

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

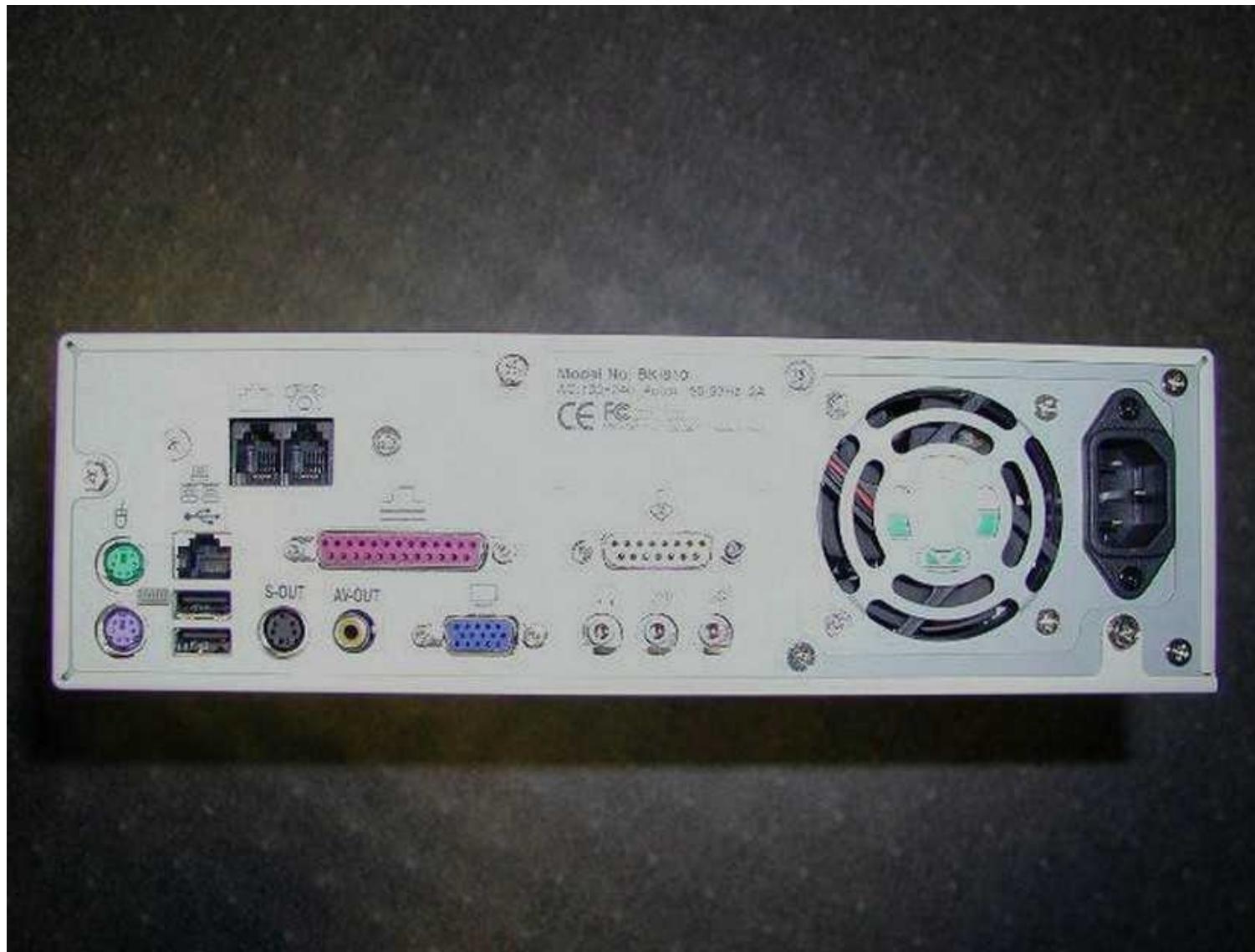
(C) Abhijit A.M.
Available under Creative Commons Attribution-ShareAlike License
V3.0+

Credits: Slides of “OS Book” ed10.

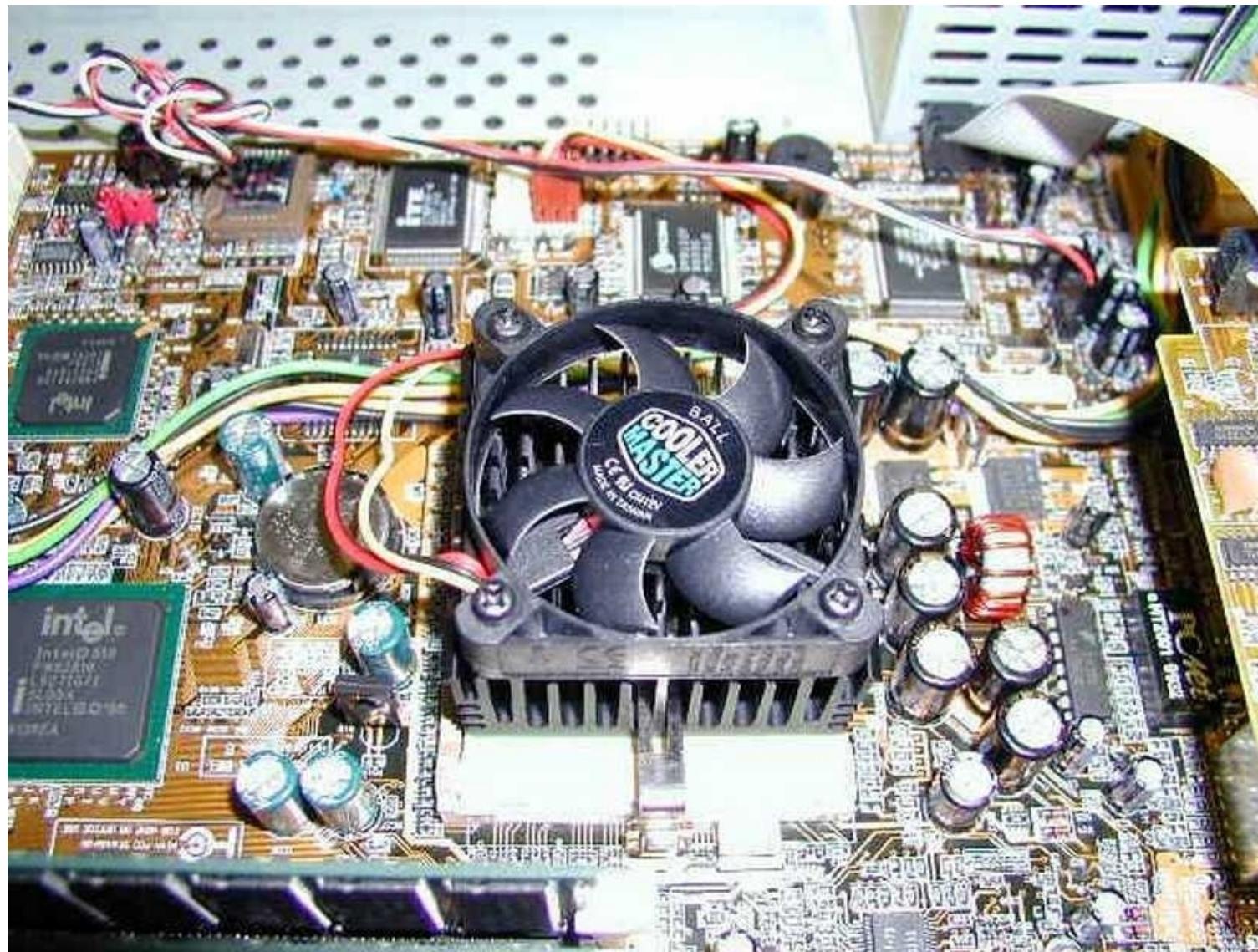
Initial lectures

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”

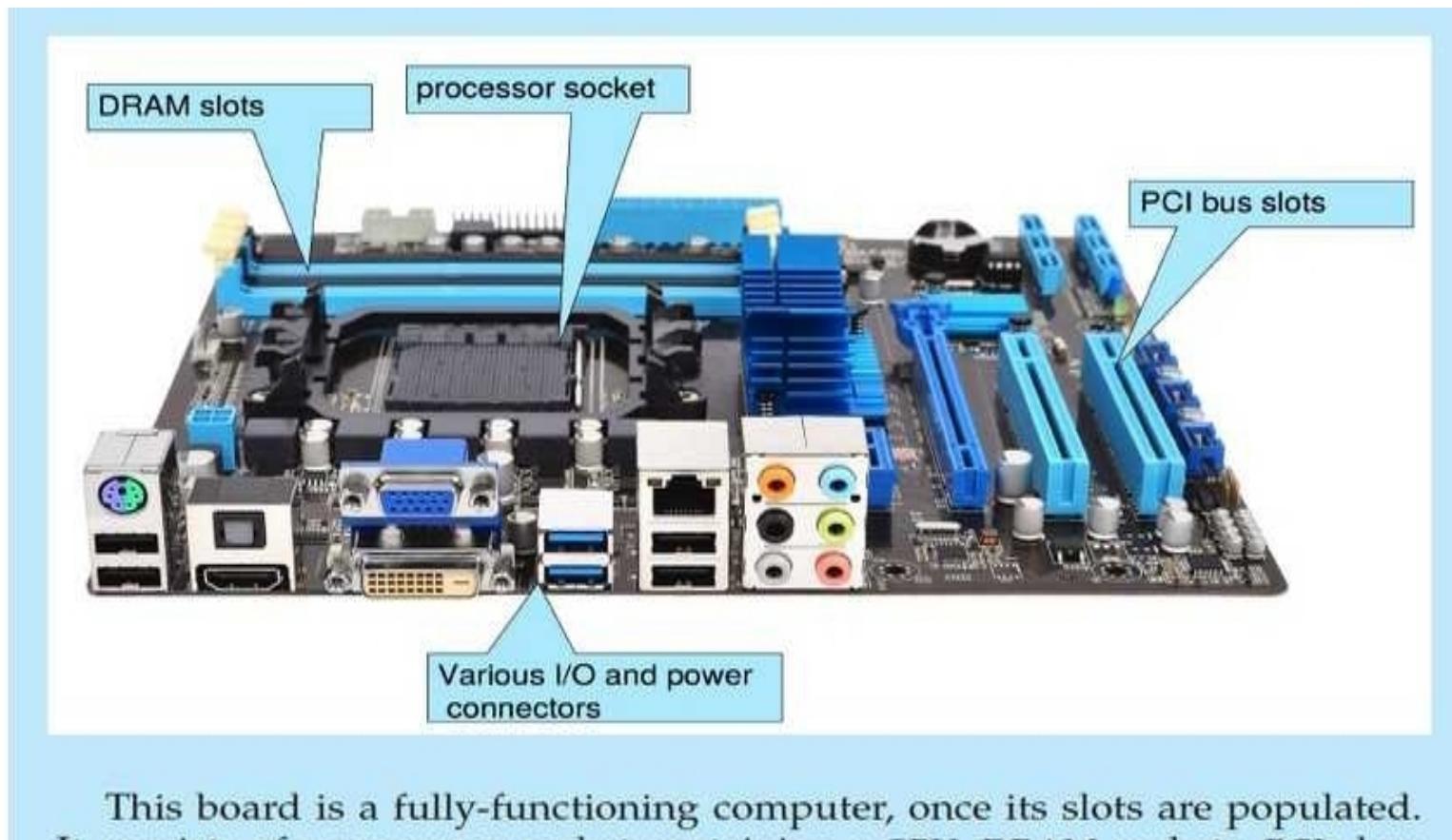
Runs “machine instructions”

The actual “computer”

Questions:

Where are the instructions that the processor runs?

What's on the motherboard?

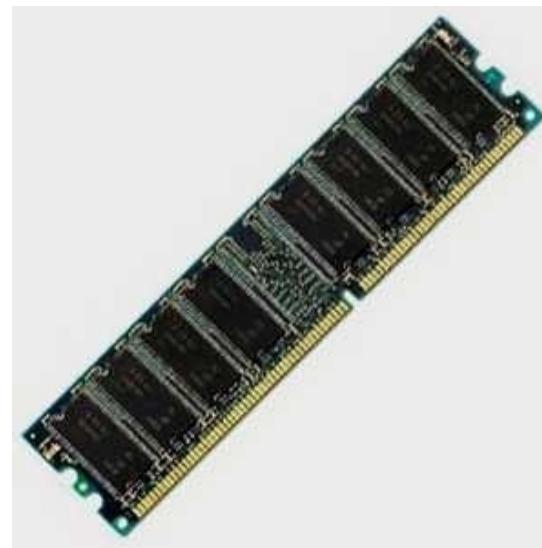


Memory

- Random Access Memory (RAM)

Same time of access to any location – randomly accessible

Semiconductor device

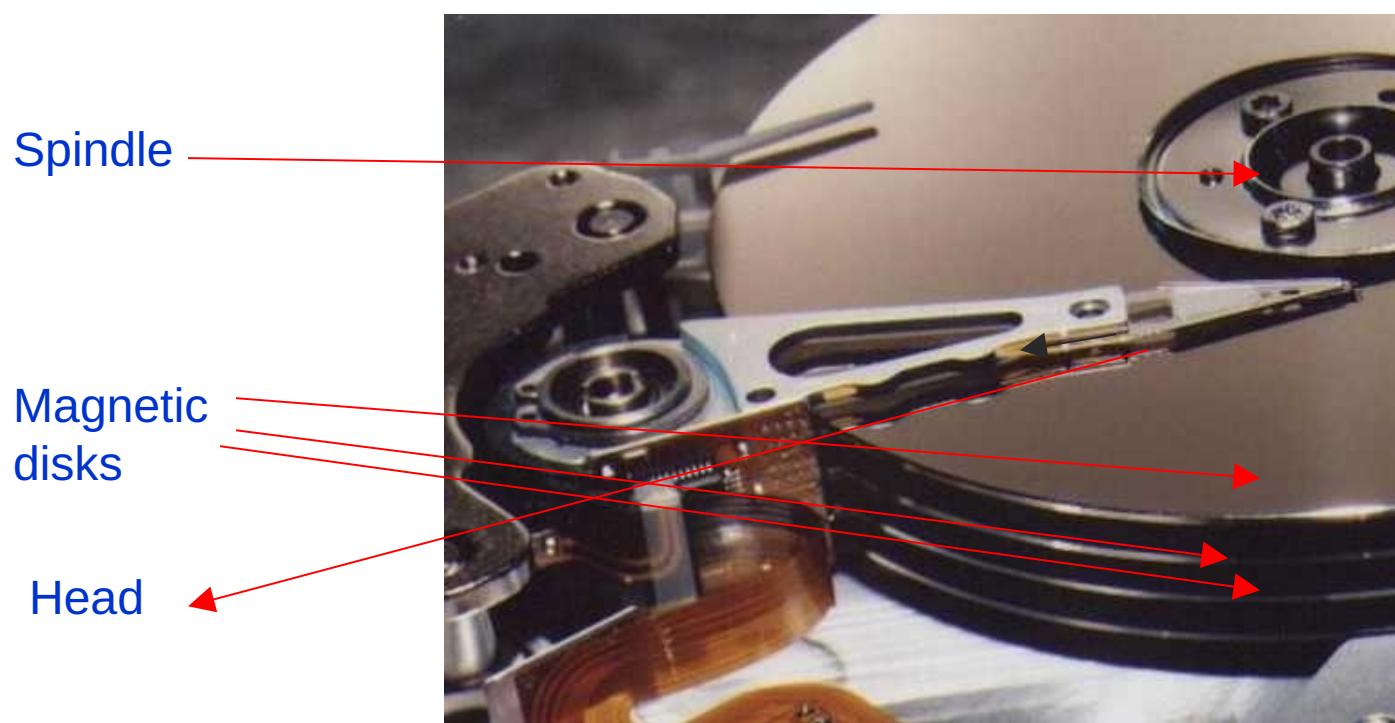


Question:
Can we add more RAM to a computer?

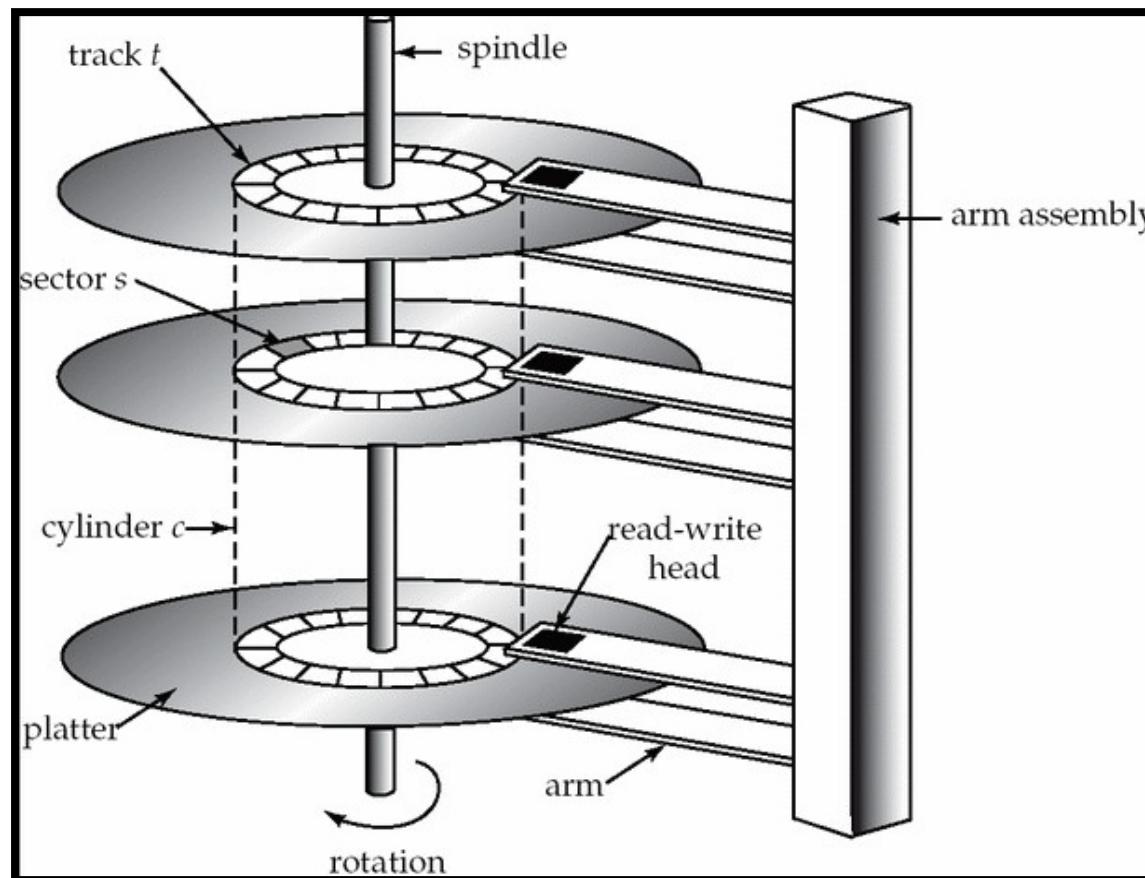
The Hard Drive



The Hard Drive



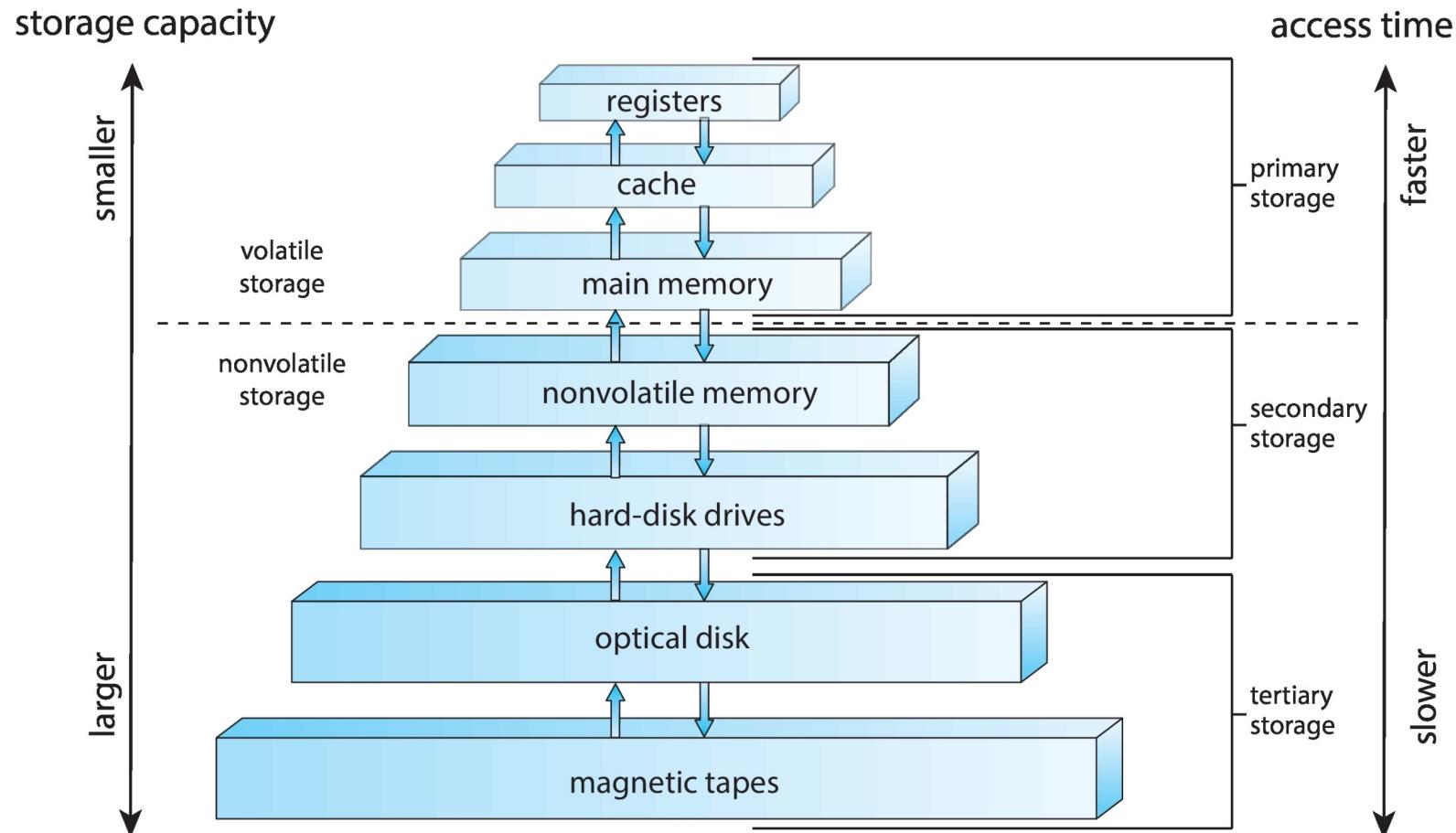
The hard Drive



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

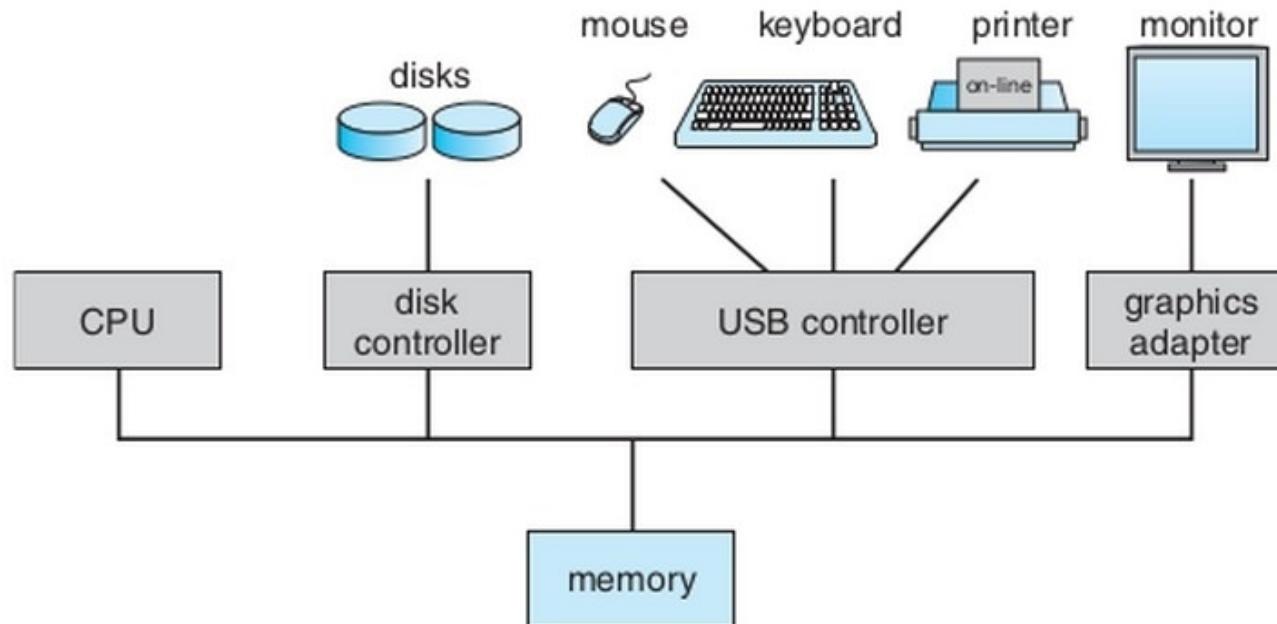


Figure 1.2 A modern computer system.

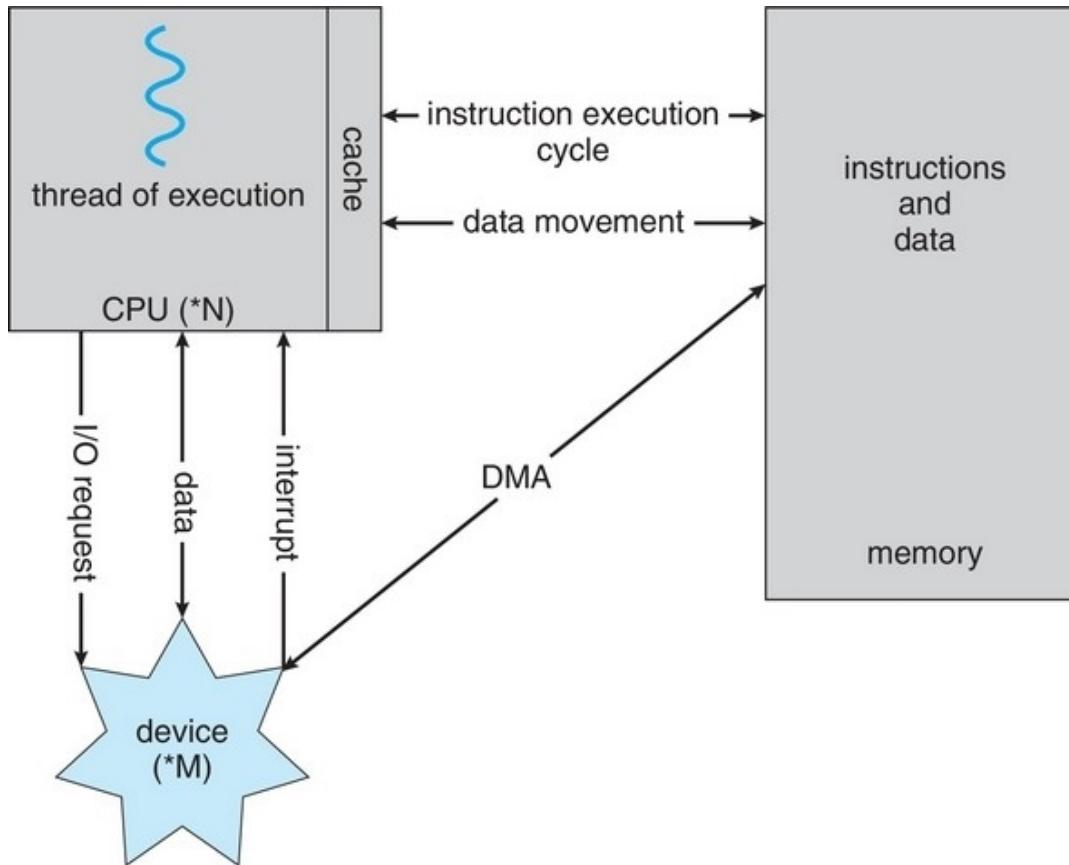
Important Facts

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

- 1)Decode the instruction and execute it

While doing this may fetch some data from the RAM

- 1)While executing the instruction change/update the Program Counter
- 2)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

E.g. “Grub”

Its job is to locate the code of an OS kernel

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 heavily on the kernel and processor

E.g. Compiler, linker, loader, etc.

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

Components of a computer system

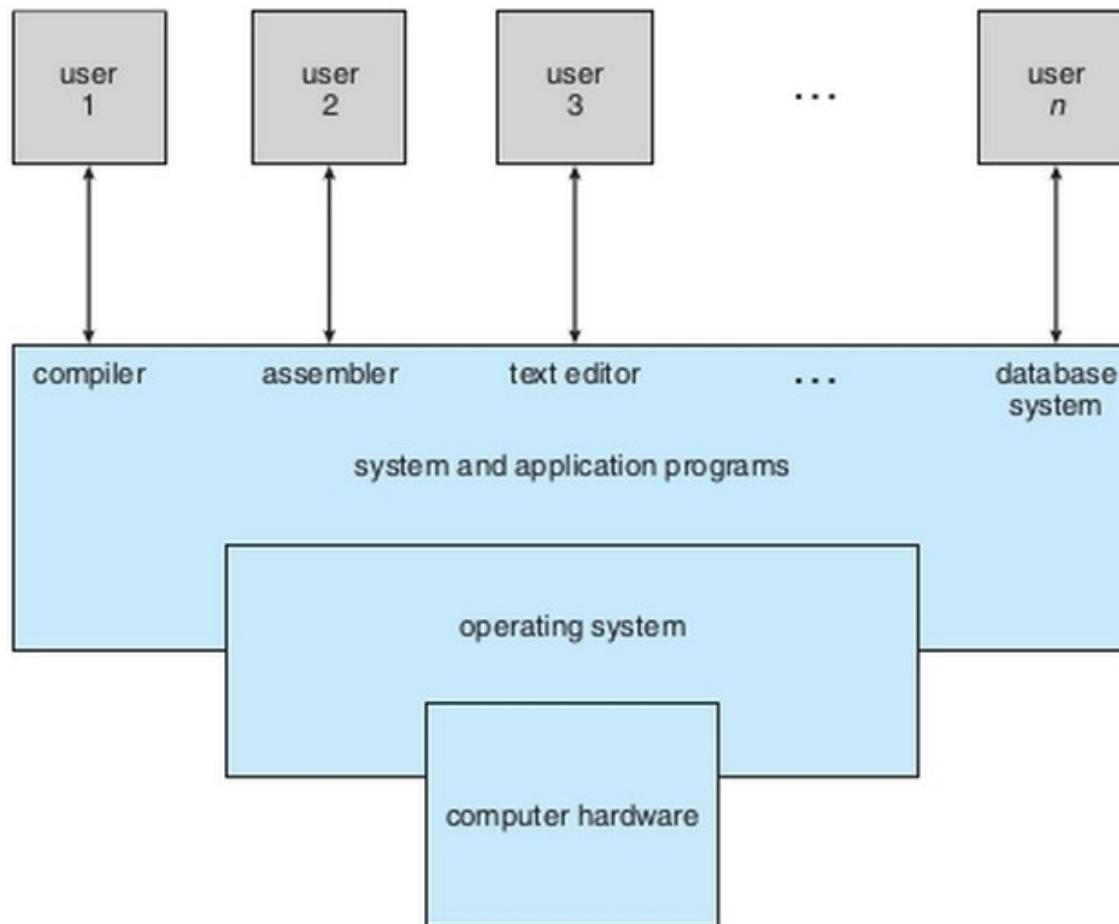


Figure 1.1 Abstract view of the components of a computer system.

Multiprocessor system: SMP

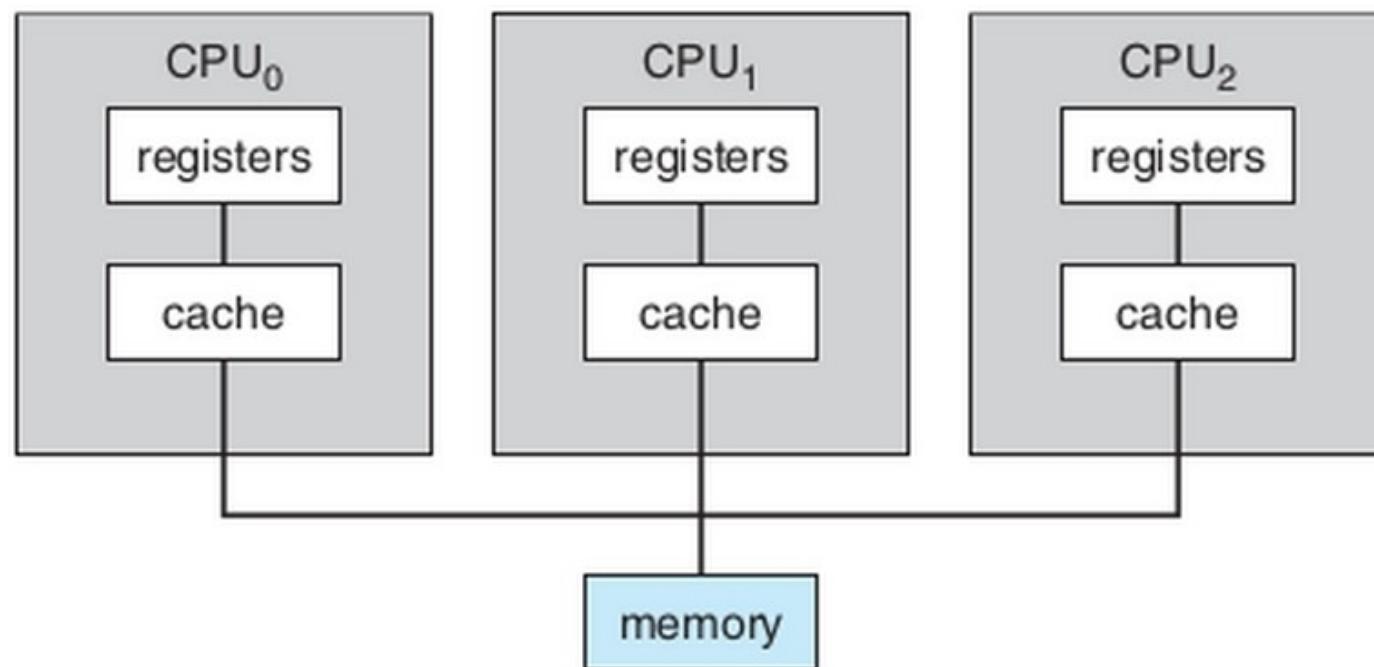


Figure 1.6 Symmetric multiprocessing architecture.

Dual Core: what's that?

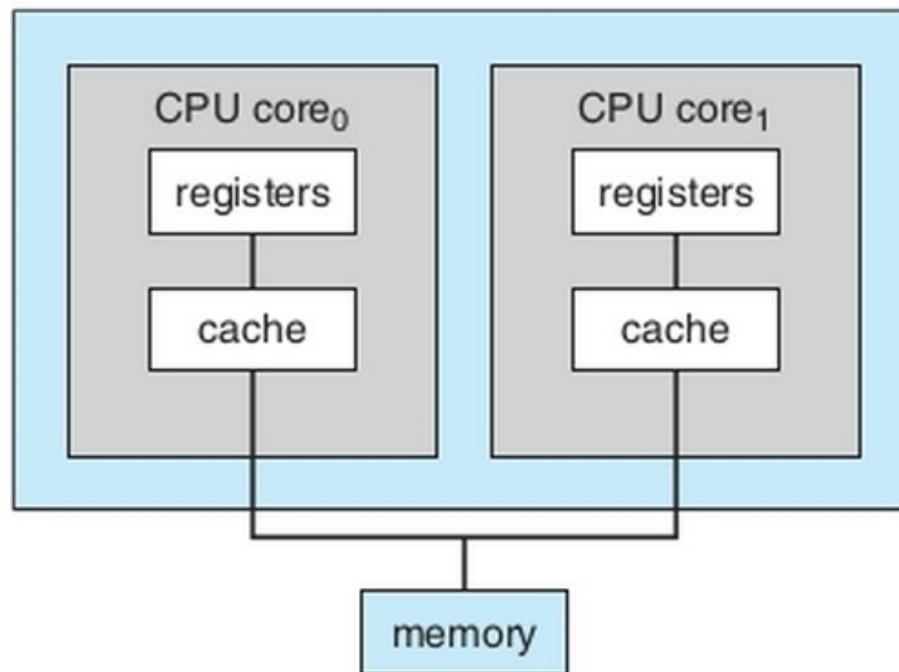


Figure 1.7 A dual-core design with two cores placed on the same chip.

The very important question:

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?

OS! How's that ?

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

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 interrupt, OS code will run!

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

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

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”, recoverable

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 instruction when the mode bit is in “User 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

**Transition from user mode to kernel mode
and vice-versa**

Special instruction called “software interrupt” instructions

E.g. INT instruction on x86

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?

Obviously, OS code. OS occupied these locations in
Memory, at Boot time.

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 !

A proper argument to INT 0x80 specified in a proper register indicates different possible request

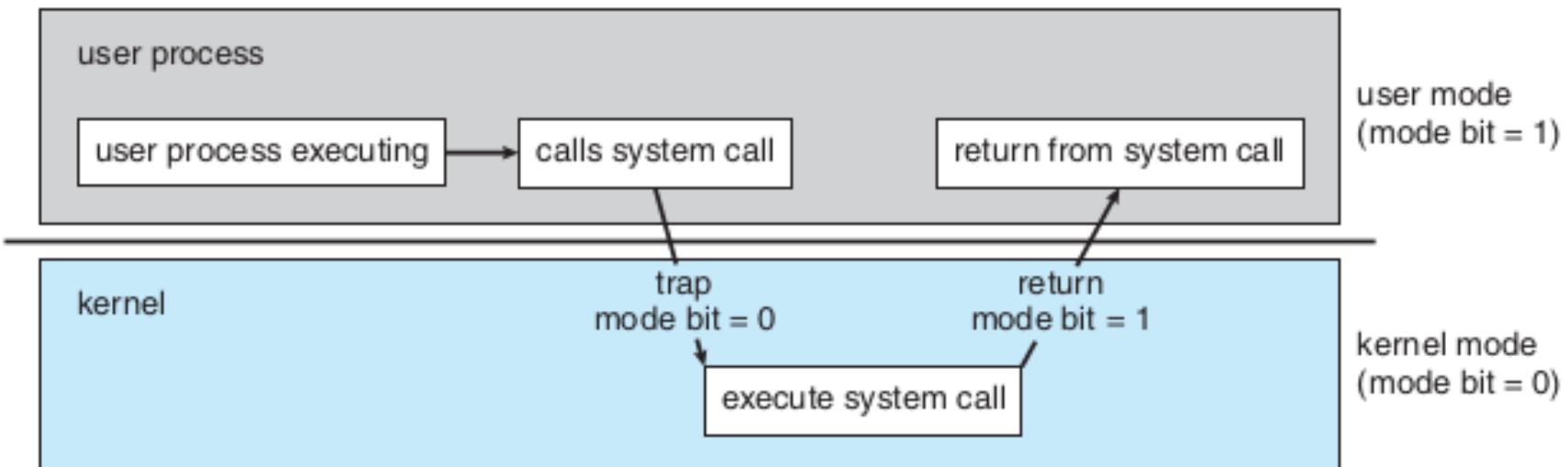


Figure 1.10 Transition from user to kernel mode.

