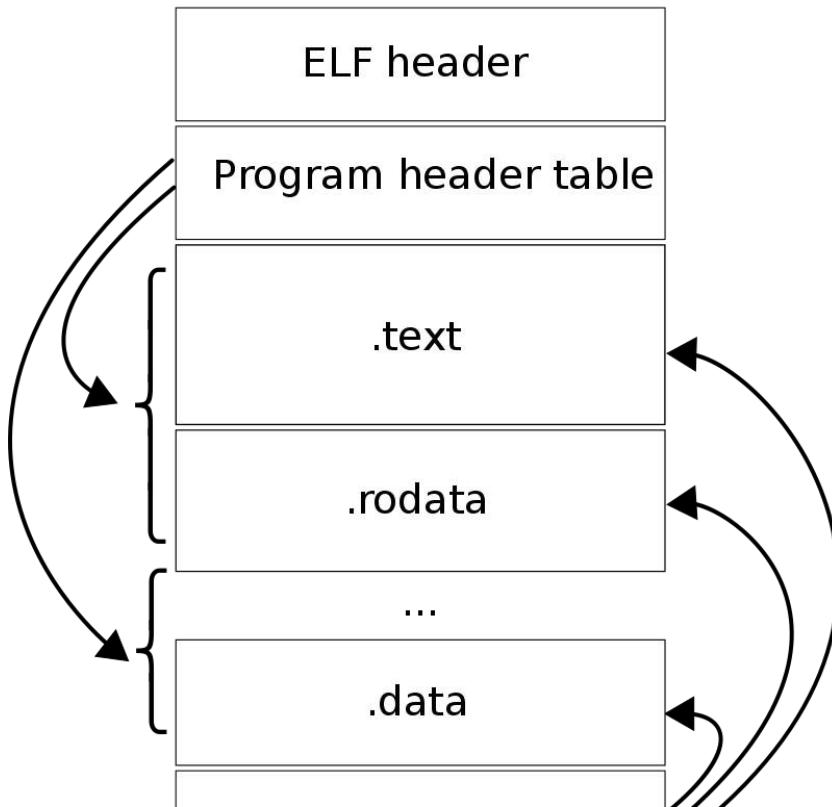


bootmain()

- Check if it's really ELF or not
- Next load kernel code from ELF file “kernel” into memory

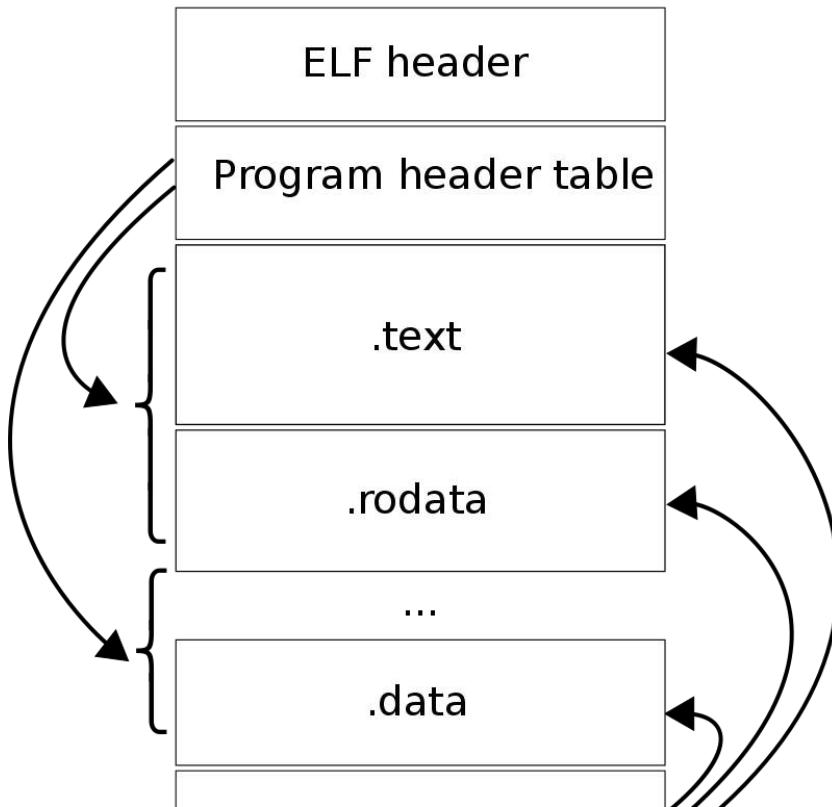
```
// Is this an ELF executable?  
if(elf->magic != ELF_MAGIC)  
    return; // let bootasm.S  
    handle error
```

ELF



```
struct elfhdr {  
    uint magic; // must  
    equal ELF_MAGIC  
    uchar elf[12];  
  
    ushort type;  
    ushort machine;  
    uint version;  
    uint entry;  
    uint phoff; // where is
```

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
```

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

```
kernel: file format elf32-i386  
kernel  
architecture: i386, flags 0x00000112:  
EXEC_P, HAS_SYMS, D_PAGED  
start address 0x0010000c
```

Program Header:

```
LOAD off 0x00001000 vaddr 0x80100000 paddr 0x00100000 align 2**12  
filesz 0x000a516 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

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)
```

Jump to Entry

```
// Call the entry point from the ELF  
header.
```

```
// Does not return!
```

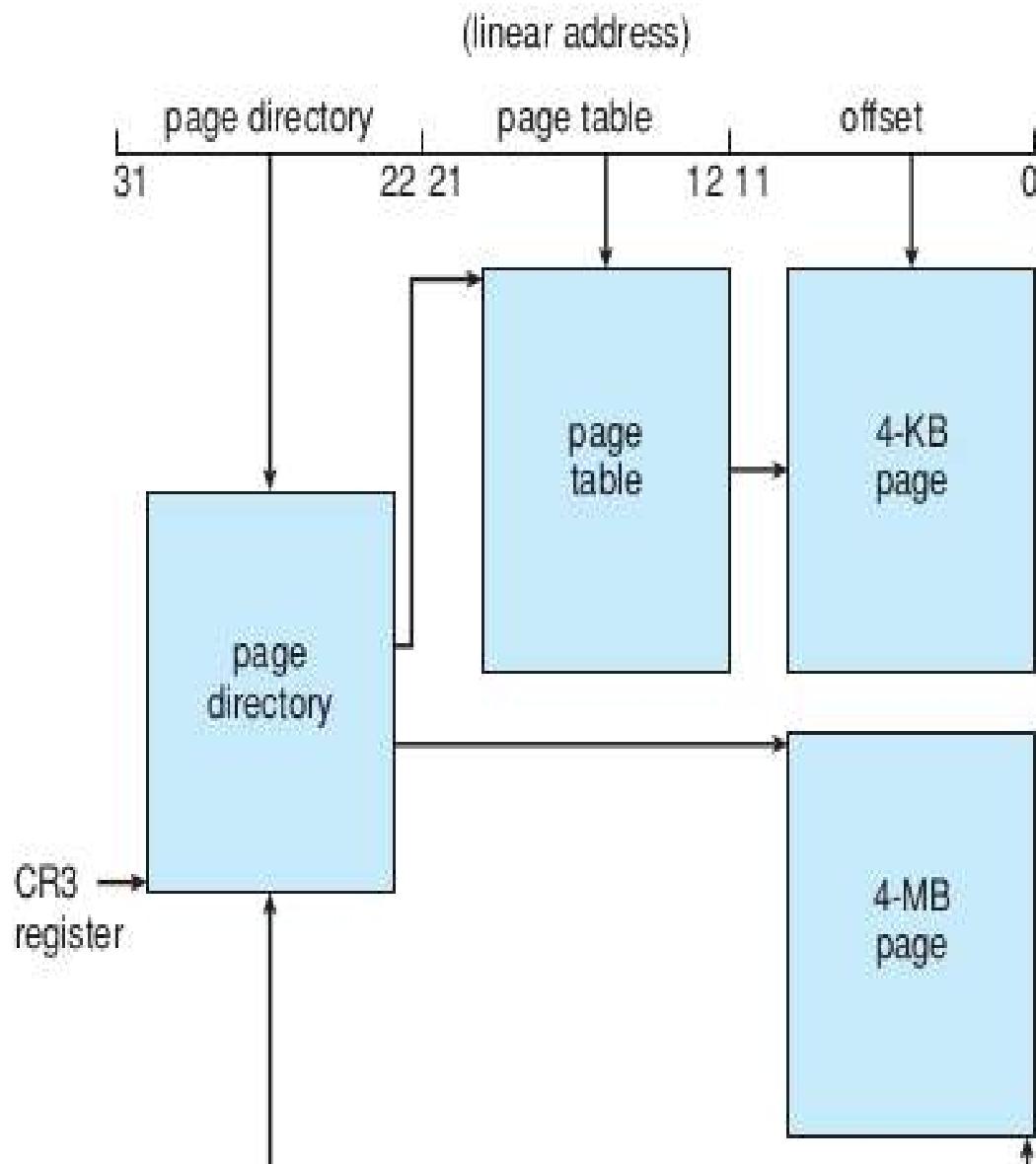
```
/* Abhijit:
```

```
* elf->entry was set by Linker using  
kernel.1d
```

```
* This is address 0x80100000
```

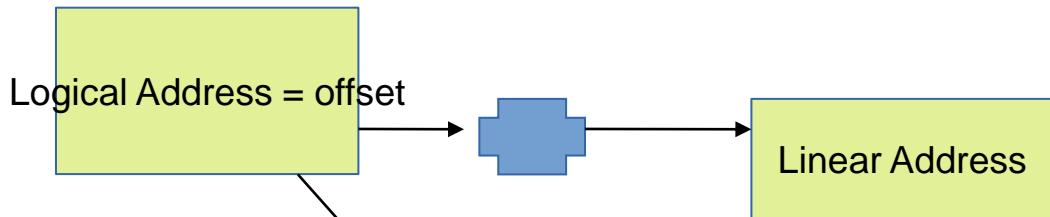
To understand
further
code

Remember: 4
MB pages
are possible



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

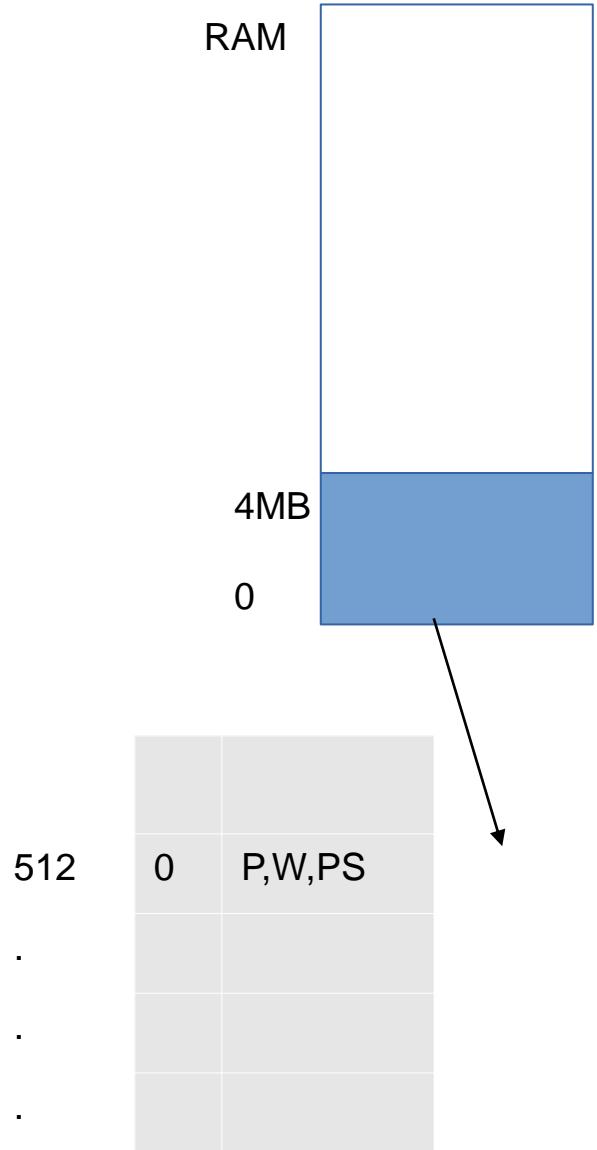
RAM



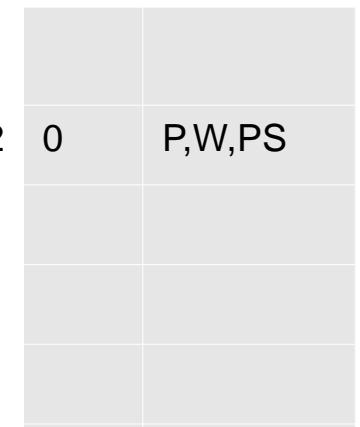
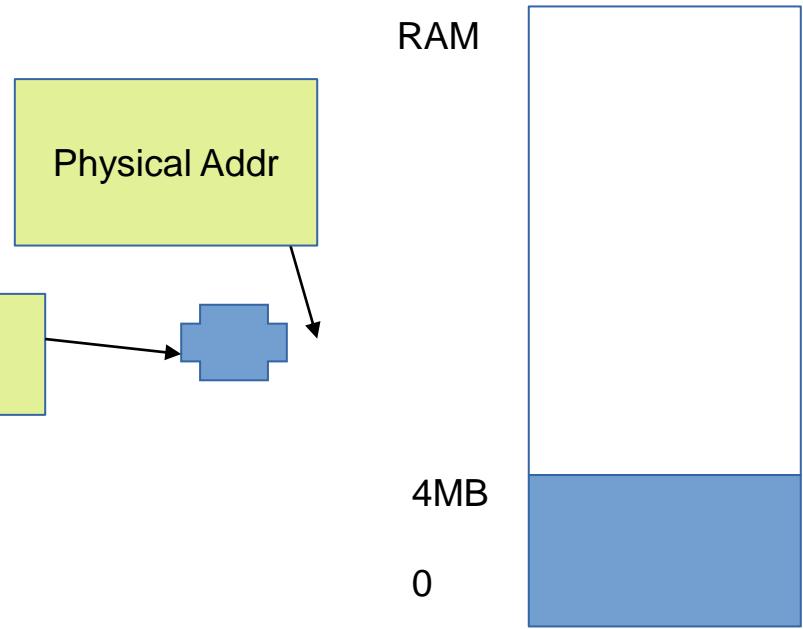
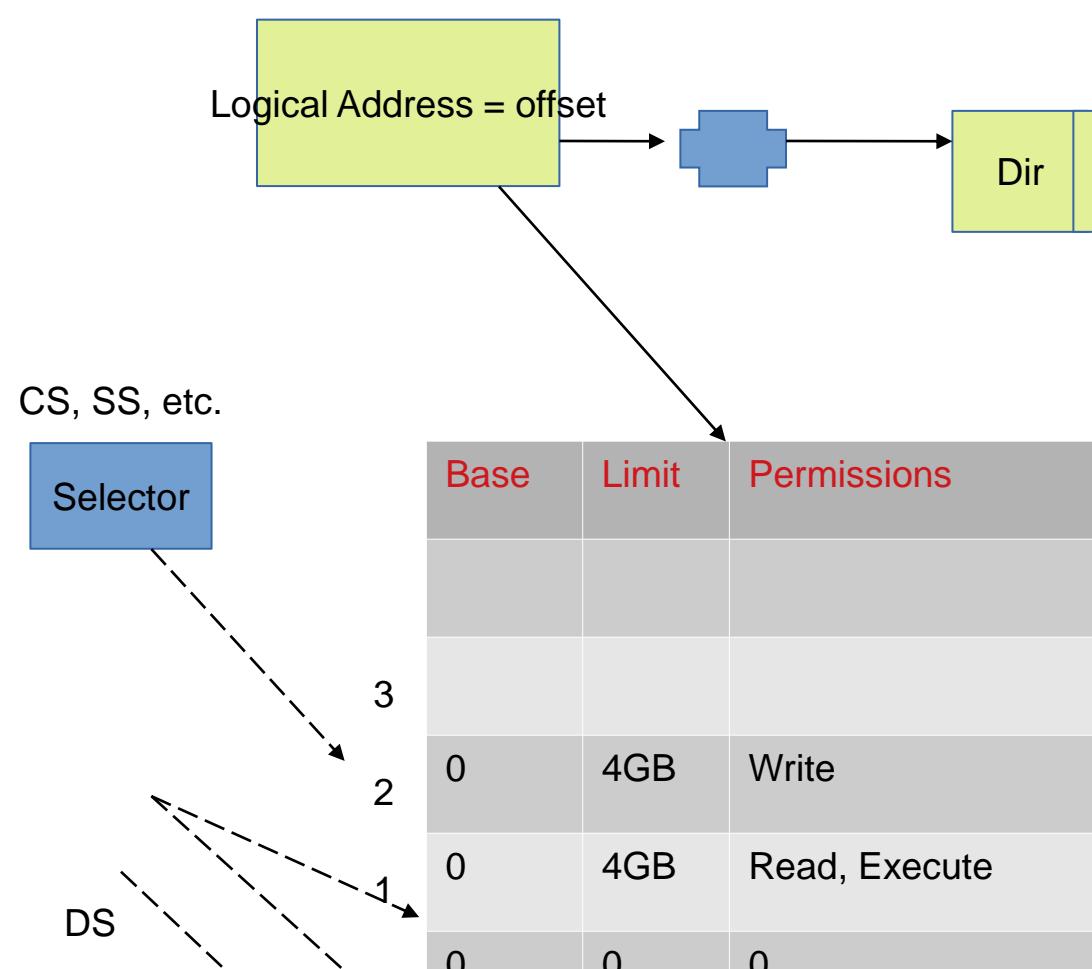
CS, SS, etc.



DS



**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,
};
```

```
#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
```

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

entry() in entry.S

entry:

```
movl %cr4, %eax
```

```
orl $(CR4_PSE), %eax
```

```
movl %eax, %cr4
```

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

- # Turn on page size extension for 4Mbyte pages
- # Set page directory. 4 MB pages (temporarily only. More later)

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

entry()

```
movl $(stack +  
KSTACKSIZE), %esp  
  
mov $main, %eax  
jmp *%eax  
  
.comm stack,  
KSTACKSIZE
```

- # Set up the stack pointer.
- # Abhijit:
+KSTACKSIZE is done as stack grows downwards
- # Jump to main(), stack grows downwards

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,**

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**
- **Initializes**
 - LAPIC on each processor, IOAPIC
 - Disables PIC
 - “Console” hardware (the standard I/O)
 - Serial Port
 - Ethernet Port

main() in main.c

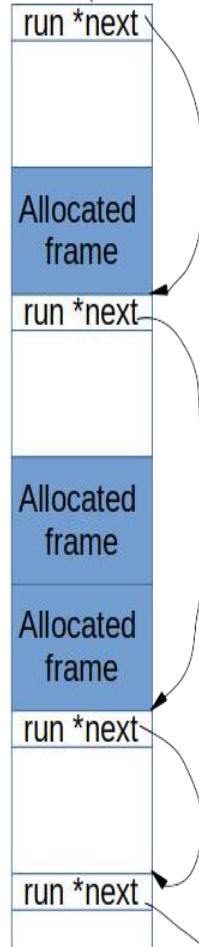
```
int  
  
main(void) {  
    kinit1(end, P2V(4*1024*1024));  
    // phys page allocator  
    kvmalloc(); // kernel page  
    table  
  
    void  
    kinit1(void *vstart, void  
    *vend) {  
        initlock(&kmem.lock,  
        "kmem");  
        kmem.use_lock = 0;
```

main() in main.c

```
void  
freerange(void *vstart, void *vend)  
{  
    char *p;  
    p =  
        (char*)PGROUNDUP((uint)vstart);  
    for(; p + PGSIZE <= (char*)vend; p  
        += PGSIZE)
```

```
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); }
```

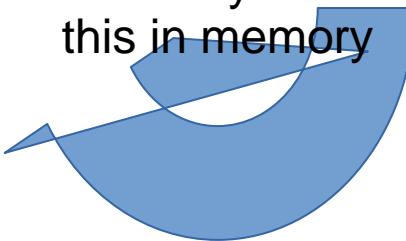
lock
uselock
run *freelist



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.

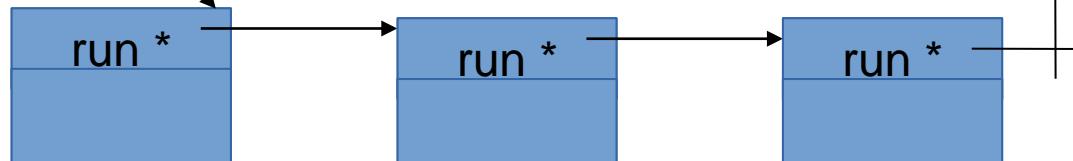
Actually like
this in memory



lock
uselock
run *freelist

kmem

Seen
independently



Back to main()

```
int  
main(void) {  
    kinit1(end,  
          P2V(4*1024*1024)); //  
          phys page allocator
```

// Allocate one page
table for the
machine for the
kernel address

// space for
scheduler
processes.

void

Back to main()

```
int  
main(void) {  
    kinit1(end,  
    P2V(4*1024*1024));  
    // phys page  
    allocator
```

// Allocate one page
table for the
machine for the
kernel address

// space for
scheduler
processes.

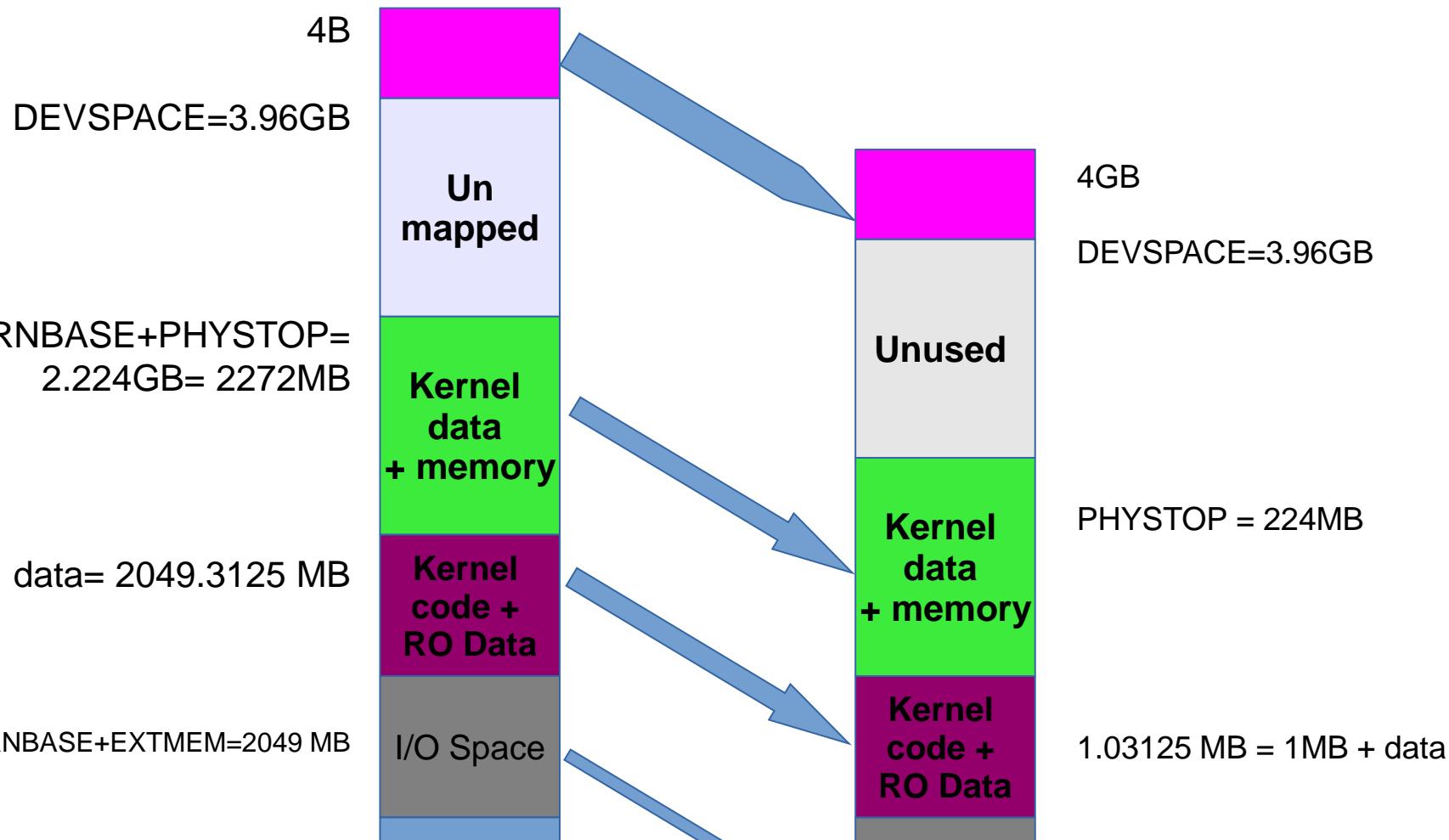
void

kymalloc(void)

```
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



Remidner: PDE and PTE entries

31

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 A C D W T U W P

PDE

P Present

w Writable

U User

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

CD Cache disabled

A Accessed

D Dirty

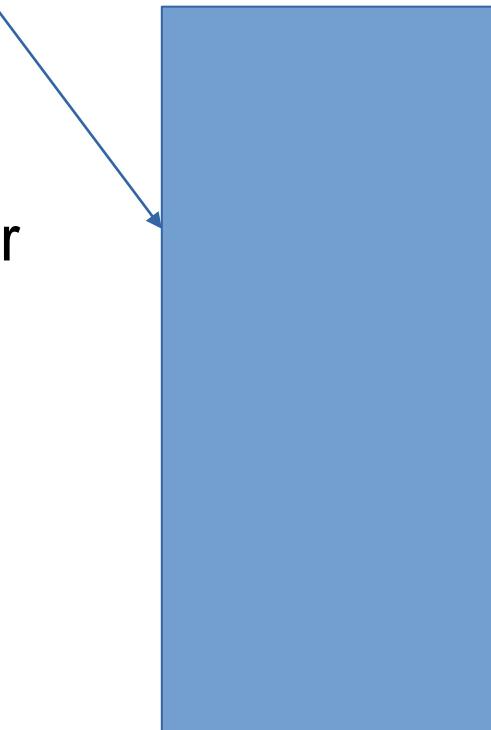
PS Page size (0=4KB, 1=4MB)

PAT Page table attribute index

31

Physical page number	A V	G A	P D	D A	C D	W T	U W	P
----------------------	--------	--------	--------	--------	--------	--------	--------	---

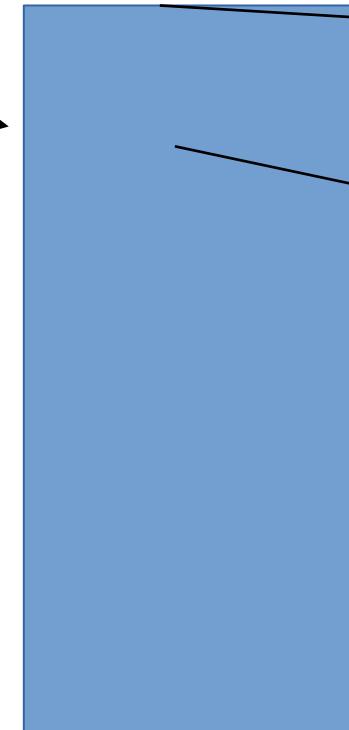
`pgdir`



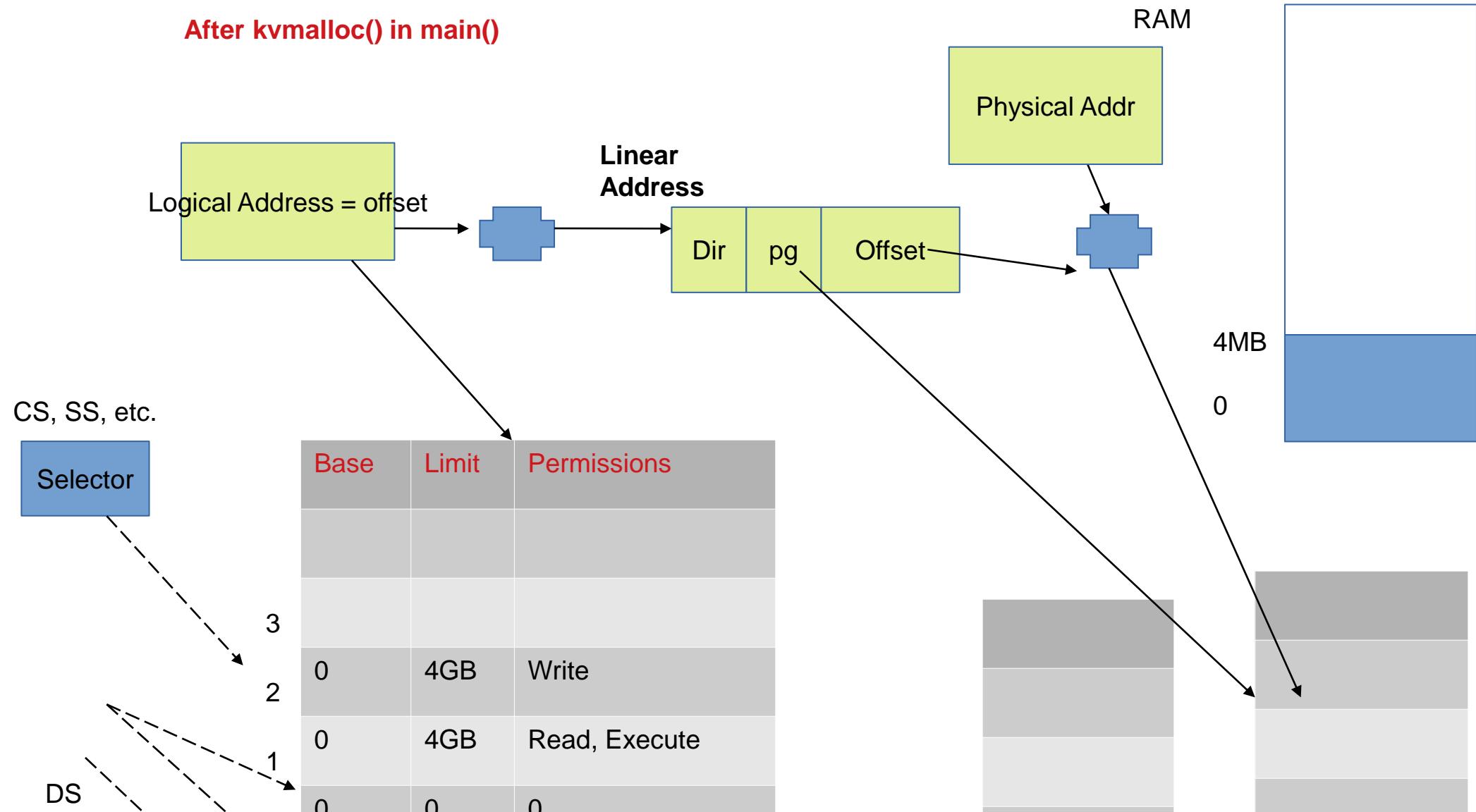
Before mappages()

pgdir

Page-no,UWP
Page-no,UWP
Page-no,UWP
Page-no,UWP
Page-no,UWP
Page-no,UWP
Page-no,UWP
Page-no,UWP
Page-no,UWP



After kvmalloc() in main()

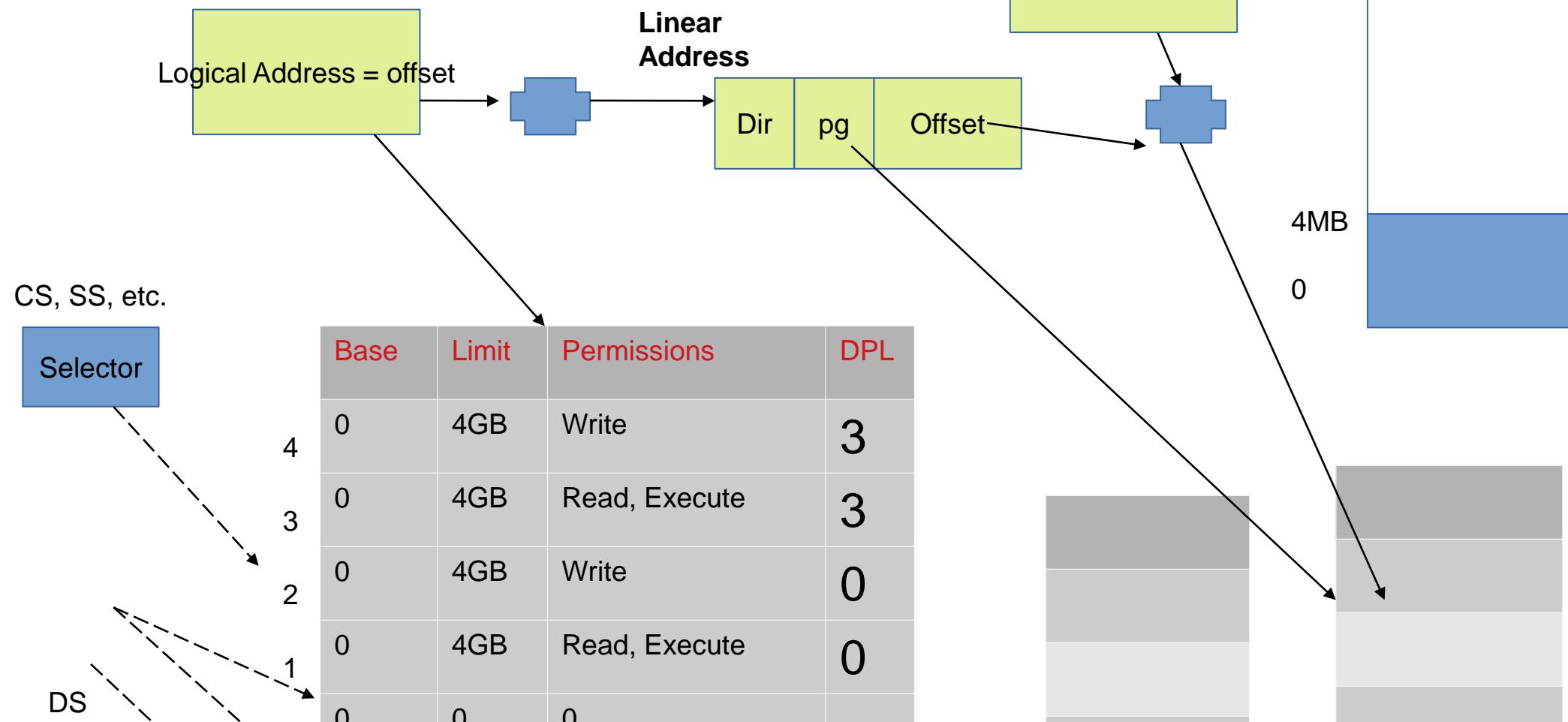


main() -> seginit()

- **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

Signals

Signals

- Can be termed as a way of Inter-process communication, but often it is not categorized as IPC (why!?)
- Processes can send each other “signals”, that is indicators of an event and receiver can “act” on the receipt of the signal
 - Integers are used to denote/signify each event

Let's see signals in action using Linux command line

□ Use of “kill” command

- Does not mean “kill” literally every time
- “kill” means send signal!

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 (asynchronous like segfault or synchronous like “kill” system call)

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 handle signals
- `sigaction()/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

Signals

□ Actions

- Term Default action is to terminate the process.
- Ign Default action is to ignore the signal.
- Core Default action is to terminate the process and dump core (see core(5)).
- Stop Default action is to stop the process.
- Cont Default action is to continue the process if it is currently stopped.

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

Signal handling by OS

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

OS: sys_signal {

Note down that process
12323 should 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

Sigaction: POSIX

```
#include <signal.h>

typedef void (*sighandler_t)(int);

sighandler_t signal(int signum, sighandler_t handler);
```

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

The “errno” And error conventions

Notes on reading xv6 code

Abhijit A. M.

abhijit.comp@coep.ac.in

Credits:

xv6 book by Cox, Kaashoek, Morris

Notes by Prof. Soray Bansal

Introduction to xv6

Structure of xv6 code

Compiling and executing xv6 code

About xv6

- **Unix Like OS**
- **Multi tasking, Single user**
- **On x86 processor**
- **Supports some system calls**
- **Small code, 7 to 10k**
- **Meant for learning OS concepts**

Use cscope and ctags with VIM

- Go to folder of xv6 code and run

```
cscope -q *. [chS]
```

- Also run

```
ctags *. [chS]
```

- Now download the file

http://cscope.sourceforge.net/cscope_maps.vim ascope_maps.vim in your ~ folder

Use call graphs (using doxygen)

- Doxygen – a documentation generator.
- Can also be used to generate “call graphs” of functions
- Download xv6
- Install doxygen on your Ubuntu machine.
- cd to xv6 folder

Use call graphs (using doxygen)

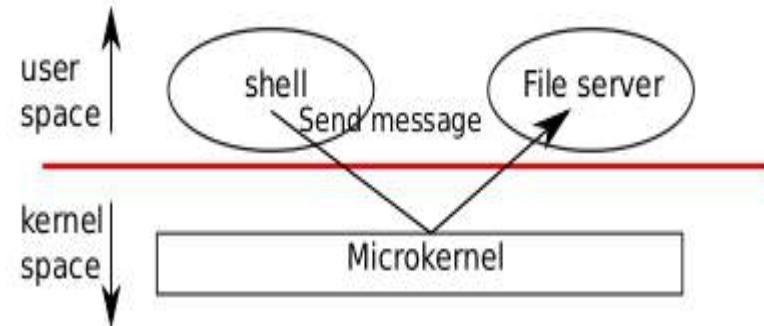
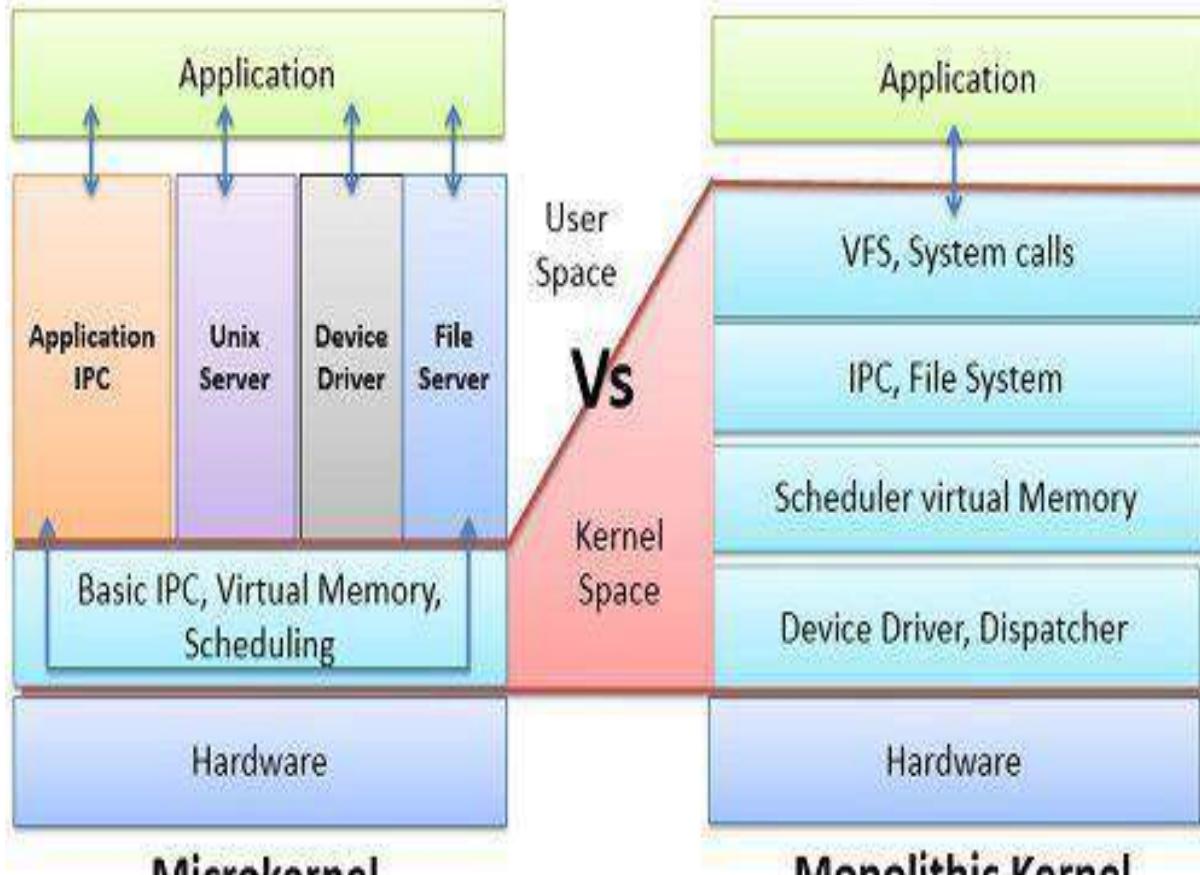
- Create a folder “doxygen”
- Open “doxyconfig” file and make these changes.

```
PROJECT_NAME = "XV6"
```

```
OUTPUT_DIRECTORY = ./doxygen
```

```
CREATE_SUBDIRS = YES
```

Xv6 follows monolithic kernel approach



qemu

- A virtual machine manager, like Virtualbox
- Qemu provides us
 - BIOS
 - Virtual CPU, RAM, Disk controller, Keyboard controller
 - IOAPIC, LAPIC
- Qemu runs xv6 using this command

qemu

□ Understanding qemu command

- -serial mon:stdio
 - the window of xv6 is also multiplexed in your normal terminal.
 - Run “make qemu”, then Press “Ctrl-a” and “c” in terminal and you get qemu prompt
- -drive file=fs.img,index=1,media=disk,format=raw
 - Specify the hard disk in “fs.img”, accessible at first slot in IDE(=SATA) controller “blk1” with “index=1”

About files in XV6 code

- **cat.c echo.c forktest.c grep.c
init.c kill.c ln.c ls.c mkdir.c
rm.c sh.c stressfs.c usertests.c
wc.c yes.c zombie.c**
 - User programs for testing xv6
- **Makefile**
 - To compile the code

About files in XV6 code

- **bootasm.S entryother.S entry.S
initcode.S swtch.S trapasm.S usys.S**
 - Kernel code written in Assembly. Total 373 lines
- **kernel.ld**
 - Instructions to Linker, for linking the kernel properly
- **README Notes LICENSE**
 - Misc files

Using Makefile

- **make qemu**
 - Compile code and run using “qemu” emulator
- **make xv6.pdf**
 - Generate a PDF of xv6 code
- **make mkfs**
 - Create the mkfs program
- **make clean**

Files generated by Makefile

- **.o files**
 - Compiled from each .c file
 - No need of separate instruction in Makefile to create .o files
 - _%: %.o \$(ULIB) line is sufficient to build each .o for a _xyz file

Files generated by Makefile

- **asm files**
 - Each of them has an equivalent object code file or C file. For example
 - bootblock: bootasm.S bootmain.c
 - \$(CC) \$(CFLAGS) -fno-pic -O -nostdinc -I. -c bootmain.c
 - \$(CC) \$(CFLAGS) -fno-pic -nostdinc -I. -c bootasm.S

Files generated by Makefile

- **_ln, _ls, etc**
 - Executable user programs
 - Compilation process is explained after few slides

Files generated by Makefile

- **xv6.img**
 - Image of xv6 created
 - xv6.img: bootblock kernel
 - dd if=/dev/zero of=xv6.img count=10000
 - dd if=bootblock of=xv6.img conv=notrunc
 - dd if=kernel of=xv6.img seek=1

Files generated by Makefile

- bootblock

```
bootblock: bootasm.S bootmain.c
```

```
$ (CC) $ (CFLAGS) -fno-pic -O -  
nostdinc -I. -c bootmain.c
```

```
$ (CC) $ (CFLAGS) -fno-pic -nostdinc -  
I. -c bootasm.S
```

```
$ (LD) $ (LDFLAGS) -N -e start -Ttext
```

Files generated by Makefile

kernel

Files generated by Makefile

- **fs.img**
 - A disk image containing user programs and README
 - `fs.img: mkfs README $(UPROGS)`
 - `./mkfs fs.img README $(UPROGS)`
- **.sym files**
 - Symbol tables of different programs

Size of xv6 C code

- **wc *[ch] | sort -n**
 - 10595 34249 278455 total
 - Out of which
 - 738 4271 33514 dot-bochssrc
- **wc cat.c echo.c forktest.c grep.c init.c kill.c
ln.c ls.c mkdir.c rm.c sh.c stressfs.c
usertests.c wc.c yes.c zombie.c**

List of commands to try (in given order)

usertests # Runs lot of tests and takes upto 10 minutes to run

stressfs # opens , reads and writes to files in parallel

ls # out put is filetyp, inode number, type

cat README

ls -ls

List of commands to try (in this order)

`echo README | grep Wa`

`echo README | grep Wa | grep ty # does not work`

`cat README | grep Wa | grep bl # works`

`ls .. # works from inside test`

`cd # fails`

`cd / # works`

`wc README`

`rm out`

User Libraries: Used to link user land programs

- **Ulib.c**
 - Strcpy, strcmp,strlen, memset, strchr, stat, atoi, memmove
 - Stat uses open()
- **Usys.S -> compiles into usys.o**
 - Assembly code file. Basically converts all calls like open() (e.g. used in ulib.c) into assembly code using “int” instruction.
 - Run following command see the last 4 lines in the output

objdump -d usys.o

00000048 <open>:

User Libraries: Used to link user land programs

- **printf.c**
 - Code for printf()!
 - Interesting to read this code.
 - Uses variable number of arguments. Normal technique in C is to use va_args library, but here it uses pointer arithmetic.
 - Written using two more functions: printint() and putc() - both call write()

User Libraries: Used to link user land programs

- **umalloc.c**
 - This is an implementation of malloc() and free()
 - Almost same as the one done in “The C Programming Language” by Kernighan and Ritchie
 - Uses sbrk() to get more memory from xv6 kernel

Understanding the build process in more details

- Run

make qemu | tee make-output.txt

- You will get all compilation commands in
make-output.txt

Compiling user land programs

Normally when you compile a program on Linux

You compile it for the same ‘target’ machine (= CPU + OS)

The compiler itself runs on the same OS

To compile a user land program for xv6, we don’t have a compiler on xv6,

So we compile the programs (using make, cc) on Linux , for xv6

Obviously they can’t link with the standard libraries on Linux

Compiling user land programs

```
ULIB = ulib.o usys.o printf.o umalloc.o

_%: %.o $(ULIB)

$(LD) $(LDFLAGS) -N -e main -Ttext 0 -o $@ $^

$(OBJDUMP) -S $@ > $*.asm

$(OBJDUMP) -t $@ | sed '1,/SYMBOL TABLE/d; s/ .* / /;
/^$$/d' > $*.sym
```

\$@ is the name of the file being generated

\$^ is dependencies . i.e. \$(ULIB) and %.o in this case

Compiling user land programs

```
gcc -fno-pic -static -fno-builtin -fno-strict-
aliasing -O2 -Wall -MD -ggdb -m32 -Werror -fno-omit-
frame-pointer -fno-stack-protector -fno-pie -no-pie -
c -o cat.o cat.c
```

```
ld -m elf_i386 -N -e main -Ttext 0 -o _cat cat.o
ulib.o usys.o printf.o umalloc.o
```

```
objdump -S _cat > cat.asm
```

```
objdump -t _cat | sed '1,/SYMBOL TABLE/d; s/ .* / /;
/^$/d' > cat.sym
```

Compiling user land programs

Mkfs is compiled like a Linux program !

gcc -Werror -Wall -o mkfs mkfs.c

How to read kernel code ?

- **Understand the data structures**
 - Know each global variable, typedefs, lists, arrays, etc.
 - Know the purpose of each of them
- **While reading a code path, e.g. exec()**
 - Try to ‘locate’ the key line of code that does major work

Pre-requisites for reading the code

- **Understanding of core concepts of operating systems**
 - Memory Management, processes, fork-exec, file systems, synchronization, x86 architecture, calling convention , computer organization
- **2 approaches:**
 - 1) **Read OS basics first, and then start reading code**

Threads

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

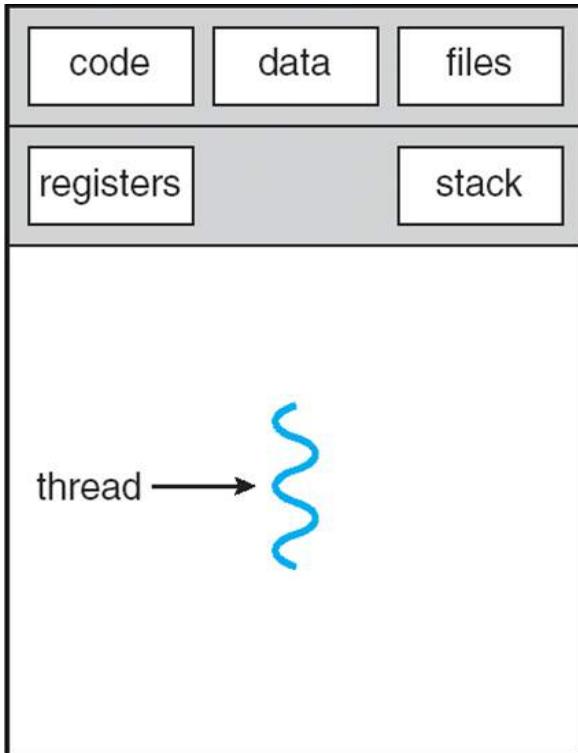
Threads

- **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**

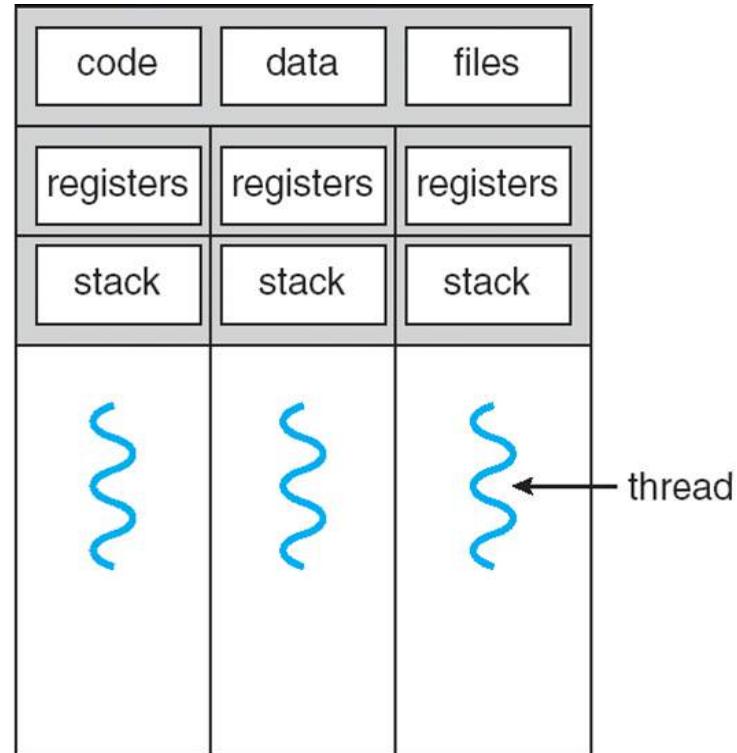
Threads

- **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

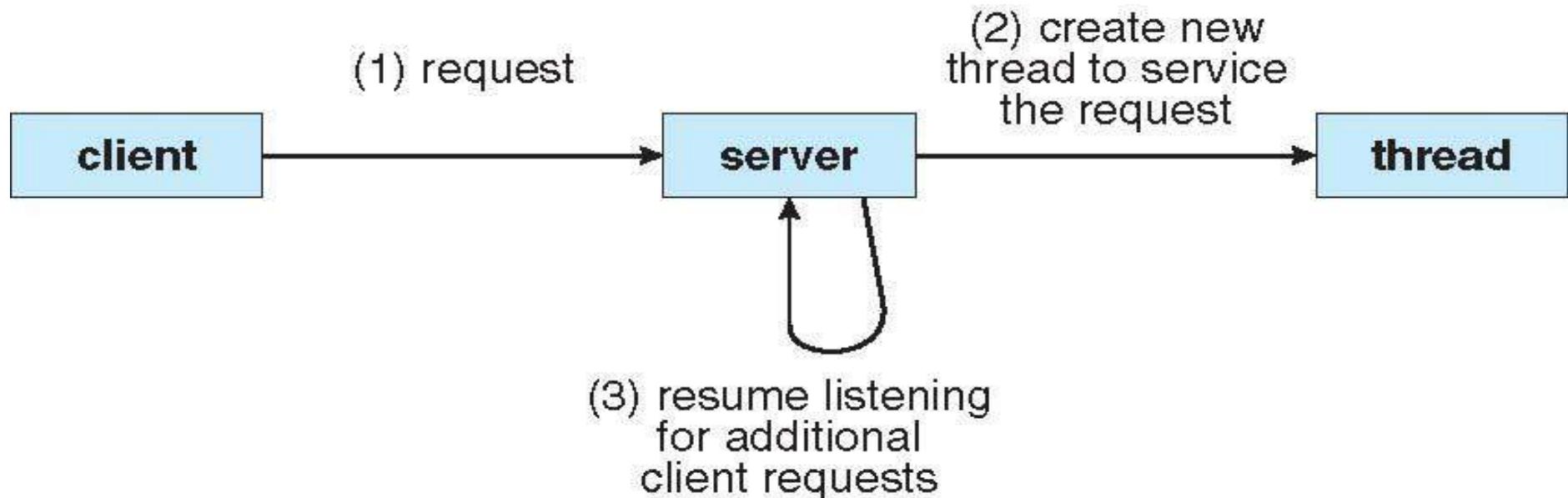


single-threaded process



multithreaded process

A multithreaded server

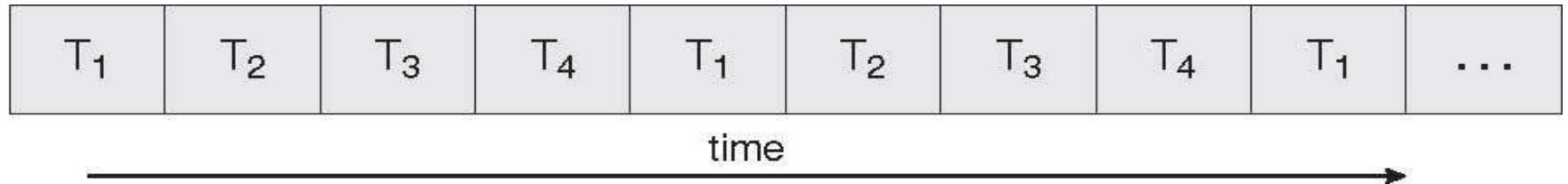


Benefits of threads

- **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
- **Kernel Threads:**
Supported by the
Kernel
- **Examples**
 - Windows XP/2000
 - Solaris
 - Linux
 - Tru64 UNIX

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

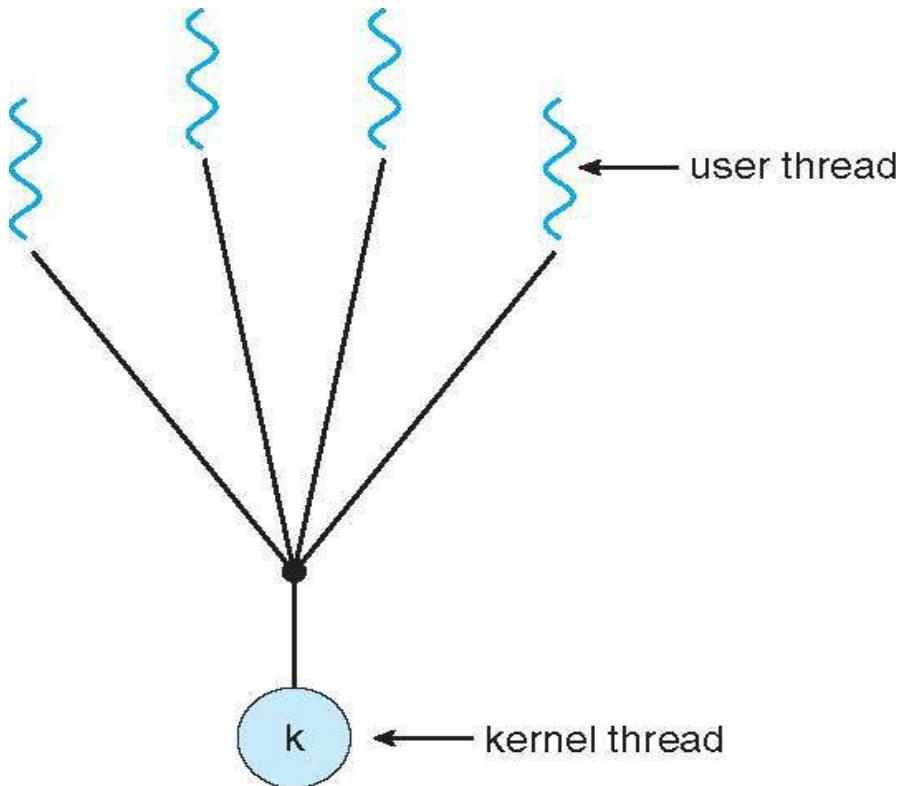
□ Kernel Threads

- 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

Multithreading models

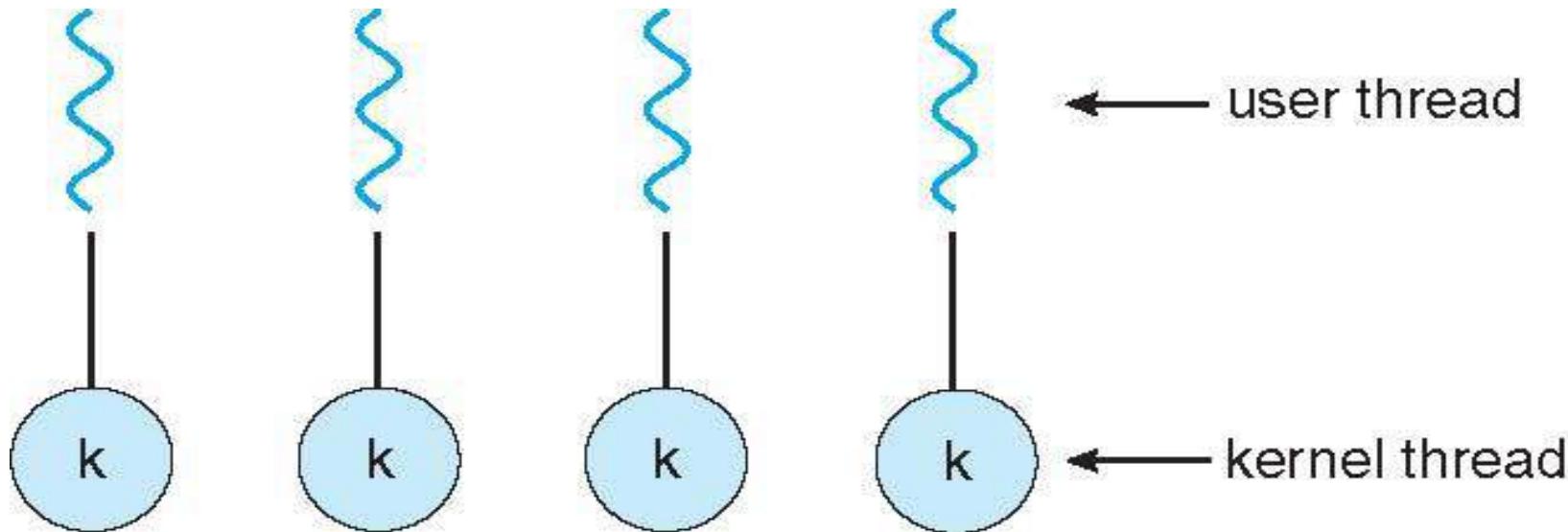
- **How to map user threads to kernel threads?**
 - Many-to-One
 - One-to-One
 - Many-to-Many
- **What if there are no kernel threads?**
 - Then only “one” process. Hence many-one mapping possible, to be done by user level thread library

Many-One Model



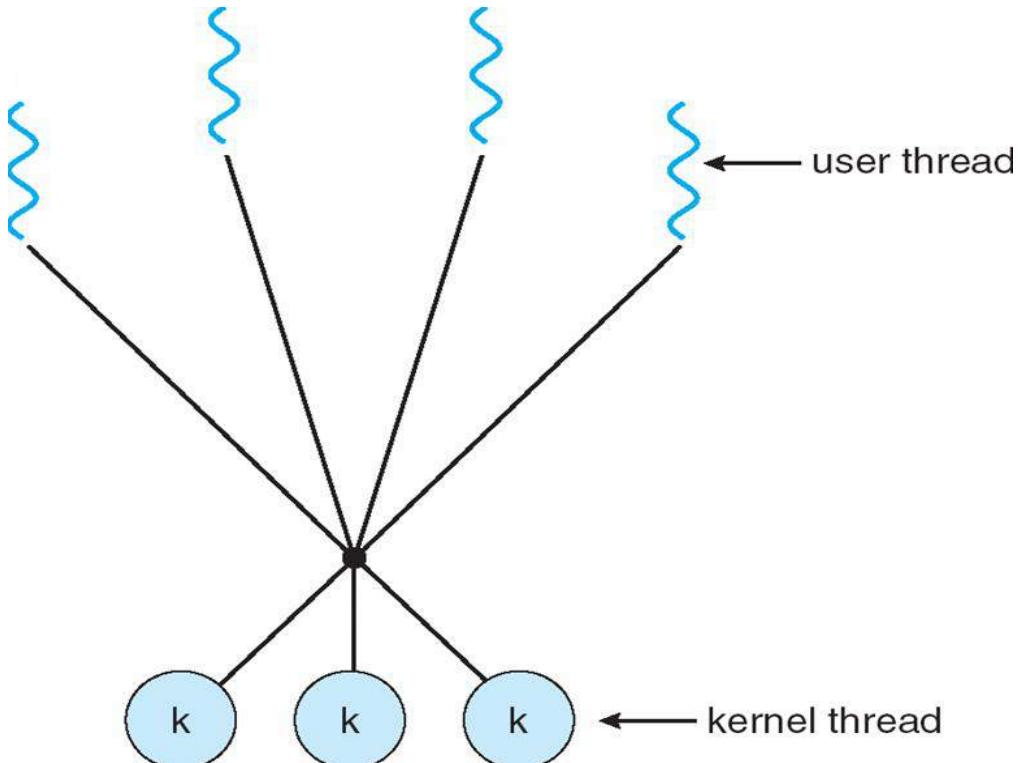
- **Many user-level threads mapped to single kernel thread**
- **Examples:**
 - Solaris Green Threads
 - GNU Portable Threads

One-One Model



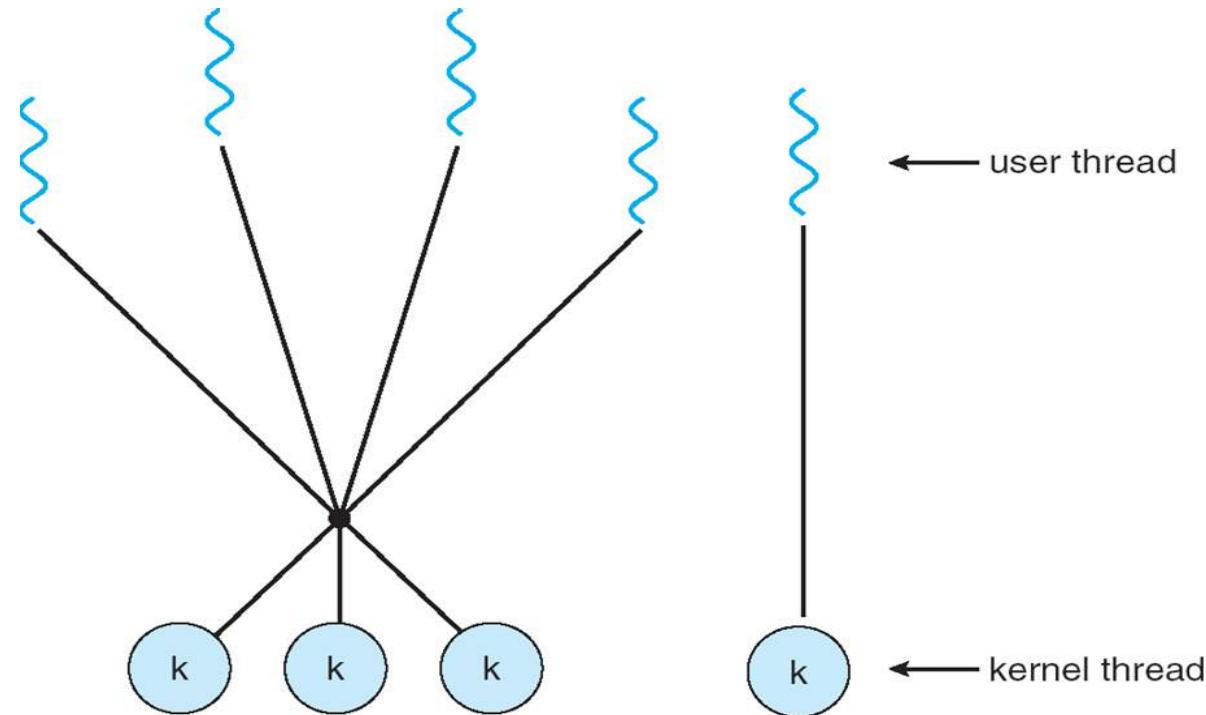
- **Each user-level thread maps to kernel thread**
- **Examples**

Many-Many Model



- **Allows many user level threads to be mapped to many kernel threads**
- **Allows the operating system to**

Two Level Model



- Similar to M:M, except that it allows a user thread to be bound to kernel thread
- Examples
 - IRIX

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

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.

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 bounded 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

```
int arr[16];
```

```
int f() {
```

```
    a(); b(); c();
```

```
}
```

```
int g() {
```

```
    x(); y();
```

```
}
```

```
int h() {
```

Thread Local Storage (TLS) in pthreads

□ Functions

- `pthread_key_create`
- `pthread_key_delete`
- `pthread_setspecific`
- `pthread_getspecific`

□ See

- `thrd_specific.c` file

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

Scheduler activations for threads

Library

```
upcall_handler() {
```

Create one more
LWP and schedule
threads on this;
}

application

```
f() {
```

```
scanf();
```

```
}
```

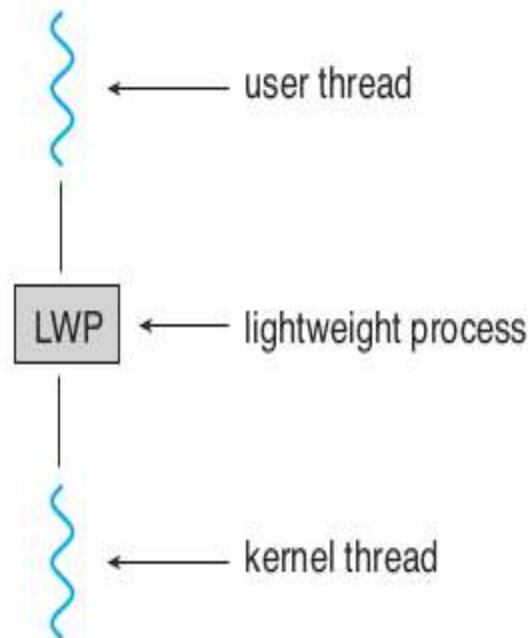
```
g() {
```

```
recv();
```

Kernel

```
register_upcall(function f) {  
    proc->upcall = f;  
}  
  
sys_write() {  
    // before calling sleep() going  
    // to block  
    myproc()->upcall(); // tricky!  
}
```

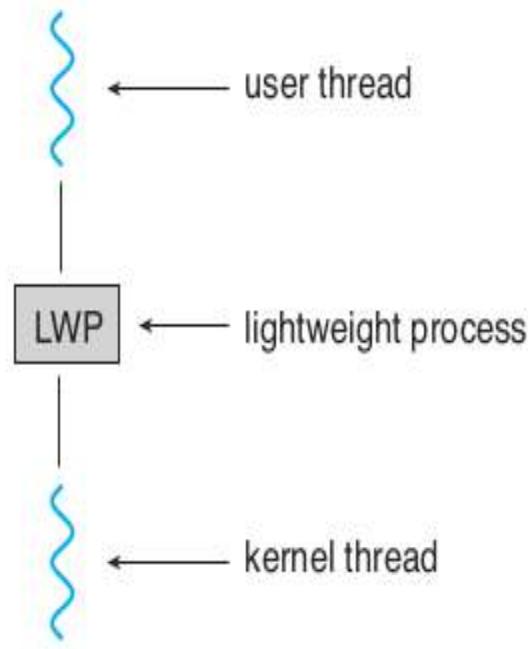
Issues with threads



- **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.

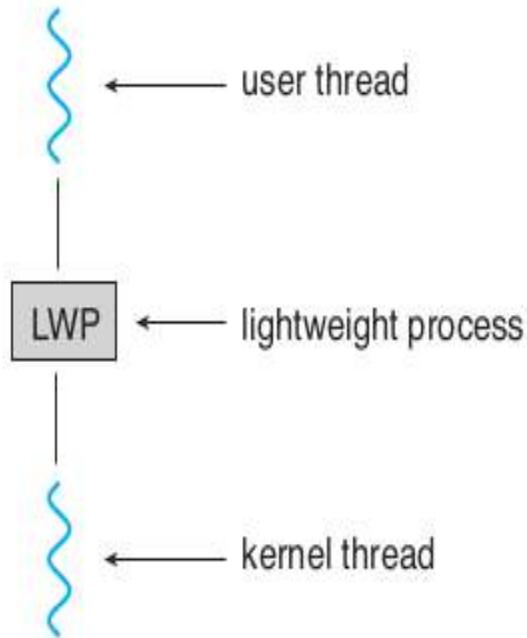
Issues with threads



- **Scheduler Activations:
LWP approach**

- Kernel needs to inform application about events like: a thread is about to block, or wait is over : this is ‘upcall’
- This will help application relinquish the LWP or

Issues with threads



- **Example NetBSD**
 - “An Implementation of Scheduler Activations on the NetBSD Operating System”
 - <https://web.mit.edu/nathanael/www/usenix/freenix-sa/freenix-sa.html>

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)**

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()

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

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

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

Signal handling by OS

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

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

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

OS: sys_kill {
Note down in PCB of
process 12323 that
signal number 19 is
pending for you

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

Synchronization

My formulation

- **OS = data structures + synchronization**
- **Synchronization problems make writing OS code challenging**
- **Demand exceptional coding skills**

Race problem

```
long c = 0, c1 = 0, c2 =    int main() {  
0, run = 1;  
  
void *thread1(void  
*arg) {  
  
while(run == 1) {  
    c++;  
  
    c1++;  
    c2++;  
  
    if(c1 > c2) {  
        //printf("c1: %d, c2: %d\n", c1, c2);  
    }  
}  
}  
  
pthread_t th1, th2;  
pthread_create(&th1,  
NULL, thread1, NULL);  
pthread_create(&th2,  
NULL, thread2, NULL);  
  
//fprintf(stderr, "Final value of c: %d\n", c);  
}
```

Race problem

- On earlier slide
 - Value of c should be equal to $c_1 + c_2$, but it is not!
 - Why?
- There is a “race” between thread1 and thread2 for updating the variable c
- thread1 and thread2 may get scheduled in any order and *interrupted* any point in time

Race problem

- C++, when converted to assembly code, could be

mov c, r1

add r1, 1

mov r1, c

- Now following sequence of instructions is possible among thread1 and thread2

Races: reasons

- **Interruptible kernel**

- If entry to kernel code does not disable interrupts, then modifications to any kernel data structure can be left incomplete
- This introduces concurrency

- **Multiprocessor systems**

- On SMP systems: memory is shared, kernel and process code run on all processors

Critical Section problem

```
do {  
    entry section  
    critical section  
    exit section  
    remainder section  
} while (TRUE);
```

Critical Section Problem

- Consider system of n processes {p₀, p₁, ..., p_{n-1}}
- Each process has critical section segment of code
 - Process may be changing common variables, updating table, writing file, etc
 - When one process in critical section, no other may be in its critical section

Expected solution characteristics

- **1. Mutual Exclusion**

- If process P_i is executing in its critical section, then no other processes can be executing in their critical sections

- **2. Progress**

- If no process is executing in its critical section and there exist some processes that wish to enter their critical section, then the selection of the processes

suggested solution - 1

```
int flag = 1;
```

```
void *thread1(void  
*arg) {
```

```
while(run == 1) {
```

```
while(flag == 0)
```

```
;
```

- What's wrong here?
- Assumes that
while(flag ==) ; flag = 0
- will be atomic

suggested solution - 2

```
int flag = 0;  
void *thread1(void  
*arg) {  
while(run == 1) {  
if(flag)  
c++;
```

```
void *thread2(void  
*arg) {  
while(run == 1) {  
if(!flag)  
c++;  
else
```

Peterson's solution

- Two process solution
- Assume that the LOAD and STORE instructions are atomic; that is, cannot be interrupted
- The two processes share two variables:
`int turn;`

Boolean flag[2]

Peterson's solution

- **do {**
- **flag[i] = TRUE;**
- **turn = j;**
- **while (flag[j] && turn == j)**
- **;**
- **critical section**
- **flag[i] = FALSE;**
- **remainder section**

Hardware solution – the one actually implemented

- **Many systems provide hardware support for critical section code**
- **Uniprocessors – could disable interrupts**
 - Currently running code would execute without preemption
 - Generally too inefficient on multiprocessor systems
 - Operating systems using this not broadly scalable

Solution using test-and-set

```
lock = false; //global
```

```
do {  
    while ( TestAndSet (&lock ) )  
    ; // do nothing  
    // critical section  
    lock = FALSE;  
    // remainder section  
} while (TRUE);
```

Definition:

```
boolean TestAndSet (boolean  
*target)
```

```
{  
    boolean rv = *target;  
    *target = TRUE;  
    return rv;  
}
```

Solution using swap

```
lock = false; //global
```

```
do {  
    key = true  
    while ( key == true))  
        swap(&lock, &key)  
        // critical section  
        lock = FALSE;  
        // remainder section  
    } while (TRUE);
```

Spinlock

- A lock implemented to do ‘busy-wait’
- Using instructions like T&S or Swap
- As shown on earlier slides
- `spinlock(int *lock){`

`While(test-and-set(lock))`

`};`

Bounded wait M.E. with T&S

```
do {  
    waiting[i] = TRUE;  
    key = TRUE;  
    while (waiting[i] && key)  
        key = TestAndSet(&lock);  
    waiting[i] = FALSE;  
    // critical section  
    j = (i + 1) % n;  
    while ((j != i) && !waiting[j])  
        j = (j + 1) % n;  
    if (j == i)  
        lock = FALSE;
```

sleep-locks

- **Spin locks result in busy-wait**
- **CPU cycles wasted by waiting processes/threads**
- **Solution – threads keep waiting for the lock to be available**
 - Move thread to wait queue
 - The thread holding the lock will wake up one of

Sleep locks/mutexes

```
//ignore syntactical  
issues  
typedef struct mutex  
{  
wait(mutex *m) {  
}  
  
Block(mutex *m,  
spinlock *sl) {  
  
}  
  
release(mutex *m) {
```

Some thumb-rules of spinlocks

- ❑ Never block a process holding a spinlock !
- ❑ Typical code:

```
while(condition)
```

```
{ Spin-unlock()
```

```
Schedule()
```

```
Spin-lock()
```

Locks in xv6 code

struct spinlock

// Mutual exclusion lock.

struct spinlock {

uint locked; // Is the lock held?

// For debugging:

char *name; // Name of lock.

spinlocks in xv6 code

```
struct {
    struct spinlock lock;
    struct buf buf[NBUF];
    struct buf head;
} bcache;

struct {
    struct spinlock lock;
    static struct spinlock idelock;
    struct {
        struct spinlock lock;
        int use_lock;
        struct run *freelist;
    } kmem;
    struct log {
```

```
static inline uint
xchg(volatile uint
*addr, uint newval)
{
    uint result;
    // The + in "+m"
    // denotes a read-
    // modify-write operand.
    asm volatile("lock;
    xchgl %0,%1"
        : "+m"(result), "=m"(newval)
        : "0"(newval));
    return result;
}
```

Spinlock in xv6

```
void acquire(struct
spinlock *lk)
{
    pushcli(); // disable
    // interrupts to avoid
    // deadlock.
    // The xchg is atomic.
```

```
Void acquire(struct  
spinlock *lk)  
{  
    pushcli(); // disable  
interrupts to avoid  
deadlock.  
  
if(holding(lk))  
panic("acquire");  
.....
```

spinlocks

- Pushcli() - disable interrupts on that processor
- One after another many acquire() can be called on different spinlocks

Keep a count of them

```
void  
release(struct spinlock  
*lk)  
{  
...  
asm volatile("movl $0,  
%0" : "+m" (lk-  
>locked) : );  
popcli();
```

spinlocks

- **Popcli()**

- Restore interrupts if last popcli() call restores ncli to 0 & interrupts were enabled before pushcli() was called

spinlocks

- Always disable interrupts while acquiring spinlock
 - Suppose `iderw` held the idelock and then got interrupted to run `ideintr`.
 - `ideintr` would try to lock `idelock`, see it was held, and wait for it to be released.
 - In this situation, `idelock` will never be released

sleeplocks

- Sleeplocks don't spin. They move a process to a wait-queue if the lock can't be acquired
- XV6 approach to “wait-queues”
 - Any memory address serves as a “wait channel”
 - The sleep() and wakeup() functions just use that address as a ‘condition’
 - There are no per condition process queues! Just

void **sleep()**

**sleep(void *chan,
struct spinlock *lk)**

{

**struct proc *p =
myproc();**

....

**if(lk != &ptable.lock){
acquire(&ptable.lock);**

- At call must hold lock on the resource on which you are going to sleep
- since you are going to change p-> values & call sched() hold

Calls to sleep() : examples of “chan” (output from cscope)

0 console.c consoleread 251
sleep(&input.r, &cons.lock);

2 ide.c iderw 169 sleep(b,
&idelock);

3 log.c begin_op 131
sleep(&log, &log.lock);

6 pipe.c piperead 111
sleep(&p->nread, &p->lock);

7 proc.c wait 317
sleep(curproc,
&phtable.lock);

8 sleeplock.c
acquiresleep 28
sleep(lk, &lk->lk);

9 sysproc.c sys_sleep

```
void wakeup(void  
*chan)  
{  
    acquire(&ptable.lock);  
    wakeup1(chan);  
    release(&ptable.lock);  
}  
  
static void  
wakeup1(void *chan)
```

Wakeup()

- Acquire ptable.lock since you are going to change ptable and p-> values
- just linear search in process table for a process where p->chan is given

sleeplock

// Long-term locks for processes

```
struct sleeplock {
```

```
    uint locked; // Is the lock held?
```

```
    struct spinlock lk; // spinlock protecting this  
    sleep lock
```

Sleeplock acquire and release

```
void  
acquiresleep(struct  
sleeplock *lk)  
{  
    acquire(&lk->lk);  
    while (lk->locked) {
```

```
void  
releasesleep(struct  
sleeplock *lk)  
{  
    acquire(&lk->lk);  
    lk->locked = 0;
```

Where are sleeplocks used?

- **struct buf**
 - waiting for I/O on this buffer
 - **struct inode**
 - waiting for I/o to this inode
- Just two !

Sleeplocks issues

- **sleep-locks support yielding the processor during their critical sections.**
- **This property poses a design challenge:**
 - if thread T1 holds lock L1 and has yielded the processor (waiting for some other condition),
 - and thread T2 wishes to acquire L1,
 - we have to ensure that T1 can execute

More needs of synchronization

- Not only critical section problems
- Run processes in a particular order
- Allow multiple processes read access, but only one process write access
- Etc.

Semaphore

- Synchronization tool that does not require busy waiting
- Semaphore S – integer variable
- Two standard operations modify S and O
- Can only be accessed via two indivisible (atomic) operations
 - wait (S) {
 - while S <= 0
 - ; // no-op

Semaphore for synchronization

- **Counting** semaphore – integer value can range over an unrestricted domain
- **Binary** semaphore – integer value can range only between 0 and 1; can be simpler to implement

Also known as **mutex locks**

- Can implement a counting semaphore **S** as a binary semaphore
- Provides mutual exclusion
- **Semaphore mutex; // initialized to 1**

Semaphore implementation

```
Wait(sem *s) {  
    while(s <=0)  
        block(); // could be ";"  
    s--;  
}  
signal(sem *s) {
```

- Left side – expected behaviour
- Both the wait and signal should be atomic.
- This is the semantics of the semaphore.

Semaphore implementation? - 1

```
struct semaphore {          signal(semaphore *s) {  
    int val;                spinlock(*(s->sl));  
    spinlock lk;            (s->val)++;  
};                          spinunlock(*(s->sl));  
  
sem_init(semaphore  
*s, int initval) {          }  
                            - suppose 2 processes
```

Semaphore implementation? - 2

```
struct semaphore {  
    int val;  
    spinlock lk;  
};  
  
sem_init(semaphore  
*s, int initval) {  
    wait(semaphore *s) {  
        spinlock(&(s->sl));  
        while(s->val <=0) {  
            spinunlock(&(s->sl));  
            spinlock(&(s->sl));  
        }  
    }  
}
```

Semaphore implementation? - 3, idea

```
struct semaphore {          wait(semaphore *s) {  
    int val;                spinlock(&(s->sl));  
    spinlock lk;             while(s->val <=0) {  
};                           Block();  
sem_init(semaphore         }  
*s, int initval) {           (s->val)--;
```

Semaphore implementation? - 3a

```
struct semaphore {  
    int val;  
    spinlock lk;  
    list l;  
};  
  
sem_init(semaphore
```

```
wait(semaphore *s) {  
    spinlock(&(s->sl));  
    while(s->val <=0) {  
        spinunlock(&(s->sl));  
        block(s);  
    }  
}
```

Semaphore implementation? - 3b

```
struct semaphore {          wait(semaphore *s) {  
    int val;                spinlock(&(s->sl));  
    spinlock lk;             while(s->val <=0) {  
    list l;                  block(s);  
};  
sem_init(semaphore           }  
               (s->val)--;
```

Semaphore implementation? - 3c

```
struct semaphore {  
    int val;  
    spinlock lk;  
    list l;  
};  
sem_init(semaphore
```

```
wait(semaphore *s) {  
    spinlock(&(s->sl)); // A  
    while(s->val <=0) {  
        block(s);  
        spinlock(&(s->sl)); // B  
    }  
}
```

Semaphore Implementation

- Must guarantee that no two processes can execute `wait ()` and `signal ()` on the same semaphore at the same time
- Thus, implementation becomes the critical section problem where the wait and signal code are placed in the critical section
 - Could now have busy waiting in critical section implementation

Semaphore in Linux

```
struct semaphore {  
    raw_spinlock_t lock;  
    unsigned int count;  
    struct list_head wait_list;  
};  
  
static noinline void __sched  
__down(struct semaphore  
*sem)
```

```
void down(struct semaphore  
*sem)  
{  
    unsigned long flags;  
  
    raw_spin_lock_irqsave(&sem->lock, flags);  
    if (likely(sem->count > 0))
```

Semaphore in Linux

```
static inline int __sched
__down_common(struct semaphore *sem, long state,
long timeout)
{
    struct task_struct *task =
    current;
    struct semaphore_waiter
    waiter;
    for (;;) {
        if (signal_pending_state(state,
task))
            goto interrupted;
        if (unlikely(timeout <= 0))
            goto timed_out;
        __set_task_state(task, state);
        raw_spin_unlock_irq(&sem->lock);
```

Different uses of semaphores

For mutual exclusion

/*During initialization*/

semaphore sem;

initsem (&sem, 1);

/* On each use*/

P (&sem);

Event-wait

/* During initialization */

semaphore event;

initsem (&event, 0); /* probably at boot time */

**/* Code executed by thread that must wait on
event */**

Control countable resources

```
/* During initialization */  
semaphore counter;  
initsem (&counter, resourceCount);  
  
/* Code executed to use the resource */  
P (&counter); /* Blocks until resource is  
available */
```

Drawbacks of semaphores

- Need to be implemented using lower level primitives like spinlocks
- Context-switch is involved in blocking and signaling – time consuming
- Can not be used for a short critical section

Deadlocks

Deadlock

- two or more processes are waiting indefinitely for an event that can be caused by only one of the waiting processes
- Let S and Q be two semaphores initialized to 1

- **P0 P1**

- **wait (S); wait (Q);**

Example of deadlock

- Let's see the pthreads program : `deadlock.c`
- Same program as on earlier slide, but with `pthread_mutex_lock();`

Non-deadlock, but similar situations

- **Starvation – indefinite blocking**
 - A process may never be removed from the semaphore queue in which it is suspended
- **Priority Inversion**
 - Scheduling problem when lower-priority process holds a lock needed by higher-priority process (so it can not pre-empt lower priority process), and a medium priority process (that does not need the

Livelock

- **Similar to deadlock, but processes keep doing ‘useless work’**
- **E.g. two people meet in a corridor opposite each other**
 - Both move to left at same time
 - Then both move to right at same time
 - Keep Repeating!

Livelock example

```
#include <stdio.h>
#include <pthread.h>

struct person {
    int otherid;
    int otherHungry;
    int myid;
    /* thread two runs in
       this function */
    int spoonWith = 1;
    void *eat(void *param)
    {
        int eaten = 0;
```

More on deadlocks

- Under which conditions they can occur?
- How can deadlocks be avoided/prevented?
- How can a system recover if there is a deadlock ?

System model for understanding deadlocks

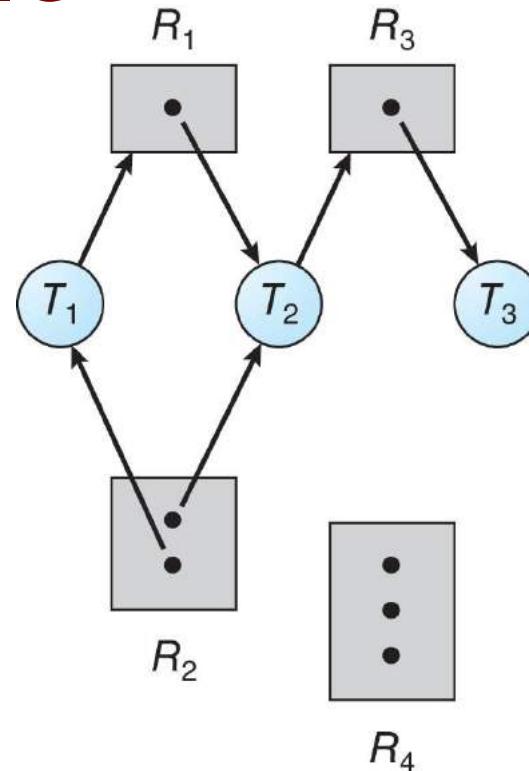
- **System consists of resources**
- **Resource types R₁, R₂, . . . , R_m**
 - CPU cycles, memory space, I/O devices
 - Resource: Most typically a lock, synchronization primitive
- **Each resource type R_i has W_i instances.**
- **Each process utilizes a resource as follows:**

Deadlock characterisation

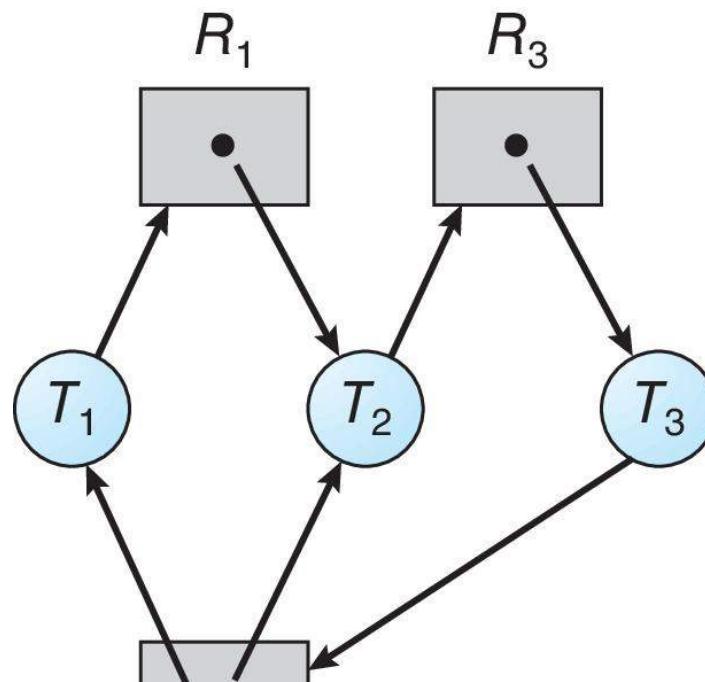
- **Deadlock is possible only if ALL of these conditions are TRUE at the same time**
 - Mutual exclusion:only one process at a time can use a resource
 - Hold and wait: a process holding at least one resource is waiting to acquire additional resources held by other processes
 - No preemption:a resource can be released only

Resource Allocation Graph Example

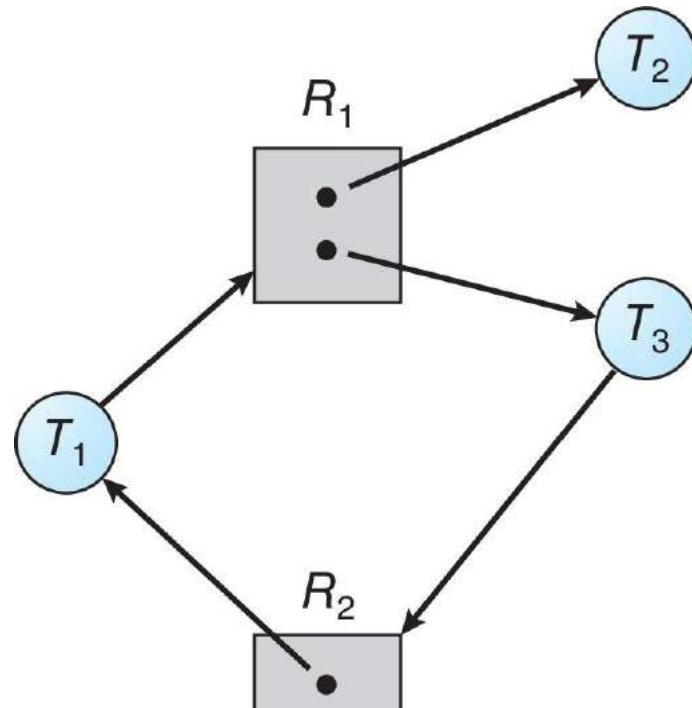
- One instance of R1
- Two instances of R2
- One instance of R3
- Three instances of R4
- T1 holds one instance of R2 and is waiting for an



Resource Allocation Graph with a Deadlock



Graph with a Cycle But no Deadlock



Basic Facts

- **If graph contains no cycles -> no deadlock**
- **If graph contains a cycle :**
 - if only one instance per resource type, then deadlock
 - if several instances per resource type, possibility of deadlock

Methods for Handling Deadlocks

- Ensure that the system will never enter a deadlock state:
 - 1) Deadlock prevention
 - 2) Deadlock avoidance
- 3) Allow the system to enter a deadlock state and then recover
- 4) Ignore the problem and pretend that

(1) Deadlock Prevention

- Invalidate one of the four necessary conditions for deadlock:
- Mutual Exclusion – not required for sharable resources (e.g., read-only files); must hold for non-sharable resources
- Hold and Wait – must guarantee that whenever a process requests a resource, it can obtain it immediately

(1) Deadlock Prevention (Cont.)

- **No Preemption:**
 - If a process that is holding some resources requests another resource that cannot be immediately allocated to it, then all resources currently being held are released
 - Preempted resources are added to the list of resources for which the process is waiting
 - Process will be restarted only when it can regain its

(1) Deadlock prevention: Circular Wait

- **Invalidating the circular wait condition is most common.**
- **Simply assign each resource (i.e., mutex locks) a unique number**

```
/* thread_one runs in this function */
void *do_work_one(void *param)
{
    pthread_mutex_lock(&first_mutex);
    pthread_mutex_lock(&second_mutex);
    /**
     * Do some work
     */
    pthread_mutex_unlock(&second_mutex);
    pthread_mutex_unlock(&first_mutex);

    pthread_exit(0);
}

/* thread_two runs in this function */
void *do_work_two(void *param)
{
    pthread_mutex_lock(&second_mutex);
    pthread_mutex_lock(&first_mutex);
```

(1) Preventing deadlock: cyclic wait

- **Locking hierarchy : Highly preferred technique in kernels**
 - Decide an ordering among all ‘locks’
 - Ensure that on ALL code paths in the kernel, the locks are obtained in the decided order!
 - Poses coding challenges!
 - A key differentiating factor in kernels

(1) Prevention in Xv6: Lock Ordering

- lock on the directory, a lock on the new file's inode, a lock on a disk block buffer, idelock, and ptable.lock.

(2) Deadlock avoidance

- **Requires that the system has some additional a priori information available**
 - Simplest and most useful model requires that each process declare the maximum number of resources of each type that it may need
 - The deadlock-avoidance algorithm dynamically examines the resource-allocation state to ensure that there can never be a circular-wait condition

(2) Deadlock avoidance

- Please see: concept of safe states, unsafe states, Banker's algorithm

(3) Deadlock detection and recovery

- **How to detect a deadlock in the system?**
- **The Resource-Allocation Graph is a graph.**
 - Need an algorithm to detect cycle in a graph.**
- **How to recover?**
 - Abort all processes or abort one by one?
 - Which processes to abort?
 - Priority ?

“Condition” Synchronization Tool

What is condition variable?

- A variable with a sleep queue
- Threads can sleep on it, and wake-up all remaining

Struct condition {

Proc *next

Proc *prev

Code for condition variables

```
//Spinlock s is held  
before calling wait
```

```
void wait (condition  
*c, spinlock_t *s)
```

```
(
```

```
spin_lock (&c-  
>listLock);
```

```
void do_signal  
(condition *c)
```

```
/*Wakeup one thread  
waiting on  
the condition*/
```

```
{
```

```
spin_lock (&c-
```

Semaphore implementation using condition variables?

- Is this possible?
- Can we try it?

```
typedef struct semaphore {
```

```
//something
```

```
condition c;
```

```
}semaphore;
```

Classical Synchronization Problems

Bounded-Buffer Problem

- **Producer and consumer processes**
 - N buffers, each can hold one item
- **Producer produces ‘items’ to be consumed by consumer , in the bounded buffer**
- **Consumer should wait if there are no items**
- **Producer should wait if the ‘bounded buffer’ is full**

Bounded-Buffer Problem: solution with semaphores

- **Semaphore mutex initialized to the value 1**
- **Semaphore full initialized to the value 0**
- **Semaphore empty initialized to the value N**

Bounded-buffer problem

The structure of the producer process

```
do {  
    // produce an item in  
    nextp  
  
    wait (empty);  
  
    wait (mutex);
```

The structure of the Consumer process

```
do {  
    wait (full);  
  
    wait (mutex);  
  
    // remove an item from
```

Bounded buffer problem

- Example : pipe()
- Let's see code of pipe in xv6 – a solution using sleeplocks

Readers-Writers problem

- **A data set is shared among a number of concurrent processes**
 - Readers – only read the data set; they do not perform any updates
 - Writers – can both read and write
- **Problem – allow multiple readers to read at the same time**

The structure of a writer process

```
do {  
    wait (wrt) ;  
    // writing is performed  
    signal (wrt) ;  
} while (TRUE);
```

The structure of a reader process

```
do {  
    wait (mutex) ;  
    readcount ++ ;  
    if (readcount == 1)  
        wait (wrt) ;  
    signal (mutex)  
    // reading is performed  
    wait (mutex) ;  
    readcount - - ;  
    if (readcount == 0)
```

Readers-Writers Problem Variations

- **First variation – no reader kept waiting unless writer has permission to use shared object**
- **Second variation – once writer is ready, it performs write asap**
- **Both may have starvation leading to even more variations**

Reader-write lock

- A lock with following operations on it
 - Lockshared()
 - Unlockshared()
 - LockExcl()
 - UnlockExcl()
- Possible additions
 - Downgrade() -> from excl to shared

Code for reader-writer locks

```
struct rwlock {  
    int nActive; /* num of  
active readers, or -1 if a  
writer is active */  
  
    int nPendingReads;  
  
    int nPendingWrites;  
  
    spinlock_t sl;
```

```
void lockShared  
(struct rwlock *r)  
{  
    spin_lock(&r->sl);  
  
    r->nPendingReads++;  
  
    if (r->nPendingWrites  
        > 0)
```

Code for reader-writer locks

```
void unlockShared  
(struct rwlock *r)  
  
{  
    spin_lock (&r->sl);  
  
    r->nActive--;  
  
    if (r->nActive == 0) {
```

```
void lockExclusive  
(struct rwlock *r)  
  
(  
    spin_lock (&r->sl);  
  
    r->nPendingWrites++;  
  
    while (r->nActive)
```

Code for reader-writer locks

```
void unlockExclusive  
(struct rwlock *r){  
  
    boolean t  
  
    wakeReaders;  
  
    spin_lock (&r->sl);  
  
    r->nActive = 0;  
  
    wakeReaders = (r-
```

Try writing code for
downgrade and
upgrade

Try writing a reader-
writer lock using
semaphores!

Dining-Philosophers Problem

- Philosophers spend their lives thinking and eating
- Don't interact with their neighbors, occasionally try to pick up 2 chopsticks (one at a time) to eat



Dining philosophers: One solution

The structure of Philosopher i:

```
do {  
    wait ( chopstick[i] );  
    wait ( chopStick[ (i + 1) % 5] );  
    // eat  
    signal ( chopstick[i] );
```

Dining philosophers: Possible approaches

- Allow at most four philosophers to be sitting simultaneously at the table.
- Allow a philosopher to pick up her chopsticks only if both chopsticks are available
 - to do this, she must pick them up in a critical section

Other solutions to dining philosopher's problem

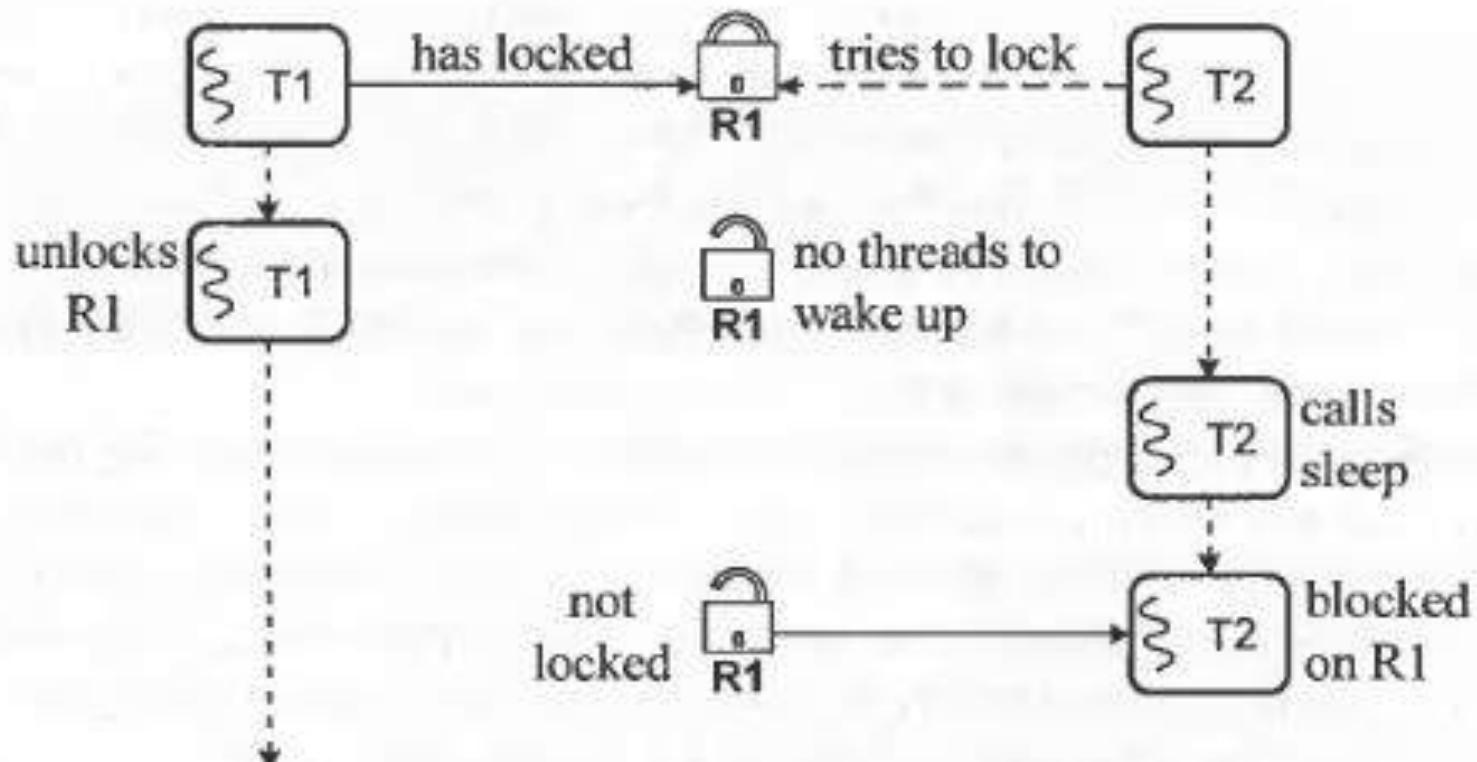
- Using higher level synchronization primitives like 'monitors'

Practical Problems

Lost Wakeup problem

- **The sleep/wakeup mechanism does not function correctly on a multiprocessor.**
- **Consider a potential race:**
 - Thread T1 has locked a resource R1.
 - Thread T2, running on another processor, tries to acquire the resource, and finds it locked.
 - T2 calls sleep() to wait for the resource.

Lost Wakeup problem



Thundering herd problem

- **Thundering Herd problem**
 - On a multiprocessor, if several threads were locked the resource
 - Waking them all may cause them to be simultaneously scheduled on different processors
 - and they would all fight for the same resource again.

Starvation

Case Studies

Linux Synchronization

- Prior to kernel Version 2.6, disables interrupts to implement short critical sections
- Version 2.6 and later, fully preemptive
- Linux provides:
 - semaphores
 - spinlocks

Linux Synchronization

- **Atomic variables**

atomic_t is the type for atomic integer

- **Consider the variables**

atomic_t counter;

int value;

Pthreads synchronization

- **Pthreads API is OS-independent**
- **It provides:**
 - mutex locks
 - condition variables
- **Non-portable extensions include:**
 - read-write locks
 - spinlocks

Synchronization issues in xv6 kernel

Difference approaches

- **Pros and Cons of locks**
 - Locks ensure serialization
 - Locks consume time !
- **Solution – 1**
 - One big kernel lock
 - Too inefficient
- **Solution – 2**

Three types of code

- **System calls code**

- Can it be interruptible?
 - If yes, when?

- **Interrupt handler code**

- Disable interrupts during interrupt handling or not?
 - Deadlock with iderw ! - already seen

- **Process's user code**

Interrupts enabling/disabling in xv6

- **Holding every spinlock disables interrupts!**
- **System call code or Interrupt handler code won't be interrupted if**
 - The code path followed took at least once spinlock !
 - Interrupts disabled only on that processor!
- **Acquire calls pushcli() before xchg()**
- **Release calls popcli() after xchg()**

Memory ordering

- Compiler may generate machine code for out-of-order execution !
- Processor pipelines can also do the same!

- Consider this

```
1)l = malloc(sizeof *l);  
2)l->data = data;  
3)acquire(&listlock);  
4)l->next = list;  
5)list = l;  
6)release(&listlock);
```

Lost Wakeup?

- **Do we have this problem in xv6?**
- **Let's analyze again!**
 - The race in acquiresleep()'s call to sleep() and releasesleep()
- **T1 holding lock, T2 willing to acquire lock**
 - Both running on different processor
 - Or both running on same processor

Code of sleep()

```
if(lk != &ptable.lock){  
    acquire(&ptable.lock);  
    release(lk);  
}
```

- Why this check?
- Deadlock otherwise!

Exercise question : 1

Sleep has to check lk != &ptable.lock to avoid a deadlock

Suppose the special case were eliminated by replacing

```
if(lk != &ptable.lock){  
    acquire(&ptable.lock);
```

bget() problem

- **bget() panics if no free buffers!**
- **Quite bad**
- **Should sleep !**
- **But that will introduce many deadlock problems. Which ones ?**

iget() and ilock()

- **iget() does no hold lock on inode**
- **Illock() does**
- **Why this separation?**
 - Performance? If you want only “read” the inode, then why lock it?
- **What if iget() returned the inode locked?**

Interesting cases in namex()

```
while((path =  
skipelem(path, name))  
!= 0){  
ilock(ip);  
if(ip->type != T_DIR){  
iunlockput(ip);  
return 0;  
}  
if((next = dirlookup(ip,  
name, 0)) == 0){  
iunlockput(ip);  
return 0;  
}  
iunlockput(ip);
```

Xv6

Interesting case of holding and releasing
ptable.lock in scheduling

One process acquires, another releases!

Giving up CPU

- **A process that wants to give up the CPU**
 - must acquire the process table lock ptable.lock
 - release any other locks it is holding
 - update its own state (proc->state),
 - and then call sched()
- **Yield follows this convention, as do sleep and exit**

Interesting race if ptable.lock is not held

- Suppose P1 calls yield()
- Suppose yield() does not take ptable.lock
 - Remember yield() is for a process to give up CPU
- Yield sets process state of P1 to RUNNABLE
- Before yield's sched() calls swtch()
- Another processor runs scheduler() and runs P1 on that processor

Homework

- **Read the version-11 textbook of xv6**
- **Solve the exercises!**

Revision of Memory management concepts

Revision

- Memory layout of C program
- Address binding times: compile, link, load
- MMU Schemes
 - No MMU, Base+Relocation, Multiple
Base+Relocation = Segmentation, Paging
 - Hierarchical paging
 - TLB,
 - External and Internal fragmentation

More on Linking, Loading

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()`
- Problems
 - Static linking: Big executable files,
 - Static loading: need to load functions even if they do not execute

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

Dynamic Linking on Linux

```
#include  
<stdio.h>
```

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

```
int main() {  
    int a, b;
```

Output of objdump -x -D

Disassembly of section .text:

```
0000000000001189 <main>:
```

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

Disassembly of section

.plt.got:

```
0000000000001080
```

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

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

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 ?

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

All parts of array a[] not accessed

Function f() may not be called

a[4096][40

96]

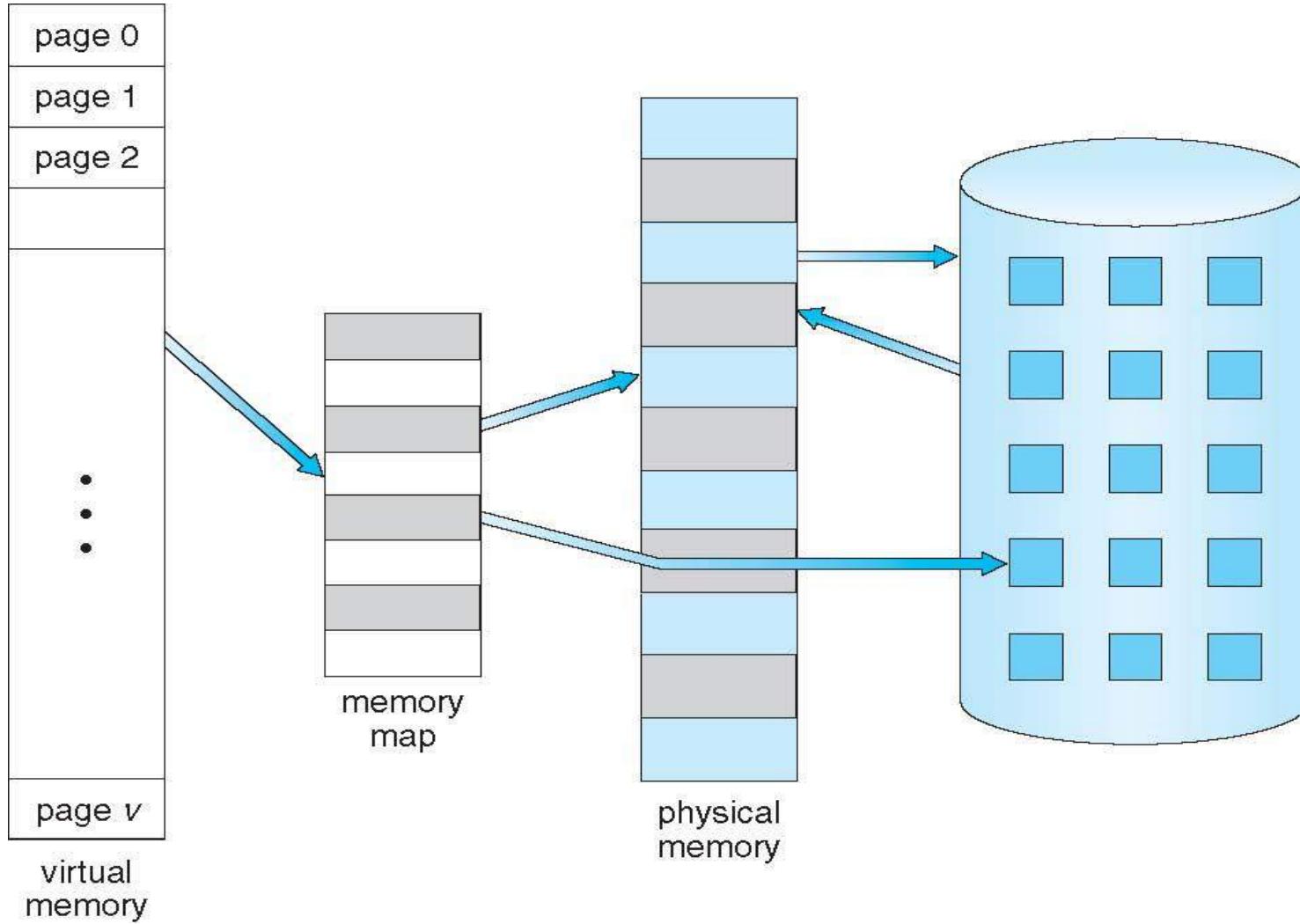
Some problems with schemes discussed so far

- 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

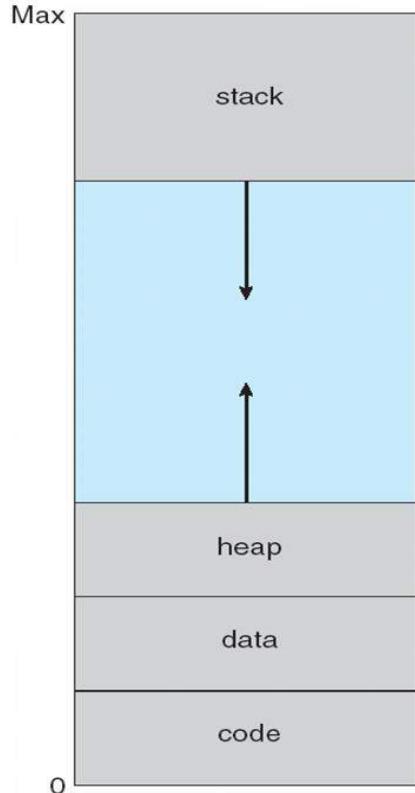
What is virtual memory?

- 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

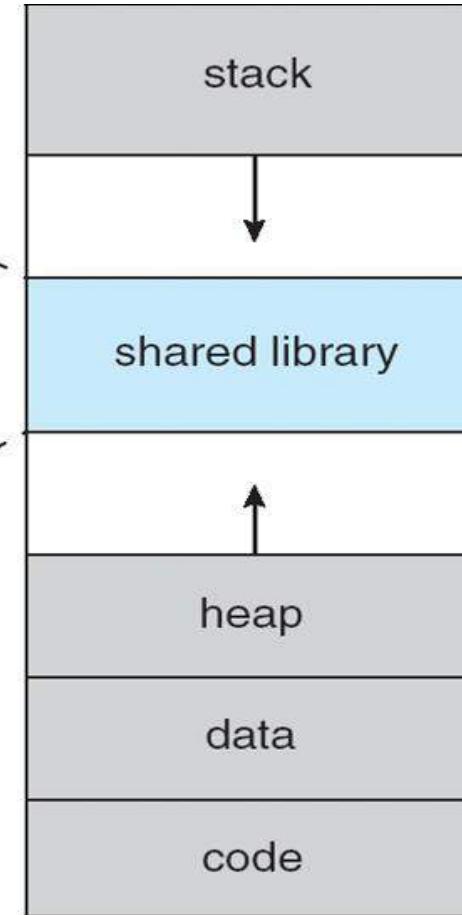
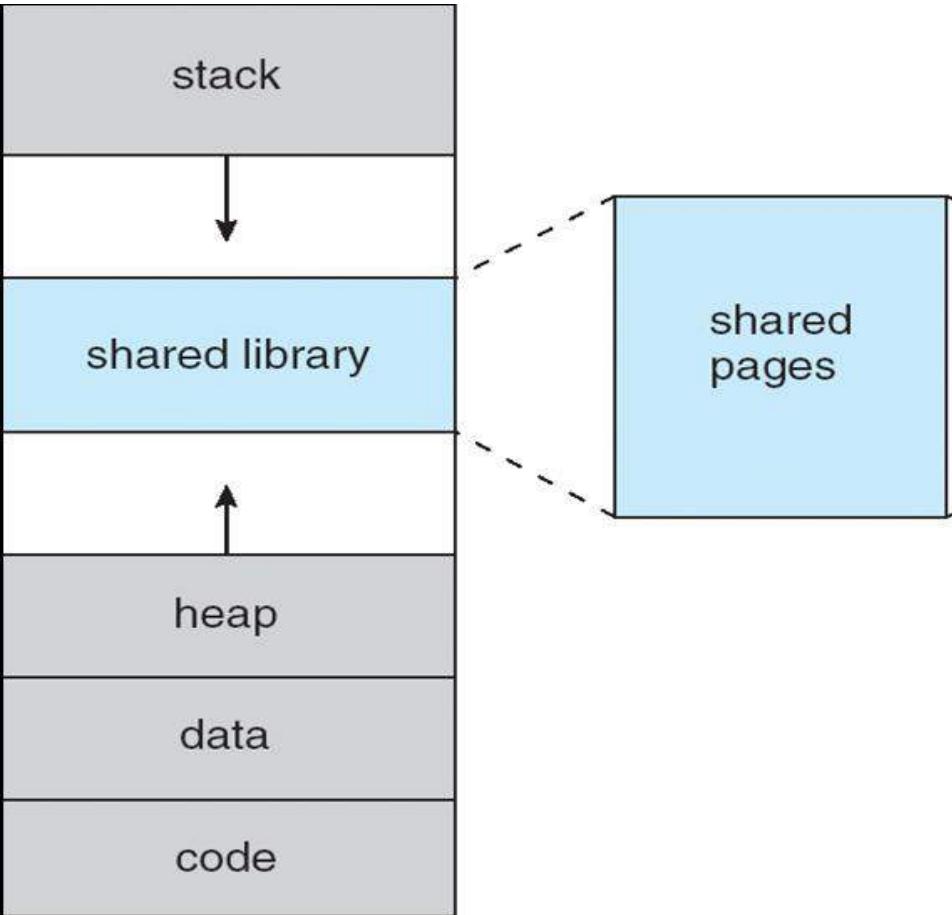
Virtual Memory
Larger
Than
Physical
Memory



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 neded (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

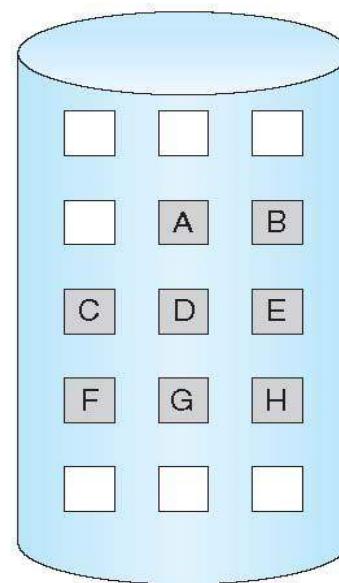
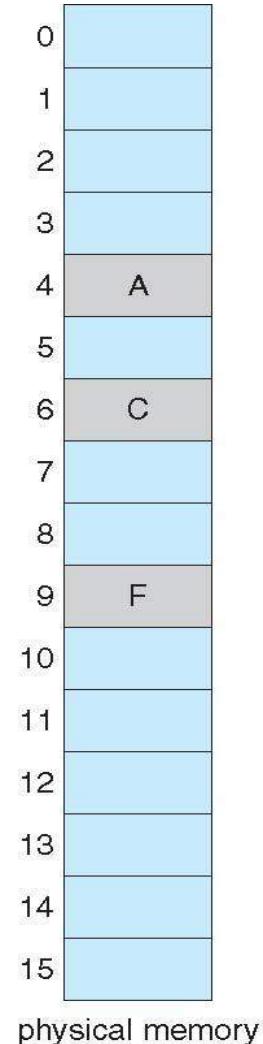
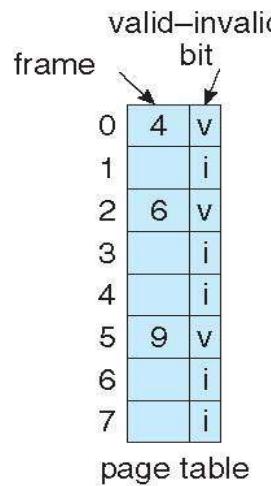
Frame #	valid-invalid bit
	v
	v
	v
	v
....	i
	i
	i

- With each page table entry a valid–invalid bit is associated
 - v: in-memory – memory resident
 - i : not-in-memory or illegal

During address

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

Important (not all) steps on a Page fault

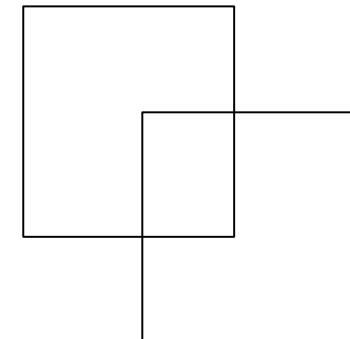
- 1) Operating system looks at another data structure (table), most likely in PCB itself, to decide:
 - If it's Invalid reference -> abort the process (segfault)
 - Just not in memory -> Need to get the page in memory
- 2) Get empty frame (this may be complicated, may need evicting a frame if no free frames available!)
- 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

Issues with page fault handling

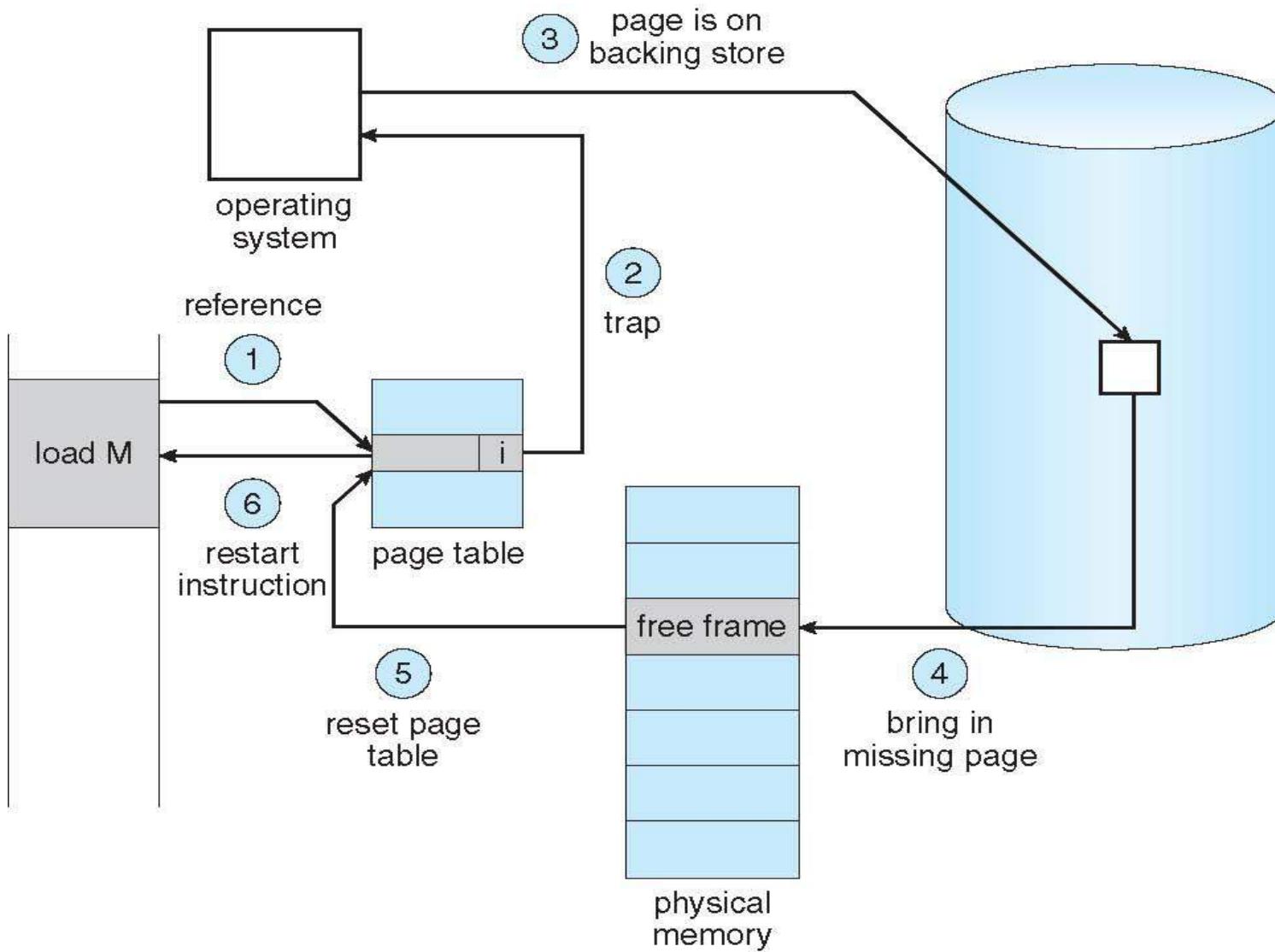
- Extreme case – start process with *no* pages in memory
 - OS sets instruction pointer to first instruction of process, non-memory-resident -> page fault
 - And for every other process pages on first access

Problem with 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 (detailed)

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

Page fault handling (detailed)

- 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

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)

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

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

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

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

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

$$p < .0000025$$

An optimization: Copy on write

The problem with
`fork()` and `exec()`.

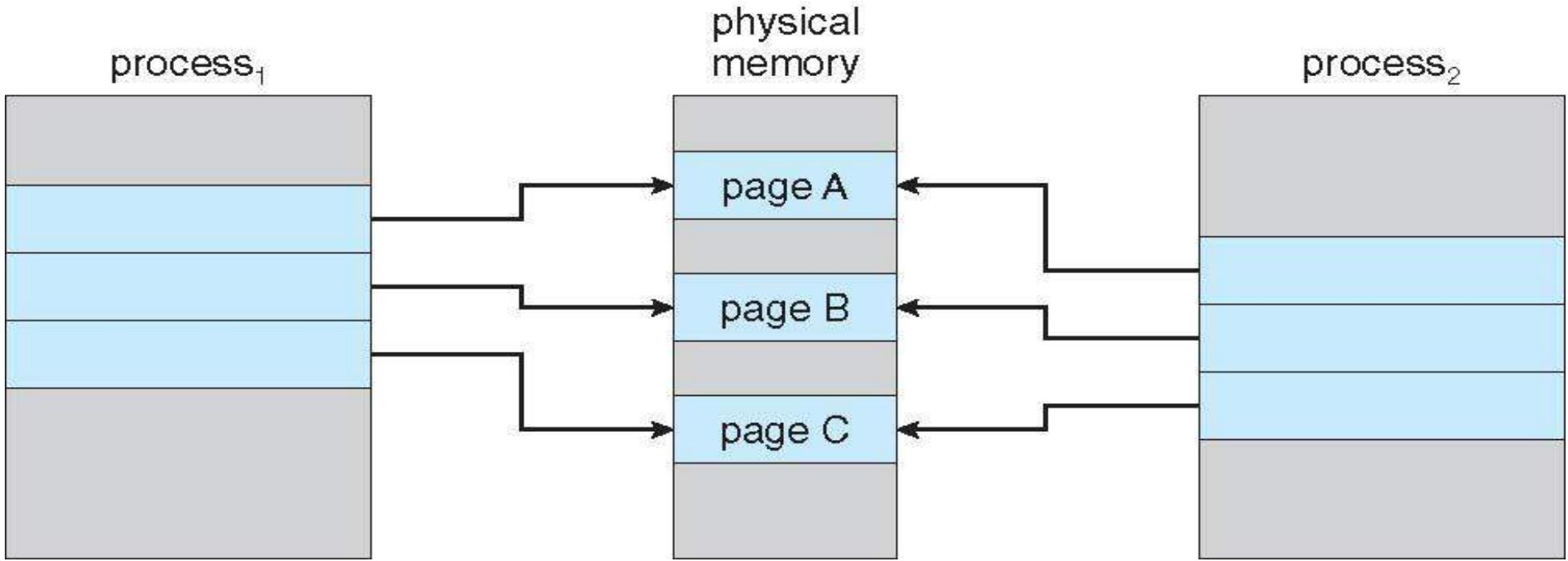
Consider the case of a
shell

```
scanf ("%s",  
cmd);  
  
if (strcmp (cmd,  
"exit") == 0)
```

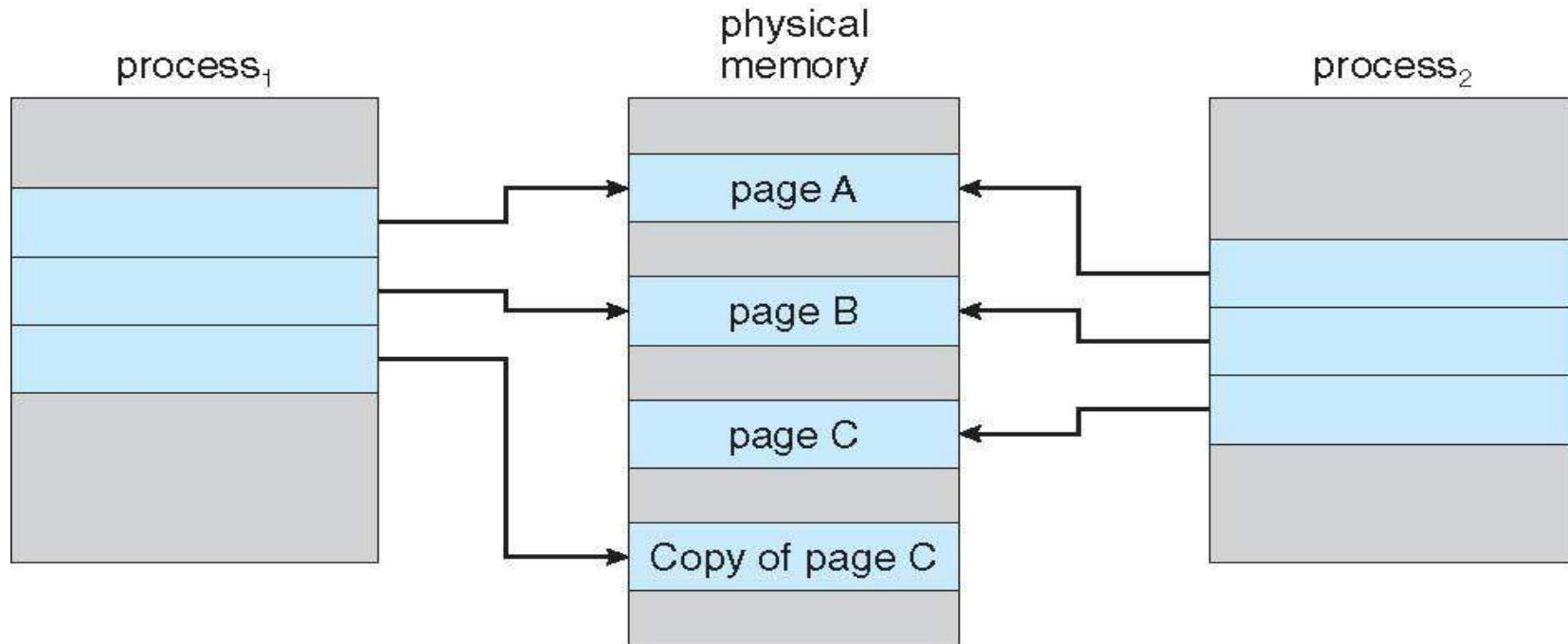
- During `fork()`
 - Pages of parent were duplicated
 - Equal amount of page frames were allocated
 - Page table for child differed from parent (as it has another set of frames)
- In `exec()`
 - The page frames of child were taken away and new frames were allocated
 - Child's page table was rebuilt!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



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

Scheduler

Scheduler – in most simple terms

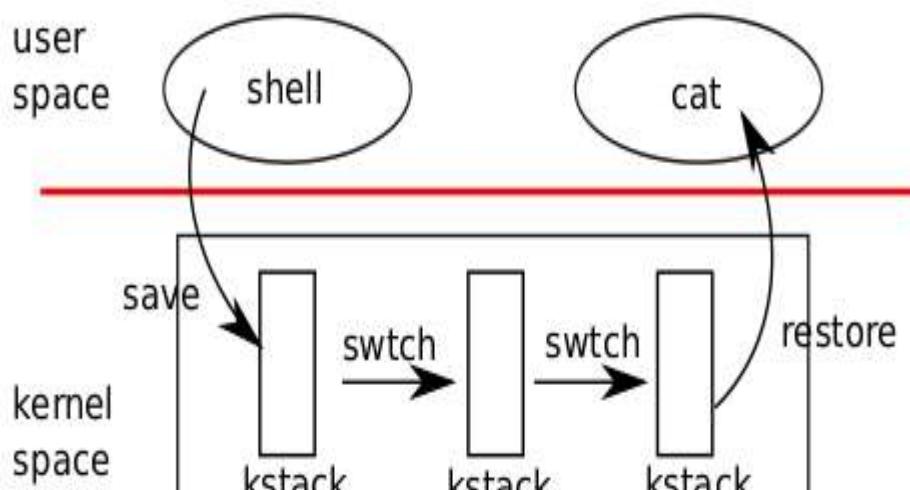
- **Selects a process to execute and passes control to it !**
 - The process is chosen out of “READY” state processes
 - Saving of context of “earlier” process and loading of context of “next” process needs to happen
- **Questions**

Steps in scheduling scheduling

- Suppose you want to switch from P1 to P2 on a timer interrupt
- P1 was doing

`F() { i++; j++; }`

4 stacks need to change!



- User stack of process ->
- kernel stack of process
- Switch to kernel stack

scheduler()

- **Enable interrupts**
- **Find a RUNNABLE process.**
Simple round-robin!
- **c->proc = p**
- **switchuvm(p) : Save TSS and**

swtch

swtch:

movl 4(%esp), %eax

movl 8(%esp), %edx

Save old callee-saved registers

pushl %ebp

scheduler()

- **swtch(&(c->scheduler), p->context)**
- **Note that when scheduler() was called, when P1 was running**
- **After call to swtch() shown**

swtch(old, new)

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

scheduler()

- Called from?
 - mpmain()
 - No where else!
- **sched() is another scheduler function !**

void

sched(void)

{

int intena;

struct proc *p =

myproc();

/* Call this after interrupt */

sched()

- **get current process**
- **Error checking code (ignore as of now)**
- **get interrupt enabled status**

sched() and schduler()

```
sched() {
```

```
...
```

```
swtch(&p-
```

```
>scheduler) saves context in c-
```

```
m>scheduler, sched() saves
```

```
scheduler(void)  
{
```

```
...
```

```
swtch(&c-
```

```
>scheduler), p-
```

```
>context in p->context
```

sched() and scheduler() as co-routines

- In **sched()**

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

- In **scheduler()**

swtch(&(c->scheduler), p-

To summarize

- On a timer interrupt during P1
 - trap() is called.
Stack has changed from
- Now the loop in scheduler()
 - calls switchkvm()
 - Then continues to find next

File System Code

**open, read, write, close, pipe, fstat, chdir,
dup, mknod, link, unlink, mkdir,**

Files, Inodes, Buffers

What we already know

- **File system related system calls**

- deal with ‘**fd**’ arrays (**ofile** in xv6). **open()** returns first empty index. **open** should ideally locate the inode on disk and initialize some data structures
- maintain ‘**offsets**’ within a ‘file’ to support sequential read/write
- **dup()** like system calls duplicate pointers in fd-array
- read/write like system calls going through ‘**ofile**’

xv6 file handling code

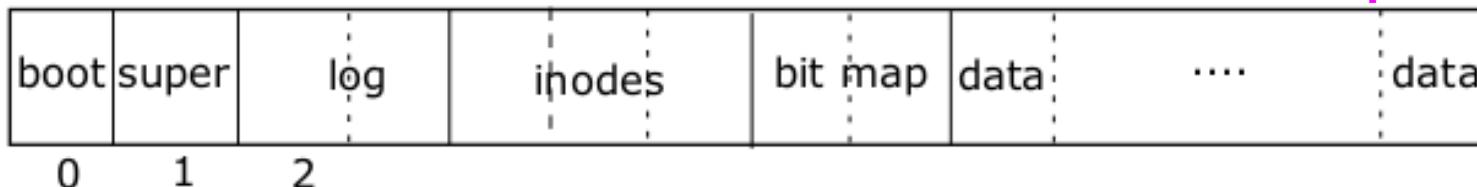
- Is a very good example in ‘design’ of a layered and modular architecture
- Splits the entire work into different modules, and modules into functions properly
- The task of each function is neatly defined and compartmentalized

Layers of xv6 file system code

System Calls	open, read, write, close, link, pipe, mknod, unlink, fstat, mkdir, chdir, dup,
File descriptor	file.c fileinit, filealloc, filedup, fileclose, filestat, fileread, filewrite,
Pathname	fs.c namex, namei, nameiparent, skipelem
Directory	fs.c dirlookup, dirlink
Inode	fs.c iiinit, ialloc, iupdate, ige, idup, ilock, unlock, iput, unlockput, itrunc, stati, readi, writei, bmap ,
Logging	Block allocation on disk: balloc , bfree log.c : begin_op, end_op, initlog, commit,
Buffer cache	bio.c binit, bget, bread, bwrite, brelse

Layout of xv6 file system

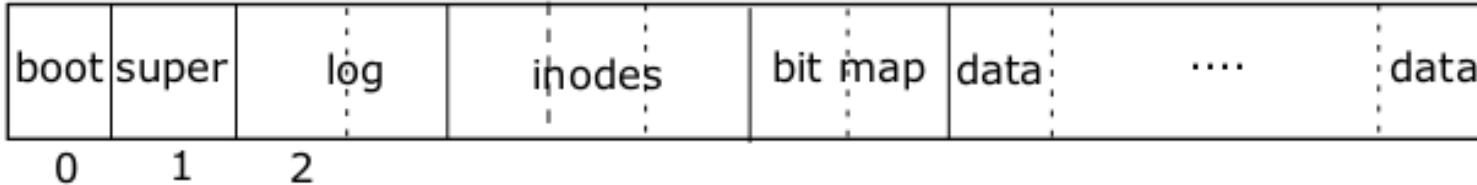
Pointer shown are conceptual



May see the code of mkfs.c to get insight into the layout

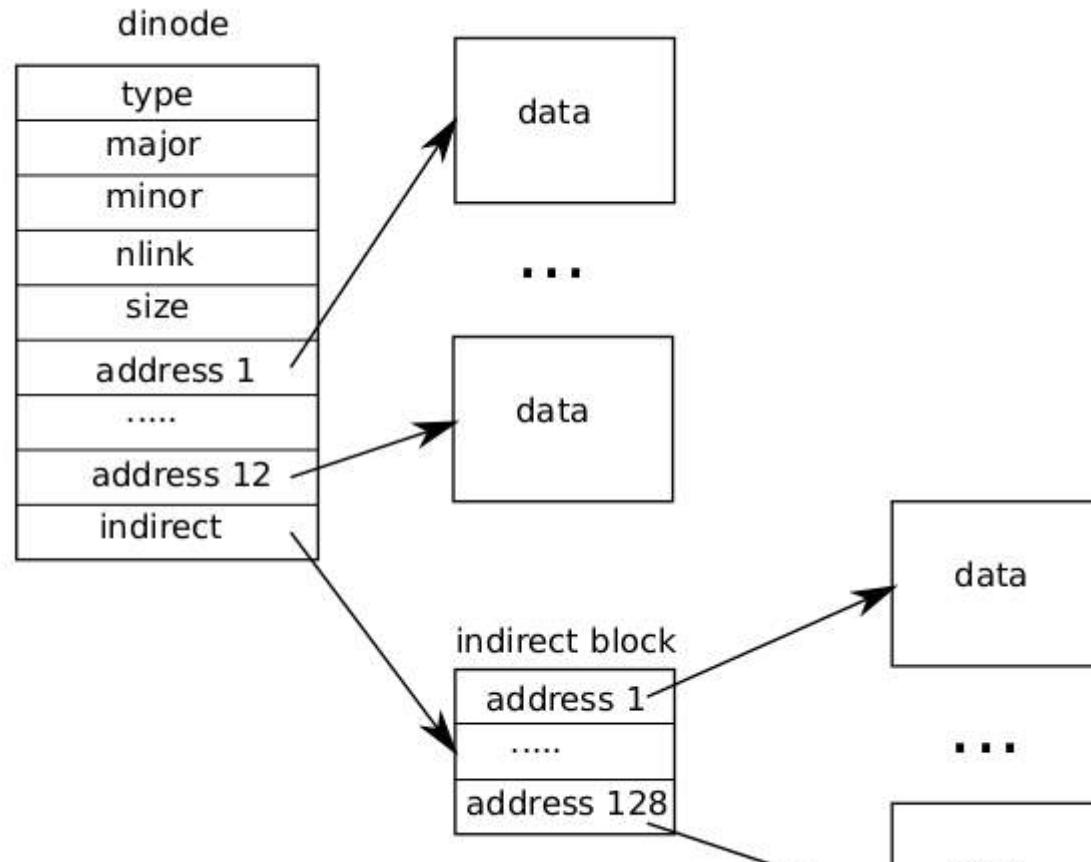
```
struct superblock {  
    uint size; // Size of file system image (blocks)  
    uint nblocks; // Number of data blocks  
    uint ninodes; // Number of inodes.  
    uint nlog; // Number of log blocks  
    uint logstart; // Block number of first log block  
    uint inodestart; // Block number of first inode block  
    uint bmapstart; // Block number of first free map block
```

Layout of xv6 file system



```
#define NDIRECT 12
#define NINDIRECT (BSIZE / sizeof(uint))
#define MAXFILE (NDIRECT + NINDIRECT)
// On-disk inode structure
struct dinode {
    short type; // File type
    short major; // Major device number (T_DEV only)
    short minor; // Minor device number (T_DEV only)
    short nlink; // Number of links to inode in file system
    uint size; // Size of file (bytes)
    uint addrs[NDIRECT+1]; // Data block addresses
};
```

File on disk



Let's discuss lowest layer first

System Calls	open, read, write, close, link, pipe, mknod, unlink, fstat, mkdir, chdir, dup,
File descriptor	file.c fileinit, filealloc, filedup, fileclose, filestat, fileread, filewrite,
Pathname	fs.c namex, namei, nameiparent, skipellem
Directory	fs.c dirlookup, dirlink
Inode	fs.c iiinit, ialloc, iupdate, ige, idup, ilock, iunlock, iput, iunlockput, itrunc, stati, readi, writei, bmap ,
Logging	Block allocation on disk: balloc , bfree log.c : begin_op, end_op, initlog, commit,
Buffer cache	bio.c binit, bget, bread, bwrite, brelse

ide.c: idewait, ideinit, idestart, ideintr, iderw

- **static struct spinlock idelock;**
- **static struct buf *idequeue;**
- **static int havedisk1;**
- **ideinit**
 - was called from **main.c: main()**
 - Initialized IDE controller by writing to certain ports

ide.c: idewait, ideinit, idestart, ideintr, iderw

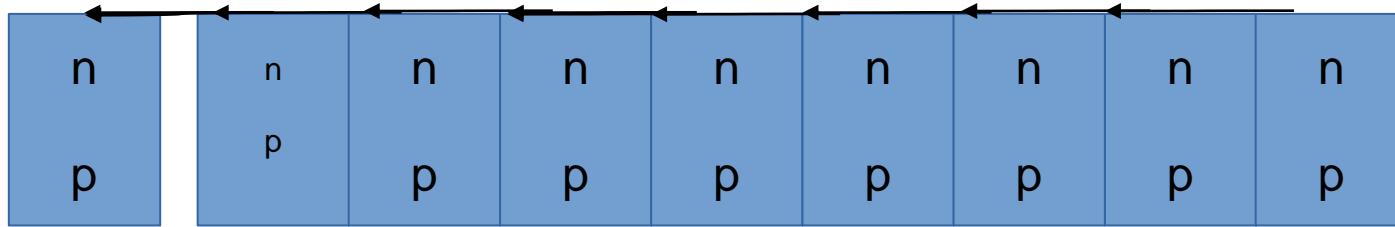
- **void idestart(buf *b)**
 - static void **idestart(struct buf *b)**
 - Calculate sector number on disk using b->blockno
 - Issue a read/write command to IDE controller.
 - (This is the first buf on **idequeue**)
- **ideintr**
 - Take **idelock**. Called on IDE interrupt (through

Let's see buffer cache layer

System Calls	open, read, write, close, link, pipe, mknod, unlink, fstat, mkdir, chdir, dup,
File descriptor	file.c fileinit, filealloc, filedup, fileclose, filestat, fileread, filewrite,
Pathname	fs.c namex, namei, nameiparent, skipellem
Directory	fs.c dirlookup, dirlink
Inode	fs.c iiinit, ialloc, iupdate, ige, idup, ilock, iunlock, iput, iunlockput, itrunc, stati, readi, writei, bmap ,
Logging	Block allocation on disk: balloc , bfree log.c : begin_op, end_op, initlog, commit,
Buffer cache	bio.c binit, bget, bread,

Reminder: After main() -> binit()

head



Conceptually
Linked lists
this

Buffers keep
moving on
list, as LRU

struct buf

```
struct buf {  
    int flags; // 0 or B_VALID or B_DIRTY  
    uint dev; // device number  
    uint blockno; // seq block number on device  
    struct sleeplock lock; // Lock to be held by process using it  
    uint refcnt; // Number of live accesses to the buf  
    struct buf *prev; // cache list  
    struct buf *next; // cache list  
    struct buf *qnext; // disk queue  
    uchar data[BSIZE]; // data 512 bytes  
};
```

buffer cache:

static struct buf* bget(uint dev, uint blockno)

- The **bcache.head** list is maintained on **Most Recently Used (MRU) basis**
 - **head.next** is the Most Recently Used (MRU) buffer
 - hence **head.prev** is the Least Recently Used (LRU)
- **Look for a buffer with **b->blockno = blockno** and **b->dev = dev****
 - Search the **head.next** list for existing buffer (MRU)

buffer cache:
struct buf* bread(uint dev, uint blockno)

```
struct buf*
bread(uint dev, uint
blockno)
{
    struct buf *b;
    b = bget(dev, blockno); panic("bwrite");
```

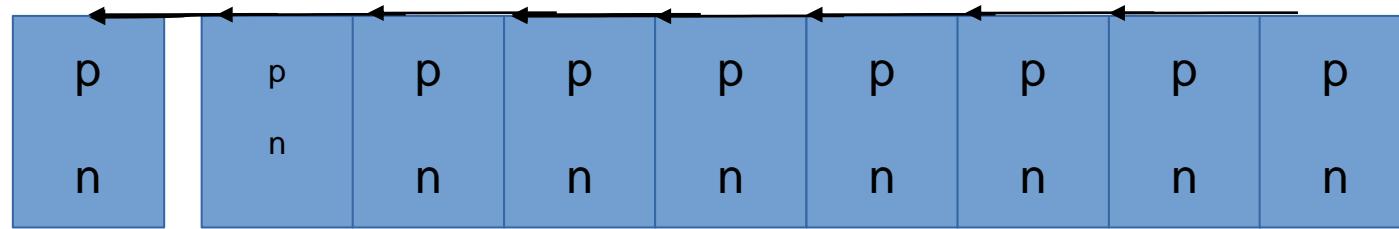
void
bwrite(struct buf *b)
{
if(!holdingsleep(&b->lock))

buffer cache: **void brelse(struct buf *b)**

- **release lock on buffer**
- **b->refcnt = 0**
- **If b->refcnt = 0**
 - Means buffer will no longer be used
 - Move it to **front** of the front of **bcache.head**

Overall in this diagram

head



Buffers keep moving to the front of the list and around
The list always contains **NBLIF=30** buffers

File descriptor layer code

System Calls	open, read, write, close, link, pipe, mknod, unlink, fstat, mkdir, chdir, dup,
File descriptor	file.c fileinit, filealloc, filedup, fileclose, filestat, fileread, filewrite,
Pathname	fs.c namex, namei, nameiparent, skipellem
Directory	fs.c dirlookup, dirlink
Inode	fs.c iiinit, ialloc, iupdate, ige, idup, ilock, iunlock, iput, iunlockput, itrunc, stati, readi, writei, bmap ,
Logging	Block allocation on disk: balloc , bfree
Buffer cache	log.c : begin_op, end_op, initlog, commit, bio.c binit, bget, bread, bwrite, brelse

data structures related to “file” layer

```
struct file {  
    enum { FD_NONE,  
        FD_PIPE, FD_INODE }  
    type;  
    int ref; // reference  
    count  
    char readable;
```

```
struct proc {  
    ...  
    struct file  
    *ofile[NOFILE]; //Open  
    files per process  
    ...  
}
```

Multiple processes accessing same file.

- **Each will get a different ‘struct file’**
 - but share the inode !
 - different offset in struct file, for each process
 - Also true, if same process opens file many times
- **File can be a PIPE (more later)**
 - what about STDIN, STDOUT, STDERR files ?
 - Figure out!

file layer functions

- **filealloc**

- find an empty struct file in ‘ftable’ and return it
- set ref = 1

- **filedup(file *)**

- simply ref++

- **fileclose**

- --ref
- if ref = 0
 - free struct file
 - input() / pipeclose()
 - note – transaction if input() called

- **filestat**

file layer functions

- **fileread**
 - call readi() or piperead()
 - readi() later calls device-read or inode read (using bread())
- **filewrite**
 - call pipewrite() or
- **Why does readi() call read on the device , why not fileread() itself call device read ?**

Reading Directory Layer

System Calls	open, read, write, close, link, pipe, mknod, unlink, fstat, mkdir, chdir, dup,
File descriptor	file.c fileinit, filealloc, filedup, fileclose, filestat, fileread, filewrite,
Pathname	fs.c namex, namei, nameiparent, skipelem
Directory	fs.c dirlookup, dirlink
Inode	fs.c iiinit, ialloc, iupdate, iget, idup, ilock, iunlock, iput, iunlockput, itrunc, stati, readi, writei, bmap ,
Logging	Block allocation on disk: balloc, bfree log.c : begin_op, end_op, initlog, commit,
Buffer cache	bio.c binit, bget, bread, bwrite, brelse

directory entry

```
#define DIRSIZ 14
```

```
struct dirent {  
    ushort inum;  
    char name[DIRSIZ];  
};
```

struct inode*
dirlookup(struct inode *dp, char *name, uint *poff)

- **Given a pointer to directory inode (dp), name of file to be searched**
 - return the pointer to inode of that file (NULL if not found)
 - set the ‘offset’ of the entry found, inside directories data blocks, in poff
- **How was ‘dp’ obtained? Who should be calling dirlookup? Why is poff returned?**

int

dirlink(struct inode *dp, char *name, uint inum)

- **Create a new entry for ‘name’_’inum’ in directory given by ‘dp’**
 - inode number must have been obtained before calling this. How to do that?
- **Use dirlookup() to verify entry does not exist!**
- **Get empty slot in directory’s data block**

namex

- **Called by namei(), or nameiparent()**
- **Just iteratively split a path using “/” separator and get inode for last component**
- **iget() root inode, then**
- **Repeatedly calls**
 - split on “/”, dirlookup() for next component

races in namex()

- **Crucial. Called so many times!**
- **one kernel thread is looking up a pathname
another kernel thread may be changing the
directory by calling unlink**
 - when executing dirlookup in namex, the lookup
thread holds the lock on the directory and
dirlookup() returns an inode that was obtained using
iget.

Let's see Inode Layer

System Calls	open, read, write, close, link, pipe, mknod, unlink, fstat, mkdir, chdir, dup,
File descriptor	file.c fileinit, filealloc, filedup, fileclose, filestat, fileread, filewrite,
Pathname	fs.c namex, namei, nameiparent, skipellem
Directory	fs.c dirlookup, dirlink
Inode	fs.c iinit, ialloc, iupdate, ige, idup, ilock, iunlock, iput, iunlockput, itrunc, stati, readi, writei, bmap,
Logging	
Buffer cache	Block allocation on disk: balloc, bfree log.c : begin_op, end_op, initlog, commit,

On disk & in memory inodes

```
struct {  
    struct spinlock  
    lock;  
  
    struct inode  
    inode[NINODE];
```

// in-memory copy of
an inode

```
struct inode {  
    uint dev; // Device  
    number  
  
    uint inum; // Inode  
    number
```

In memory inodes

- Kernel keeps a subset of on disk inodes, those in use, in memory
 - as long as ‘ref’ is >0
- The **iget** and **iput** functions acquire and release pointers
- See the caller graph of **iget()**
 - all those who call **iget()**
- Sleep lock in ‘inode’ protects
 - fields in inode

iget and iupdate

- **iget**

- searches for an existing/free inode in icache and returns pointer to one
- if found, increments ref and returns pointer to inode

- **iupdate(inode *ip)**

- read on disk block of inode
- get on disk inode
- modify it as specified in ‘ip’
- modify disk block of inode

itrunc , iput

- **iput(ip)**
 - if ref is 1
 - itrunc(ip)
 - type = 0
 - iupdate(ip)
 - i->valid = 0 // free in memory
 - else
- **itrunc(ip)**
 - write all data blocks of inode to disk
 - using bfree()
 - ip->size = 0
 - Inode is freed from use
 - iupdate(ip)

race in iput ?

- A concurrent thread might be waiting in ilock to use this inode

- and won't be prepared to find the inode is not longer allocated