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)

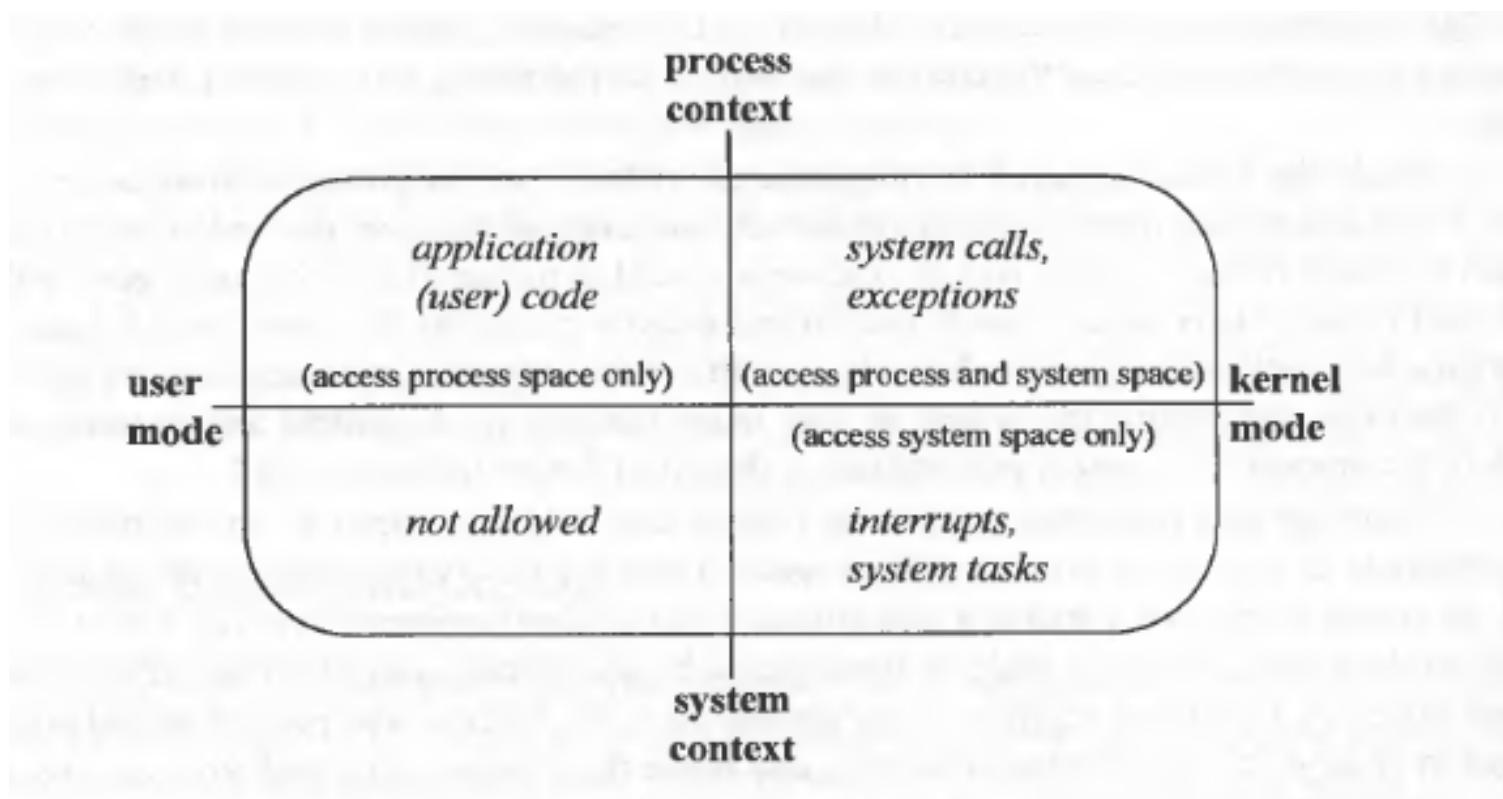
OS sets timer interrupt and “passes control” over to some application code. Now only application code running on CPU !

When time allocated to process is over, the timer interrupt occurs and control jumps back to OS (hardware interrupt mechanism)

The OS code that runs here (the ISR) is called “scheduler”

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.

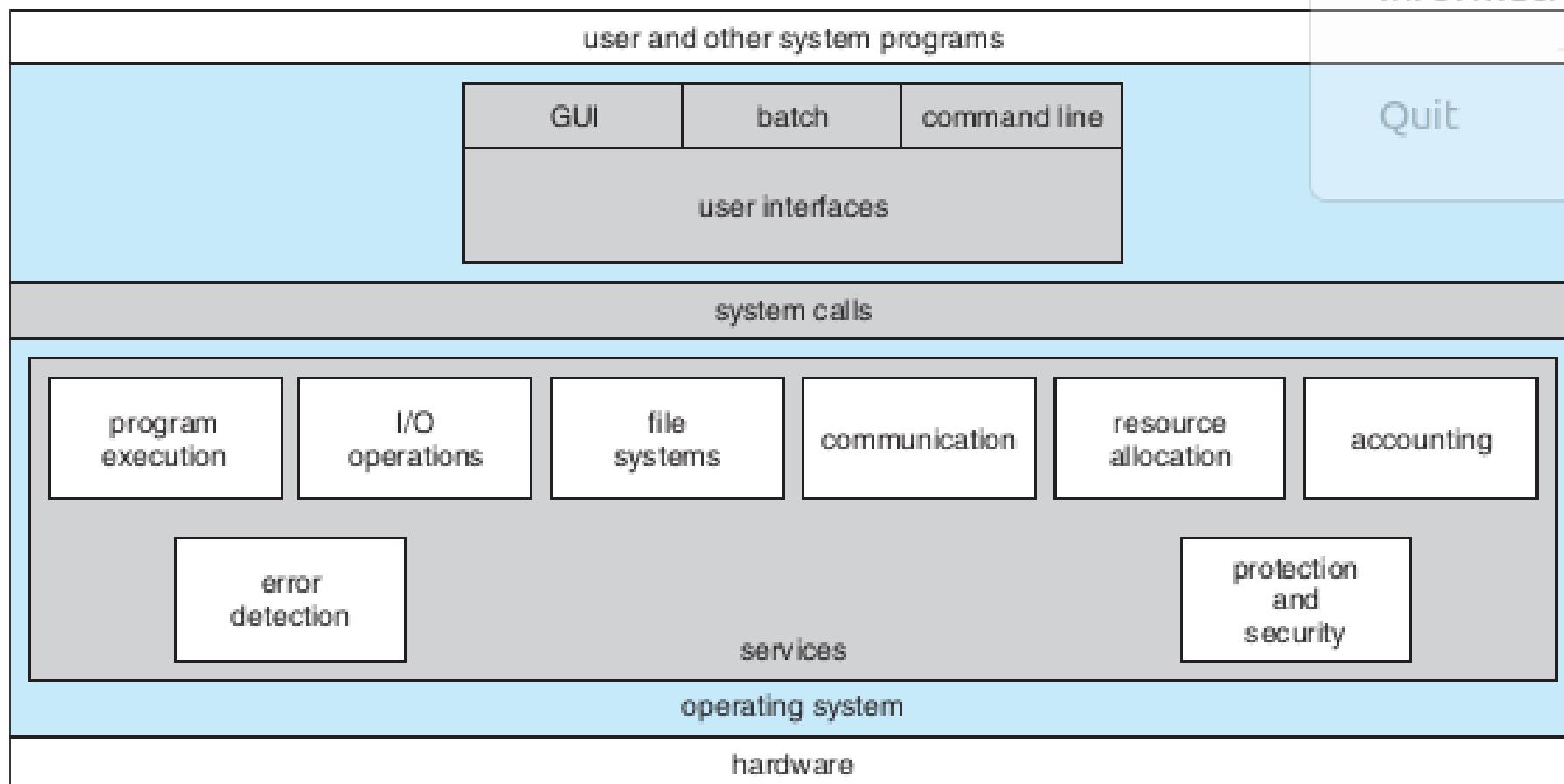


Figure 2.1 A view of operating system services.

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

E.g. open(), read(), write(), fork(), mmap(), etc.

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 maintenance – time,date

https://linuxhint.com/list_of_linux_syscalls/

Communication between processes (IPC)

Read man syscalls

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, ...);  
}  
int write(int fd, char *,  
        int len){
```

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
 operation,

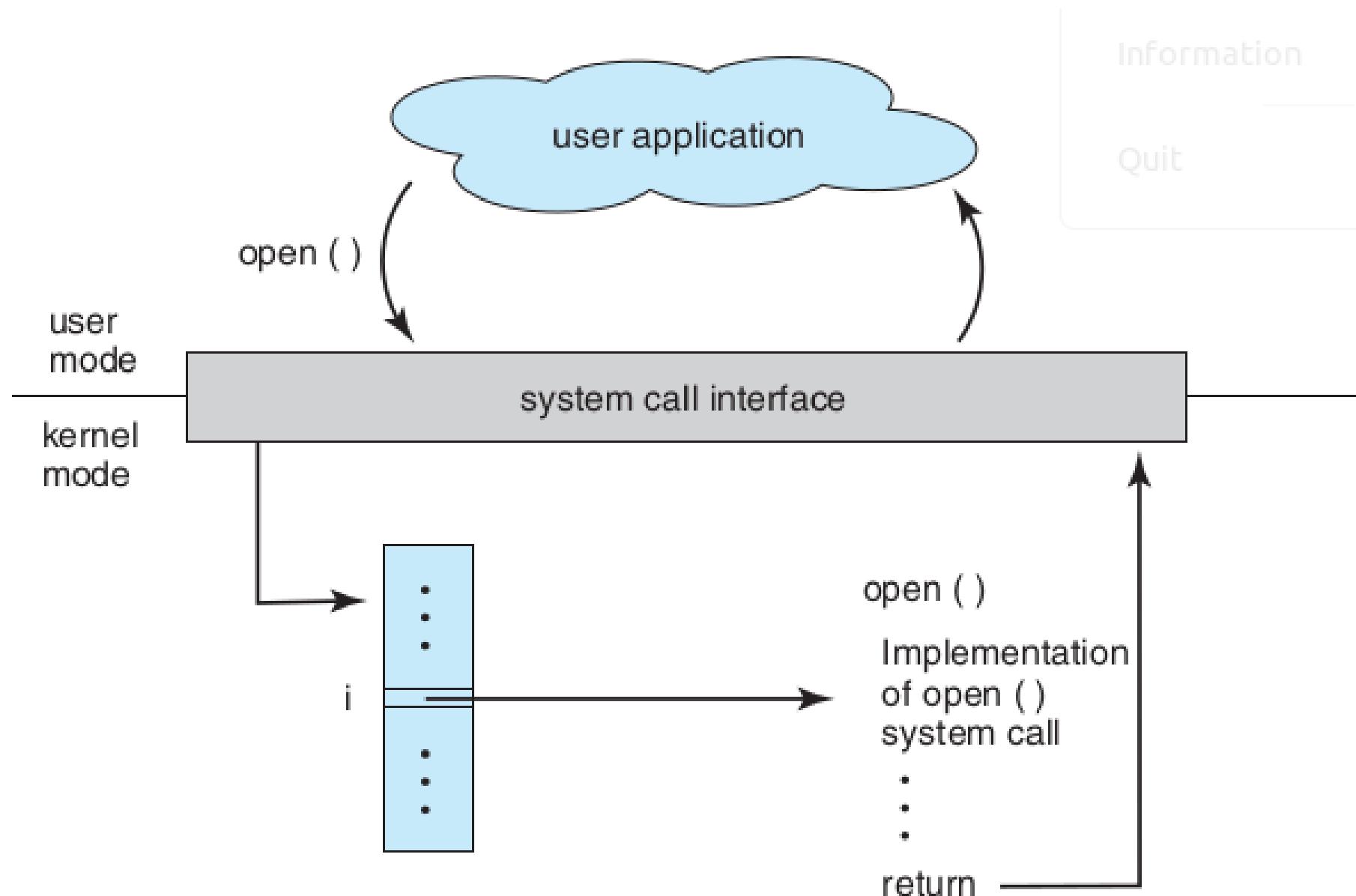


Figure 2.6 The handling of a user application invoking the `open()` system call.

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

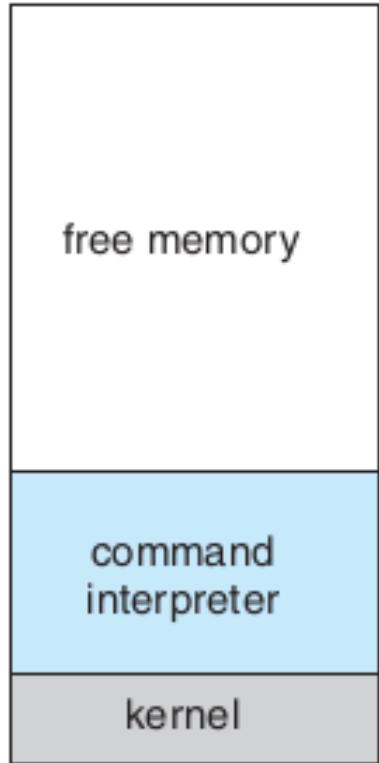
Global variables (data)

Stack (stores local variables of functions)

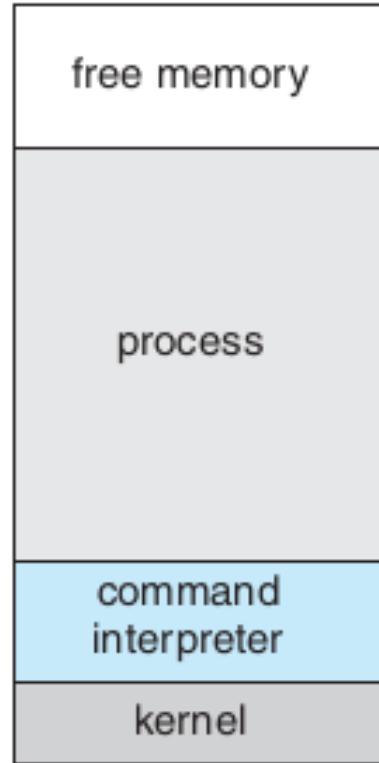
Heap (stores malloced memory)

Shared libraries (e.g. code of printf, etc)

Few other things, may be



(a)

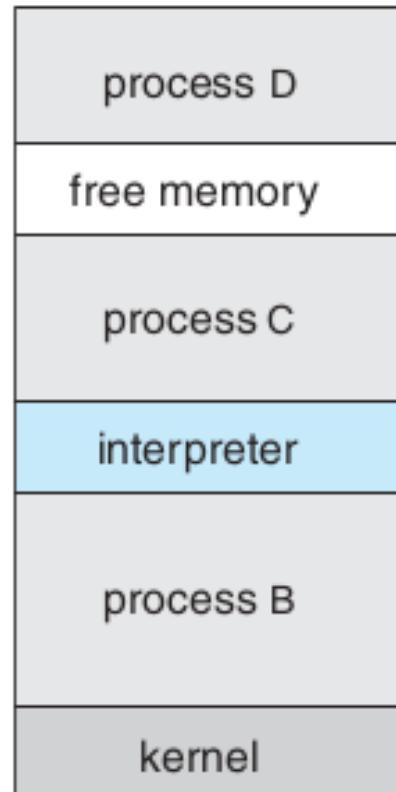


(b)

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 an advanced version of the shell

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

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

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

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

Input is a file and output is also a file.

Other examples: g++ (for C++), javac (for java)



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

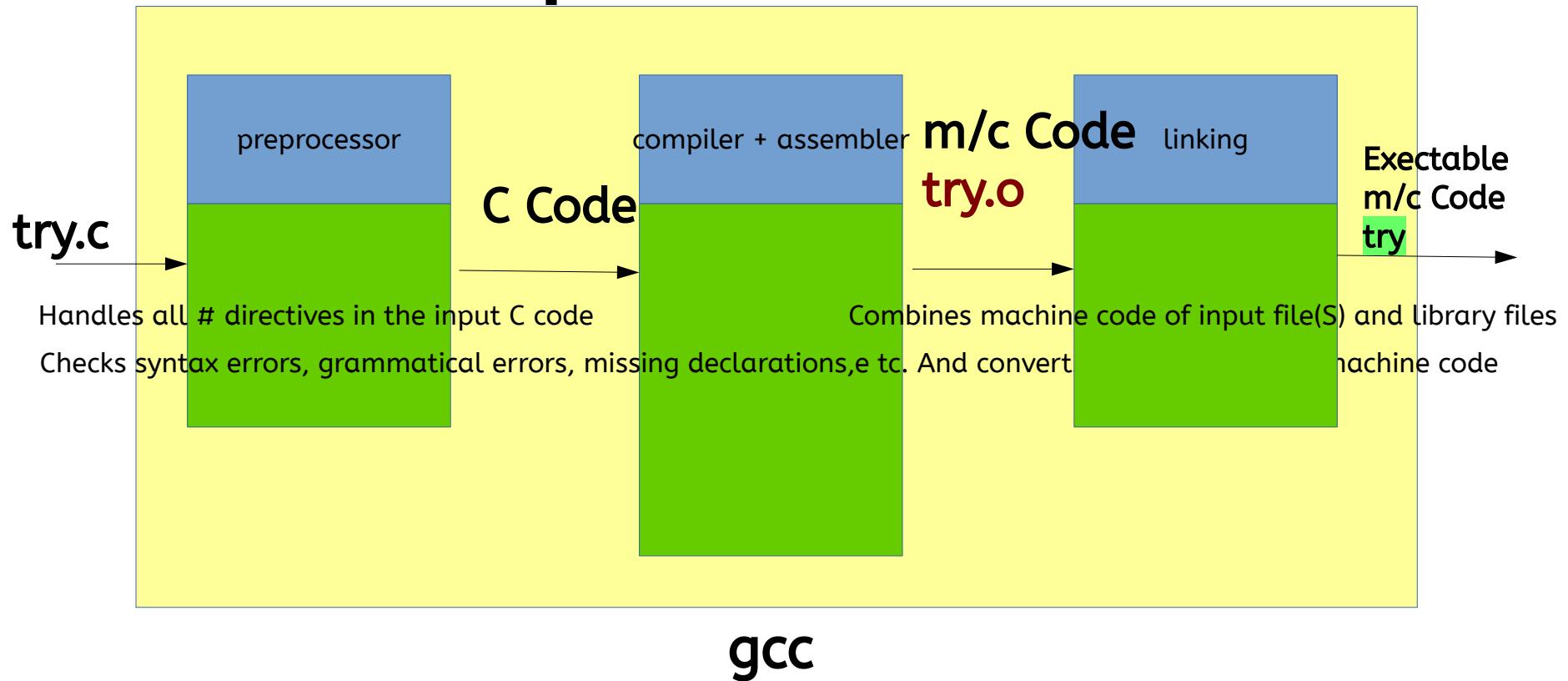
add 10, r1

mov r1, 500

Usage. eg..

\$ as something.s -o something

Compilation Process



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 u

```
$ 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 stages

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 a 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 formats

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 inseparable.

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

That's why the phrase "system programs"

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

Compiler needs to map the features of C into processor's features, and then generate machine code

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 can differentiate between integers and reals (mov, movw, addl, addf etc.)

Compiler and Machine code generation

Example, code inside a function

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

add r1, r2 # result in r1

mov r1, <location of c in memory>

Compiler and Machine code generation

Across function calls

**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

Where are the local variables in memory?

The only way to store them is on a stack.

Why?

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 esp!)

Unlike a “stack data type” data structure, this

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 ?

What goes on stack?

Local variables

Function Parameters

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 -S” output assembly code.

x86 instructions

leave

Equivalent to

```
mov %ebp, %esp # esp  
= 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

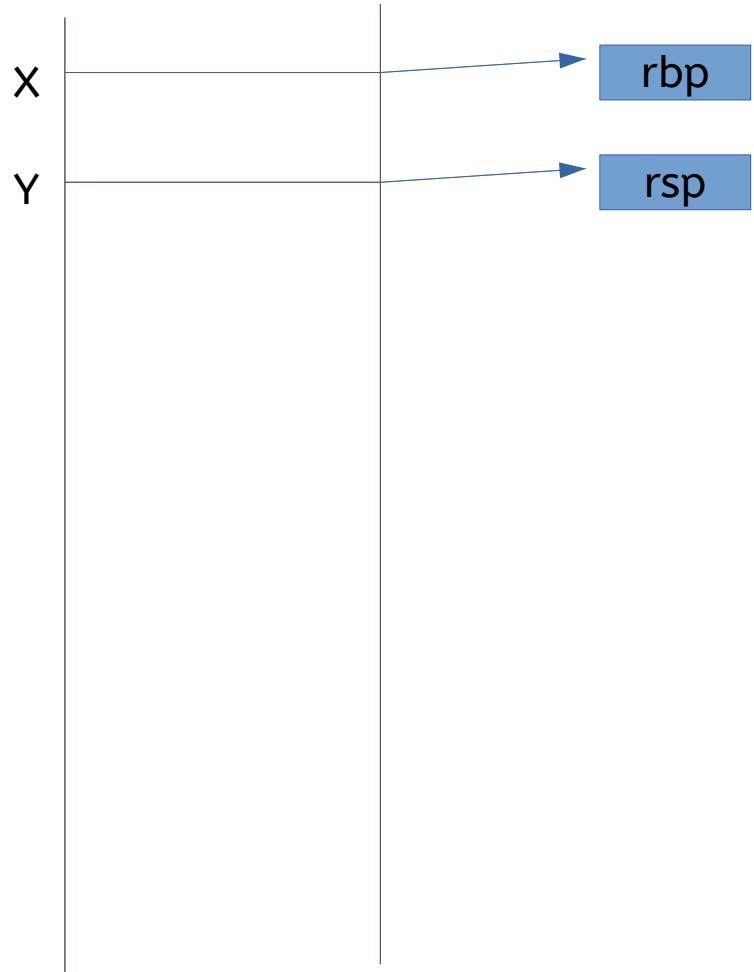
See code and understand

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;  
}
```

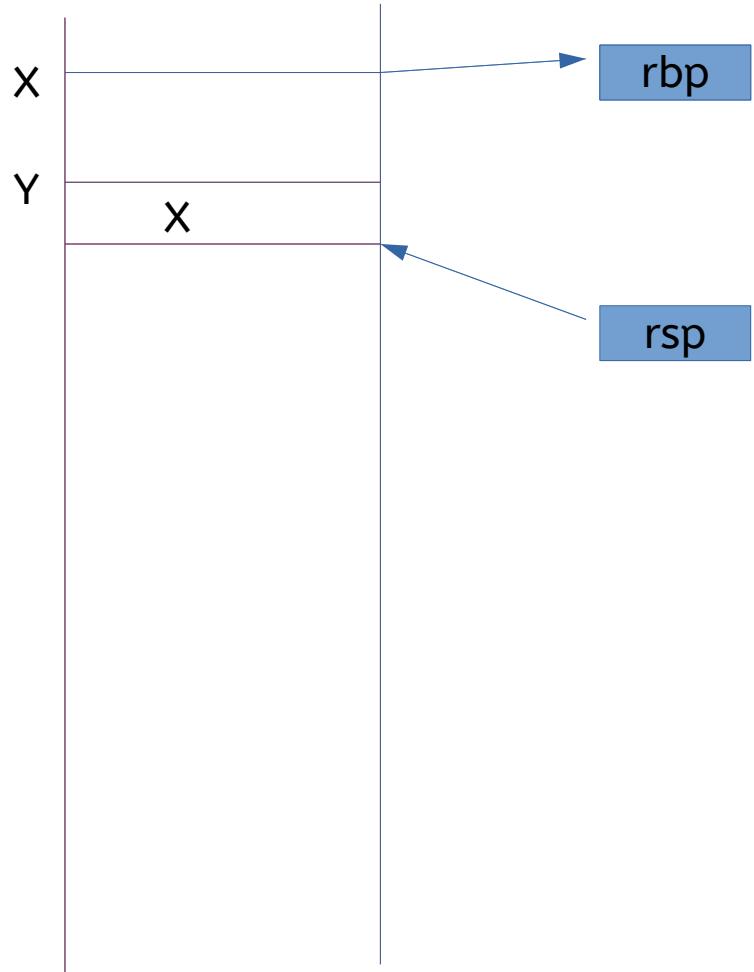
```
main:  
    endbr64  
    pushq %rbp  
    movq %rsp, %rbp  
    subq $16, %rsp  
    movl $20, -8(%rbp)  
    movl $30, -4(%rbp)  
    movl -8(%rbp), %eax  
    movl %eax, %edi  
    call f  
    movl %eax, -4(%rbp)  
    movl -4(%rbp), %eax  
    leave  
    ret
```

```
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
```

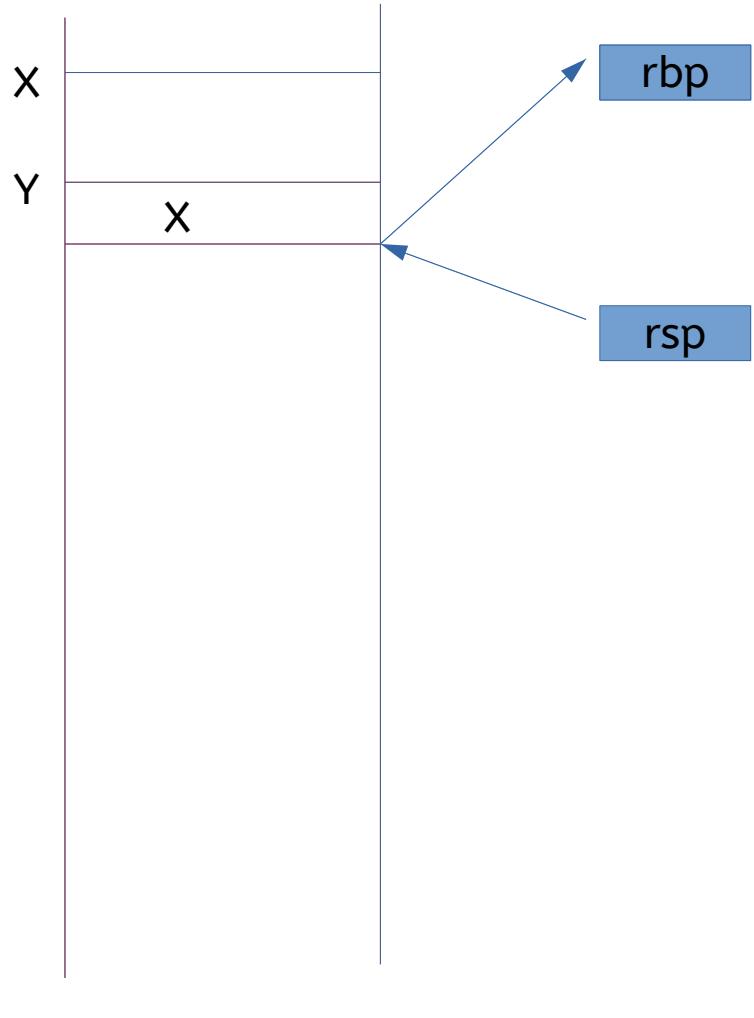


```
main:  
endbr64  
pushq %rbp  
movq %rsp, %rbp  
subq $16, %rsp  
movl $20, -8(%rbp)  
movl $30, -4(%rbp)  
movl -8(%rbp), %eax  
movl %eax, %edi  
call f  
movl %eax, -4(%rbp)  
movl -4(%rbp), %eax  
leave  
ret
```

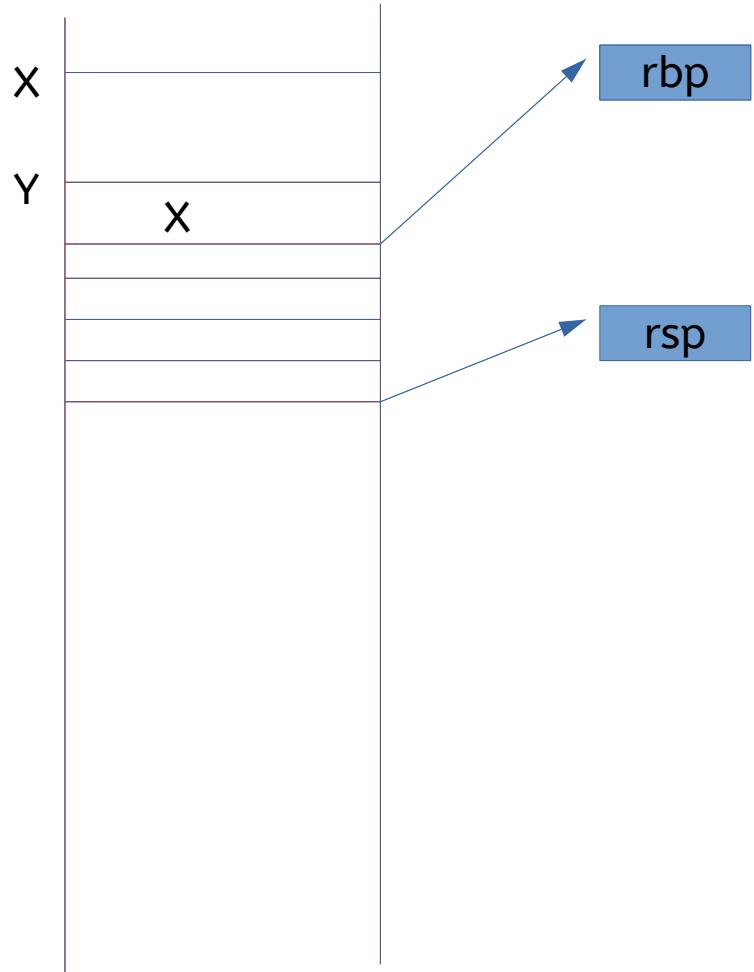
```
int main() {  
    int a = 20, b = 30;  
    b = f(a);  
    return b;  
}
```



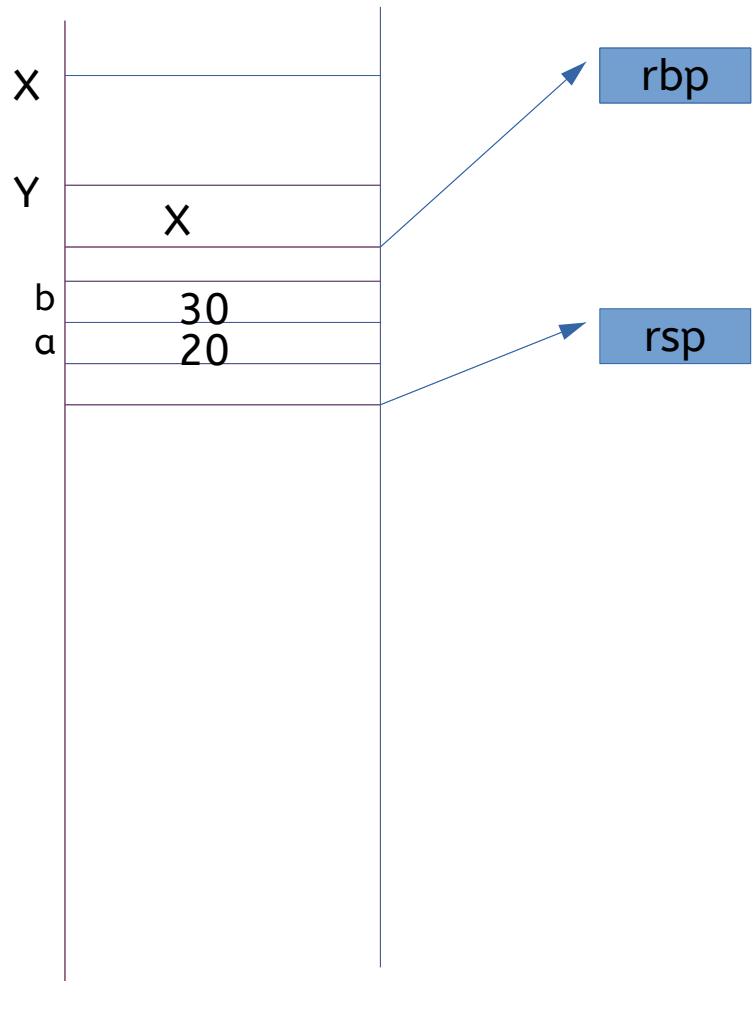
```
main:  
endbr64  
    pushq %rbp  
    movq %rsp, %rbp  
    subq $16, %rsp  
    movl $20, -8(%rbp)  
    movl $30, -4(%rbp)  
    movl -8(%rbp), %eax  
    movl %eax, %edi  
    call f  
    movl %eax, -4(%rbp)  
    movl -4(%rbp), %eax  
    leave  
    ret  
int main() {  
    int a = 20, b =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

```

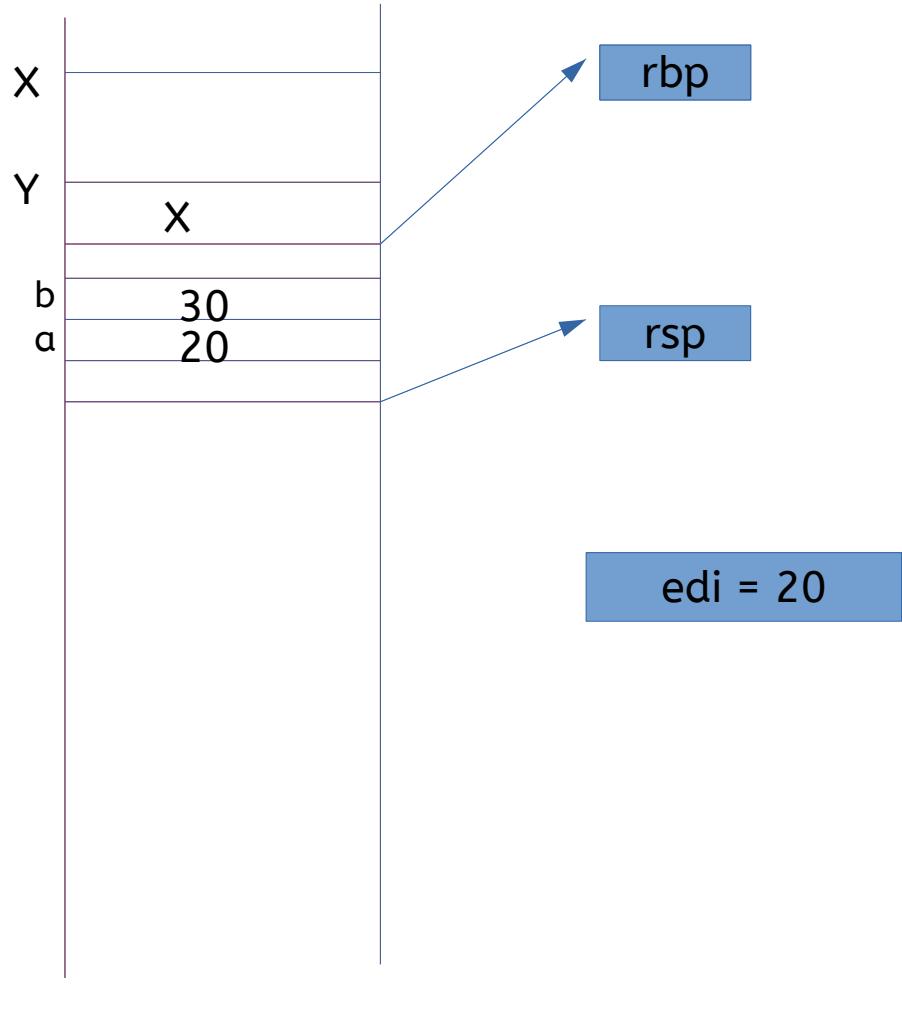
pushq %rbp
movq %rsp, %rbp
subq $16, %rsp
movl $20, -8(%rbp)
movl $30, -4(%rbp)
movl -8(%rbp), %eax
movl %eax, %edi
call f
movl %eax, -4(%rbp)
movl -4(%rbp), %eax
leave
ret

```

```

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

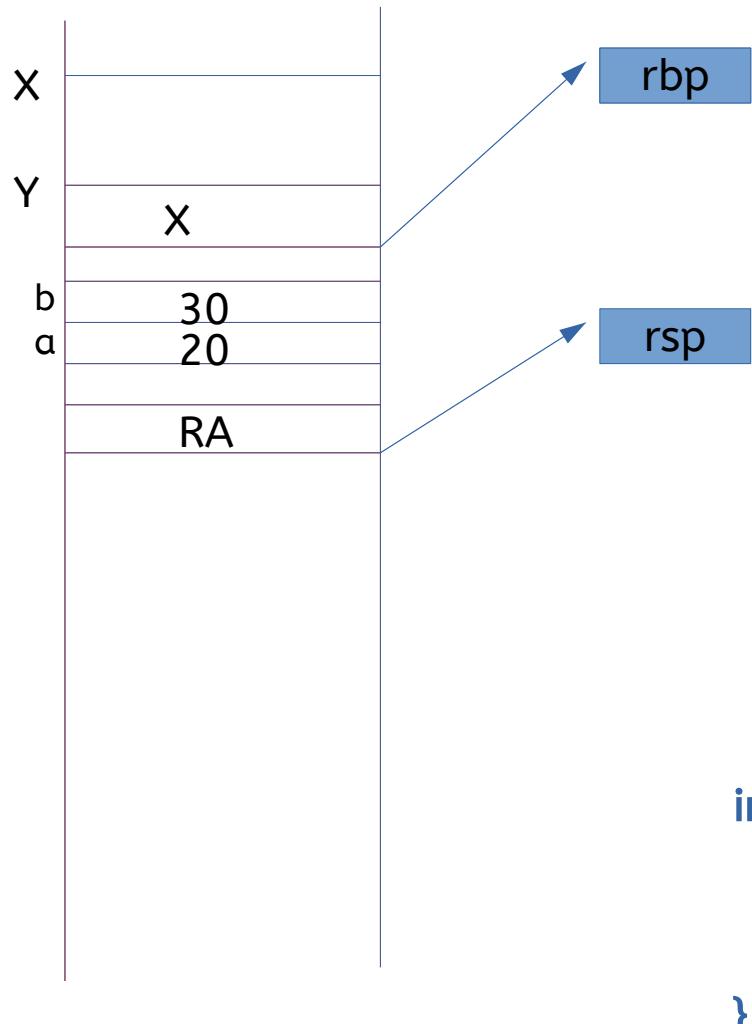
```

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 RA:
movl %eax, -4(%rbp)
movl -4(%rbp), %eax
Leave
ret
```