```
void  
iput(struct inode *ip)  
{  
    acquireSleep(&ip->lock);  
    if(ip->valid && ip->nlink == 0)  
        free_inode(ip);  
}
```

- This is not possible.

buffer and inode cache

- to read an **inode**, its block must be read in a buffer
- So the buffer always contains a copy of the on-disk **dinode**
 - duplicate copy in in-memory **inode**
- The **inode cache** is write-through,
 - code that modifies a cached inode must immediately write it to disk with **iupdate**
- Inode may still exist in the buffer cache

allocating inode

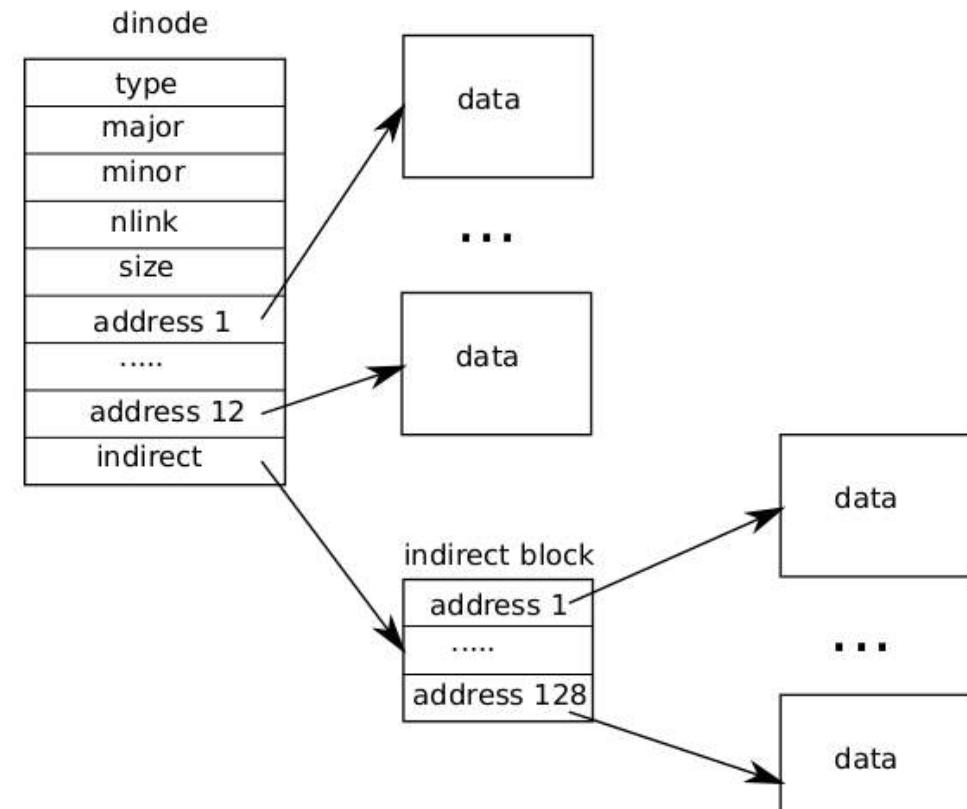
- **ialloc(dev, type)**
 - Loop over all disk inodes
 - read inode (from its block)
 - if it's free (note inum)
 - zero on disk inode
- **ilock**
 - code must acquire ilock before using inode's data/fields
 - **Ilock reads inode if it's already not in memory**

Trouble with iput() and crashes

- **iput() doesn't truncate a file immediately when the link count for the file drops to zero, because**
 - some process might still hold a reference
- **if a crash happens before the last process closes the file descriptor for the file,**
 - then the file will be marked allocated on disk but no directory

Get Inode data: bmap(ip, bn)

- Allocate ‘bn’th block for the file given by inode ‘ip’
- Allocate block on disk and store it in either direct entries or block of indirect entries



writing/reading data at a given offset in file

```
readi(struct inode *ip,  
char *dst, uint off, uint  
n)
```

```
writei(struct inode *ip,  
char *src, uint off, uint  
n)
```

- Calculate the block number in file where ‘off’ belongs
- Read sufficient blocks to read ‘n’ bytes
- using bread(), brelse()

Let's see block allocation layer

System Calls	open, read, write, close, link, pipe, mknod, unlink, fstat, mkdir, chdir, dup,
File descriptor	file.c fileinit, filealloc, filedup, fileclose, filestat, fileread, filewrite,
Pathname	fs.c namex, namei, nameiparent, skipellem
Directory	fs.c dirlookup, dirlink
Inode	fs.c iinit, ialloc, iupdate, ige, idup, ilock, iunlock, iput, iunlockput, itrunc, stati, readi, writei, bmap ,
Logging	Block allocation on disk: balloc, bfree log.c : begin_op, end_op, initlog, commit,
Buffer cache	bio.c binit, bget, bread, bwrite, brelse

allocating & deallocating blocks on DISK

- **balloc(devno)**
 - looks for a block whose bitmap bit is zero, indicating that it is free.
 - On finding updates the bitmap and returns the block.
- **bfree(devno, blockno)**
 - finds the right bitmap block and clears the right bit.
 - Also calls log_write()

Let's see logging layer

System Calls	open, read, write, close, link, pipe, mknod, unlink, fstat, mkdir, chdir, dup,
File descriptor	file.c fileinit, filealloc, filedup, fileclose, filestat, fileread, filewrite,
Pathname	fs.c namex, namei, nameiparent, skipellem
Directory	fs.c dirlookup, dirlink
Inode	fs.c iiinit, ialloc, iupdate, ige, idup, ilock, iunlock, iput, iunlockput, itrunc, stati, readi, writei, bmap ,
Logging	Block allocation on disk: balloc , bfree log.c : begin_op , end_op , initlog , commit ,
Buffer cache	

Recovery

- **Problem. Consider creating a file on ext2 file system.**

- Following on disk data structures will/may get modified
 - Directory data block, new directory data block, block bitmap, inode table,

Recovery

- **Consistency checking (e.g. fsck command) – compares data in directory structure with data blocks on disk, and tries to fix inconsistencies**
 - Can be slow and sometimes fails

Recovery

- Is a critical problem!
- Downtime is un-desired!
- A attempt at the solution: log structured / journaling file systems, e.g. ext3

Log structured file systems

- Log structured (or journaling) file systems record each metadata update to the file system as a transaction
- All transactions are written to a log

Journaling file systems

- ❑ Veritas FS
- ❑ Ext3, Ext4
- ❑ Xv6 file system!

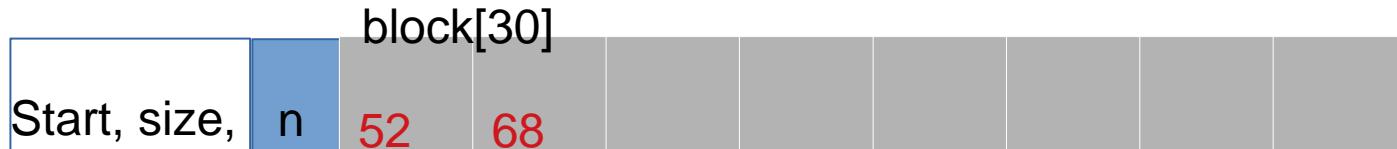
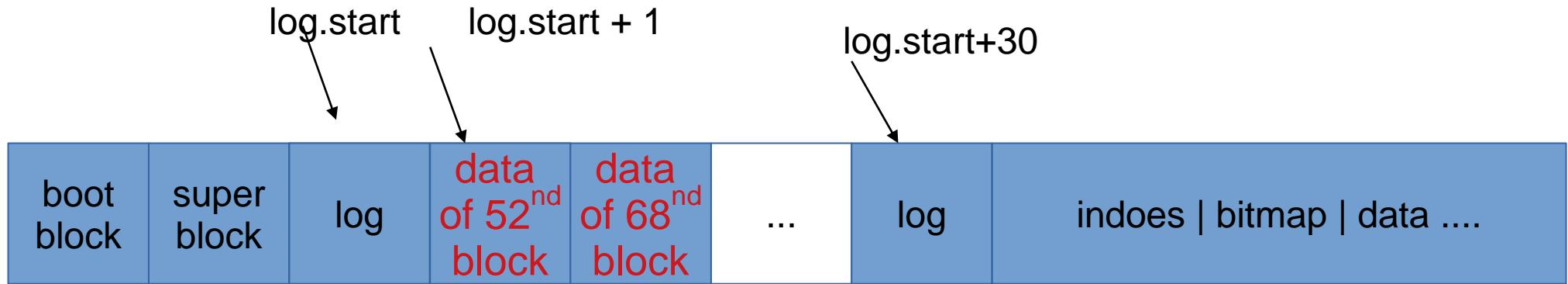
log in xv6

- a mechanism of recovery from disk
- Concept: multiple write operations needed for system calls (e.g. ‘open’ system call to create a file in a directory)
 - some writes succeed and some don’t
 - leading to inconsistencies on disk
- In the log, all changes for a ‘transaction’ (an

log in xv6

- ❑ xv6 system call does not directly write the on-disk file system data structures.
- ❑ A system call calls begin_op() at beginning and end_op() at end
 - ❑ begin_op() increments log.outstanding
 - ❑ end_op() decrements log.outstanding, and if it's 0, then calls commit()

log on disk



log

```
struct logheader { // ON DISK
    int n; // number of entries in use in block[]
    below

    int block[LOGSIZE]; // List of block numbers
    stored

};

struct log { // only in memory
```

Typical use case of logging

```
/* In a system call  
code */  
begin_op();  
...  
bp = bread(...);  
bp->data[...] = ...;
```

prepare for logging.
Wait if logging system
is not ready or
'committing'.

++outstanding

read and get access
to a data block – as a
buffer

Example of calls to logging

```
//file_write() code  
  
begin_op();  
  
ilock(f->ip);  
  
/*loop */ r = writei(f->ip,  
...);  
  
iunlock(f->ip);
```

- each writei() in turn calls bread(), log_write() and brelse()
- also calls iupdate(ip) which also calls bread, log_write and brelse

Logging functions

- **Initlog()**
 - Set fields in global **log.xyz** variables, using FS superblock
 - Recovery if needed
 - Called from first **forkret()**
 - **write_log(void)**
 - Called only from **commit()**
 - Use block numbers specified in **log.lh.block** and copy those blocks from memory to log-blocks
- Following three**

pipes

```
struct pipe {  
    struct spinlock lock;  
    char data[PIPE_SIZE];  
    uint nread;  
    // number of bytes  
    read
```

- **functions**

- pipealloc
- pipeclose
- piperead
- pipewrite

pipes

- **pipealloc**

- allocate two struct file
- allocate pipe itself using kalloc (it's a big structure with array)
- init lock
- initialize both struct file as 2 ends (r/w)

- **pipewrite**

- wait if pipe full
- write to pipe
- wakeup processes waiting to read

- **piperead**

- wait if no data

Further to reading system call code now

- **Now we are ready to read the code of system calls on file system**
 - sys_open, sys_write, sys_read , etc.
- **Advise: Before you read code of these, contemplate on what these functions should do using the functions we have studied so far.**

VFS

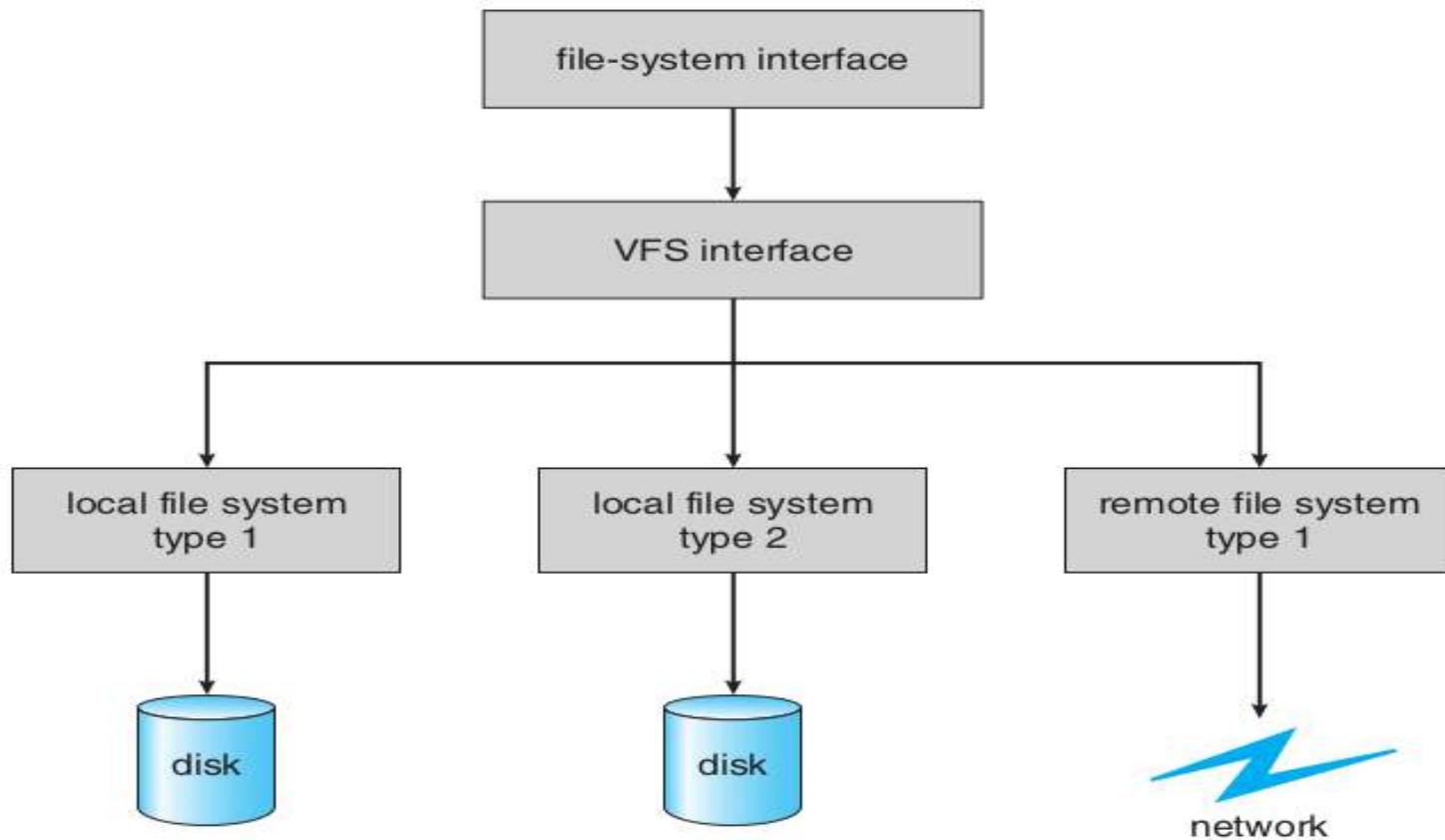


Figure 15.5 Schematic view of a virtual file system.

VFS

- Consider this

/dev/sda1 is “/”

/dev/sda2 is mounted on “/a/b” folder

How does this work in kernel?

`open(“/a/b/c/d”, O_RDONLY)`

- Consider xv6 code

- `sys_open -> namei -> namex -> (skipelem,`

VFS

- Object Oriented Programming in C (let's see example of this)
 - Clever use of function pointers
- There is an “abstract” file system class (VFS), and there are concrete

```
struct inode_operations
{
    int (*readi) (int, char *, int);
    int (*writei) (int, char *, int);
    ....
}
```

struct inode {

**Efficiency and Performance
(and the risks created
while trying to achieve it!)**

Efficiency

- **Efficiency dependent on:**
 - Disk allocation and directory algorithms
 - Types of data kept in file's directory entry
 - Pre-allocation or as-needed allocation of metadata structures
 - Fixed-size or varying-size data structures

Performance

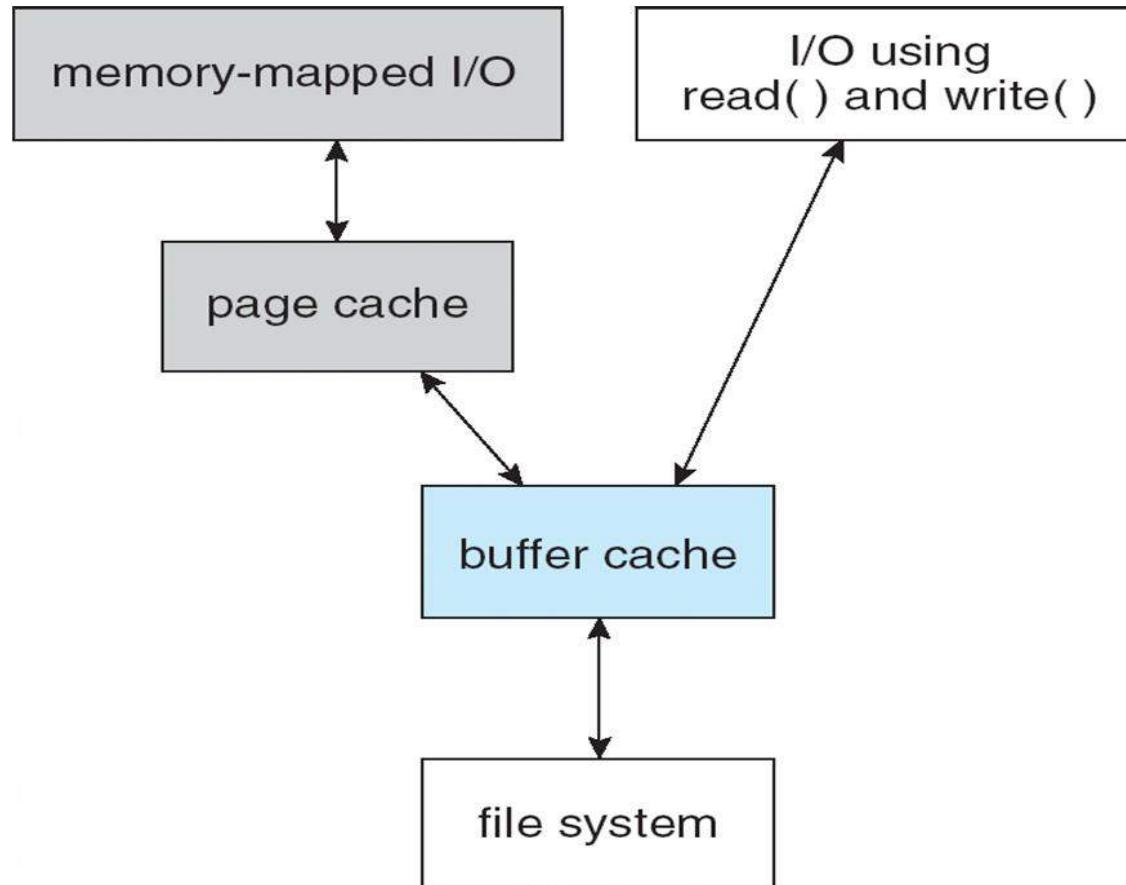
- Keeping data and metadata close together
- Buffer cache – separate section of main memory for frequently used blocks
- Synchronous writes sometimes requested by apps or needed by OS
- No buffering / caching – writes must hit disk before acknowledgement

Asynchronous writes more common, buffer

Page cache

- A page cache caches pages rather than disk blocks using virtual memory techniques and addresses
- Memory-mapped I/O uses a page cache
- Routine I/O through the file system uses the buffer (disk) cache
- This leads to the following figure

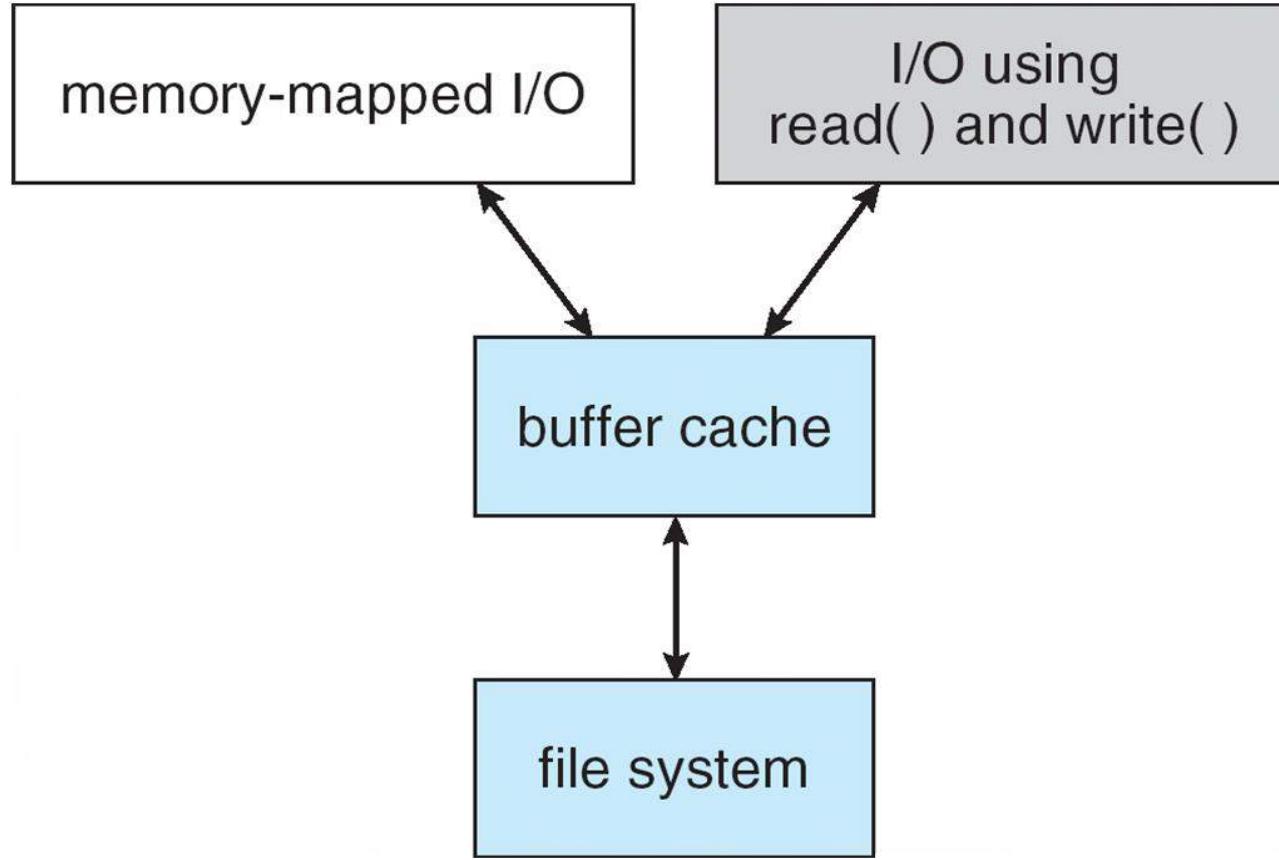
I/O Without a Unified Buffer Cache



Unified buffer cache

- A unified buffer cache uses the same page cache to cache both memory-mapped pages and ordinary file system I/O to avoid double caching
- But which caches get priority, and what replacement algorithms to use?

I/O Using a Unified Buffer Cache



Recovery

- **Problem. Consider creating a file on ext2 file system.**
 - Following on disk data structures will/may get modified
 - Directory data block, new directory data block, block bitmap, inode table, inode table bitmap, group descriptor, super block, data blocks for new file, more data block bitmaps, ...
 - All cached in memory by OS

Recovery

- **fsck: Consistency checking – compares data in directory structure with data blocks on disk, and tries to fix inconsistencies**
 - Can be slow and sometimes fails
- **Use system programs to back up data from disk to another storage device (magnetic tape, other magnetic disk, optical)**
- **Recover lost file or disk by restoring data**

Log structured file systems

- Log structured (or journaling) file systems record each metadata update to the file system as a transaction
- All transactions are written to a log
 - A transaction is considered committed once it is written to the log (sequentially)
 - Sometimes to a separate device or section of disk
 - However, the file system may not yet be updated

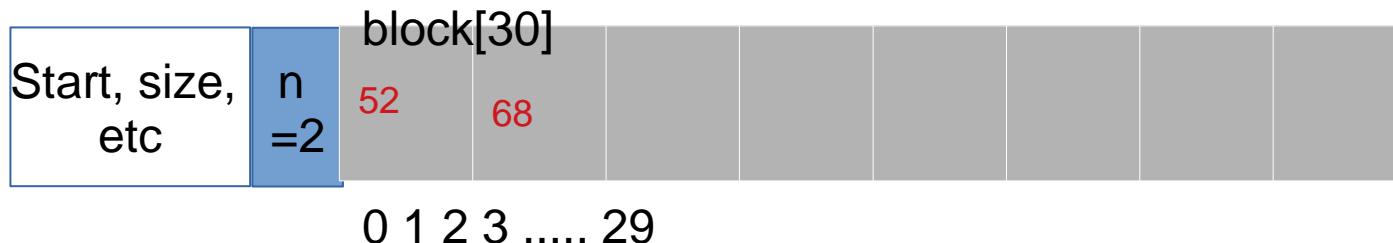
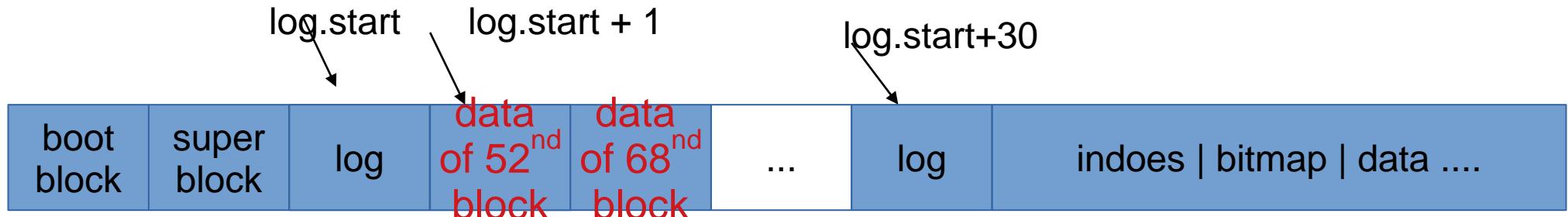
Journaling file systems

- Veritas FS
- Ext3, Ext4
- Xv6 file system!

log in xv6

- a mechanism of recovery from disk
- Concept: multiple write operations needed for system calls (e.g. ‘open’ system call to create a file in a directory)
 - some writes succeed and some don’t
 - leading to inconsistencies on disk
- In the log, all changes for a ‘transaction’ (an operation) are either written completely or

log on disk



log in xv6

- ❑ xv6 system call does not directly write the on-disk file system data structures.
- ❑ A system call calls begin_op() at begining and end_op() at end
 - ❑ begin_op() increments log.outstanding
 - ❑ end_op() decrements log.outstanding, and if it's 0, then calls commit()
- ❑ During the code of system call, whenever a

log

```
struct logheader { // ON DISK
    int n; // number of entries in use in block[]
    below

    int block[LOGSIZE]; // List of block numbers
    stored

};

struct log { // only in memory
```

Typical use case of logging

```
/* In a system call  
code */
```

```
begin_op();
```

```
...
```

```
bp = bread(...);
```

```
bp->data[...] = ...;
```

```
log_write(bp);
```

prepare for logging.
Wait if logging system
is not ready or
'committing'.
++outstanding

read and get access
to a data block – as a
buffer

match colors in code and comments on right-side

Example of calls to logging

```
//file_write() code  
begin_op();  
ilock(f->ip);  
/*loop */ r = writei(f->ip,  
...);  
iunlock(f->ip);  
end_op();
```

- each writei() in turn calls bread(), log_write() and brelse()
- also calls iupdate(ip) which also calls bread, log_write and brelse
- Multiple writes are

Logging functions

- **Initlog()**
 - Set fields in global **log.xyz** variables, using FS superblock
 - Recovery if needed
 - Called from first forkret()
- **Following three called by FS code**
- **write_log(void)**
 - Called only from commit()
 - Use block numbers specified in **log.lh.block** and copy those blocks from memory to log-blocks
- **commit(void)**

Scheduling Algorithms

Abhijit A.M.

abhijit.comp@coep.ac.in

Credits: Slides from os-book.com

Calculations of different scheduling criteria

- If you want to evaluate an algorithm practically, you need a proper workload !
 - Processes with CPU and I/O bursts
 - Different durations of CPU bursts
 - Different durations of I/O bursts
 - How to do this programmatically?
 - How to ensure that after 2 seconds an I/O takes place?

Calculations of different criteria

□ CPU Utilization

- % time spent in doing ‘useful’ work
- What is useful work?
 - On linux
 - there is an “idle” thread, scheduled when no other task is RUNNABLE
 - Not running idle thread is productive work
 - Includes process + scheduling time + interrupts
 - On other systems?

Calculations of different criteria

□ Throughput

- # processes that complete execution per unit time
- Formula: total # processes completed / total time
- Simply divide by your total workload that completed by the time taken
- Depends on the workload as well. ‘long’ or ‘short’ processes.

If you have any questions, feel free to ask them in the Q&A section.

Calculations of different criteria

□ Turnaround time

- Amount of time required for one process to complete
- For every process, note down the starting and ending time, difference is TA-time
- For process P1 : (Time when process ended – time when process started)

– Sum of time spent in (ready queue + running

Calculations of different criteria

- **Waiting time**
 - amount of time a process has been waiting in the *ready queue*.
 - To be minimised.
 - Part of Turn Around time
 - CPU scheduling does not affect waiting time in I/O queues, it affects time in ready queue

Scheduling Criteria

- **Response time**
 - amount of time it takes from when a request was submitted until the first response (not full output) is produced, (for time-sharing environment).
 - To be minimised.
 - E.g. time between your press of a key , and that key being shown in screen

Challenges in implementing the scheduling algorithms

- Not possible to know number of CPU and I/O bursts and the duration of each before the process runs !**
 - Although when we do numerical problems around each algorithm, we assume some values for CPU and I/O bursts, so the problems are solved in “hindsight” !**

GANTT chart

- A timeline chart showing the sequence in which processes get scheduled
- Used for analysing a scheduling algorithm

Scheduling Algorithms

First- Come, First-Served (FCFS) Scheduling

Process	Burst Time
---------	------------

P1	24
----	----

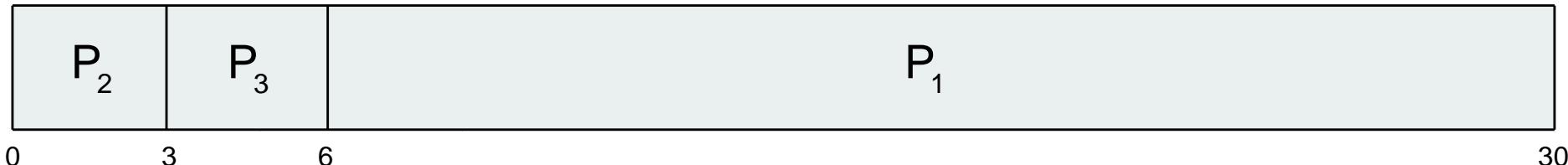
D2	2
----	---



P3	3
----	---

FCFS Scheduling (Cont.)

Suppose that the processes arrive in the order:



The Gantt chart for the schedule is:

FCFS: Convoy effect

- Consider one CPU-bound and many I/O-bound processes
- CPU bound process over, goes for I/O. I/O bound processes run quickly, move

FCFS: further evaluation

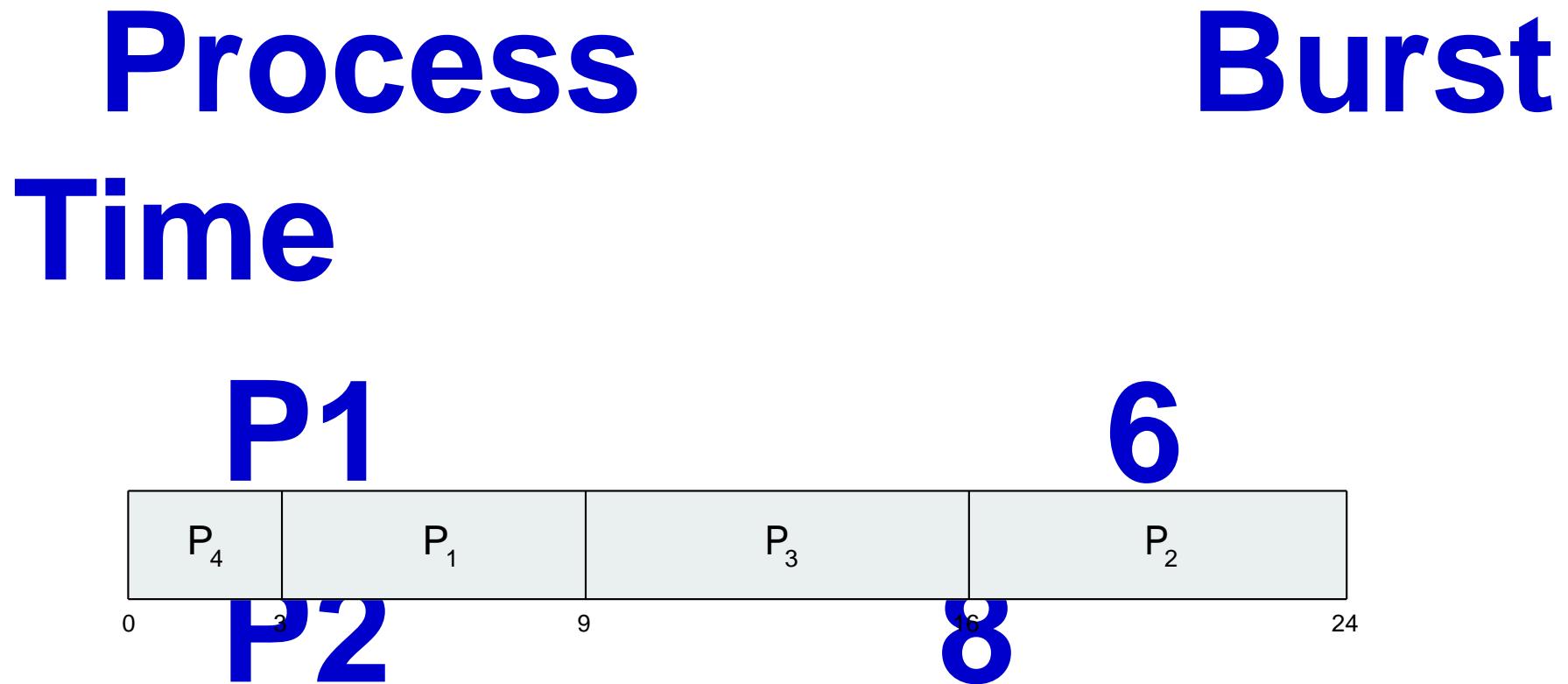
- **Troublesome for interactive processes**
 - CPU bound process may hog CPU
 - Interactive process may not get a chance to run early and response time may be quite bad

Shortest-Job-First (SJF) Scheduling

- **Associate with each process the length of its next CPU burst**
 - Use these lengths to schedule the process with the shortest time. Better name – **Shortest Next CPU Burst Scheduler**
- **SJF is optimal – gives minimum average waiting time for a given set of processes**

The difficulty is knowing the length of the next CPU

Example of SJF



Determining Length of Next CPU Burst

- Not possible to implement SJF as can't know "next" CPU burst. Can only estimate the length – should be similar to the previous one

- Then pick process with shortest predicted next CPU burst

1. t_n = actual length of n^{th} CPU burst

2. T_{n+1} = predicted value for the next CPU burst

3. $\alpha, 0 \leq \alpha \leq 1$

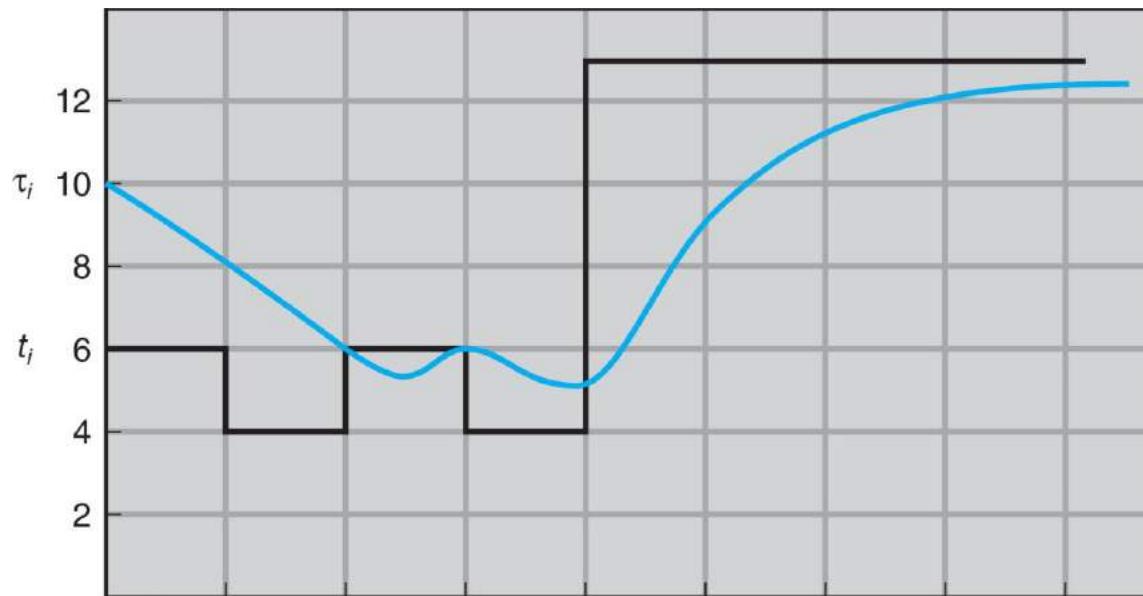
Can be done by using the length of

Examples of Exponential Averaging

- $\alpha = 0$
 - $\tau_{n+1} = \tau_n$
 - Recent history does not count
- $\alpha = 1$
 - $\tau_{n+1} = \alpha t_n$
 - Only the actual last CPU burst counts

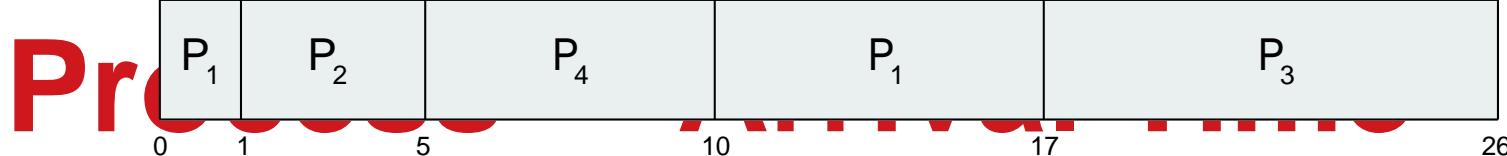
Prediction of the Length of the Next CPU Burst

- $\alpha = 1/2$, $\tau_0 = 10$



Example of Shortest-remaining-time-first

Preemptive SJF = SRTF. Now we add the concepts of varying arrival times and preemption to the analysis



Round Robin (RR) Scheduling

- **Each process gets a small unit of CPU time (time quantum q), usually 10-100 milliseconds.**
 - After this time has elapsed, the process is preempted and added to the end of the ready queue.
- **If there are n processes in the ready queue and the time quantum is q, then each**

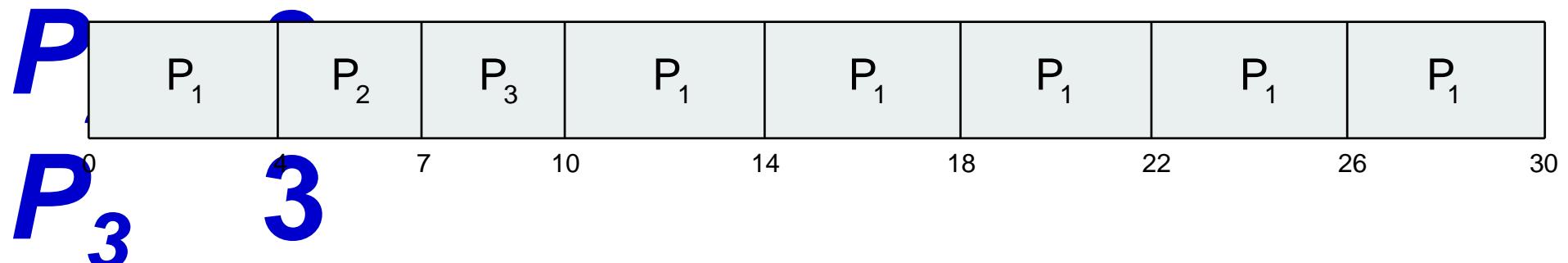
Round Robin (RR) Scheduling

- Timer interrupts every quantum to schedule next process
- Performance
 - q large \Rightarrow FIFO
 - q small \Rightarrow q must be large with respect to context switch, otherwise overhead is too high

Example of RR with Time Quantum = 4

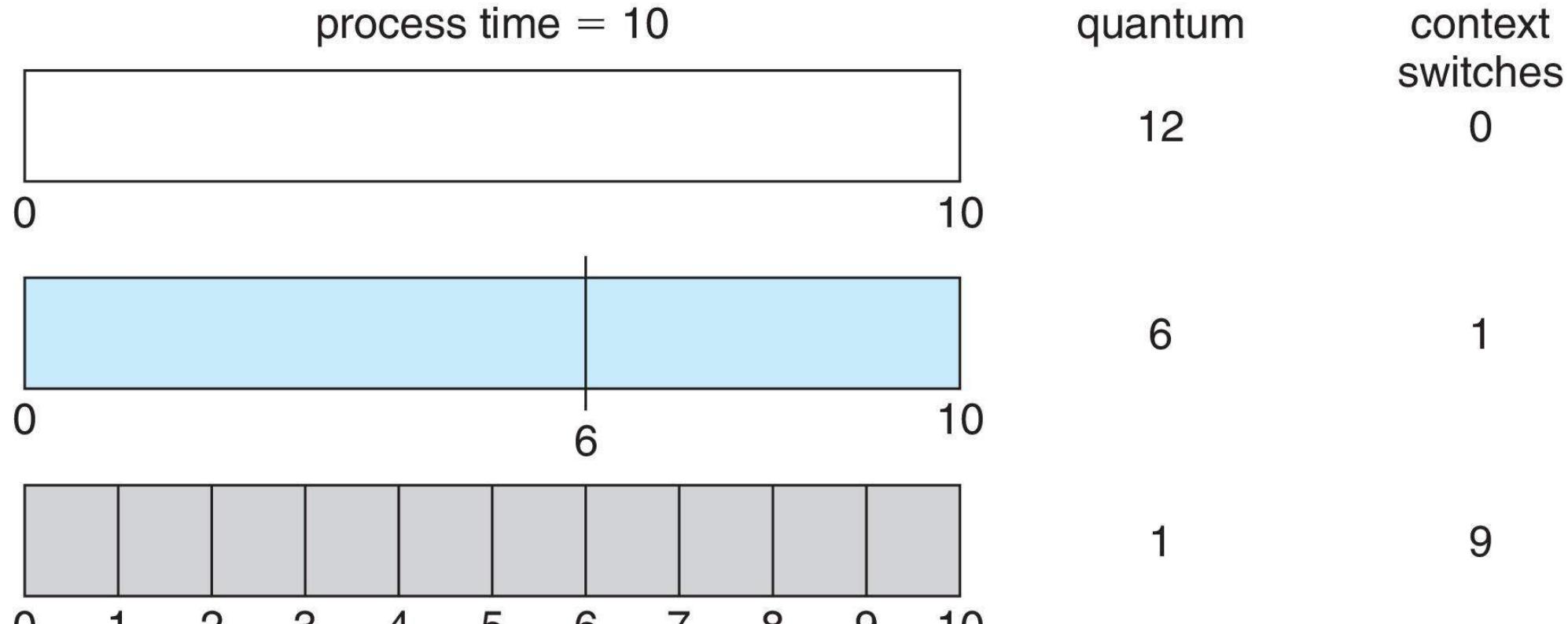
<u>Process</u>	<u>Burst Time</u>
----------------	-------------------

P_1	24
-------	----

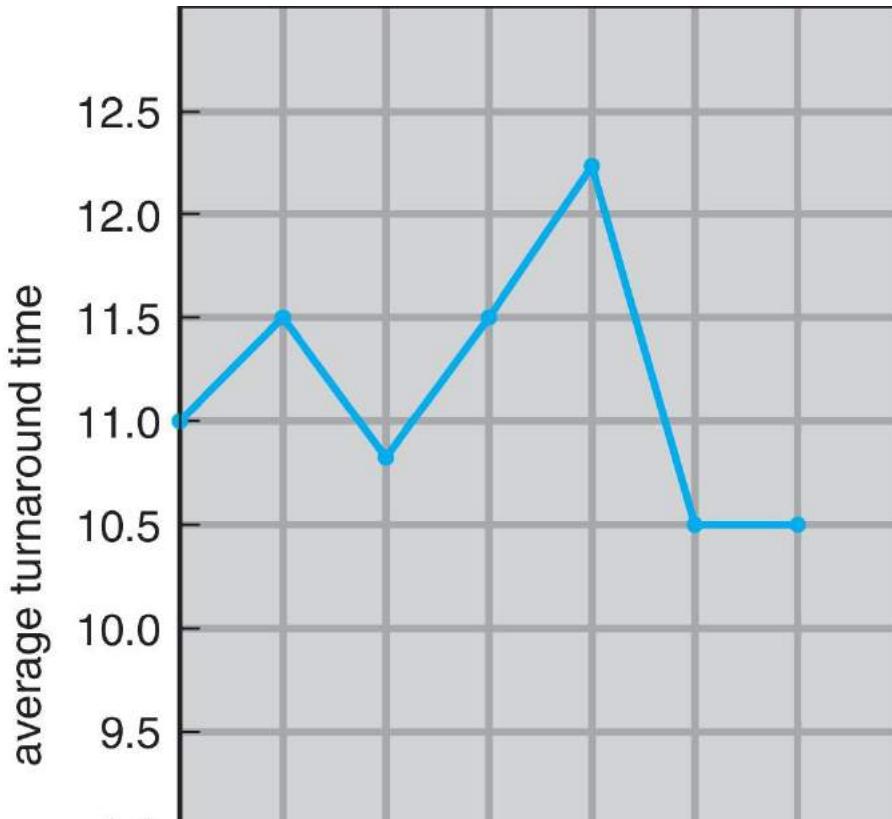


The Gantt chart is:

Time Quantum and Context Switch Time



Turnaround Time Varies With The Time Quantum



process	time
P_1	6
P_2	3
P_3	1
P_4	7

80% of CPU bursts should

Priority Scheduling

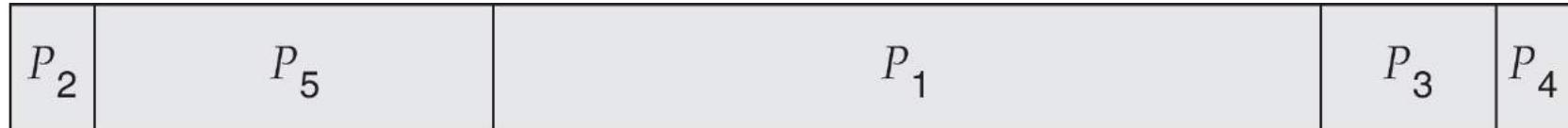
- A priority number (integer) is associated with each process
- The CPU is allocated to the process with the highest priority (smallest integer \equiv highest priority)

Preemptive (timer interrupt, more time)

Example of Priority Scheduling

	<u>Process</u>	<u>Burst Time</u>	<u>Priority</u>
□	P_1	10	3
□	P_2	1	1
□	P_3	2	4
□	P_4	1	5
□	P_5	5	2

- Priority scheduling Gantt Chart



Priority Scheduling with Round-Robin

<u>Process</u>	<u>Burst Time</u>	<u>Priority</u>
P_1	4	3
P_2	5	2
P_3	8	2
P_4	7	1
P_5	3	3

Run the process with the highest priority. Processes with the same priority run round-robin

Gantt Chart with 2 ms time quantum

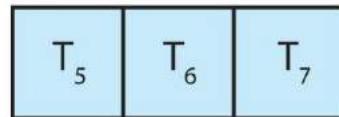
Multilevel Queue

- With priority scheduling, have separate queues for each priority.
- Schedule the process in the highest-priority queue!

priority = 0



priority = 1

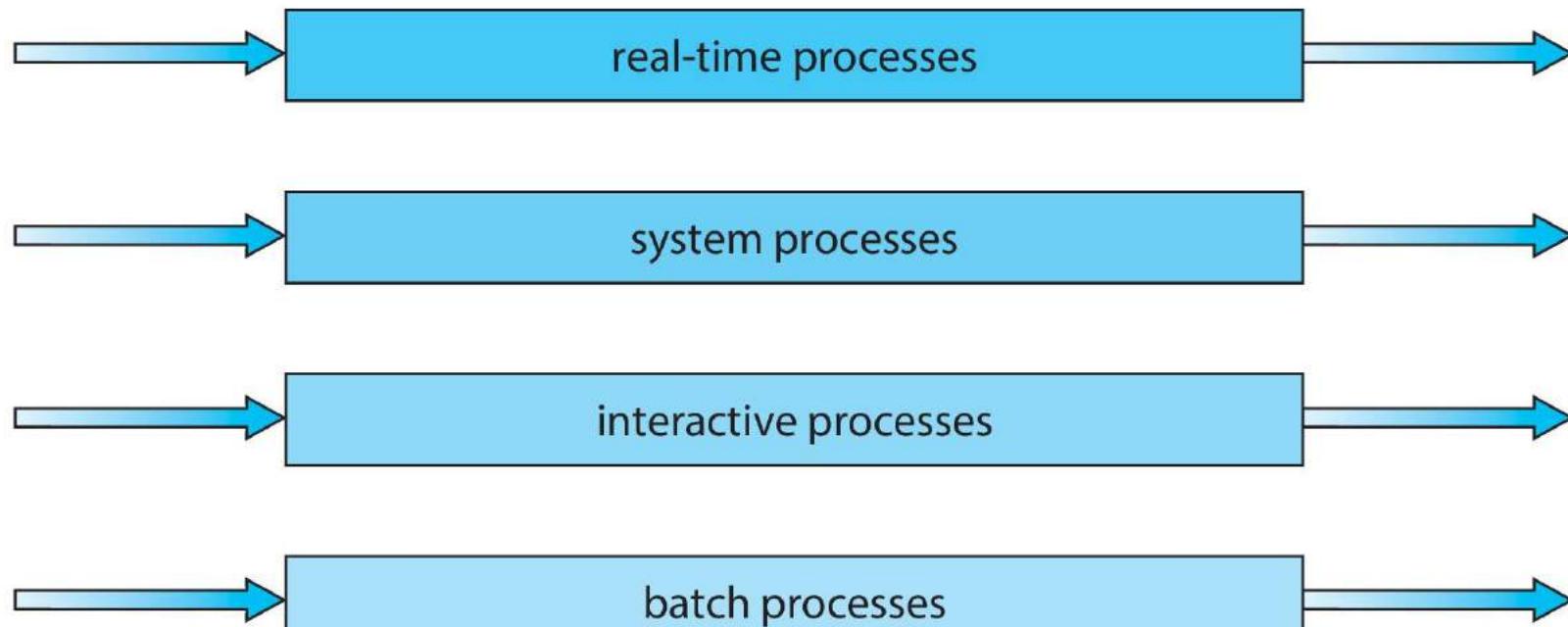


priority = 2



Multilevel Queue

highest priority



Implementing multilevel queue

- **Processes need to have a priority**
 - Either modify fork()/exec() to have a priority
 - Or add a nice() system call to set priority
- **How to know the priority?**
 - The end user of the computer system needs to know this from needs of real life
 - E.g. on a database system, the database process

Multilevel Feedback Queue

- A process can move between the various queues; aging can be implemented this way
- Multilevel-feedback-queue scheduler defined by the following parameters:
 - number of queues
 - scheduling algorithms for each queue
 - method used to determine when to upgrade a

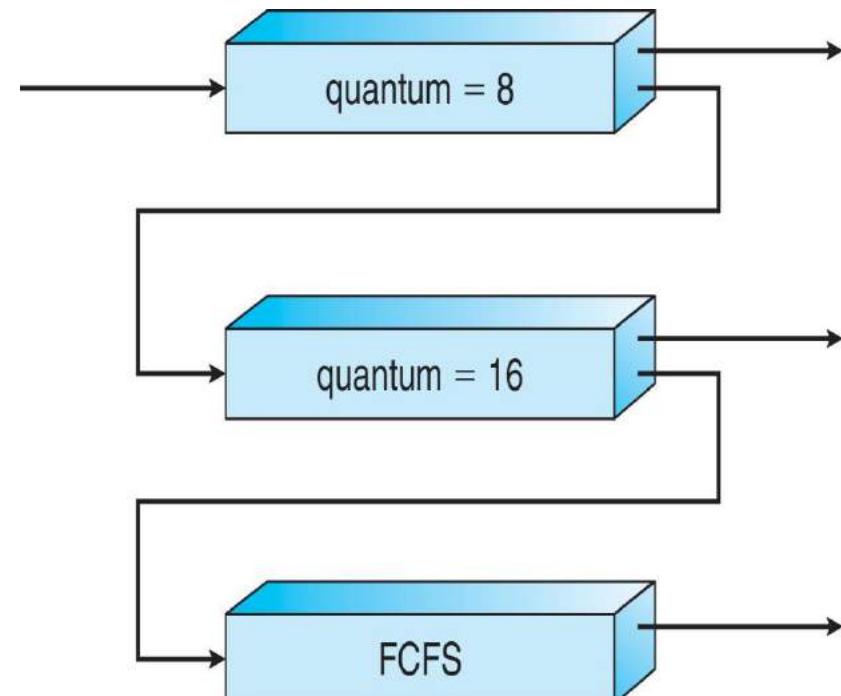
Example of Multilevel Feedback Queue

- **Three queues:**

- Q_0 – RR with time quantum 8 milliseconds
- Q_1 – RR time quantum 16 milliseconds
- Q_2 – FCFS

- **Scheduling rules**

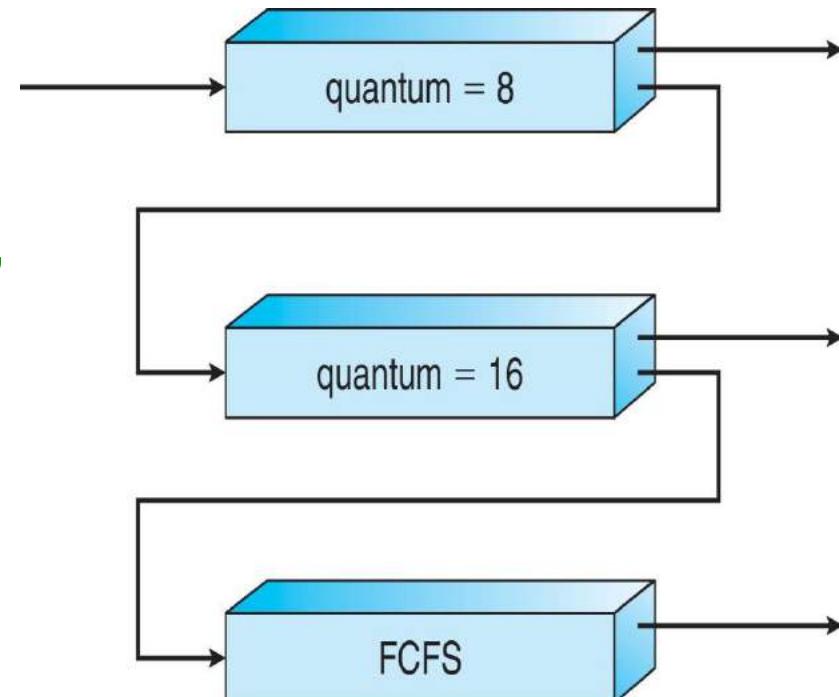
- Serve all processes in Q_0 first
- Only when Q_0 is empty, serve



Example of Multilevel Feedback Queue

Scheduling

- A new job enters queue Q_0
 - When it gains CPU, job receives 8 milliseconds
 - If it does not finish in 8 milliseconds, job is moved to queue Q_1
- At Q_1 , job receives 16 additional milliseconds
 - If it still does not complete, it is preempted and moved to queue Q_2
- To prevent starvation, move a



Thread Scheduling

- **Distinction between user-level and kernel-level threads**
- **When threads supported, threads scheduled, not processes**
- **Many-to-one and many-to-many models, thread library schedules user-level threads to run on LWP**

Pthread Scheduling

- ❑ **PTHREAD_SCOPE_PROCESS** schedules threads using PCS scheduling
- ❑ **PTHREAD_SCOPE_SYSTEM** schedules threads using SCS scheduling
- ❑ Linux and macOS only allow **PTHREAD_SCOPE_SYSTEM**
- ❑ Let's see a Demo using a program

Multiple-Processor Scheduling – Load Balancing

- **Soft affinity** – the operating system attempts to keep a thread running on the same processor, but no guarantees.
- **Hard affinity** – allows a process to specify a set of processors it may run on.

End

Device files Volume Manager

Device Files

- See
 - `$ ls -l /dev`
 - “c” character special device files
 - ‘b’ block device files
- Device file represents a hardware device
- Open() and then read(), write() on device files will read/write from the device (if supported)

Device Files

- Major Number and Minor Number
 - The size field in inode is reused as major-minor number field for device files
 - Major number: type of device (identifies the device driver)
 - Minor number: device number of that type (tells the device driver, which device)
- Xv6

Block Device Files

- For “block” devices
 - `read()`,`write()` happen in multiples of “block”
- E.g.
 - `/dev/sda1`
 - `/dev/hda1`
 - `/dev/nvme0p1`

Issues with disk partitions

- Use of disk partitions

```
$ fdisk /dev/sdb # create partitions
```

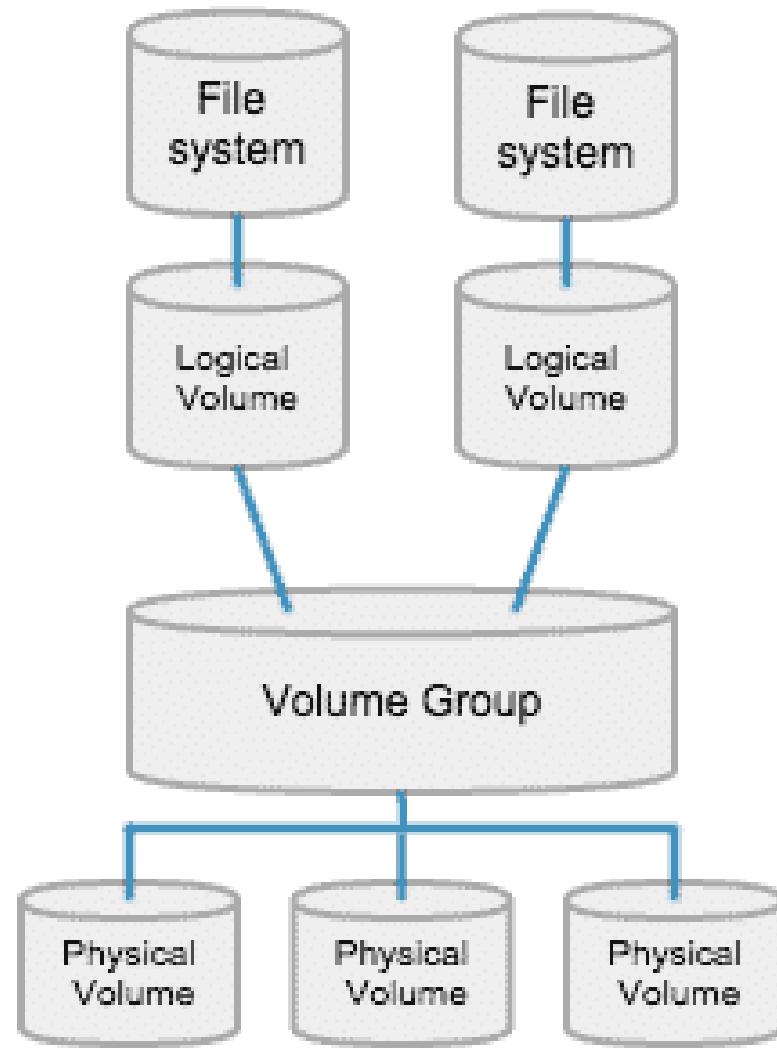
```
$ mkfs -t ext2 /dev/sdb1 # format
```

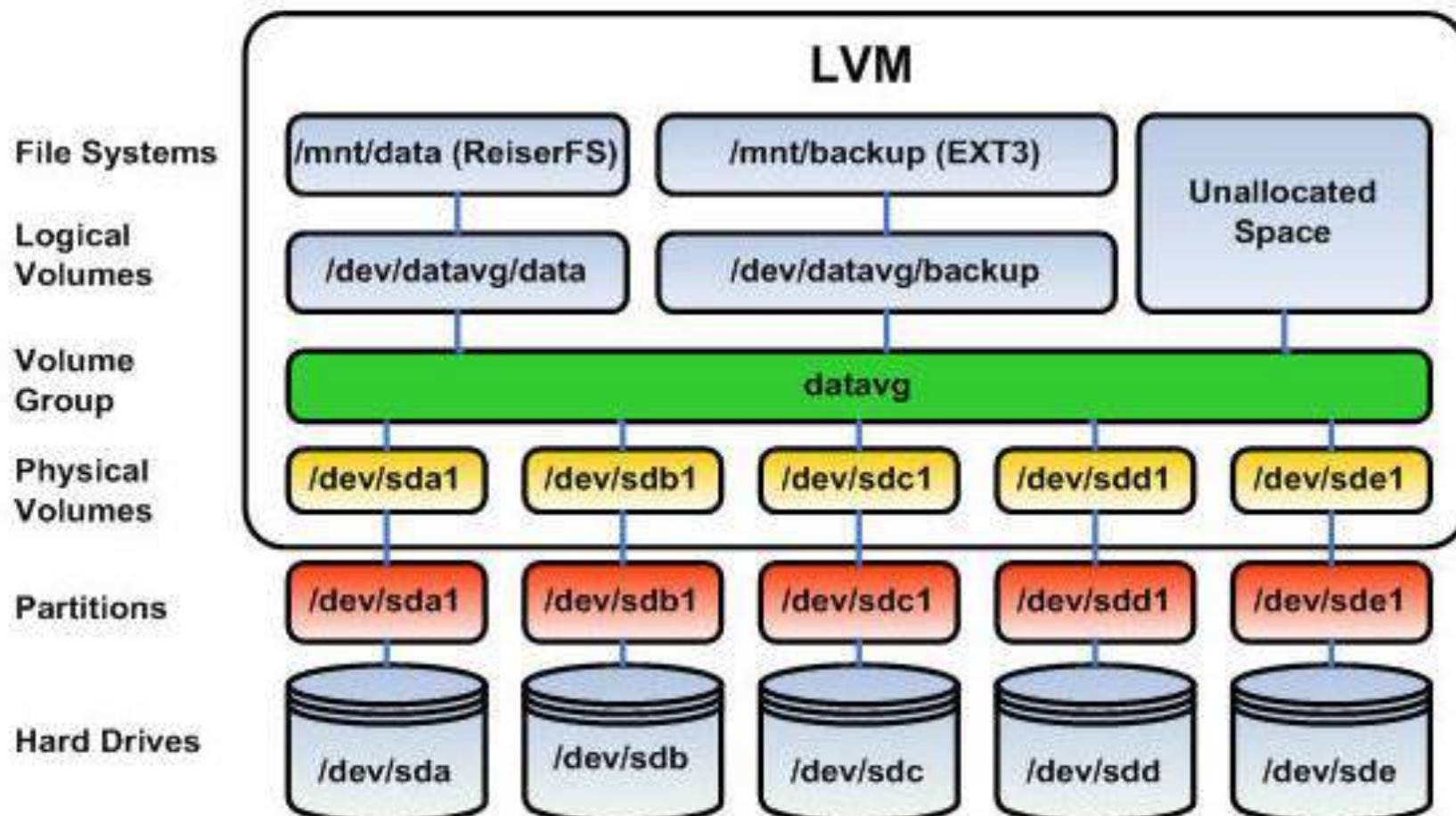
```
$ mount -t ext2 /dev/sdb1 /a/b #  
mounted on /a/b
```

- Fixed size
- Can't easily grow them

Logical Volume Manager (LVM)

- A layer (indirection) between physical partitions (physical volumes) and file system
 - Enables easy grouping , re-grouping, extending, shrinking, ...
- Logical Volume Group:
 - Parallel to a physical disk (but extendible)
- Logical Volume
 - Parallel to a physical partision (but extendable!)





LVM

File Systems
Sistema de arquivos

/home
(ext4)

/data
(xfs)

Logical Volume (LV)
Volume Lógico

/dev/vgroup/lv_home

/dev/vgroup/lv_data

Volume Groups (VG)
Grupos de volumes

vgroup

www.linuxnaweb.com

Physical Volumes (PV)
Volumes físicos

/dev/sdb1

/dev/sdb2

/dev/sdc1

/dev/sdc2

Partitions
Partições

/dev/sdb1
8e LVM

/dev/sdb2
8e LVM

/dev/sdc1
8e LVM

/dev/sdc2
8e LVM

Physical Drives
Drivers físicos



/dev/sdb



/dev/sdc

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 (kinit1, knit2 in xv6)
- Frames are used for allocating memory to a process
 - But may also be used for managing kernel's own data structures also (in xv6, all kernel data is statically allocated)
- 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

Need for Page Replacement

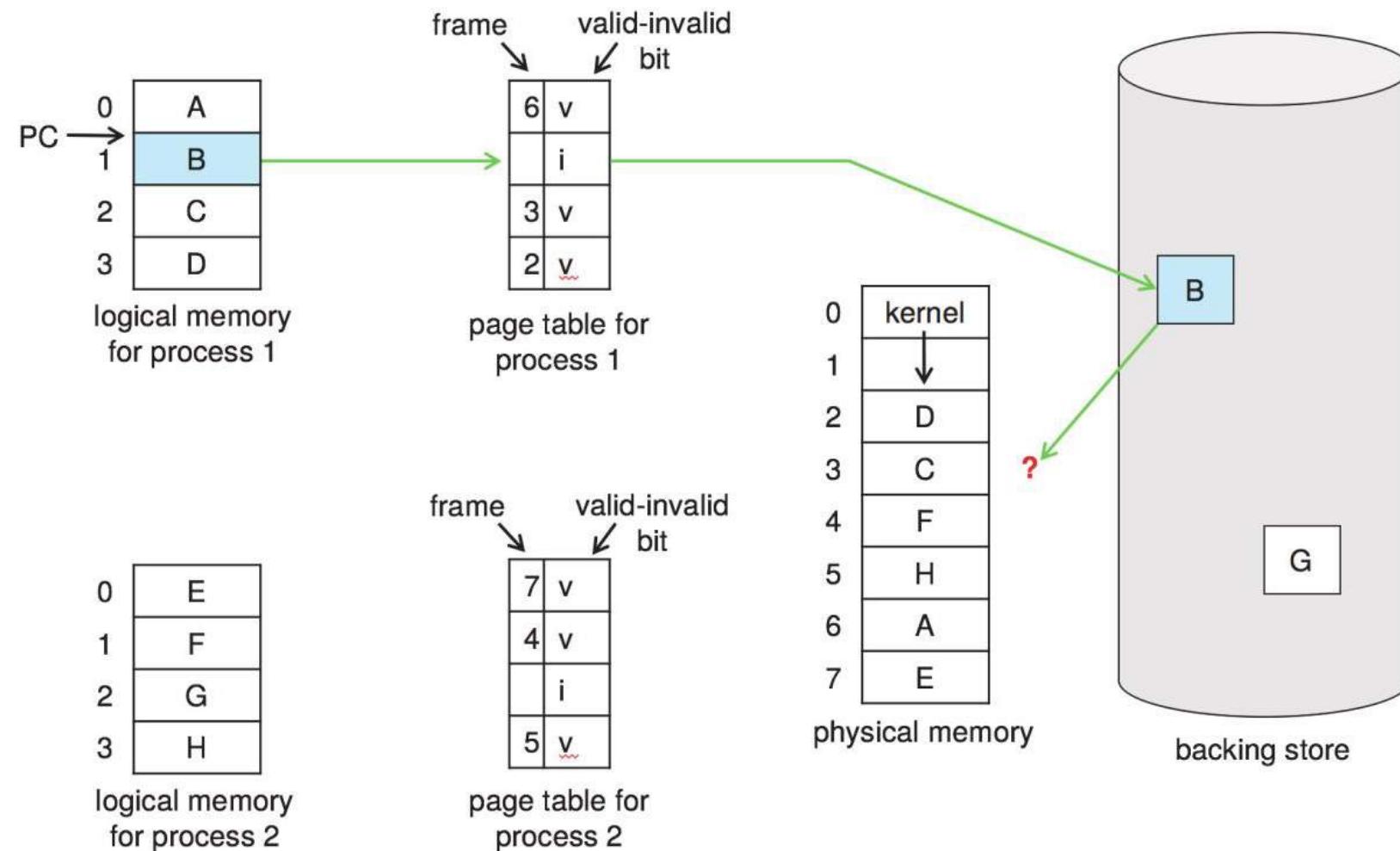


Figure 10.9 Need for page replacement.

Page replacement

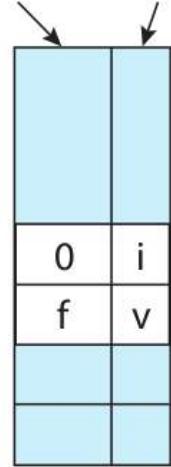
- Strategies for performance
 - Prevent over-allocation of memory by modifying page-fault service routine to include page replacement
 - Use modify (dirty) bit in page table. To reduce overhead of page transfers – only modified pages are written to disk. If page is not modified, just reuse it (a copy is already there in backing store)

Basic Page replacement

- 1) Find the location of the desired page on disk
- 2) Find a free frame:
- 3)- If there is a free frame, use it
- 4)- If there is no free frame, use a page replacement algorithm to select a victim frame & write victim frame to disk if dirty
- 5) Bring the desired page into the free frame; update the page table of process and global frame table/list

Page Replacement

frame valid-invalid bit

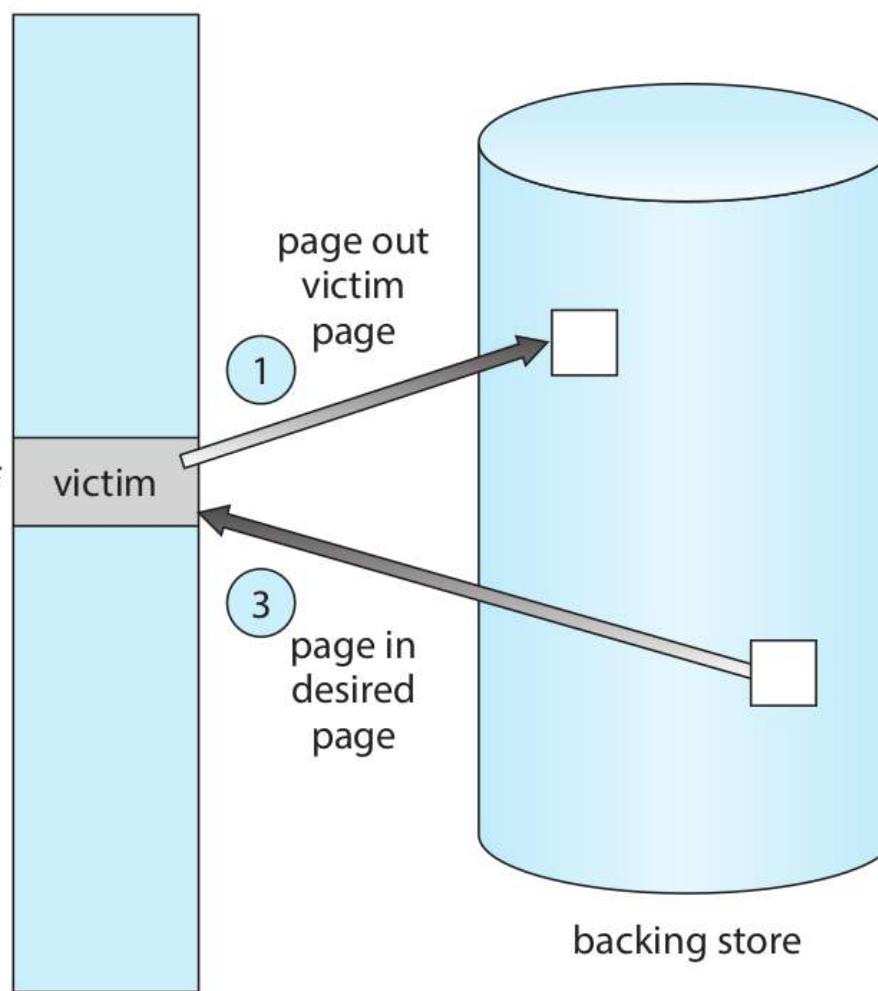


2 change to invalid

4 reset page table for new page

f

physical
memory

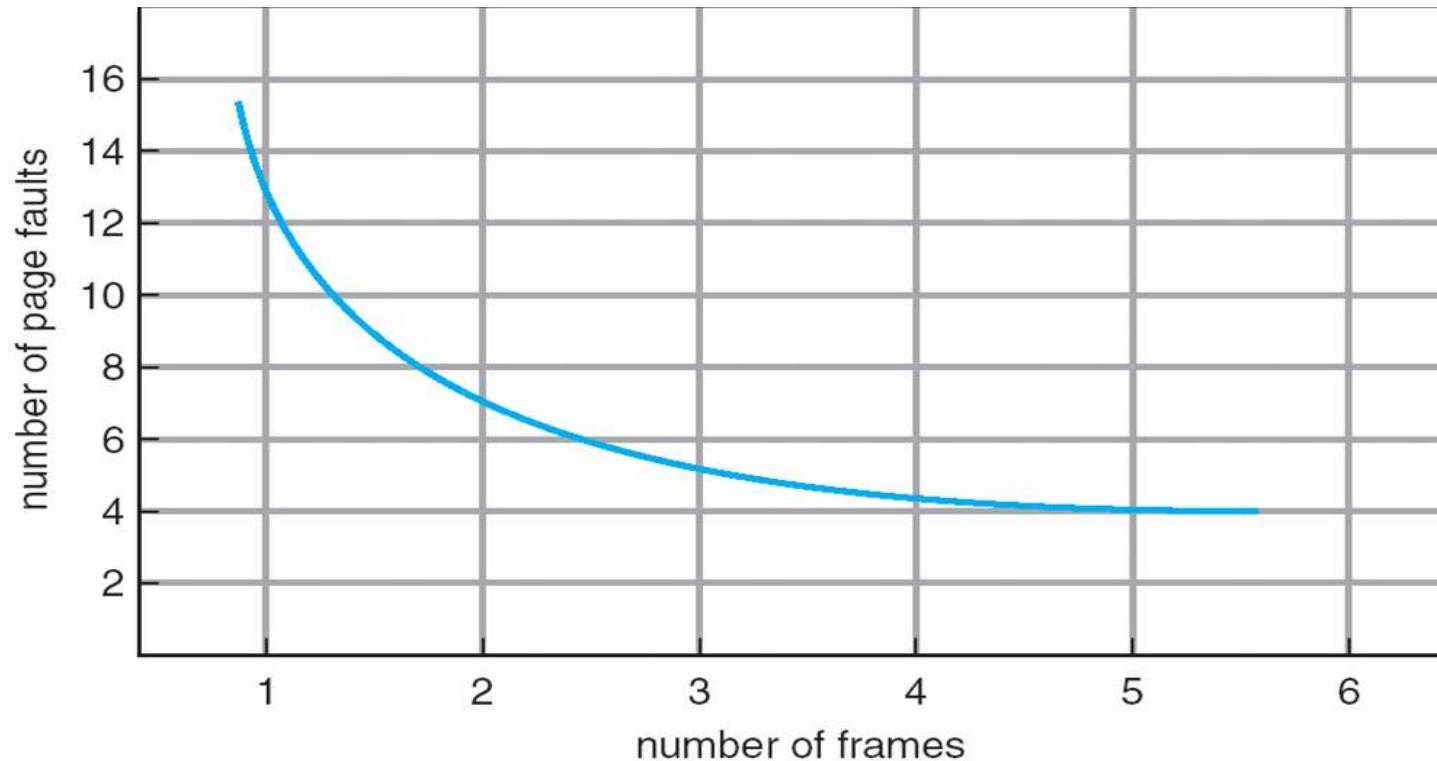


Two problems to solve

- 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



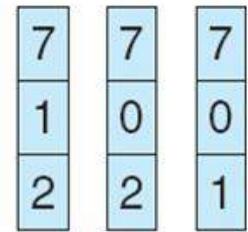
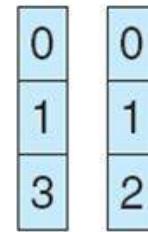
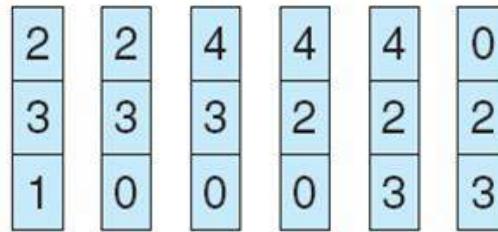
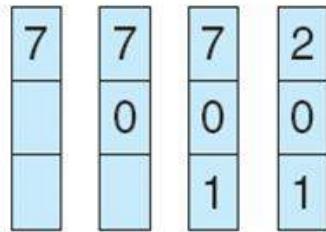
**An
Expectation**

**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

15 page
faults

FIFO Algorithm

1	5	1 4 5
2	3	2 1 3
3	4	3 2 4

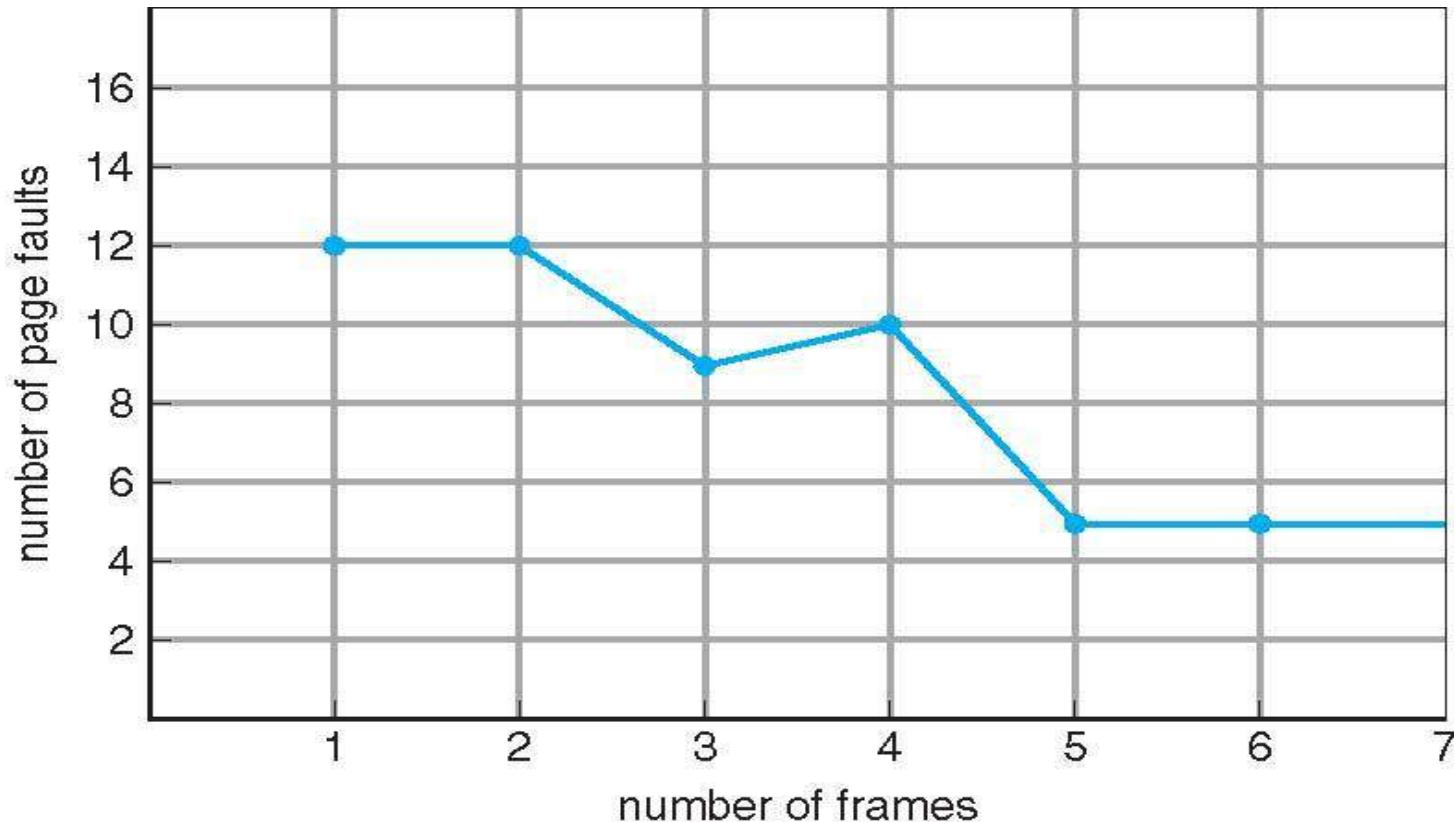
9
PF

1	4	1 5 4
2	5	2 1 5
3	2	3 2
4	3	4 3

10
PF

- 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



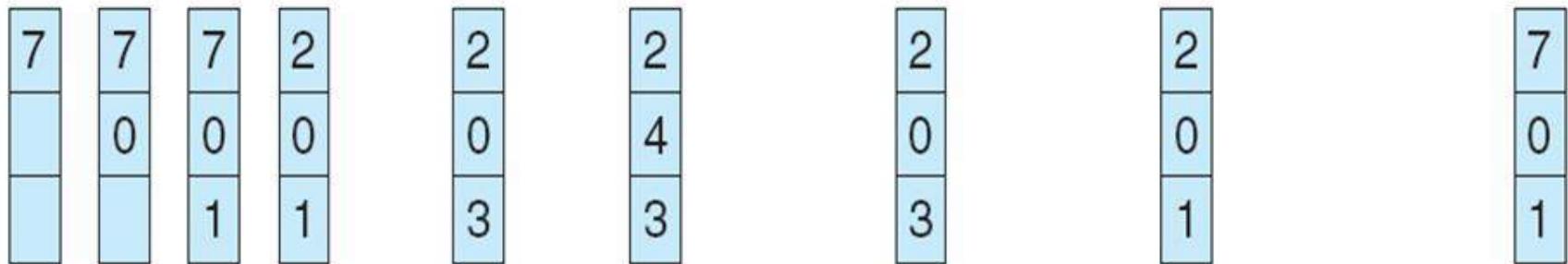
Optimal Algorithm

- 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

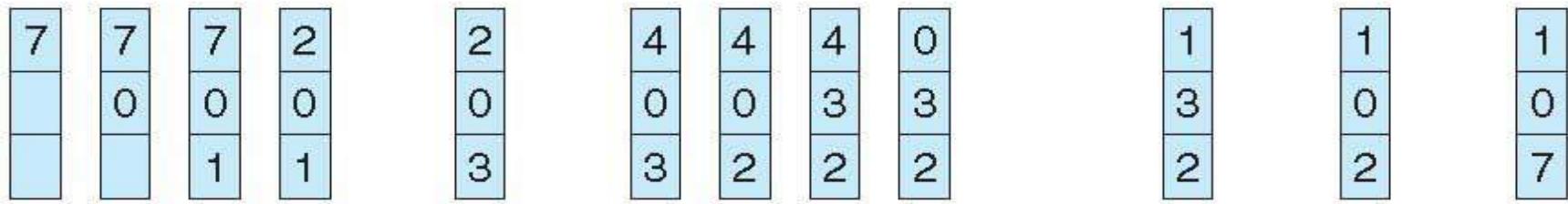


page frames

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

reference string

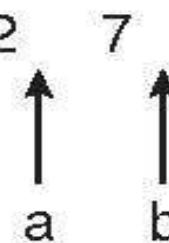
4 7 0 7 1 0 1 2 1 2 7 1 2

2
1
0
7
4

stack
before
a

7
2
1
0
4

stack
after
b



**Use of a
Stack to
Record The
Most Recent
Page
References**

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

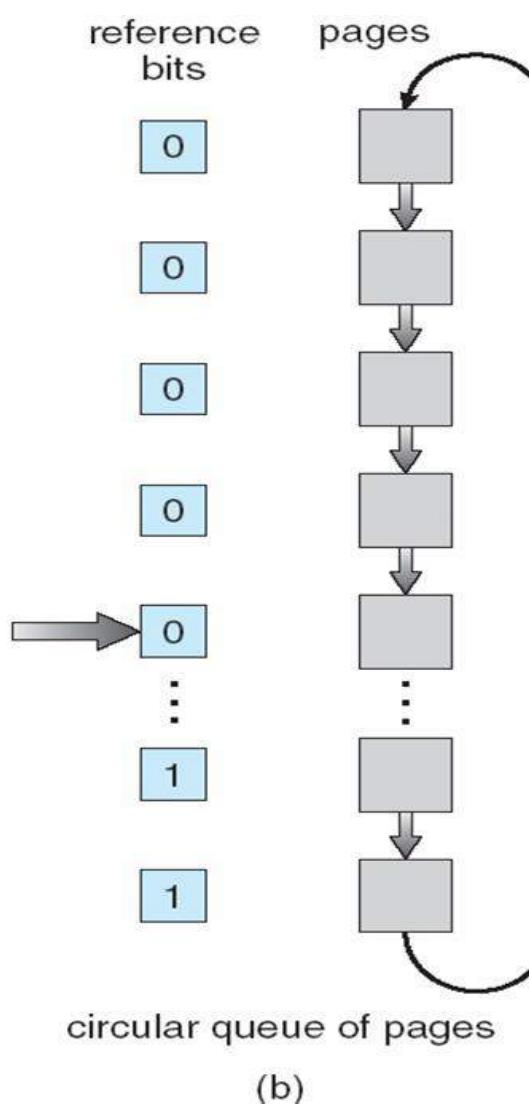
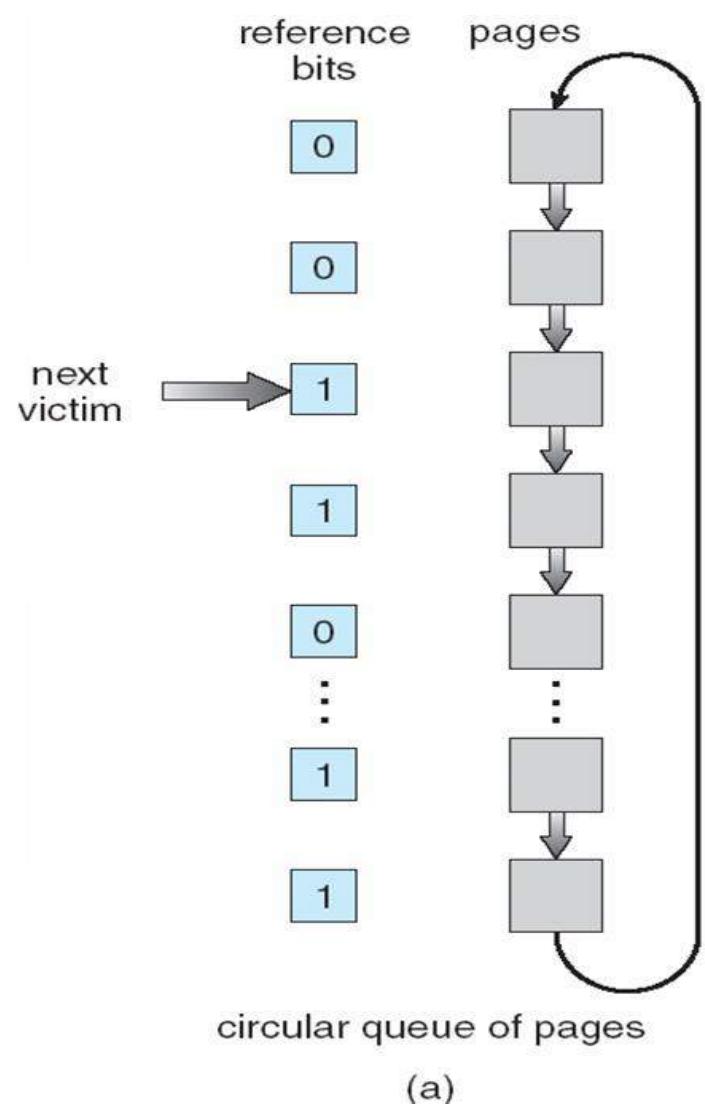
LRU: Approximation algorithms

- 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-

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 frames list

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

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

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

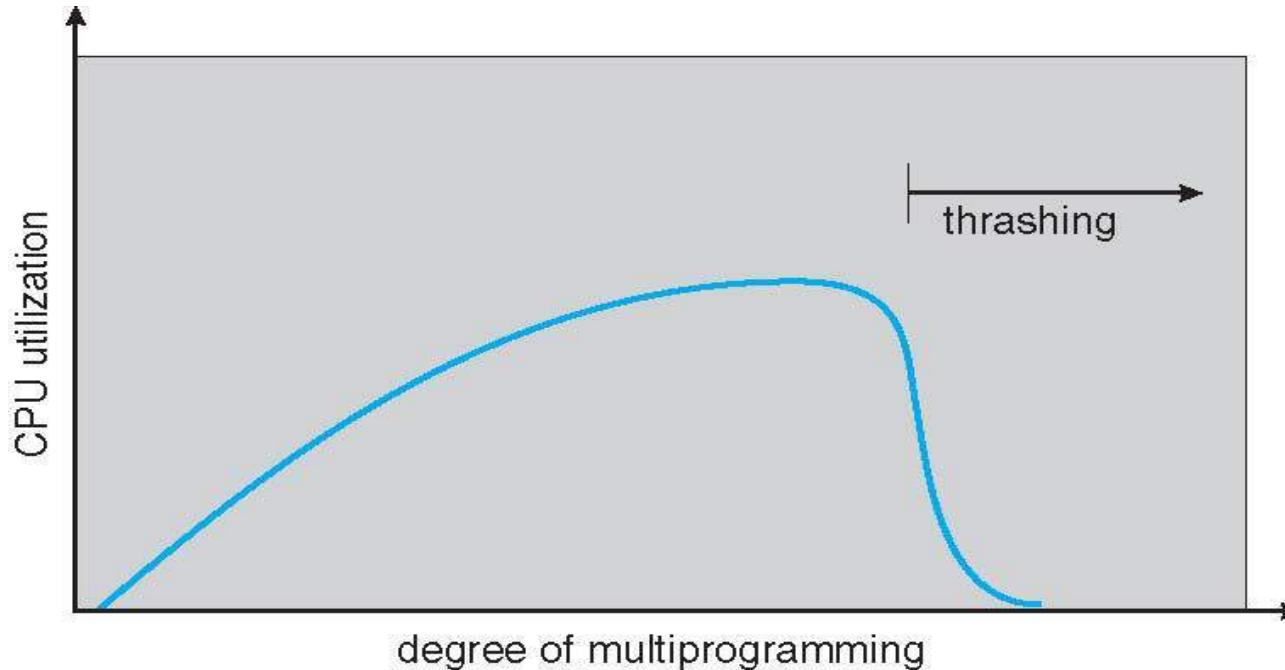
- Problem of Thrashing and possible solutions
- Mmap(), Memory mapped files
- Kernel Memory Management
- Other Considerations

Thrashing

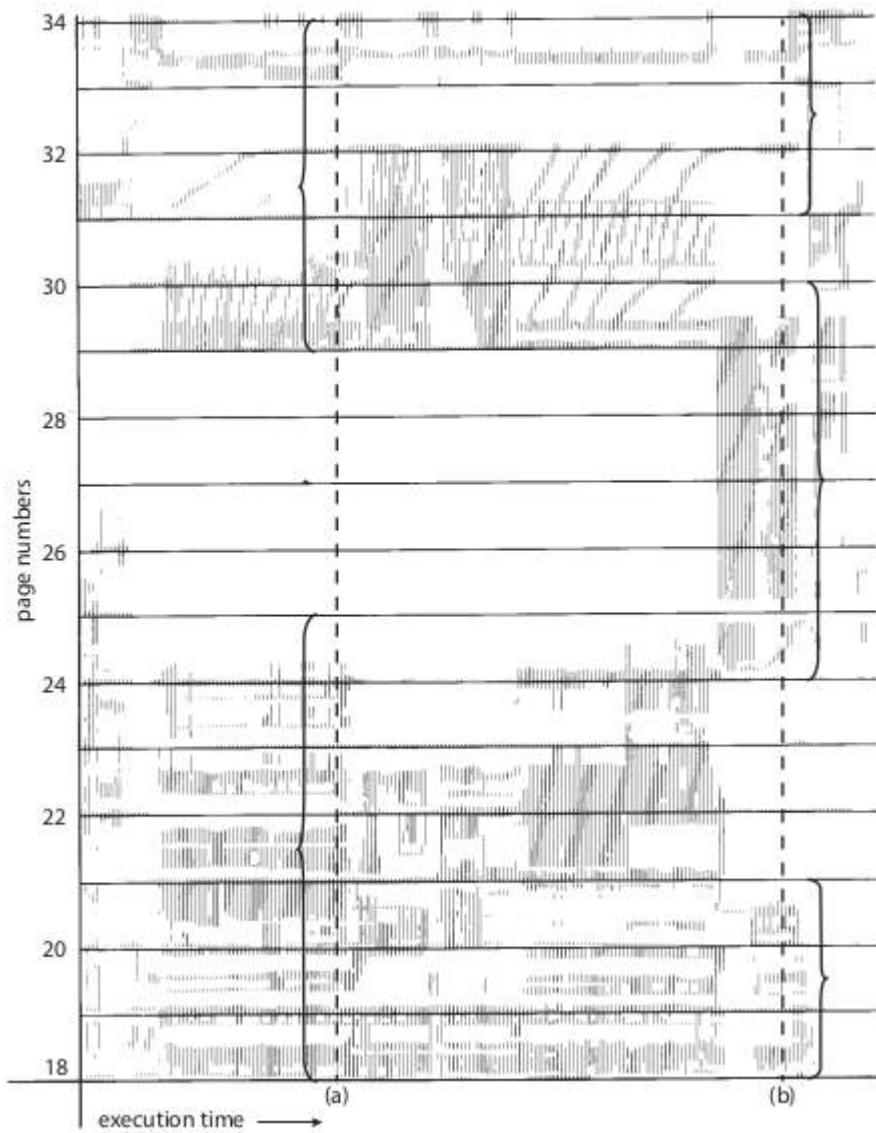
Thrashing

- 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



Locality In A Memory-Reference Pattern

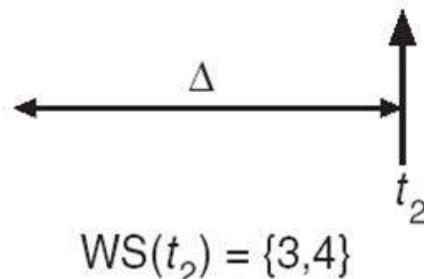
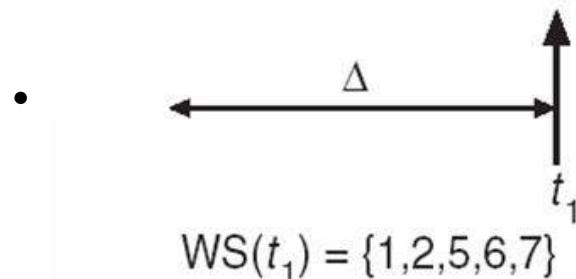


Demand paging and thrashing

- 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

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
 - page reference table
 - $\dots 2\ 6\ 1\ 5\ 7\ 7\ 7\ 5\ 1\ 6\ 2\ 3\ 4\ 1\ 2\ 3\ 4\ 4\ 4\ 3\ 4\ 3\ 4\ 4\ 4\ 1\ 3\ 2\ 3\ 4\ 4\ 4\ 3\ 4\ 4\ 4\ \dots$

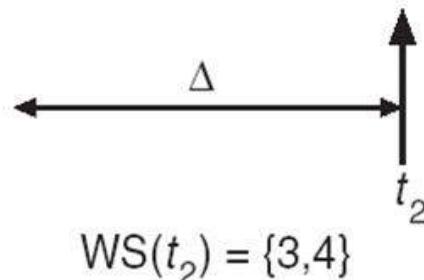
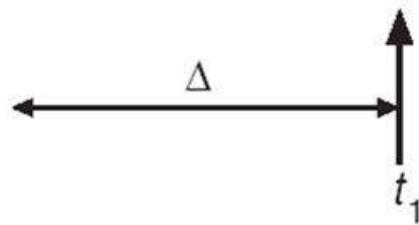


Working set model

- $D = \sum WSS_i \equiv$ 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 ...

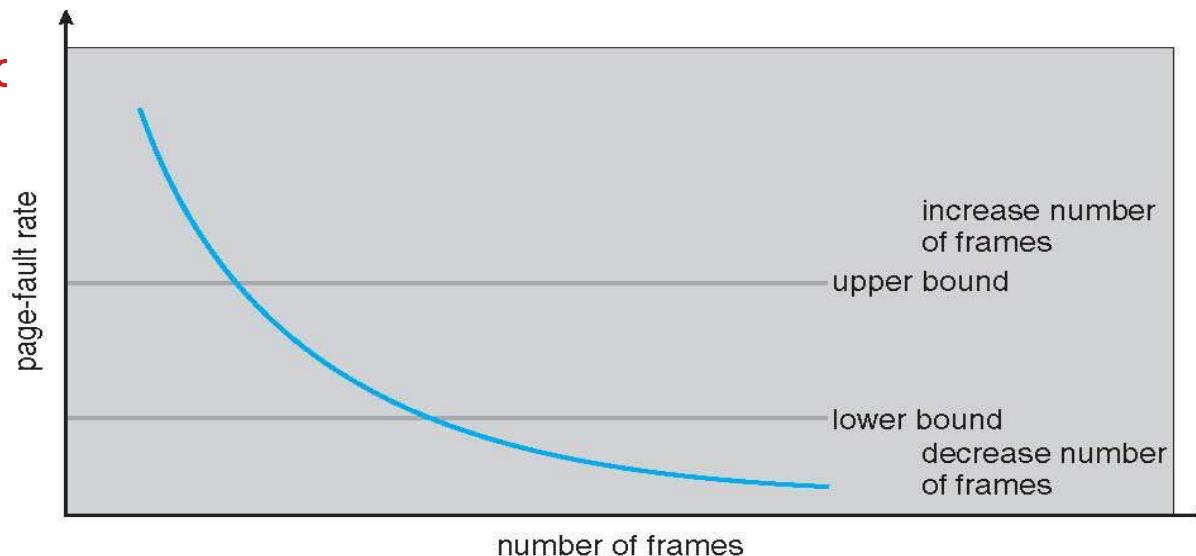


Keeping Track of the Working Set

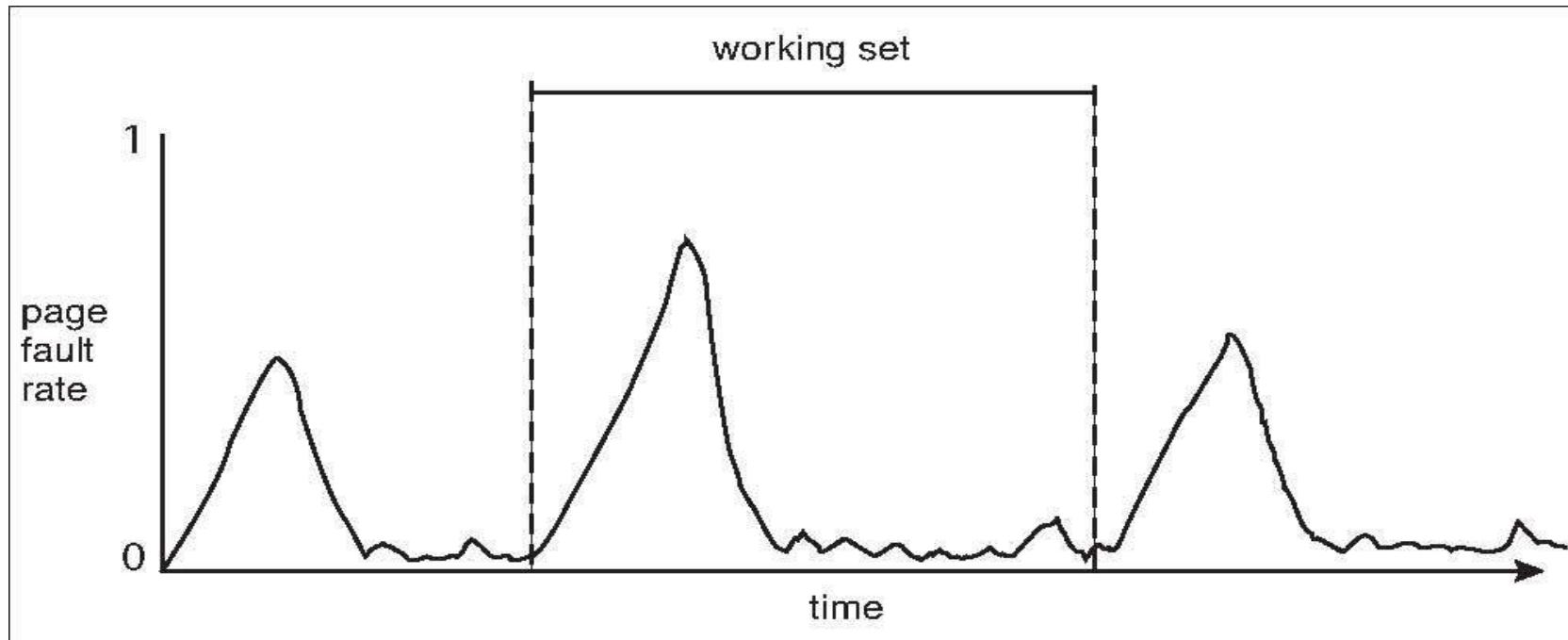
- Approximate with interval timer + a reference bit
- Example: $\Delta = 10,000$
 - Timer interrupts after every 5000 time units
 - Keep in memory 2 bits for each page
 - Whenever a timer interrupt occurs copy (to memory) and sets the values of all reference bits to 0
 - If one of the bits in memory = 1 \Rightarrow page in working set
- Why is this not completely accurate?

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 ac



Working Sets and Page Fault Rates

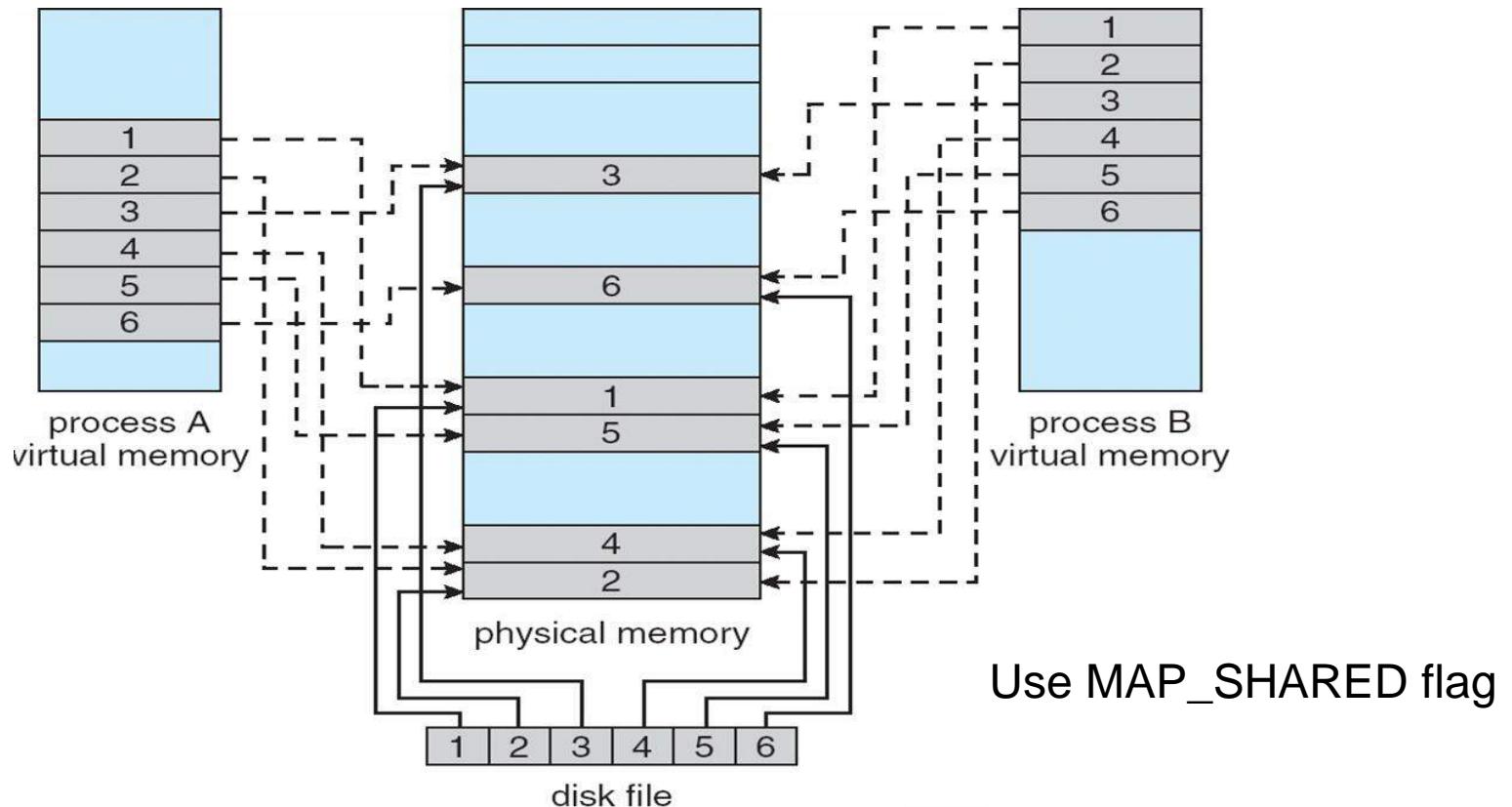


Memory Mapped Files

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

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

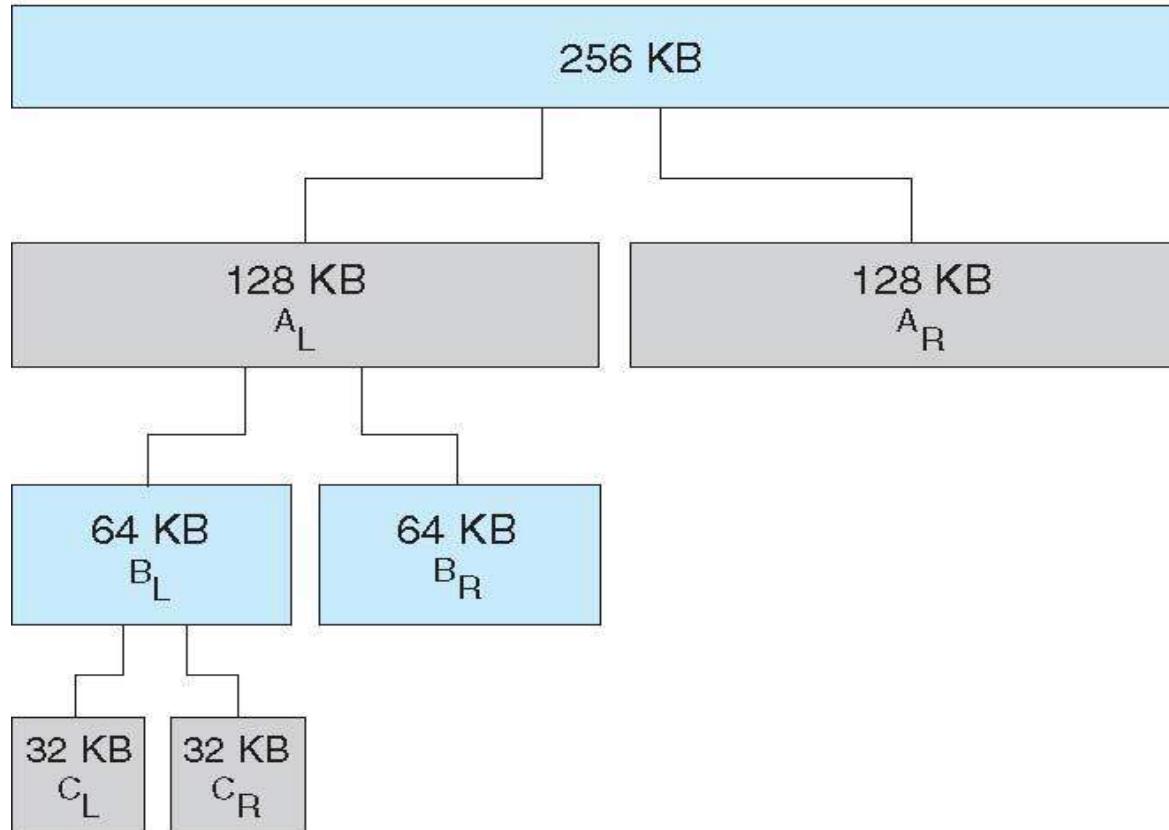
Allocating Kernel Memory

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



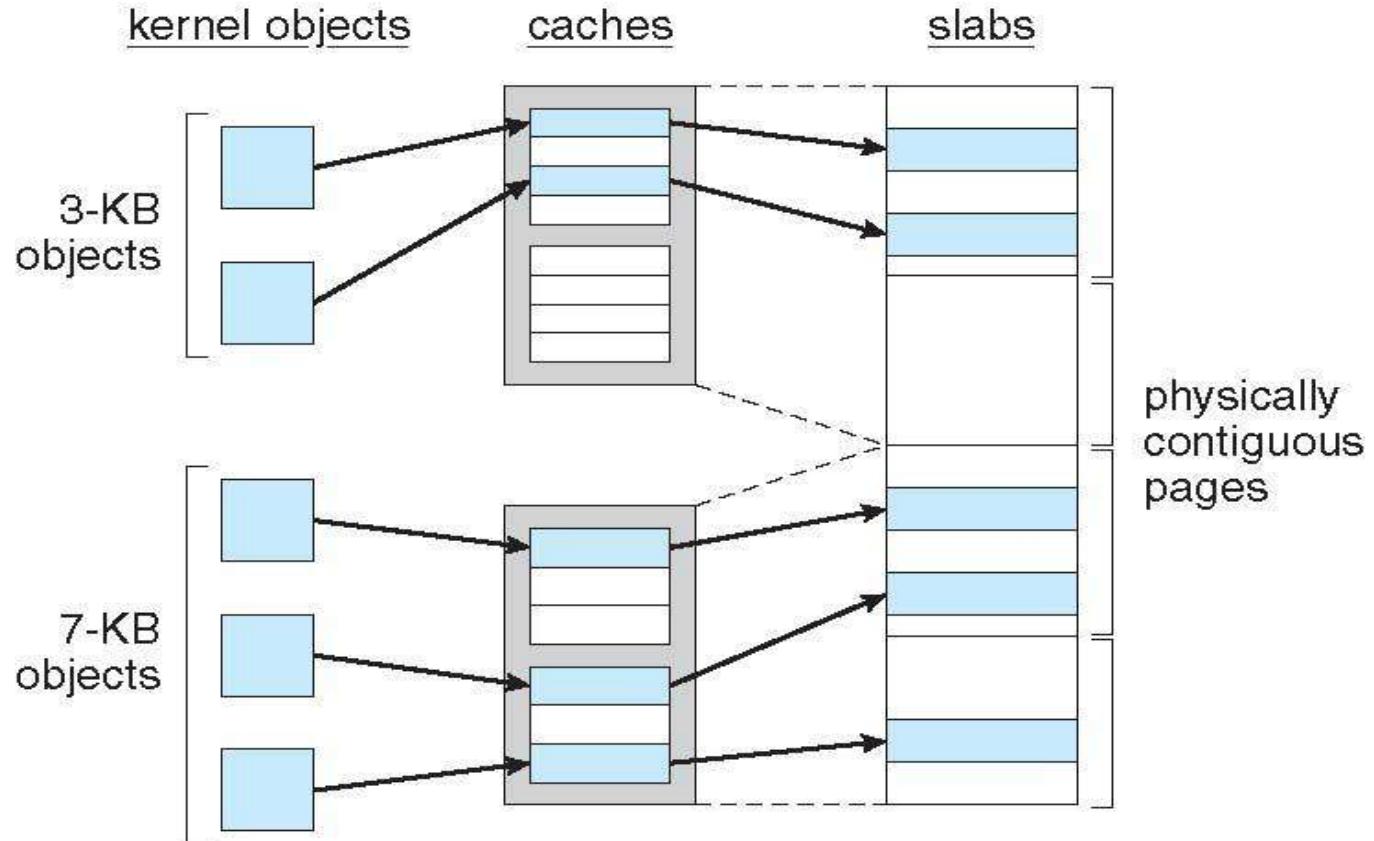
Buddy Allocator

- 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

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?

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

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

Program Structure

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

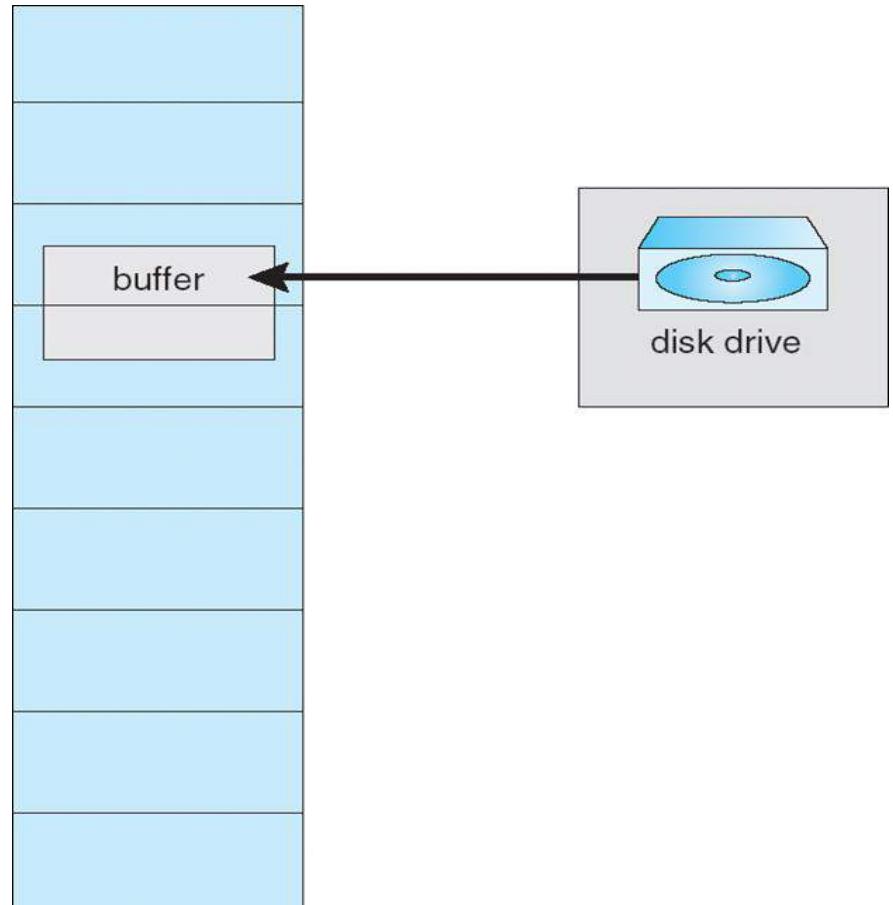
```
for (j = 0; j < 128; j++)
    for (i = 0; i < 128; i++)
        data[i, j] = 0;
```

$128 \times 128 = 16,384$ page faults

- Program 2
- ```
for (i = 0; i < 128; i++)
 for (j = 0; j < 128; j++)
 data[i, j] = 0;
```

# 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



# **Understanding Boot Process**

## **BIOS, UEFI, etc.**

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

All credits: “Hands on Booting” by Yogesh Babar

# What do we already know about “boot sequence”?

- When POWERed-ON
  - The CPU’s IP (or CS+IR, etc) has manufacturer defined value
    - The Motherboard manufacturer has ensured that a code called “Basic Input Output System(BIOS)” is at this location
  - CPU starts doing fetch-decode-execute-repeat
    - So CPU is running BIOS
  - BIOS

# What do we already know about “boot sequence”?

- BIOS (contd)
  - Reads “Sector-0” of the “Boot-device”, and loads it in RAM
    - Sector-0 is “boot-loader’ (at least that’s what we said!)
  - Jumps to it. Now boot-loader is running
- Boot-loader
  - Loads the OS kernel (boot-loader somehow knows where kernel is!) in RAM
  - Jumps to kernel code

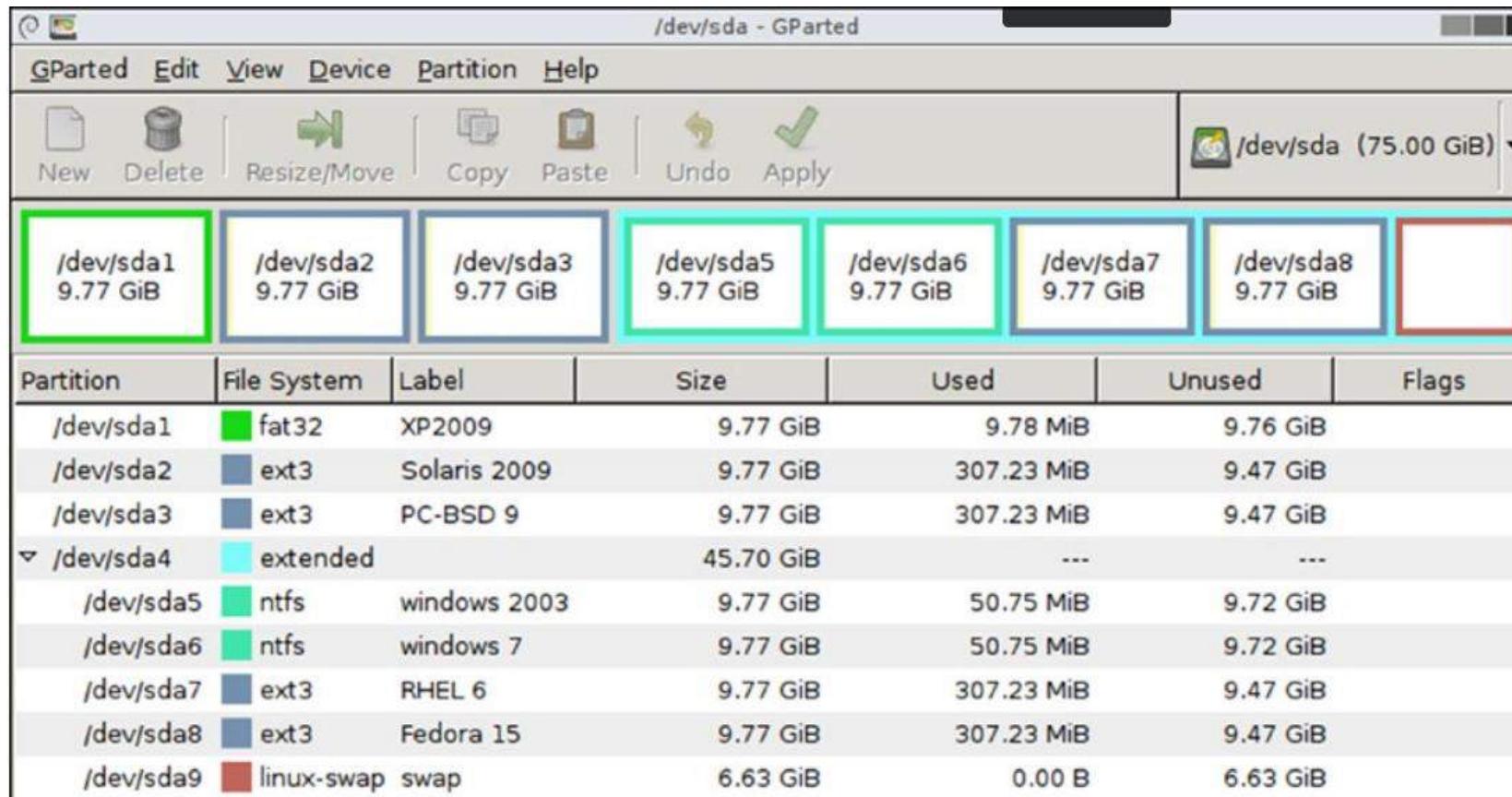
# **Legacy boot**

## **The world of BIOS + MBR**

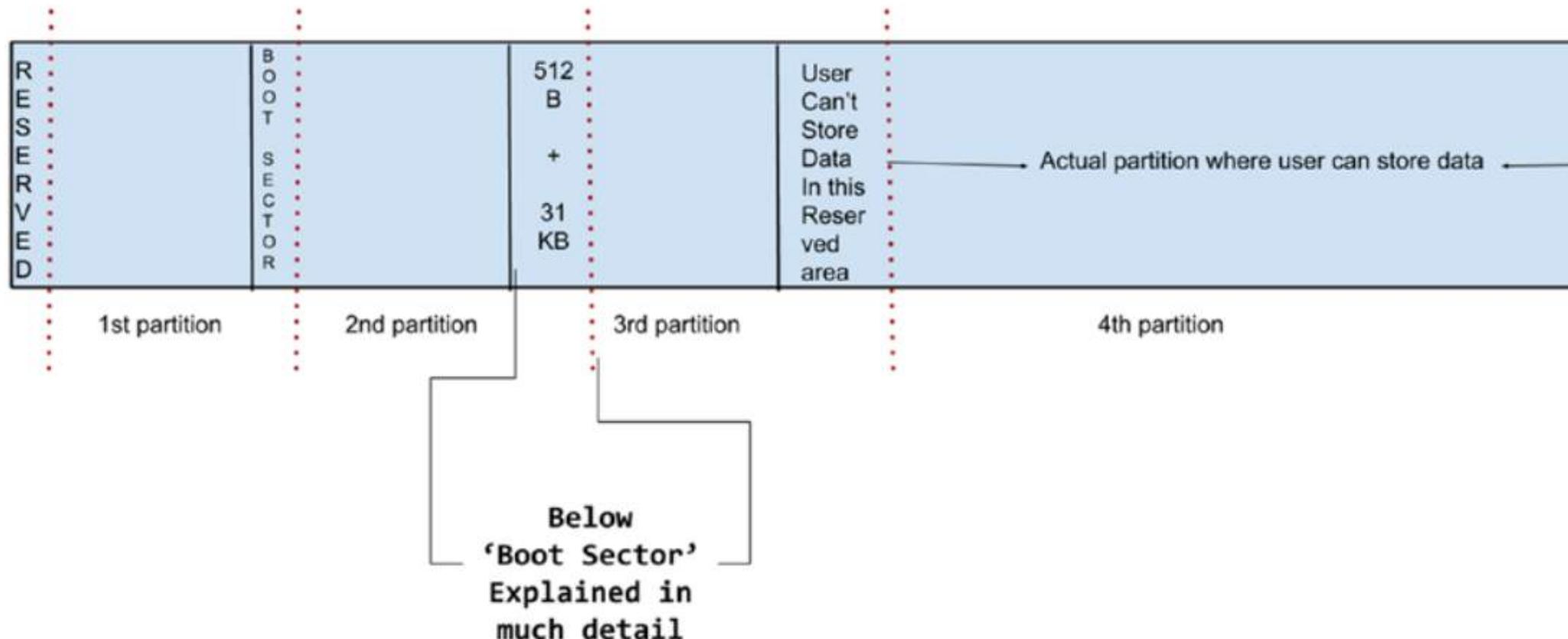
# Primary and Logical Partitions

- Max 4 primary partitions
- Want more?
  - 2<sup>nd</sup> or 3<sup>rd</sup> or 4<sup>th</sup> partition can be made “extended”
  - Within extended partition create logical partitions

# Primary and Logical Partitions



# Primary and Logical Partitions



---

| OS      | Rules                                                                                                                                                                                                                                                                                                                                                                                          |
|---------|------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| Unix    | Unix operating systems (OpenSolaris and BSD) have to be installed on a primary partition only.                                                                                                                                                                                                                                                                                                 |
| Linux   | Linux does not have any installation rules. It can be installed on any primary or logical partition.                                                                                                                                                                                                                                                                                           |
| Windows | The Windows operating system can be installed on any partition (primary or logical), but the predecessor of the Windows family has to be present on the first primary. That means you can install Windows 7 on a logical partition, but its predecessor, which is XP or win2k3, has to be present on the first primary partition. Also, you cannot break the Windows operating system sequence |

# Windows XP installer

## Windows XP Professional Setup

The following list shows the existing partitions and unpartitioned space on this computer.

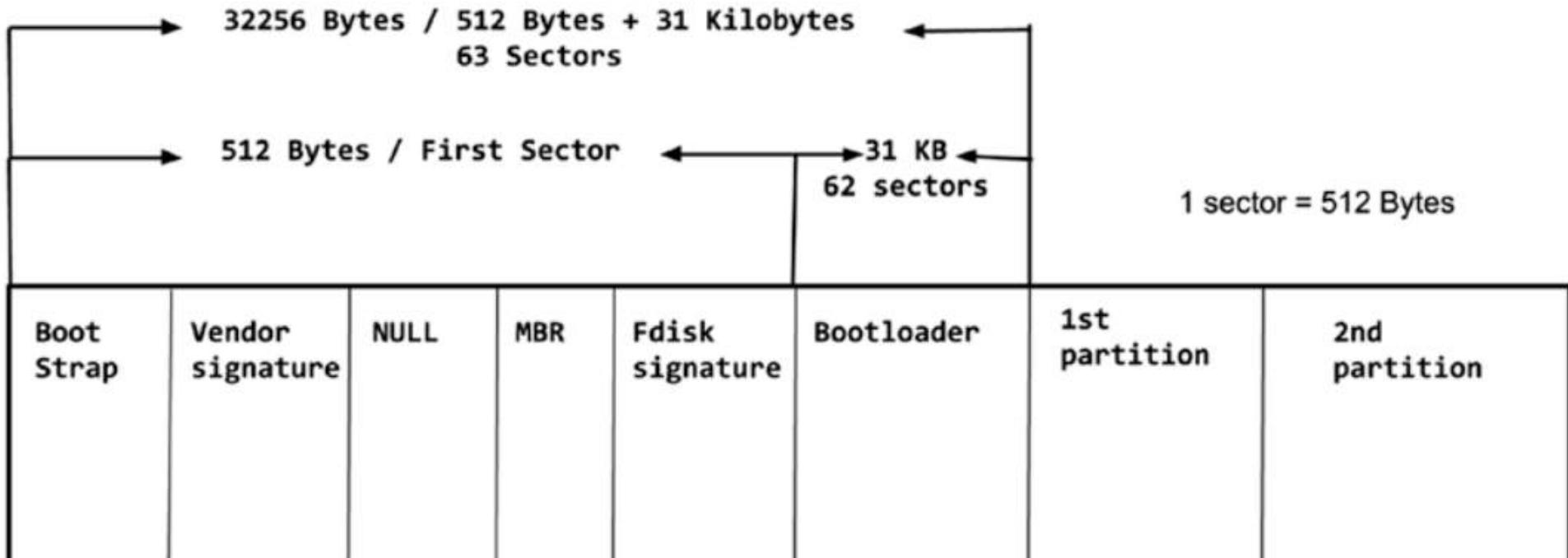
Use the UP and DOWN ARROW keys to select an item in the list.

- To set up Windows XP on the selected item, press ENTER.
- To create a partition in the unpartitioned space, press C.
- To delete the selected partition, press D.

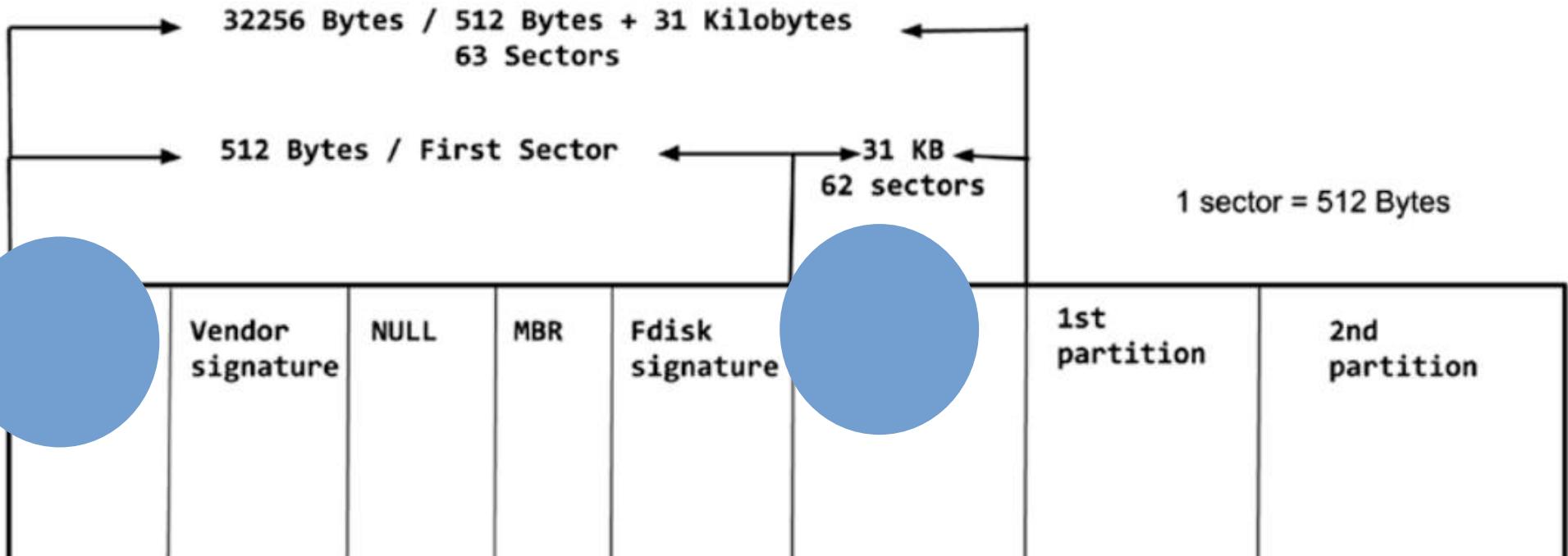
76796 MB Disk 0 at Id 1 on bus 0 on atapi [MBR]

|    |                                 |                           |
|----|---------------------------------|---------------------------|
| C: | Partition1 <XP2009> [FAT32]     | 10000 MB < 9990 MB free>  |
| G: | Partition2 [Unknown]            | 10000 MB < 10000 MB free> |
| H: | Partition3 [Unknown]            | 10000 MB < 10000 MB free> |
| D: | Partition4 <windows 200> [NTFS] | 10000 MB < 9949 MB free>  |
| E: | Partition5 <windows 7> [NTFS]   | 10000 MB < 9949 MB free>  |
| I: | Partition6 [Unknown]            | 10000 MB < 10000 MB free> |
| J: | Partition7 [Unknown]            | 10000 MB < 10000 MB free> |

# “Boot sector” is a misnomer here! it’s more than a sector!



# Bootloader code is in three parts!



# Bootloader code is in three parts!

| Location                      | Size               | Part         | Information               |
|-------------------------------|--------------------|--------------|---------------------------|
| Bootstrap                     | 440 bytes          | NTLDR part-1 | The tiniest part          |
| Bootloader                    | 31 KB              | NTLDR part-2 | Bigger compared to part-1 |
| Inside an actual OS partition | No size limitation | NTLDR part-3 | The biggest part          |

# Boot sequence of Win-Xp

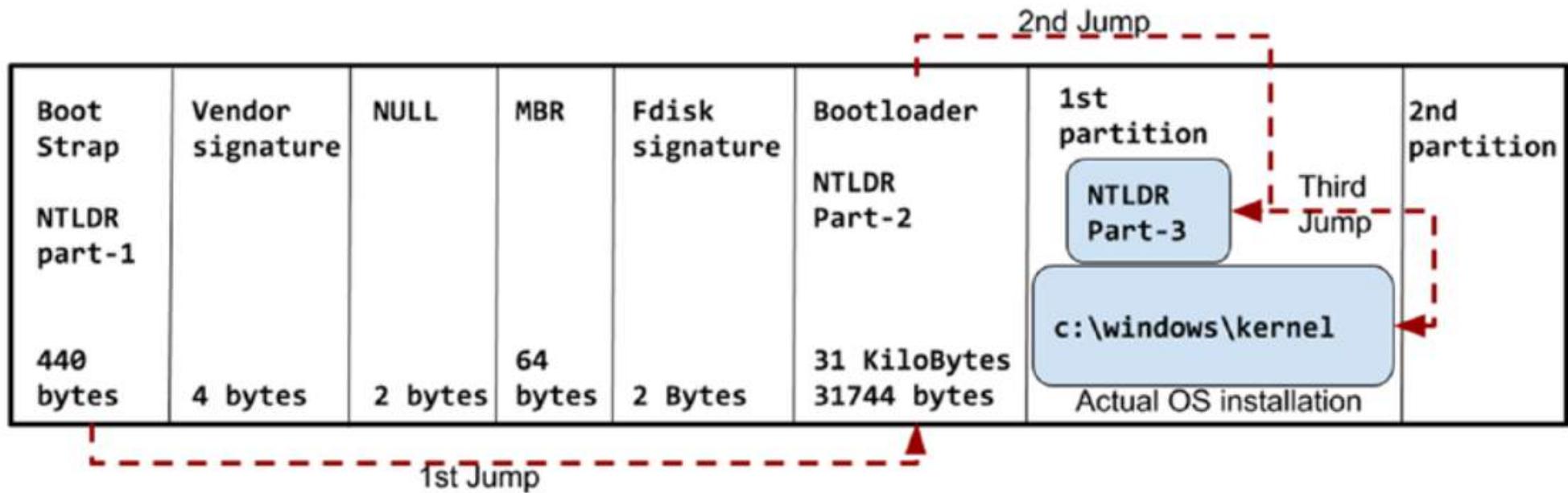
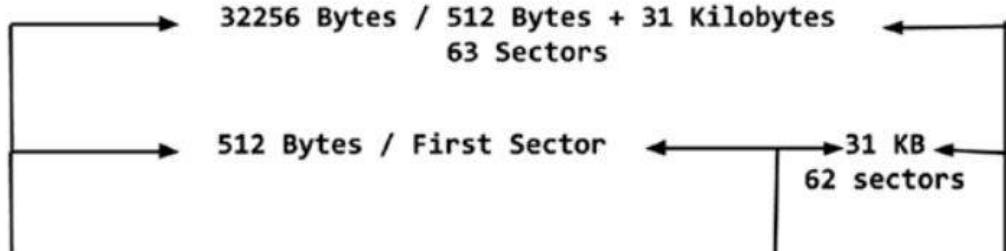


Figure 2-9. The boot sequence of Windows XP

# More on boot region

- Vendor signature: Seagate, WD, etc.
- NULL (is 0, else disk is bad)
- MBR: reason why only 4 primary partitions!



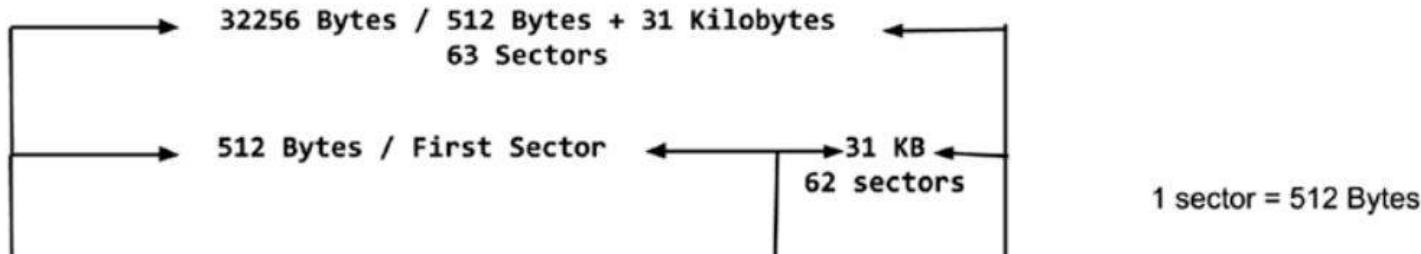
| Size     | Parts  | Stores                         |
|----------|--------|--------------------------------|
| 16 bytes | Part-1 | First partition's information  |
| 16 bytes | Part-2 | Second partition's information |
| 16 bytes | Part-3 | Third partition's information  |
| 16 bytes | Part-4 | Fourth partition's information |

1 sector = 512 Bytes

# More on boot region

## □ Fdisk signature

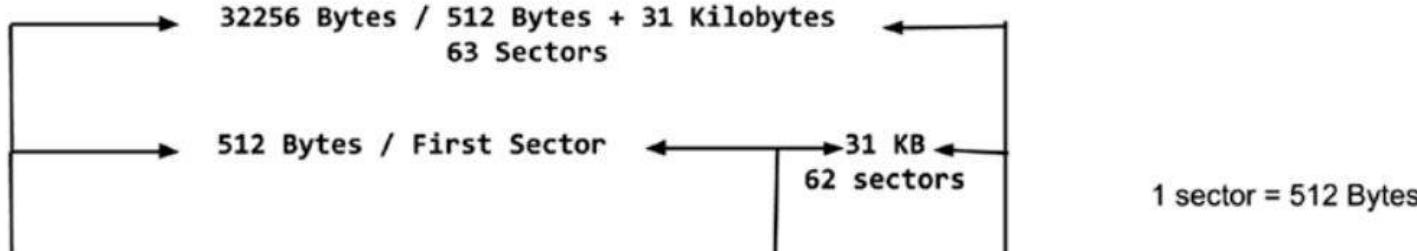
- Can be “\*” or something else
- When \*, called boot flag, or active/inactive flag



# More on boot region

## ▫ Fdisk signature

- Can be "\*" or something else
- When , called boot flag, or active/inactive flag, OS kernel is supposed to be loaded from partition with "\*" flag
  - Makes sense only when you have multi-boot system
- BIOS -> Bootstrap code (440 bytes) -> Bootloader (31 kB) of first



# Flowchart

- Run: BIOS
- Run: Partition-1, Bootloader part-1: Sector-0 , 440 bytes
- Run: Partition-1, Bootlaoder part-2: 31k
  - Checks where is the “\*” mark, that is the active flag
    - If on partition-1
      - Run Partition-1, Bootlaoder part-3

# **UEFI**

## **Unified Extended Firmware Interface**

# **BIOS Limitations**

- **BIOS will only be able to jump to the first sector, which is 512 bytes.**
- **BIOS cannot generate good graphics/GUIs.**
- **You cannot use a mouse in the BIOS.**
- **The maximum partition size is 2.2 TB.**
- **The BIOS is dumb because it does not understand the bootloader or the OS**

# **UEFI advantages**

- **UEFI can use the full CPU. Unlike the BIOS (which is stuck with 16 bits of processor), UEFI can access up to 64 bits.**
- **UEFI can use a full RAM module. Unlike 1 MB of address space of the BIOS, UEFI can support and use terabytes of RAM.**
- **Instead of 64 bytes of a tiny MBR, UEFI uses the GPT (GUID) partition table which can handle all the data.**

# UEFI advantages

- **It's a small OS**

- a) You will have full access to audio and video devices.
- b) You will be able to connect to WiFi.
- c) You will be able to use the mouse.
- d) In terms of the GUI, UEFI will provide a rich graphics interface.
- e) UEFI will have its own app store like we have for Android and Andoid



## UEFI BIOS Utility – EZ Mode

12/03/2019 19:28

English

EZ Tuning Wizard(F11)

## Information

ROG RAMPAGE VI EXTREME BIOS Ver. 1301

Intel(R) Core(TM) i9-7960X CPU @ 2.80GHz

Speed: 3600 MHz

Memory: 49152 MB (DDR4 2133MHz)

## CPU Temperature

## CPU Core Voltage

0.991 V

## Motherboard Temperature

42°C

51°C

## DRAM Status

DIMM\_A1: G-Skill 16384MB 2133MHz

DIMM\_A2: N/A

DIMM\_B1: Corsair 8192MB 2133MHz

DIMM\_B2: N/A

DIMM\_C1: G-Skill 16384MB 2133MHz

DIMM\_C2: N/A

DIMM\_D1: Corsair 8192MB 2133MHz

DIMM\_D2: N/A

X.M.P. ▾ Disabled

## SATA Information

P5: Samsung SSD 860 EVO 500GB (500.1GB)

P6: Samsung SSD 860 EVO 500GB (500.1GB)

## Intel Rapid Storage Technology

On

Off

## FAN Profile

CPU FAN  
1654 RPMCHA FAN2  
N/AHAMP  
N/ACHA FAN1  
N/ACHA FAN3  
N/ACPU OPT FAN  
1161 RPM

## CPU FAN



## EZ System Tuning

Click the icon below to apply a pre-configured profile for improved system performance or energy savings.



Normal



## Boot Priority

Choose one and drag the items.

Switch all

 Windows Boot Manager (M1: Samsung SSD 970 EVO Plus 1TB)  
⋮ M1: Samsung SSD 970 EVO Plus 1TB  
⋮ Windows Boot Manager (P6: Samsung SSD 860 EVO 500GB)  
⋮ P5: Samsung SSD 860 EVO 500GB (476940MB)  
⋮

HDD

EFI System Partition (ESP)

EFI directory

Microsoft  
BCD  
Bootloader

Fedora

GRUB  
Bootloader

Slackware

LILO  
Bootloader

Fat-32

Fedora

Windows

Ubuntu

Other....

Partition - 1

Partition - 2

Partition - 3

Partition - 4

Partition - 5