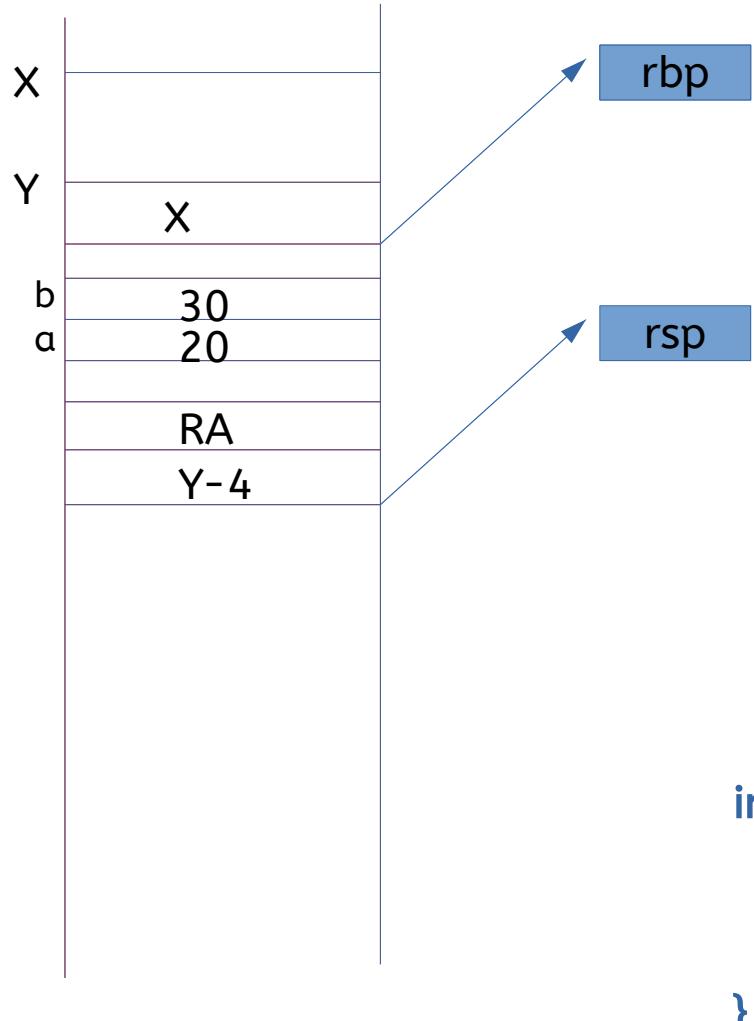
```
int f(int x) {
    int y;
    y = x + 3;
    return y;
}
```

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



```

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 RA:
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
popq %rbp
ret

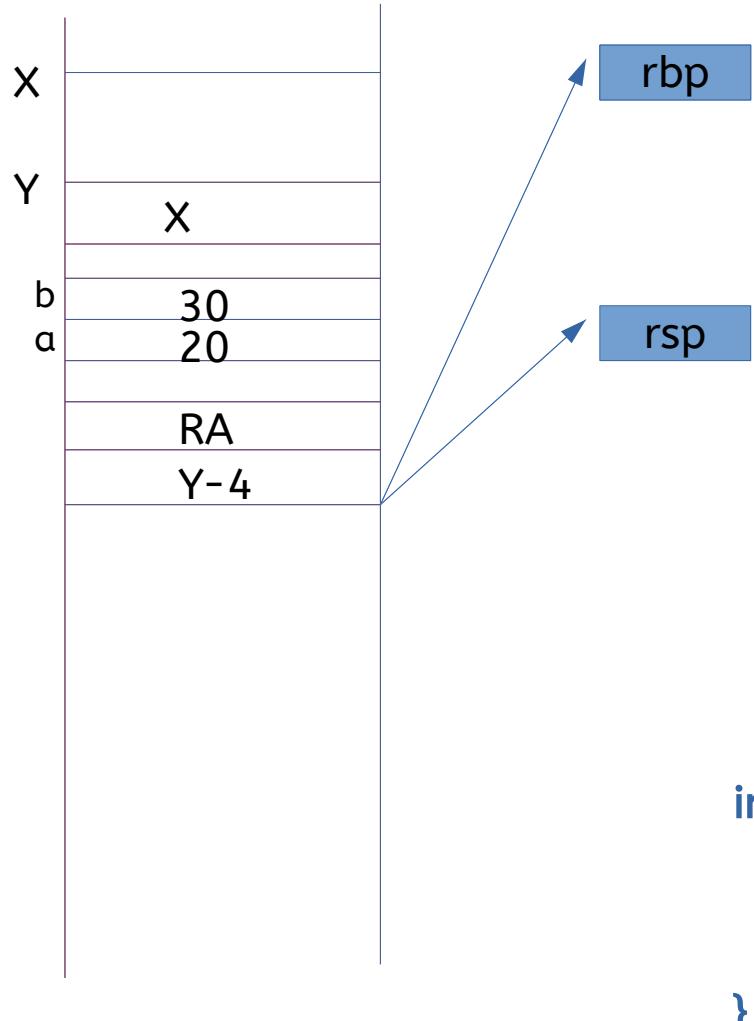
```

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 RA:
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
popq %rbp
ret

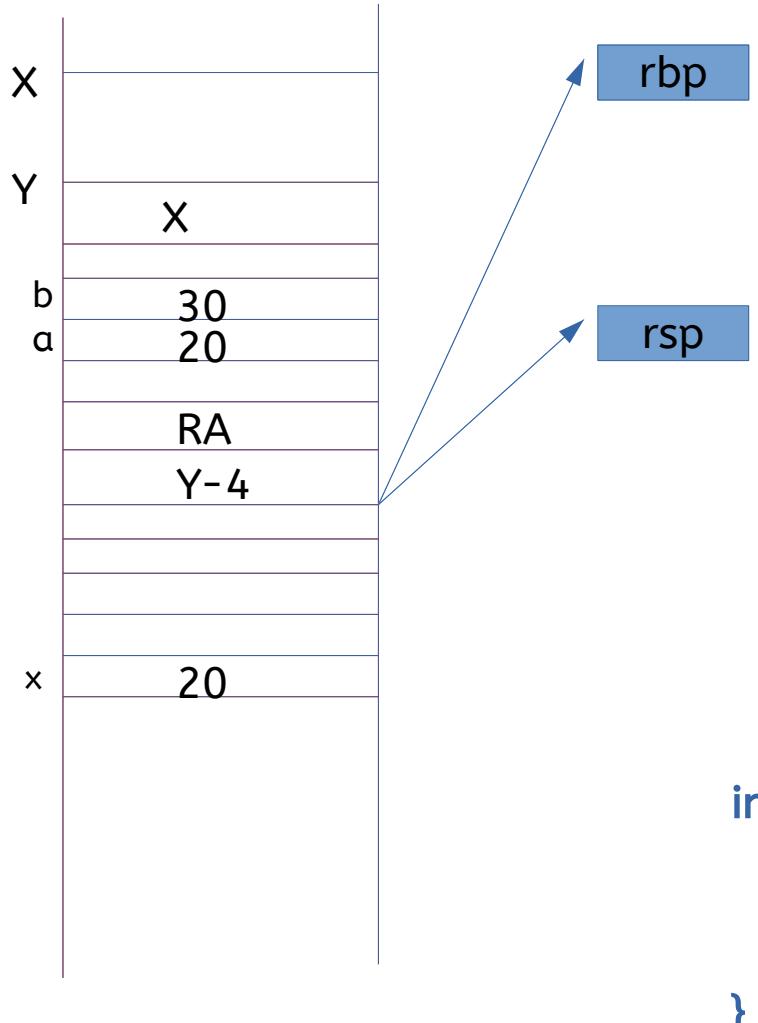
```

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 RA:
movl %eax, -4(%rbp)
movl -4(%rbp), %eax

Leave
ret

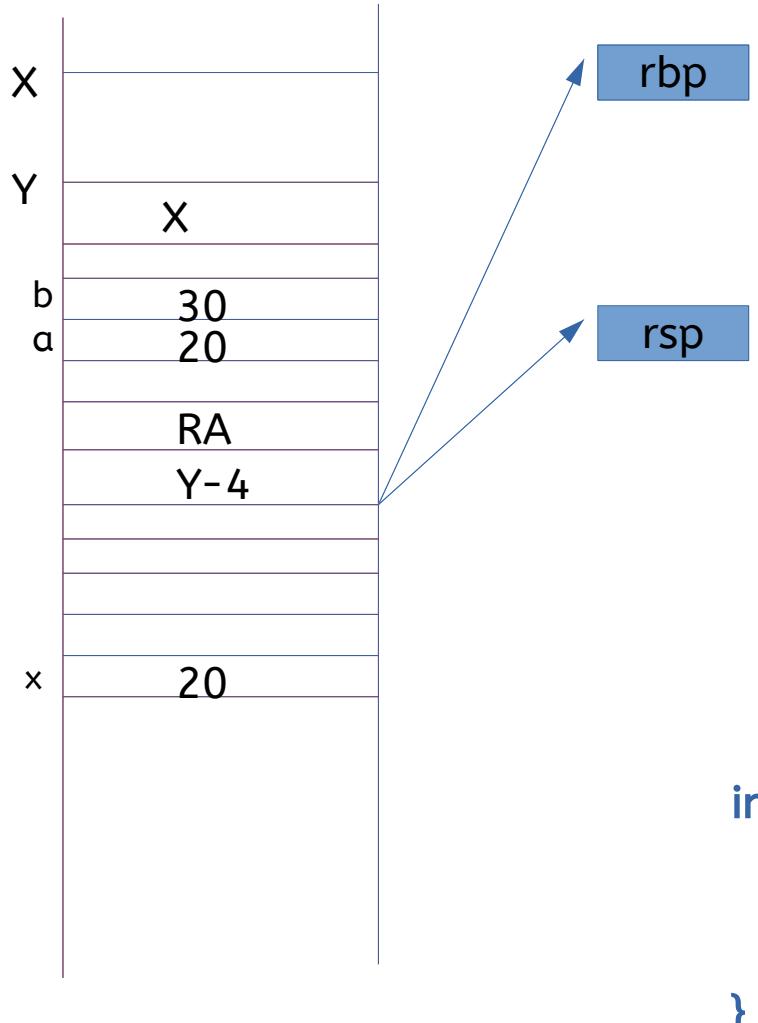
int f(int x) {
 int y;
 y = x + 3;
 return y;
}

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



```

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 RA:
movl %eax, -4(%rbp)
movl -4(%rbp), %eax
Leave
ret
    
```

```

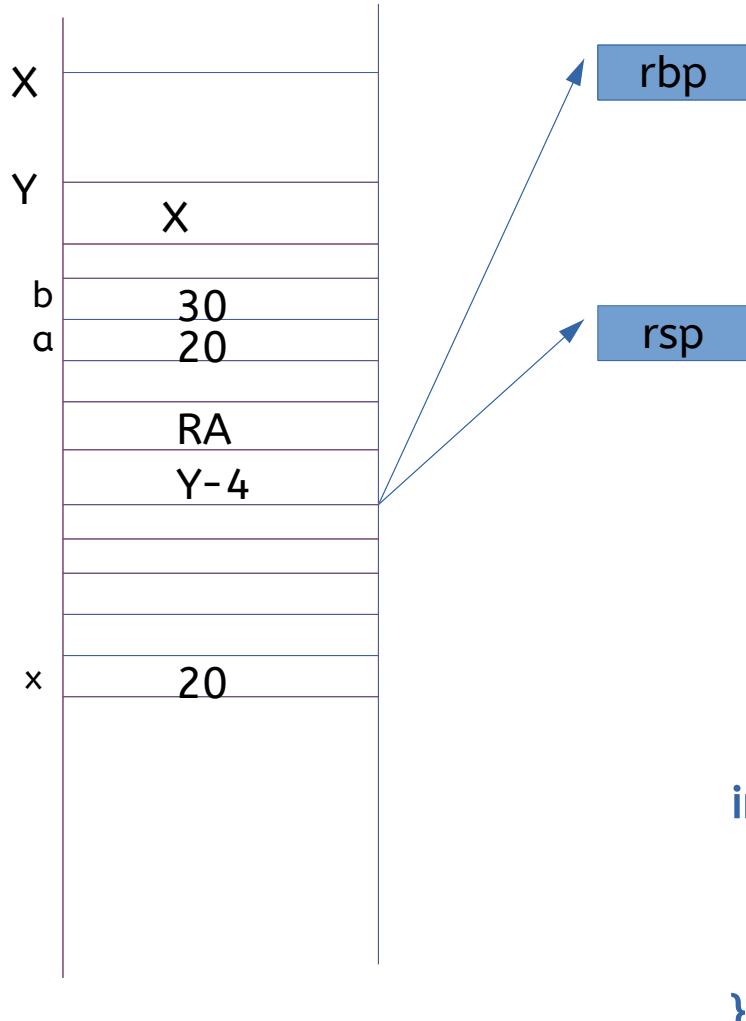
int f(int x) {
    int y;
    y = x + 3;
    return y;
}
    
```

```

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
popq %rbp
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 RA:
movl %eax, -4(%rbp)
movl -4(%rbp), %eax
Leave
ret

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

```

```

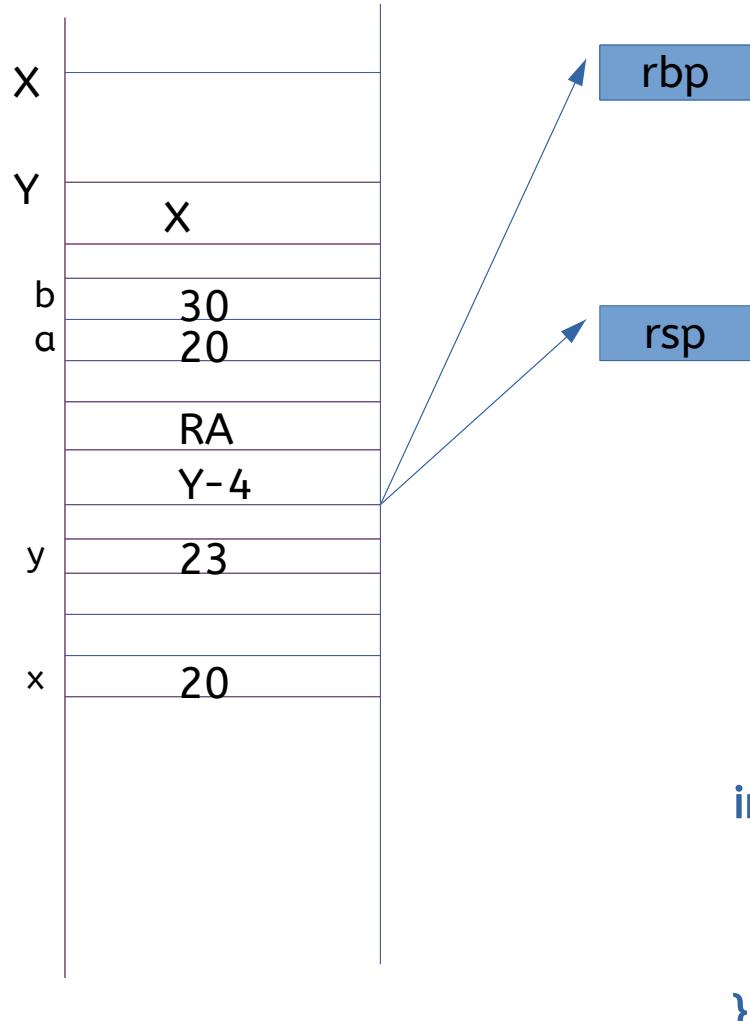
f:
endbr64
pushq %rbp
movq %rsp, %rbp
movl %edi, -20(%rbp)
movl -20(%rbp), %eax
addl $3, %eax
movl %eax, -4(%rbp)
movl -4(%rbp), %eax
popq %rbp
ret

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

```

`edi = 20`

`eax = 23`



```

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 RA:
movl %eax, -4(%rbp)
movl -4(%rbp), %eax
Leave
ret

```

```

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

```

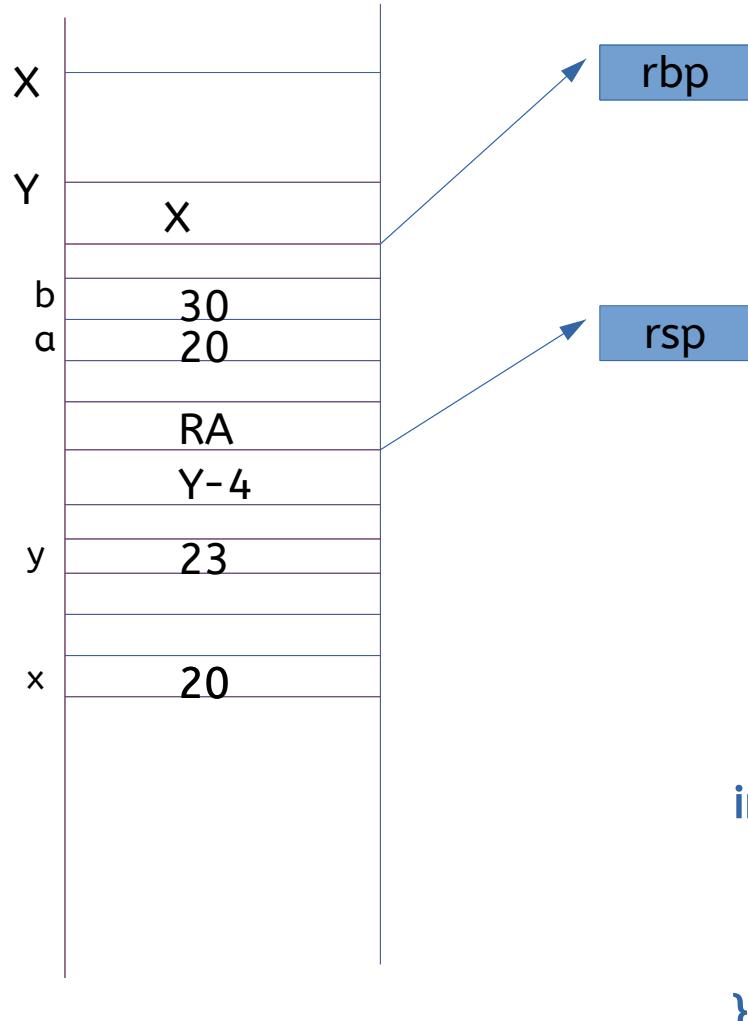
```

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
popq %rbp
ret

```

edi = 20

eax = 23



```

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 RA:
movl %eax, -4(%rbp)
movl -4(%rbp), %eax
Leave
ret

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

```

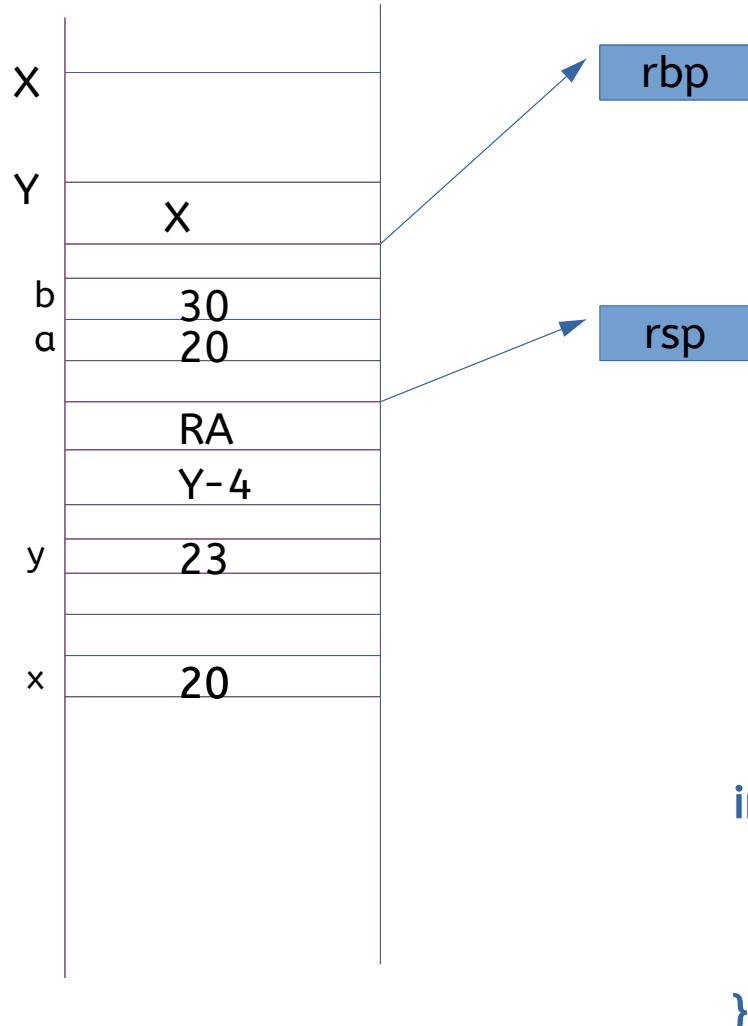
```

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
popq %rbp
ret

```

edi = 20

Eax = 23



```

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

```

```

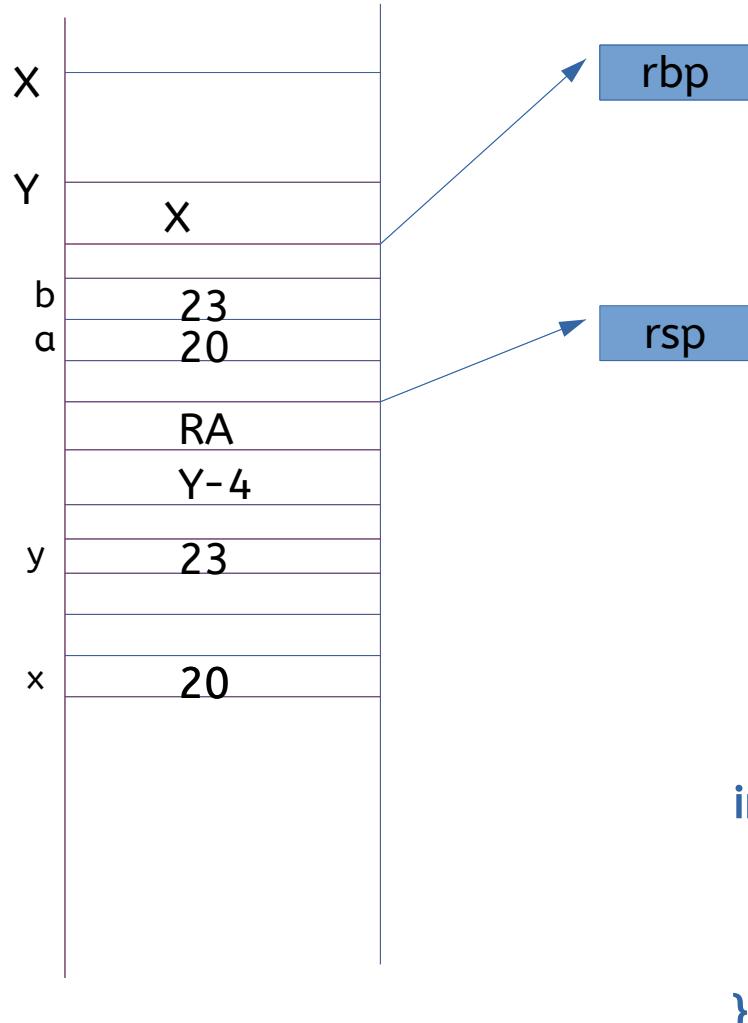
int main() {
    int a = 20, b = 30;
    b = f(a);
    return b;
}

```

`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 RA:
movl %eax, -4(%rbp) ret
movl -4(%rbp), %eax
leave
ret

```

```

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

```

```

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

```

```

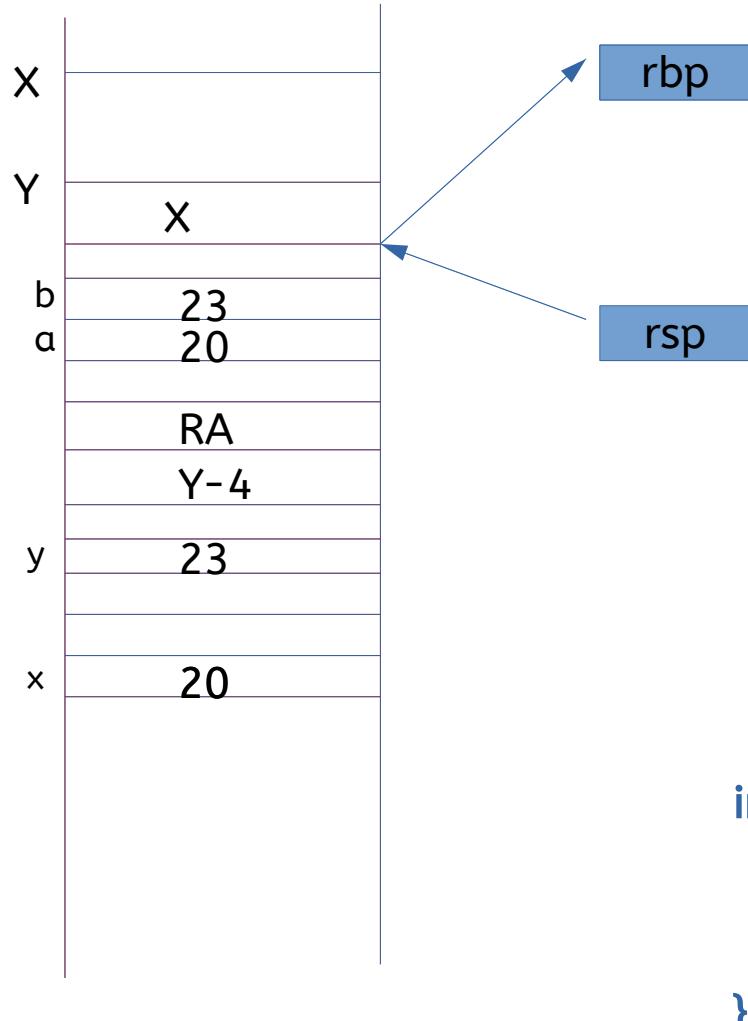
f:
endbr64
pushq %rbp
movq %rsp, %rbp
movl %edi, -20(%rbp)
movl -20(%rbp), %eax
addl $3, %eax
movl %eax, -4(%rbp)
movl -4(%rbp), %eax
popq %rbp

```

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 RA:
movl %eax, -4(%rbp)
movl -4(%rbp), %eax ret
leave
# mov rbp rsp; pop rbp
ret

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

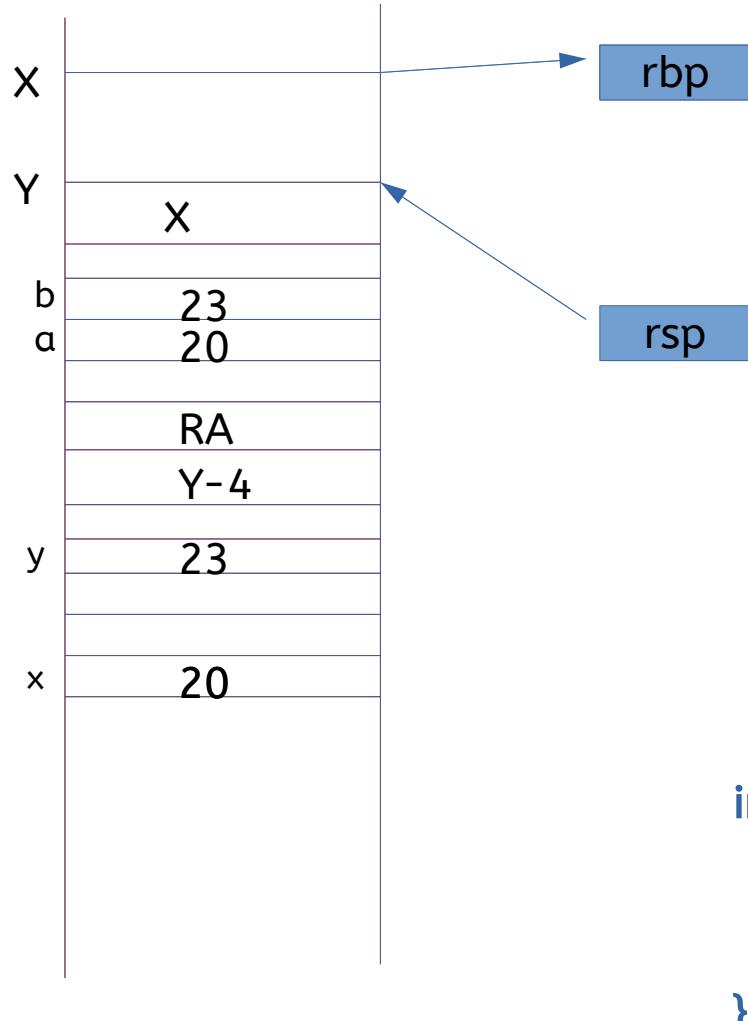
int main() {
    int a = 20, b = 30;
    b = f(a);
    return b;
}

```

`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 RA:
movl %eax, -4(%rbp)
movl -4(%rbp), %eax ret
leave
# mov ebp esp; pop ebp
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

```

edi = 20

eax = 23

eip = RA

Further on calling conventions

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

Compiler's dilemma: While generating code for a function, which registers to use?

The register might have been in use in earlier function call

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
In a register, e.g. eax**

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

3) Save callee-saved registers

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

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

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 Misruada

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
pushq %rbp
movq %rsp, %rbp
subq $48, %rsp
movl $10, -44(%rbp)
movl $20, -40(%rbp)
movl $30, -36(%rbp)
movl $40, -32(%rbp)
movl $50, -28(%rbp)
movl $60, -24(%rbp)
movl $70, -20(%rbp)
movl $80, -16(%rbp)
movl $90, -12(%rbp)
movl $100, -8(%rbp)
```

```
movl -24(%rbp), %r9d
movl -28(%rbp), %r8d
movl -32(%rbp), %ecx
movl -36(%rbp), %edx
movl -40(%rbp), %esi
movl -44(%rbp), %eax
movl -8(%rbp), %edi
pushq %rdi
movl -12(%rbp), %edi
pushq %rdi
movl -16(%rbp), %edi
pushq %rdi
movl -20(%rbp), %edi
pushq %rdi

movl %eax, %edi
call f
addq $32, %rsp
movl %eax, -4(%rbp)
movl -4(%rbp), %eax
leave
ret
```

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

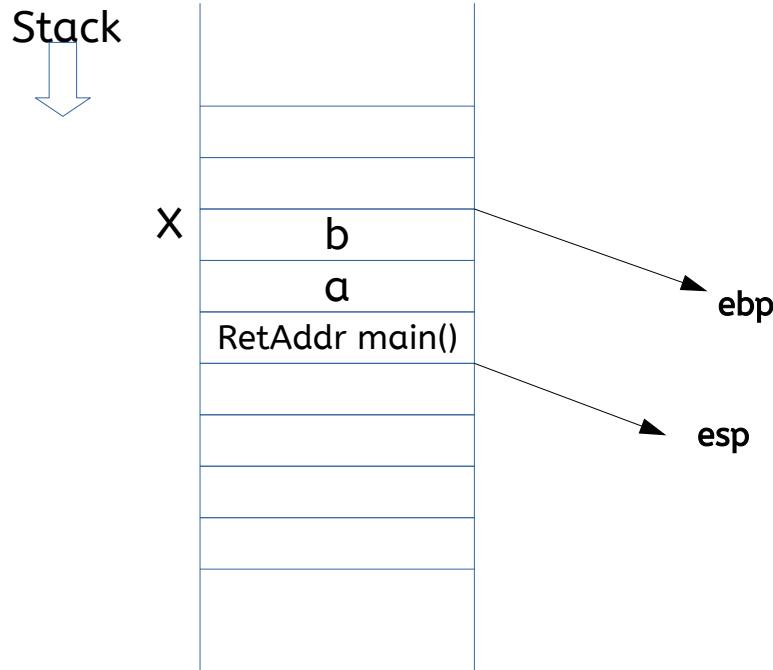
```
int mult(int a, int b) {  
    int c, d = 20, e = 30, f;  
    f = add(d, e);  
    c = a * b + f;  
    return c;  
}  
  
int add(int x, int y) {  
    int z;  
    z = x + y;  
    return z;  
}
```

Translated to assembly as:

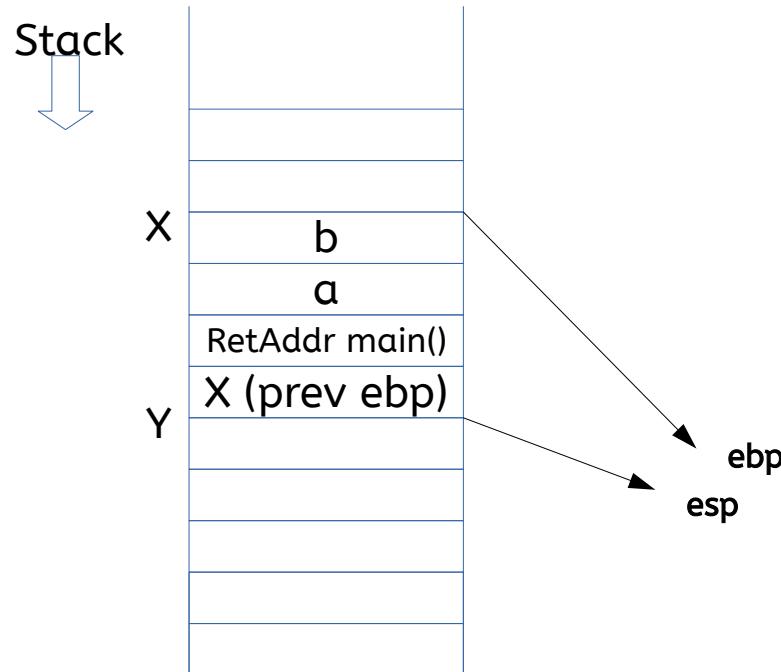
```
add:  
pushl %ebp  
movl %esp, %ebp  
subl $16, %esp  
movl 8(%ebp), %edx  
movl 12(%ebp), %eax  
addl %edx, %eax  
movl %eax, -4(%ebp)  
movl -4(%ebp), %eax  
leave  
ret
```

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
```



```
/* 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;
    /* Control is here */
    return z;
    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;
}

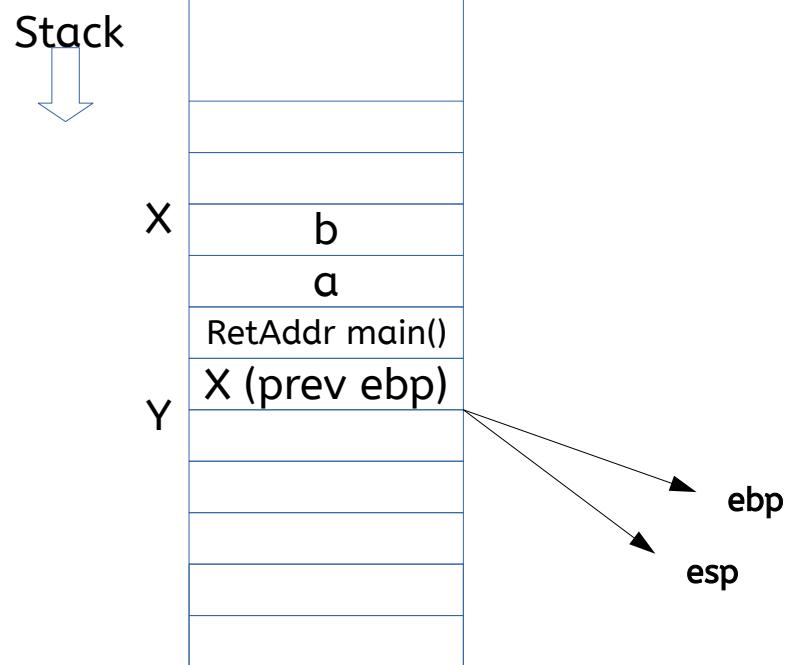
```

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

```



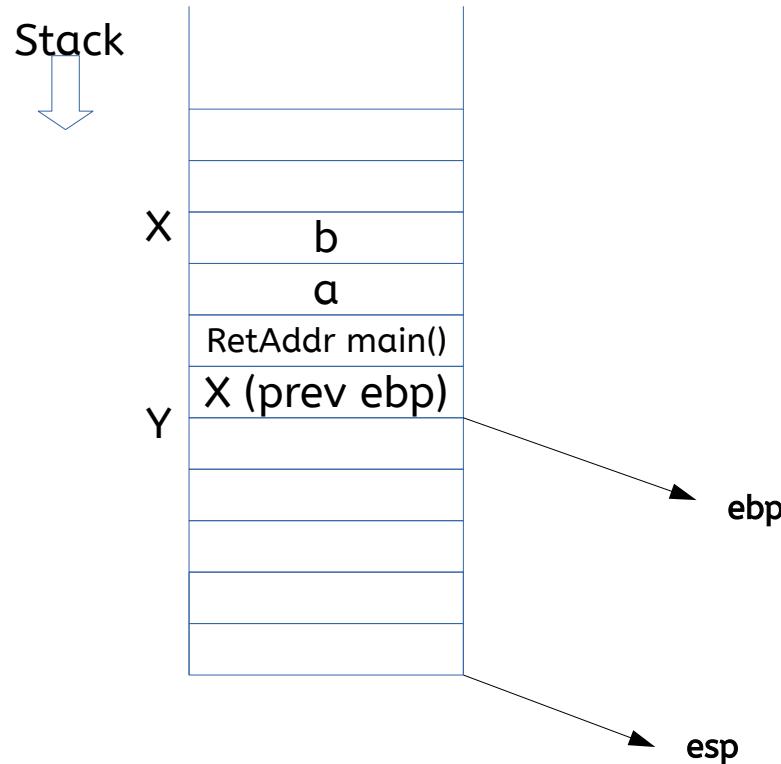
```