```
root@yogesh-virtual-machine:/home/yogesh# ls -l /boot/efi/EFI/ubuntu/
total 3724
-rwx----- 1 root root 108 Dec 2 17:47 BOOTX64.CSV
drwx----- 2 root root 4096 Dec 2 17:47 fw
-rwx----- 1 root root 75992 Dec 2 17:47 fwupx64.efi
-rwx----- 1 root root 126 Dec 2 17:47 grub.cfg
-rwx----- 1 root root 1116024 Dec 2 17:47 grubx64.efi
-rwx----- 1 root root 1269496 Dec 2 17:47 mmx64.efi
-rwx----- 1 root root 1334816 Dec 2 17:47 shimx64.efi
root@yogesh-virtual-machine:/home/yogesh# █
```

Figure 2-111. The EFI directory of Ubuntu

## Boot Manager

Boot normally

Continue to boot using  
the default boot order.

ubuntu

EFI VMware Virtual SCSI Hard Drive (0.0)

EFI VMware Virtual SATA CDROM Drive (1.0)

EFI Network

EFI Internal Shell (Unsupported option)

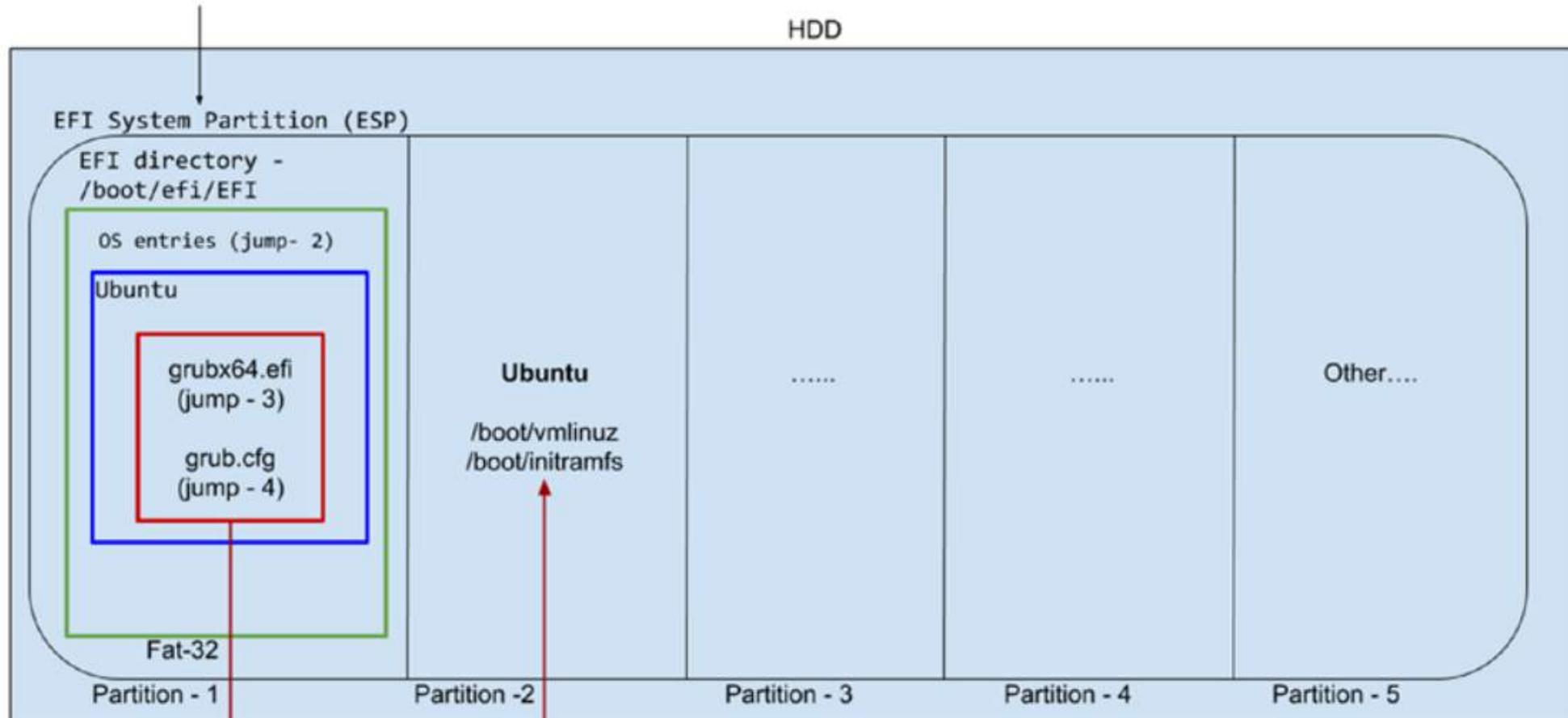
Enter setup

Reset the system

Shut down the system

UEFI to ESP Jump -1

HDD



## Boot Manager

Boot normally

Continue to boot using  
the default boot order.

ubuntu

Windows Boot Manager

EFI VMware Virtual SCSI Hard Drive (0.0)

EFI VMware Virtual SATA CDROM Drive (1.0)

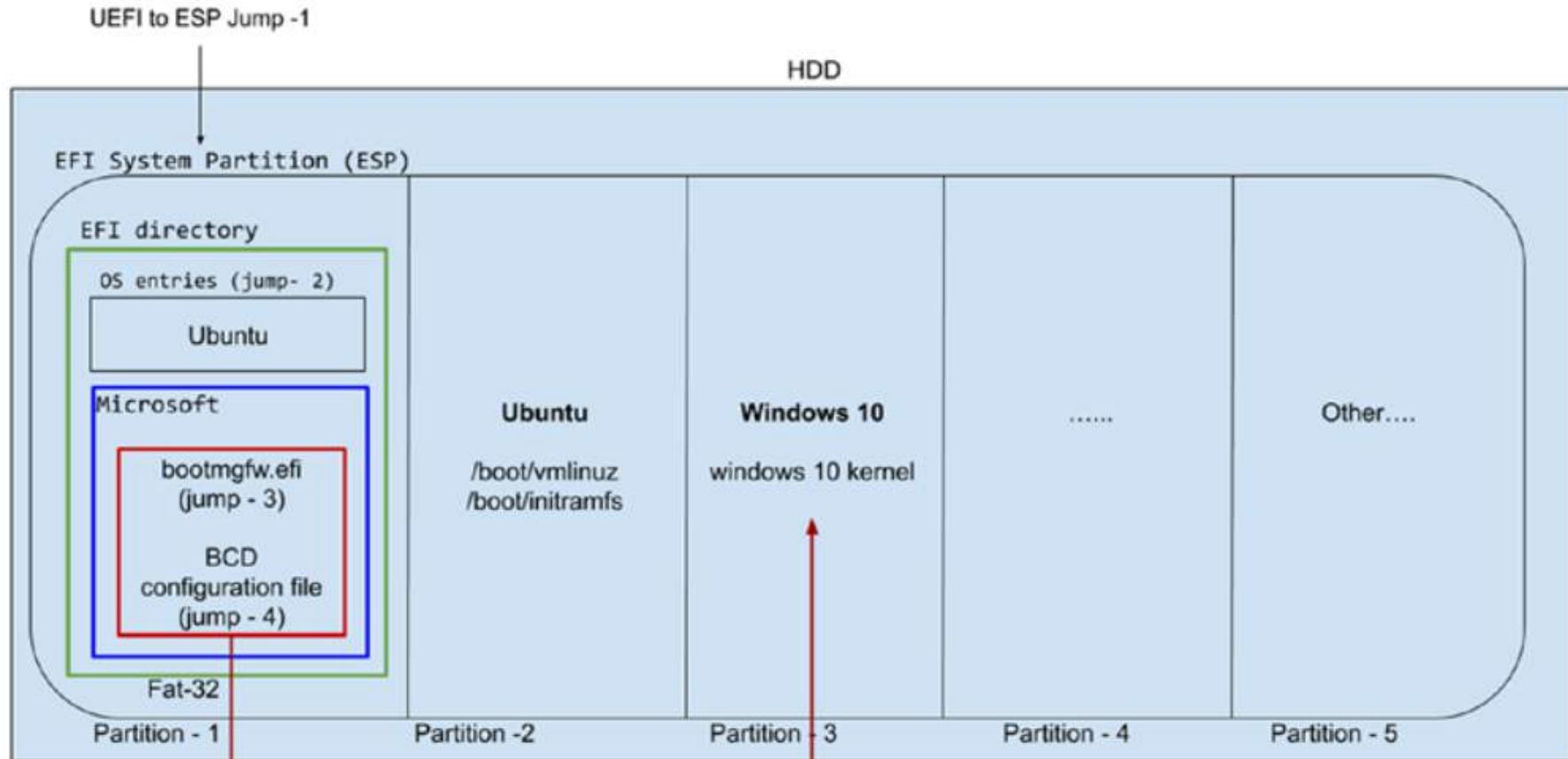
EFI Network

EFI Internal Shell (Unsupported option)

Enter setup

Reset the system

Shut down the system





# 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.

# Process Control Block



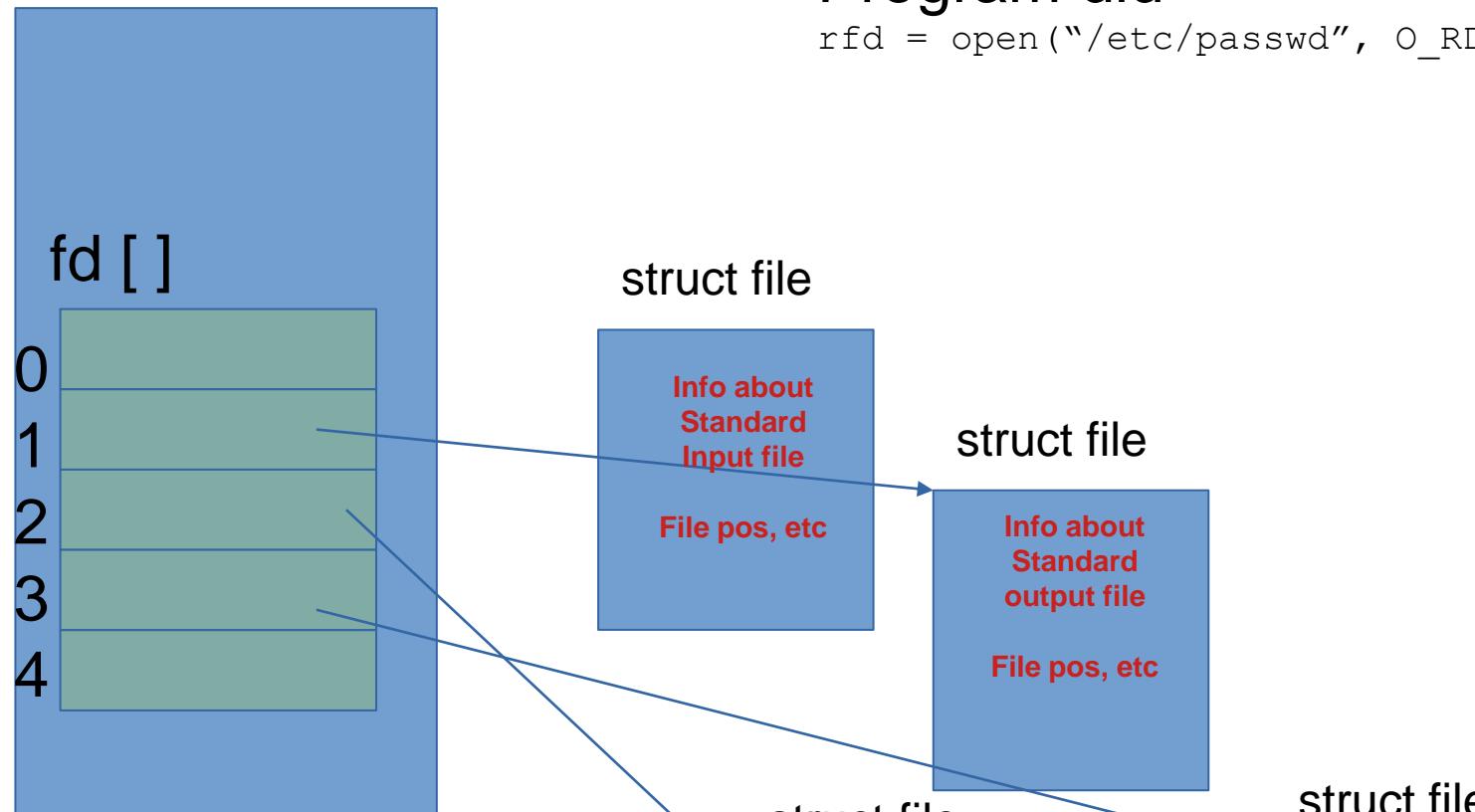
# Fields in PCB



# 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

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

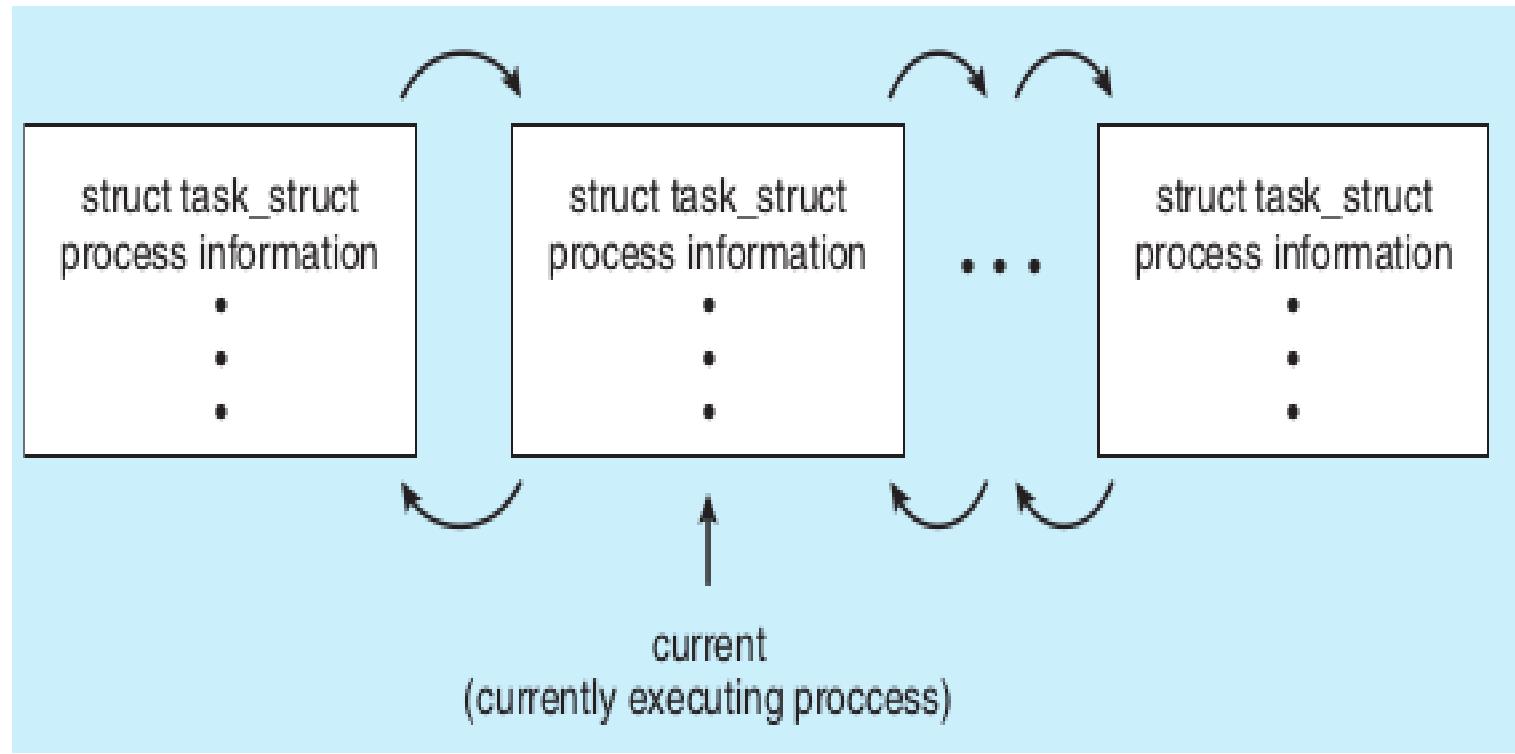
struct proc {
 uint sz; // Size of process memory (bytes)
 pde_t* pgdir; // Page table
 char *kstack; // Bottom of kernel stack for this process
 enum procstate state; // Process state
 int pid; // Process ID
 struct proc *parent; // Parent process
 struct trapframe *tf; // Trap frame for current syscall
 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[NOFILE]; // Open files
 struct inode *cwd; // Current directory
 char name[16]; // Process name (debugging)
};
```

struct {

```
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

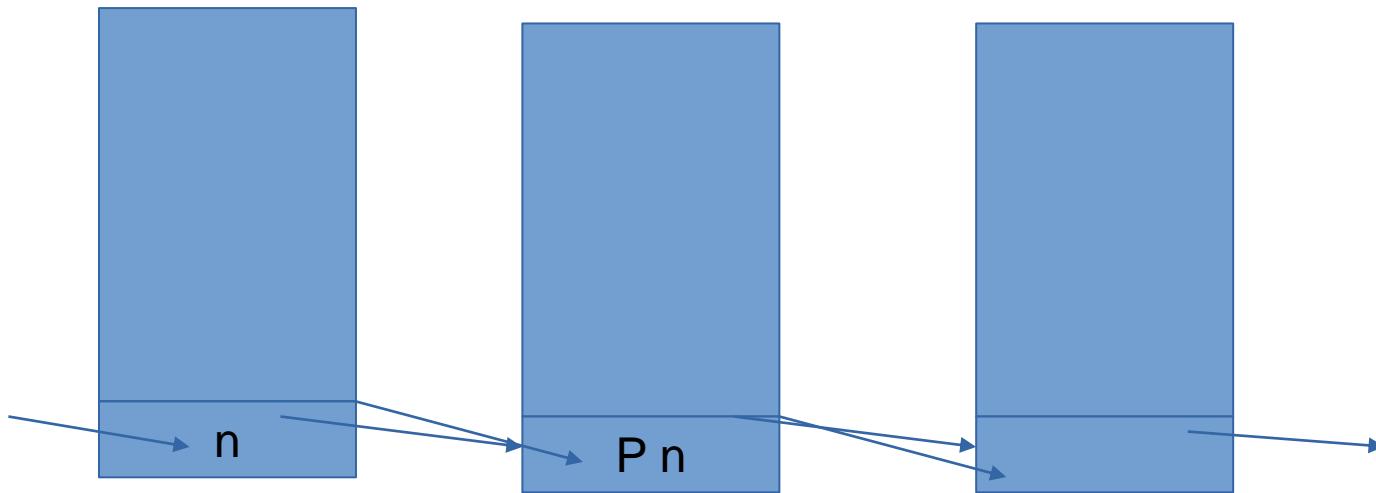
- Different types of queues/lists can be maintained by OS for the processes
  - A queue of processes which need to be scheduled
  - A queue of processes which have

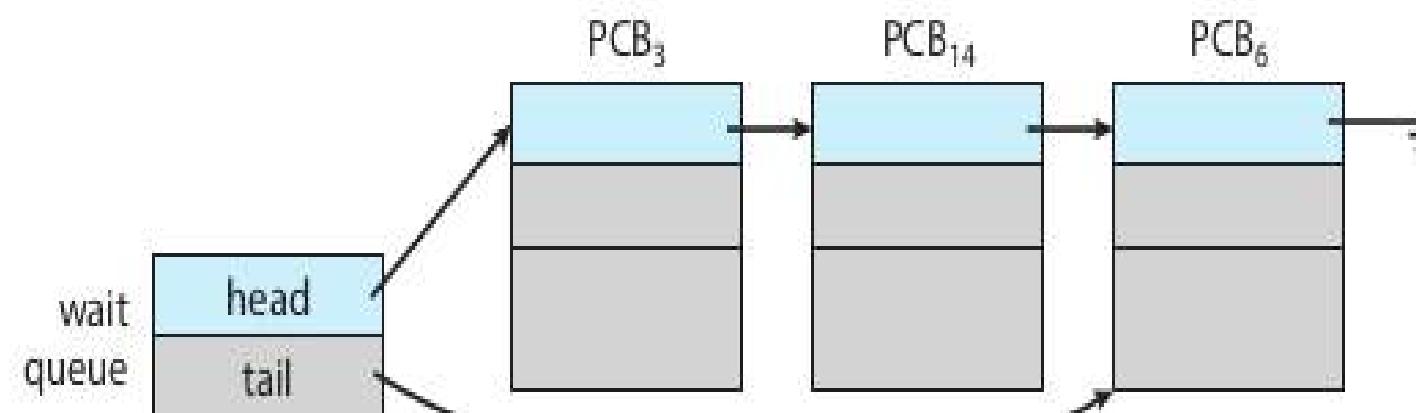
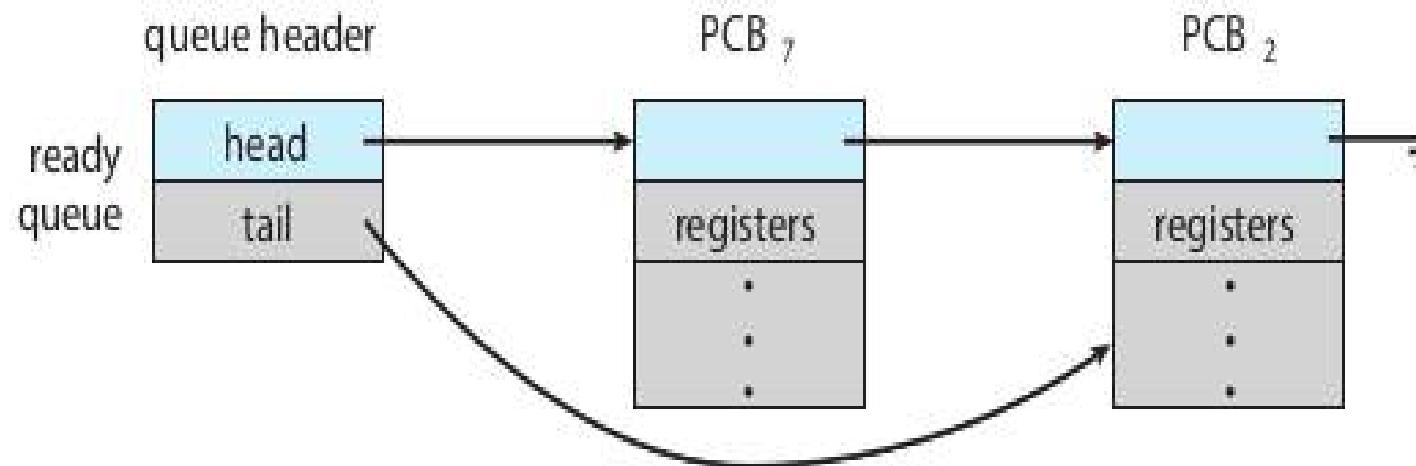


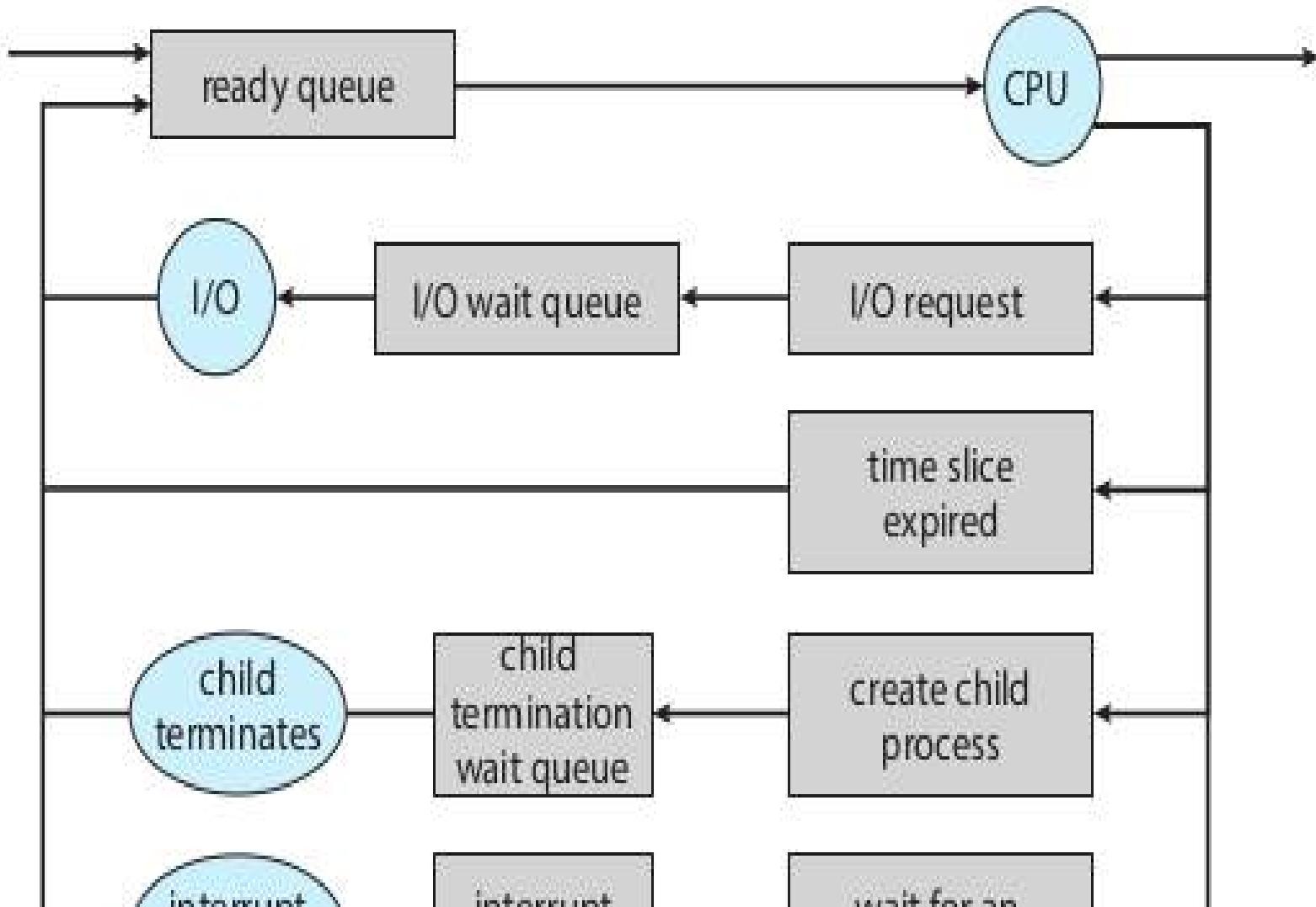
## // Linux data structure

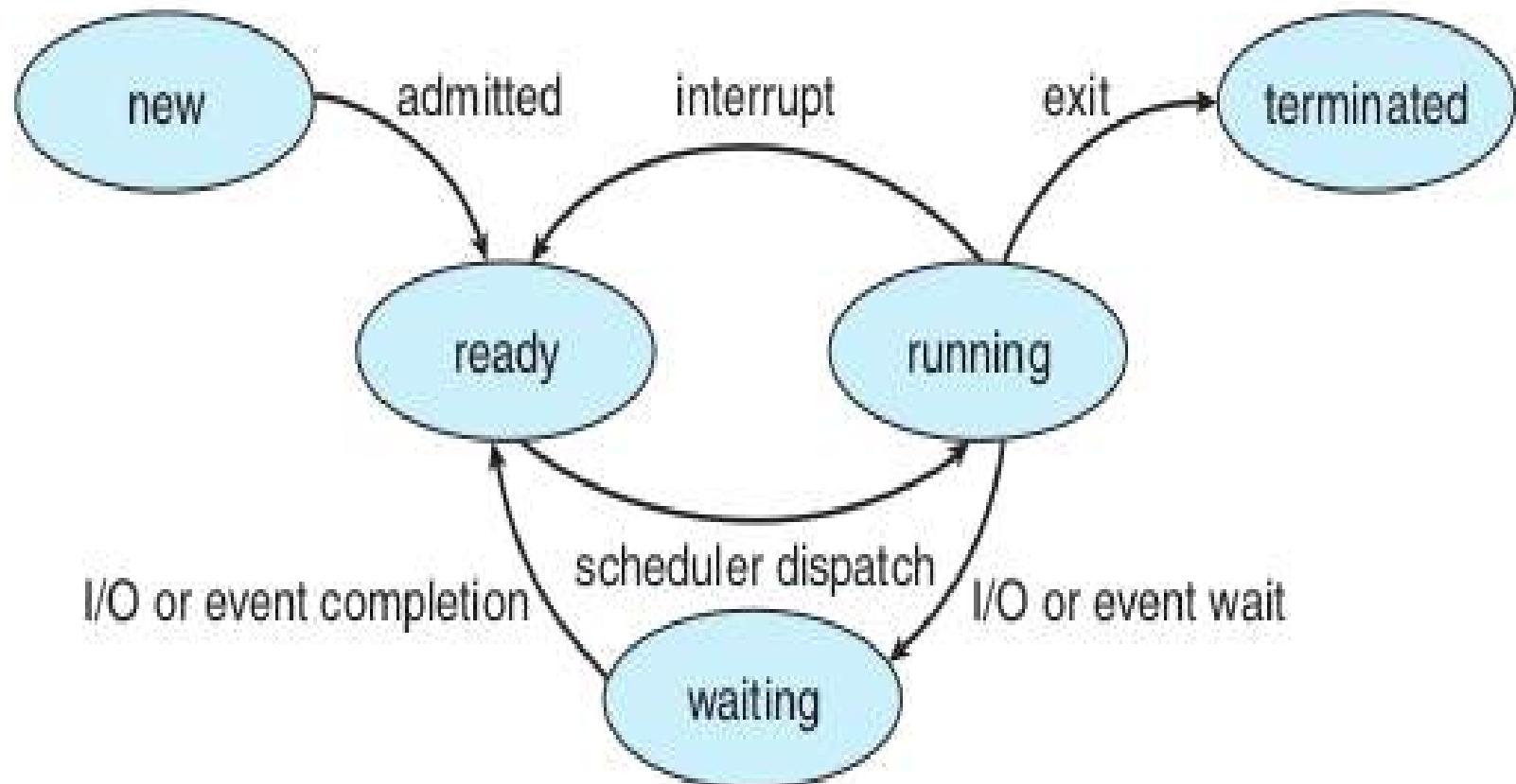
```
struct task_struct {
 long state; /* state of the process */
 struct sched_entity se; /* scheduling information */
 struct task_struct *parent; /* this process's parent */
```

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









**Figure 3.2** Diagram of process state.

See state in output of  
ps axu  
(BSD style options)

# “Giving up” CPU by a process or blocking

```
int main() {
 i = j + k;
 scanf("%d", &k);
}

int scanf(char
```

OS Syscall

```
sys_read(int fd,
char *buf, int len)
```

```
{
```

```
 file f = current-
 >fdarray[fd];
```

# **“Giving up” CPU by a process or blocking**

The relevant code in xv6 is in

Sleep()

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

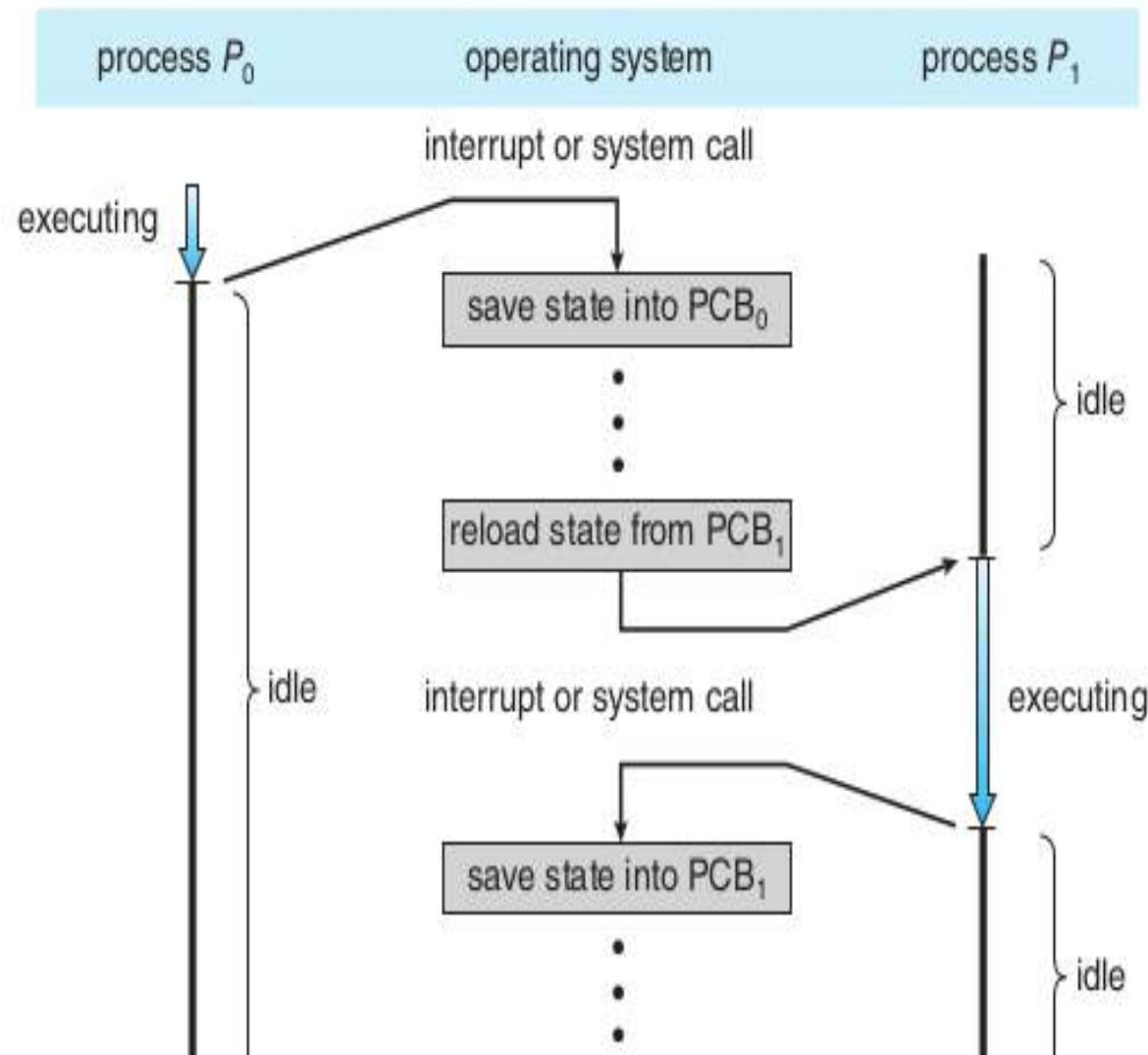
To be seen later

# 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

# Context Switch

- Is an overhead
- No useful work happening while doing a context switch
- Time can vary from hardware to hardware



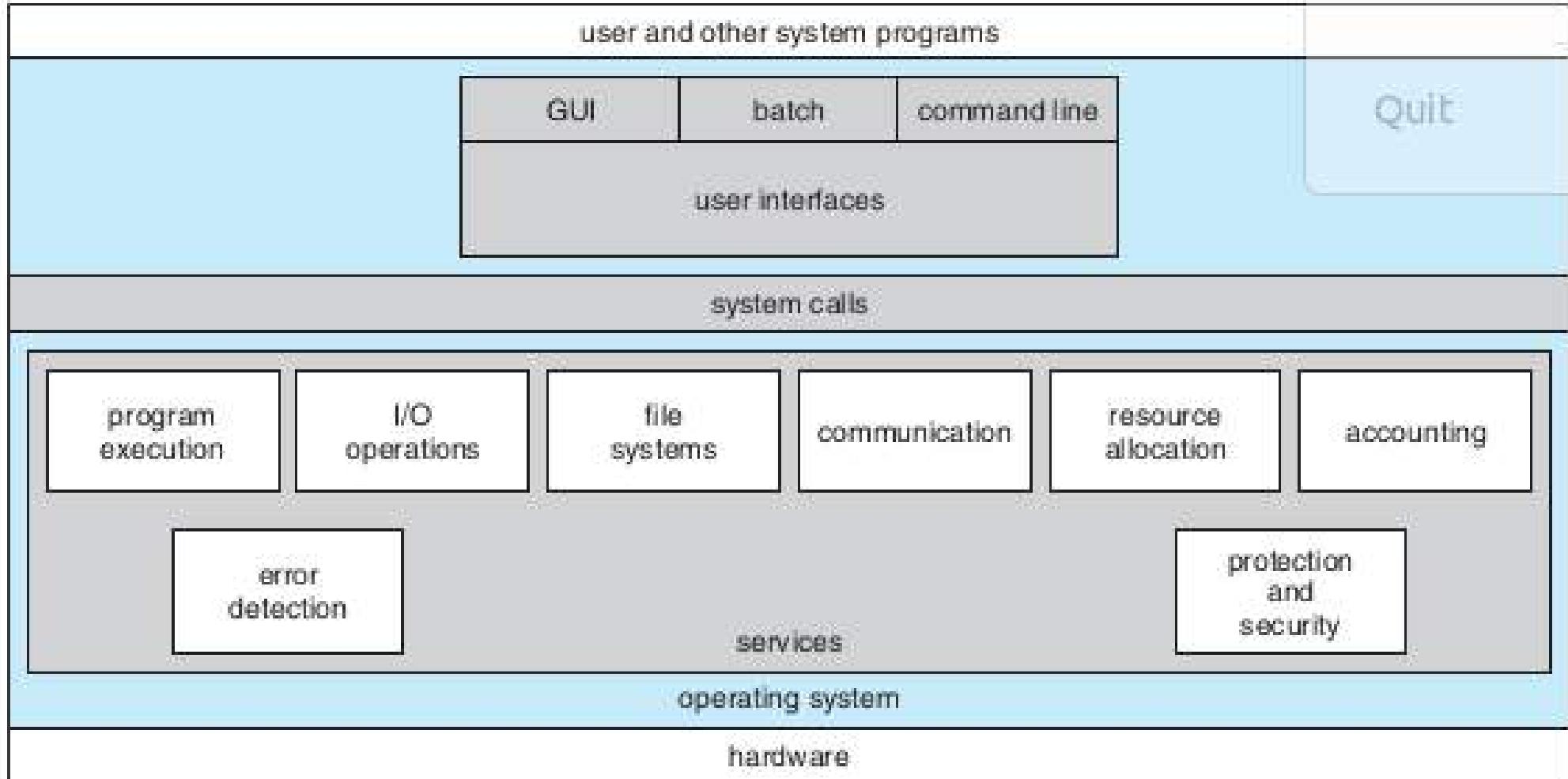
# Peculiarity of context switch

- When a process is running, the function calls work in LIFO fashion
  - Made possible due to calling convention
- When an interrupt occurs

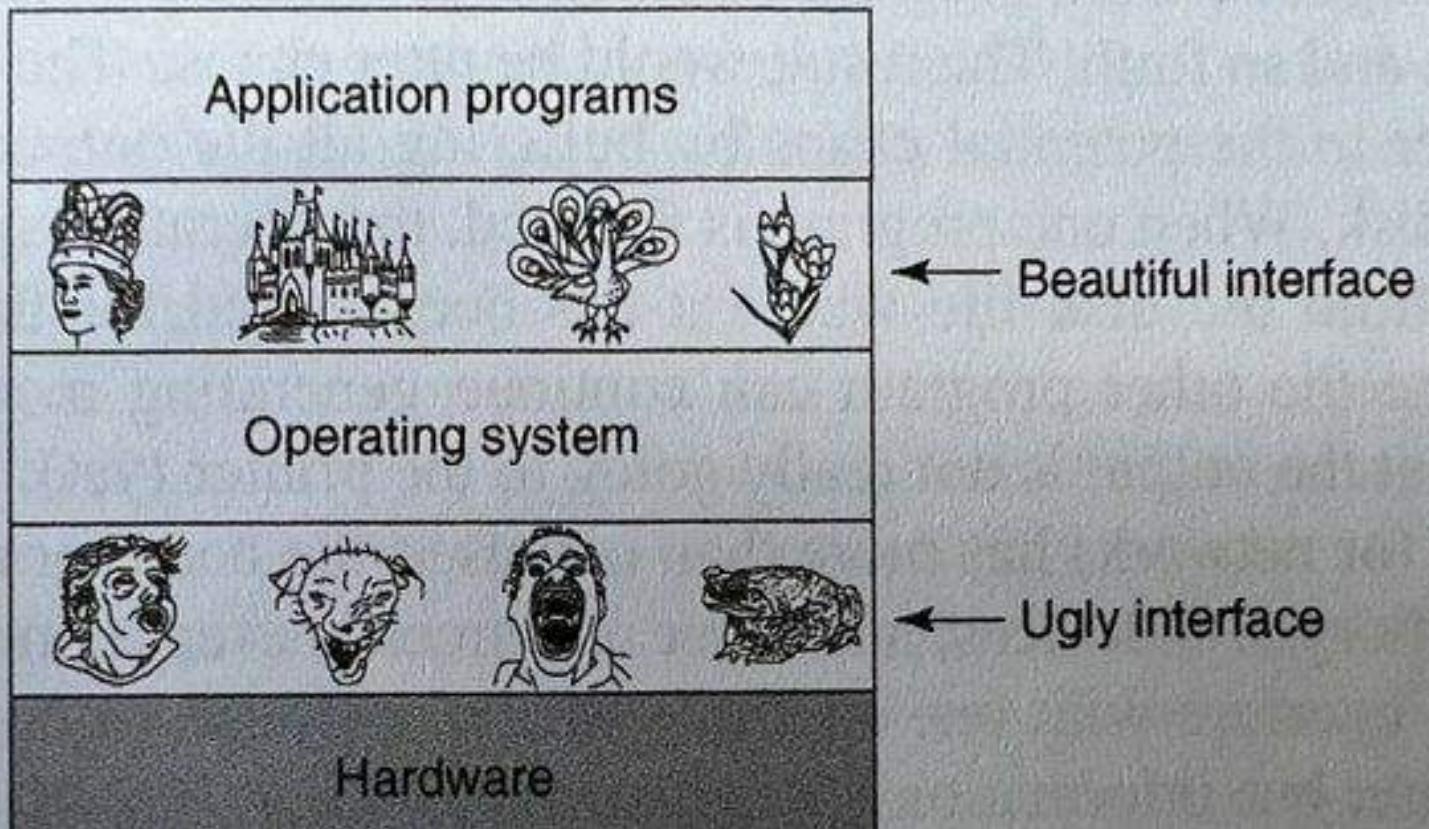
# **System Calls, fork(), exec()**

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

(C) Abhijit A.M.  
Available under Creative Commons Attribution-ShareAlike License V3.0+



# WHAT IS AN OPERATING SYSTEM?



# System Calls

- **Services provided by operating system to applications**
  - Essentially available to applications by calling the particular software interrupt application
    - All system calls essentially involve the “INT 0x80” on x86 processors + Linux
    - Different arguments specified in EAX register inform the kernel about different system calls

# Types of System Calls

- **File System Related**
  - Open(), read(), write(), close(), etc.
- **Processes Related**
  - Fork(), exec(), ...
- **Memory management related**
  - Mmap(), shm\_open(), ...

## Device Management

## *EXAMPLES OF WINDOWS AND UNIX SYSTEM CALLS*

|                                | <b>Windows</b>                                             | <b>Unix</b>                            |
|--------------------------------|------------------------------------------------------------|----------------------------------------|
| <b>Process Control</b>         | CreateProcess()<br>ExitProcess()<br>WaitForSingleObject()  | fork()<br>exit()<br>wait()             |
| <b>File Manipulation</b>       | CreateFile()<br>ReadFile()<br>WriteFile()<br>CloseHandle() | open()<br>read()<br>write()<br>close() |
| <b>Device Manipulation</b>     | SetConsoleMode()<br>ReadConsole()<br>WriteConsole()        | ioctl()<br>read()<br>write()           |
| <b>Information Maintenance</b> | GetCurrentProcessID()<br>SetTimer()<br>Sleep()             | getpid()<br>alarm()<br>sleep()         |
| <b>Communication</b>           | CreatePipe()                                               | pipe()                                 |

```
int main() {
 int a = 2;

 printf("hi\n");

}
```

-----

## C Library

-----

```
int printf("void *a, ...){
```

## Code schematic

-----user-kernel-  
mode-boundary-----

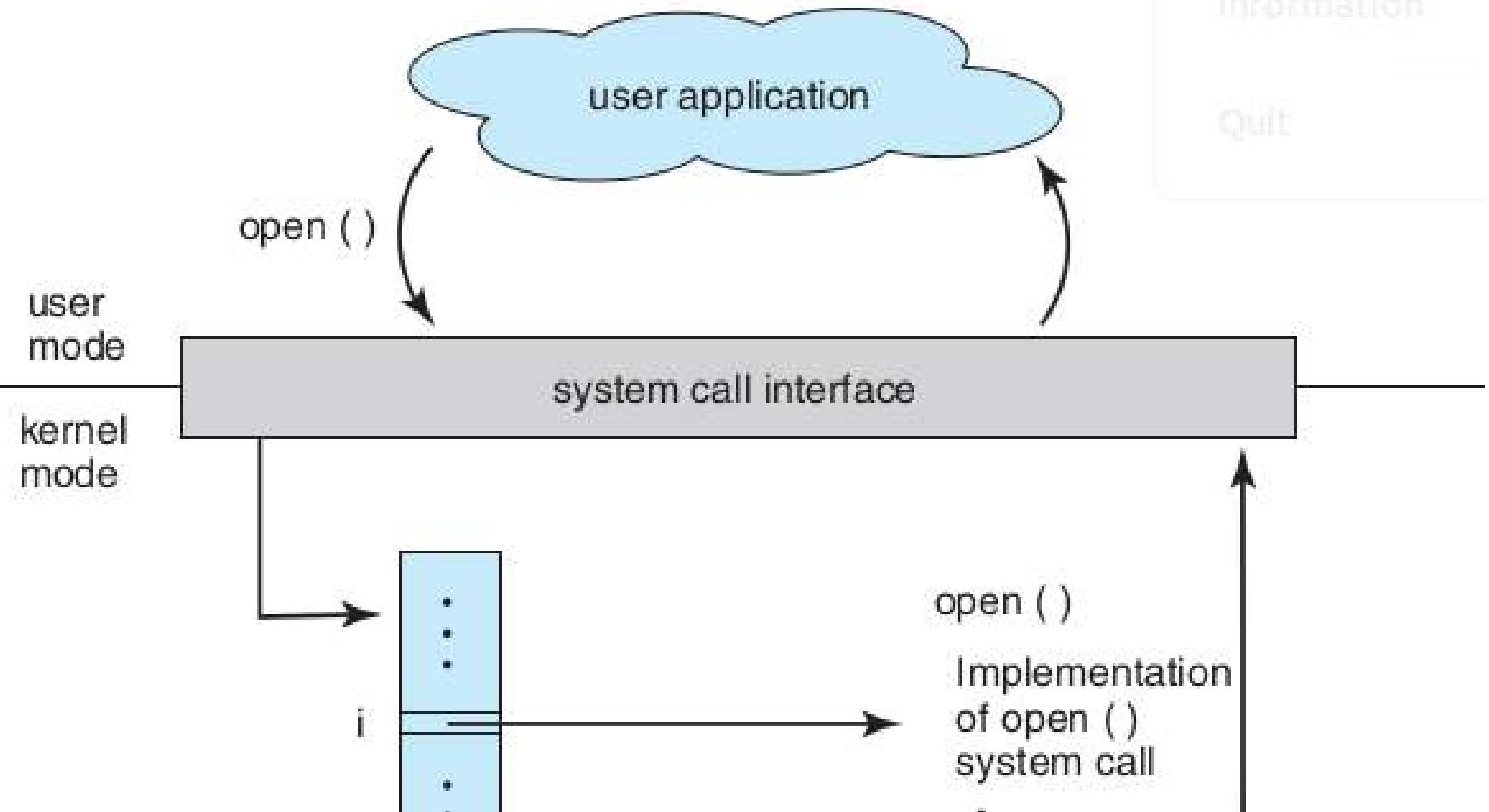
//OS code

```
int sys_write(int fd,
char *, int len) {
```

figure out location on  
disk

information

Quit



Two important system calls  
Related to processes

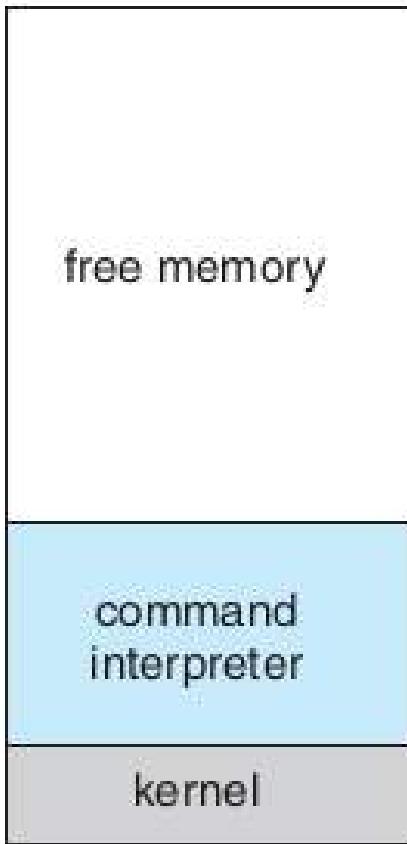
**fork() and exec()**

# Process

- A program in execution
- Exists in RAM
- Scheduled by OS
  - In a timesharing system, intermittantly scheduled by allocating a time quantum, e.g. 20 microseconds
- The “ps” command on Linux

# Process in RAM

- **Memory is required to store the following components of a process**
  - Code
  - Global variables (data)
  - Stack (stores local variables of functions)
  - Heap (stores malloced memory)
  - Shared libraries (e.g. code of printf, etc)

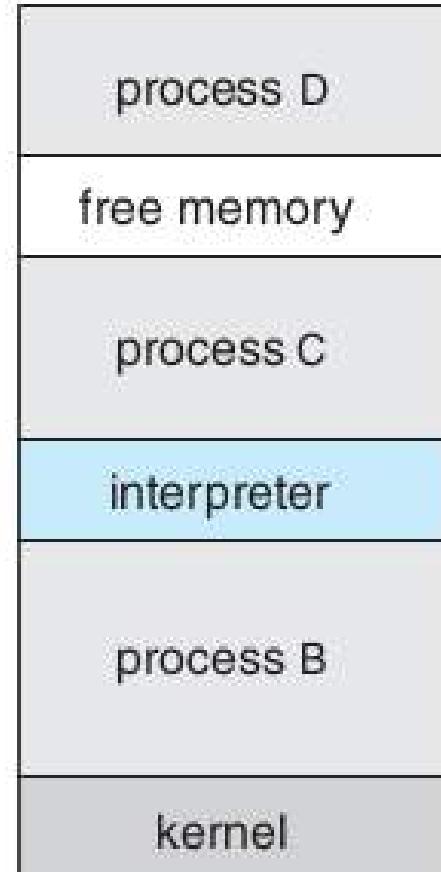


(a)



(b)

**Figure 2.9** MS-DOS execution. (a) At system startup. (b) Running a program.



# fork()

- A running process creates it's duplicate!
- After call to fork() is over
  - Two processes are running
  - Identical
  - The calling function returns in two places!
  - Caller is called parent, and the new process is called child

# **exec()**

- **Variants: execvp(), execl(), etc.**
- **Takes the path name of an executable as an argument**
- **Overwrites the existing process using the code provided in the executable**
- **The original process is OVER ! Vanished!**

**The new program starts running**

# Shell using fork and exec

- Demo
- The only way a process can be created on Unix/Linux is using `fork()` + `exec()`
- All processes that you see were started by some other process using `fork()` + `exec()` , except the initial “*init*” process
- When you click on “firefox” icon, the user-

# The boot process, once again

- BIOS
- Boot loader
- OS – kernel
- Init created by kernel by Hand(kernel mode)
- Kernel schedules init (the only process)
- Init fork-execs some programs (user mode)

# **Event Driven kernel Multi-tasking, Multi-programming**

# Earlier...

- **Boot process**

- BIOS -> Boot Loader -> OS -> “init” process

- **CPU doing**

**for(;;) {**

**fetch from @PC;**

**deode+execute;**

# Understanding hardware interrupts

- **Hardware devices (keyboard, mouse, hard disk, etc) can raise “hardware interrupts”**
- **Basically create an electrical signal on some connection to CPU (/bus)**
- **This is notified to CPU (in hardware)**
- **Now CPU’s normal execution is interrupted!**
  - What’s the normal execution?

# Understanding hardware interrupts

## □ On Hardware interrupt

- The PC changes to a location pre-determined by CPU manufacturers!
- Now CPU resumes normal execution
  - What's normal?
  - Same thing: Fetch, Decode, Execute, Change PC, repeat!
  - But...

# Boot Process

- BIOS runs “automatically” because at the initial value of PC (on power ON), the manufacturers have put in the BIOS code in ROM
  - CPU is running BIOS code . BIOS code also occupies all the addresses corresponding to hardware interrupts.
- BIOS looks up boot loader (on default boot

# Hardware interrupts and OS

- **When OS starts running initially**
  - It copies relevant parts of it's own code at all possible memory addresses that a hardware interrupt can lead to!
  - Intelligent, isn't' it?
- **Now what?**
  - Whenever there is a hardware interrupt – what will happen?

# Key points

- **Understand the interplay of hardware features + OS code + clever combination of the two to achieve OS control over hardware**
- **Most features of computer systems / operating systems are derived from hardware features**
  - We will keep learning this through the course

# Time Shared CPU

- **Normal use: after the OS has been loaded and Desktop environment is running**
- **The OS and different application programs keep executing on the CPU alternatively (more about this later)**

The CPU is time shared between

# Multiprogramming

## □ Program

- Just a binary (machine code) file lying on the **hard drive**. E.g. /bin/ls
- Does not do anything!

## □ Process

- A program that is executing
- Must **exist in RAM** before it executes. **Why?**

# Multiprogramming

# □ Multiprogramming

- A system where multiple processes(!) exist at the same time in the RAM
  - But only one runs at a time!
    - Because there is only one CPU

## □ Multi tasking

- Time sharing between multiple processes in a multi-tasking system

# Question

- Select the correct one
  - 1) A multiprogramming system is not necessarily multitasking
  - 2) A multitasking system is not necessarily multiprogramming

# **Events , that interrupt CPU's functioning**

- Three types of “traps” : Events that make the CPU run code at a pre-defined address

**1) Hardware interrupts**

**2) Software interrupt instructions (trap)**

**E.g. instruction “int”**

**3) Exceptions**

# Multi tasking requirements

- **Two processes should not be**
  - Able to steal time of each other
  - See data/code of each other
  - Modify data/code of each other
  - Etc.
- **The OS ensures all these things. How?**
  - To be seen later.

**But the OS is “always” “running”  
“in the background”  
Isn’t it?**

**Absolutely No!**

**Let's understand  
What kind of  
Hardware, OS interplay  
makes  
Multitasking possible**

# **Two types of CPU instructions and two modes of CPU operation**

- **CPU instructions can be divided into two types**
- **Normal instructions**
  - mov, jmp, add, etc.
- **Privileged instructions**
  - Normally related to hardware devices
  - E.g.

# **Two types of CPU instructions and two modes of CPU operation**

- **CPUs have a mode bit (can be 0 or 1)**
- **The two values of the mode bit are called:  
User mode and Kernel mode**
- **If the bit is in user mode**
  - Only the normal instructions can be executed by CPU
- **If the bit is in kernel mode**

# **Two types of CPU instructions and two modes of CPU operation**

- **The operating system code runs in kernel mode.**
  - How? Wait for that!
- **The application code runs in user mode**
  - How? Wait !
  - So application code can not run privileged hardware instructions

# Software interrupt instruction

- E.g. INT on x86 processors
- Does two things at the same time!
  - Changes mode from user mode to kernel mode in CPU
  - + Jumps to a pre-defined location! Basically changes PC to a pre-defined value.
    - Close to the way a hardware interrupt works. Isn't it?

# Software interrupt instruction

- **What's the use of these type of instructions?**
  - An application code running INT 0x80 on x86 will now cause
    - Change of mode
    - Jump into OS code
  - Effectively a request by application code to OS to do a particular task!
  - E.g. read from keyboard or write to screen !

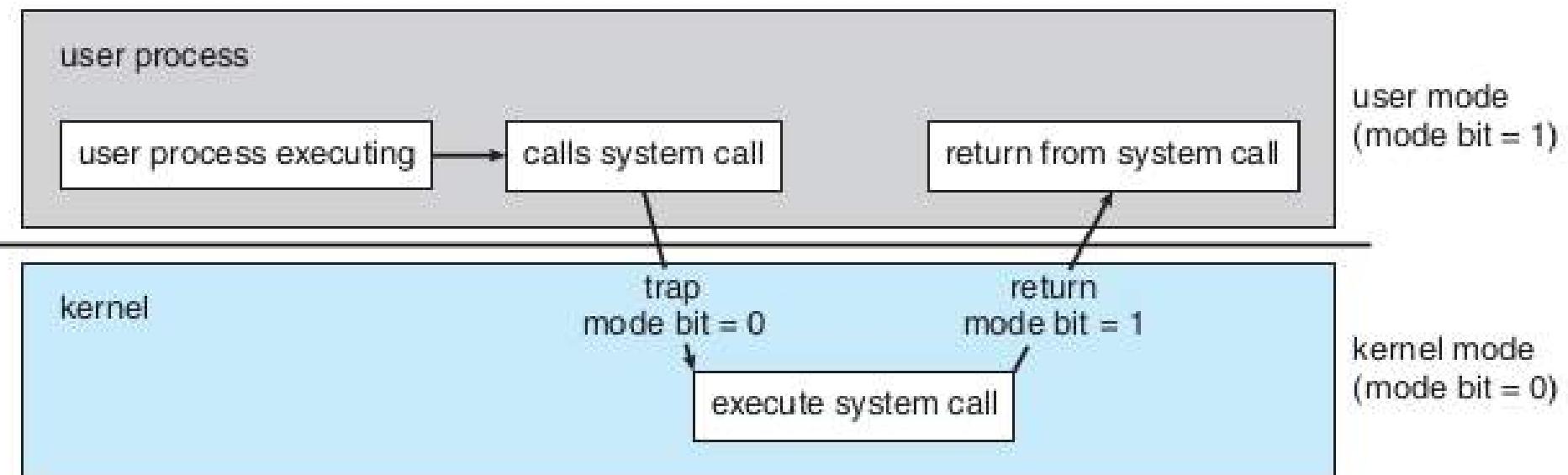


Figure 1.10 Transition from user to kernel mode

# Software interrupt instruction

- How does application code run INT instruction?
  - C library functions like printf(), scanf() which do I/O requests contain the INT instruction!
  - Control flow
    - Application code -> printf -> INT -> OS code -> back to printf code -> back to application code

# Example: C program

```
int main() {
 int i, j, k;
 k = 20;
 scanf("%d", &i); // This jumps into OS and
 returns back
 j = k + i;
```

# Interrupt driven OS code

- **OS code is sitting in memory , and runs intermittantly . When?**
  - On a software or hardware interrupt or exception!
  - Event/Interrupt driven OS!
  - Hardware interrupts and exceptions occur asynchronously (un-predictedly) while software interrupts are caused by application code

# Interrupt driven OS code

- **Timer interrupt and multitasking OS**
  - Setting timer register is a privileged instruction.
  - After setting a value in it, the timer keeps ticking down and on becoming zero the timer interrupt is raised again (in hardware automatically)
  - OS sets timer interrupt and “passes control” over to some application code. Now only application code running on CPU !

# What runs on the processor ?

- 4 possibilities.

|           |  | process context             |                                   |                                     |
|-----------|--|-----------------------------|-----------------------------------|-------------------------------------|
|           |  | application<br>(user) code  | system calls,<br>exceptions       |                                     |
| user mode |  | (access process space only) | (access process and system space) | kernel mode                         |
|           |  |                             |                                   |                                     |
|           |  | <i>not allowed</i>          |                                   | <i>interrupts,<br/>system tasks</i> |

# **Chapter-I and II**

## **Introduction to OS**

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

# OS or kernel?

- **Debate on terminology**
- **What you use in daily life is not kernel, but**
  - GUI, Shell, Applications, ...
  - One view: OS = kernel + (GUI, Shell, Libraries, System programs, Minimum Applications,...)
  - Correct, IMO!
- **What we study in this course is kernel**

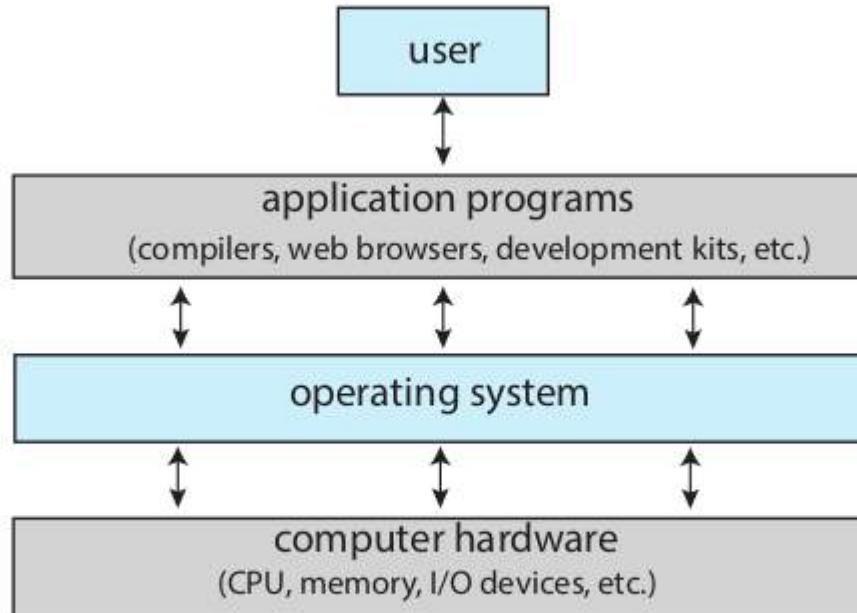
# Building an “OS”

## □ E.g. Debian

- A collection of thousands of applications, libraries, system programs, ... and Linux kernel !
- Linux kernel is at the heart, but heart is not a human without the body !
- Job of “Debian Developers”
  - Collect the source code of all things you want
  - Create an “Environment” for compiling things : bootstrap

# Components of a computer system

## Abstract view

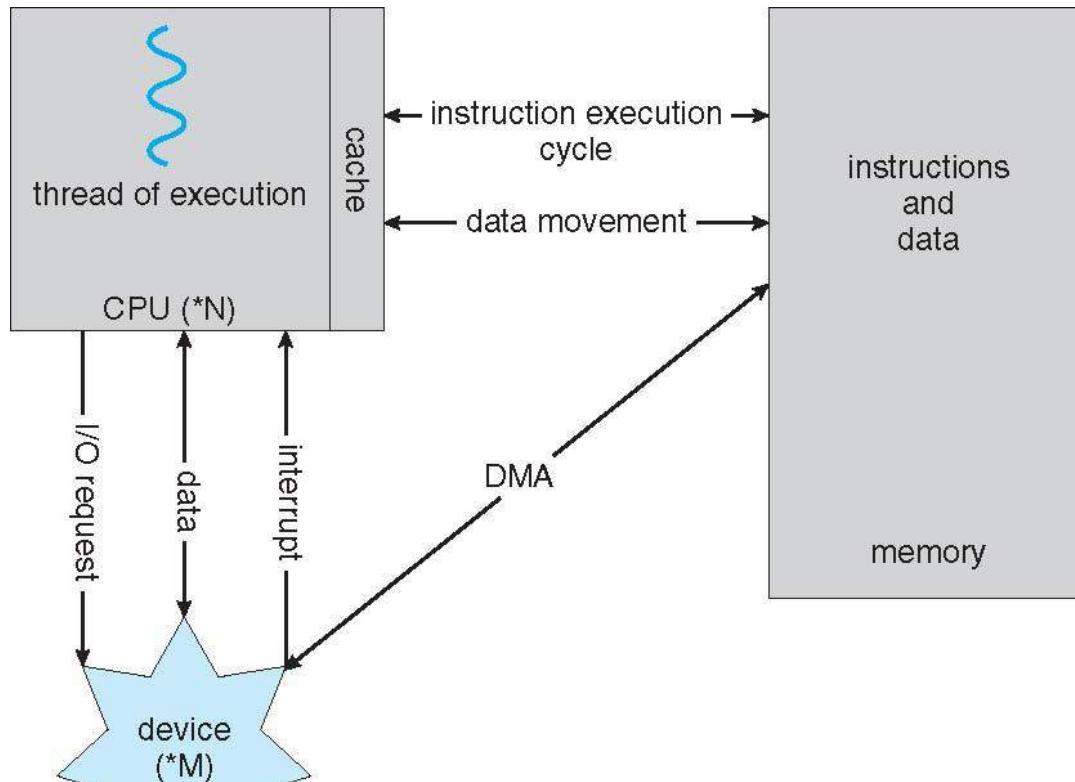


**Figure 1.1** Abstract view of the components of a computer system.

# Today, different type of kernels to serve different needs

- **Each “Computing Environment” has a different requirement of managing resources, processes, etc**
  - Desktop / Laptop
  - Thin Clients
  - Workstations
  - Handheld
  - “Servers”

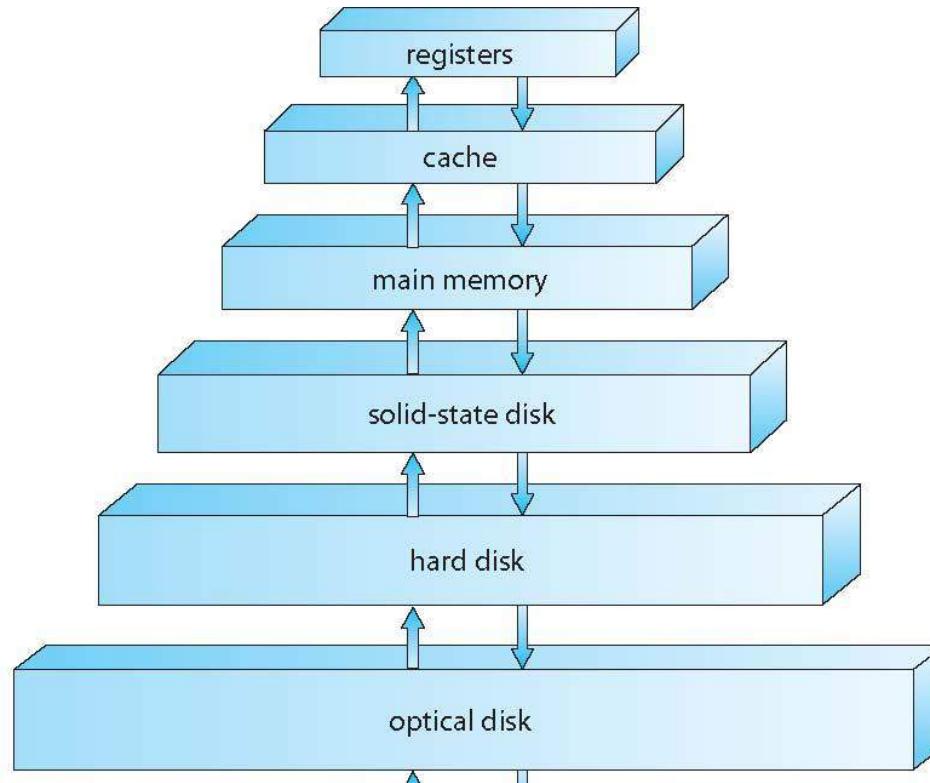
# Von Neumann architecture



# A key to “understanding”

- **Everything happens on processor**
- **Processor is always running some instruction**
- **We should be able to tell possible execution sequences on processor**

# Memory hierarchy



# Memory hierarchy

| Level                     | 1                                         | 2                                | 3                | 4                | 5                |
|---------------------------|-------------------------------------------|----------------------------------|------------------|------------------|------------------|
| Name                      | registers                                 | cache                            | main memory      | solid state disk | magnetic disk    |
| Typical size              | < 1 KB                                    | < 16MB                           | < 64GB           | < 1 TB           | < 10 TB          |
| Implementation technology | custom memory with multiple ports<br>CMOS | on-chip or off-chip<br>CMOS SRAM | CMOS SRAM        | flash memory     | magnetic disk    |
| Access time (ns)          | 0.25 - 0.5                                | 0.5 - 25                         | 80 - 250         | 25,000 - 50,000  | 5,000,000        |
| Bandwidth (MB/sec)        | 20,000 - 100,000                          | 5,000 - 10,000                   | 1,000 - 5,000    | 500              | 20 - 150         |
| Managed by                | compiler                                  | hardware                         | operating system | operating system | operating system |

# System calls

- **We have seen**

- Fork(), exec(): system calls to create processes
  - Open(), read(), write() : system calls to play with files

- **System call: A service given by kernel**

# System calls: the problem

- **On a time shared system**

- Applications and kernel run on the CPU taking turns
- Kernel allocates time to application , by setting a value in timer register, and then makes application run
  - Timer ticks down every clock cycle
- *Timer hardware interrupt* occurs when timer = 0 , that is time of application is over
  - Kernel runs again

# System calls: the problem

- **Kernel must provide “services” to applications**
  - Create processes
  - Grant access to resources to processes
  - Etc
- **Process should be able to “call” kernel’s service**
- **But everything runs on processor!**
  - That is both processes and kernel

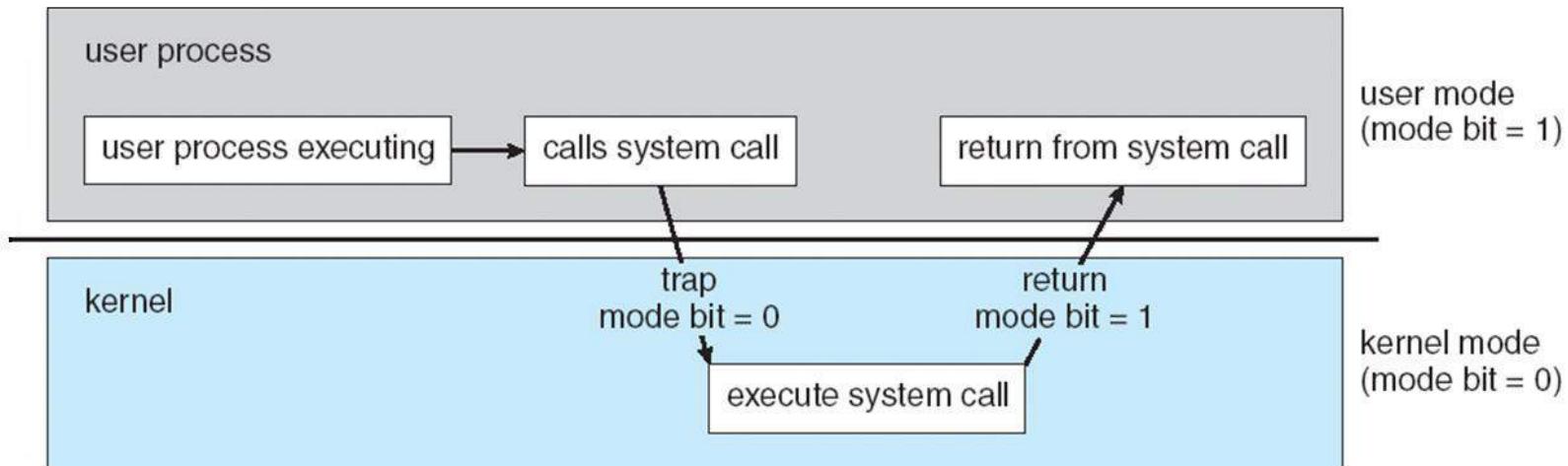
# System calls: the problem

- **Privileged Instructions**
  - Set the timer register
  - Set a value in registers of I/O devices
  - Set the MMU
  - Etc
- **How to ensure that only kernel code can run privileged instructions?**

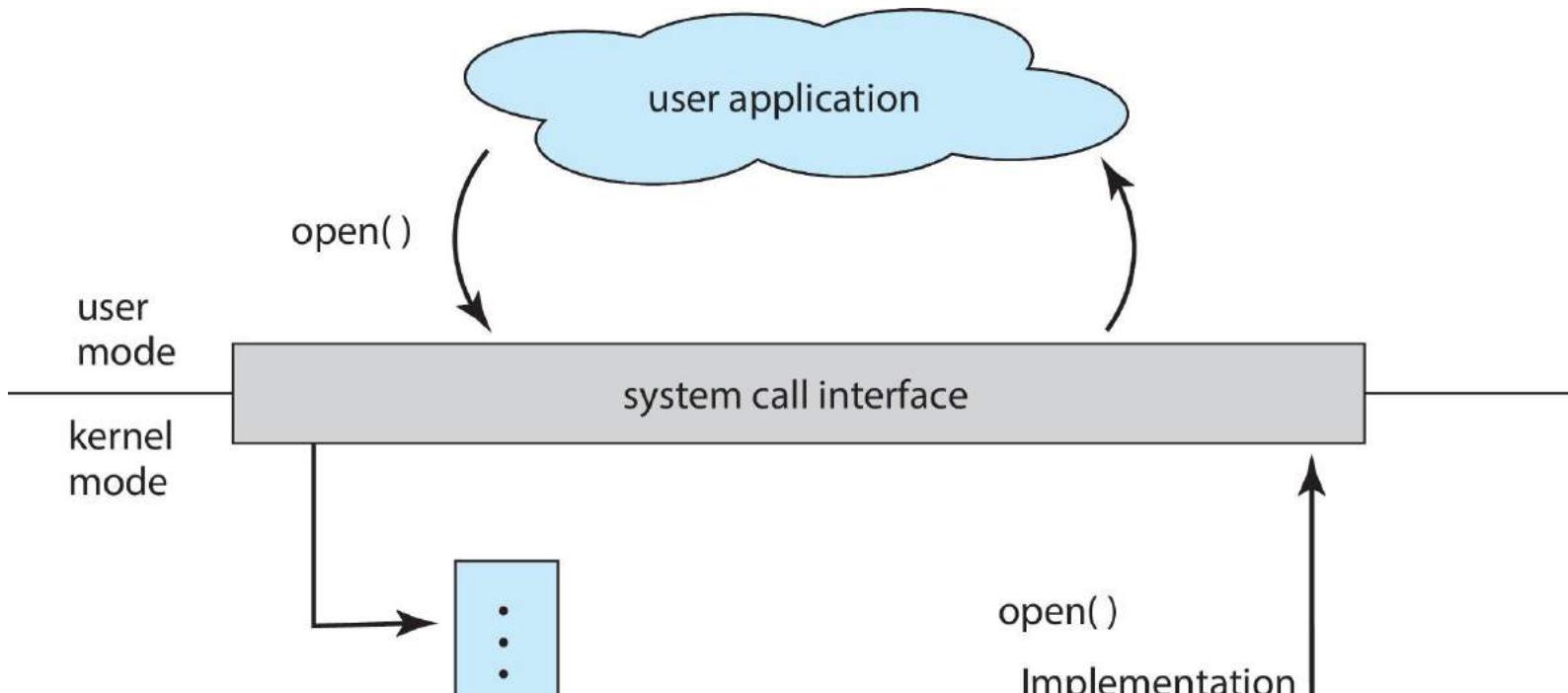
# System calls: the problem

- **CPU: 2 modes**
  - User mode and kernel mode
  - All instructions can be executed in kernel mode
  - Non-privileged instructions can be executed in User mode
  - How to ensure this?
- **Special instruction: software interrupt instruction**

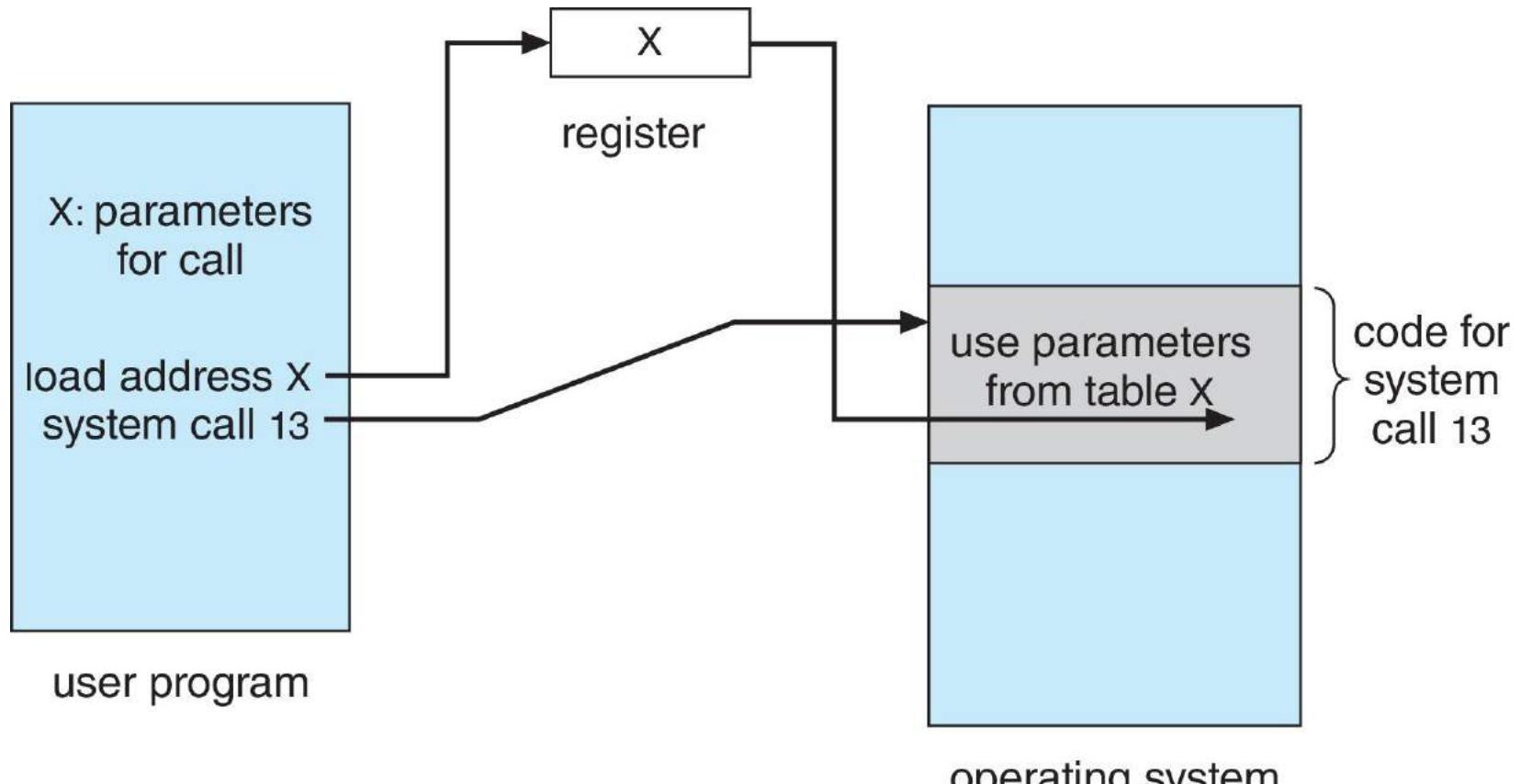
# 2 modes of operation of CPU needed for multi-tasking system



# Application – kernel interaction in system calls



# System call – parameter passing



# What a modern multi-processor, multi-tasking kernel does

- **Process Management**

- Create an environment (fork, exec, exit, etc) so that processes can be created, exited, ...

- **Resource management**

- Processes can access “resources” only through kernel
  - System calls like: open(), brk(), ...
  - Includes storage management – enable a “tree” type

# System calls exist for ...

## □ Process control

- create process (fork), terminate process (exit)
- end, abort (kill)
- load, execute (exec)
- get process attributes (getpid, ..), set process attributes (setrlimit, ...)
- wait for time (sleep, ...)

# System calls exist for ...

- **File management**

- create file (open, create), delete file (unlink)
- open, close (close()) file
- read, write, reposition (read, write, lseek, ...)
- get and set file attributes (chown, chmod, stat, ... )

- **Device management**

request device, release device

# System calls exist for ...

- **Information maintenance**

- get time or date, set time or date

- get system data, set system data

- get and set process, file, or device attributes

- time, fcntl, ...

- **Communications**

- create, delete communication connection

- send/receive messages if message passing module is loaded

# System calls exist for ...

- **Protection**

- Control access to resources

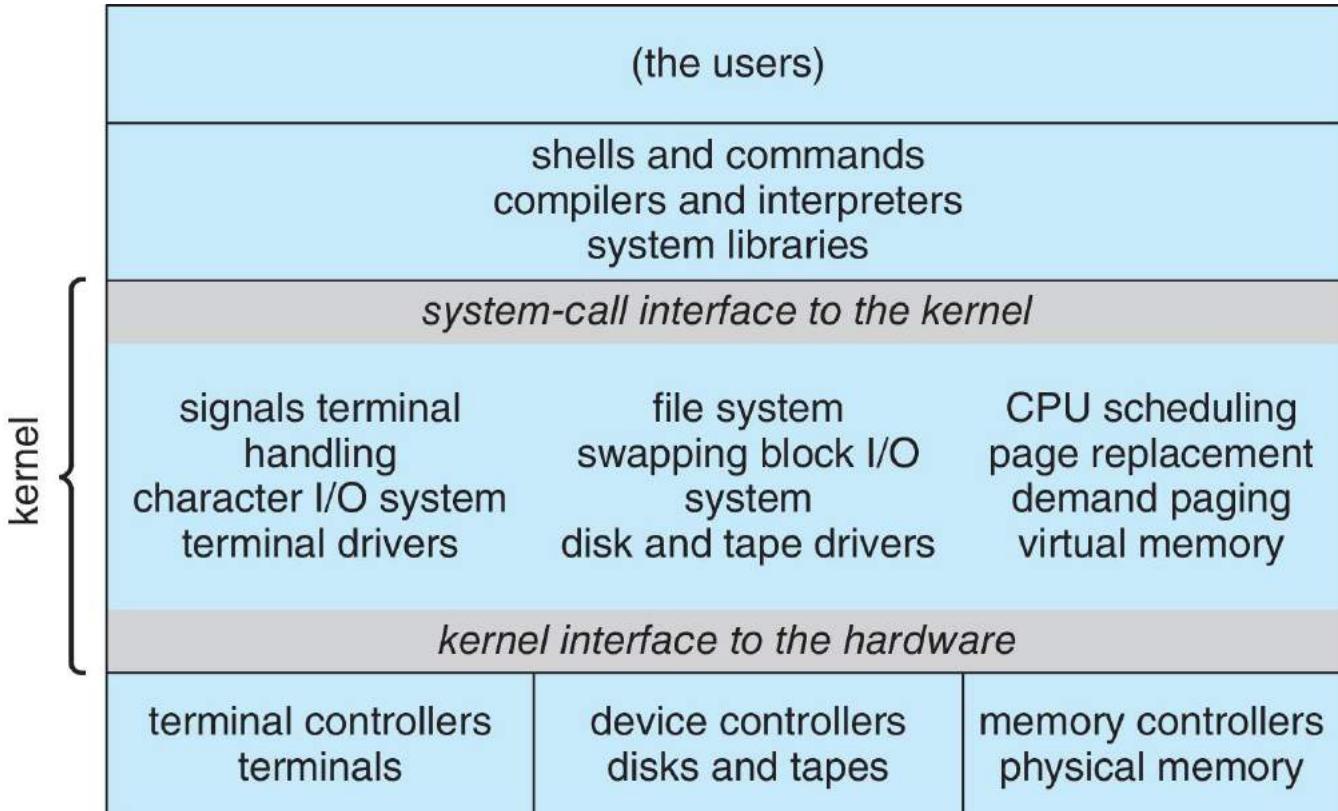
- Get and set permissions

- Allow and deny user access

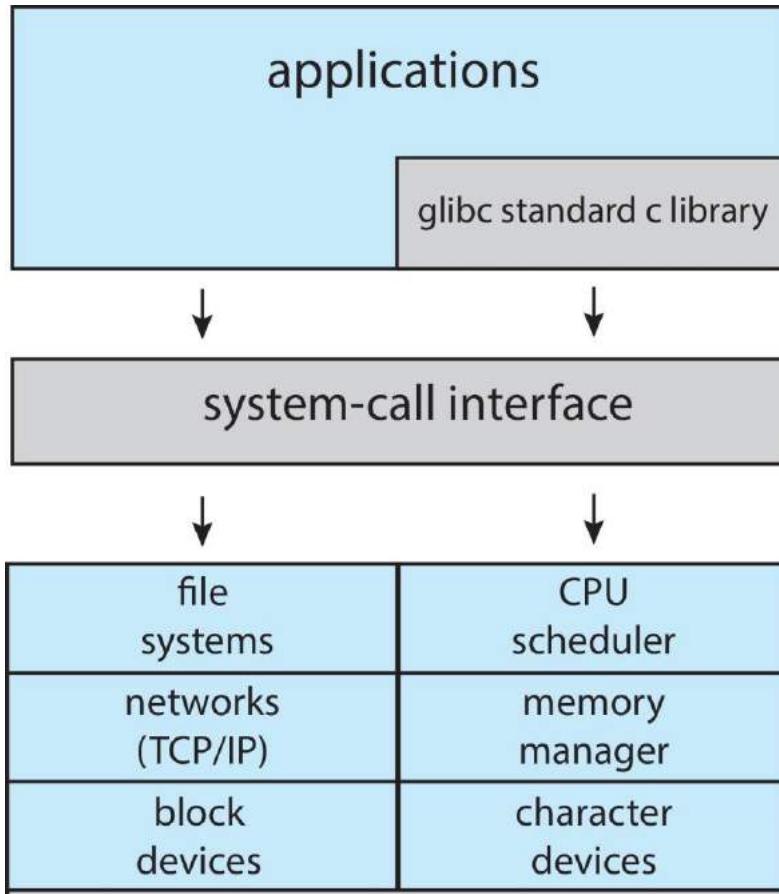
- **More**

- On LinuxL “man syscalls”
    - POSIX: Portable Operating System Interface

# Traditional Unix system structure



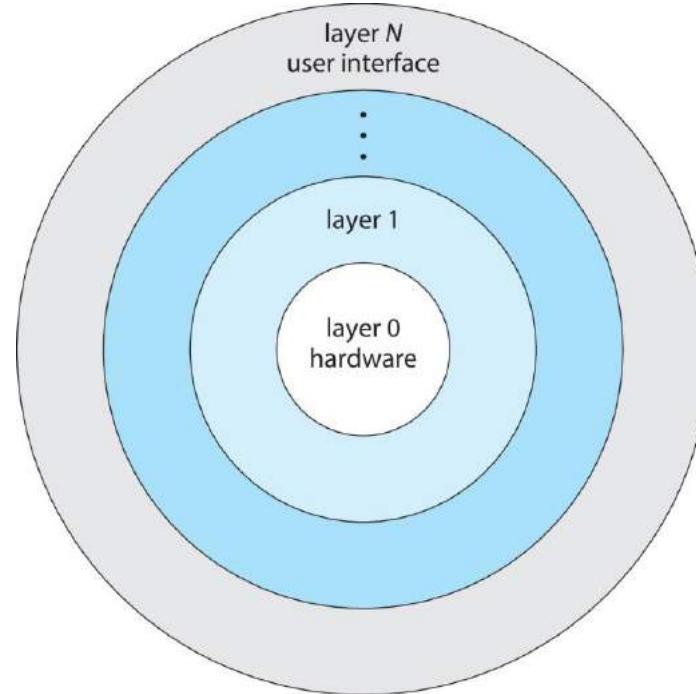
# Linux system structure



- **Linux kernel is extendible**
  - Kernel modules
  - lsmod, insmod, ..

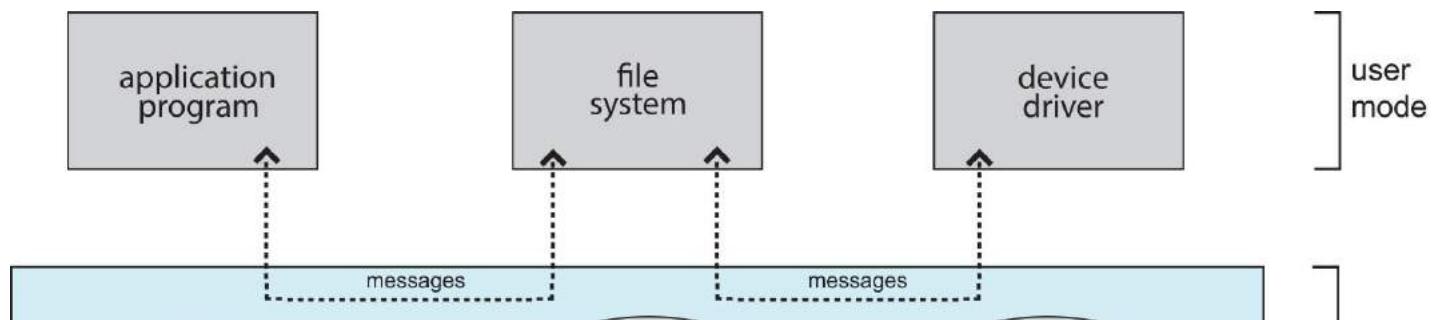
# Layered approach

□ xv6



# Microkernel approach

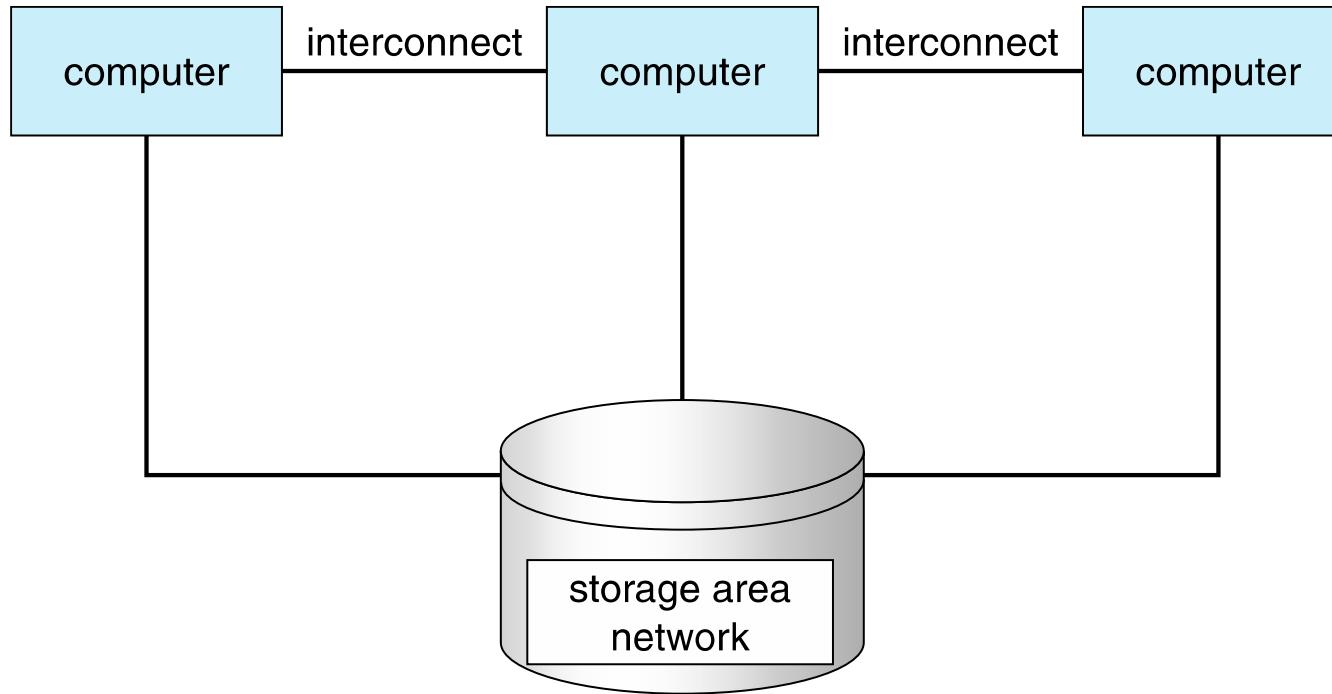
- Mach, Mac OS X (based on Mach)
- GNU herd
- Against “monolithic” approach (Linux kerenal)



# Skipped

- **IOS, Android structure**

# Clustered system



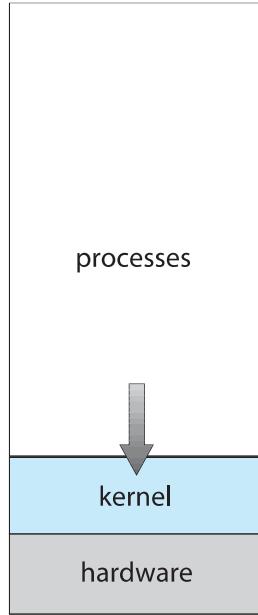
# Free Software / Open Source kernels

- **Linux**
- **BSD Unix**
- **GNU Herd**
- **xv6**
- **Minix**
- **FreeRTOS**

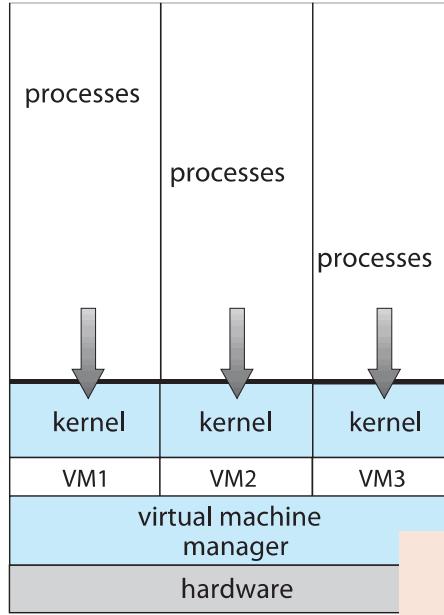
# Virtualization

- A general concept!
- Create a “virtual reality” out of “reality”
  - Black boxes that do something
  - End users need not know “how”, but only “what”
- Following all are “virtualizations”
  - Function
  - Class

# Virtual Machines, an example

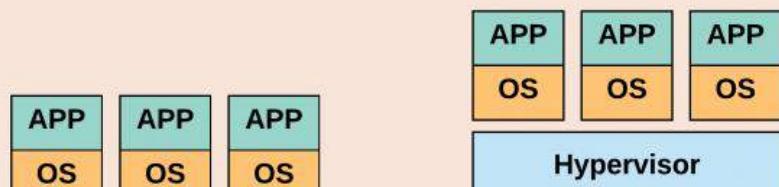


(a)



(b)

## Types of Hypervisor



# **Introduction to Linux and Basics of System Calls**

Abhijit A. M.  
[abhijit.comp@coep.ac.in](mailto:abhijit.comp@coep.ac.in)

# **Why GNU/Linux ?**

# Few Key Concepts

- ***Files don't open themselves***
  - Always some application/program open()s a file.
- ***Files don't display themselves***
  - A file is displayed by the program which opens it. Each program has its own way of handling files

# Few Key Concepts

- **Programs don't run themselves**
  - You click on a program, or run a command --> equivalent to request to Operating System to run it. The OS runs your program
- **Users (humans) request OS to run programs, using Graphical or Command line interface**
  - and programs open files

# Path names

- Tree like directory structure
- Root directory called /
- A complete path name for a file
  - /home/student/a.c
- Relative path names

**concept: every running program has a *current* directory**

# A command

- **Name of an executable file**
  - For example: 'ls' is actually “/bin/ls”
- **Command takes arguments**
  - E.g. ls /tmp/
- **Command takes options**
  - E.g. ls -a

# A command

- **Command can take both arguments and options**
  - E.g. ls -a /tmp/
- **Options and arguments are basically argv[] of the main() of that program**

# Basic Navigation Commands

- **pwd**
- **ls**
  - ls -l
  - ls -l /tmp/
  - ls -l /home/student/Desktop
  - ls -l ./Desktop
  - ls -a
  - \ls -F
- **cd**
  - cd /tmp/

Map these commands  
to navigation using a  
graphical file browser

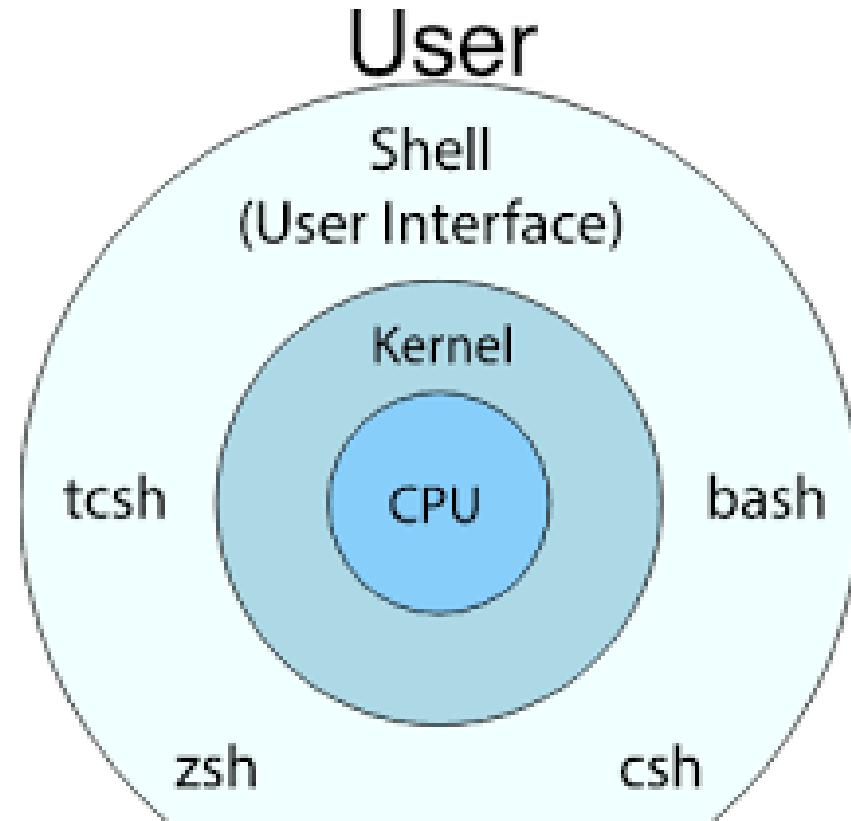
# Before the command line, the concept of Shell and System calls

## □ **System Call**

- A function from OS code
- Does specific operations with hardware (e.g. reading from keyboard, writing to screen, *opening* a file from disk, etc.)
- Applications can't access hardware directly, they have to request the OS using system calls
- Examples

# The Shell

- **Shell = Cover**
- **Covers some of the Operating System's "System Calls" (mainly fork+exec) for the *Applications***
- **Talks with Users and Applications**



# The Shell

Shell waits for user's input

Requests the OS to run a program which the user  
has asked to run

Again waits for user's input

# Let's Understand fork() and exec()

```
#include <unistd.h>

int main() {
 fork();
 printf("hi\n");
 return 0;
}
```

```
#include <unistd.h>

int main() {
 printf("hi\n");
 execl("/bin/ls", "ls",
 NULL);
 printf("bye\n");
 return 0;
}
```

# A simple shell

```
#include <stdio.h>
#include <unistd.h>
int main() {
char string[128];
int pid;
while(1) {
printf("prompt>");
scanf("%s", string);
pid = fork();
if(pid == 0) {
execl(string, string, NULL);
} else {
```

# Users on Linux

## □ Root and others

- root
  - superuser, can do (almost) everything
  - Uid = 0
- Other users
  - Uid != 0
- UID, GID understood by kernel

# File Permissions on Linux

- **3 sets of 3 permission**
  - Octal notation: Read = 4, Write = 2, Execute = 1
  - 644 means
    - Read-Write for owner, Read for Group, Read for others
- **chmod command, used to change permissions, uses these notations**
  - It calls the chmod() system call

# File Permissions on Linux

-rw-r--r-- 1 abhijit abhijit 1183744 May 16 12:48 01\_linux\_basics.ppt

-rw-r--r-- 1 abhijit abhijit 341736 May 17 10:39 Debian Family Tree.svg

drwxr-xr-x 2 abhijit abhijit 4096 May 17 11:16 fork-exec

-rw-r--r-- 1 abhijit abhijit 7831341 May 11 12:13 foss.odp

3 sets of 3 permissions

Owner size name

# File Permissions on Linux

- **r on a file : can read the file**
  - open(... O\_RDONLY) works
- **w on a file: can modify the file**
  - open(... O\_WRONLY) works
- **x on a file: can ask the os to run the file as an executable program**
  - exec(...) works

# Access rights examples

- **-rw-r--r--**  
Readable and writable for file owner (actually a process started by the owner!), only readable for others
- **-rw-r----**  
Readable and writable for file owner, only readable for users belonging to the file group.
- **drwx-----**  
Directory only accessible by its owner
- **-----r-x**  
File executable by others but neither by your friends nor by yourself.  
Nice protections for a trap...

# Permissions: more !

- Setuid/setgid bit

\$ ls -l /usr/bin/passwd

-rwsr-xr-x 1 root root 68208 Nov 29 17:23 /usr/bin/passwd

- How to set the s bit?

- chmod u+s <filename>

- What does this mean?

- Any user can run this process, but the process itself runs as if run by the owner of the file

- passwd runs as if run by “root” even if you run it

# Man Pages

## □ Manpage

- \$ man ls
- \$ man 2 mkdir
- \$ man man
- \$ man -k mkdir

## □ Manpage sections

- 1 User-level cmds and apps
  - /bin/mkdir
- 2 System calls

- 4 Device drivers and network protocols
  - /dev/tty
- 5 Standard file formats
  - /etc/hosts
- 6 Games and demos
  - /usr/games/fortune
- 7 Misc. files and docs
  - man 7 locale
- 8 System admin. Cmds

# GNU / Linux filesystem structure

Not imposed by the system. Can vary from one system to the other, even between two GNU/Linux installations!

/ Root directory

/bin/ Basic, essential system commands

/boot/ Kernel images, initrd and configuration files

/dev/ Files representing devices

/dev/hda: first IDE hard disk

/etc/ System and application configuration files

/home/ User directories

/lib/ Basic system shared libraries

# GNU / Linux filesystem structure

**/lost+found** Corrupt files the system tried to recover

**/media** Mount points for removable media:

**/media/usbdisk, /media/cdrom**

**/mnt/** Mount points for temporarily mounted filesystems

**/opt/** Specific tools installed by the sysadmin

**/usr/local/** often used instead

**/proc/** Access to system information

**/proc/cpuinfo, /proc/version ...**

**/root/** root user home directory

**/sbin/** Administrator-only commands

**/sys/** System and device controls  
**(cpu frequency, device power, etc.)**

# GNU / Linux filesystem structure

**/tmp/** Temporary files

**/usr/** Regular user tools (not essential to the system)

**/usr/bin/, /usr/lib/, /usr/sbin...**

**/usr/local/** Specific software installed by the sysadmin  
(often preferred to /opt/)

**/var/** Data used by the system or system servers

**/var/log/, /var/spool/mail** (incoming mail),

**/var/spool/lpd** (print jobs)...

# Files: cut, copy, paste, remove,

- **cat <filenames>**

- cat /etc/passwd
- cat fork.c
- cat <filename1>  
<filename2>

- **cp <source> <target>**

- cp a.c b.c

- **mv <source> <target>**

- mv a.c b.c
- mv a.c /tmp/
- mv a.c /tmp/b.c

- **rm <filename>**

- rm a.c
- rm a.c b.c c.c
- rm -r /tmp/a

# Useful Commands

- **echo**

- echo hi
- echo hi there
- echo “hi there”
- j=5; echo \$j

- **sort**

- sort

- **grep**

- grep bash /etc/passwd
- grep -i display  
/etc/passwd
- egrep -i 'a|b' /etc/passwd

- **less <filename>**

- **head <filename>**

head -5 <filename>

# Useful Commands

- alias

**alias ll='ls -l'**

- tar

**tar cvf folder.tar folder**

- gzip

**gzip a.c**

- touch

**touch xy.txt**

- strings

**strings a.out**

- adduser

**sudo adduser test**

- su

**su administrator**

# Useful Commands

- **df**

- df -h**

- **du**

- du -hs .**

- **bc**

- **time**

- **date**

- **diff**

# Network Related Commands

- **ifconfig**
- **ssh**
- **scp**
- **telnet**
- **ping**
- **w**
- **last**
- **whoami**

# Unix job control

- Start a background process:
  - gedit a.c &
  - gedit
- ***hit* `ctrl-z`**
- **`bg`**
- Where did it go?
  - `jobs`
  - `ps`
- Terminate the job: kill it

# Configuration Files

- Most applications have configuration files in TEXT format
- Most of them are in */etc*
- */etc/passwd* and */etc/shadow*
  - Text files containing user accounts
- */etc/resolv.conf*
  - DNS configuration
- */etc/network/interfaces*
  - *Network configuration*
- */etc/hosts*

# `~/.bashrc` file

- `~/.bashrc`

**Shell script read each time a bash shell is started**

- **You can use this file to define**

- Your default environment variables (`PATH`, `EDITOR`...).
  - Your aliases.
  - Your prompt (see the `bash` manual for details).
  - A greeting message.
- **Also `~/.bash_history`**

# **Mounting**

# Partition

- **What is C:\ , D:\, E:\ etc on your computer ?**
  - “Drive” is the popular term
  - Typically one of them represents a CD/DVD RW
- **What do the others represent ?**
  - They are “partitions” of your “hard disk”

# Partition

- **Your hard disk is one contiguous chunk of storage**
  - Lot of times we need to “logically separate” our storage
  - Partition is a “logical division” of the storage
  - Every “drive” is a partition
- **A logical chunk of storage is partition**
  - Hard disk partitions (C:, D:), CD-ROM, Pen drive,

# Partitions

Disk Management

File Action View Help

Volume Layout Type File System Status Capacity Free Space % Fr

| Volume             | Layout    | Type  | File System | Status         | Capacity | Free Space | % Fr |
|--------------------|-----------|-------|-------------|----------------|----------|------------|------|
| (E:)               | Partition | Basic | FAT32       | Healthy        | 9.76 GB  | 8.37 GB    | 85 % |
| (F:)               | Partition | Basic | FAT32       | Healthy        | 9.76 GB  | 7.24 GB    | 74 % |
| OLDDRIVE (H:)      | Partition | Basic | FAT32       | Healthy (A...) | 4.99 GB  | 586 MB     | 11 % |
| WINDOWS XP (C:)    | Partition | Basic | FAT32       | Healthy (S...) | 9.76 GB  | 2.61 GB    | 26 % |
| Windows Vista (G:) | Partition | Basic | NTFS        | Healthy        | 8.00 GB  | 1.61 GB    | 20 % |
| XPBACKUP (I:)      | Partition | Basic | FAT32       | Healthy        | 4.99 GB  | 4.33 GB    | 86 % |
| XXCOPY (J:)        | Partition | Basic | FAT32       | Healthy        | 9.00 GB  | 4.32 GB    | 47 % |

Disk 0  
Basic  
37.30 GB  
Online

WINDOWS XP (C:) 9.77 GB FAT32 Healthy (System)

(E:) 9.77 GB FAT32 Healthy

(F:) 9.77 GB FAT32 Healthy

Windows Vista (G:) 8.00 GB NTFS Healthy

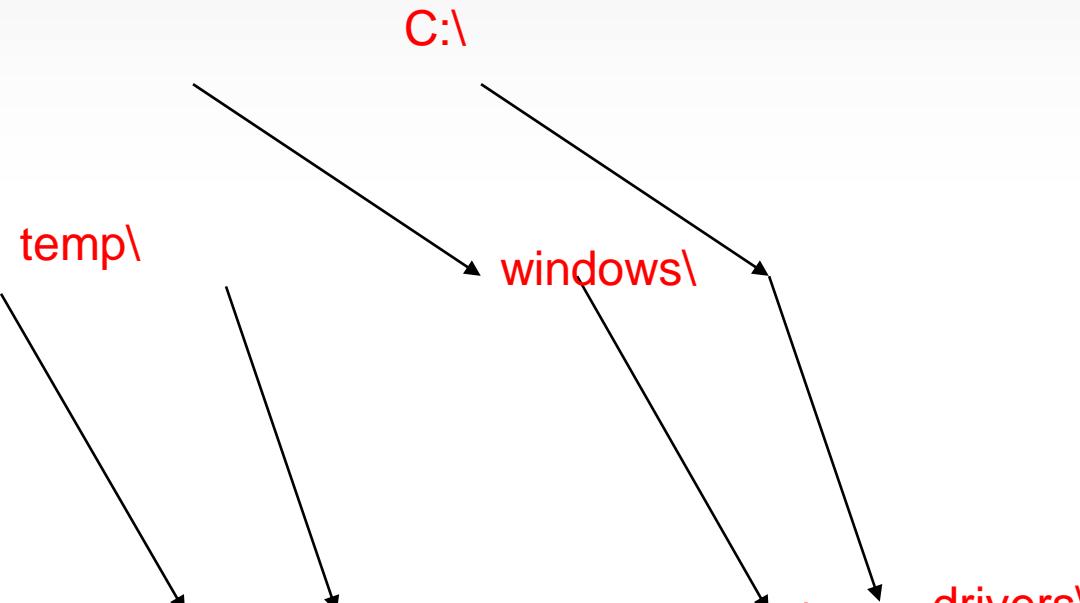
# Managing partitions and hard drives

- **System → Administration → Disk Utility**
- **Use gparted or fdisk to partition drives on Linux**
- **Hard drive partition names on Linux**
  - /dev/sda → Entire hard drive
  - /dev/sda1, /dev/sda2, /dev/sda3, .... Different partitions of the hard drive
  - Each partition has a *type* – ext4, ext3, ntfs, fat32, etc.

# Windows Namespace

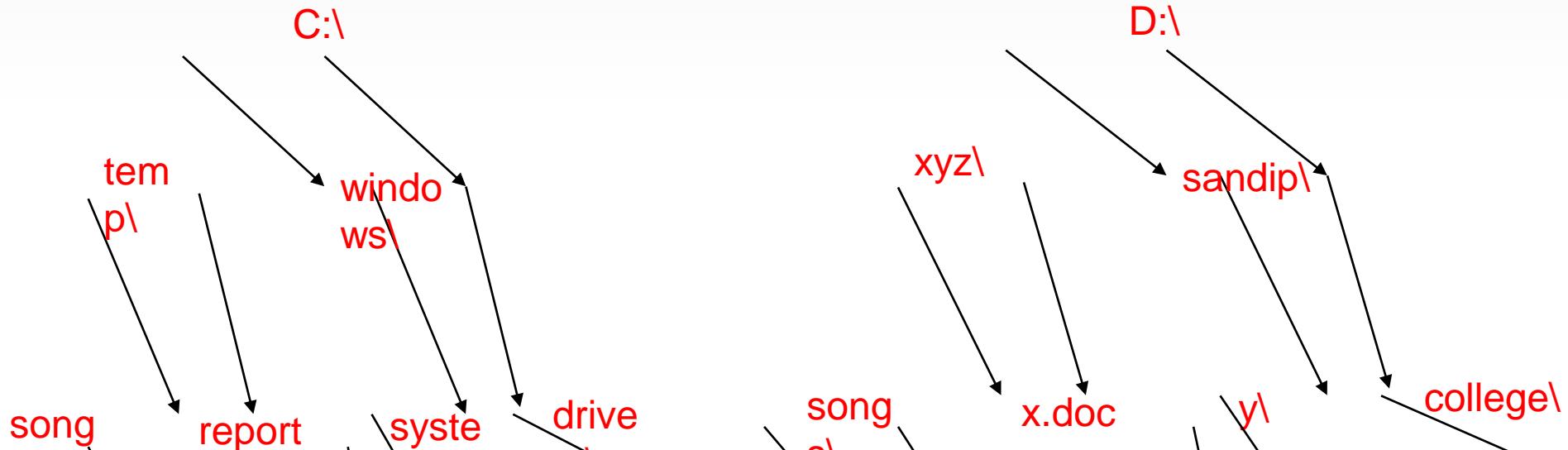
c:\temp\songs\xyz.mp3

- Root is C:\ or D:\ etc
- Separator is also “\”



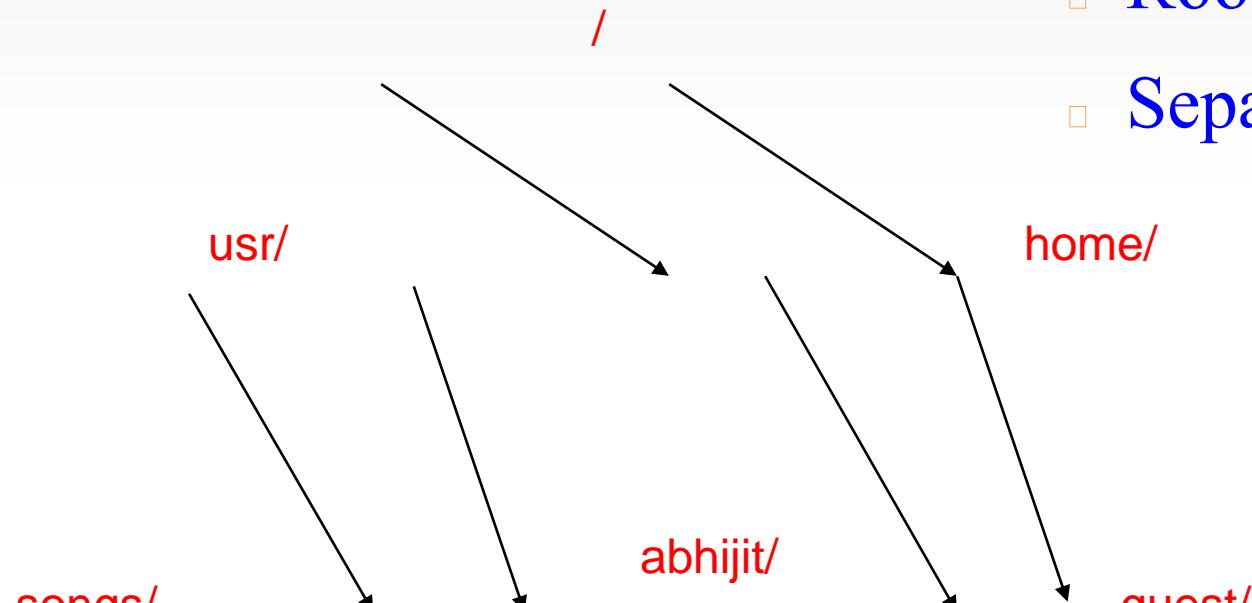
# Windows Namespace

- C:\ D:\ Are partitions of the disk drive
- Typical convention: C: contains programs, D: contains data
- One “tree” per partition
  - Together they make a “forest”



# Linux Namespace: On a partition

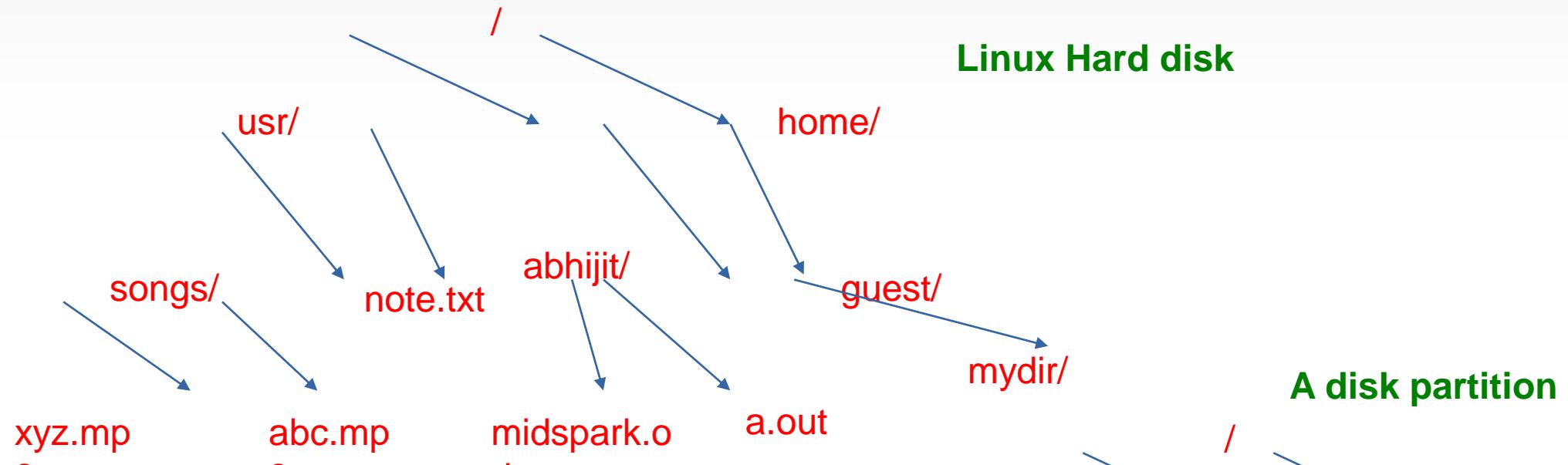
/usr/songs/xyz.mp3



- On every partition:
  - Root is “/”
  - Separator is also “/”

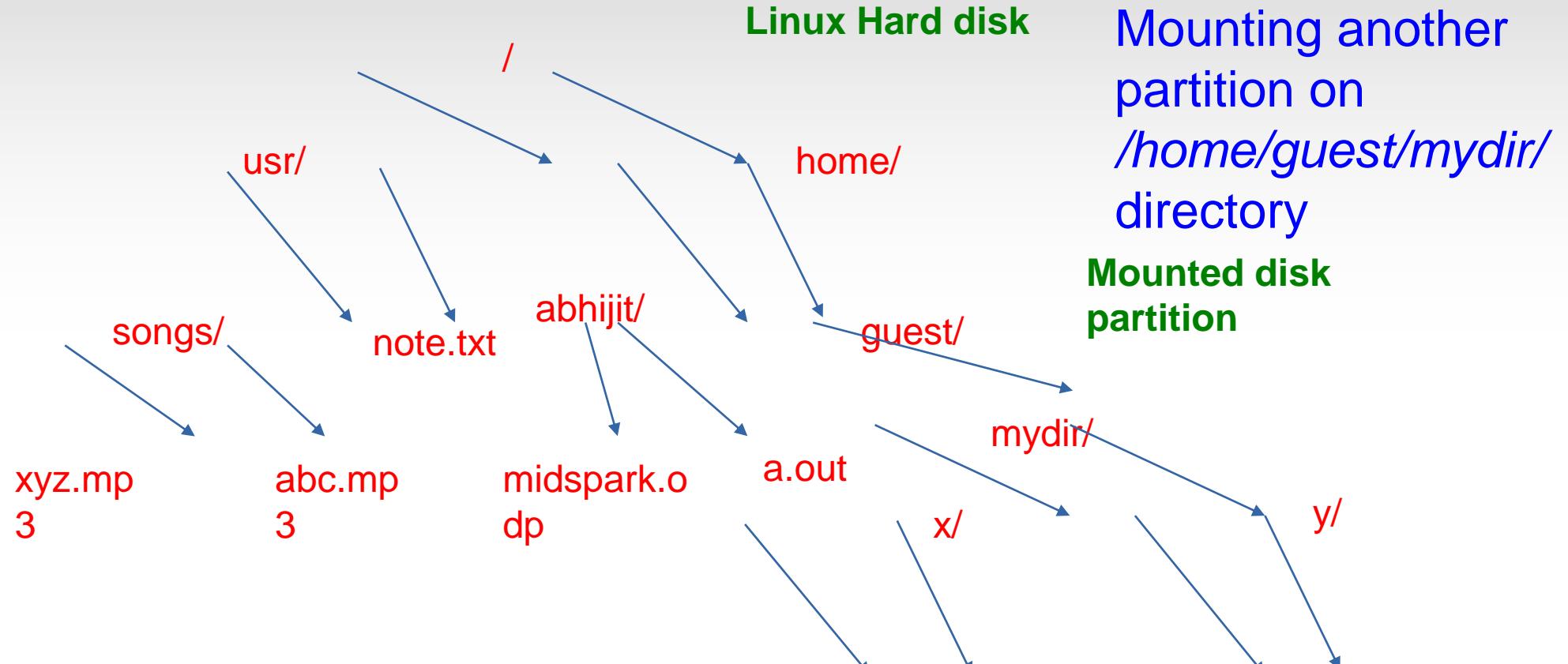
# Linux namespace: Mount

- Linux namespace is a single “tree” and not a “forest” like Windows
- Combining of multiple trees is done through “mount”



# Linux namespace

## Mounting a partition



# Mounting across network!

Using Network File System (NFS)

```
sudo apt install nfs-common
```

```
$ sudo mount 172.16.1.75:/mnt/data /myfolder
```

**Files that are not regular/directory**

# Special devices (1)

## Device files with a special behavior or contents

- **/dev/null**

The data sink! Discards all data written to this file.

Useful to get rid of unwanted output, typically log information:

```
mplayer black_adder_4th.avi &> /dev/null
```

- **/dev/zero**

Reads from this file always return \0 characters

Useful to create a file filled with zeros:

```
dd if=/dev/zero of=disk.img bs=1k count=2048
```

See `man null` or `man zero` for details

# Special devices (2)

- **/dev/random**

Returns random bytes when read. Mainly used by cryptographic programs. Uses interrupts from some device drivers as sources of true randomness (“entropy”).

Reads can be blocked until enough entropy is gathered.

- **/dev/urandom**

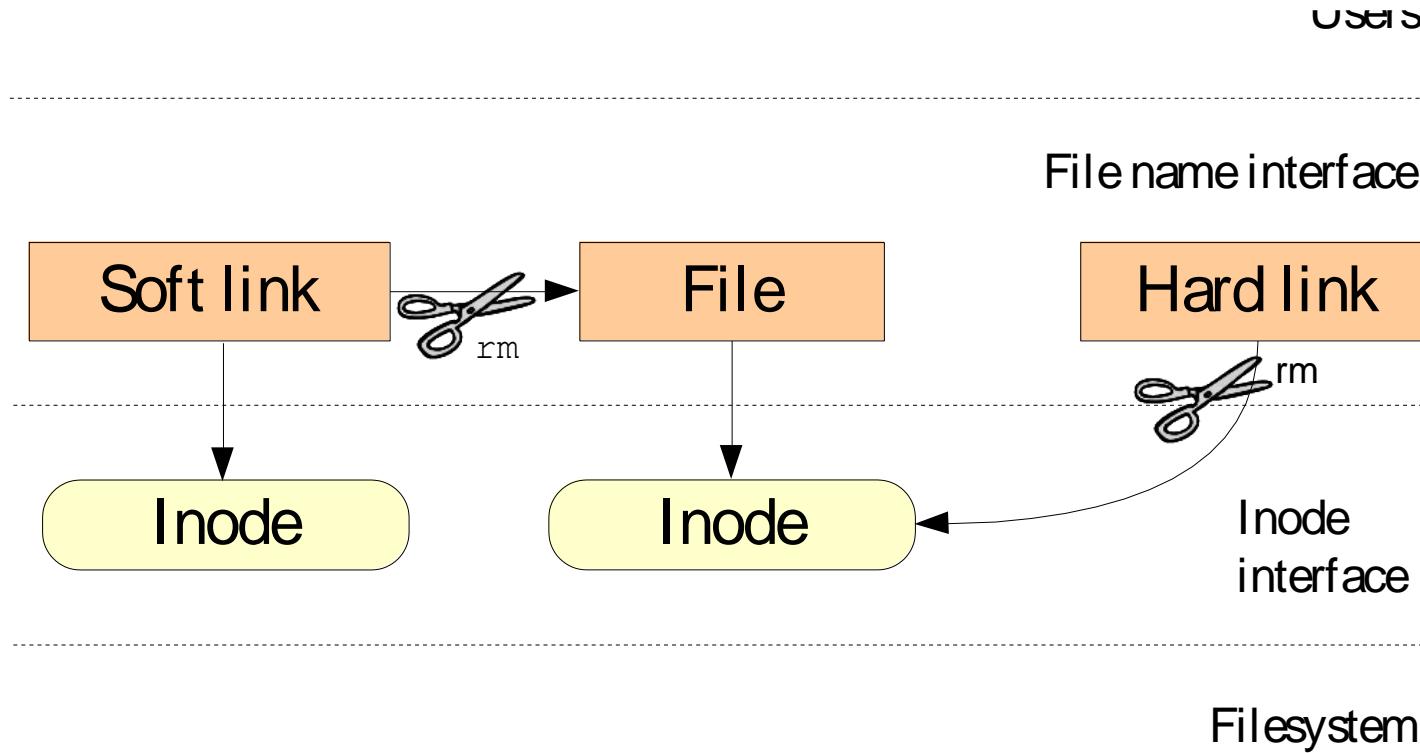
For programs for which pseudo random numbers are fine.

Always generates random bytes, even if not enough entropy is available (in which case it is possible, though still difficult, to predict future byte sequences from past ones).

See `man random` for details.

# Files names and inodes

## Hard Links Vs Soft Links



# Creating “links”

- Hard link

```
$ touch m
```

```
$ ls -l m
```

```
-rw-rw-r-- 1 abhijit abhijit 0 Jan 5 16:18 m
```

```
$ ln m mm
```

```
$ ls -l m mm
```

# **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

# Dynamic Linking on Linux

```
#include <stdio.h>

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

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

# **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)

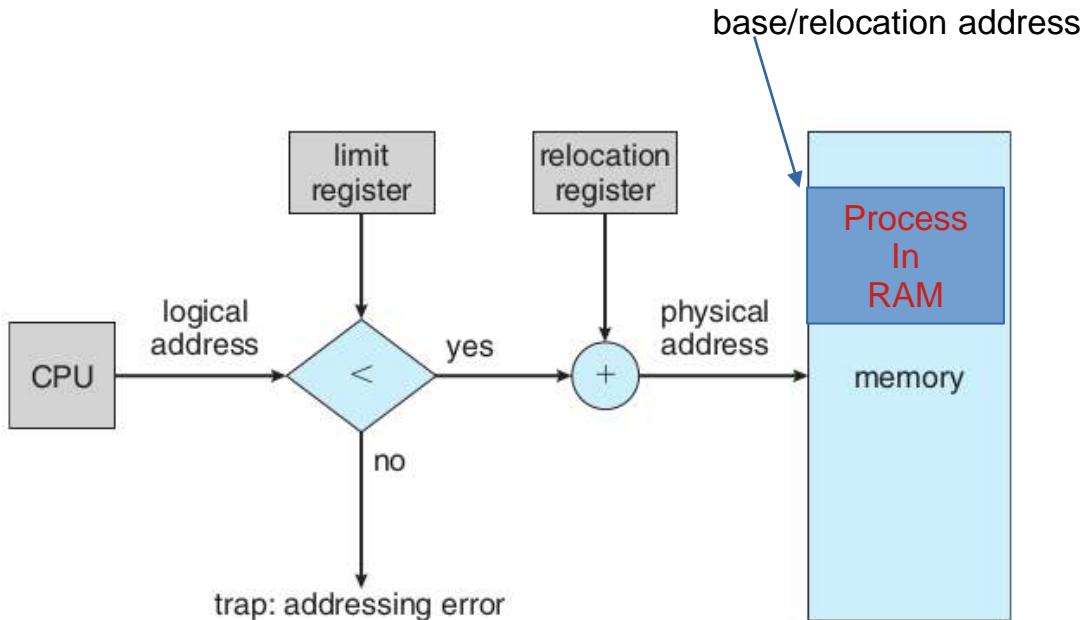


Figure 9.6 Hardware support for relocation and limit registers.

# **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**

# Variable partition scheme

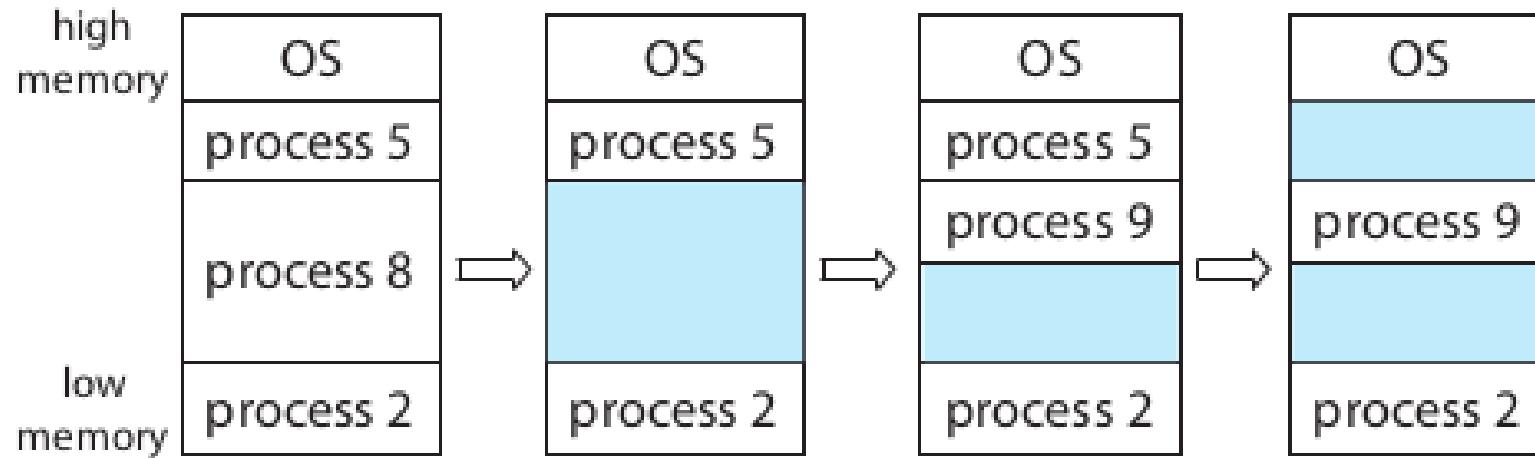


Figure 9.7 Variable partition.

# **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

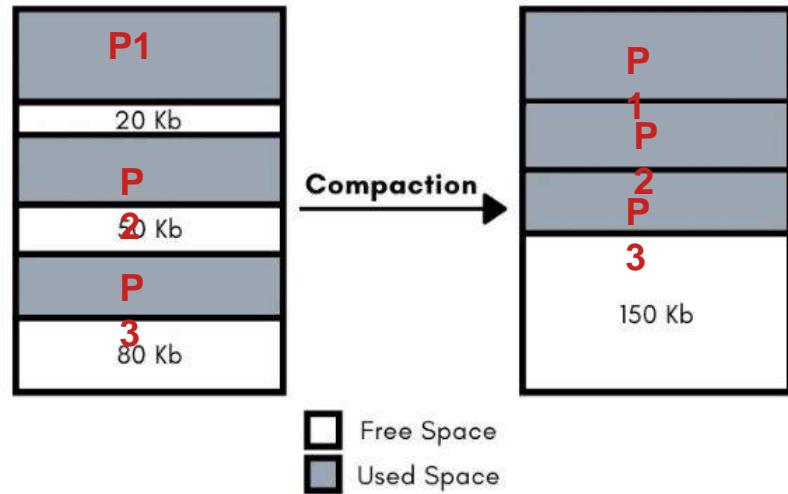
- 6k, 17k, 16k, 40k holes . Need 15k.
- **Best fit:** Find the smallest hole, larger than process. Ans: 16k
- **Worst fit:** Find the largest hole. Ans: 40k
- **First fit:** Find the “first” hole larger than the process. Ans: 17k

# **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



# 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



# 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**

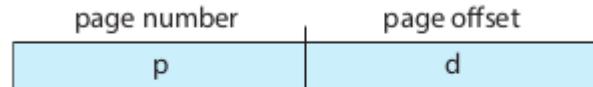
- . Reduce the size of the fixed sized partition**
- . How small then ?**
  - Smaller partitions mean more overhead for the operating system in allocating deallocating**

# 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

# Logical address seen as



# Paging hardware

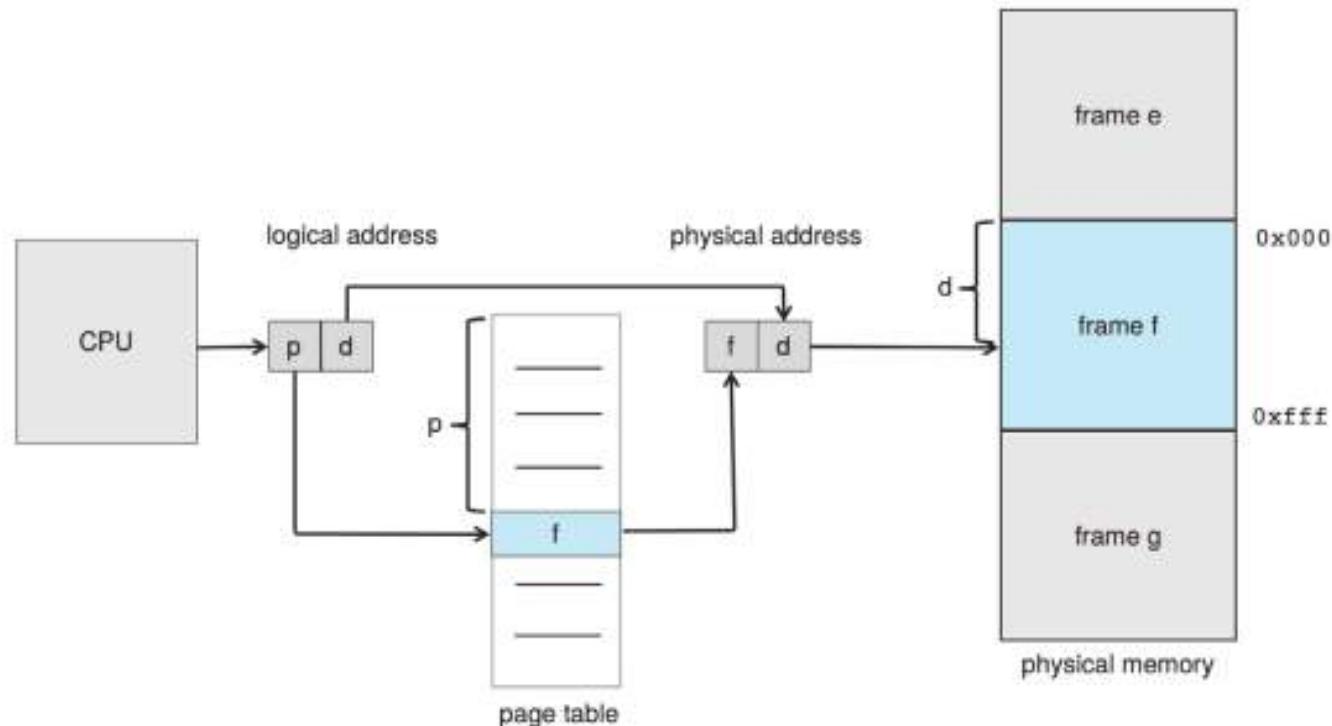


Figure 9.8 Paging hardware.

# **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**

# 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

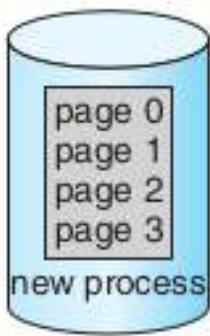
14

13

18

20

15



13

14

15

16

17

18

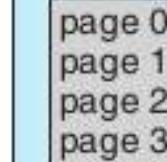
19

20

21

free-frame list

15



|   |    |
|---|----|
| 0 | 14 |
| 1 | 13 |
| 2 | 18 |
| 3 | 20 |

new-process page table

13 page 1

14 page 0

15

16

17

18 page 2

19

20 page 3

21

(a)

(b)

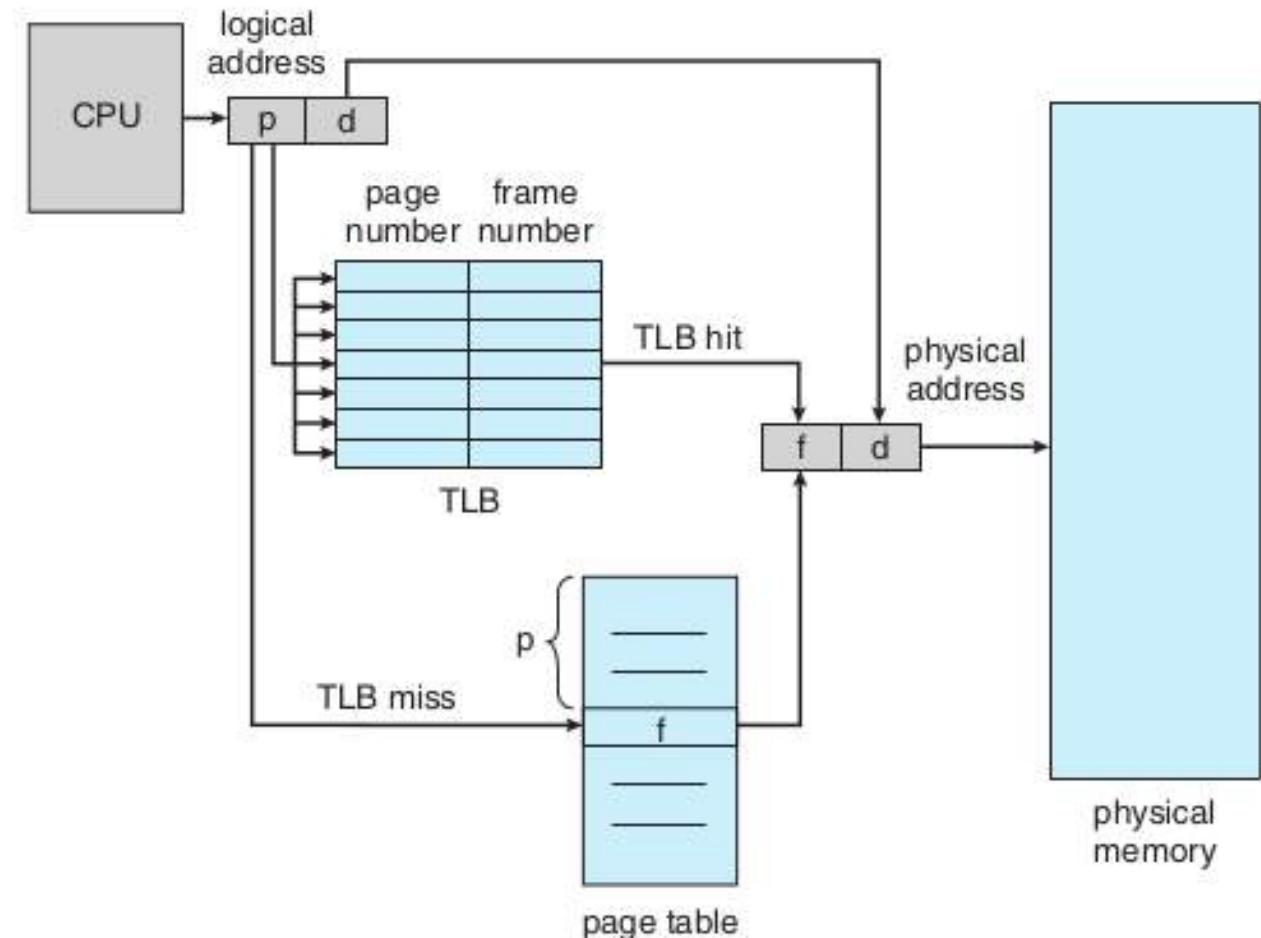
Figure 9.11 Free frames (a) before allocation and (b) after allocation.

# 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

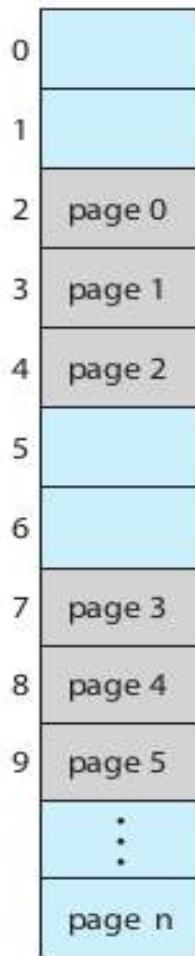
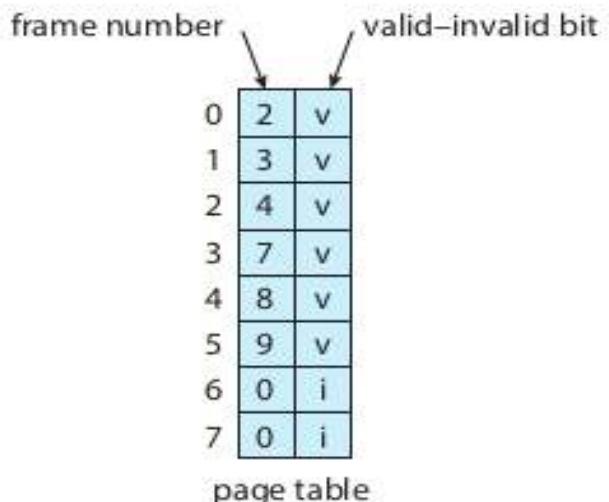
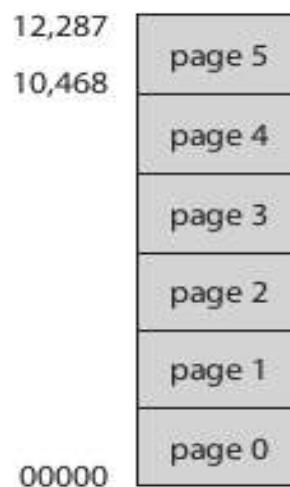


Searched

# 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

$$\text{effective access time} = 0.80 \times 10 + 0.20 \times 20$$



## Memory protection with paging

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

# X86 PDE and PTE

31

12 11 10 9 8 7 6 5 4 3 2 1 0

Page-table physical page number

|             |             |             |             |             |             |             |             |
|-------------|-------------|-------------|-------------|-------------|-------------|-------------|-------------|
| A<br>V<br>L | G<br>S<br>T | P<br>A<br>D | 0<br>A<br>D | A<br>C<br>D | C<br>W<br>T | W<br>U<br>U | U<br>W<br>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

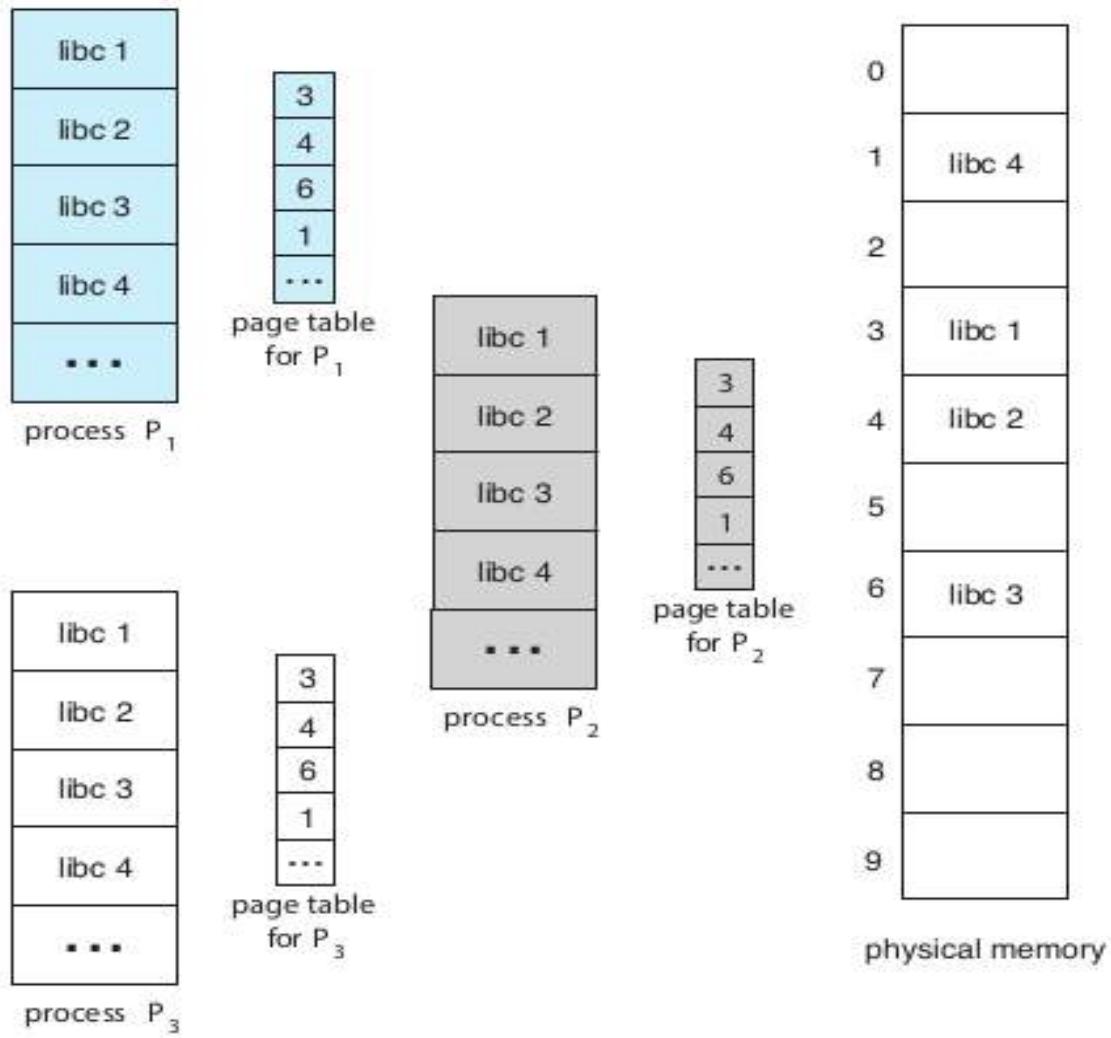
31

12 11 10 9 8 7 6 5 4 3 2 1 0

Physical page number

|             |             |             |             |             |             |             |             |
|-------------|-------------|-------------|-------------|-------------|-------------|-------------|-------------|
| A<br>V<br>L | G<br>A<br>T | P<br>D<br>A | 0<br>D<br>A | A<br>C<br>D | C<br>W<br>T | W<br>U<br>U | U<br>W<br>P |
|-------------|-------------|-------------|-------------|-------------|-------------|-------------|-------------|

PTE



# 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

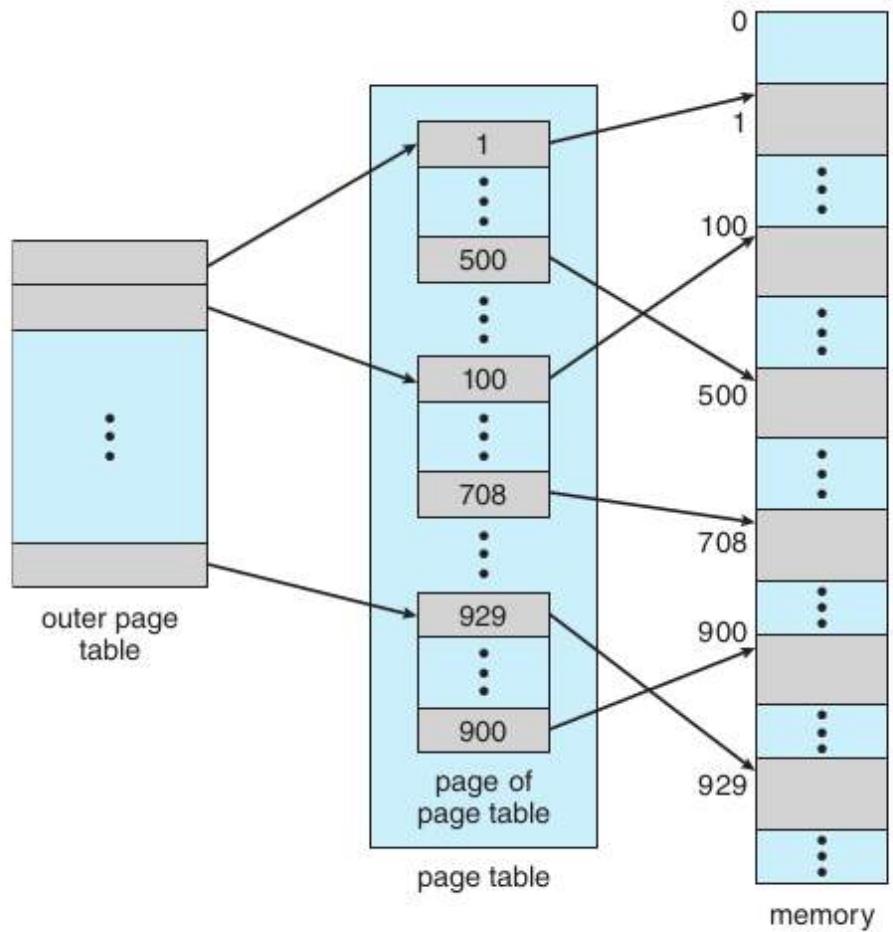
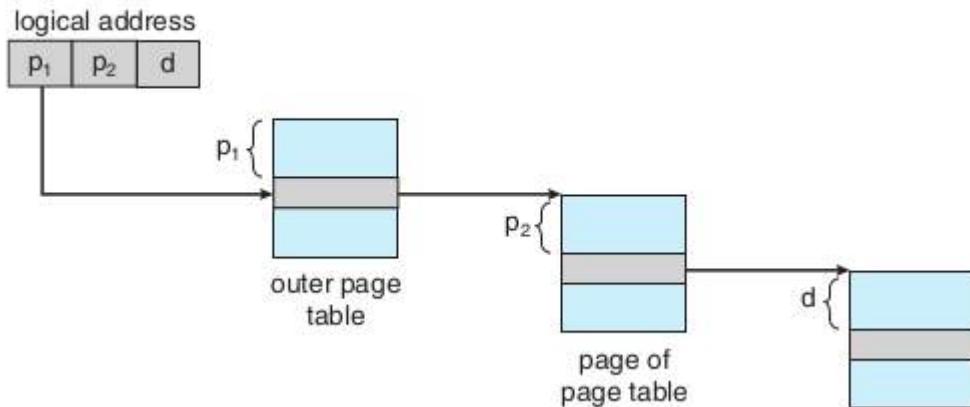
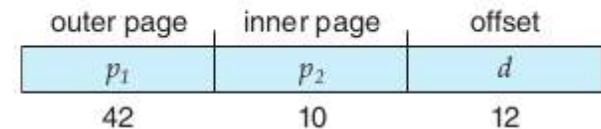


Figure 9.15 A two-level page-table scheme.



# 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

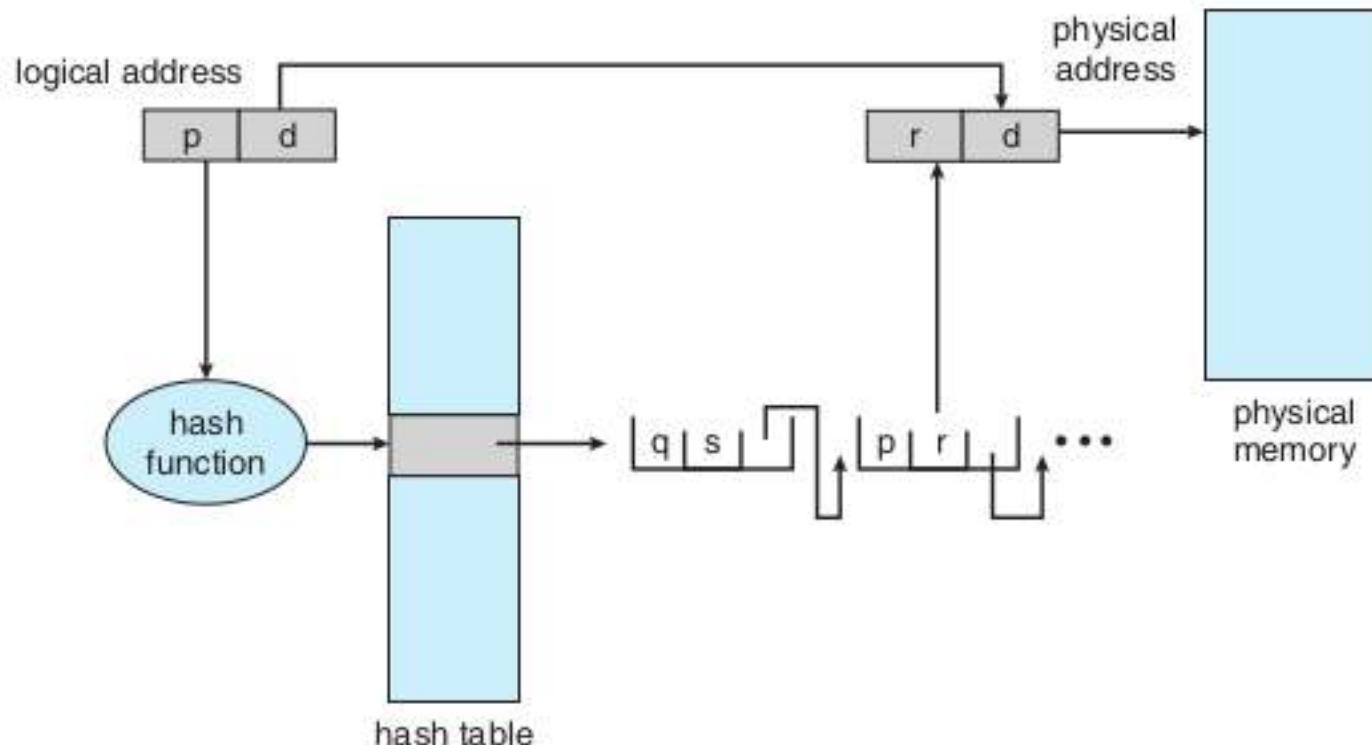


Figure 9.17 Hashed page table.

# Inverted page table

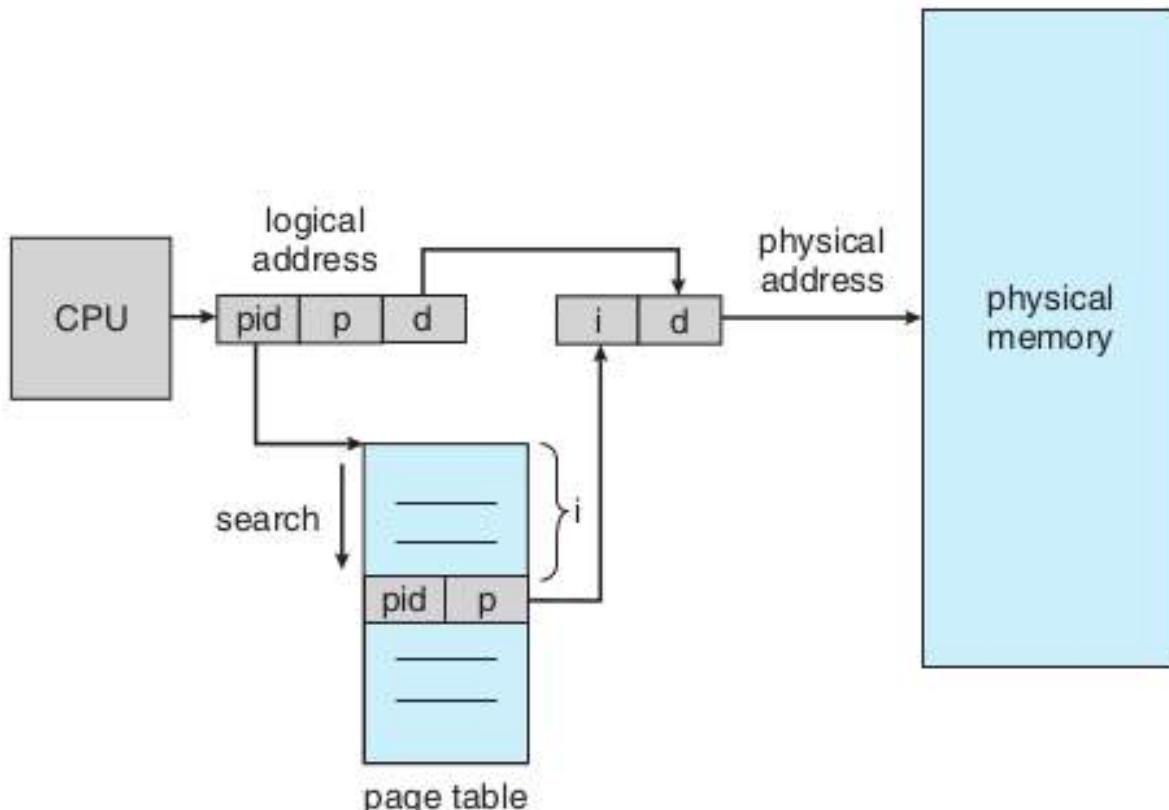


Figure 9.18 Inverted page table.

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,

# **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

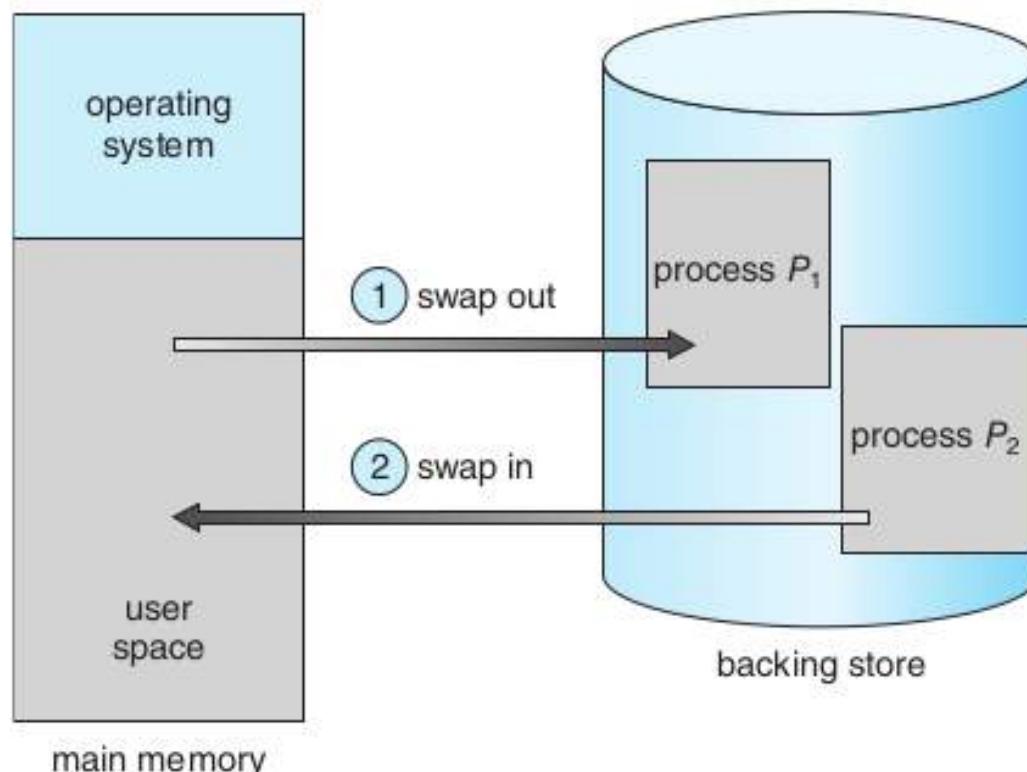
# **Case Study: Oracle SPARC Solaris**

- 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.