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

```

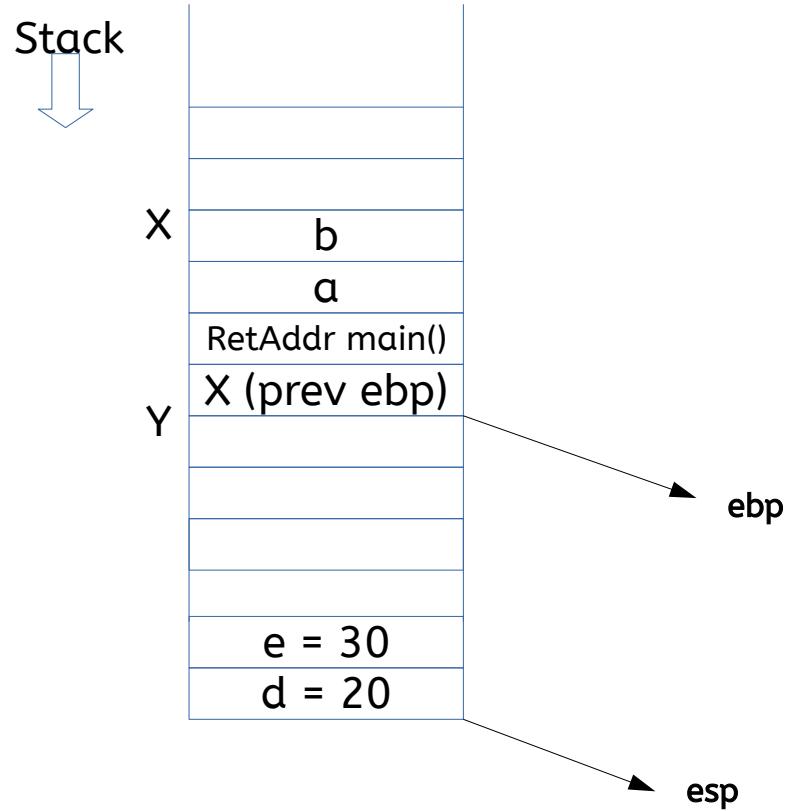


```

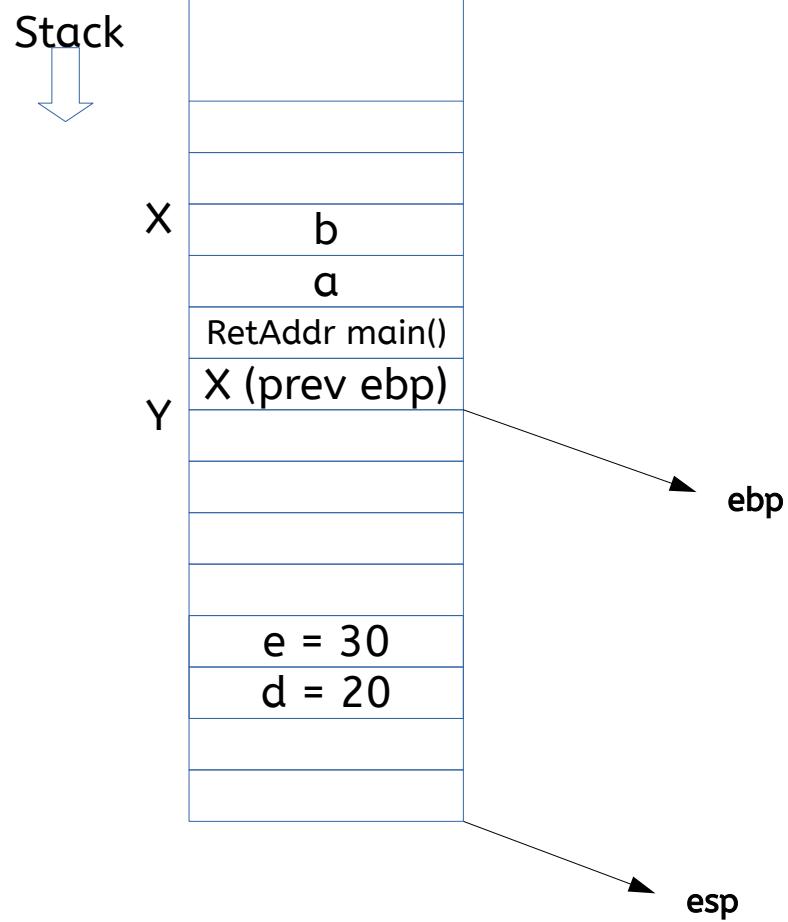
int mult(int a, int b) {
    int c, d = 20, e = 30, f;
    f = add(d, e);
    c = a * b + f;
    return c;
}
int add(int x, int y) {
    int z;
    z = x + y;
    return z;
}

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
```



```

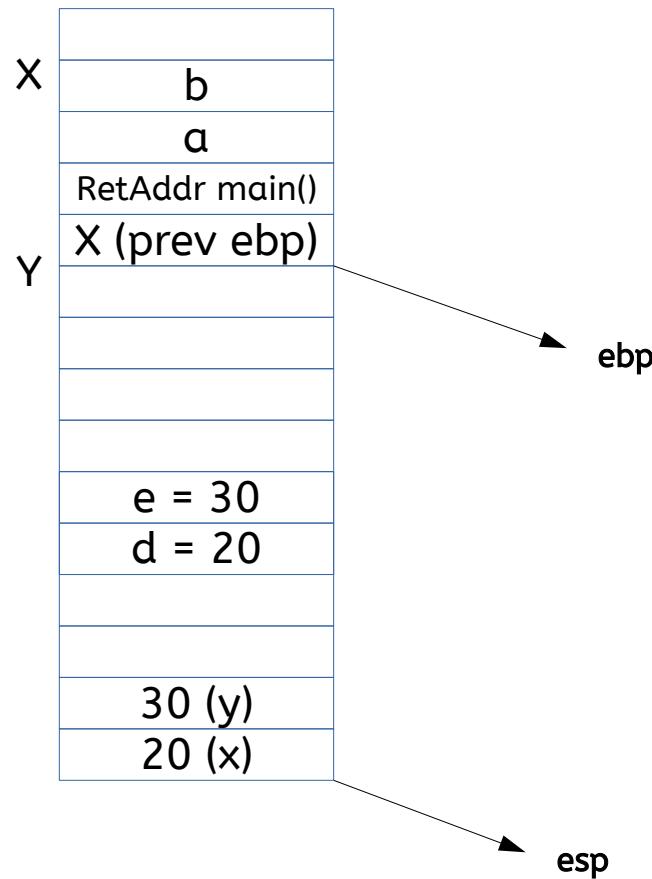
int mult(int a, int b) {
    int c, d = 20, e = 30, f;
    f = add(d, e);
    c = a * b + f;
    return c;
}

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

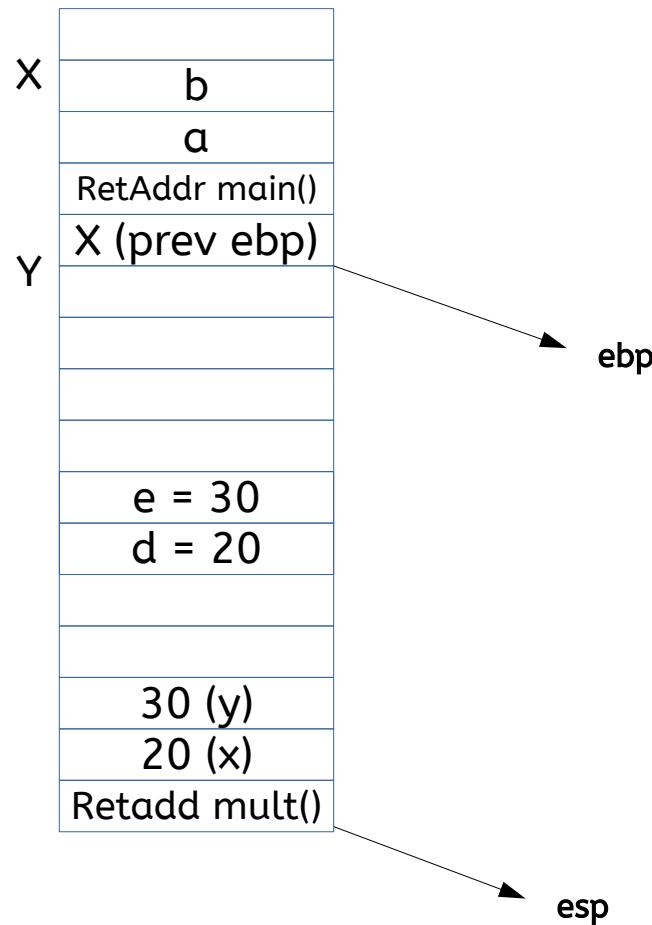
```

Stack
↓



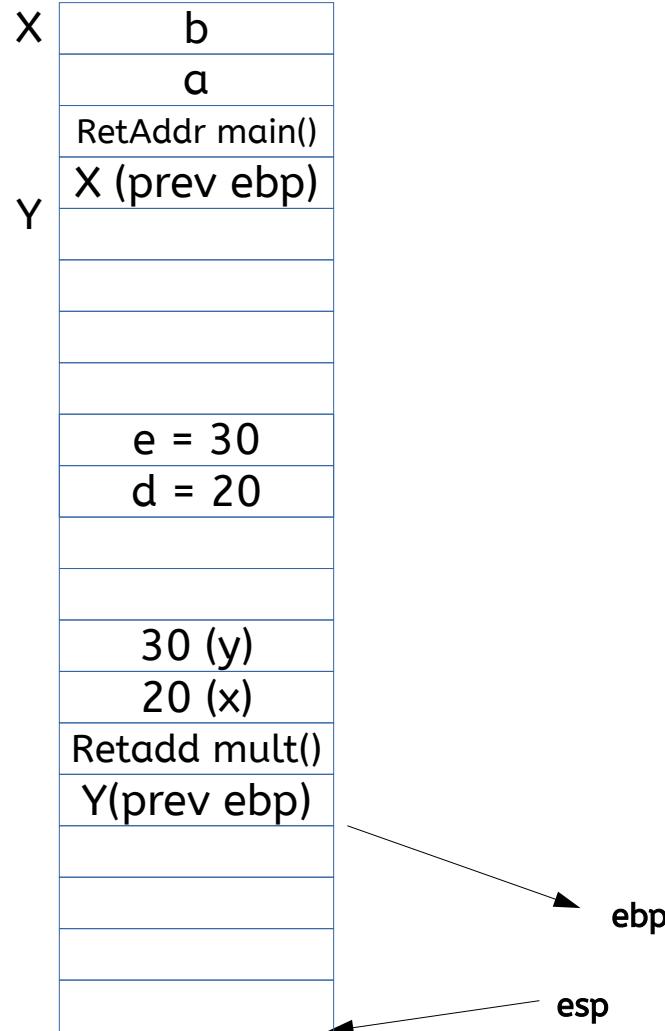
```
int mult(int a, int b) {  
    int c, d = 20, e = 30, f;  
    f = add(d, e);  
    c = a * b + f;  
    return c;  
}  
int add(int x, int y) {  
    int z;  
    z = x + y;  
    return z;  
}  
mult:  
    pushl %ebp  
    movl %esp, %ebp  
    subl $24, %esp  
    movl $20, -24(%ebp)  
    movl $30, -20(%ebp)  
    subl $8, %esp  
    pushl -20(%ebp)  
    pushl -24(%ebp)  
    call add
```

Stack
↓



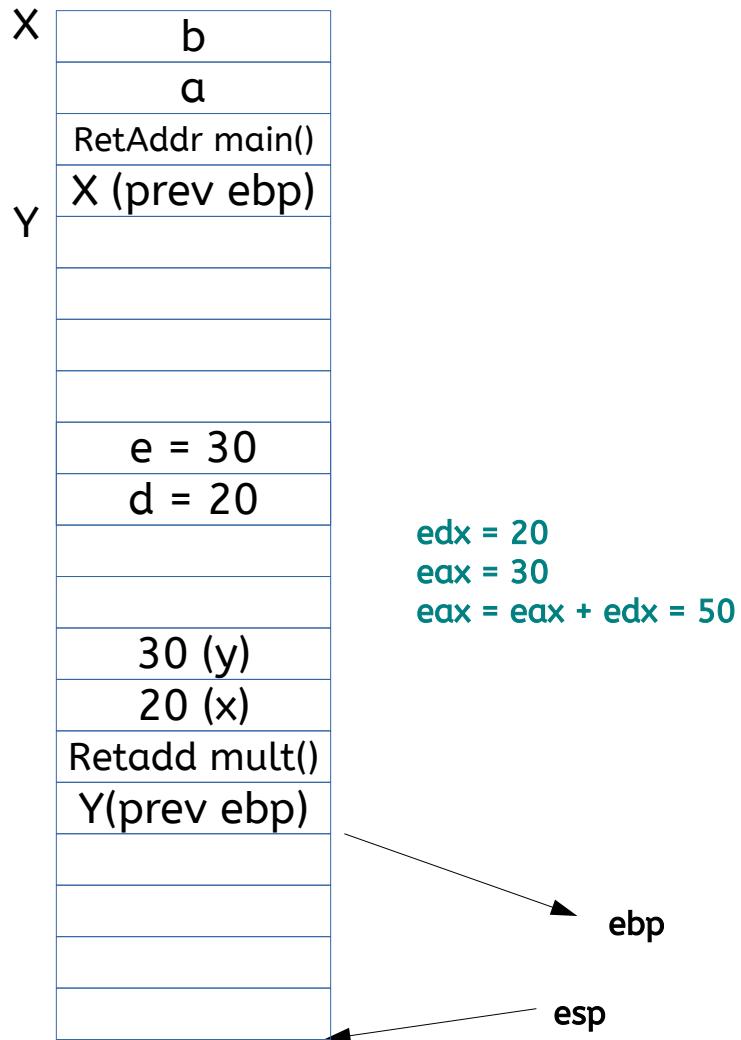
```
int mult(int a, int b) {  
    int c, d = 20, e = 30, f;  
    f = add(d, e);  
    c = a * b + f;  
    return c;  
}  
int add(int x, int y) {  
    int z;  
    z = x + y;  
    return z;  
}  
mult:  
    pushl %ebp  
    movl %esp, %ebp  
    subl $24, %esp  
    movl $20, -24(%ebp)  
    movl $30, -20(%ebp)  
    subl $8, %esp  
    pushl -20(%ebp)  
    pushl -24(%ebp)  
    call add
```

Stack



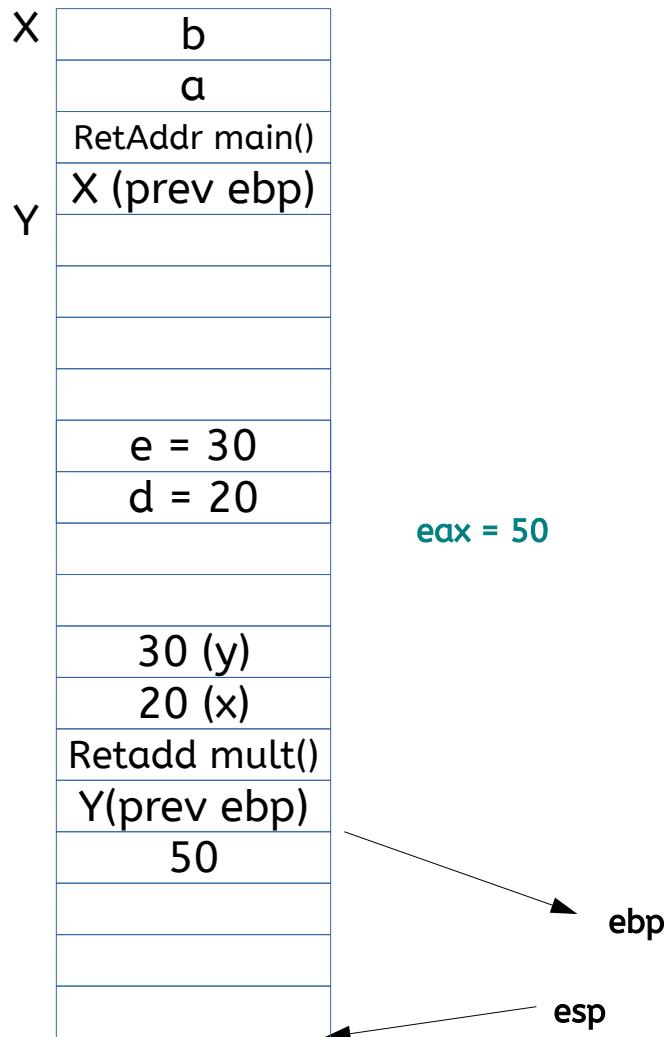
```
int mult(int a, int b) {  
    int c, d = 20, e = 30, f;  
    f = add(d, e);  
    c = a * b + f;  
    return c;  
}  
int add(int x, int y) {  
    int z;  
    z = x + y;  
    add:  
    return z;  
    pushl %ebp  
    movl %esp, %ebp  
    subl $16, %esp  
    movl 8(%ebp), %edx  
    movl 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;  
    add:  
    return z;  
    }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;  
    add:  return z;  
    }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 i
```

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

leave: step 1

```
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;  
    add:  return z;  
    }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 EBP.  
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

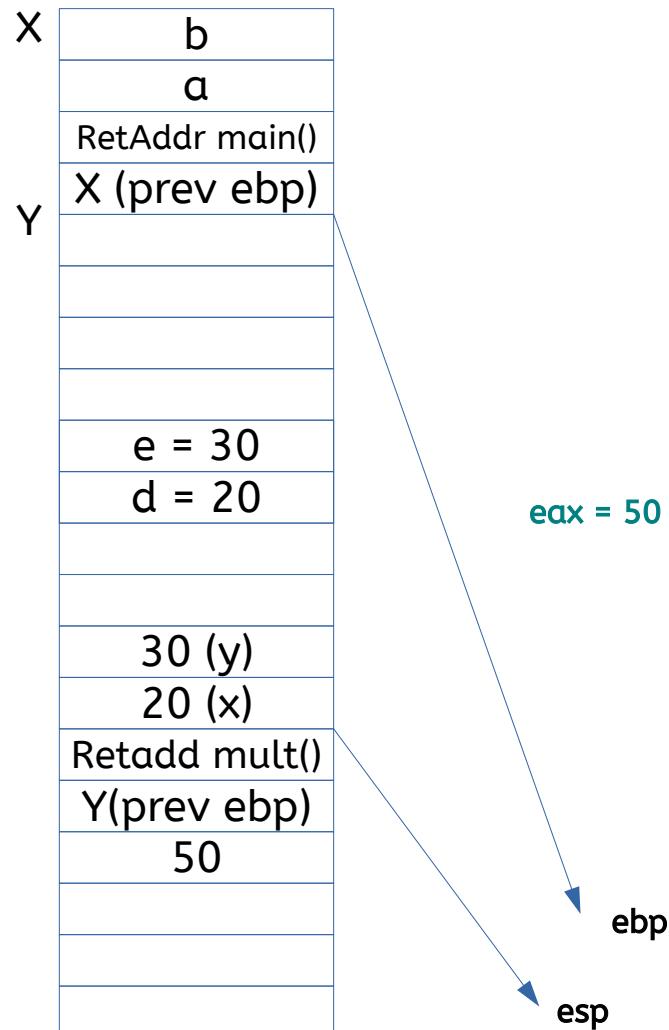
leave: step 2

eax = 50

ebp
esp

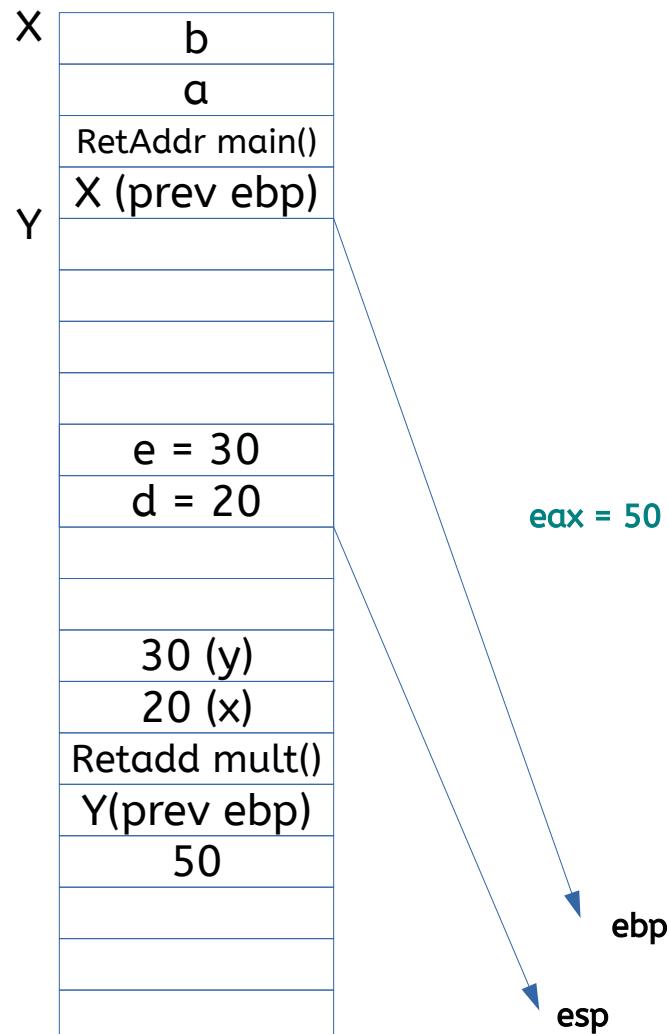
```
int mult(int a, int b) {  
    int c, d = 20, e = 30, f;  
    f = add(d, e);  
    c = a * b + f;  
    return c;  
}  
  
int add(int x, int y) {  
    int z;  
    z = x + y;  
    add:    return z;  
    }pushl %ebp  
    movl %esp, %ebp  
    subl $16, %esp  
    movl 8(%ebp), %edx  
    movl 12(%ebp), %eax  
    addl %edx, %eax  
    movl %eax, -4(%ebp)  
    movl -4(%ebp), %eax  
  
leave # # Set ESP to EBP, then pop EBP.  
ret
```

Stack
↓



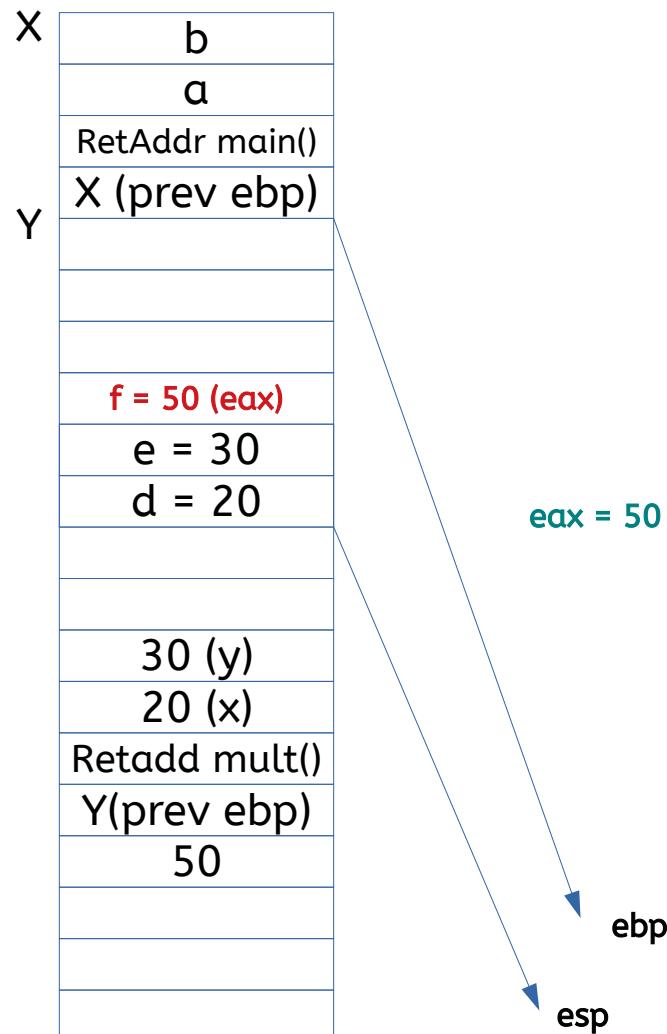
```
int mult(int a, int b) {  
    int c, d = 20, e = 30, f;  
    f = add(d, e); // here  
    c = a * b + f;  
    return c;  
}  
  
int add(int x, int y) {  
    int z;  
    z = x + y;  
    add:    return z;  
    }pushl %ebp  
    movl %esp, %ebp  
    subl $16, %esp  
    movl 8(%ebp), %edx  
    movl 12(%ebp), %eax  
    addl %edx, %eax  
    movl %eax, -4(%ebp)  
    movl -4(%ebp), %eax  
  
    leave # # Set ESP to EBP, then pop EBP.  
    ret
```

Stack
↓



```
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;  
}....  
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
↓



```
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

Again some redundant code

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

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

And add line “`source ~/.cscope_maps.vim`” in
your `~/.vimrc` file

Read this tutorial

http://cscope.sourceforge.net/cscope_vim_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

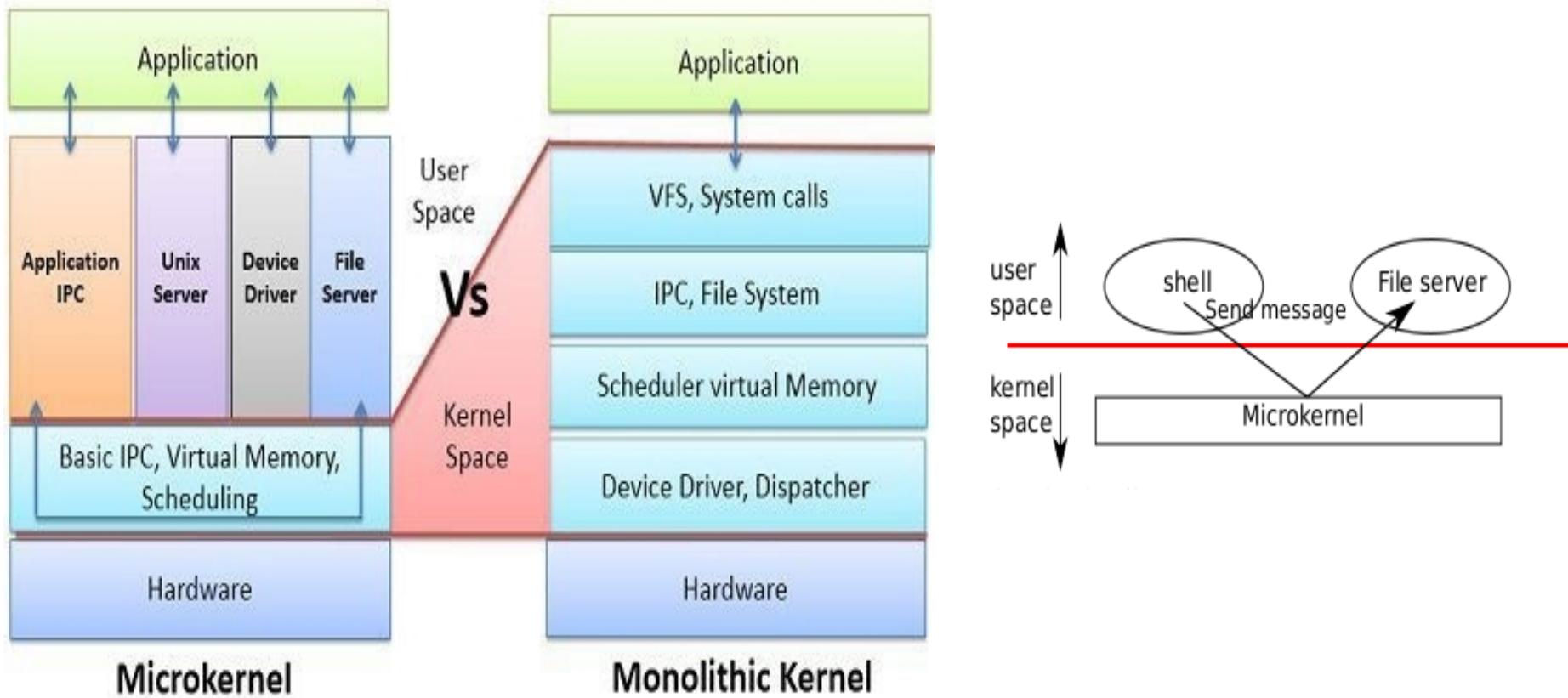
CREATE_SUBDIRS = YES

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

CALL_GRAPH = 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 -serial mon:stdio -drive  
file=fs.img,index=1,media=disk,format=raw -  
drive  
file=xv6.img,index=0,media=disk,format=raw -  
smp 2 -m 512
```

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

Two cores in SMP mode to be simulated

-m 512

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

\$(OBJDUMP) -S bootblock.o >
bootblock.asm

\$(OBJCOPY) -S -O binary -j .text

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**

**\$(OBJCOPY) -S -O binary -j .text
bootblock.o bootblock**

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

E.g. for file “kernel”