# 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.

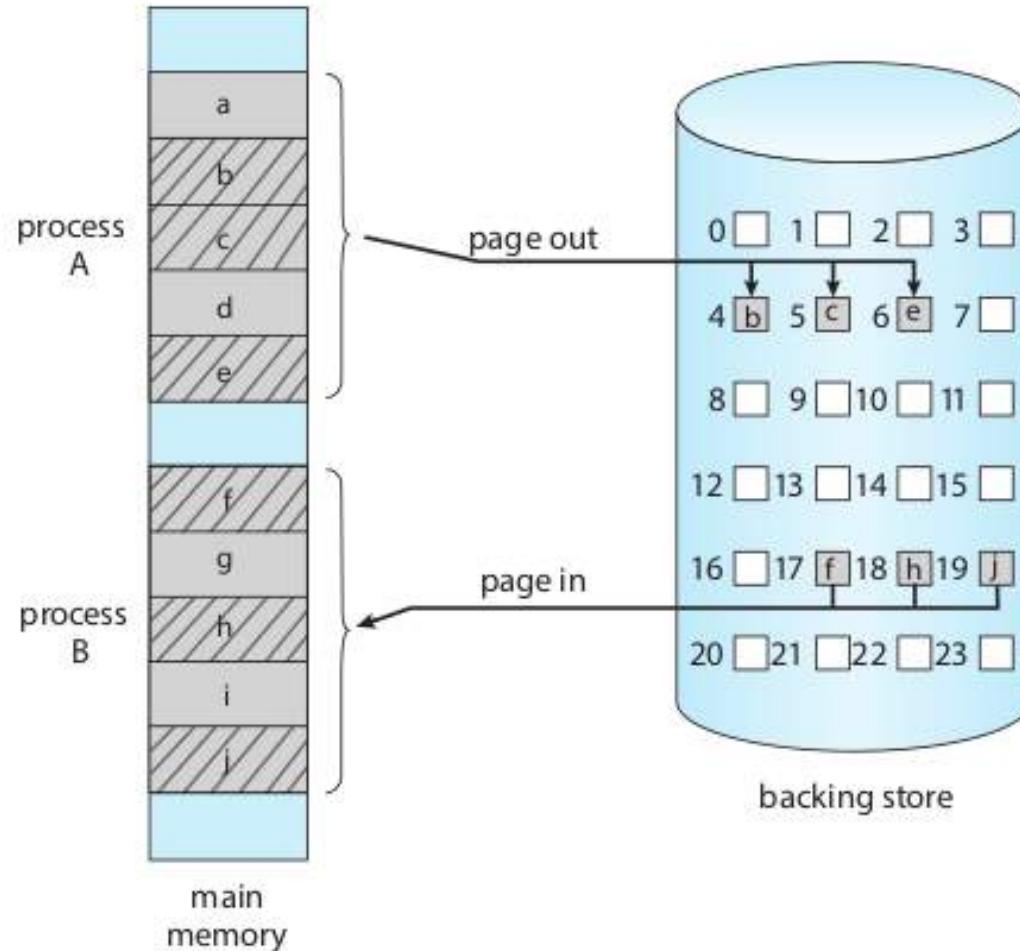
# **Swapping**

- **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**

# **Memory Management Basics**

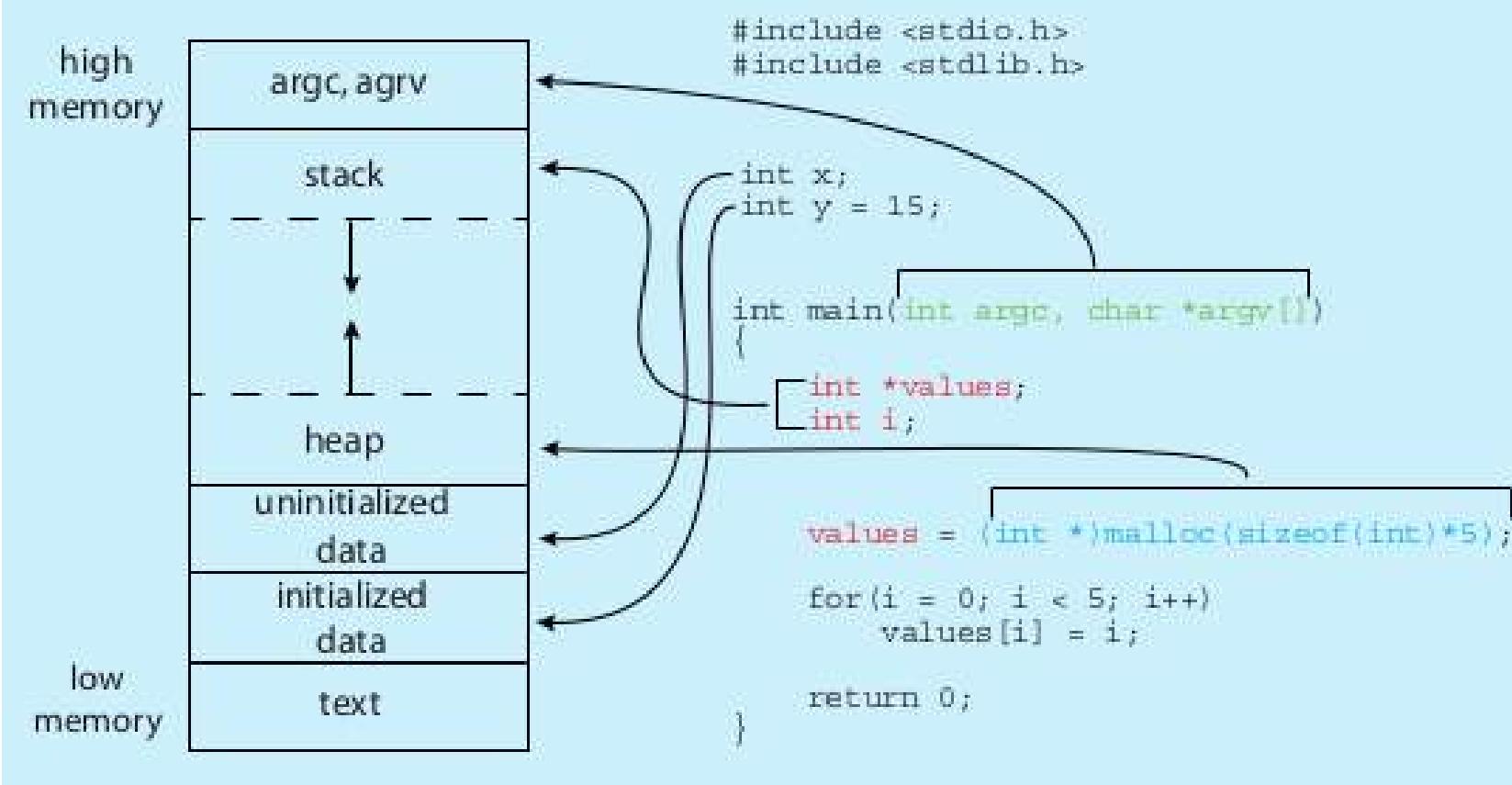
# Summary

- 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

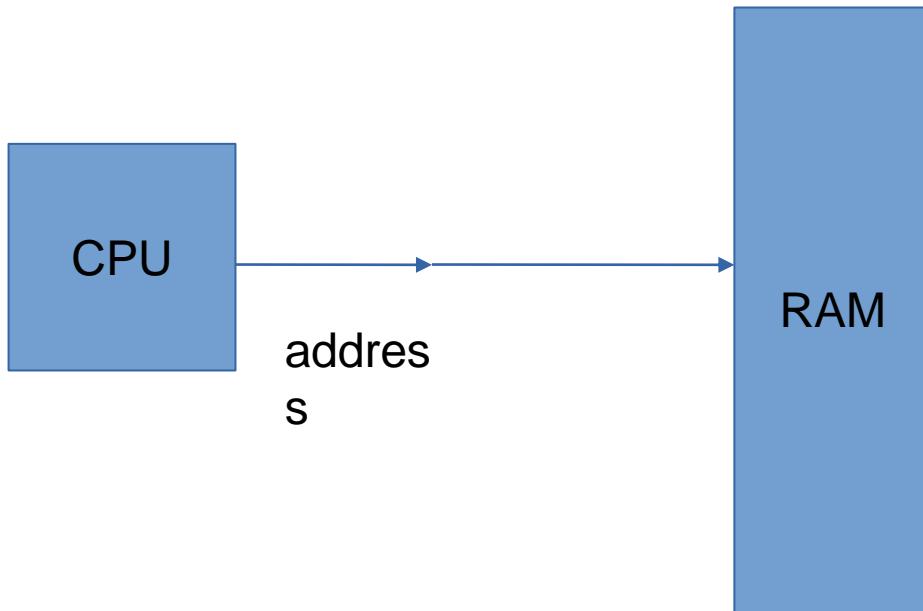
# Different ‘times’

- 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

# Different types of Address binding

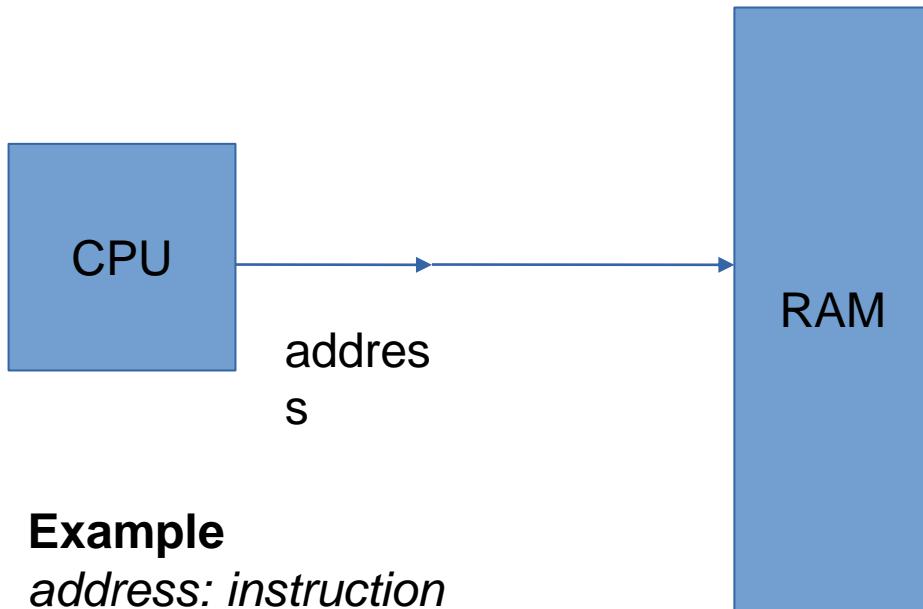
- Compile time address binding
    - Address of code/variables is fixed by compiler
    - Very rigid scheme
    - Location of process in RAM can not be changed !  
Non-relocatable code.
  - Load time address binding
    - Address of code/variables is fixed by loader
    - Location of process in RAM is decided at load time,
- 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

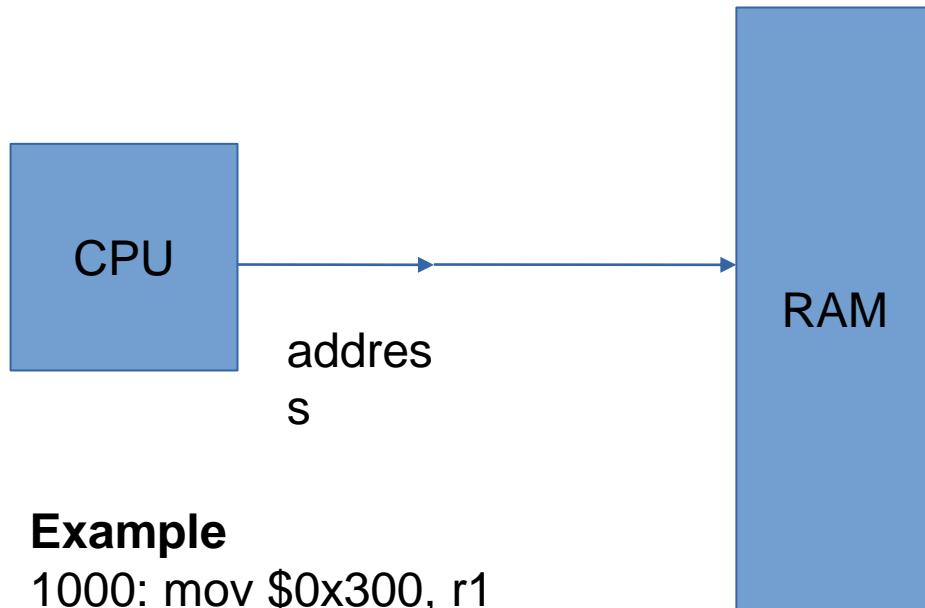
*address: instruction*

```
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**

# Simplest case

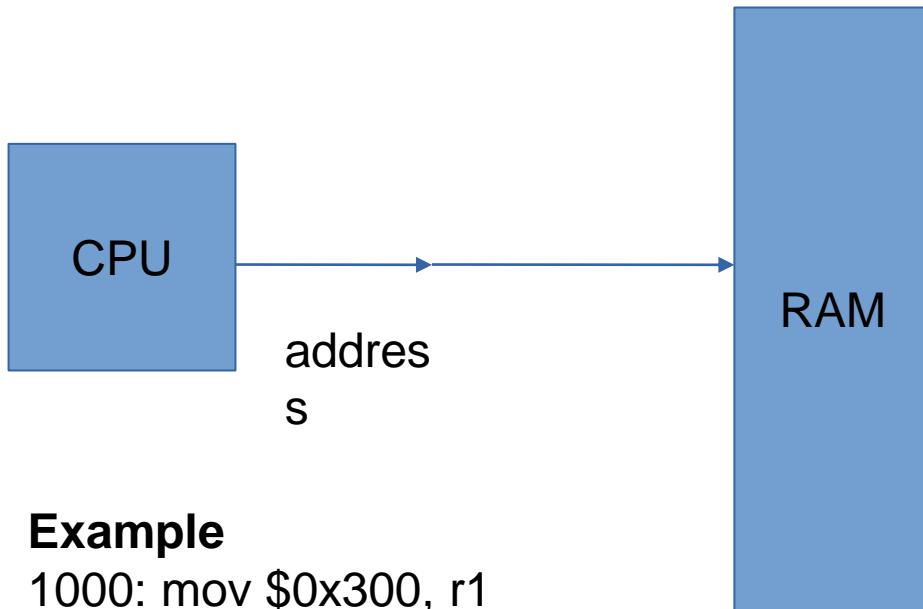


## Example

```
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**

# 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

# Base/Relocation + Limit scheme

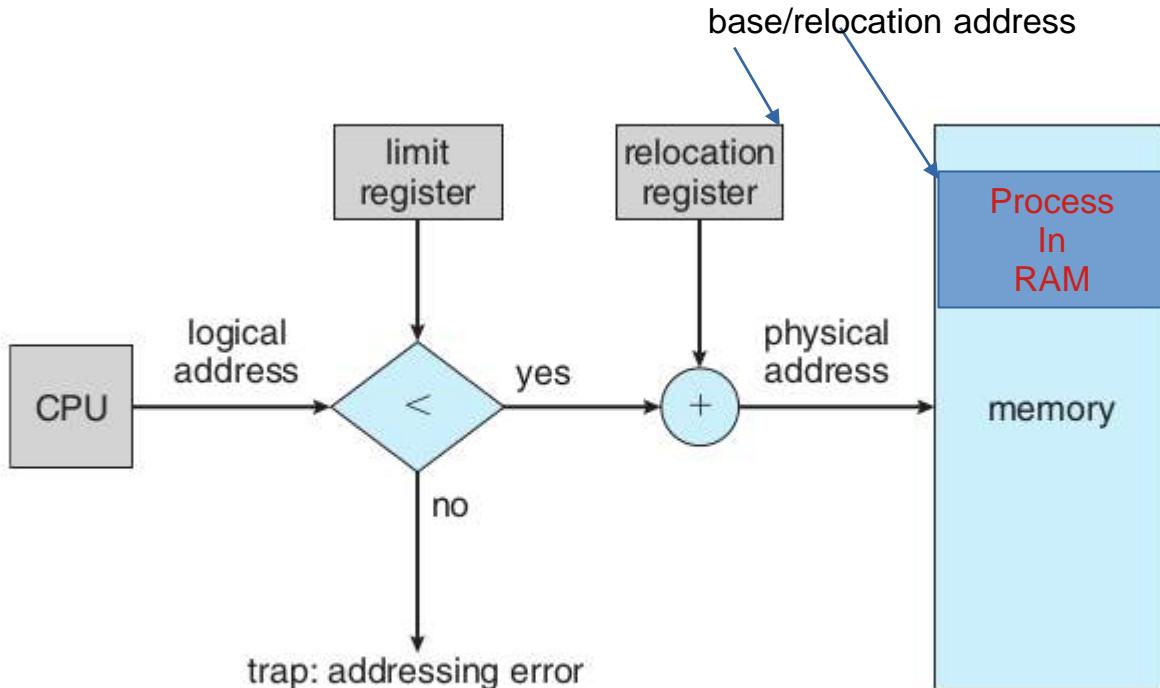
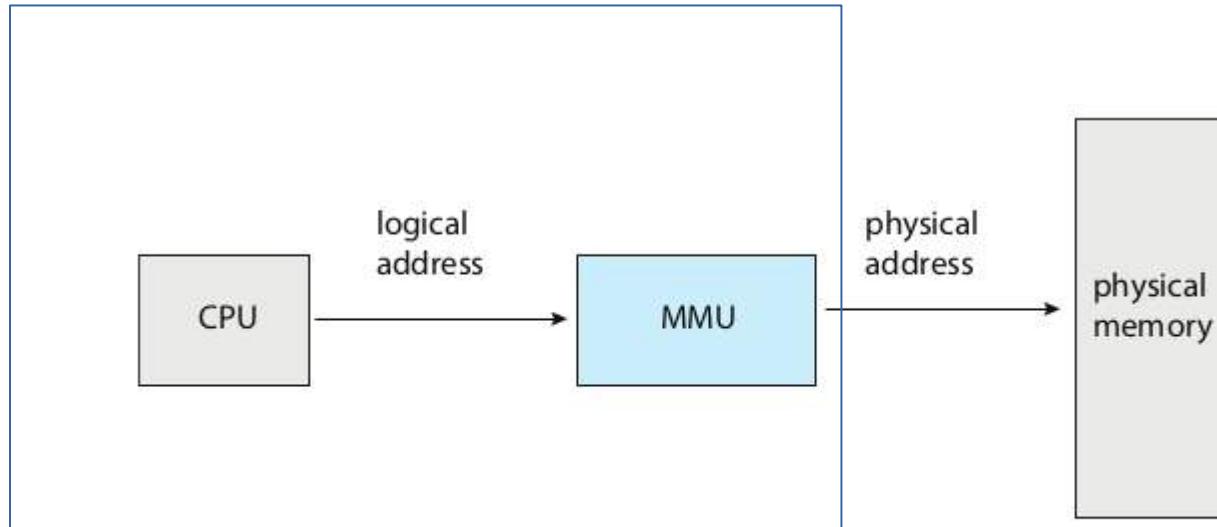


Figure 9.6 Hardware support for relocation and limit registers.

- Base and Limit are two registers inside CPU's Memory Management Unit
- 'base' is added to the address generated by

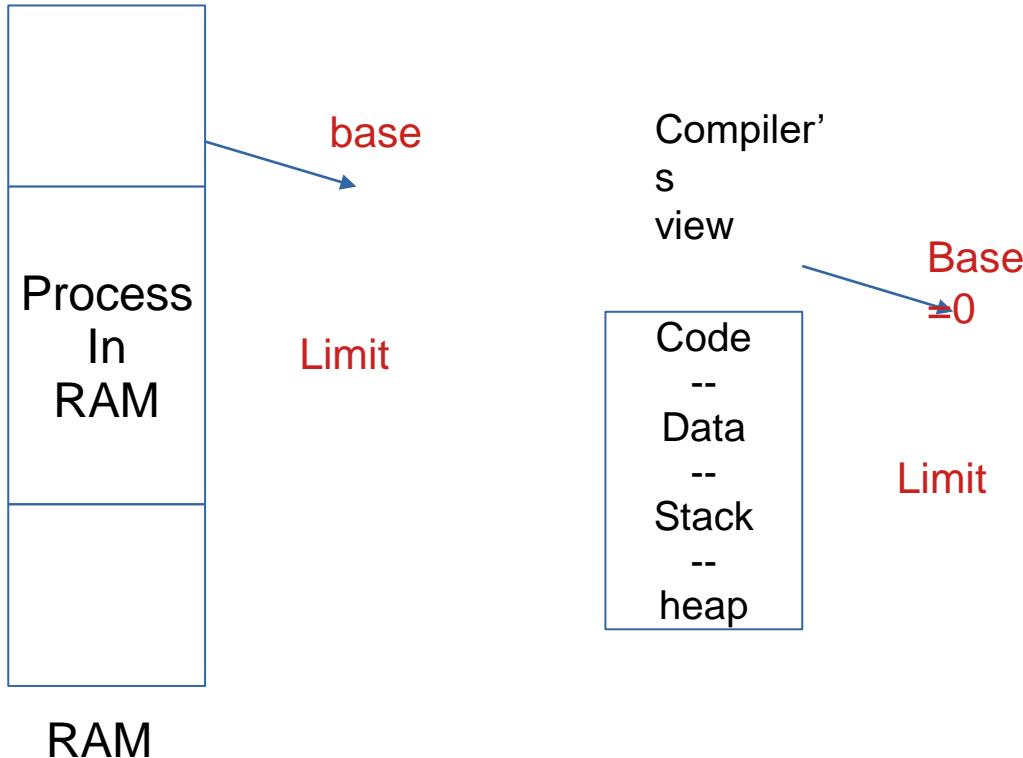
# Memory Management Unit (MMU)



**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

# 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

# Base/Relocation + Limit scheme

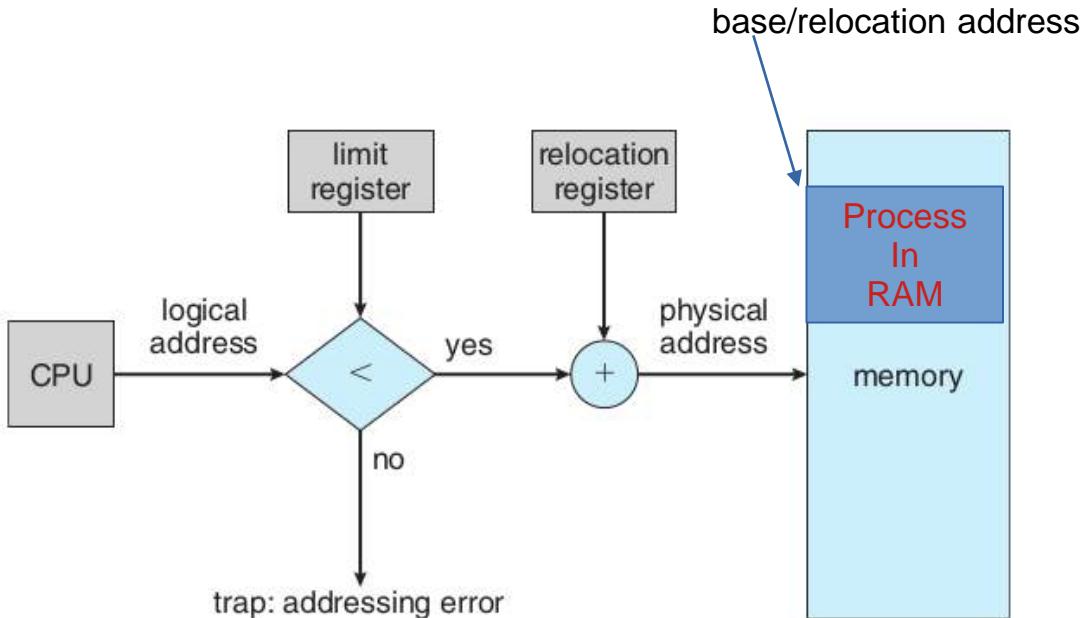


Figure 9.6 Hardware support for relocation and limit registers.

- OS's work
  - While loading the process in memory – must load as one continuous segment
  - Fill in the ‘base’ register with the actual address

# Base/Relocation + Limit scheme

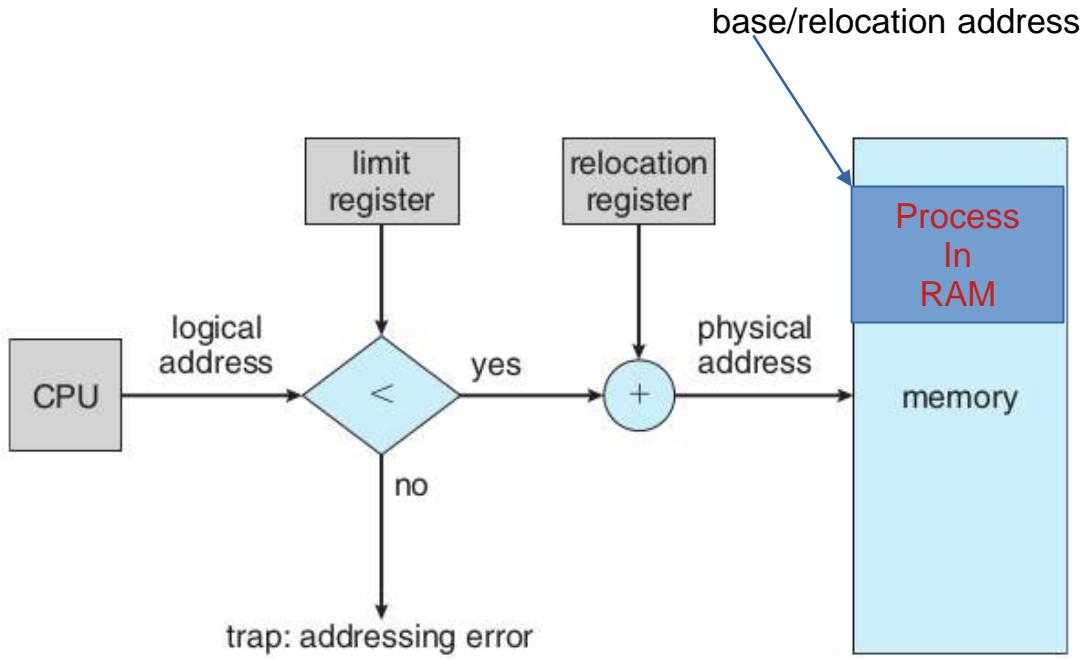
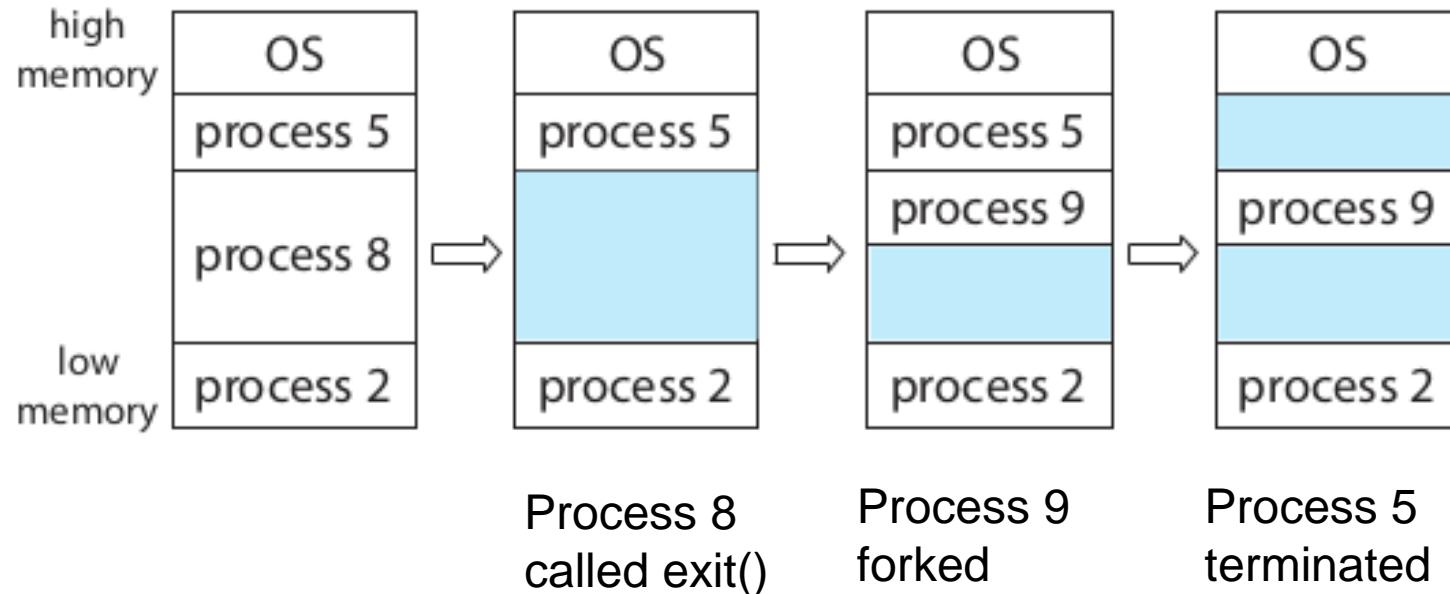


Figure 9.6 Hardware support for relocation and limit registers.

- Combined effect
  - “**Relocatable code**” – the process can go anywhere in RAM at the time of loading
  - Some memory violations can be detected

# Example scenario of memory in base+limit scheme

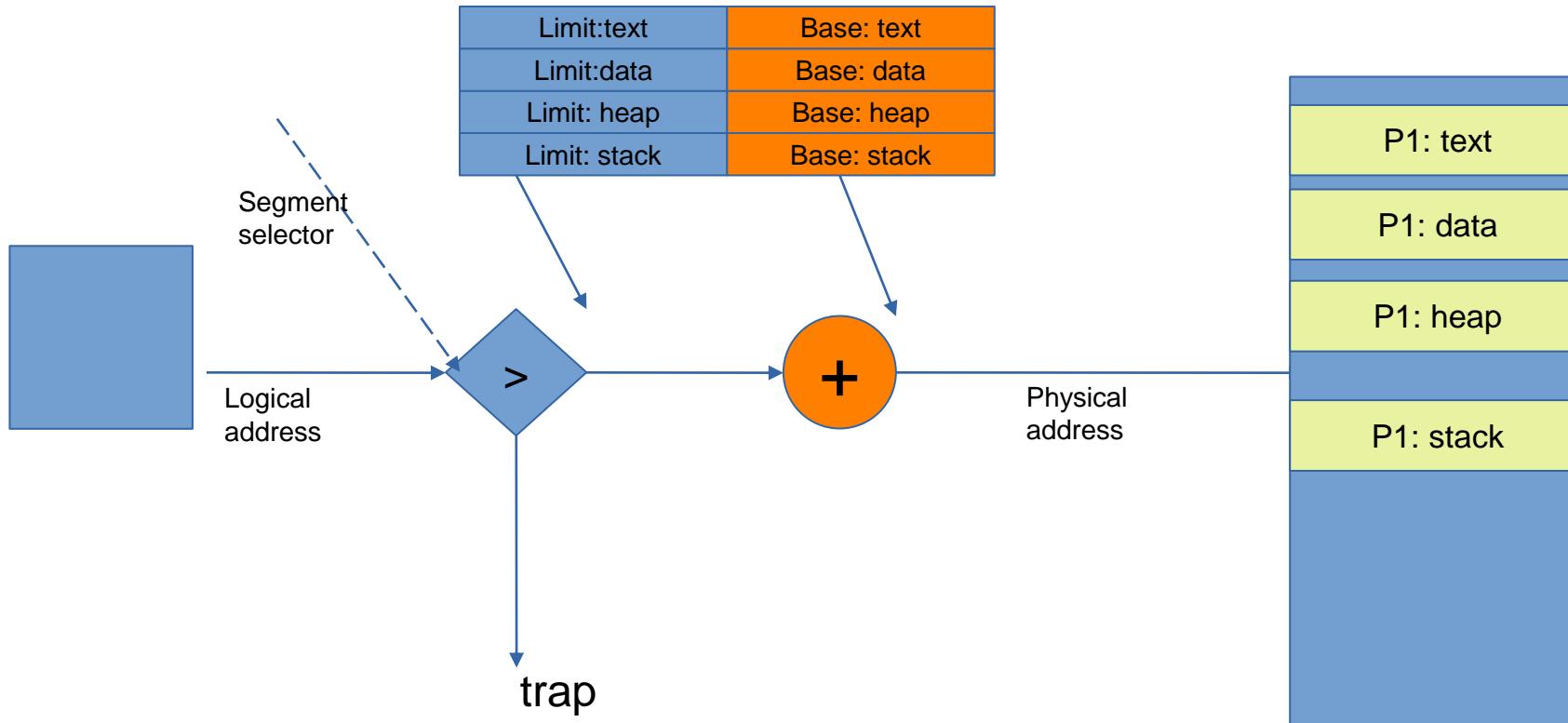


It should be possible to have relocatable code  
even with “simplest case”

By doing extra work during “loading”.

How?

# Next scheme: Multiple base +limit pairs



# **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

# **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 table

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

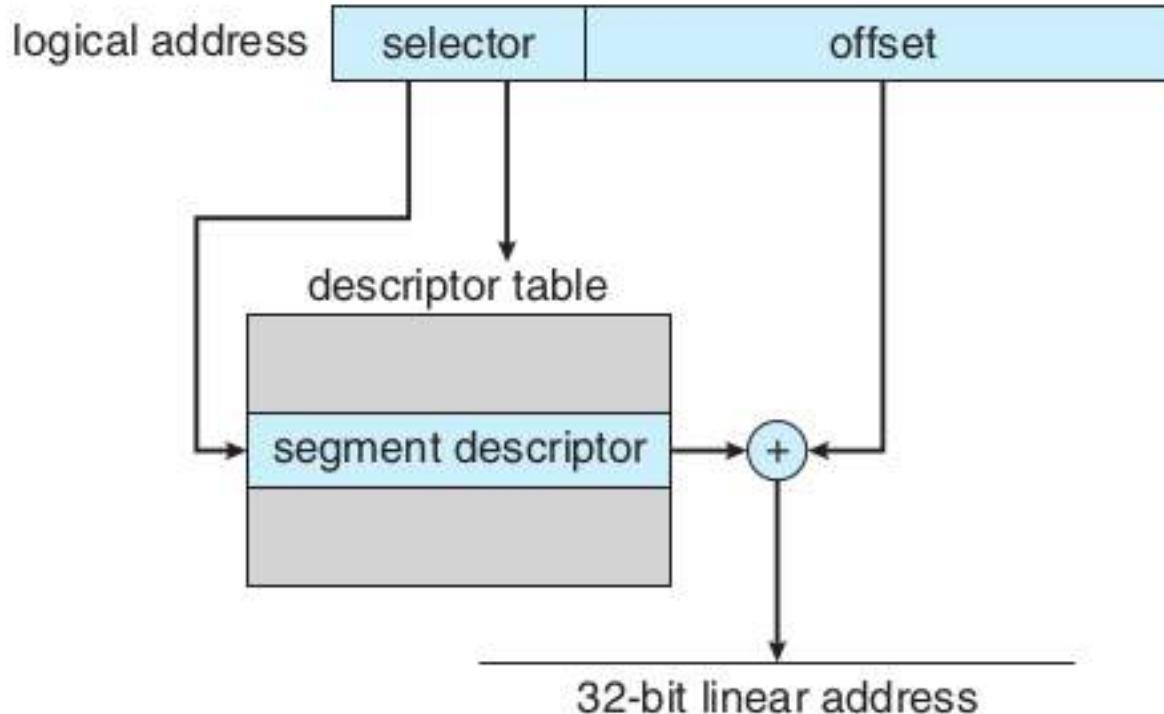


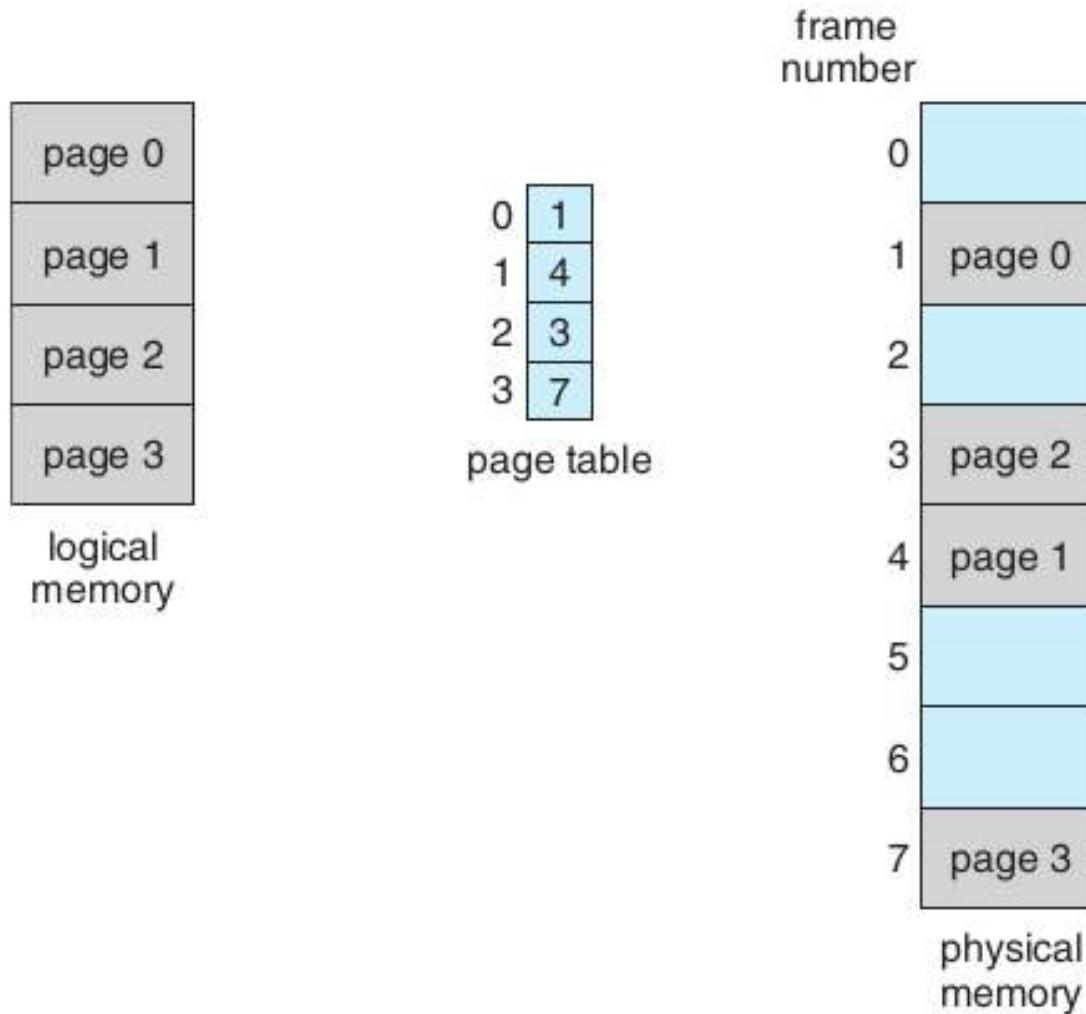
Figure 9.22 IA-32 segmentation.

# 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**

# 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

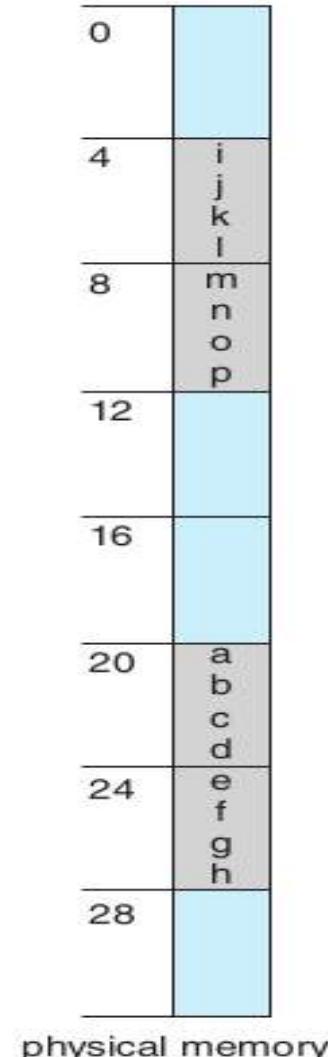
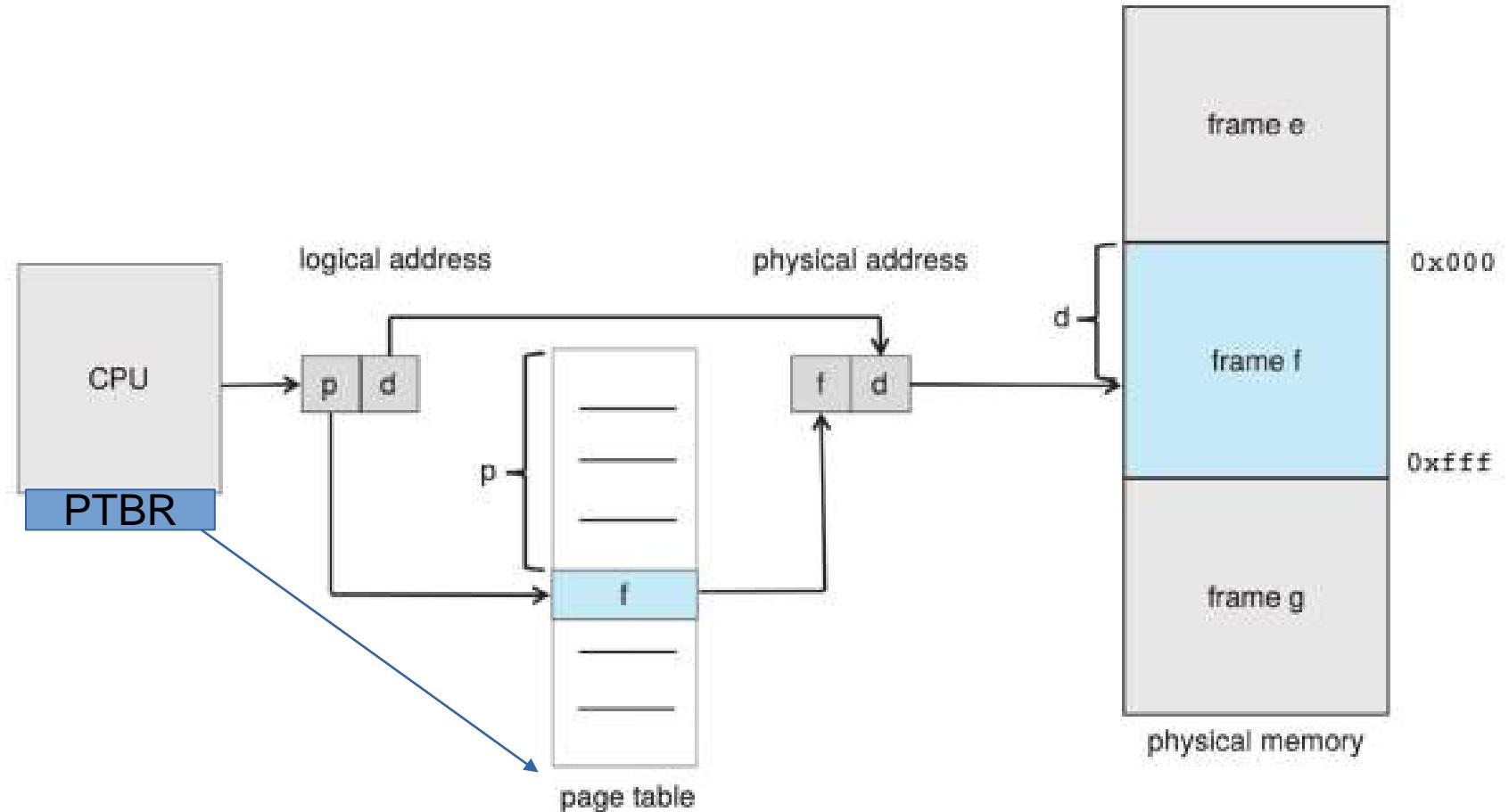


Figure 9.10 Paging example for a 32-byte memory with 4-byte pages.



**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

# 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

# X86 memory management



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

# Segmentation in x86

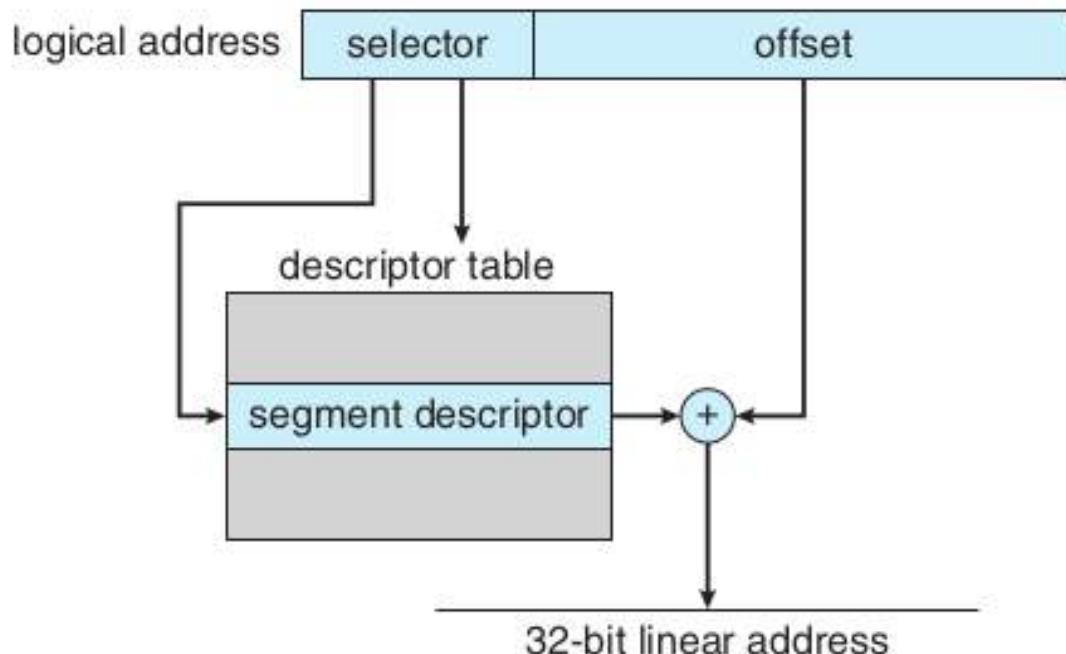


Figure 9.22 IA-32 segmentation.

- The selector is automatically chosen using Code Segment (CS) register, or Data Segment

# 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

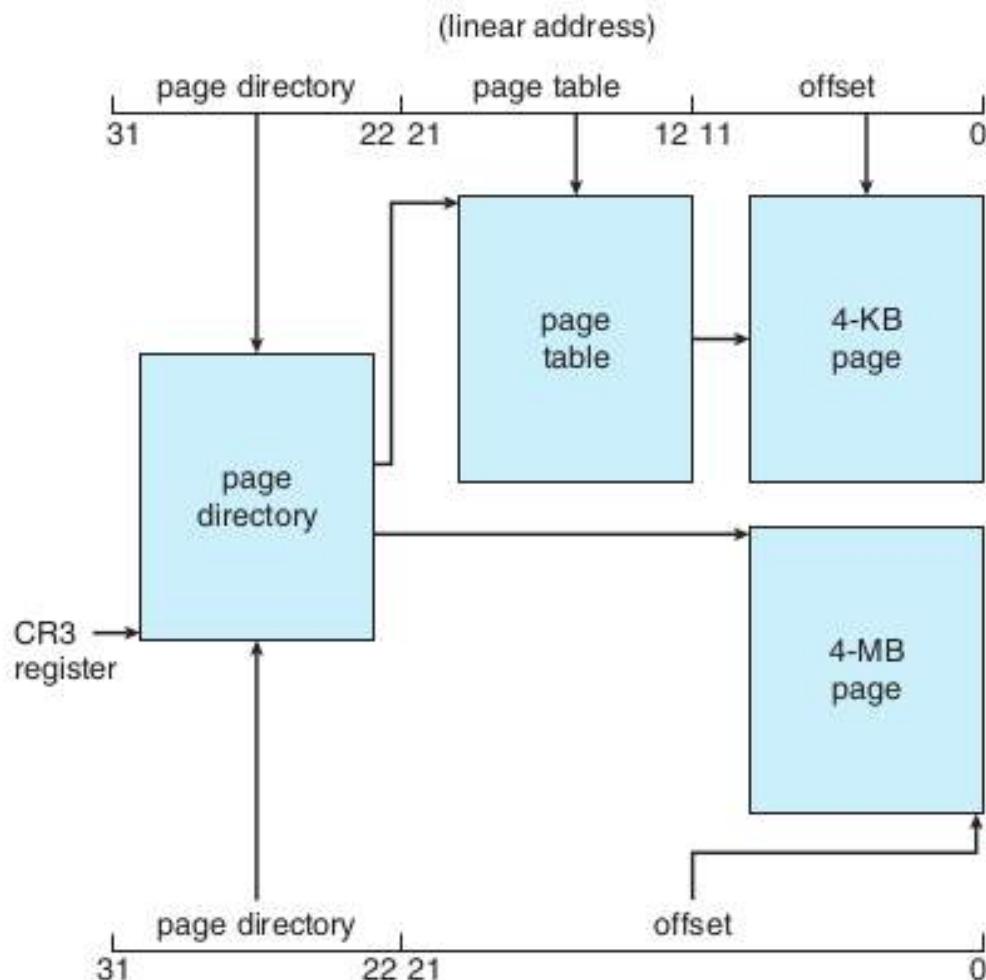


Figure 9.23 Paging in the IA-32 architecture.

# Scheduling

Abhijit A.M.

[abhijit.comp@coep.ac.in](mailto:abhijit.comp@coep.ac.in)

Credits: Slides from os-book.com

# Necessity of scheduling

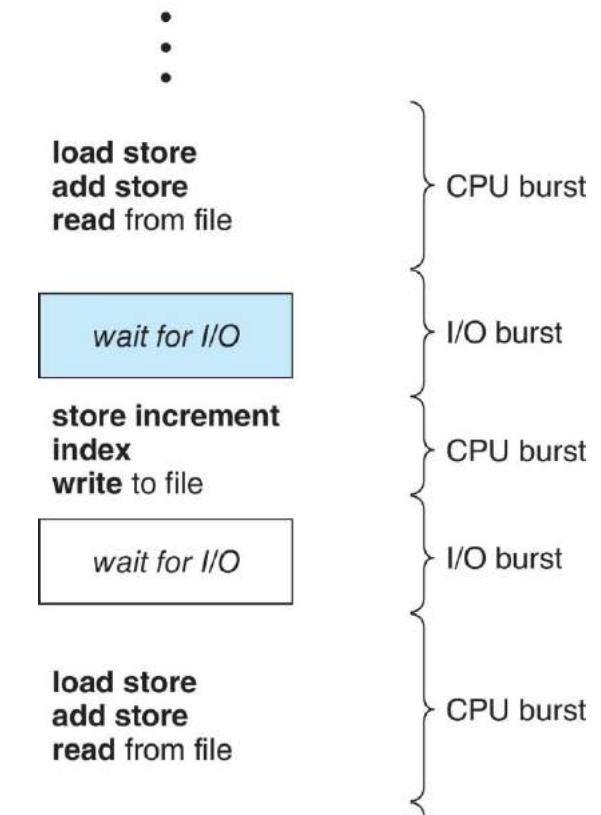
- Multiprogramming
- Increase use of CPU
  - CPU utilisation

# CPU Scheduling

- The task of selecting ‘next’ process/thread to execute on CPU and doing a context switch
- Scheduling algorithm
  - Criteria for selecting the ‘next’ process/thread and it’s implementation
- Why is it important?

# Observation: CPU, I/O Bursts

- Process can ‘wait’ for an event (disk I/O, read from keyboard, etc.)
- During this period another process can be scheduled



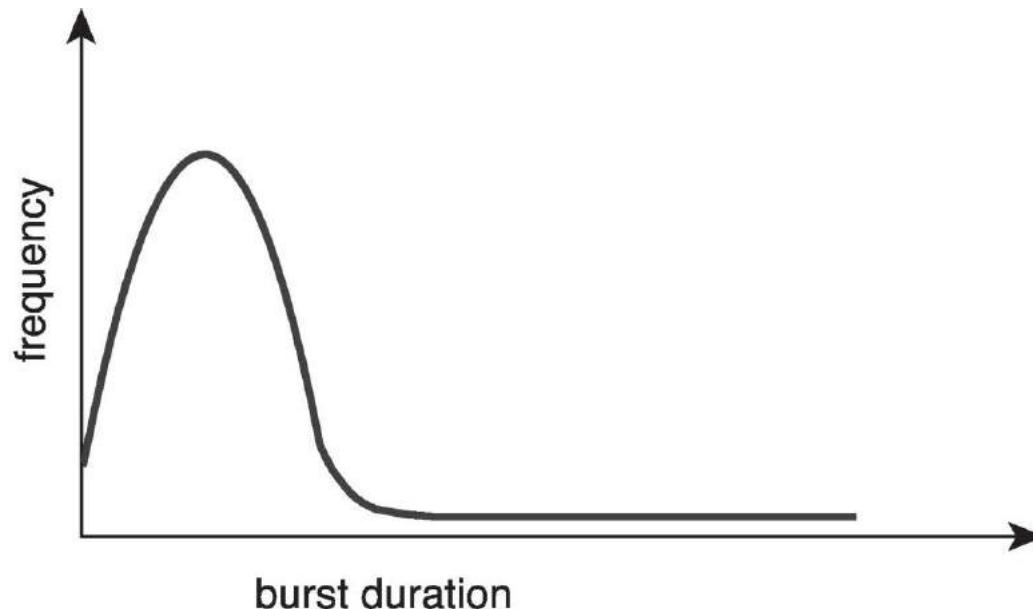
# Let's understand the problem

- Programs have alternate CPU and I/O bursts
  - Some are CPU intensive
  - Some are I/O intensive
  - Some are mix of both

A C code example:

```
f(int i, int j, int k) {
 j = k * i; // CPU burst
 scanf("%d", &i); //
 I/O burst
 k = i * j;// CPU burst
```

# CPU bursts: observation



# Scheduler, what does it do?

- **From a list of processes, ready to run**
  - Selects a process for execution
  - Allocates a CPU to the process for execution
  - Does “context switch”

# Context and Context Switch

## □ Context

- Set of registers.  
Which ones?
- All or some ?
- Following registers on xv6 kernel: edi, esi, ebx, ebp, eip
  - Related to calling

## □ Context switch

- Process context -> kernel context
  - On interrupt, system call, or exception
- Kernel context -> process context
  - Returning from system call, timer interrupt

# When is scheduler invoked?

## 1) Process Switches from running to waiting state

- Waiting for I/O, etc.

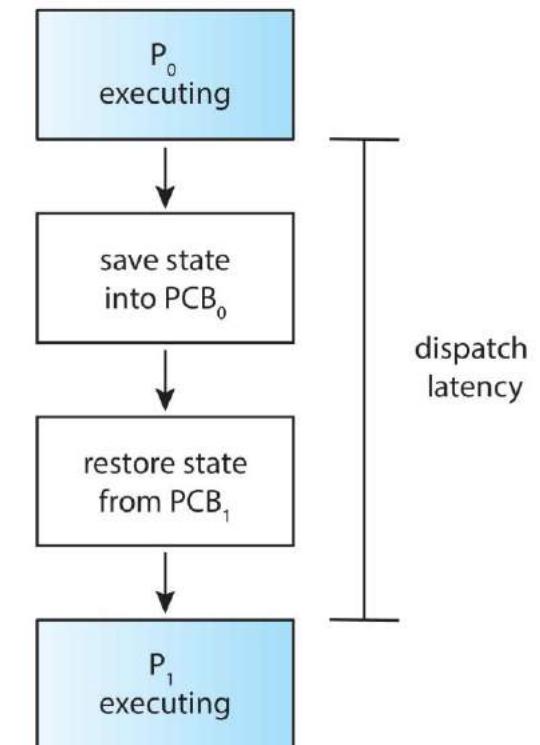
## 2) Switches from running to ready state

- Scheduling under 1 and 4 is **nonpreemptive**
- All other scheduling is **preemptive**
  - Consider access to shared data

# Dispatcher: A part of scheduler

- Gives control of the CPU to the process selected by the short-term scheduler; this involves:

- switching context
- switching to user mode
- jumping to the proper location in the user program to restart



# Dispatcher in action on Linux

- Run “`vmstat 1 3`”
  - Means run vmstat 3 times at 1 second delay
- In output, look at **CPU:cs**
  - Context switches every second
- Also for a process with pid 3323
  - Run “`cat /proc/3323/status`”
    - See
      - voluntary\_ctxt\_switches
      - --> Process left CPU

# Scheduling criteria

- **CPU utilization: Maximise**
  - keep the CPU as busy as possible. Linux: idle task is scheduled when no process to be scheduled.
- **Throughput : Maximise**
  - # of processes that complete their execution per time unit
- **Turnaround time : Minimise**

# To be studied later

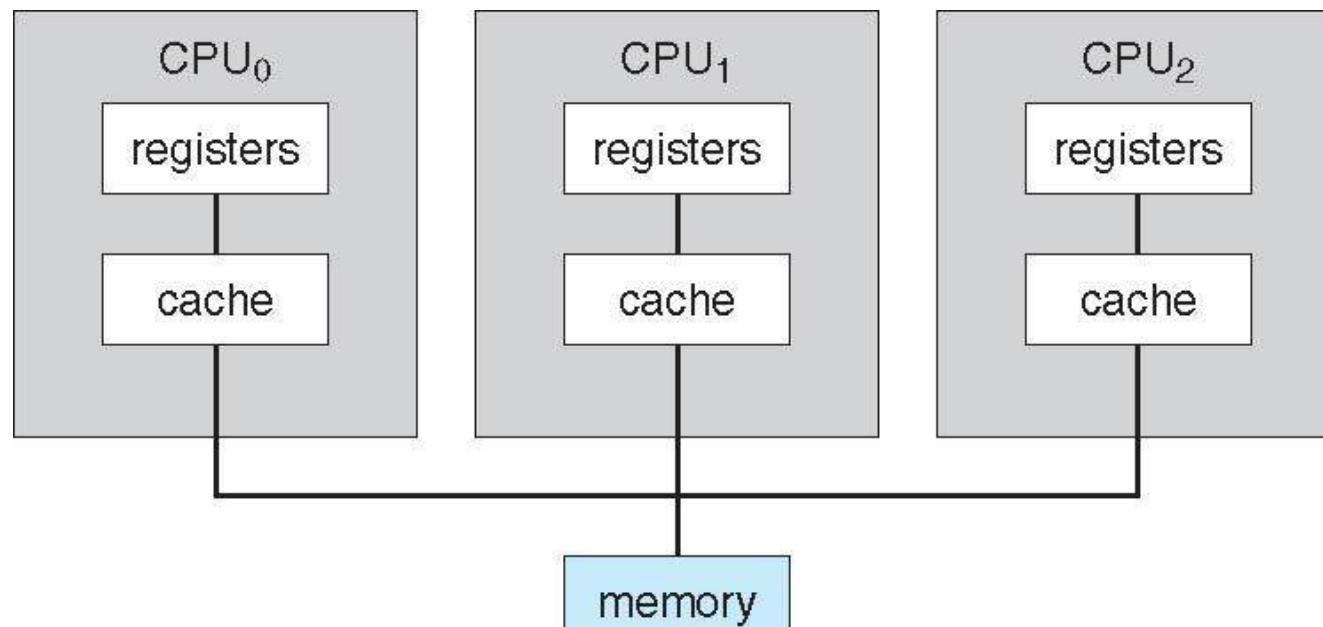
- Evaluation of scheduling criteria
- Different scheduling algorithms, and their characteristics
  - Round Robin, FIFO, Shortest Job First, Priority, Multi-level Queue, etc.
- Thread scheduling

# **Multi Processor Scheduling**

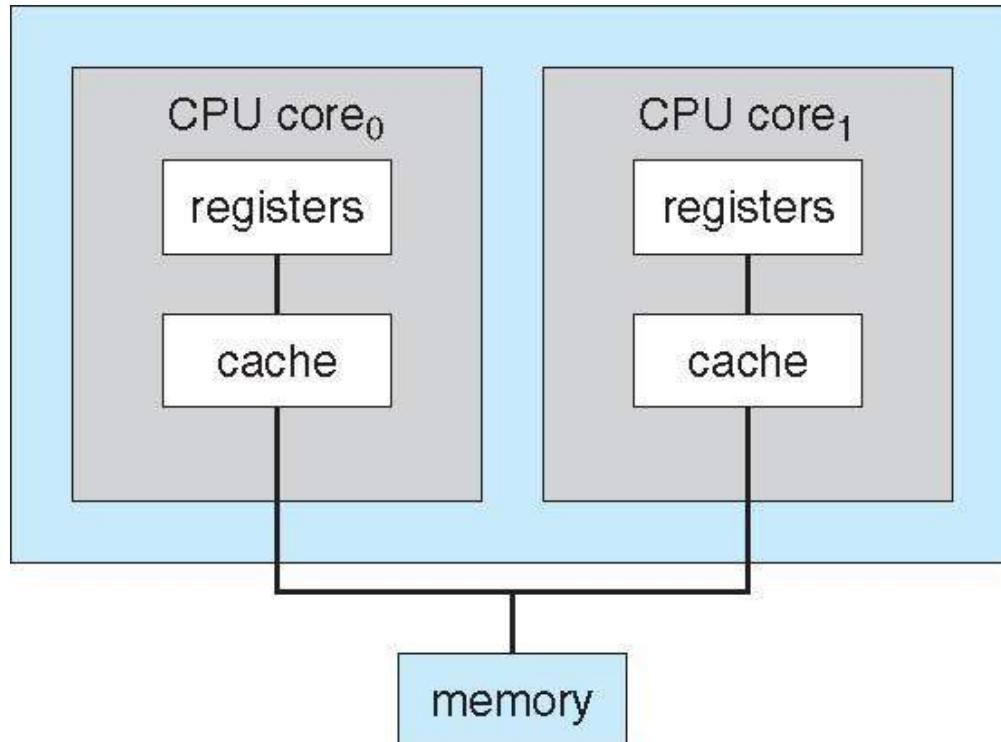
# Multiprocessor systems

- **Each processor has separate set of registers**
  - All: eip, esp, cr3, eax, ebx, etc.
- **Each processor runs independently of others**
- **Main difference is in how do they access memory**

# Symmetric multiprocessing (SMP)



# Multicore systems (also SMP)

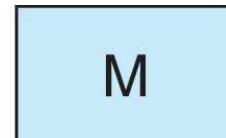


# Multicore Processors

- Recent trend to place multiple processor cores on same physical chip
- Faster and consumes less power
- Multiple threads per core also growing
  - Takes advantage of memory stall to make progress on another thread while memory retrieve happens



compute cycle



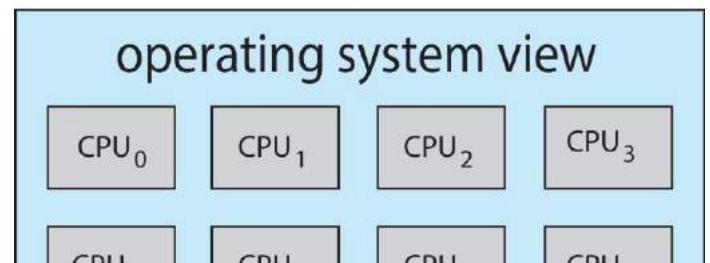
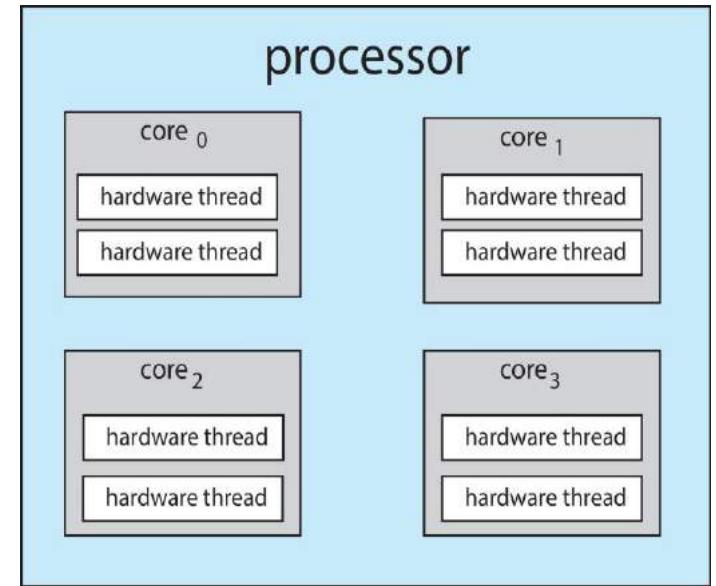
memory stall cycle

# Multithreaded Multicore System

Each core has > 1 hardware threads.

**Chip-multithreading** (CMT) assigns each core multiple hardware threads.  
(Intel refers to this as **hyperthreading**.)

On a quad-core system with 2 hardware threads per core, the operating system sees 8 logical processors.



If one thread has a memory stall, switch

# More on SMP systems

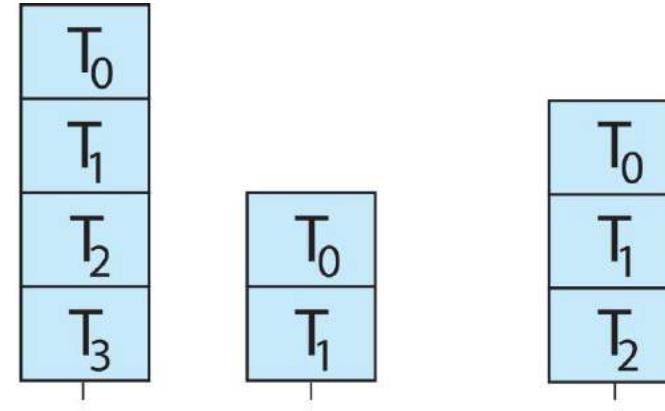
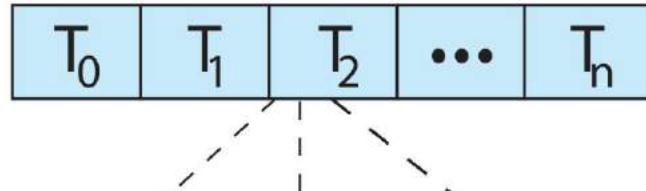
- During booting – each CPU needs to be turned on
  - Special I/O instructions writing to particular ports
  - See lpicstartap() in xv6
  - Need to setup CR3 on each processor
  - Segmentation, Page tables are shared (same memory for all CPUs)

# Multiple-Processor Scheduling

- CPU scheduling more complex when multiple CPUs are available
- Multiprocess may be any one of the following architectures:
  - Multicore CPUs
  - Multithreaded cores
  - NUMA systems

# Multiple-Processor Scheduling

- Symmetric multiprocessing (SMP) is where each processor is self scheduling.
- All threads may be in a common ready queue (a)
- Each processor may have its own private queue of threads (b)



# SMP in xv6

- ❑ Only one process queue
- ❑ No load balancing, no affinity (more later)
- ❑ A process may run any CPU-burst /allotted-time-quantum on any processor randomly

**End**

# Compilation, Linking, Loading

Abhijit A M

# Review of last few lectures

- Boot sequence: BIOS, boot-loader, kernel
- Boot sequence: Process world
  - kernel->init -> many forks+execs() -> ....
- Hardware interrupts, system calls, exceptions
- Event driven kernel
- System calls
  - Fork, exec, ... open, read, ...

# What are compiler, assembler, linker and loader, and C library

System Programs/Utilities

Most essential to make a kernel really usable

# Standard C Library

- A collection of some of the most frequently needed functions for C programs
  - `scanf`, `printf`, `getchar`, system-call wrappers (`open`, `read`, `fork`, `exec`, etc.), ...
- An machine/object code file containing the machine code of all these functions
  - Not a source code! Neither a header file. More later.
- Where is the C library on your computer?

# Compiler

- application program, which converts one (programming) language to another

- Most typically compilers convert a **high level language** like C, C++, etc. to **Machine code language**



- E.g. GCC /usr/bin/gcc

- Usage: e.g.
  - \$ gcc main.c -o main
  - Here main.c is the C code and "main" is the

# Assembler

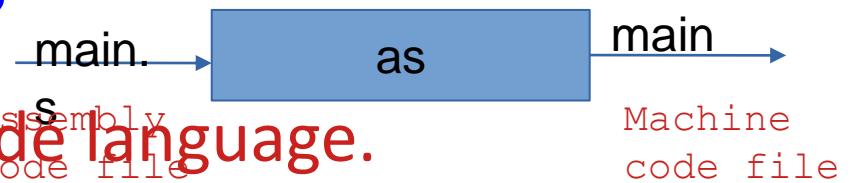
- application program, converts assembly code into machine code

- What is assembly language?

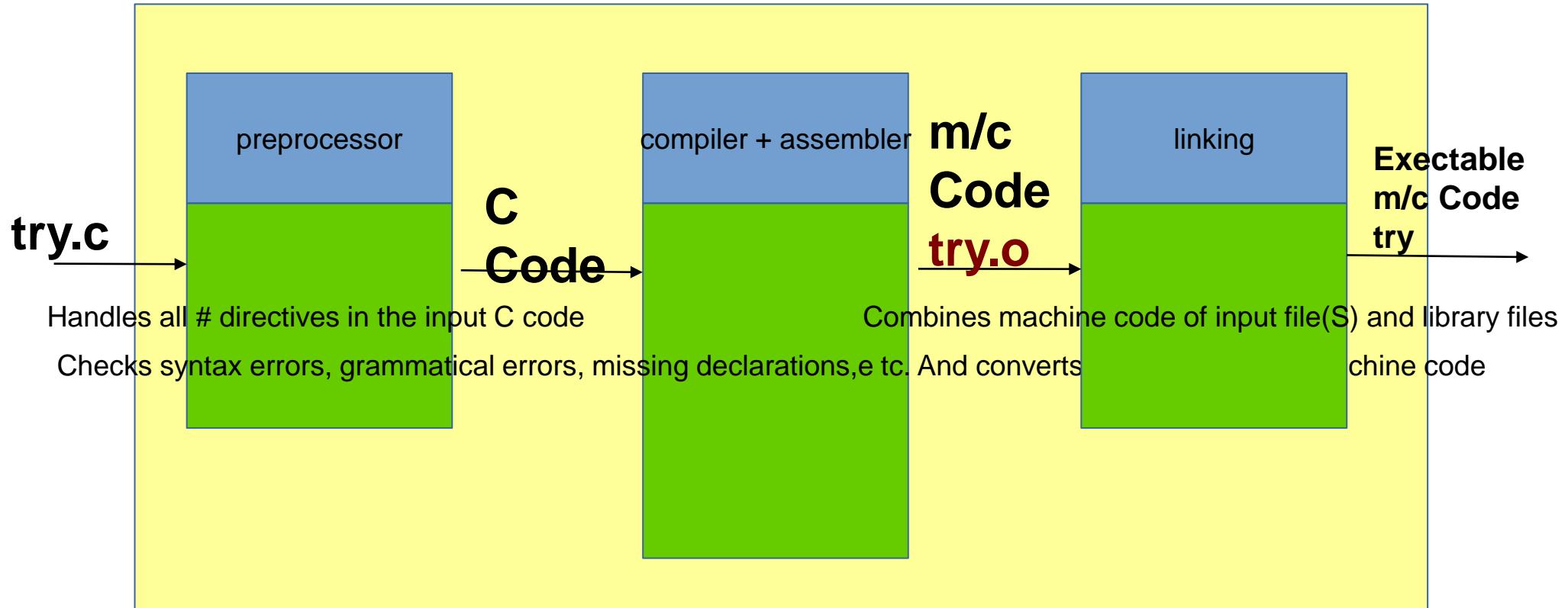
- Human readable machine code language.

- E.g. x86 assembly code

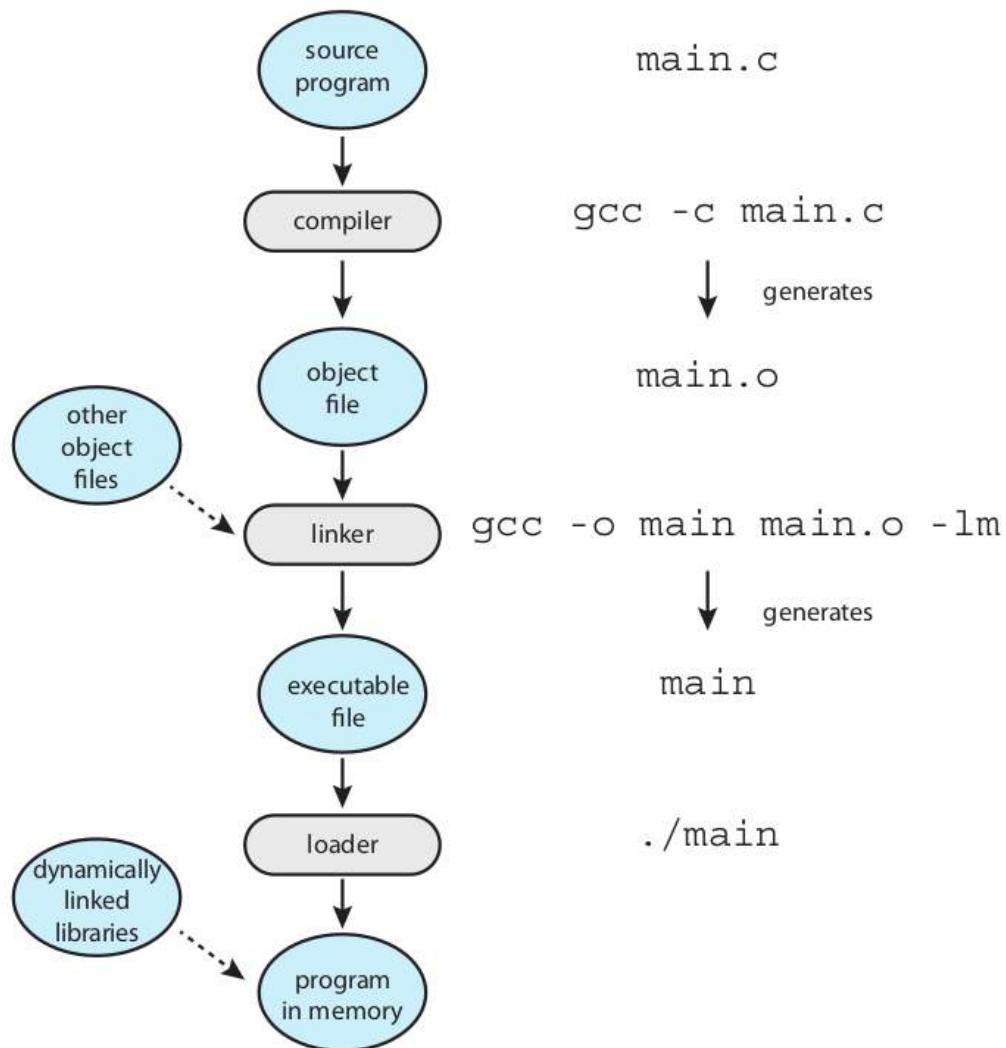
- mov 50, r1
  - add 10, r1



# Compilation Process



gcc



- From the textbook

**Figure 2.11** The role of the linker and loader.

# Example

**try.c**

```
#include <stdio.h>
#define MAX 30
int f(int, int);
int main() {
 int i, j, k;
 scanf("%d%d", &i, &j);
 k = f(i, j) + MAX;
 printf("%d\n", k);
 return 0;
}
```

**f.c**

```
int g(int);
#define ADD(a, b) (a + b)
int f(int m, int n) {
 return ADD(m,n) + g(10);
}
```

**g.c**

```
int g(int x) {
 return x + 10;
}
```

Try these commands, observe the output/errors/warnings, and try to understand what is happening

```
$ gcc try.c
$ gcc -c try.c
$ gcc -c f.c
$ gcc -c g.c
$ gcc try.o f.o g.o -o try
$ gcc -E try.c
$ gcc -E f.c
```

# More about the steps

- Pre-processor
  - `#define ABC XYZ`
    - cut ABC and paste XYZ
  - `# include <stdio.h>`
    - copy-paste the file stdio.h
    - There is no CODE in stdio.h, only typedefs, #includes, #define, #ifdef, etc.
- Linking
  - Normally links with the standard C-library by default
  - To link with other libraries, use the -l option of gcc
    - `cc main.c -lm -lncurses -o main` # links with libm.so and

# Using gcc itself to understand the process

- Run only the preprocessor
  - `cc -E test.c`
  - Shows the output on the screen
- Run only till compilation (no linking)
  - `cc -c test.c`
  - Generates the “test.o” file , runs compilation + assembler
  - `gcc -S main.c`
  - One step before machine code generation, stops at assembly code
- Combine multiple .o files (only linking part)

`cc test.o main.o try.o -o something`

# Linking process

- Linker is an application program
  - On linux, it's the "ld" program
  - E.g. you can run commands like \$ ld a.o b.o -o c.o
  - Normally you have to specify some options to ld to get a proper executable file.
- When you run gcc
  - \$ cc main.o f.o g.o -o try
  - the CC will internally invoke "ld" . ld does the job of linking

# Linking process

- The resultatnt file "try" here, will contain the codes of all the functions and linkages also.
- **What is linking?**
  - "connecting" the call of a function with the code of the function.
- What happens with the code of printf()?
  - The linker or CC will automatically pick up code from the libc.so.6 file for the functions.

# Executable file format

- An executable file needs to execute in an environment created by OS and on a particular processor
  - Contains machine code + other information for OS
  - Need for a structured-way of storing machine code
- ELF : The format on Linux.
- Try this
  - \$ file /bin/ls
  - \$ file /usr/lib/x86\_64-linux-gnu/libc-2.31.so

# Exec() and ELF

- When you run a program
  - \$ ./try
  - Essentially there will be a fork() and exec("./try", ...)
  - So the kernel has to read the file "./try" and understand it.
  - So each kernel will demand its own object code file format.
- ELF is used not only for executable (complete machine code) programs, but also for partially compiled files e.g. main.o and library files like libc.so.6
- What is a.out?
  - "a.out" was the name of a format used on earlier Unixes.
  - It so happened that the

# Utilities to play with object code files

- objdump
  - \$ objdump -D -x /bin/ls
  - Shows all disassembled machine instructions and “headers”
- hexdump
  - \$ hexdump /bin/ls
  - Just shows the file in hexadecimal
- readelf
  - ar
    - To create a “statically linked” library file
    - \$ ar -crs libmine.a one.o two.o
  - Gcc to create shared library
    - \$ gcc hello.o -shared -o libhello.so
  - To see how gcc invokes as, ld, etc; do this

# Linker, Loader, Link-Loader

- Linker or linkage-editor or link-editor
  - The “ld” program. Does linking.
- Loader
  - The exec(). It loads an executable in the memory.
- Link-Loader
  - Often the linker is called link-loader in literature. Because where were days when the linker and loader’s jobs were quite over-lapping.

# Static, dynamic / linking, loading

- Both linking and loading can be
  - Static or dynamic
  - More about this when we learn memory management
- An important fundamental:
  - memory management features of processor, memory management architecture of kernel, executable/object-code file format, output of linker and job of loader, are all interdependent and in-separable.

# Cross-compiler

- Compiler on system-A, but generate object-code file for system-B (target system)
  - E.g. compile on Ubuntu, but create an EXE for windows
- Normally used when there is no compiler available on target system
  - see gcc -m option
- See [https://wiki.osdev.org/GCC\\_Cross-Compiler](https://wiki.osdev.org/GCC_Cross-Compiler)

# **Inter Process Communication**

# Revision of process related concepts

- PCB, struct proc
- Process lifecycle – different states
- Queues/Lists of processes
- What is “Blocking”
- Event driven kernel

# Before IPC, let's learn more about file related system calls

- Redirection

`ls > /tmp/file`

`cat < /etc/passwd`

- how does this work?
- File descriptors are inherited across fork

# 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

# Shared Memory Vs Message Passing

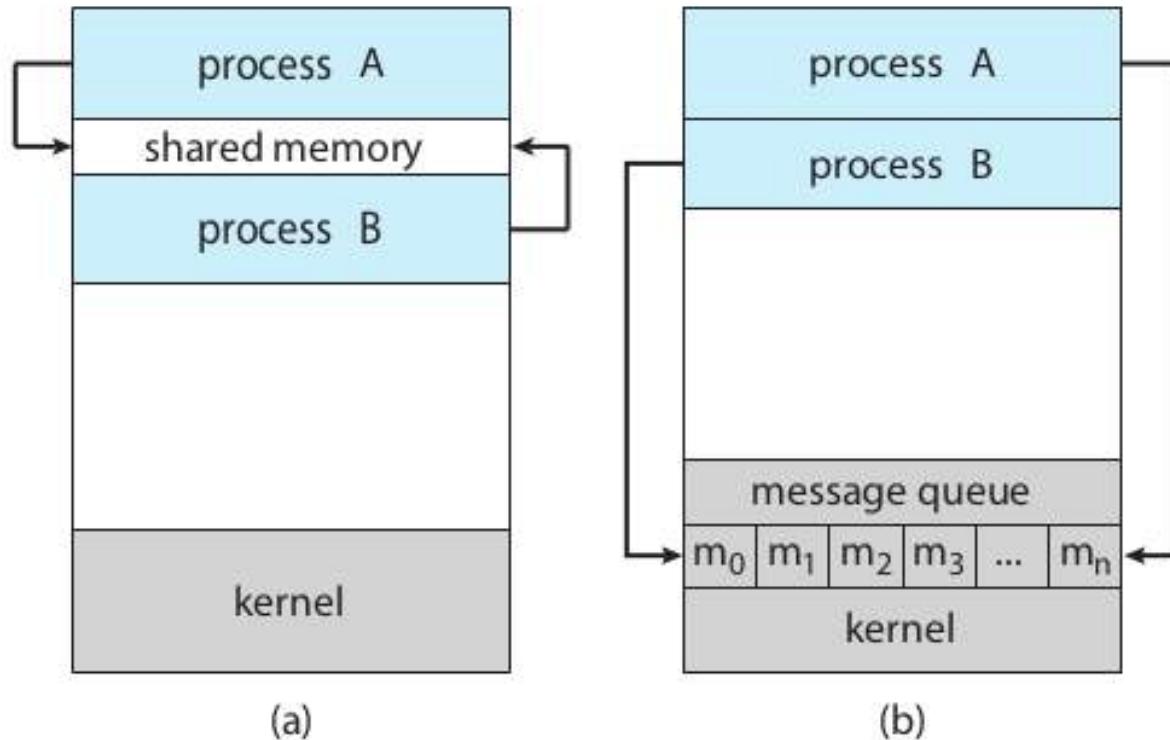


Figure 3.11 Communications models. (a) Shared memory. (b) Message passing.

- Each requires OS to provide system calls for
  - Creating the IPC mechanism
  - To read/write

# **Typical code of shared memory type of solutions**

**Process P1**

**Process P2**

# **Typical code of message passing type solutions**

**Process P1**

**Process P2**

# 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];
```

# 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

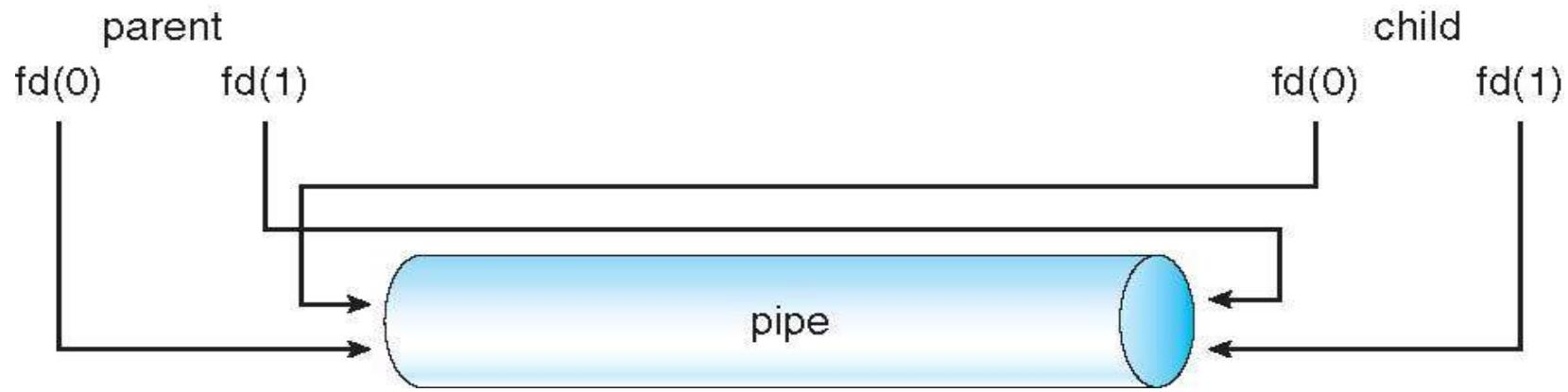
# Pipes for IPC

- **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



# 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

# 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 processes could write to the shared

# 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:**

# 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

# **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

# 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

# **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**

# 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





## Ext2

The **Second Extended Filesystem (ext2fs)** is a rewrite of the original *Extended Filesystem* and as such, is also based around the concept of "inodes." Ext2 served as the de facto filesystem of Linux for nearly a decade from the early 1990s to the early 2000s when it was superseded by the journaling file systems [ext3](#) and [ReiserFS](#). It has native support for UNIX ownership / access rights, symbolic- and hard-links, and other properties that are common among UNIX-like operating systems. Organizationally, it divides disk space up into groups called "block groups." Having these groups results in distribution of data across the disk which helps to minimize head movement as well as the impact of fragmentation. Further, some (if not all) groups are required to contain backups of important data that can be used to rebuild the file system in the event of disaster.

*Note: Most of the information here is based off of work done by Dave Poirier on the ext2-doc project (see the [links](#) section) which is graciously released under the [GNU Free Documentation License](#). Be sure to buy him a beer the next time you see him.*

### Contents [hide]

- 1 Basic Concepts
  - 1.1 What is a Block?
  - 1.2 What is a Block Group?
  - 1.3 What is an Inode?
- 2 Superblock
  - 2.1 Locating the Superblock
  - 2.2 Determining the Number of Block Groups
  - 2.3 Base Superblock Fields
    - 2.3.1 File System States
    - 2.3.2 Error Handling Methods
    - 2.3.3 Creator Operating System IDs
  - 2.4 Extended Superblock Fields
    - 2.4.1 Optional Feature Flags
    - 2.4.2 Required Feature Flags
    - 2.4.3 Read-Only Feature Flags
- 3 Block Group Descriptor Table
  - 3.1 Locating the Block Group Descriptor Table
  - 3.2 Block Group Descriptor
- 4 Inodes
  - 4.1 Determining which Block Group contains an Inode
  - 4.2 Finding an inode inside of a Block Group
  - 4.3 Reading the contents of an inode
  - 4.4 Inode Data Structure
    - 4.4.1 Inode Type and Permissions
    - 4.4.2 Inode Flags
    - 4.4.3 OS Specific Value 1
    - 4.4.4 OS Specific Value 2
  - 4.5 Directories
  - 4.6 Directory Entry
    - 4.6.1 Directory Entry Type Indicators
- 5 Quick Summaries
  - 5.1 How To Read An Inode
  - 5.2 How To Read the Root Directory
- 6 See Also
  - 6.1 External Links

### Basic Concepts

**Important Note:** All values are little-endian unless otherwise specified

#### What is a Block?

The Ext2 file system divides up disk space into logical blocks of contiguous space. The size of blocks need not be the same size as the sector size of the disk the file system resides on. The size of blocks can be determined by reading the field starting at byte 24 in the [Superblock](#).

#### What is a Block Group?

Blocks, along with inodes, are divided up into "block groups." These are nothing more than contiguous groups of blocks.

Each block group reserves a few of its blocks for special purposes such as:

- A bitmap of free/allocated blocks within the group
- A bitmap of allocated inodes within the group
- A table of inode structures that belong to the group
- Depending upon the revision of Ext2 used, some or all block groups may also contain a backup copy of the [Superblock](#) and the [Block Group Descriptor Table](#).

#### What is an Inode?

An inode is a structure on the disk that represents a file, directory, symbolic link, etc. Inodes do not contain the data of the file / directory / etc. that they represent. Instead, they link to the blocks that actually contain the data. This lets the inodes themselves have a well-defined size which lets them be placed in easily indexed arrays. Each block group has an array of inodes it is responsible for, and conversely every inode within a file system belongs to one of such tables (and one of such block groups).

### Superblock

The first step in implementing an Ext2 driver is to find, extract, and parse the superblock. The Superblock contains all information about the layout of the file system and possibly contains other important information like what optional features were used to create the file system. Once you have finished with the Superblock, the next step is to look at the [Block Group Descriptor Table](#)

#### Locating the Superblock

The Superblock is always located at byte 1024 from the beginning of the volume and is exactly 1024 bytes in length. For example, if the disk uses 512 byte sectors, the Superblock will begin at LBA 2 and will occupy all of sector 2 and 3.

#### Determining the Number of Block Groups

From the Superblock, extract the size of each block, the total number of inodes, the total number of blocks, the number of blocks per block group, and the number of inodes in each block group. From this information we can infer the number of block groups there are by:

- Rounding up the total number of blocks divided by the number of blocks per block group
- Rounding up the total number of inodes divided by the number of inodes per block group
- Both (and check them against each other)

#### Base Superblock Fields

These fields are present in all versions of Ext2

### Filesystems

#### Virtual Filesystems

VFS

#### Disk Filesystems

FAT 12/16/32, VFAT, ExFAT

Ext 2/3/4

LEAN

HPFS

NTFS

HFS

HFS+

MFS

ReiserFS

FFS (Amiga)

FFS (BSD/UFS)

BeFS

BFS

XFS

SFS

ZDSFS

ZFS

USTAR

#### CD/DVD Filesystems

ISO 9660

Joliet

UDF

#### Network Filesystems

NFS

RFS

AFS

#### Flash Filesystems

JFFS2

YAFFS

| Starting Byte | Ending Byte | Size in Bytes | Field Description                                                                                                        |
|---------------|-------------|---------------|--------------------------------------------------------------------------------------------------------------------------|
| 0             | 3           | 4             | Total number of inodes in file system                                                                                    |
| 4             | 7           | 4             | Total number of blocks in file system                                                                                    |
| 8             | 11          | 4             | Number of blocks reserved for superuser (see offset 80)                                                                  |
| 12            | 15          | 4             | Total number of unallocated blocks                                                                                       |
| 16            | 19          | 4             | Total number of unallocated inodes                                                                                       |
| 20            | 23          | 4             | Block number of the block containing the superblock (also the starting block number, NOT always zero.)                   |
| 24            | 27          | 4             | $\log_2$ (block size) - 10. (In other words, the number to shift 1,024 to the left by to obtain the block size)          |
| 28            | 31          | 4             | $\log_2$ (fragment size) - 10. (In other words, the number to shift 1,024 to the left by to obtain the fragment size)    |
| 32            | 35          | 4             | Number of blocks in each block group                                                                                     |
| 36            | 39          | 4             | Number of fragments in each block group                                                                                  |
| 40            | 43          | 4             | Number of inodes in each block group                                                                                     |
| 44            | 47          | 4             | Last mount time (in POSIX time <a href="#">?</a> )                                                                       |
| 48            | 51          | 4             | Last written time (in POSIX time <a href="#">?</a> )                                                                     |
| 52            | 53          | 2             | Number of times the volume has been mounted since its last consistency check ( <a href="#">fsck</a> <a href="#">?</a> )  |
| 54            | 55          | 2             | Number of mounts allowed before a consistency check ( <a href="#">fsck</a> <a href="#">?</a> ) must be done              |
| 56            | 57          | 2             | Ext2 signature (0xef53), used to help confirm the presence of Ext2 on a volume                                           |
| 58            | 59          | 2             | File system state ( <a href="#">see below</a> )                                                                          |
| 60            | 61          | 2             | What to do when an error is detected ( <a href="#">see below</a> )                                                       |
| 62            | 63          | 2             | Minor portion of version (combine with Major portion below to construct full version field)                              |
| 64            | 67          | 4             | POSIX time <a href="#">?</a> of last consistency check ( <a href="#">fsck</a> <a href="#">?</a> )                        |
| 68            | 71          | 4             | Interval (in POSIX time <a href="#">?</a> ) between forced consistency checks ( <a href="#">fsck</a> <a href="#">?</a> ) |
| 72            | 75          | 4             | Operating system ID from which the filesystem on this volume was created ( <a href="#">see below</a> )                   |
| 76            | 79          | 4             | Major portion of version (combine with Minor portion above to construct full version field)                              |
| 80            | 81          | 2             | User ID that can use reserved blocks                                                                                     |
| 82            | 83          | 2             | Group ID that can use reserved blocks                                                                                    |

#### File System States

| Value | State Description      |
|-------|------------------------|
| 1     | File system is clean   |
| 2     | File system has errors |

#### Error Handling Methods

| Value | Action to Take                   |
|-------|----------------------------------|
| 1     | Ignore the error (continue on)   |
| 2     | Remount file system as read-only |
| 3     | Kernel panic                     |

#### Creator Operating System IDs

| Value | Operating System                                                                                                                           |
|-------|--------------------------------------------------------------------------------------------------------------------------------------------|
| 0     | Linux <a href="#">?</a>                                                                                                                    |
| 1     | GNU HURD <a href="#">?</a>                                                                                                                 |
| 2     | MASIX (an operating system developed by Rémy Card, one of the developers of ext2)                                                          |
| 3     | FreeBSD <a href="#">?</a>                                                                                                                  |
| 4     | Other "Lites" (BSD4.4-Lite derivatives such as NetBSD <a href="#">?</a> , OpenBSD <a href="#">?</a> , XNU/Darwin <a href="#">?</a> , etc.) |

#### Extended Superblock Fields

These fields are only present if Major version (specified in the base superblock fields), is greater than or equal to 1.

| Starting Byte | Ending Byte | Size in Bytes | Field Description                                                                                                                                     |
|---------------|-------------|---------------|-------------------------------------------------------------------------------------------------------------------------------------------------------|
| 84            | 87          | 4             | First non-reserved inode in file system. (In versions < 1.0, this is fixed as 11)                                                                     |
| 88            | 89          | 2             | Size of each inode structure in bytes. (In versions < 1.0, this is fixed as 128)                                                                      |
| 90            | 91          | 2             | Block group that this superblock is part of (if backup copy)                                                                                          |
| 92            | 95          | 4             | Optional features present (features that are not required to read or write, but usually result in a performance increase. <a href="#">see below</a> ) |
| 96            | 99          | 4             | Required features present (features that are required to be supported to read or write. <a href="#">see below</a> )                                   |
| 100           | 103         | 4             | Features that if not supported, the volume must be mounted read-only <a href="#">see below</a> )                                                      |
| 104           | 119         | 16            | File system ID (what is output by blkid)                                                                                                              |
| 120           | 135         | 16            | Volume name (C-style string: characters terminated by a 0 byte)                                                                                       |
| 136           | 199         | 64            | Path volume was last mounted to (C-style string: characters terminated by a 0 byte)                                                                   |
| 200           | 203         | 4             | Compression algorithms used (see Required features above)                                                                                             |
| 204           | 204         | 1             | Number of blocks to preallocate for files                                                                                                             |
| 205           | 205         | 1             | Number of blocks to preallocate for directories                                                                                                       |

|     |      |    |                                                     |
|-----|------|----|-----------------------------------------------------|
| 206 | 207  | 2  | (Unused)                                            |
| 208 | 223  | 16 | Journal ID (same style as the File system ID above) |
| 224 | 227  | 4  | Journal inode                                       |
| 228 | 231  | 4  | Journal device                                      |
| 232 | 235  | 4  | Head of orphan inode list                           |
| 236 | 1023 | X  | (Unused)                                            |

### Optional Feature Flags

These are optional features for an implementation to support, but offer performance or reliability gains to implementations that do support them.

| Flag Value | Description                                                                                                                                        |
|------------|----------------------------------------------------------------------------------------------------------------------------------------------------|
| 0x0001     | Preallocate some number of (contiguous?) blocks (see byte 205 in the superblock) to a directory when creating a new one (to reduce fragmentation?) |
| 0x0002     | AFS server inodes exist                                                                                                                            |
| 0x0004     | File system has a journal (Ext3)                                                                                                                   |
| 0x0008     | Inodes have extended attributes                                                                                                                    |
| 0x0010     | File system can resize itself for larger partitions                                                                                                |
| 0x0020     | Directories use hash index                                                                                                                         |

### Required Feature Flags

These features if present on a file system are required to be supported by an implementation in order to correctly read from or write to the file system.

| Flag Value | Description                             |
|------------|-----------------------------------------|
| 0x0001     | Compression is used                     |
| 0x0002     | Directory entries contain a type field  |
| 0x0004     | File system needs to replay its journal |
| 0x0008     | File system uses a journal device       |

### Read-Only Feature Flags

These features, if present on a file system, are required in order for an implementation to write to the file system, but are not required to read from the file system.

| Flag Value | Description                                                                  |
|------------|------------------------------------------------------------------------------|
| 0x0001     | Sparse superblocks and group descriptor tables                               |
| 0x0002     | File system uses a 64-bit file size                                          |
| 0x0004     | Directory contents are stored in the form of a <a href="#">Binary Tree</a> ☰ |

## Block Group Descriptor Table

The Block Group Descriptor Table contains a descriptor for each block group within the file system. The number of block groups within the file system, and correspondingly, the number of entries in the Block Group Descriptor Table, is described [above](#). Each descriptor contains information regarding where important data structures for that group are located.

### Locating the Block Group Descriptor Table

The table is located in the block immediately following the Superblock. So if the block size (determined from a field in the superblock) is 1024 bytes per block, the Block Group Descriptor Table will begin at block 2. For any other block size, it will begin at block 1. Remember that blocks are numbered starting at 0, and that block numbers don't usually correspond to physical block addresses.

### Block Group Descriptor

A Block Group Descriptor contains information regarding where important data structures for that block group are located.

| Starting Byte | Ending Byte | Size in Bytes | Field Description                     |
|---------------|-------------|---------------|---------------------------------------|
| 0             | 3           | 4             | Block address of block usage bitmap   |
| 4             | 7           | 4             | Block address of inode usage bitmap   |
| 8             | 11          | 4             | Starting block address of inode table |
| 12            | 13          | 2             | Number of unallocated blocks in group |
| 14            | 15          | 2             | Number of unallocated inodes in group |
| 16            | 17          | 2             | Number of directories in group        |
| 18            | 31          | X             | (Unused)                              |

## Inodes

Like blocks, each inode has a numerical address. It is extremely important to note that unlike block addresses, **inode addresses start at 1**.

With Ext2 versions prior to Major version 1, inodes 1 to 10 are reserved and should be in an allocated state. Starting with version 1, the first non-reserved inode is indicated via a field in the Superblock. Of the reserved inodes, number 2 subjectively has the most significance as it is used for the root directory.

Inodes have a fixed size of either 128 for version 0 Ext2 file systems, or as dictated by the field in the Superblock for version 1 file systems. All inodes reside in inode tables that belong to block groups. Therefore, looking up an inode is simply a matter of determining which block group it belongs to and indexing that block group's inode table.

### Determining which Block Group contains an Inode

From an inode address (remember that they start at 1), we can determine which group the inode is in, by using the formula:

```
block group = (inode - 1) / INODES_PER_GROUP
```

where INODES\_PER\_GROUP is a field in the Superblock

### Finding an inode inside of a Block Group

Once we know which group an inode resides in, we can look up the actual inode by first retrieving that block group's inode table's starting address (see [Block Group Descriptor](#) above). The index of our inode in this block group's inode table can be determined by using the formula:

```
index = (inode - 1) % INODES_PER_GROUP
```

where % denotes the [Modulo operation](#) and INODES\_PER\_GROUP is a field in the Superblock (the same field which was used to determine which block group the inode belongs to).

Next, we have to determine which block contains our inode. This is achieved from:

```
containing block = (index * INODE_SIZE) / BLOCK_SIZE
```

where INODE\_SIZE is either fixed at 128 if VERSION < 1 or defined by a field in the Superblock if VERSION >= 1.0, and BLOCK\_SIZE is defined by a field in the Superblock.

Finally, mask and shift as necessary to extract only the inode data from the containing block.

### Reading the contents of an inode

Each inode contains 12 direct pointers, one singly indirect pointer, one doubly indirect block pointer, and one triply indirect pointer. The direct space "overflows" into the singly indirect space, which overflows into the doubly indirect space, which overflows into the triply indirect space.

**Direct Block Pointers:** There are 12 direct block pointers. If valid, the value is non-zero. Each pointer is the block address of a block containing data for this inode.

**Singly Indirect Block Pointer:** If a file needs more than 12 blocks, a separate block is allocated to store the block addresses of the remaining data blocks needed to store its contents. This separate block is called an indirect block because it adds an extra step (a level of indirection) between an inode and its data. The block addresses stored in the block are all 32-bit, and the capacity of stored addresses in this block is a function of the block size. The address of this indirect block is stored in the inode in the "Singly Indirect Block Pointer" field.

**Doubly Indirect Block Pointer:** If a file has more blocks than can fit in the 12 direct pointers and the indirect block, a double indirect block is used. A double indirect block is an extension of the indirect block described above only now we have two intermediate blocks between the inode and data blocks. The inode structure has a "Doubly Indirect Block Pointer" field that points to this block if necessary.

**Triply Indirect Block Pointer:** Lastly, if a file needs still more space, it can use a triple indirect block. Again, this is an extension of the double indirect block. So, a triple indirect block contains addresses of double indirect blocks, which contain addresses of single indirect blocks, which contain address of data blocks. The inode structure has a "Triply Indirect Block Pointer" field that points to this block if present.

[This image from Wikipedia](#) illustrates what is described above pretty well.

### Inode Data Structure

| Starting Byte | Ending Byte | Size in Bytes | Field Description                                                                                                                                             |
|---------------|-------------|---------------|---------------------------------------------------------------------------------------------------------------------------------------------------------------|
| 0             | 1           | 2             | Type and Permissions ( <a href="#">see below</a> )                                                                                                            |
| 2             | 3           | 2             | User ID                                                                                                                                                       |
| 4             | 7           | 4             | Lower 32 bits of size in bytes                                                                                                                                |
| 8             | 11          | 4             | Last Access Time (in <a href="#">POSIX time</a> )                                                                                                             |
| 12            | 15          | 4             | Creation Time (in <a href="#">POSIX time</a> )                                                                                                                |
| 16            | 19          | 4             | Last Modification time (in <a href="#">POSIX time</a> )                                                                                                       |
| 20            | 23          | 4             | Deletion time (in <a href="#">POSIX time</a> )                                                                                                                |
| 24            | 25          | 2             | Group ID                                                                                                                                                      |
| 26            | 27          | 2             | Count of hard links (directory entries) to this inode. When this reaches 0, the data blocks are marked as unallocated.                                        |
| 28            | 31          | 4             | Count of disk sectors (not Ext2 blocks) in use by this inode, not counting the actual inode structure nor directory entries linking to the inode.             |
| 32            | 35          | 4             | Flags ( <a href="#">see below</a> )                                                                                                                           |
| 36            | 39          | 4             | Operating System Specific value #1                                                                                                                            |
| 40            | 43          | 4             | Direct Block Pointer 0                                                                                                                                        |
| 44            | 47          | 4             | Direct Block Pointer 1                                                                                                                                        |
| 48            | 51          | 4             | Direct Block Pointer 2                                                                                                                                        |
| 52            | 55          | 4             | Direct Block Pointer 3                                                                                                                                        |
| 56            | 59          | 4             | Direct Block Pointer 4                                                                                                                                        |
| 60            | 63          | 4             | Direct Block Pointer 5                                                                                                                                        |
| 64            | 67          | 4             | Direct Block Pointer 6                                                                                                                                        |
| 68            | 71          | 4             | Direct Block Pointer 7                                                                                                                                        |
| 72            | 75          | 4             | Direct Block Pointer 8                                                                                                                                        |
| 76            | 79          | 4             | Direct Block Pointer 9                                                                                                                                        |
| 80            | 83          | 4             | Direct Block Pointer 10                                                                                                                                       |
| 84            | 87          | 4             | Direct Block Pointer 11                                                                                                                                       |
| 88            | 91          | 4             | Singly Indirect Block Pointer (Points to a block that is a list of block pointers to data)                                                                    |
| 92            | 95          | 4             | Doubly Indirect Block Pointer (Points to a block that is a list of block pointers to Singly Indirect Blocks)                                                  |
| 96            | 99          | 4             | Triply Indirect Block Pointer (Points to a block that is a list of block pointers to Doubly Indirect Blocks)                                                  |
| 100           | 103         | 4             | Generation number (Primarily used for NFS)                                                                                                                    |
| 104           | 107         | 4             | In Ext2 version 0, this field is reserved. In version >= 1, Extended attribute block (File ACL).                                                              |
| 108           | 111         | 4             | In Ext2 version 0, this field is reserved. In version >= 1, Upper 32 bits of file size (if feature bit set) if it's a file, Directory ACL if it's a directory |
| 112           | 115         | 4             | Block address of fragment                                                                                                                                     |
| 116           | 127         | 12            | Operating System Specific Value #2                                                                                                                            |

### Inode Type and Permissions

The type indicator occupies the top hex digit (bits 15 to 12) of this 16-bit field

| Type value in hex | Type Description |
|-------------------|------------------|
| 0x1000            | FIFO             |
| 0x2000            | Character device |
| 0x4000            | Directory        |
| 0x6000            | Block device     |

|        |               |
|--------|---------------|
| 0x8000 | Regular file  |
| 0xA000 | Symbolic link |
| 0xC000 | Unix socket   |

Permissions occupy the bottom 12 bits of this 16-bit field

| Permission value in hex | Permission value in octal | Permission Description   |
|-------------------------|---------------------------|--------------------------|
| 0x001                   | 00001                     | Other—execute permission |
| 0x002                   | 00002                     | Other—write permission   |
| 0x004                   | 00004                     | Other—read permission    |
| 0x008                   | 00010                     | Group—execute permission |
| 0x010                   | 00020                     | Group—write permission   |
| 0x020                   | 00040                     | Group—read permission    |
| 0x040                   | 00100                     | User—execute permission  |
| 0x080                   | 00200                     | User—write permission    |
| 0x100                   | 00400                     | User—read permission     |
| 0x200                   | 01000                     | Sticky Bit               |
| 0x400                   | 02000                     | Set group ID             |
| 0x800                   | 04000                     | Set user ID              |

#### Inode Flags

| Flag Value | Description                                                 |
|------------|-------------------------------------------------------------|
| 0x00000001 | Secure deletion (not used)                                  |
| 0x00000002 | Keep a copy of data when deleted (not used)                 |
| 0x00000004 | File compression (not used)                                 |
| 0x00000008 | Synchronous updates—new data is written immediately to disk |
| 0x00000010 | Immutable file (content cannot be changed)                  |
| 0x00000020 | Append only                                                 |
| 0x00000040 | File is not included in 'dump' command                      |
| 0x00000080 | Last accessed time should not be updated                    |
| ...        | (Reserved)                                                  |
| 0x00010000 | Hash indexed directory                                      |
| 0x00020000 | AFS directory                                               |
| 0x00040000 | Journal file data                                           |

#### OS Specific Value 1

| Operating System | How they use this field |
|------------------|-------------------------|
| Linux            | (reserved)              |
| HURD             | "translator"?           |
| MASIX            | (reserved)              |

#### OS Specific Value 2

| Operating System | How they use this field |             |               |                                                                       |
|------------------|-------------------------|-------------|---------------|-----------------------------------------------------------------------|
|                  | Starting Byte           | Ending Byte | Size in Bytes | Field Description                                                     |
| Linux            | 116                     | 116         | 1             | Fragment number                                                       |
|                  | 117                     | 117         | 1             | Fragment size                                                         |
|                  | 118                     | 119         | 2             | (reserved)                                                            |
|                  | 120                     | 121         | 2             | High 16 bits of 32-bit User ID                                        |
|                  | 122                     | 123         | 2             | High 16 bits of 32-bit Group ID                                       |
|                  | 124                     | 127         | 4             | (reserved)                                                            |
| HURD             | Starting Byte           | Ending Byte | Size in Bytes | Field Description                                                     |
|                  | 116                     | 116         | 1             | Fragment number                                                       |
|                  | 117                     | 117         | 1             | Fragment size                                                         |
|                  | 118                     | 119         | 2             | High 16 bits of 32-bit "Type and Permissions" field                   |
|                  | 120                     | 121         | 2             | High 16 bits of 32-bit User ID                                        |
|                  | 122                     | 123         | 2             | High 16 bits of 32-bit Group ID                                       |
|                  | 124                     | 127         | 4             | User ID of author (if == 0xFFFFFFFF, the normal User ID will be used) |

| MASIX | Starting Byte | Ending Byte | Size in Bytes | Field Description |
|-------|---------------|-------------|---------------|-------------------|
|       | 116           | 116         | 1             | Fragment number   |
|       | 117           | 117         | 1             | Fragment size     |
|       | 118           | 127         | X             | (reserved)        |

## Directories

Directories are inodes which contain some number of "entries" as their contents. These entries are nothing more than a name/inode pair. For instance the inode corresponding to the root directory might have an entry with the name of "etc" and an inode value of 50. A directory inode stores these entries in a linked-list fashion in its contents blocks.

The root directory is Inode 2.

The total size of a directory entry may be longer then the length of the name would imply (The name may not span to the end of the record), and records have to be aligned to 4-byte boundaries. Directory entries are also not allowed to span multiple blocks on the file-system, so there may be empty space in-between directory entries. Empty space is however not allowed in-between directory entries, so any possible empty space will be used as part of the preceding record by increasing its record length to include the empty space. Empty space may also be equivalently marked by a separate directory entry with an inode number of zero, indicating that directory entry should be skipped.

## Directory Entry

| Starting Byte | Ending Byte | Size in Bytes | Field Description                                                                                                                                        |
|---------------|-------------|---------------|----------------------------------------------------------------------------------------------------------------------------------------------------------|
| 0             | 3           | 4             | Inode                                                                                                                                                    |
| 4             | 5           | 2             | Total size of this entry (Including all subfields)                                                                                                       |
| 6             | 6           | 1             | Name Length least-significant 8 bits                                                                                                                     |
| 7             | 7           | 1             | Type indicator (only if the feature bit for "directory entries have file type byte" is set, else this is the most-significant 8 bits of the Name Length) |
| 8             | 8+N-1       | N             | Name characters                                                                                                                                          |

## Directory Entry Type Indicators

| Value | Type Description          |
|-------|---------------------------|
| 0     | Unknown type              |
| 1     | Regular file              |
| 2     | Directory                 |
| 3     | Character device          |
| 4     | Block device              |
| 5     | FIFO                      |
| 6     | Socket                    |
| 7     | Symbolic link (soft link) |

## Quick Summaries

---

### How To Read An Inode

1. Read the Superblock to find the size of each block, the number of blocks per group, number Inodes per group, and the starting block of the first group (Block Group Descriptor Table).
2. Determine which block group the inode belongs to.
3. Read the Block Group Descriptor corresponding to the Block Group which contains the inode to be looked up.
4. From the Block Group Descriptor, extract the location of the block group's inode table.
5. Determine the index of the inode in the inode table.
6. Index the inode table (taking into account non-standard inode size).

Directory entry information and file contents are located within the data blocks that the Inode points to.

### How To Read the Root Directory

The root directory's inode is defined to always be 2. Read/parse the contents of inode 2.

## See Also

---

### External Links

- ext2-doc project: Second Extended File System - implementation-oriented documentation, describes internal structure in human language.
- Design and Implementation of the Second Extended Filesystem (overview)
- State of the Art: Where we are with the Ext3 filesystem - Paper by Mingming Cao, Theodore Y. Ts'o, Badri Pulavarty, and Suparna Bhattacharya describing extended features for ext2

Category: Filesystems

This page was last modified on 23 July 2022, at 23:33.

This page has been accessed 235,371 times.

[Privacy policy](#) [About OSDev Wiki](#) [Disclaimers](#)



# setjmp, Longjmp and User-Level Threads

*Failures are much more fun to hear about afterward,  
they are not so funny at the time.*

# **setjmp and longjmp**

- ❑ You can jump between functions, conditionally!
- ❑ Use a jump buffer to save the return point.
- ❑ Use a long-jump to jump to a return point.
- ❑ Header file `setjmp.h` is required.
- ❑ A jump buffer is of type `jmp_buf`.
- ❑ Set up a jump buffer with function `setjmp()`.
- ❑ Execute a long jump with function `longjmp()`.

# Here is the concept

- ❑ Declare a variable of type jmp\_buf:  
`jmp_buf JumpBuffer;`
- ❑ Call function setjmp () to mark a return point:  
`setjmp (JumpBuffer) ;`
- ❑ Later on, use function longjmp () to jump back:  
`longjmp (JumpBuffer, 1) ;`

The jump buffer is used in both calls

The meaning of this argument will be clear later

# But, you need to know more!

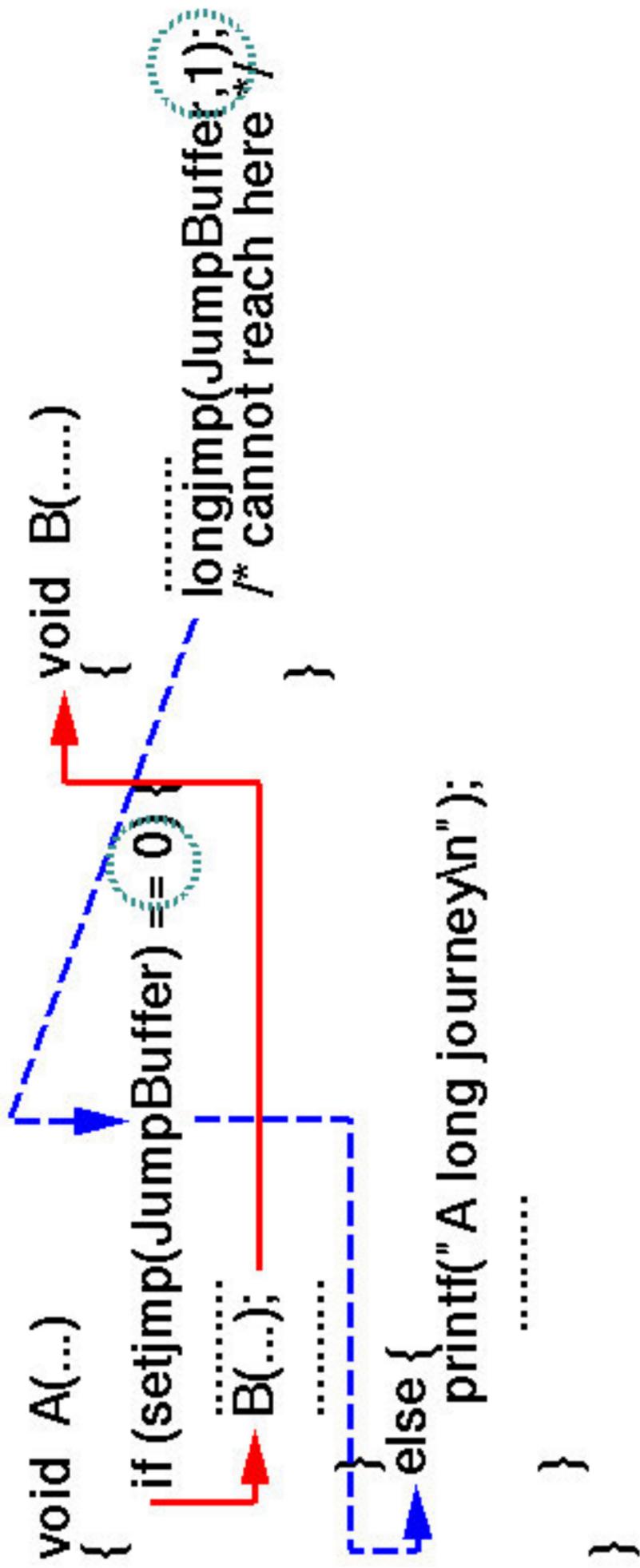
- When `setjmp()` is called, it saves the current state of execution and returns 0.
- When `longjmp()` is called, it sends the control back to a marked return point and let `setjmp()` to return its *second* argument?????

```
#include <setjmp.h>
jmp_buf Buf;
void A(...)

{
 if (setjmp(Buf)==0) {
 printf("Marked!\n");
 /* other statements */
 B(...); first time here
 }
 else {
 printf("Returned from"
 " a long journey\n");
 /* other statement */
 }
}

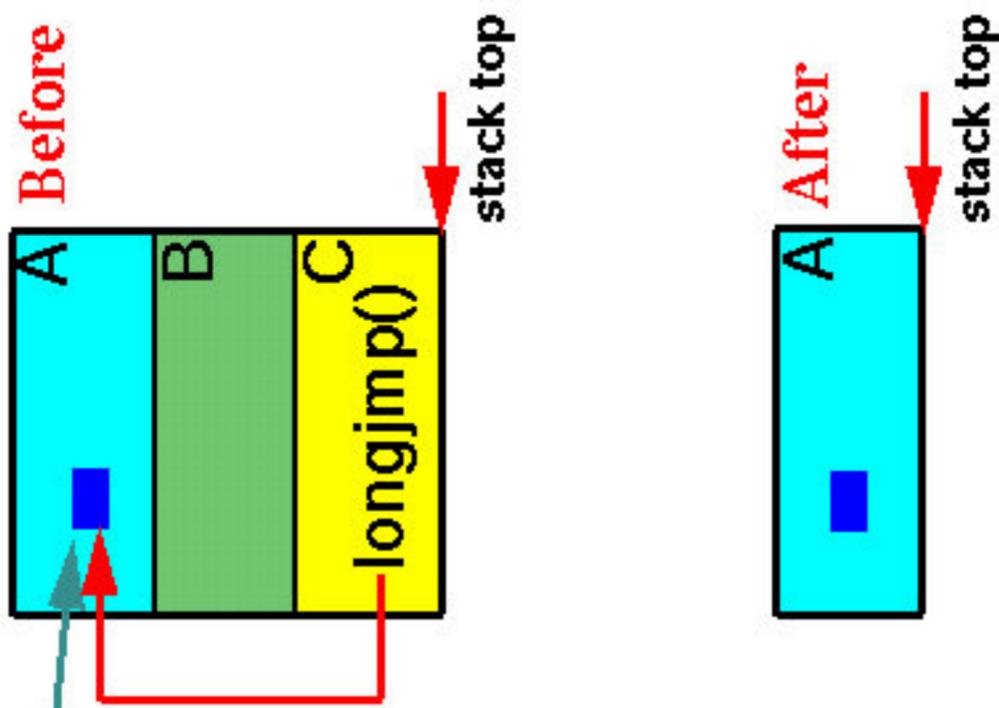
void B(...)
{
 /* other statement */
 longjmp(Buf, 1);
}
```

# Control Flow of setjmp() and longjmp()



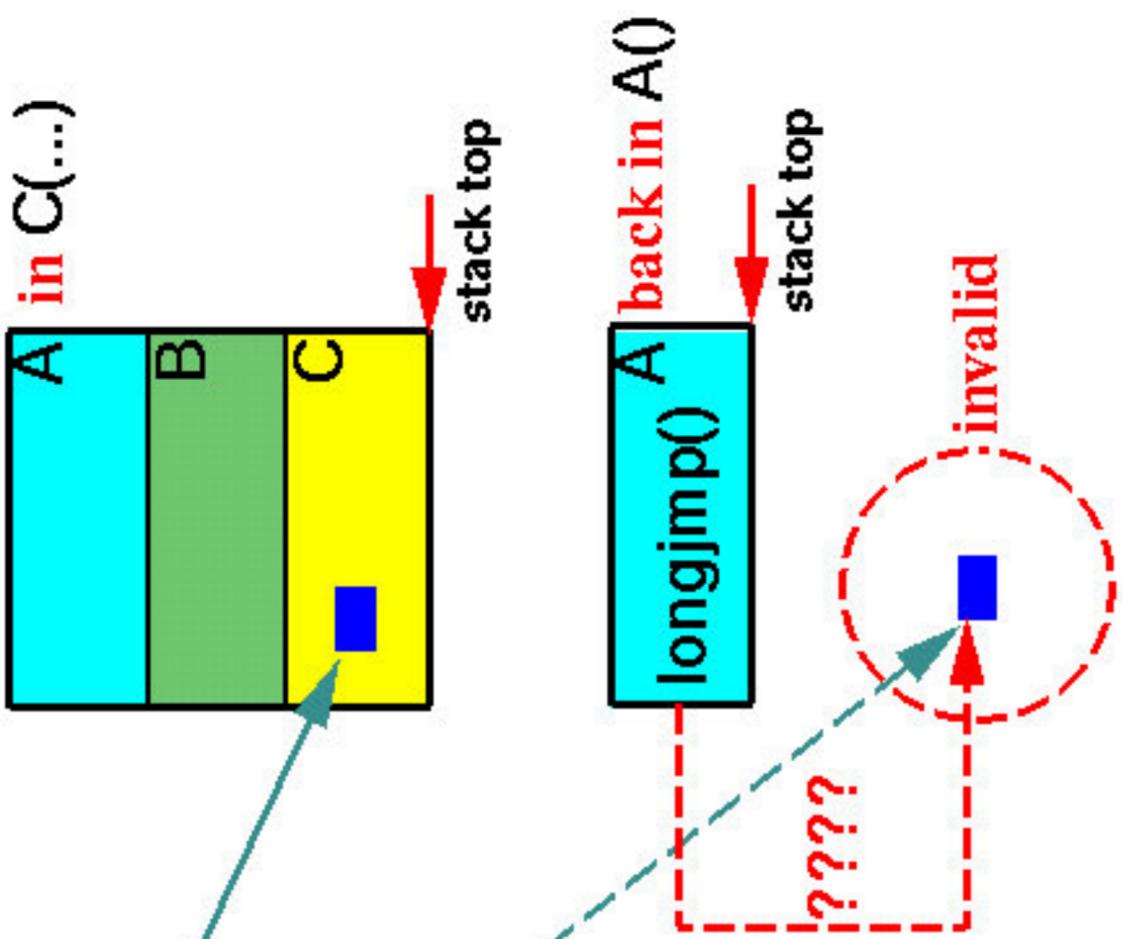
# The content of a jmp buffer when execute a longjmp() must be valid

```
jmp_buf Buf;
void A(...){ if (setjmp(Buf) == 0)
 B(...);
else
 printf("I am back\n");
}
void B(...){ C(...) }
void C(...){ longjmp(Buf,1); }
```

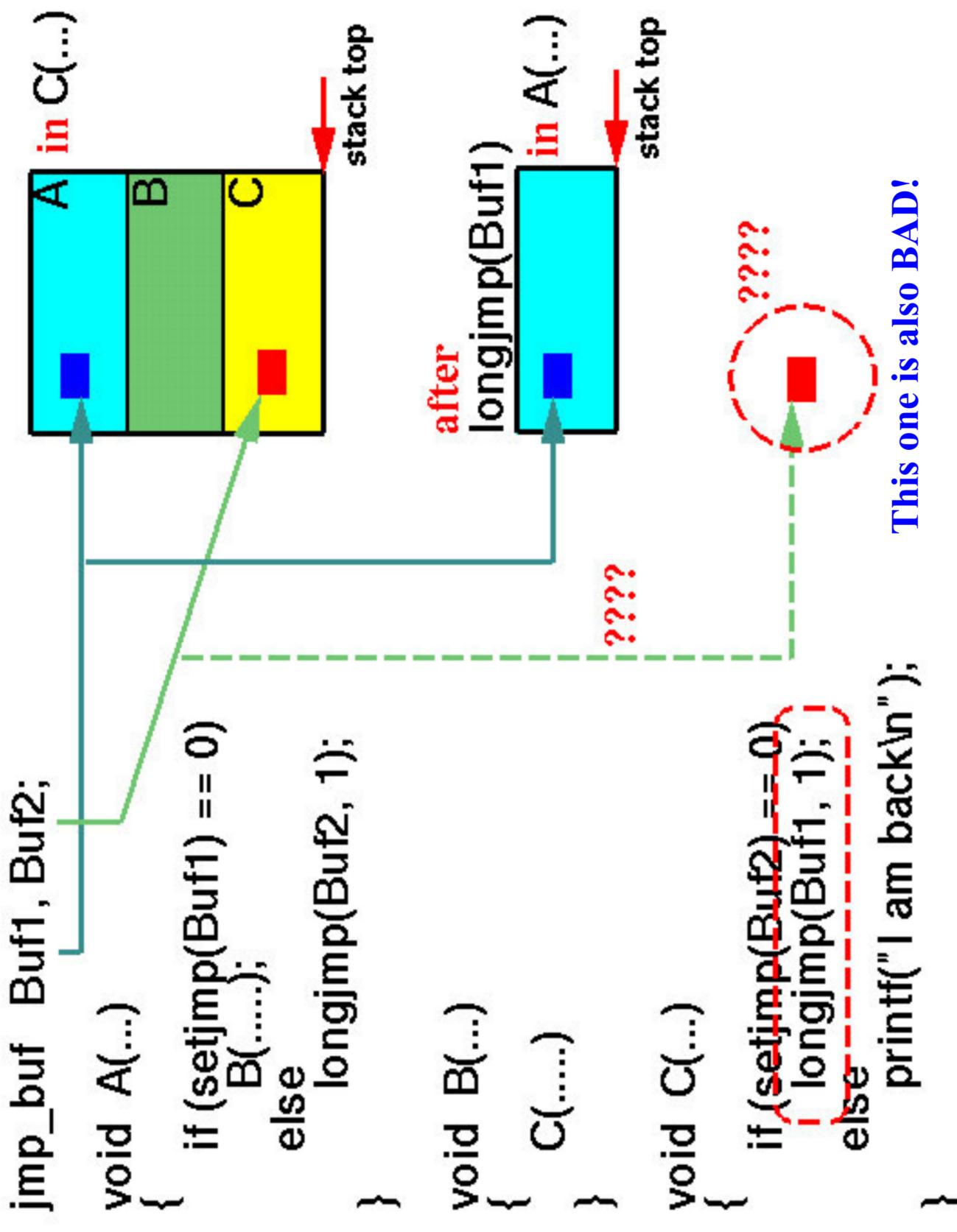


# The content of a jump buffer when execute a longjmp() must be valid

```
jmp_buf Buf;
void A(...){ B(...); longjmp(Buf,1); }
void B(...){ C(...); }
void C(...){
 if (setjmp(Buf) == 0)
 return;
 else
 printf("I am back\n");
}
```



# The content of a jump buffer when execute a longjmp() must be valid



# Jump Buffer Example: Factorial: 1/3

```
#include <stdio.h>
#include <setjmp.h>

jmp_buf JumpBuffer;

int result;

void main(int argc, char *argv[])
{
 int n;
 n = atoi(argv[1]);
 if (setjmp(JumpBuffer) == 0)
 factorial(n);
 else
 printf("%d\n", n, result);
 exit(0);
}
```

Execution will return to here!

Result is in here!

```
graph LR; A["Result is in here!"] --> B["Result"]; A --> C["Execution will return to here!"]
```

# Jump Buffer Example: Factorial: 2/3

```
void factorial(int n)
{
 fact(1, n);
}
```

```
void fact(int Result, int Count, int n)
{
 if (Count <= n)
 fact(Result*Count, Count+1);
 else {
 result = Result;
 longjmp(JumpBuffer, 1);
 }
}
```

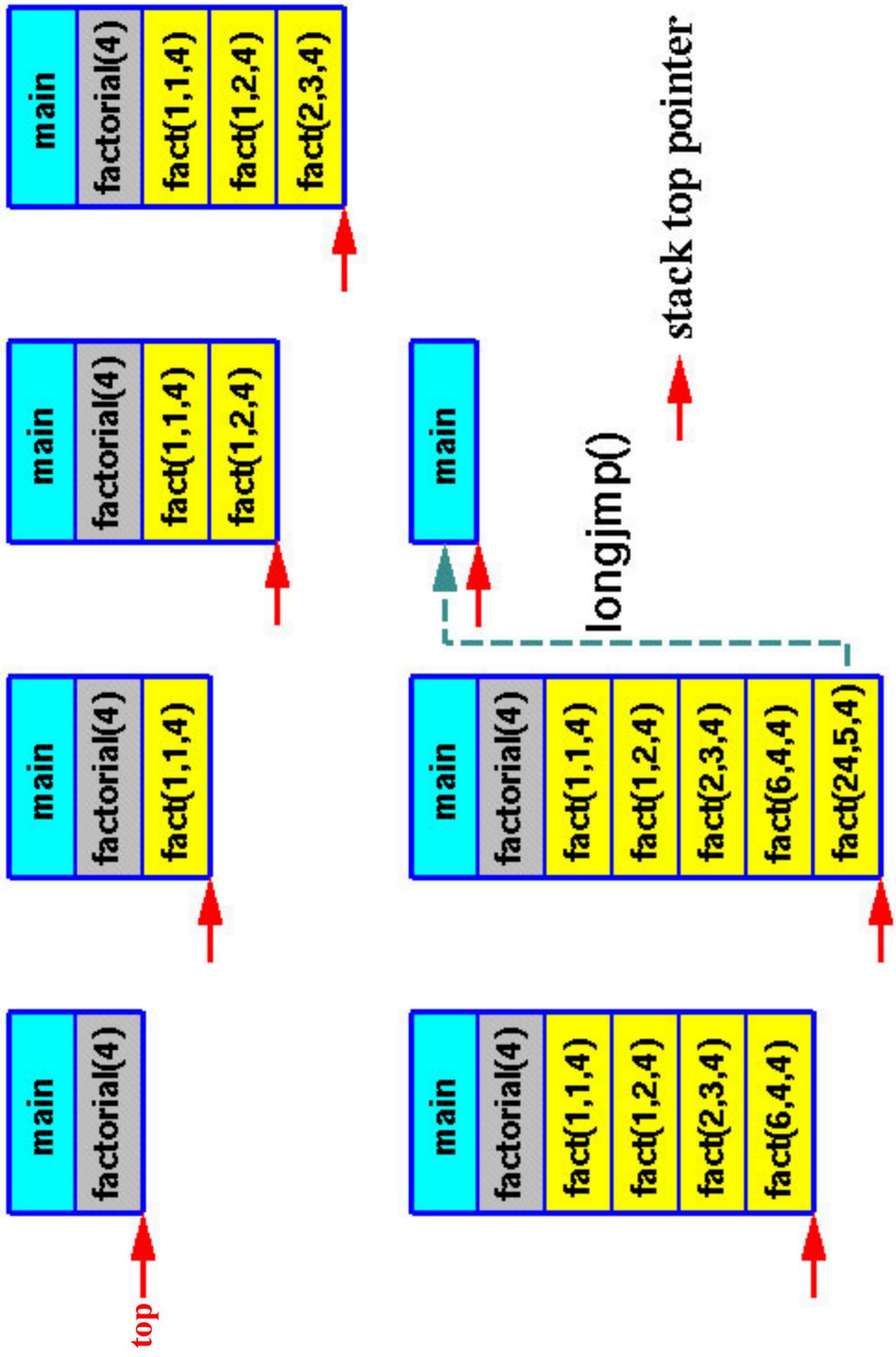
Why not Count++ or ++Count?

Count++: the value of current (rather than the next one) Count is passed  
++Count: We don't know the evaluation order of the argument.

Left to Right: OK

Right to Left: Oops! The value of Count in Result\*Count is wrong.

# Jump Buffer Example: Factorial: 3/3



# Jump Buffer Example: Signals-1

```
#include <stdio.h>
#include <signal.h>
#include <setjmp.h>

jmp_buf JumpBuffer;
void handler(int);

void main(void)
{
 signal(SIGINT, handler);
 while(1) {
 if (setjmp(JumpBuffer)==0) {
 printf("Hit Ctrl-C ...\\n");
 pause();
 }
 }
}
```

The diagram illustrates the control flow between the main function and the signal handler. A blue arrow points from the 'if' condition in the main loop to the 'longjmp' call in the handler. A red arrow points back from the 'longjmp' call to the 'if' condition, indicating that control returns to the point where it was interrupted. A blue box highlights the 'longjmp' call itself.

Without this `longjmp()`, control returns to here!

# Jump Buffer Example: Signals-2

(1/2)

```
#define START 0
#define FROM_CTRL_C 1
#define FROM_ALARM 2
#define ALARM 5

jmp_buf Buf;
INT(int);
ALRM(int);
void
void main(void)
{
 int Return;
 signal(SIGINT, INT);
 signal(SIGALRM, ALRM);
}

while (1) {
 if ((Return=setjmp(Buf))==START) {
 alarm(ALARM);
 pause();
 }
 else if (Return == FROM_CTRL_C) {
 }
 else if (Return == FROM_ALARM) {
 print("Alarm reset to %d sec.\n",
 ALARM);
 alarm(ALARM);
 }
}
}
```

# Jump Buffer Example: Signals-2 (2/2)

```
void INT(int sig)
{
 char c;
 signal(SIGALRM, SIG_IGN);
 signal(SIGINT, SIG_IGN);
 signal(SIGALRM, ALRM);
 printf("Got an alarm\n");
 alarm(0); /* reset alarm */
 signal(SIGALRM, ALRM);
 signal(SIGINT, INT);
 longjmp(Buf FROM_ALARM);
}

void ALRM(int sig)
{
 signal(SIGINT, SIG_IGN);
 signal(SIGALRM, SIG_IGN);
 print("Want to quite?");
 c = getchar();
 if (c=='Y' || c=='Y')
 exit(0);
 signal(SIGINT, INT);
 signal(SIGALRM, ALRM);
 longjmp(Buf, FROM_CTRL_C);
}
```

alarm clock has no effect

# A Strange Use: 1/2

```
#include <stdio.h>
#include <setjmp.h>

int max, iter;
jmp_buf Main, PointA, PointB;
void Ping(void), Pong(void);

void main(int argc, char *argv[])
{
 max = abs(atoi(argv[1]));
 iter = 1;
 if (setjmp(Main) == 0) {
 Ping();
 if (setjmp(Main) == 0)
 Pong();
 longjmp(PointA, 1);
 }
}

Set return point
Main & call Ping()

Set return point
Main & call Pong()
```

# A Strange Use: 2/2

```
void Ping (void)
{
 if (setjmp (PointA)==0)
 longjmp (Main,1);
 while (1) {
 printf ("Ping--");
 if (setjmp (PointA)==0)
 longjmp (PointB,1);
 }
}
void Pong (void)
{
 if (setjmp (PointB)==0)
 longjmp (Main,1);
 while (1) {
 printf ("Pong--");
 if (iter++ > max)
 exit(0);
 if (setjmp (PointB)==0)
 longjmp (PointA,1);
 }
}
```

This program does not work if there are local variables. Why?

Output:

Ping-Pong-Ping-Pong-Pong-.....

# A Not-So-Correct Multithread System: 1/10

- ❑ Before going into more details, we examine a not-so-correct way to build a user-level thread system.
- ❑ First, we need a (simplified) TCB data structure.

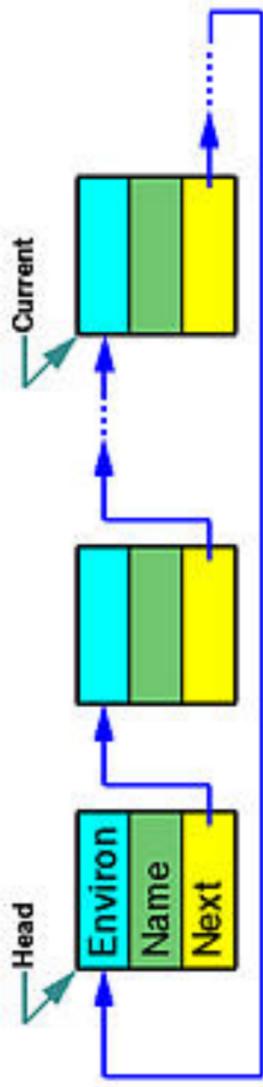
```
typedef struct PCB_NODE *PCB_ptr; /* pointer to a PCB */

/* a PCB:
 * jmp_buf Environment;
 * int Name;
 * PCB_ptr Next;
 */
PCB;
```

# A Not-So-Correct

## Multithread System: 2/10

- ❑ We need two more jump buffers `MAIN` and `SCHEDULER`.
- ❑ The former is used to save the main program's environment, and the latter is for the scheduler.
- ❑ Because the main program and the scheduler are not scheduled by the scheduler, they are not in the PCB list.
- ❑ There are two pointers: `Head` pointing to the head of the PCB list and `Current` to the running thread.



# A Not-So-Correct

## Multithread System: 3/10

- The scheduler is simple.
- Initially the scheduler `Scheduler()` is called by the main program to set an entry in jump buffer `SCHEDULER` and jump back to the main program using jump buffer `MAIN` that was setup *before* the call to `Scheduler`.
- After this, we use a long jump to `SCHEDULER` rather than via function call.

```
void Scheduler(void)
{
 if (setjmp(SCHEDULER) == 0)
 longjmp(MAIN, 1);
 Current = Current->Next;
 longjmp(Current->Environment, 1); /* jump to its environ */
}
```

# A Not-So-Correct

## Multithread System: 4/10

- ❑ `THREAD_YIELD()` is very simple.
- ❑ Release CPU voluntarily.
- ❑ What we need is saving the current environment to this thread's environment (actually a jump buffer) and transferring the control to the scheduler via a `longjmp`.
- ❑ Because this is so simple, we use `#define`

```
#define THREAD_YIELD(name) {
 if (setjmp(Current->Environment) == 0) {
 longjmp(SCHEDULER, 1);
 }
}
```

# A Not-So-Correct

## Multithread System: 5/10

- ☐ `THREAD_INIT()` can be part of `THREAD_CREATE()`.
- ☐ We create and initialize a PCB, set its return point, and long jump back to the main program.

```
#define THREAD_INIT(name) {
 work = (PCB_ptr) malloc(sizeof(PCB));
 work->Name = name;
 if (Head == NULL)
 Head = work;
 else
 Current->Next = work;
 work->Next = Head;
 Current = work;
 if (setjmp(work->Environment) == 0)
 longjmp(MAIN, 1);
}
```

# A Not-So-Correct Multithread System: 6/10

- ☐ `THREAD_CREATE()` is simple.
- ☐ We just set the return point of `MAIN` and call the function.

```
#define THREAD_CREATE(function, name) { \
 if (setjmp(MAIN) == 0) \
 (function)(name); \
}
```

```
void main(void)
{
 Head = Current = NULL; /* initialize pointers */
 THREAD_CREATE(funct_1, 1); /* initialize threads */
 THREAD_CREATE(funct_2, 2);
 THREAD_CREATE(funct_3, 3);
 THREAD_CREATE(funct_4, 4);
 if (setjmp(MAIN) == 0) /* initialize scheduler */
 Scheduler();
 longjmp(SCHEDULER, 1); /* start scheduler */
}
```

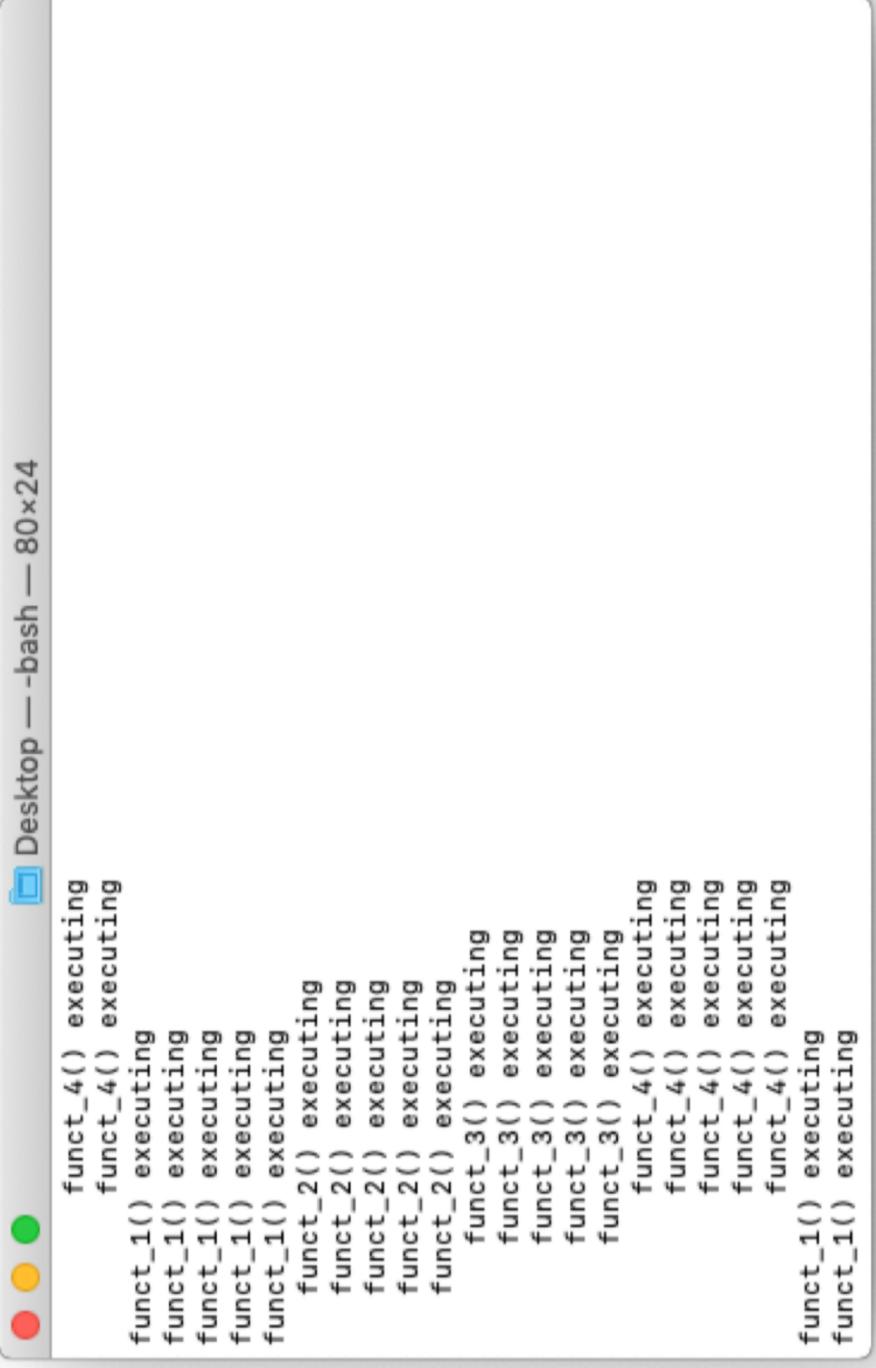
# A Not-So-Correct Multithread System: 7/10

- Each function to be run as a thread must call `THREAD_INIT()`.

```
void funct_1(int name)
{
 int i;
 THREAD_INIT(name); /* initialize as thread */
 while (1) { /* running the thread */
 for (i = 1; i <= MAX_COUNT; i++)
 printf("funct_1() executing\n");
 THREAD_YIELD(name); /* yield control */
 }
}
```

# A Not-So-Correct Multithread System: 8/10

- ❑ This implementation appears to be correct. The following is a screenshot.



The screenshot shows a terminal window titled "Desktop — -bash — 80x24". The window contains the following text:

```
funct_4() executing
funct_4() executing
funct_1() executing
funct_1() executing
funct_1() executing
funct_1() executing
funct_1() executing
funct_1() executing
funct_2() executing
funct_2() executing
funct_2() executing
funct_2() executing
funct_2() executing
funct_2() executing
funct_3() executing
funct_3() executing
funct_3() executing
funct_3() executing
funct_3() executing
funct_3() executing
funct_4() executing
funct_4() executing
funct_4() executing
funct_4() executing
funct_4() executing
funct_1() executing
funct_1() executing
```

# A Not-So-Correct

## Multithread System: 9/10

- ❑ It is **not!** **Why?** But you have all the ideas!
- ❑ We do not use many local variables, in fact only one variable `i`. In a function, say `funct_1()`, `i` is used before `THREAD_YIELD()`.
- ❑ Once `THREAD_YIELD()` is called, the stack frame of `funct_1()` becomes invalid. However, this is fine, because after returning from `THREAD_YIELD()` this variable is reinitialized.
- ❑ Is the jump buffer Environment of each thread correct? In general it is not. However, the PC is correct because it is not stored there.

# A Not-So-Correct Multithread System: 10/10

- ❑ The key issue making this system not-so-correct is that each thread does not have its stack frame.
- ❑ As a result, once it long jumps out of its environment the stack frame allocated by the system becomes invalid.
- ❑ The solution is simple: allocating a separate stack frame for each “created” thread so that it won’t go away.
- ❑ This is what we intend to do.

# Let Us Solve the Problem: 1/10

- We need a better TCB. Env is a redefined jmpbuf.

```
typedef struct TCB_NODE *TCB_ptr;
typedef TCB_ptr THREAD_t;

typedef struct TCB_NODE { /* thread control block
 int Name; /* thread ID
 int Status; /* thread state
 Env Environment; /* processor context area
 void *StackBottom; /* bottom of stack attached
 int Size; /* stack size
 void (*Entry) (int, char**); /* entry point (function)
 int Argc; /* # of arguments
 char **Argv; /* argument list
 Queue JoinList; /* joining list of threads
} TCB;
```

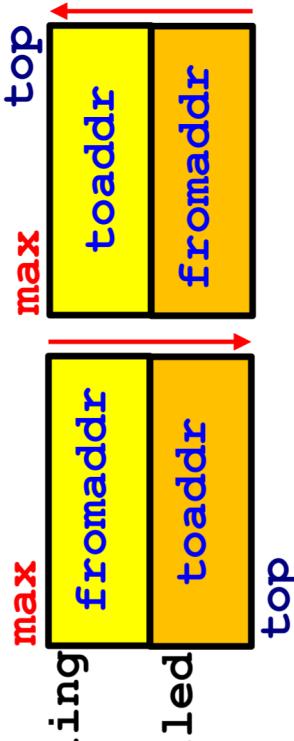
# Initialize the Coroutines: 2/10

- ❑ Initialize the coroutines structure. Refer to slides used in CS3331 Concurrent Computing for the concept of coroutines.

```
static int THREAD_SYS_INIT (void)
{
 int *stack;
 Running = (THREAD_t) malloc(sizeof(TCB)); /* dummy running thread */
 stack = (int *) malloc(64); /* 64 bytes for stack */
 Running->Name = mtu_MAIN;
 Running->StackBottom = stack;
 ReadyQ_Head = ReadyQ_Tail = NULL;
 SuspendQ_Head = SuspendQ_Tail = NULL;
 SysTable_Head = SysTable_Tail = NULL;
 SYSTEM_INITIALIZE = TRUE;
 return mtu_NORMAL;
}
```

# Stack Growing Direction: 3/10

- ❑ Because the user stack can grow up (stack at the highest address) or grow down (stack at the end of code and data sections), we need to know which way it goes.
- ❑ **fromaddr** is a variable in the calling function, and **toaddr** is a variable in the called function.



```
static int growsdown (void *fromaddr)
{
 int toaddr;

 return fromaddr > (void *) &toaddr;
}
```

# Wrap the Created Thread: 4/10

- ❑ `THREAD_WRAP()` wraps up the created thread and runs it as a function. The function to be run is indicated by `Running`.

```
static volatile void THREAD_WRAP (void)
{
 (*Running->Entry) (Running->Argc, Running->Argv) ; /* run thread */
 THREAD_EXIT() ;
 /* if user did not call
 * THREAD_EXIT, do it here */
}
```

# THREAD\_INIT: System Dependent

- ❑ Initialize a new thread's environment.
- ❑ Newer version of gcc may not allow you to modify the jump buffer.

```
void THREAD_INIT (volatile TCB *volatile NewThread, void *StackPointer)
{
 /* In Linux 1.0 the code maybe like following three lines
 * NewThread->Environment->__sp = StackPointer;
 * NewThread->Environment->__bp = StackPointer;
 * NewThread->Environment->__pc = (void *)THREAD_WRAP;
 * Here is THREAD_INIT for Linux 2.0
 */
 NewThread->Environment[0].__jmpbuf[JB_SP] = (int)StackPointer;
 NewThread->Environment[0].__jmpbuf[JB_BP] = (int)StackPointer;
 NewThread->Environment[0].__jmpbuf[JB_PC] = (int)THREAD_WRAP;
}
```

# THREAD\_CREATE() : 1/2

- ❑ THREAD\_CREATE() allocates a TCB and a stack for the thread being created and initialize the TCB.

```
THREAD_t THREAD_CREATE (void (*Entry) (), int Size, int Flag,
int Argc, char **Argv)
{
 THREAD_t NewThread;
 int *StackBottom, FromAddr;
 void *StackPointer;

 NewThread = (THREAD_t) malloc (sizeof(TCB)); /* new thread TCB */
 if (NewThread == NULL)
 return (THREAD_t) mtu_ERROR;
 Size += sizeof(StackAlign); /* get new stack size
StackBottom = (int *) malloc (Size);
StackPointer = (void *) (growsdown (&FromAddr) ?
 (Size+(int) StackBottom)-sizeof(StackAlign) : (int) StackBottom);
THREAD_INIT (NewThread, StackPointer); /* initialize thread
 32*/ /* architecture-dependent! */
```

[-- Next Page --](#)

# THREAD\_CREATE () : 2/2

- ❑ Continue from previous page.

```
/* from previous page */
NewThread->Name = NextThreadName++; /* initial TCB values */
NewThread->Status = mtu_READY;
NewThread->Entry = (void (*) (int, char**)) Entry;
NewThread->Argc = Argc; NewThread->Argv = Argv;
NewThread->StackBottom = StackBottom;
NewThread->Size = Size;
NewThread->JoinList.Head = NULL; NewThread->JoinList.Tail = NULL;
THREAD_READY (NewThread); /* thread into Ready Q */
THREAD_READY (Running);
if (Flag == THREAD_SUSPENDED)
 THREAD_SUSPEND (NewThread);
 THREAD_SCHEDULER ();
return NewThread;
}
```

# THREAD\_EXIT() : 1/2

□ Continue from previous page.

```
int THREAD_EXIT (void)
{
 THREAD_t temp;

 if (Running->Name == mtu_MAIN) { /* if main, exit there
 /* have no thread remain
 /* in the ready Q */
 if (ReadyQ.Head != NULL)
 return mtu_ERROR;
 if (SuspendQ.Head != NULL) /* and in suspend queue
 /* and in suspend queue */
 return mtu_ERROR;
 return mtu_NORMAL;
 }

 while (Running->JoinList.Head != NULL) { /* check for joining
 /* temp = (THREAD_t) THREAD_Remove (&(Running->JoinList));
 /* temp->Status = mtu_READY;
 /* THREAD_Append (&ReadyQ, (void *) temp);
 /* continue to next page */
 }
}
```

# THREAD\_EXIT() : 2/2

- Continue from previous page.

```
Running->Name = mtu_INVALID; /* set current thread's TCB */
Running->Status = mtu_TERMINATED; /* and status = terminated */
Running = NULL;
THREAD_SCHEDULER();
return mtu_ERROR;
}
```

## THREAD\_YIELD()

- ❑ THREAD\_YIELD() puts the running thread back to READY and calls THREAD\_SCHEDULE() to reschedule.

```
void THREAD_YIELD (void)
{
 THREAD_READY(Running) ; /* put the running one to Ready */
 THREAD_SCHEDULE() ; /* ask scheduler to reschedule. */
}
```

# THREAD\_SCHEDULE () : 1/2

- ❑ THREAD\_SCHEDULE () finds and runs the next ready thread.

```
static int THREAD_SCHEDULE (void)
{
 THREAD_t volatile Nextp;
 THREAD_t (THREAD_t) THREAD_Remove (&ReadyQ); /* find a thread */
 if (Nextp == NULL) {
 mtuMP_errno=mtuMP_DEADLOCK; /* if ready queue is empty */
 ShowDeadlock ();
 exit (0);
 return mtu_ERROR;
 }
 if (Running==NULL) {
 Running = Nextp;
 Nextp->Status = mtu_RUNNING;
 RestoreEnvironment (Running->Environment); /* restore env */
 }
 /* continue to next page */
}
```

# THREAD\_SCHEDULE() : 2/2

- ❑ THREAD\_SCHEDULE() finds and runs the next ready thread.

```
/* continue from previous page */
if ((Running != NextTp) &&
(SaveEnvironment(Running->Environment) == 0)) {
 /* else save running's env */
 /* let next thread run */
 NextTp->Status = mtu_RUNNING;
 RestoreEnvironment(Running->Environment); /* restore env */
}
return mtu_NORMAL;
```

# Dining Philosophers: 1/2

```
void Philosopher(int No)
{
 int Left = No;
 int Right = (No + 1) % PHILOSOPHERS;
 int RandomTimes, i, j;
 char spaces[PHILOSOPHERS*2+1];

 for (i = 0; i < 2*No; i++) /* build leading spaces */
 spaces[i] = ' ';
 spaces[i] = '\0';

 printf("%sPhilosopher %d starts\n", spaces, No);
 for (i = 0; i < Iteration; i++) {
 printf("%sPhilosopher %d is thinking\n", spaces, No);
 SimulatedDelay();
 SEMAPHORE_WAIT(Seats);
 printf("%sPhilosopher %d has a seat\n", spaces, No);
 MUTEX_LOCK(Chopstick[Left]);
 MUTEX_LOCK(Chopstick[Right]);
 printf("%sPhilosopher %d gets chopsticks and eats\n", spaces, No);
 SimulatedDelay();
 printf("%sPhilosopher %d finishes eating\n", spaces, No);
 MUTEX_UNLOCK(Chopstick[Left]);
 MUTEX_UNLOCK(Chopstick[Right]);
 SEMAPHORE_SIGNAL(Seats);
 }
 THREAD_EXIT();
}
```

# Dining Philosophers: 2/2

```
int main(int argc, char *argv[])
{
 THREAD_t Philosophers[PHILOSOPHERS];
 int SeatNo[PHILOSOPHERS];
 int i;

 Iteration = abs(atoi(argv[1]));
 srand((unsigned int)time(NULL));
 /* initialize random number */
 for (i = 0; i < PHILOSOPHERS; i++) /* create mutex locks
 Chopstick[i] = MUTEX_INIT();
 Seats = SEMAPHORE_INIT(PHILOSOPHERS-1); /* seat semaphore */
 for (i = 0; i < PHILOSOPHERS; i++) { /* create philosophers */
 /* philosopher number */
 /* create a thread */
 /* the thread function */
 /* stack size */
 /* the thread flag */
 /* play a trick here */
 /* no argument list */
 if (Philosophers[i] == (THREAD_t)mtu_ERROR) { /* if failed */
 printf("Thread creation failed.\n");
 }
 }
 for (i=0; i<PHILOSOPHERS; i++) /* wait until all done */
 THREAD_JOIN(Philosophers[i]);
}

return 0;
}
```

# Conclusions

- ❑ In a kernel, the kernel has to allocate a stack differently.
- ❑ Context switch has to be done differently and directly rather than using a jump buffers.
- ❑ The remaining should be very similar and could be copied easily.
- ❑ The not-so-correct system is discussed here:  
<http://www.cs1.mtu.edu/cs4411.ck/www/NOTES/non-local-goto/index.html>
- ❑ A simple user-level thread system is in the common directory mtuthread.tar.gz. Also refer to this page:  
<http://www.cs1.mtu.edu/cs4411.ck/www/PROG/PJ/proj.html>

The End