```
$(OBJDUMP) -t kernel | sed '1,/SYMBOL TABLE/d;  
s/ * / /; /^$$/d' > kernel.sym
```

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 filetyp, inode number, type

cat README

ls;ls

cat README | grep BUILD

echo hi there

echo hi there | grep hi

echo "hi there"

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 # fails

ls .. # works from
inside test

cd # fails

cd / # works

wc README

rm out

ls . test # listing both
directories

ln cat xyz; ls

rm 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

Remembering is important !

To understand kernel code, you should be

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 your understanding of the code

Memory Management Basics

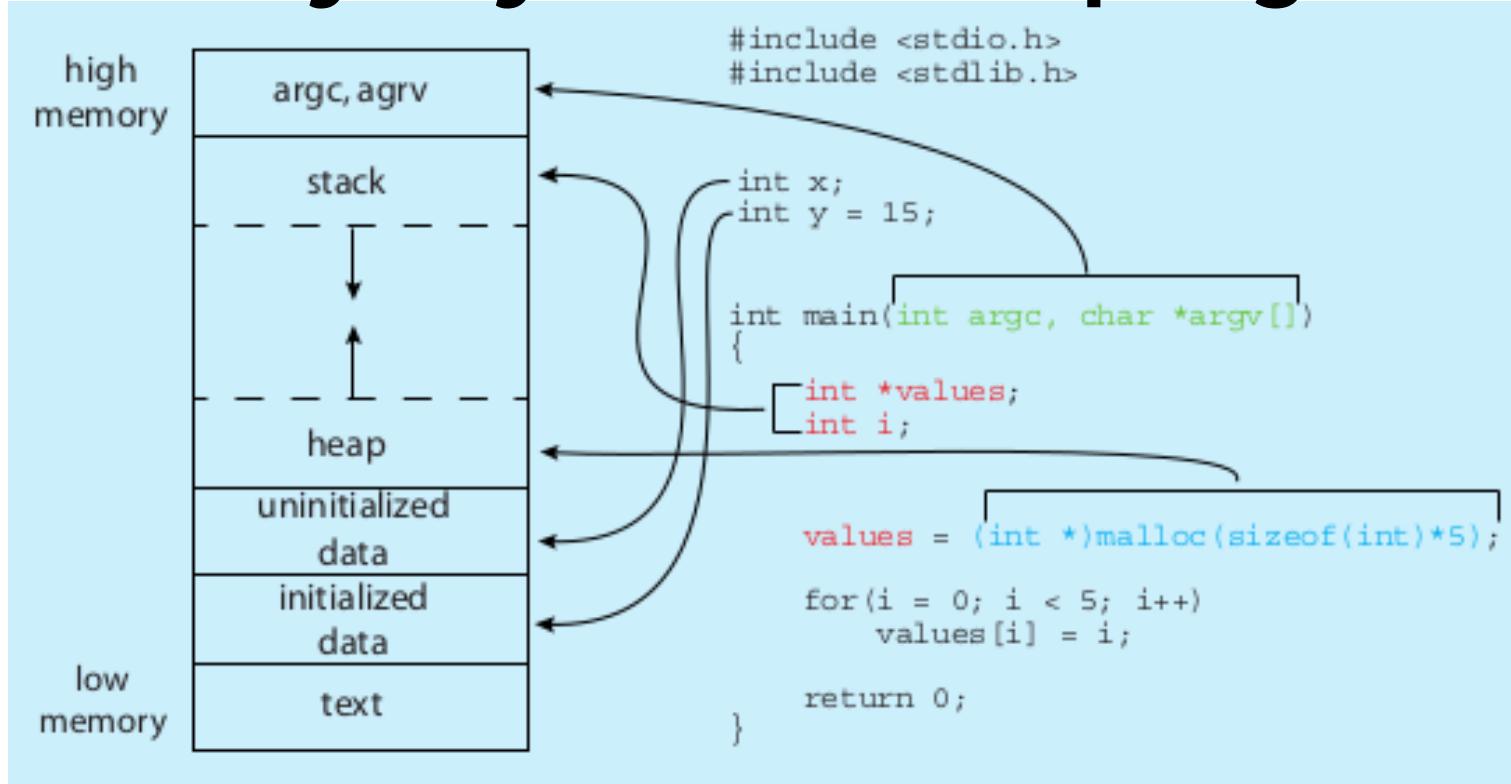
Summary

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

Addresses issued by CPU

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

Memory layout of a C program



\$ size /bin/ls

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

Desired from a multi-tasking system

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

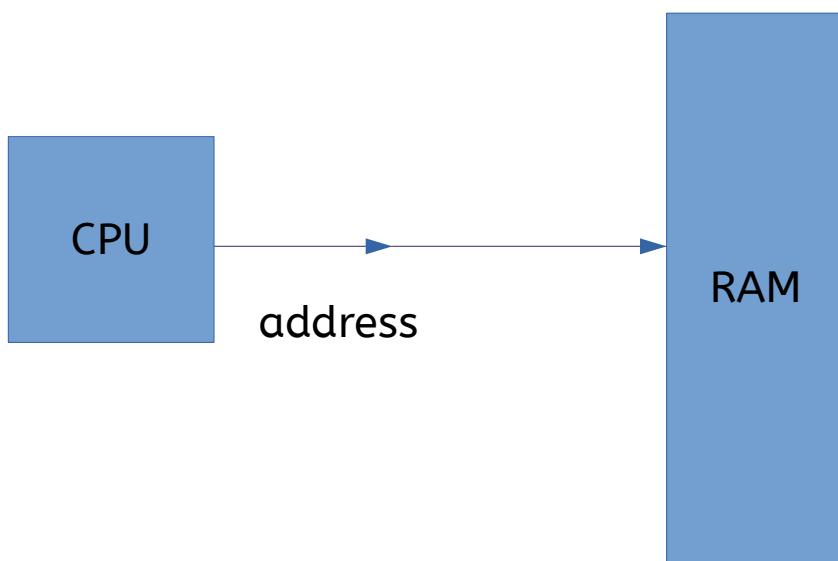
Different ‘times’

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

Different types of Address binding

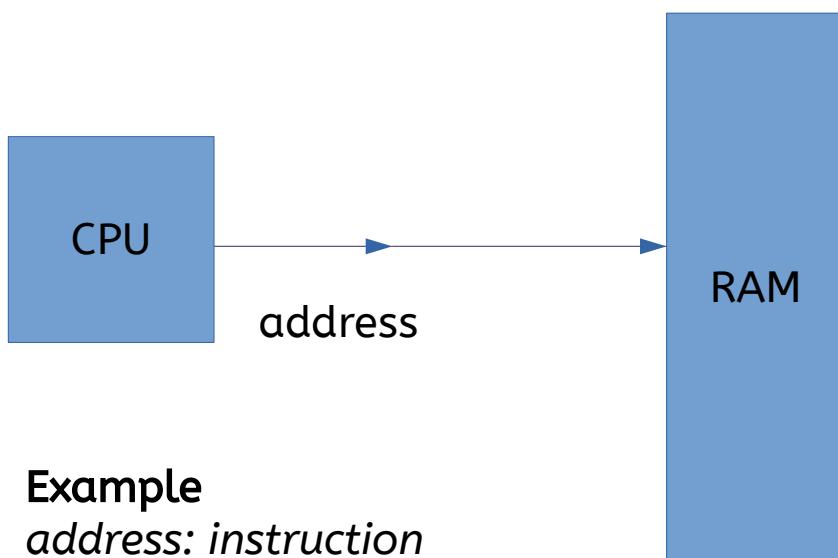
- Compile time address binding
 - Address of code/variables is fixed by compiler
 - Very rigid scheme
 - Location of process in RAM can not be changed !
Non-relocatable code.
 - Load time address binding
 - Address of code/variables is fixed by loader
 - Location of process in RAM is decided at load time, but can't be changed later
 - Flexible scheme, relocatable code
 - Run time address binding
- Which binding is actually used, is mainly determined by the operating system.

Simplest case



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

Simplest case



Example
address: *instruction*

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

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

- How does this impact the compiler and OS ?
- When a process is running the addresses issued by it, will reach the **RAM directly**
 - So exact addresses of globals, addresses in “jmp” and “call” must be

Simplest case



Example

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

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

Simplest case



Example

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

- Problem with this solution
 - Programs once loaded in RAM must stay there, can't be moved
 - What about 2 programs?
 - Compilers being “programs”, will make same assumptions and are likely to generate same/overlapping

Base/Relocation + Limit scheme

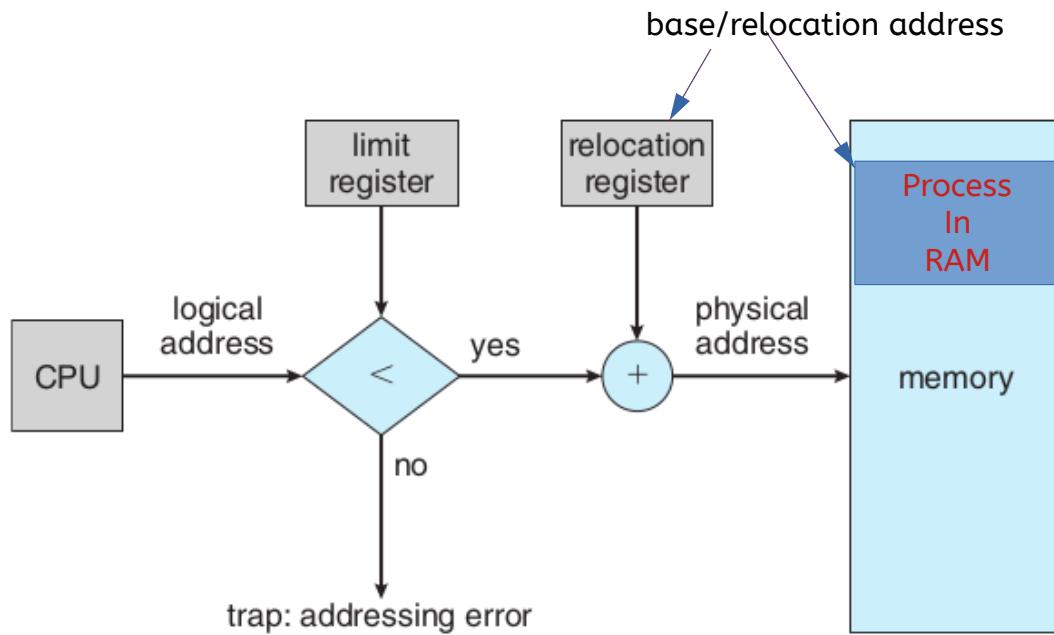


Figure 9.6 Hardware support for relocation and limit registers.

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

Memory Management Unit (MMU)

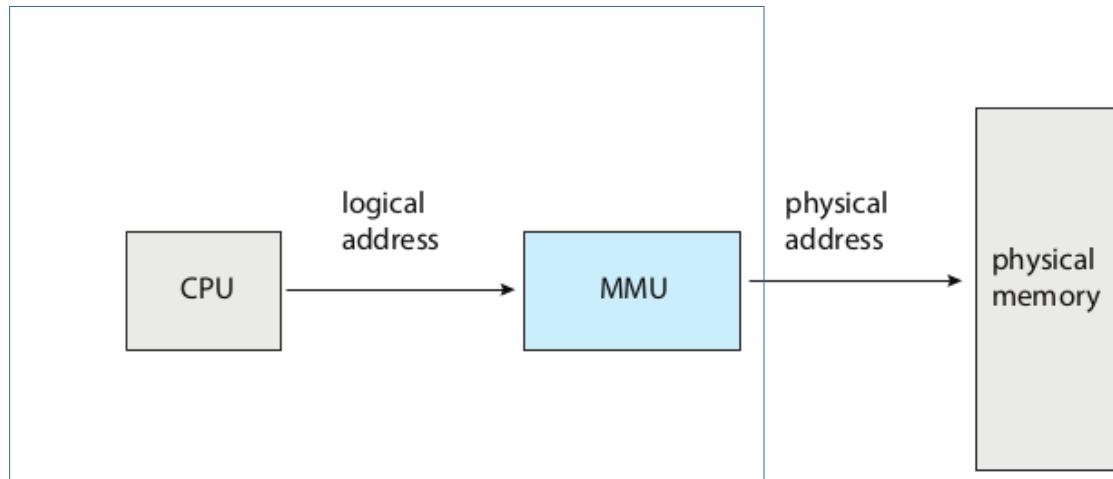
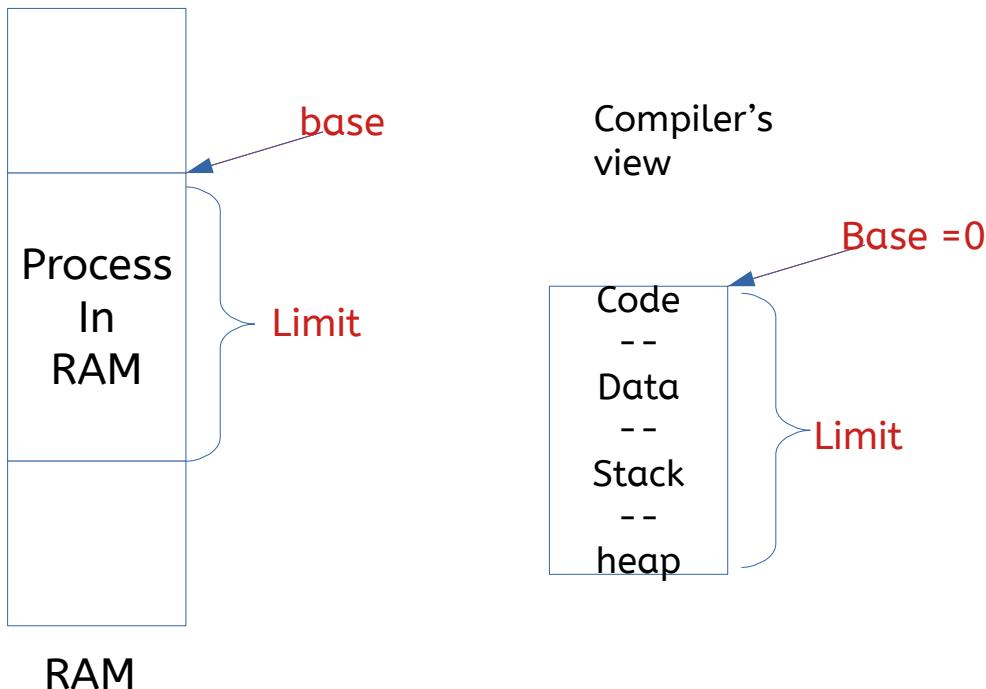


Figure 9.4 Memory management unit (MMU).

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

Base/Relocation + Limit scheme



- Compiler's work

- Assume that the process is one continuous chunk in memory, with a size limit
- Assume that the process starts at address zero (!) and calculate addresses for globals, code

Base/Relocation + Limit scheme

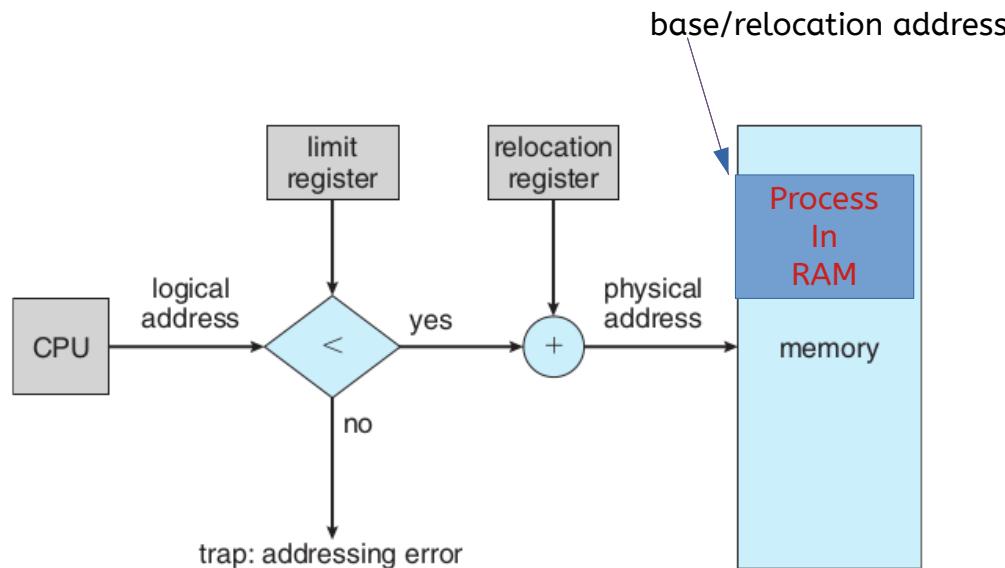


Figure 9.6 Hardware support for relocation and limit registers.

- OS's work
 - While loading the process in memory – must load as one continuous segment
 - Fill in the ‘base’ register with the actual address of the process in RAM.
 - Setup the limit

Base/Relocation + Limit scheme

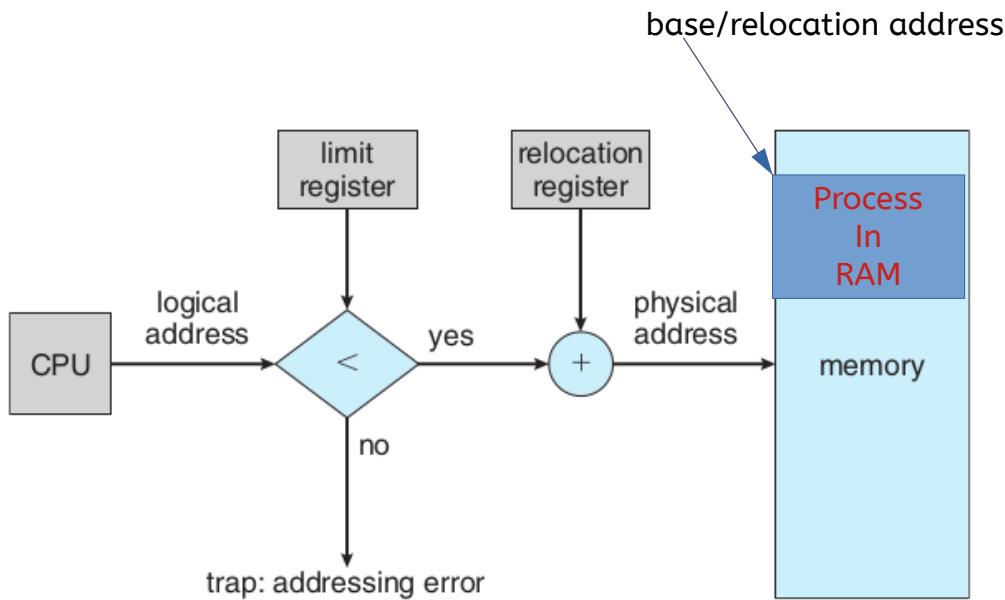
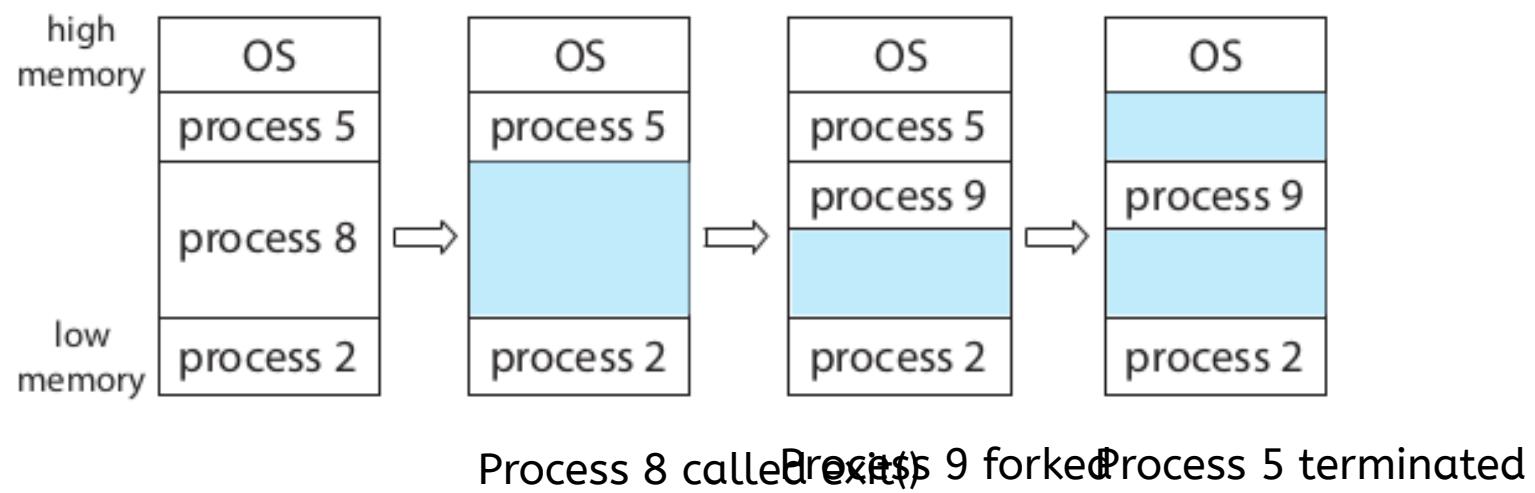


Figure 9.6 Hardware support for relocation and limit registers.

- Combined effect
 - “**Relocatable code**” – the process can go anywhere in RAM at the time of loading
 - Some memory violations can be detected – a memory access beyond base+limit will raise interrupt

Example scenario of memory in base+limit scheme



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

By doing extra work during “loading”.

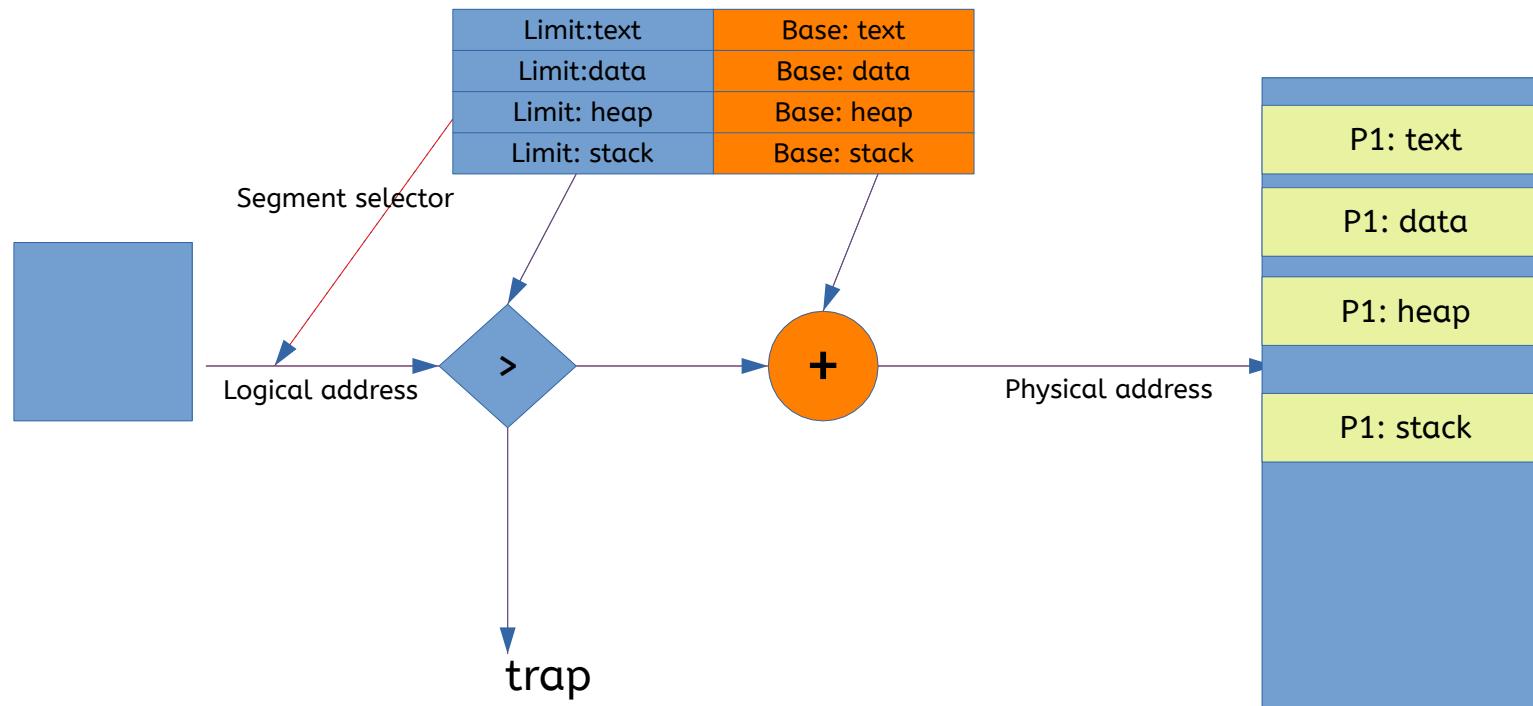
How?

Next scheme: Segmentation

Multiple base + limit pairs

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

Next scheme: Multiple base + limit pairs



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

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

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

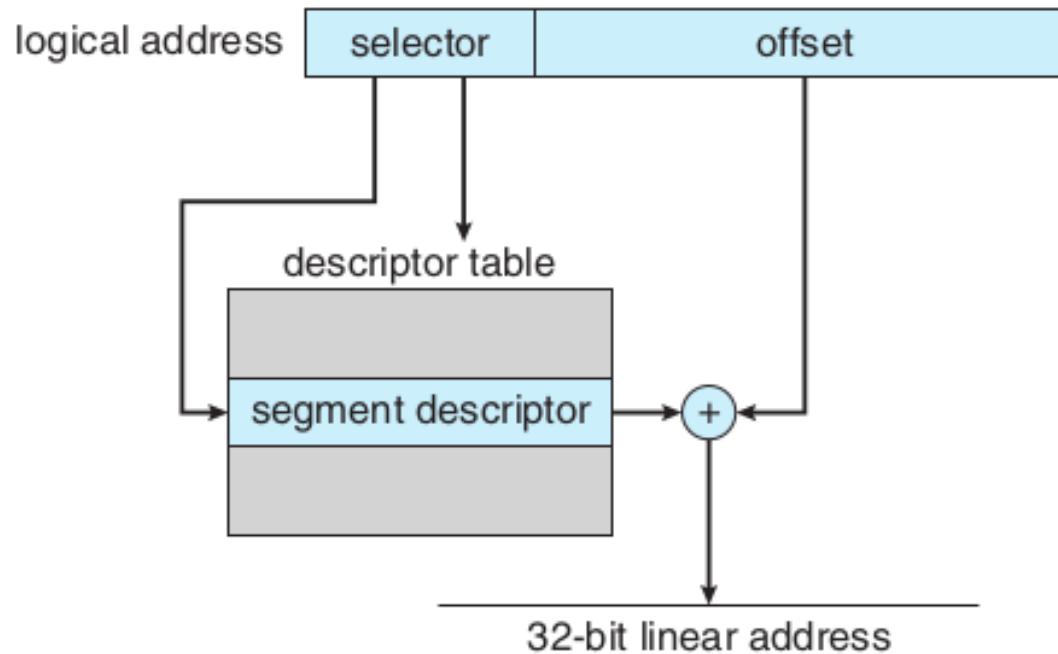


Figure 9.22 IA-32 segmentation.

Problems with segmentation schemes

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

Solving external fragmentation problem

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

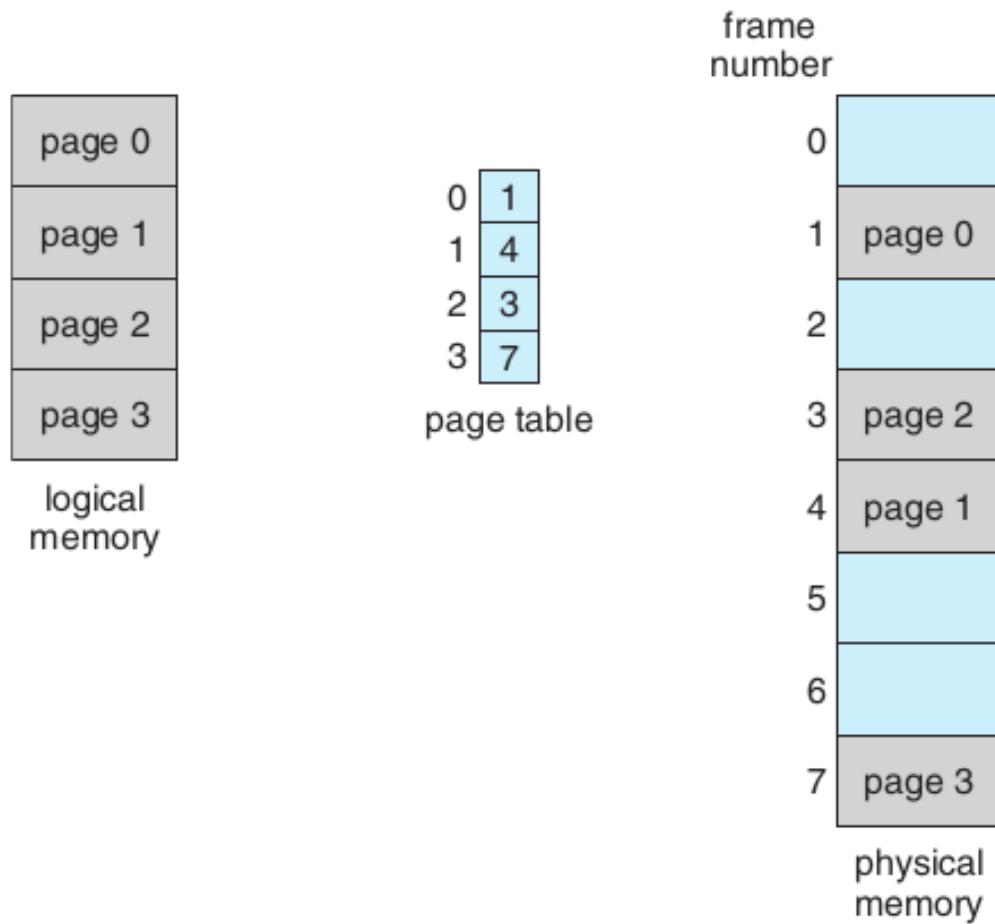


Figure 9.9 Paging model of logical and physical memory.

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

logical memory

0	5
1	6
2	1
3	2

page table

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

physical memory

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

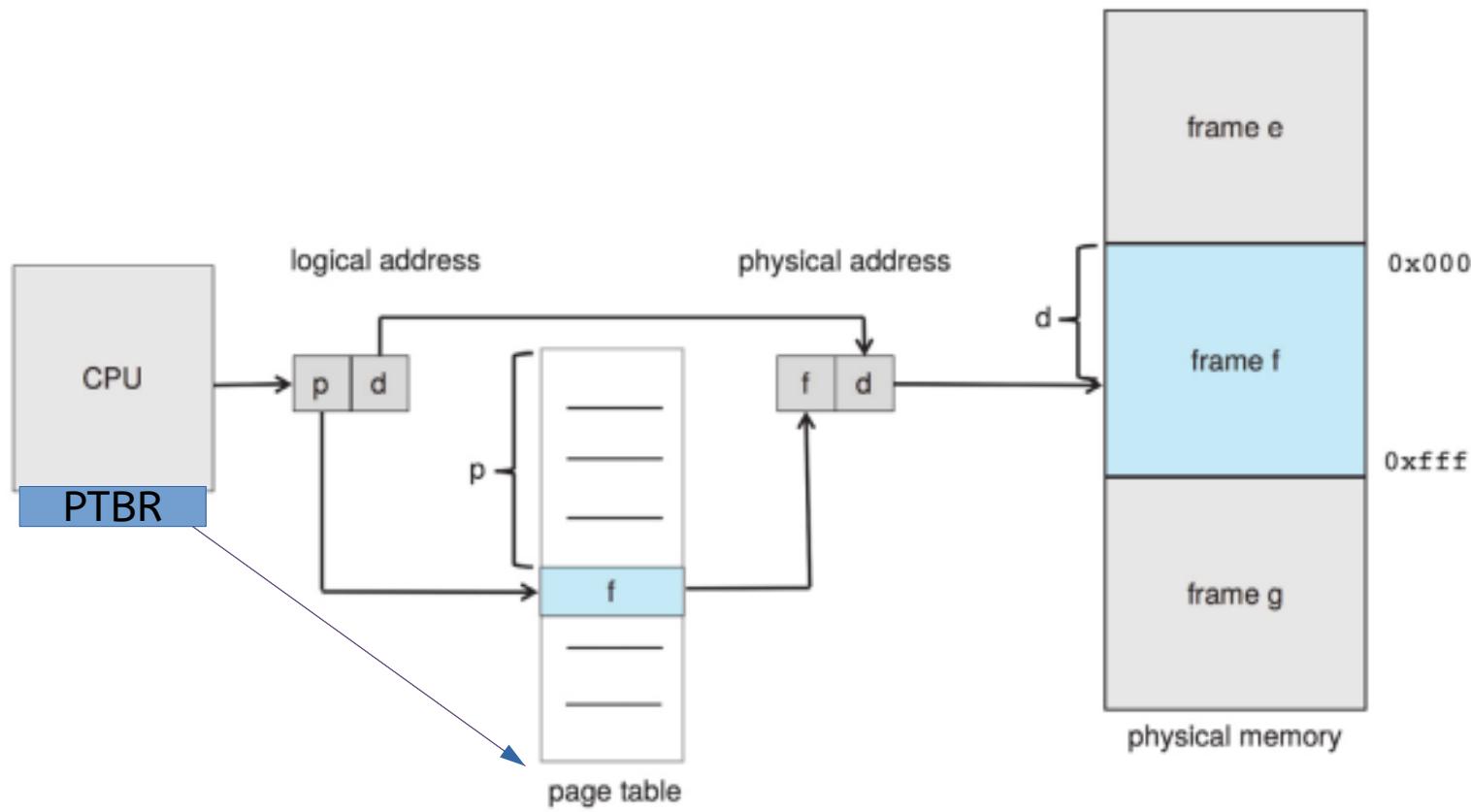


Figure 9.8 Paging hardware.

Paging

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

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

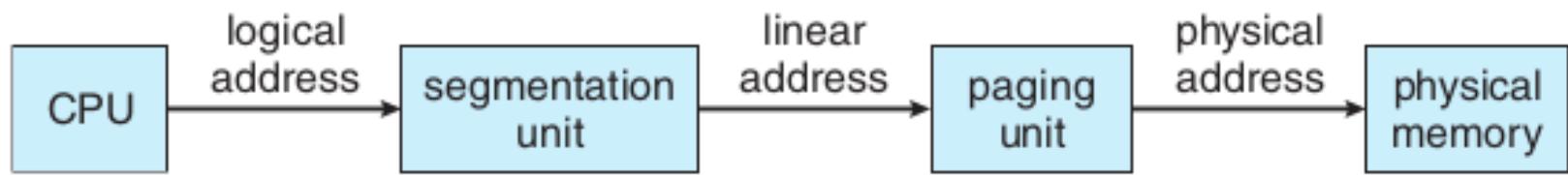


Figure 9.21 Logical to physical address translation in IA-32.

Segmentation in x86

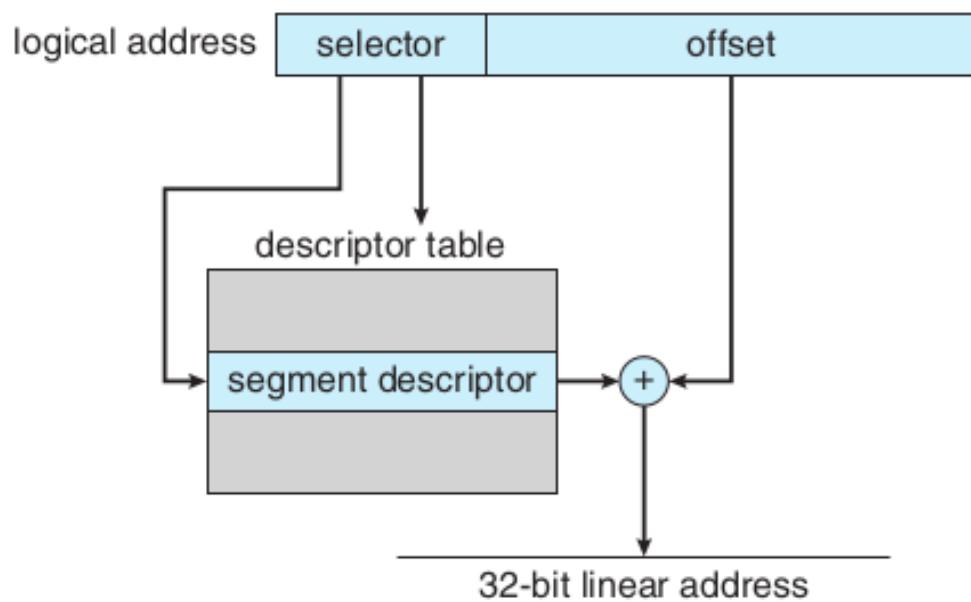


Figure 9.22 IA-32 segmentation.

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

Paging in x86

- 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

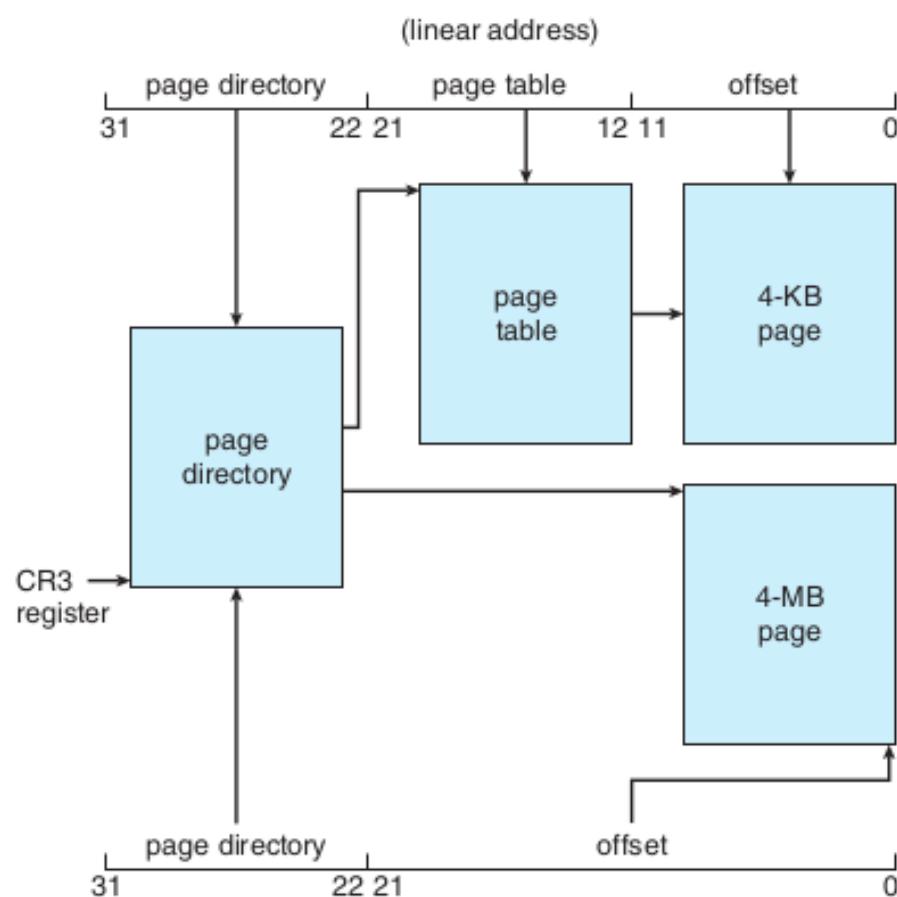


Figure 9.23 Paging in the IA-32 architecture.

Memory Management – Continued

More on Linking, Loading, Paging

Review of last class

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

More on Linking and Loading

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

Dynamic Linking

- **Linker is normally invoked as a part of compilation process**
 - Links
 - function code to function calls
 - references to global variables with “extern” declarations
- **Dynamic Linker**
 - Does not combine function code with the object code file
 - Instead introduces a “stub” code that is indirect reference to actual code
 - At the time of “loading” (or executing!) the

Dynamic Linking on Linux

```
#include <stdio.h>

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

~~PLT: Procedure Linkage Table~~

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

Output of objdump -x -D

Disassembly of section .text:

```
0000000000001189 <main>:
```

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

Disassembly of section .plt.got:

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

1080: .endbr64

```
1084:    bnd jmpq *0x2f3d(%rip)      #
3fc8 <printf@GLIBC\_2.2.5>
```

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

Dynamic Loading

- **Loader**

- Loads the program in memory
- Part of exec() code
- Needs to understand the format of the executable file (e.g. the ELF format)

- **Dynamic Loading**

- Load a part from the ELF file only if needed during execution
- Delayed loading
- Needs a more sophisticated memory management by operating system – to be seen during this series of lectures

Dynamic Linking, Loading

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

Continuous memory management

What is Continuous memory management?

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

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

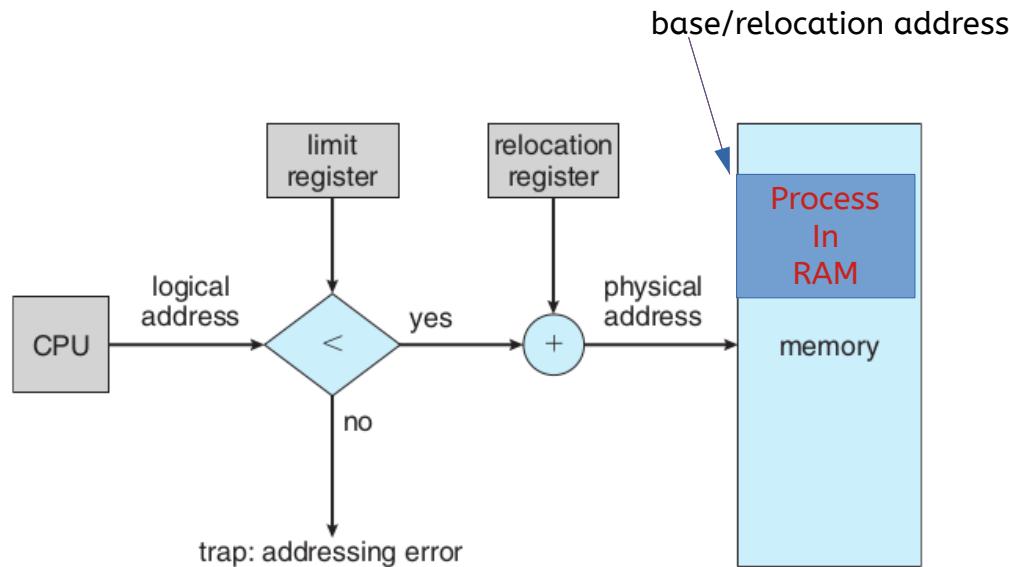


Figure 9.6 Hardware support for relocation and limit registers.

Problems faced by OS

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

Variable partition scheme

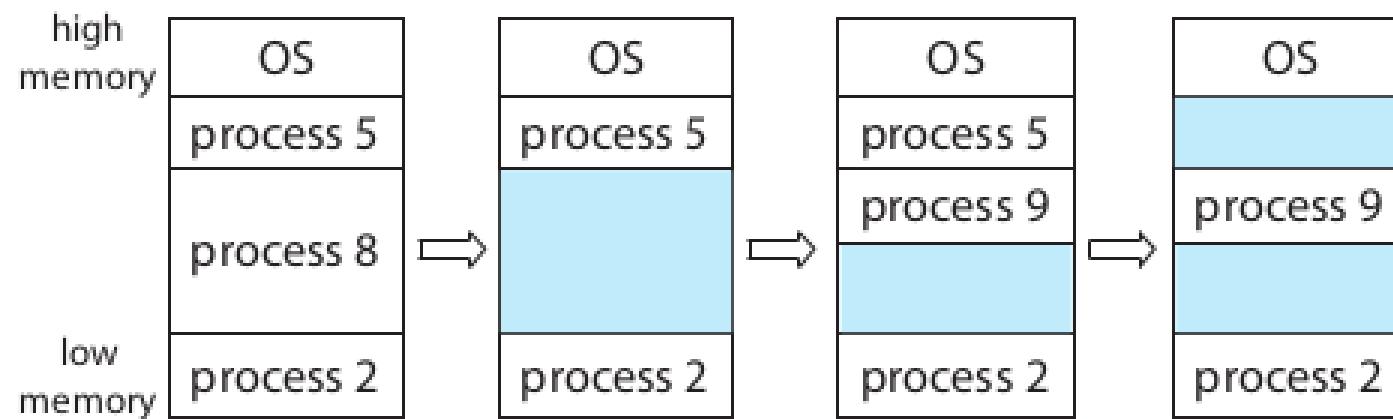


Figure 9.7 Variable partition.

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

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

Strategies for finding a free chunk

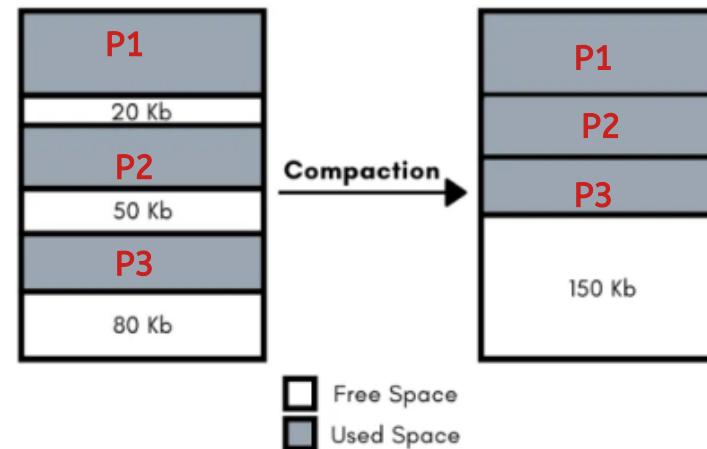
- 6k, 17k, 16k, 40k holes . Need 15k.
- **Best fit:** Find the smallest hole, larger than process. **Ans: 16k**
- **Worst fit:** Find the largest hole. **Ans: 40k**
- **First fit:** Find the “first” hole larger than the process. **Ans: 17k**

Problem : External fragmentation

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

Solution to external fragmentation

- Compaction !
- OS moves the process chunks in memory to make available continuous memory region
 - Then it must update the memory management information in PCB (e.g. base of the process) of each process



Another solution to external fragmentation: Fixed size partitions

- **Fixed partition scheme**
- **Memory is divided by OS into chunks of equal size: e.g., say, 50k**
 - If total 1M memory, then 20 such chunks
- **Allocate one or more chunks to a process, such that the total size is \geq**



Fixed partition scheme

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

Solution to internal fragmentation

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

Paging

An extended version of fixed size partitions

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

Logical address seen as



Paging hardware

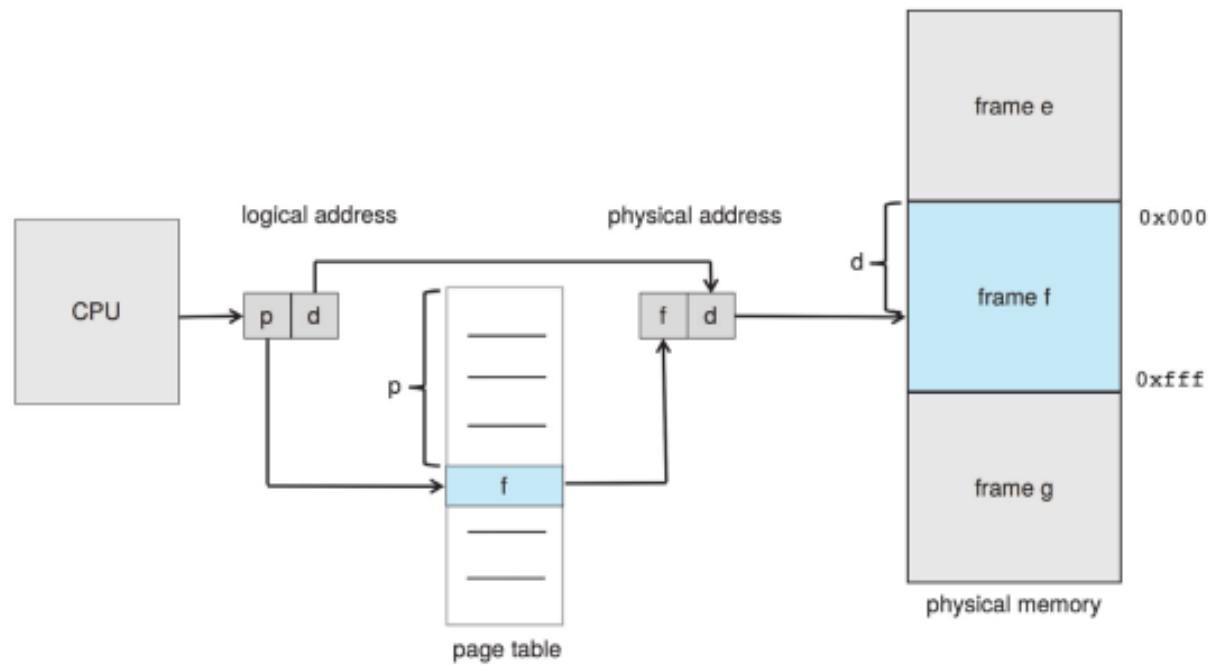


Figure 9.8 Paging hardware.

MMU's job

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

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

(Page table location is stored in a hardware register

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

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

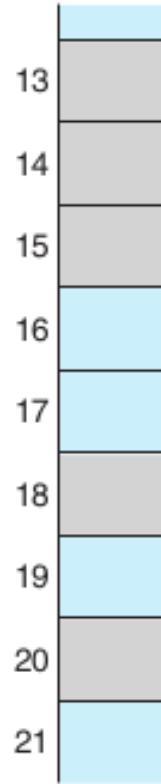
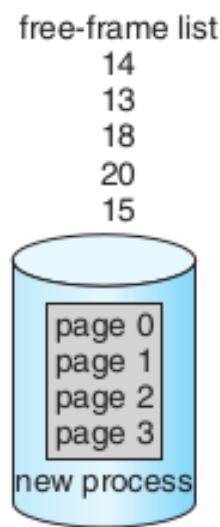
3. Replace the page number p in the logical address with the frame number f .

Job of OS

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

Job of OS

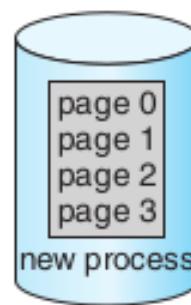
- **Maintain a list of all page frames**
 - Allocated frames
 - Free Frames (called frame table)
 - Can be done using simple linked list
 - Innovative data structures can also be used to maintain free and allocated frames list (e.g. xv6 code)



(a)

free-frame list

15



new-process page table

0	14
1	13
2	18
3	20

free-frame list

13 page 1

14 page 0

15

16

17

18 page 2

19

20 page 3

21

(b)

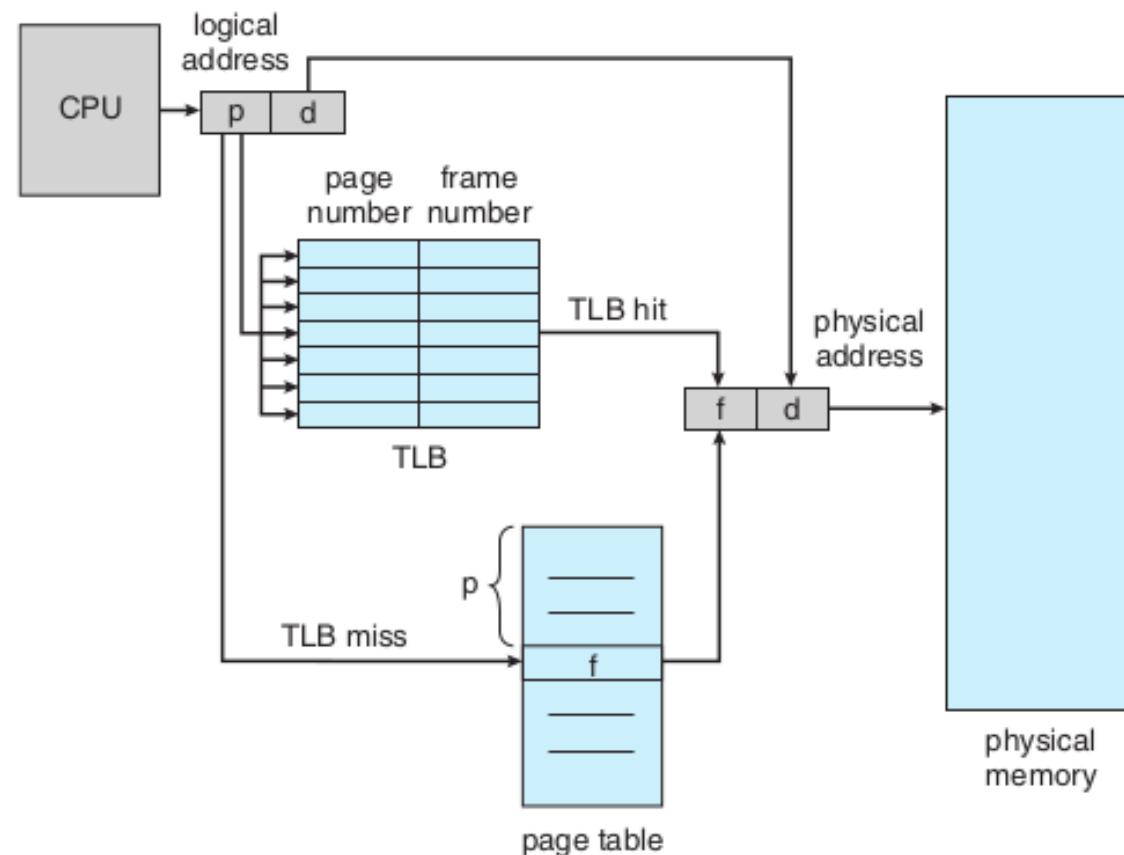
Figure 9.11 Free frames (a) before allocation and (b) after allocation.

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

$$\text{effective access time} = 0.80 \times 10 + 0.20 \times 20 \\ = 12 \text{ nanoseconds}$$

Memory protection with paging

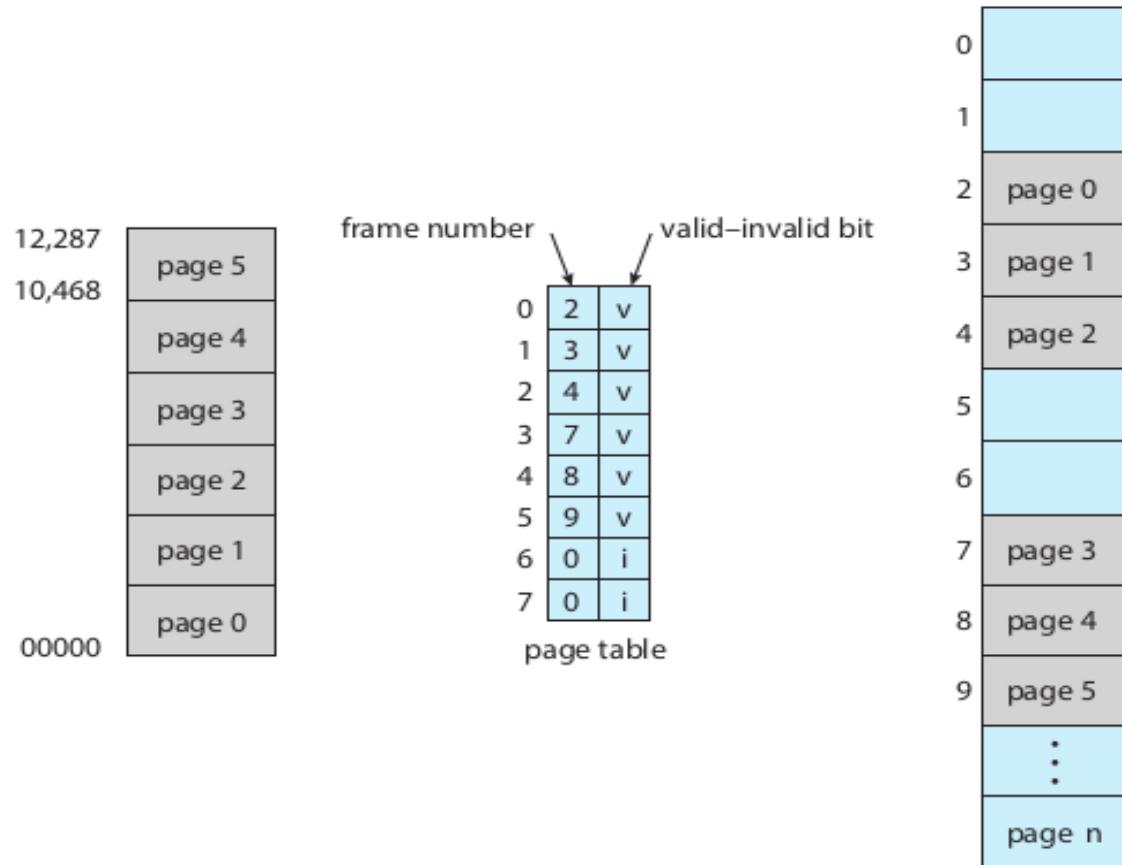


Figure 9.13 Valid (v) or invalid (i) bit in a page table.

X86 PDE and PTE

31

Page table physical page number

	12	11	10	9	8	7	6	5	4	3	2	1	0
A V L	A	G	P S	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

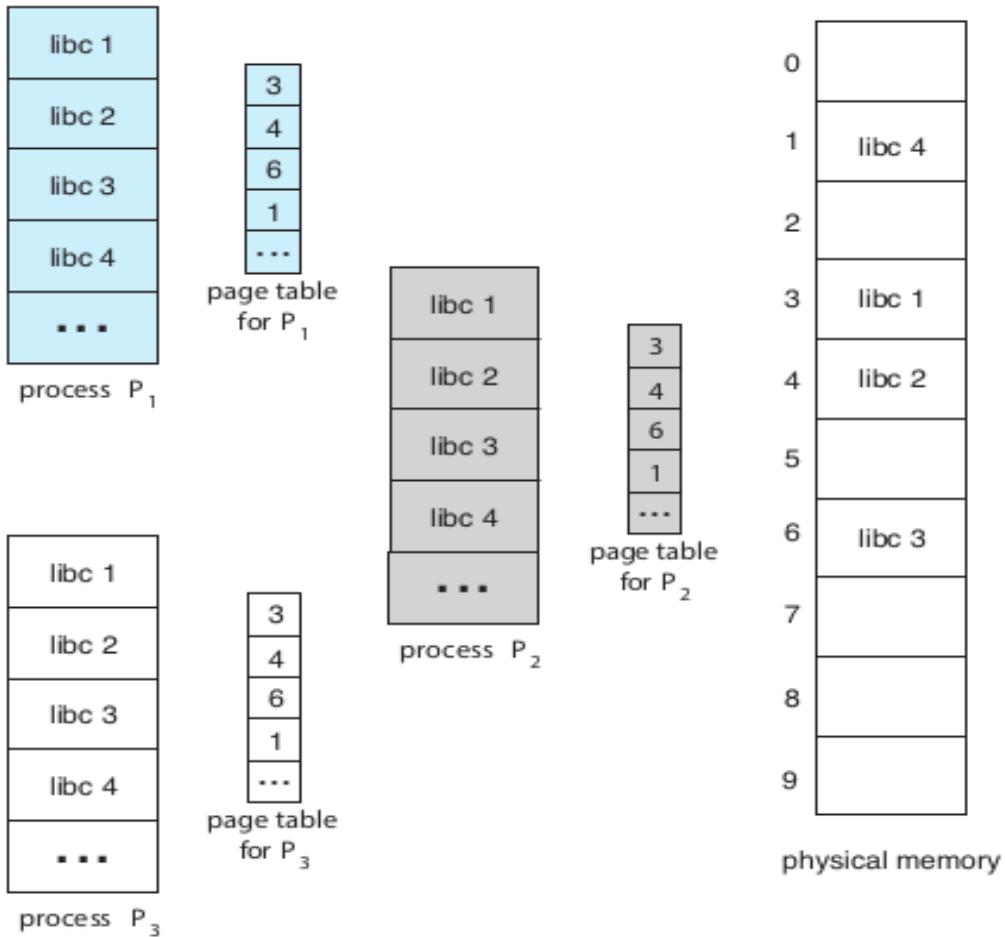
AVL Available for system use

31

Physical page number

	12	11	10	9	8	7	6	5	4	3	2	1	0
A V L	A	G	P A T	D	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 $2^{12} = 4$ 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

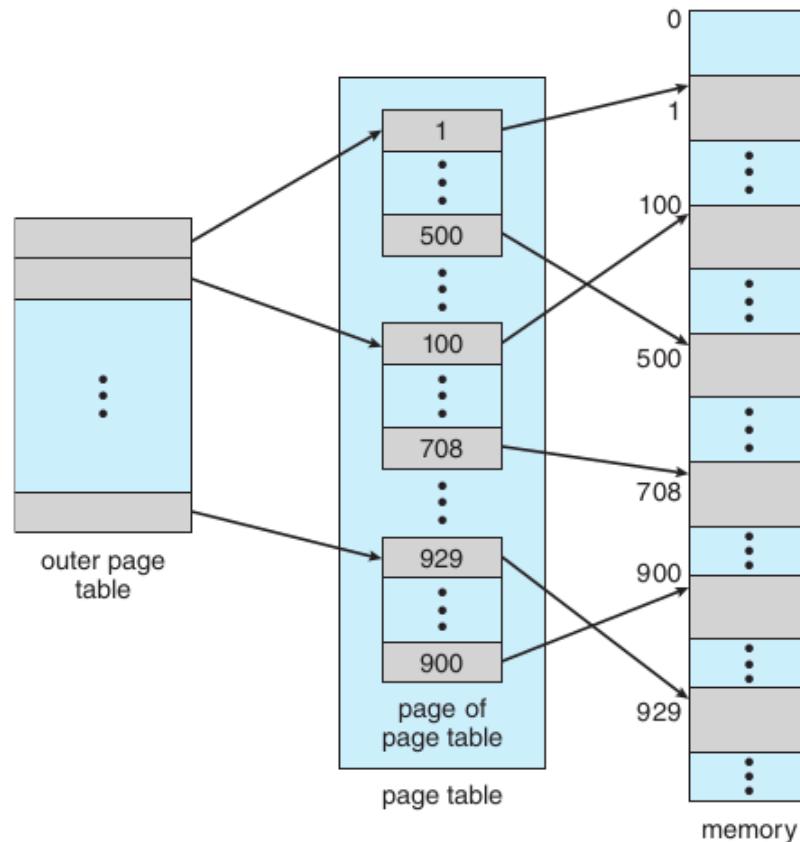
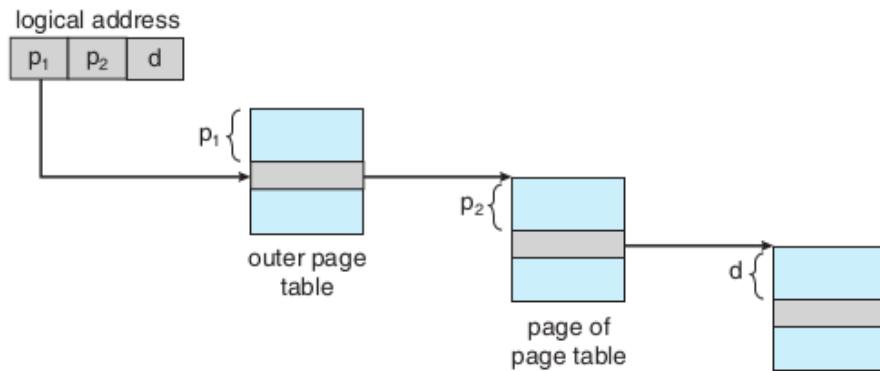
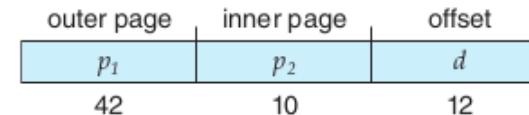


Figure 9.15 A two-level page-table scheme.



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

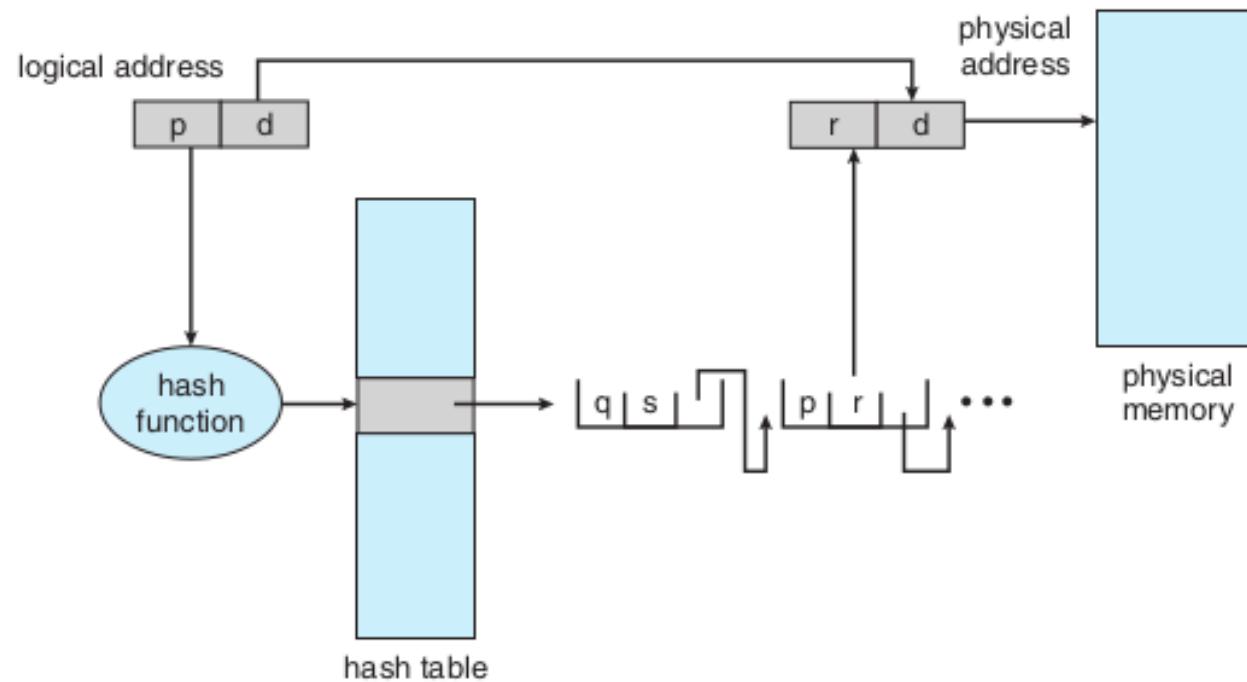


Figure 9.17 Hashed page table.

Inverted page table

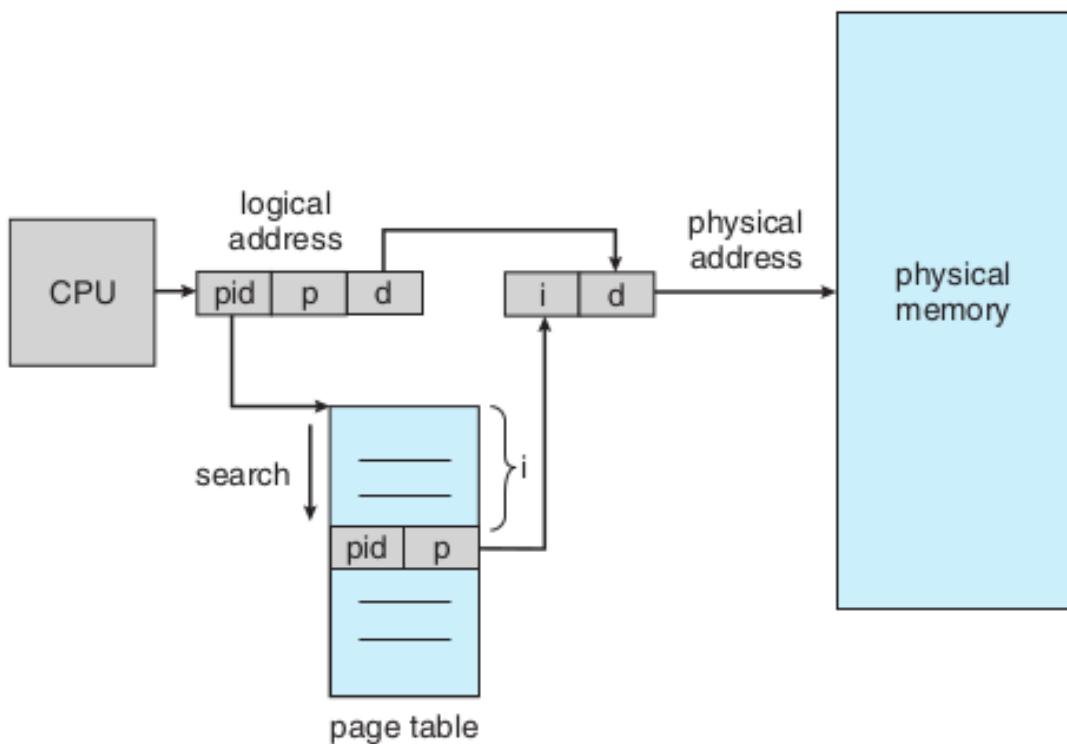


Figure 9.18 Inverted page table.

Normal page table – one per process --

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

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

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

Case Study: Oracle SPARC Solaris

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

Case Study: Oracle SPARC Solaris

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

Case Study: Oracle SPARC Solaris

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

Swapping

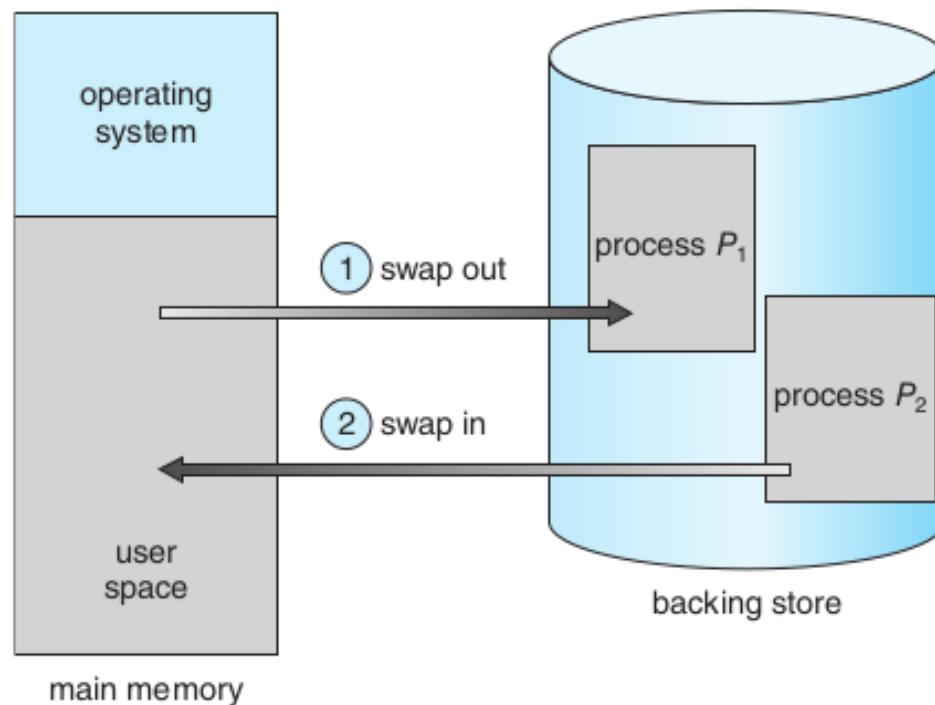


Figure 9.19 Standard swapping of two processes using a disk as a backing store.

Swapping

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

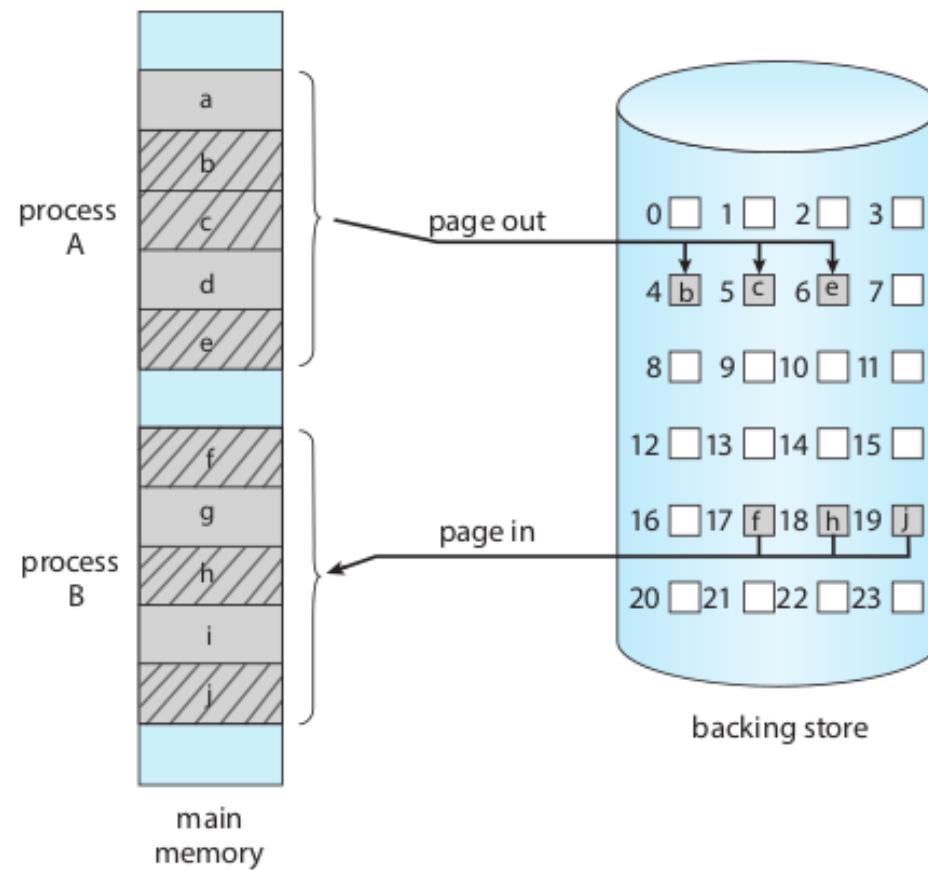


Figure 9.20 Swapping with paging.

Words of caution about ‘paging’

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

XV6 bootloader

Abhijit A. M.

abhijit.comp@coep.ac.in

Credits:

xv6 book by Cox, Kaashoek, Morris

Notes by Prof. Sorav Bansal

A word of caution

We begin reading xv6 code

But it's not possible to read this code in a
“linear fashion”

The dependency between knowing OS concepts
and reading/writing a kernel that is written
using all concepts

What we have seen

Compilation process, calling conventions

Basics of Memory Management by OS

Basics of x86 architecture

Registers, segments, memory management unit, addressing, some basic machine instructions,

ELF files

Objdump, program headers

Symbol tables

Boot-process

Bootloader itself

Is loaded by the BIOS at a fixed location in memory and BIOS makes it run

Our job, as OS programmers, is to write the bootloader code

Bootloader does

Pick up code of OS from a ‘known’ location and loads it in memory

Makes the OS run

Xv6 bootloader: bootasm.S bootmain.c (see Makefile)

bootloader

BIOS Runs (automatically)

Loads boot sector into RAM at 0x7c00

Starts executing that code

Make sure that your bootloader is loaded at 0x7c00

Makefile has

bootblock: bootblock.S bootmain.c

**$\$(CC) \$(CFLAGS) -fno-pic -nostdinc -I.$ -c
bootasm.S**

...

$\$(LD) \$(LDFLAGS) -N -e start -Ttext 0x7C00 -o$

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:

```
inb $0x64,%al # Wait  
for not busy
```

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

After this

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

And further code runs in protected mode

Real mode Vs protected mode

Real mode 16 bit registers

Protected mode

Enables segmentation + Paging both

No longer seg*16+offset calculations

Segment registers is index into segment descriptor table. But segment:offset pairs continue

`mov %esp, $32 # SS will be used with esp`

More in next few slides

Other segment registers need to be explicitly mentioned in instructions

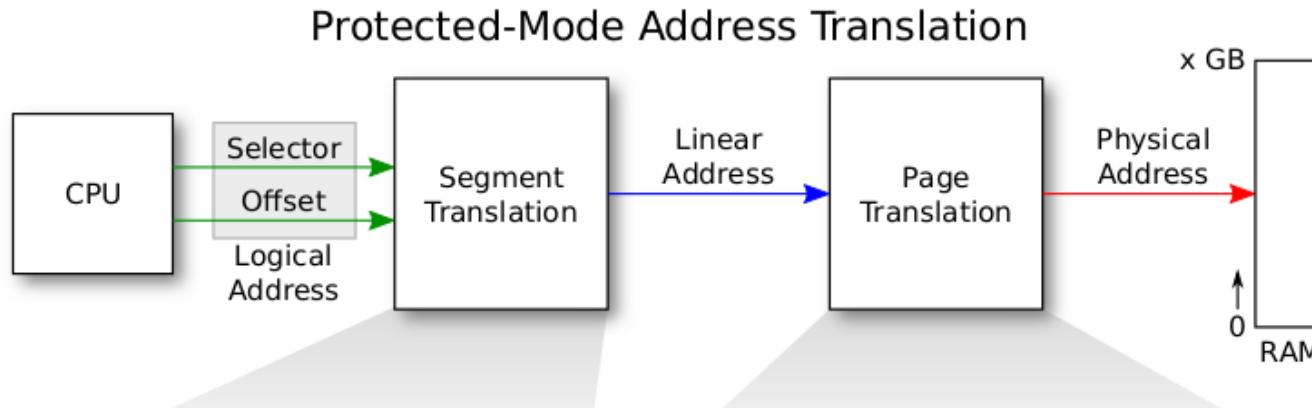
`Mov FS:$200, 30`

32 bit registers

can address upto 2^{32} memory

Can do arithmetic in 32 bits

X86 address : protected mode address translation



Both Segmentation and Paging are used in x86
X86 allows optionally one-level or two-level paging
Segmentation is a must to setup, paging is optional (needs to be enabled)
Hence different OS can use segmentation+paging in different ways

X86 segmentation

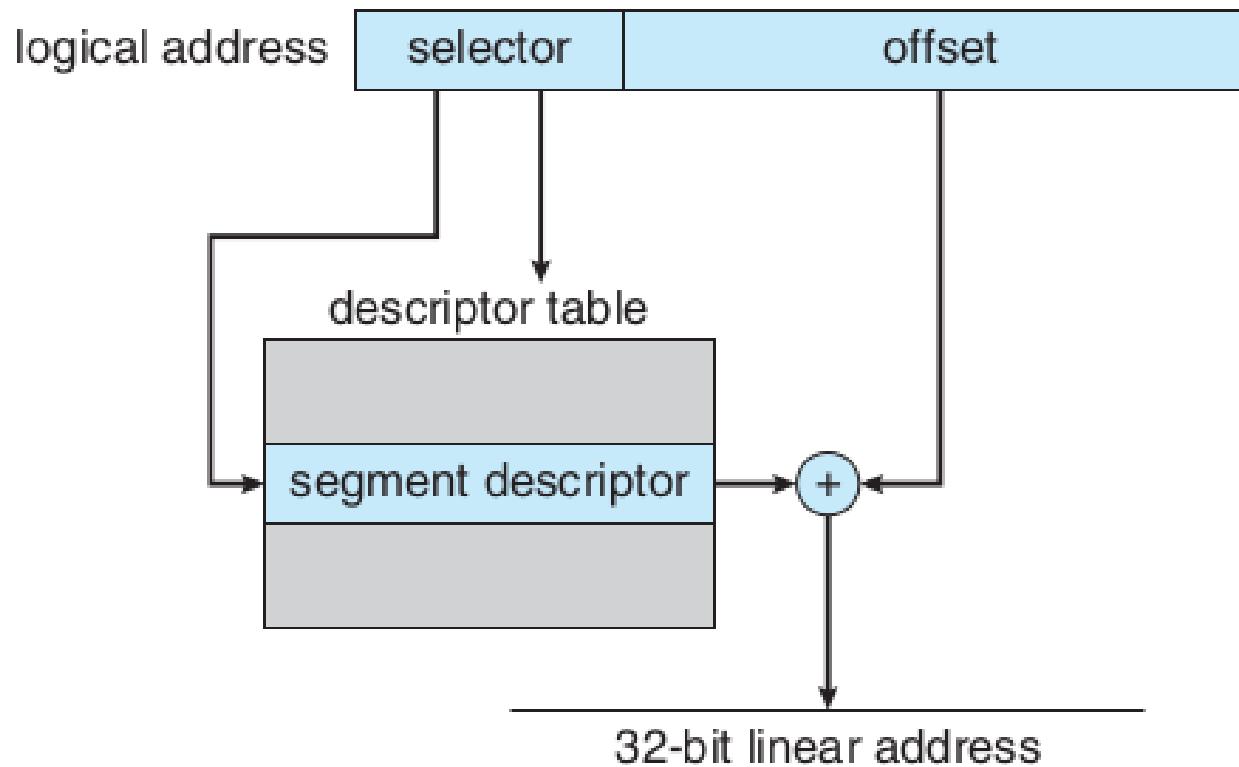


Figure 8.22 IA-32 segmentation.

Paging concept, hierarchical paging

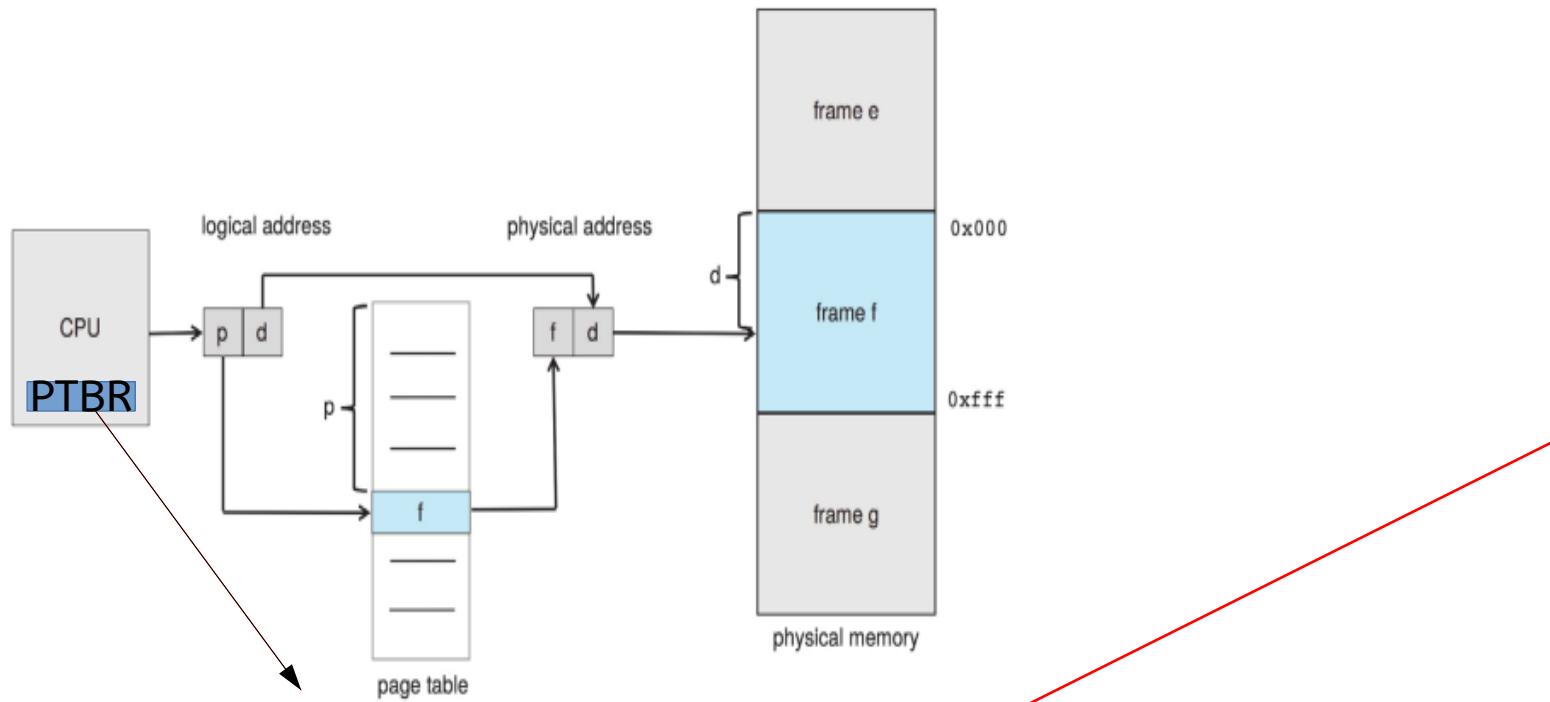


Figure 9.8 Paging hardware.

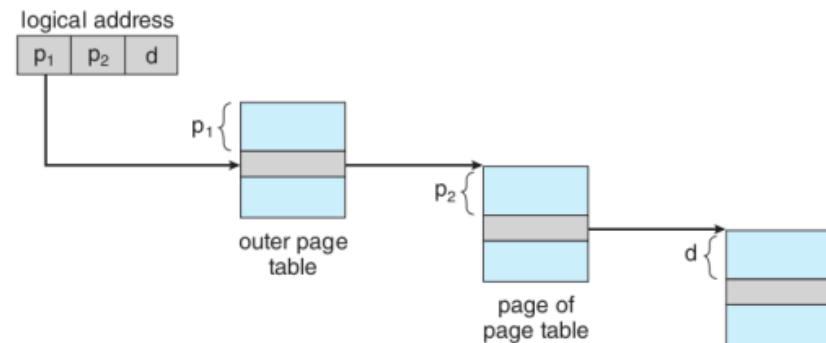


Figure 9.16 Address translation for a two-level 32-bit paging architecture.

x86 paging

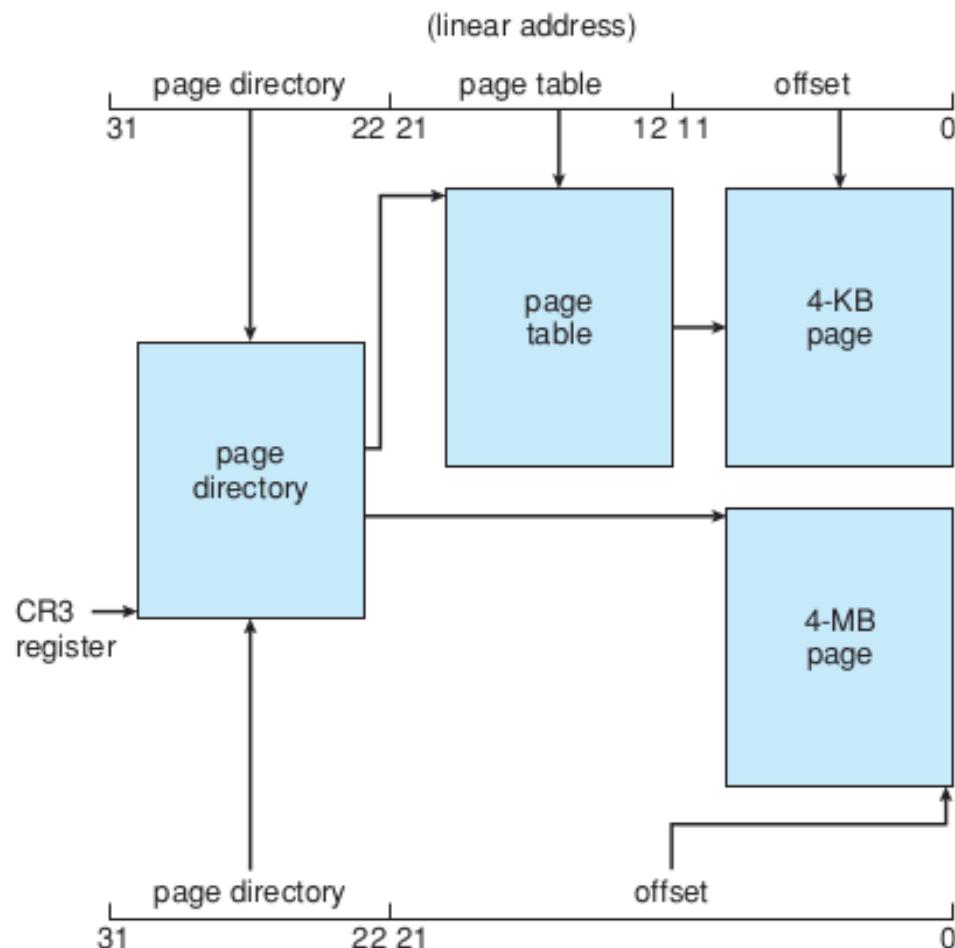


Figure 8.23 Paging in the IA-32 architecture.

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

31

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

PTE

CR3

	31	12 11	5 4	3	2	0
CR3	Page-Directory-Table Base Address		P C D	P W T		

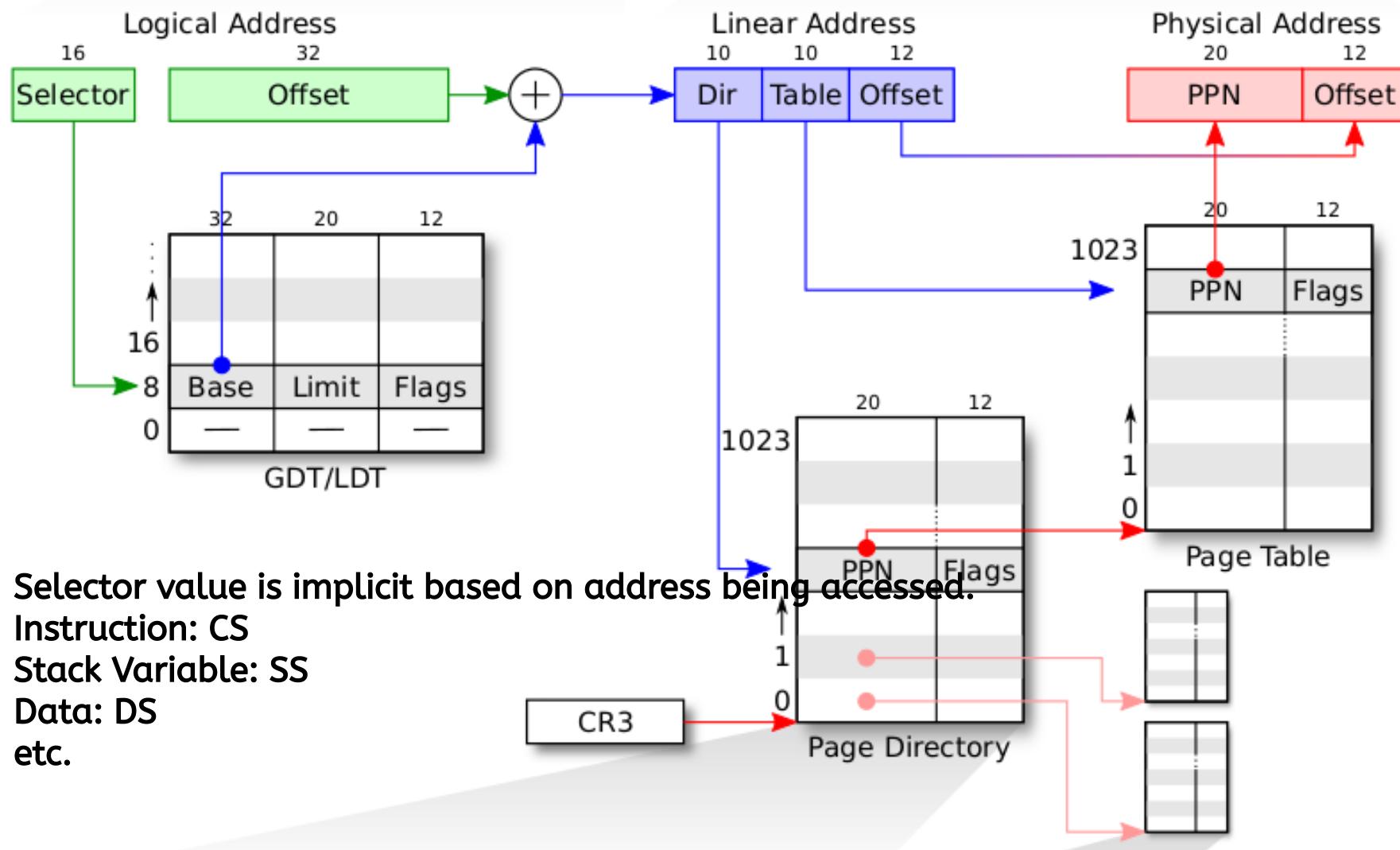
PWT Page-level writes transparent PCT Page-level cache disable

CR4

	31	11 10 9 8 7 6 5 4 3 2 1 0
CR4		O S X M O S C E O P G E E M P C A E P D S E T D P V M I E

VME Virtual-8086 mode extensions MCE Machine check enable
PVI Protected-mode virtual interrupts PGE Page-global enable
TSD Time stamp disable PCE Performance counter enable
DE Debugging extensions OSFXSR OS FXSAVE/FXRSTOR support
PSE Page size extensions OSXMM- OS unmasked exception support
PAE Physical-address extension EXCPT

Segmentation + Paging



Segmentation + Paging setup of xv6

xv6 configures the segmentation hardware by setting Base = 0, Limit = 4 GB

translate logical to linear addresses without change, so that they are always equal.

Segmentation is practically off

Once paging is enabled, the only interesting address mapping in the system will be linear to physical.

In xv6 paging is NOT enabled while loading kernel

After kernel is loaded 4 MB pages are used for a while

Later the kernel switches to 4 kB pages!

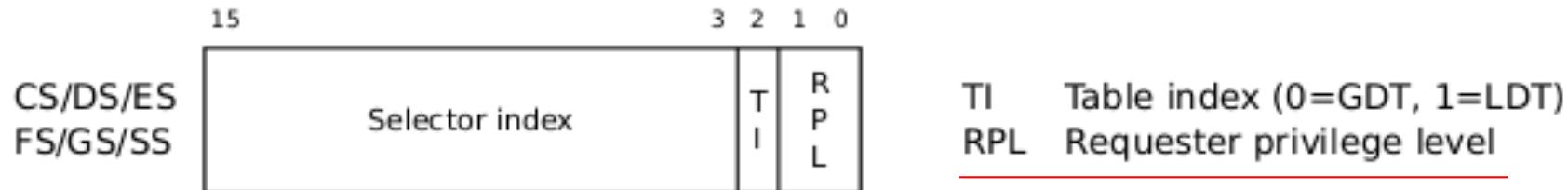
GDT Entry

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 <hr/> Base 16:23

asm.h

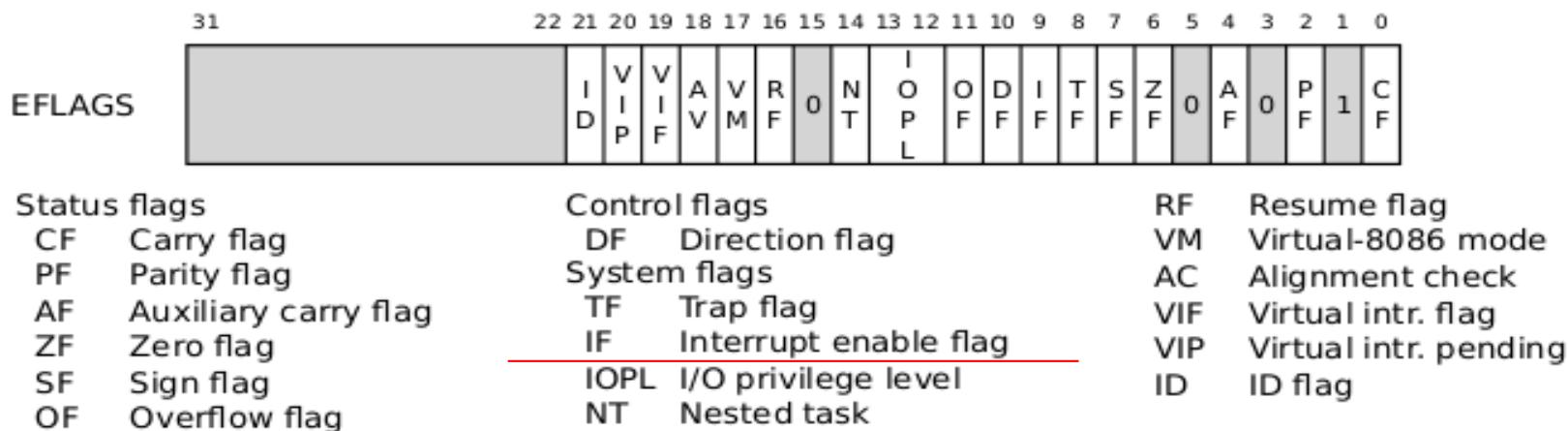
```
#define SEG_ASM(type,base,lim) \
.word (((lim) >> 12) & 0xffff), ((base) & 0xffff); \
.byte (((base) >> 16) & 0xff), (0x90 | (type)), \
(0xC0 | (((lim) >> 28) & 0xf)), (((base) >> 24) & 0xff)
```

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

SEG_ASM(STA_W, 0x0,
0xffffffff)

data seg

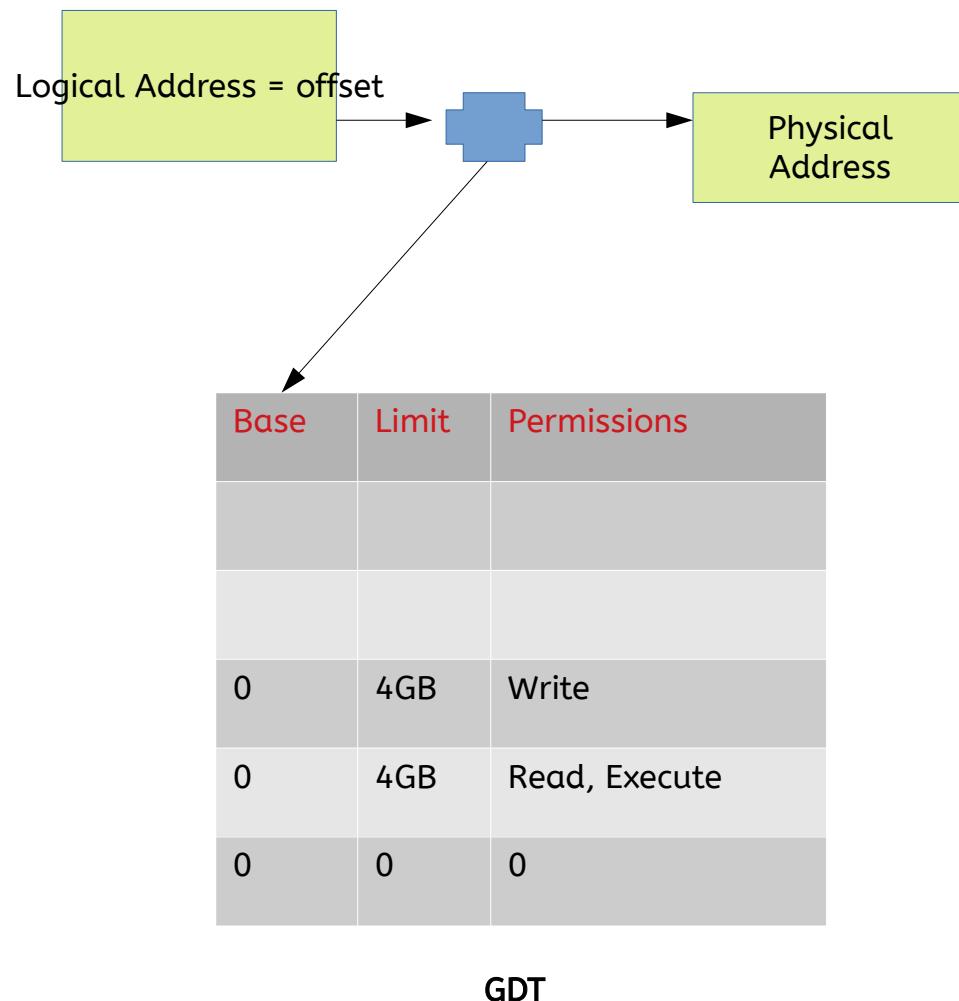
lgdt

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

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
Addresses in “kernel” file translate to s

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
```

CRO

CRO	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 S E 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.

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 is 0 / CS setting)

Jumping to “C” code

**movw
\$(SEG_KDATA<<3),
%ax # Our data
segment selector**

**movw %ax, %ds # ->
DS: Data Segment**

**movw %ax, %es # ->
ES: Extra Segment**

**movw %ax, %ss # ->
SS: Stack Segment**

**movw \$0, %ax # Zero
segments not ready**

**Setup Data, extra,
stack segment with
SEG_KDATA (=2), FS
& GS (=0)**

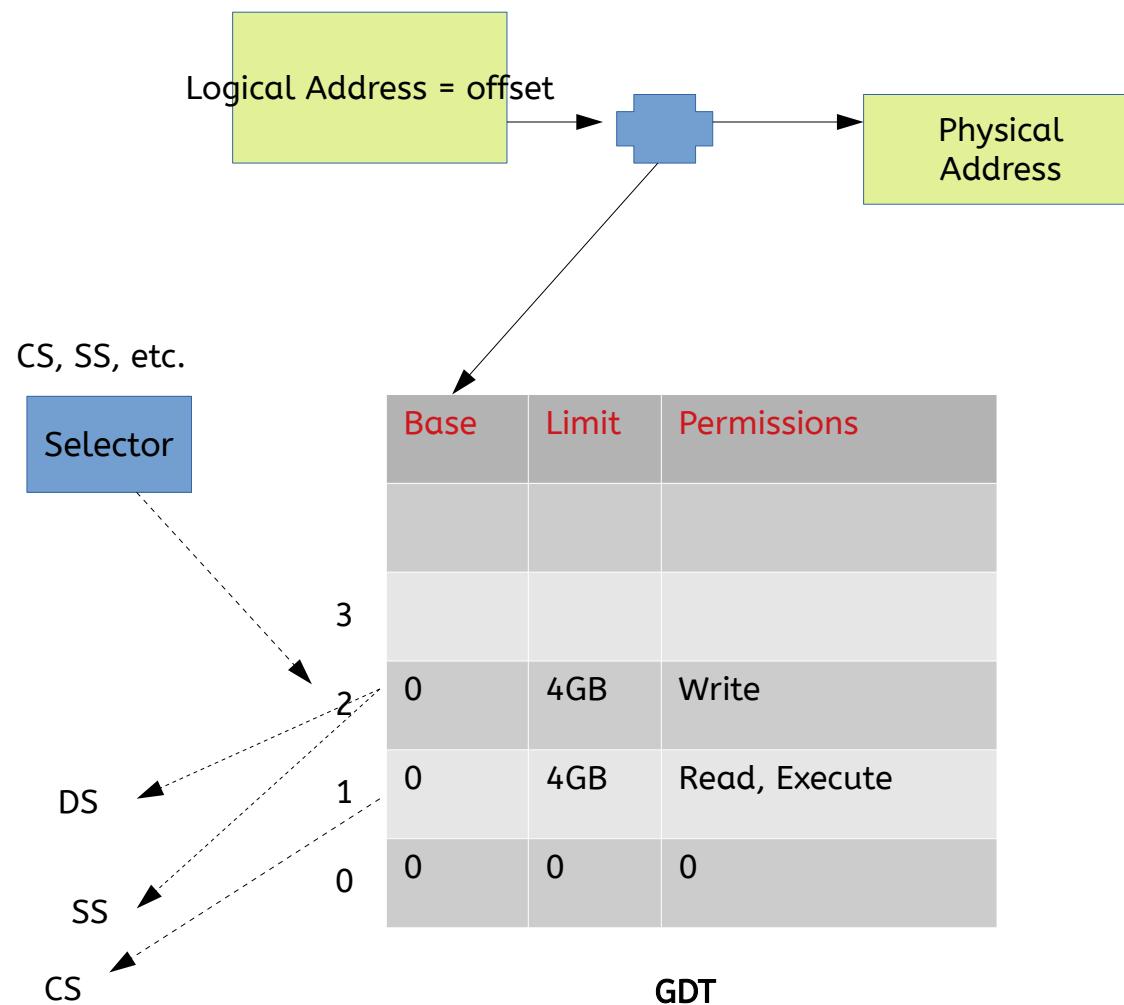
**Copy “\$start” i.e.
7c00 to stack-ptr**

**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 void
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-**

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

    elf = (struct
            elfhdr*)0x10000; //
```

bootmain()

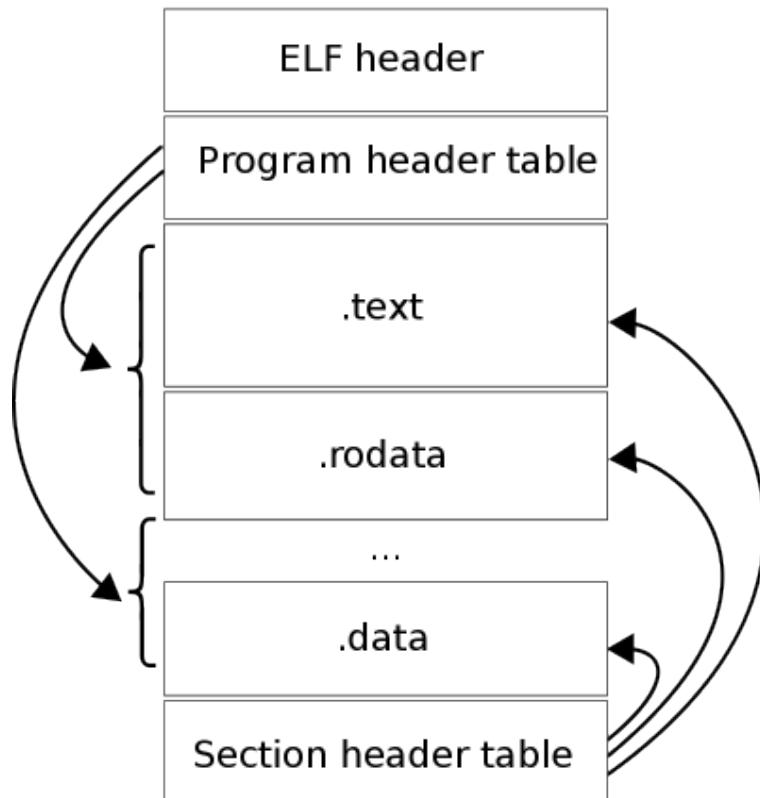
Check if it's really
ELF or not

Next load kernel
code from ELF file
“kernel” into
memory

// Is this an ELF
executable?

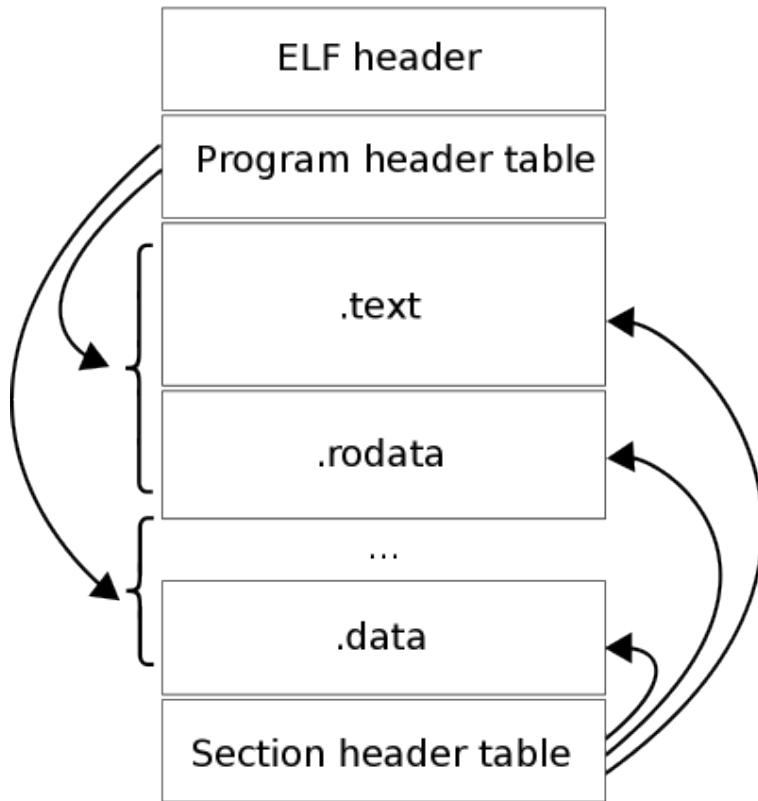
```
if(elf->magic !=  
ELF_MAGIC)  
  
return; // let  
bootasm.S handle  
error
```

ELF



```
struct elfhdr {  
    uint magic; // must  
    equal ELF_MAGIC  
    uchar elf[12];  
    ushort type;  
    ushort machine;  
    uint version;  
    uint entry;  
    uint phoff; // where  
    is 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.  
    uint paddr; //
```

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

Diff between memsz &
Code to be loaded at KERNBASE + KERNLI

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 0x0000a516 memsz 0x000154a8 flags rwx

STACK off 0x00000000 vaddr 0x00000000 paddr 0x00000000 align 2**4

 filesz 0x00000000 memsz 0x00000000 flags rwx

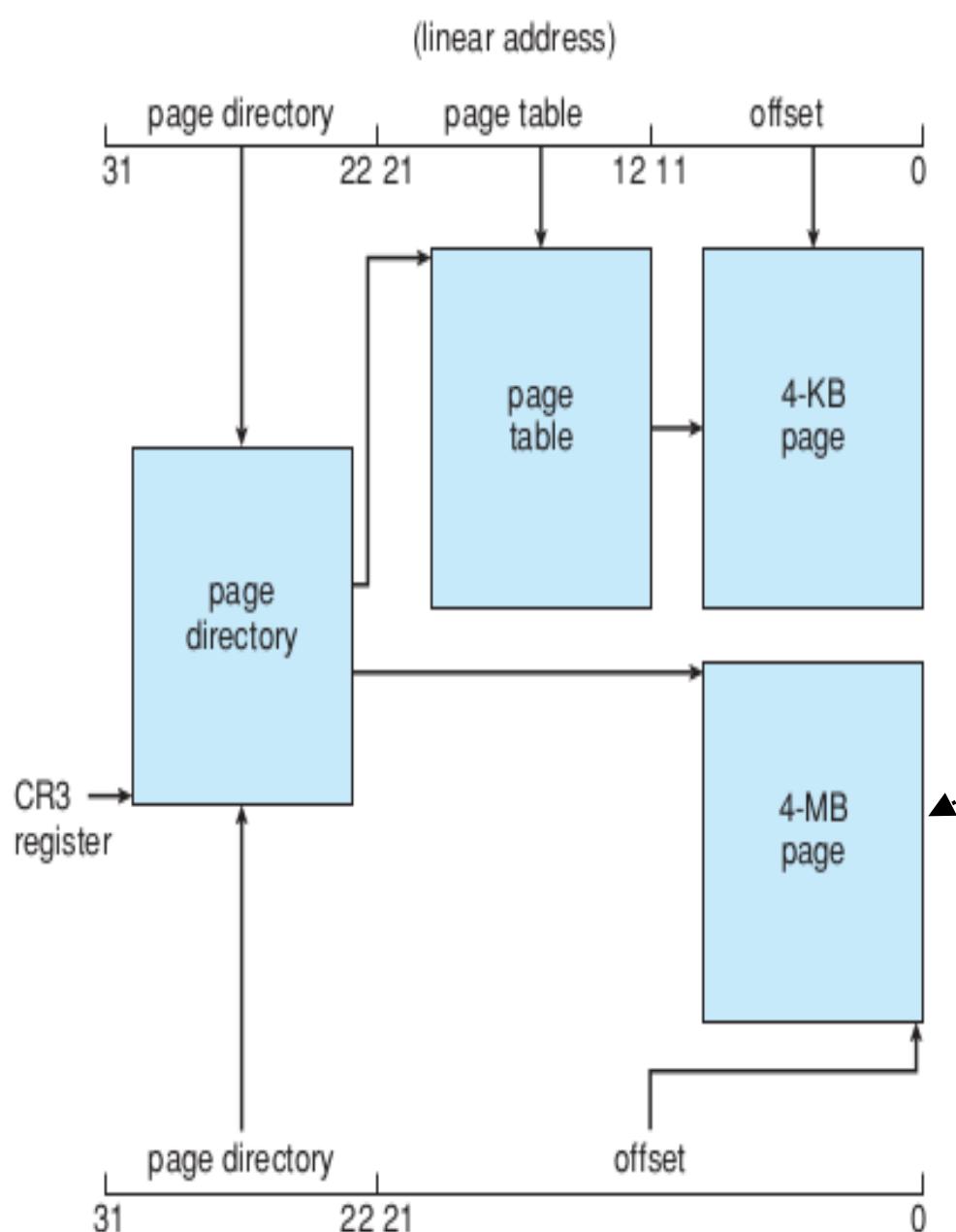
Stack : everything zeroes

Load code from ELF to memory

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

Jump to Entry

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

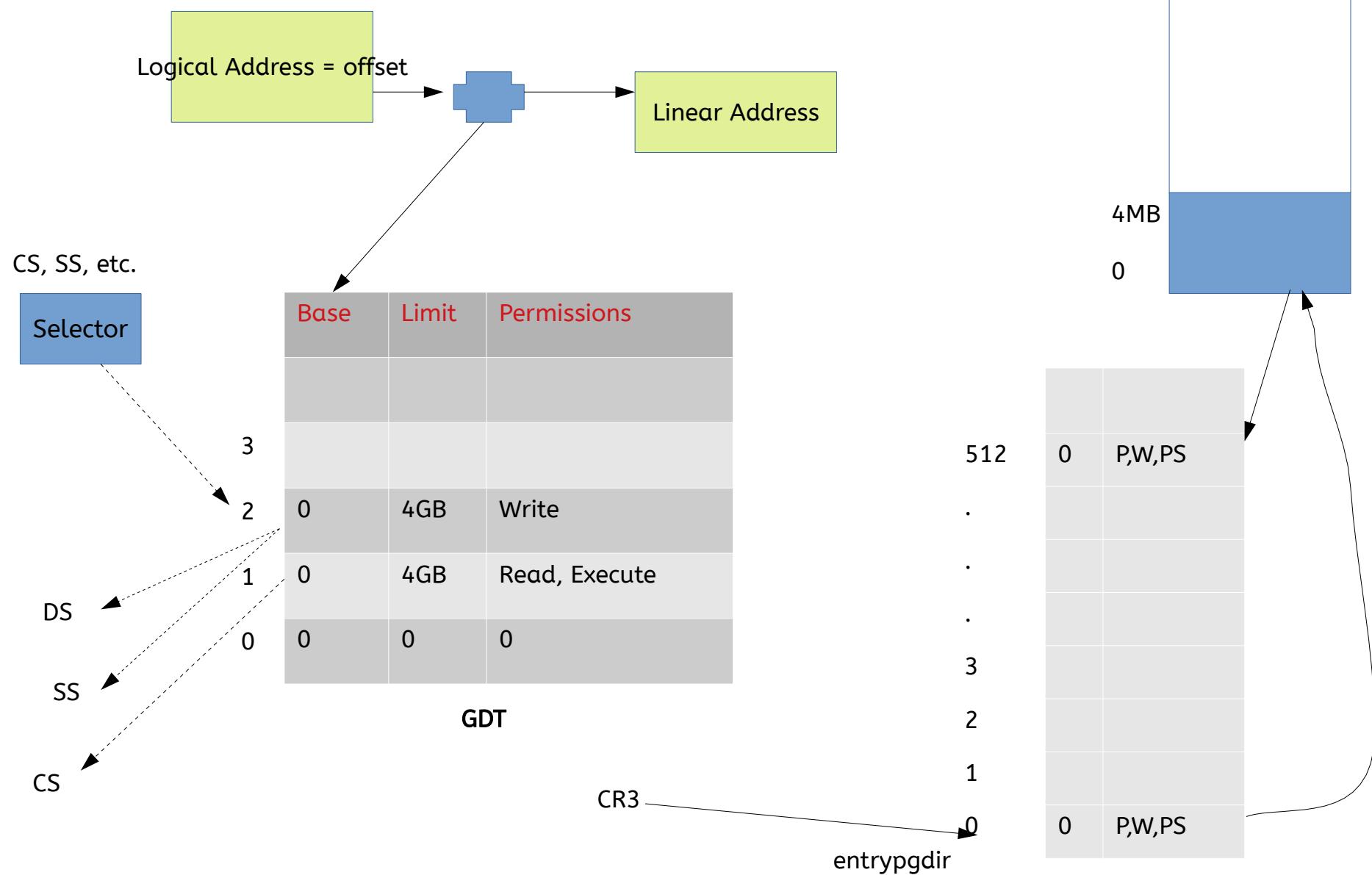


To understand further code

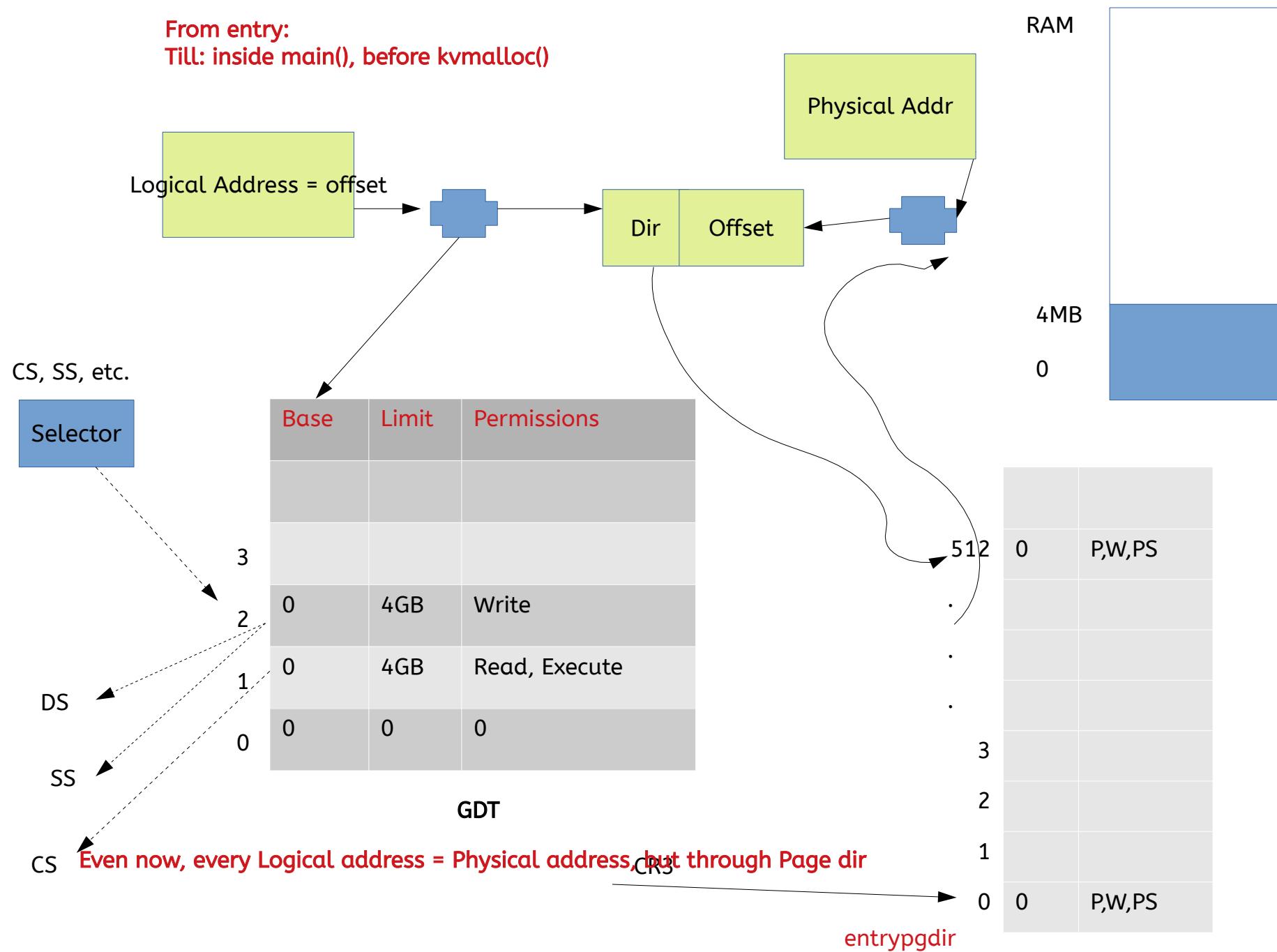
Remember: 4 MB pages are possible

Figure 8.23 Paging in the IA-32 architecture.

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



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



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

```
__attribute__((aligned(PGSIZE)))
pde_t entrypgdir[NPDENTRIES] = {

    // Map VA's [0, 4MB) to PA's [0, 4MB)
    [0] = (0) | PTE_P | PTE_W | PTE_PS,

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

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

This mapping restricts the kernel instructions and data to 4 Mbytes.

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

entry() in entry.S

entry:

```
    movl %cr4, %eax  
    orl $(CR4_PSE),  
    %eax  
    movl %eax, %cr4  
    movl $(V2P_WO(entrypgdir)),  
    %eax  
    movl %eax, %cr3  
    movl %cr0, %eax  
    orl $(CR0_PG|CR0_WP),
```

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

More about entry()

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

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

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

More about entry()

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

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

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

entry()

```
movl $(stack +  
KSTACKSIZE), %esp  
  
mov $main, %eax  
jmp *%eax  
  
.comm stack,  
KSTACKSIZE
```

Abhijit: allocate here 'stack' of
size = KSTACKSIZE

Set up the stack
pointer.

Abhijit:
+KSTACKSIZE is done
as stack grows
downwards

Jump to main(),
and switch to
executing at high
addresses. The
indirect call is
needed because the
assembler produces

bootmasm.S bootmain.c: Steps

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

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

Processes

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

Process related data structures in kernel code

- Kernel needs to maintain following types of data structures for managing processes
 - List of all processes
 - Memory management details for each, files opened by each etc.
 - Scheduling information about the process

Status of the process

Process Control Block



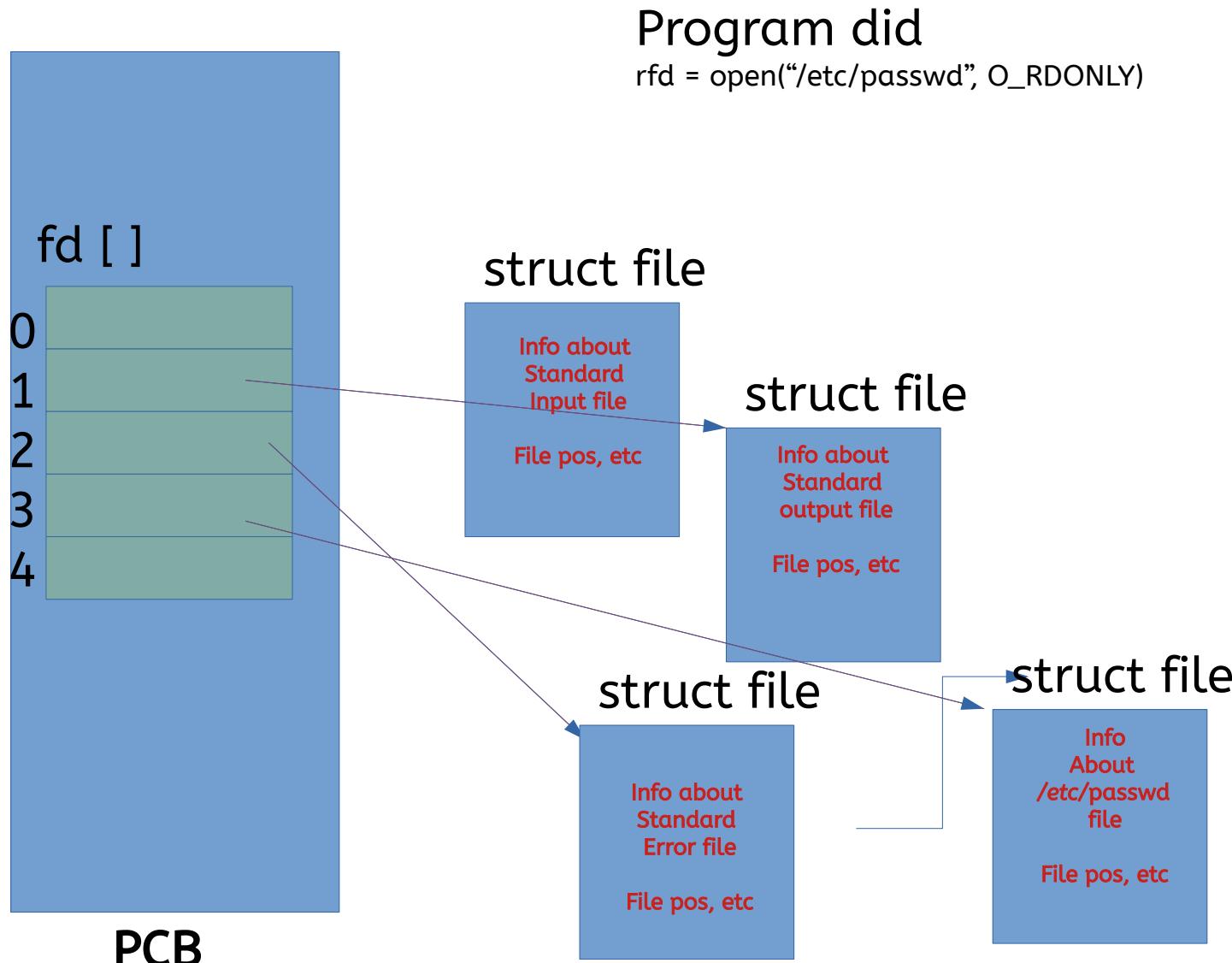
Figure 3.3 Process control block (PCB).

Fields in PCB



Figure 3.3 Process control block (PCB).

List of open files



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 };
```

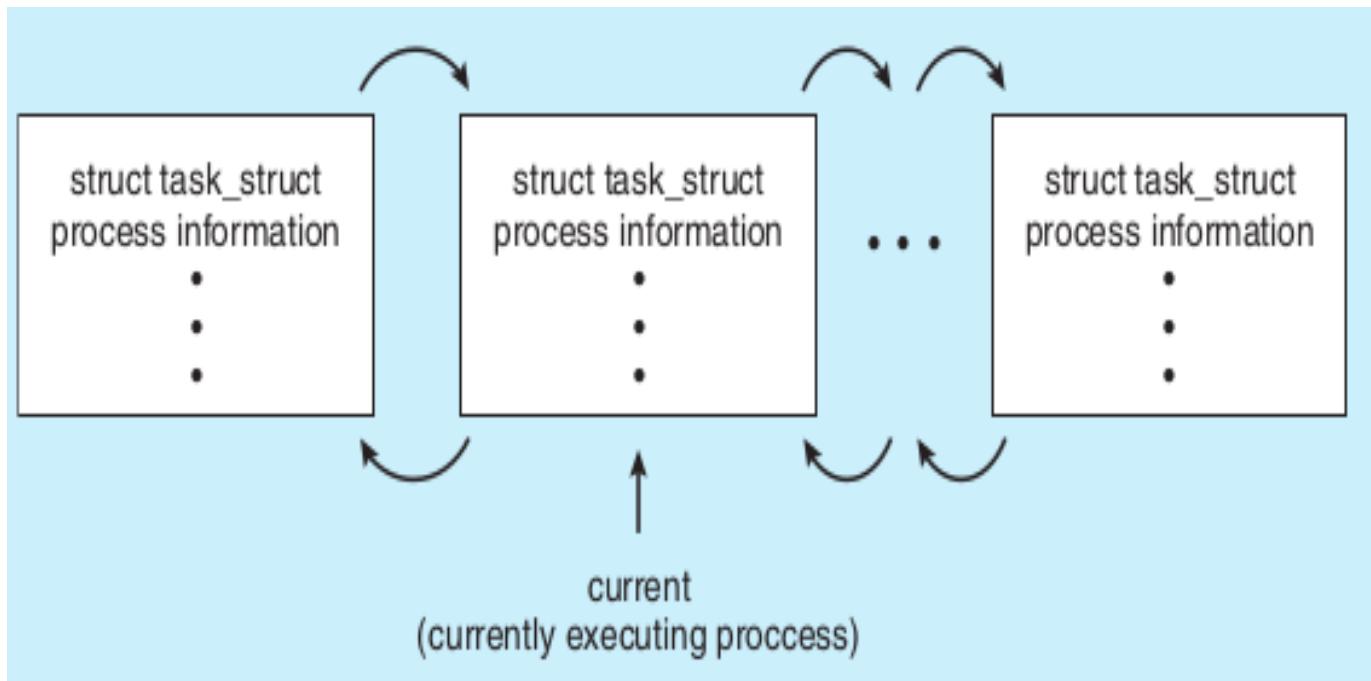
```
struct proc {
    uint sz;                      // Size of process memory (bytes)
    pde_t* pgdir;                 // Page table
    char *kstack;                 // Bottom of kernel stack for this process
    enum procstate state;        // Process state
    int pid;                      // Process 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[NFILE];   // Open files
    struct inode *cwd;            // Current directory
    char name[16];                // Process name (debugging)
};
```

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

```
struct file {
    enum { FD_NONE, FD_PIPE, FD_INODE };
    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



// Linux data structure

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

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

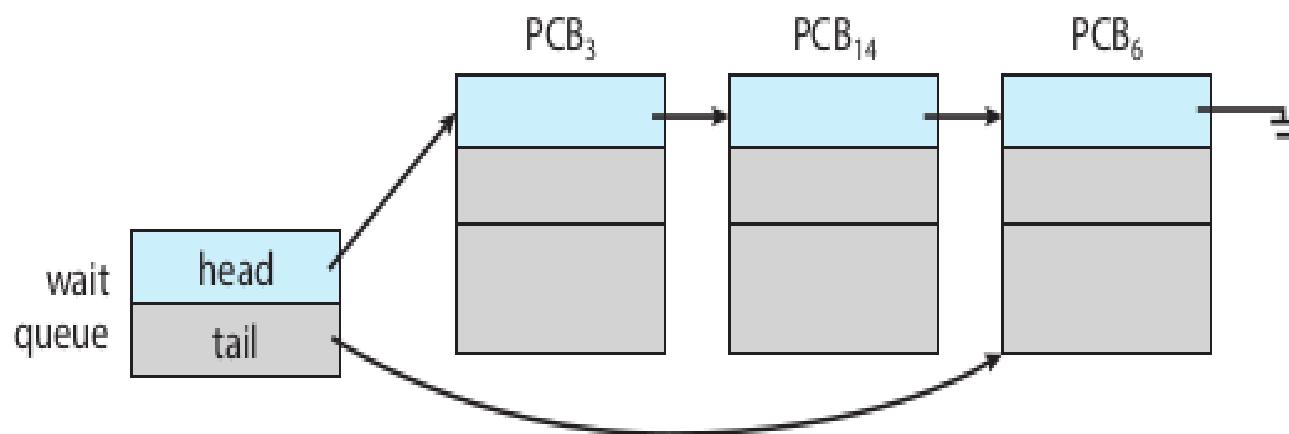
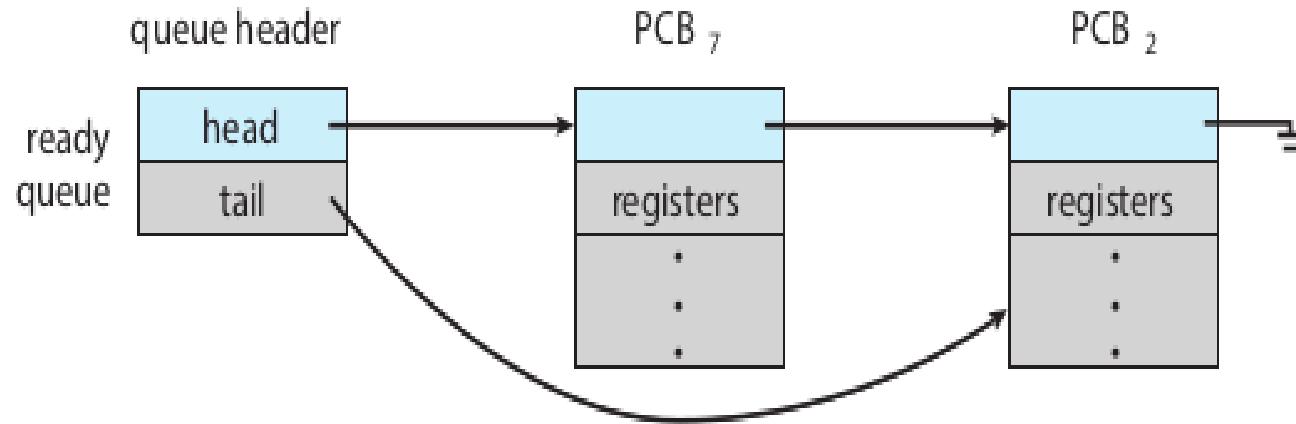


Figure 3.4 The ready queue and wait queues.

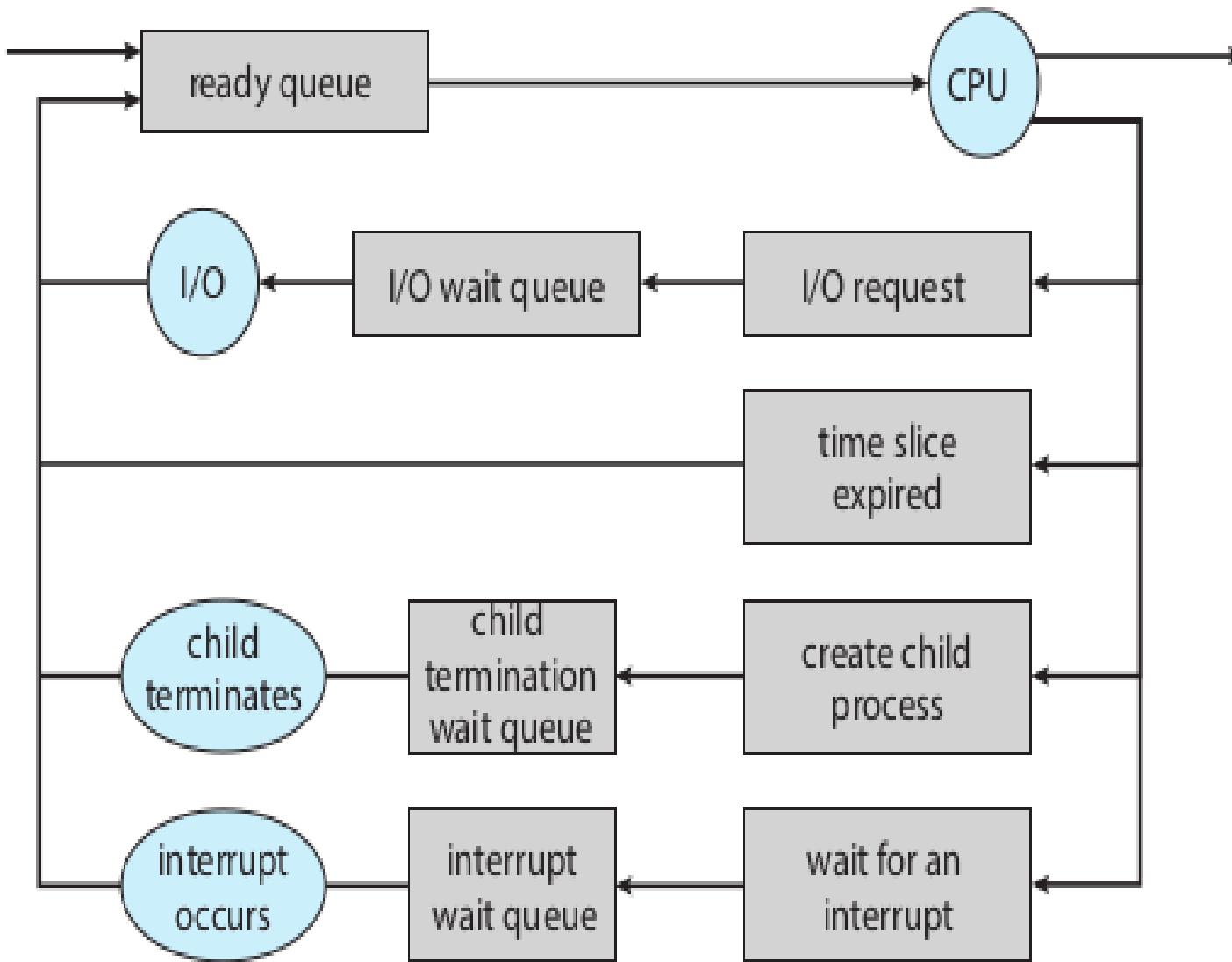


Figure 3.5 Queueing-diagram representation of process scheduling.

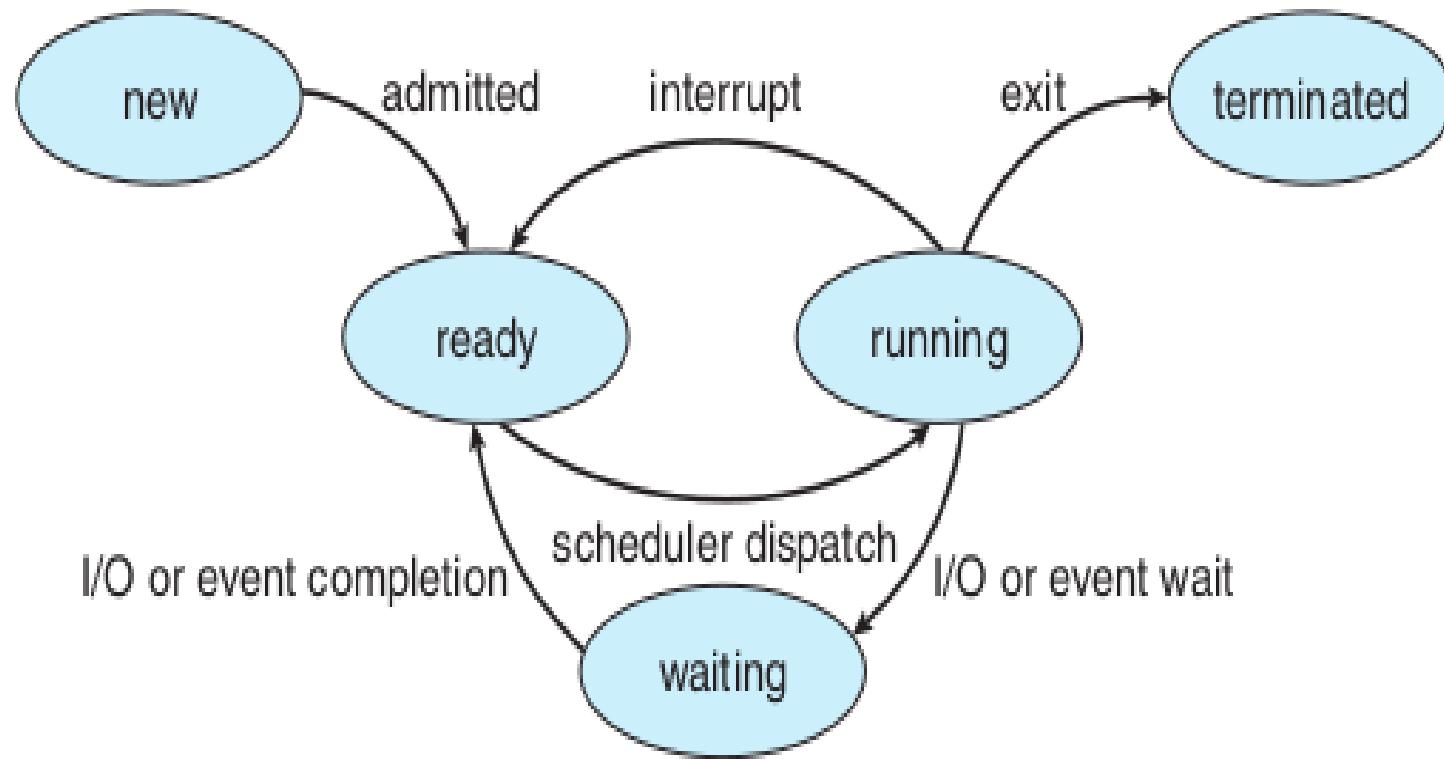


Figure 3.2 Diagram of process state.

Conceptual diagram

“Giving up” CPU by a process or blocking

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

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 process/OS to OS/another process

Context Switch

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

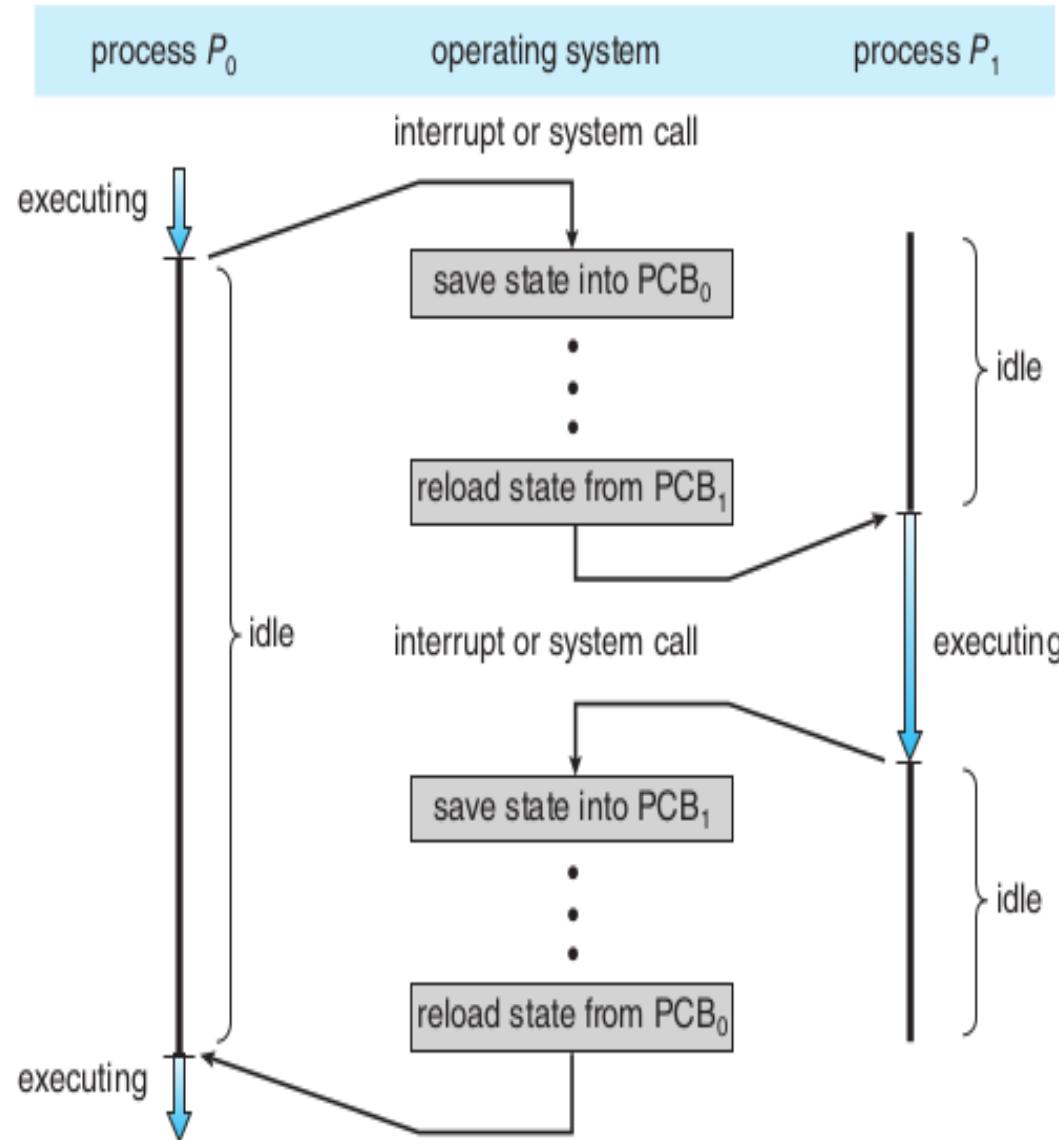


Figure 3.6 Diagram showing context switch from process to process.

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

NEXT: XV6 code overview

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

Processes in xv6 code

Process Table

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

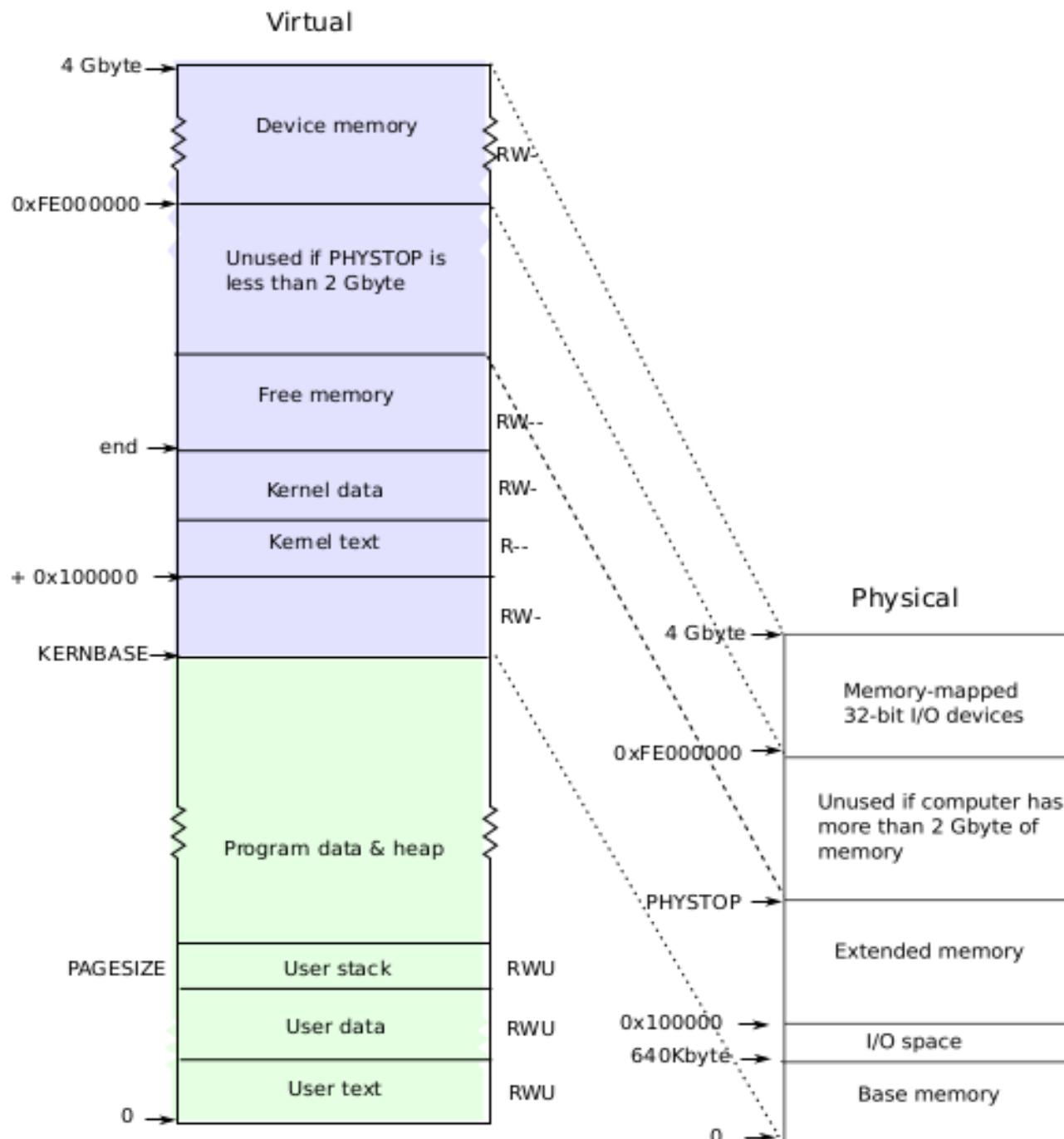
One single global array
of processes

Protected by
ptable.lock

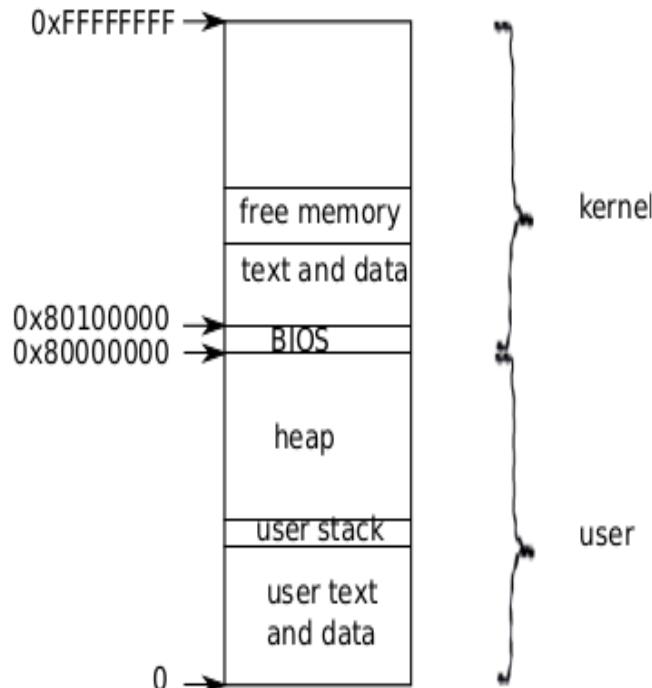
Layout of process's VA space

xv6
schema!

different
from Linux



Logical layout of memory for a process



Address 0:
code

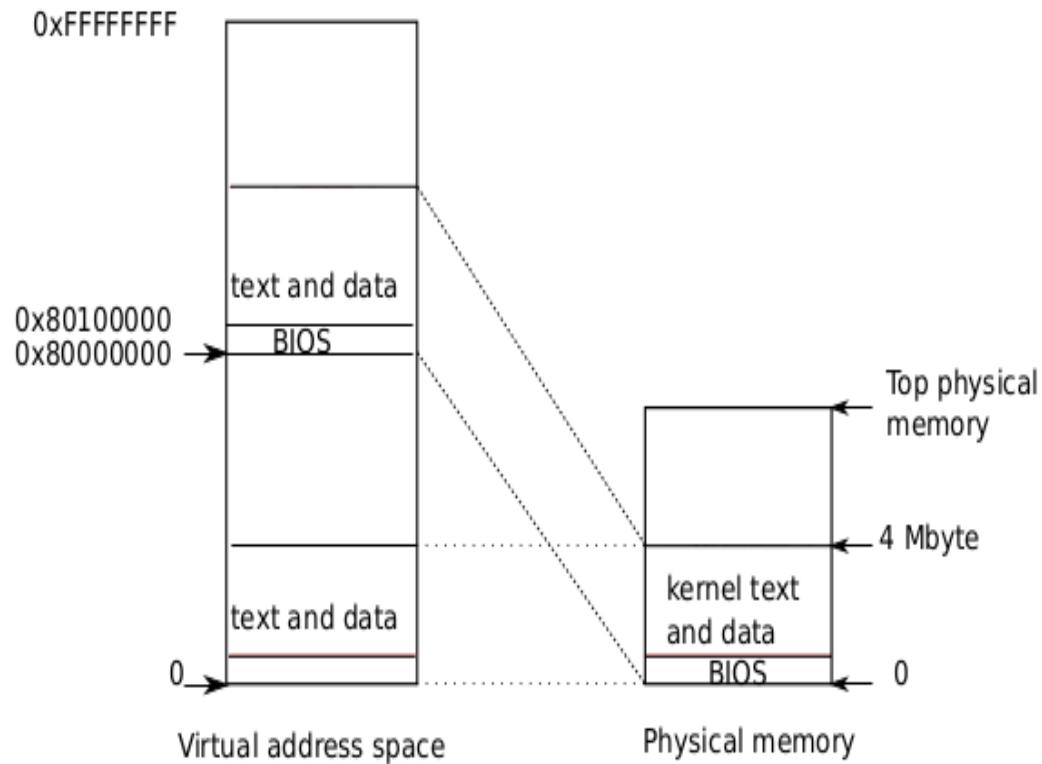
Then globals

Then stack

Then heap

Each process's
address space
maps kernel's
text, data, also

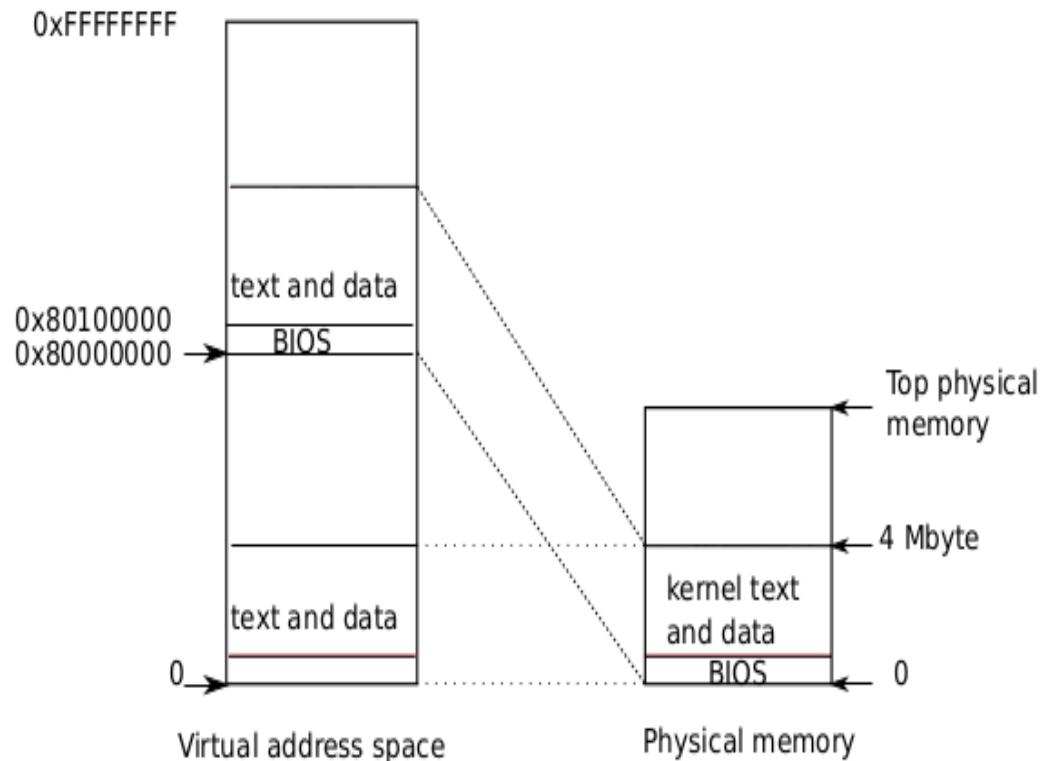
Kernel mappings in user address space actual location of kernel



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
0x800000
00
because
some
systems
may not**

Imp Concepts

A process has two stacks

user stack: used when user code is running

kernel stack: used when kernel is running on behalf of a process

Note: there is a third stack also!

The kernel stack used by the scheduler itself

Not a per process stack

Struct proc

// Per-process state

struct proc {

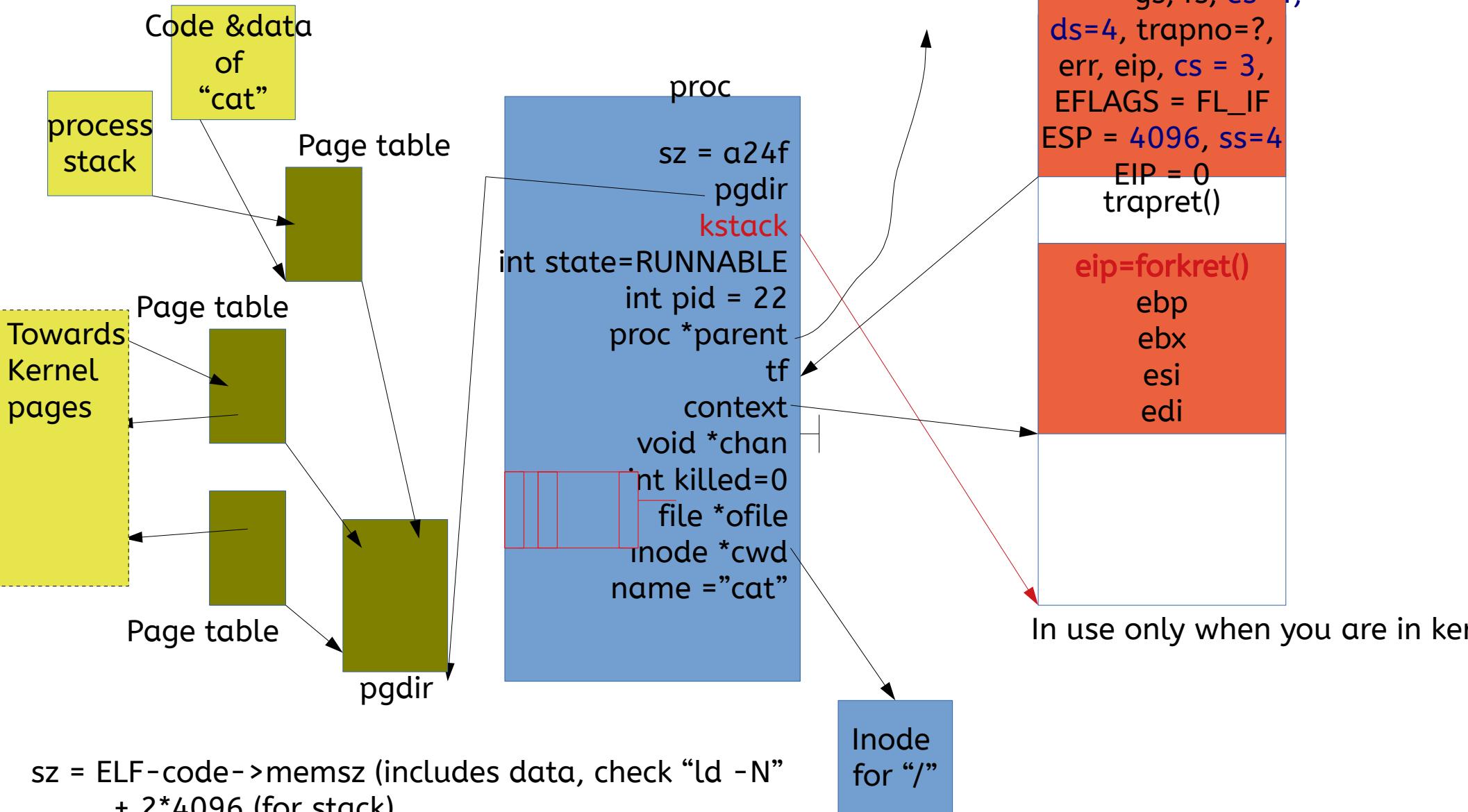
 uint sz; // Size of
process memory (bytes)

 pde_t* pgdir; // Page table

 char *kstack; // Bottom of
kernel stack for this process

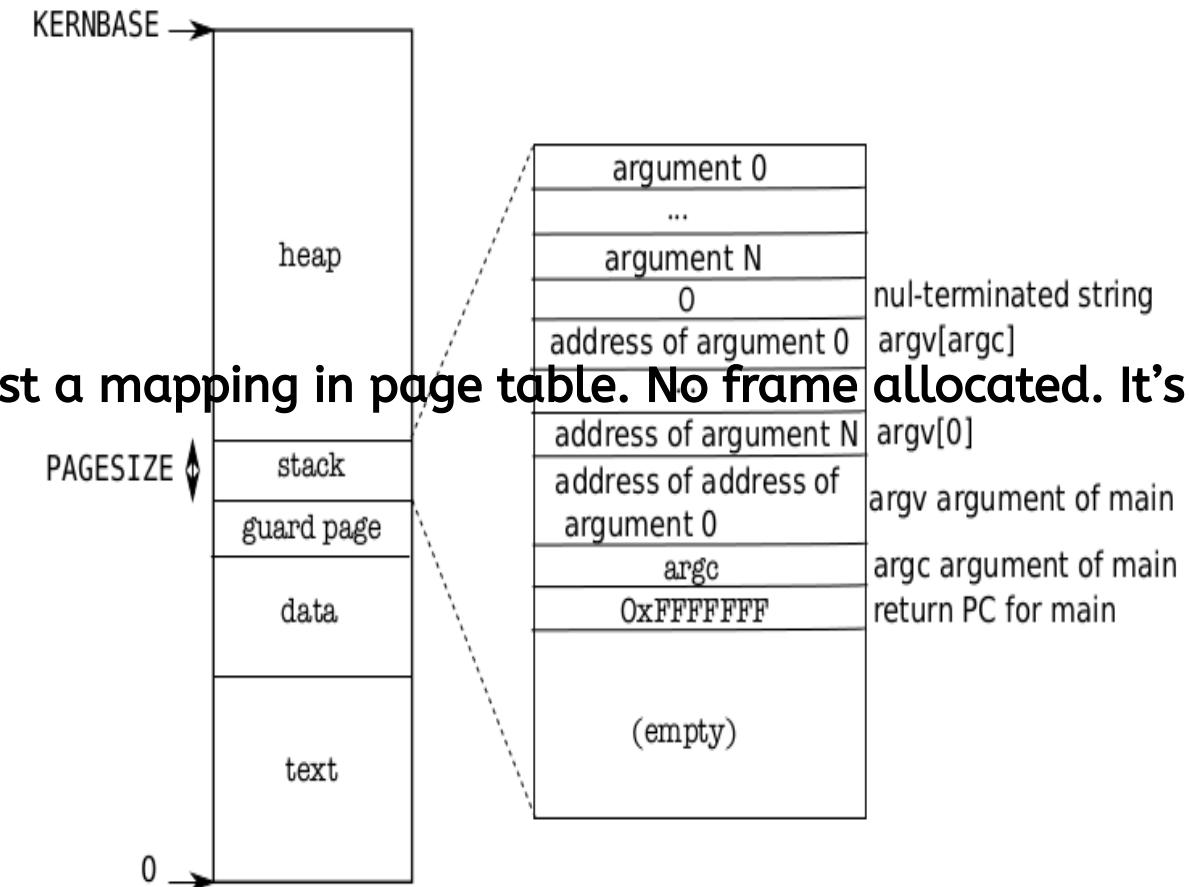
 enum procstate state; // Process
state. allocated, ready to run,
running, wait-

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

st a mapping in page table. No frame allocated. It's marked as invalid. So if stack grows (due

Handling Traps

Handling traps

Transition from user mode to kernel mode

On a system call

On a hardware interrupt

User program doing illegal work (exception)

Actions needed, particularly w.r.t. to hardware interrupts

Change to kernel mode & switch to kernel stack

Kernel to work with devices, if needed

Kernel to understand interface of device

Handling traps

Actions needed on a trap

Save the processor's registers (context) for future use

Set up the system to run kernel code (kernel context) on kernel stack

Start kernel in appropriate place (sys call, intr handler, etc)

Kernel to get all info related to event (which block I/O done?, which sys call called, which process did exception and what type, get arguments to system call, etc)

Privilege level

The x86 has 4 protection levels, numbered 0 (most privilege) to 3 (least privilege).

In practice, most operating systems use only 2 levels: 0 and 3, which are then called kernel mode and user mode, respectively.

The current privilege level with which the x86 executes instructions is stored in %cs register, in the field CPL.

Privilege level

Changes automatically on
“int” instruction
hardware interrupt
exception
Changes back on
iret

“int” 10 --> makes 10th hardware interrupt.
S/w interrupt can be used to create
hardware interrupt’

Xv6 uses “int 64” for actual system calls

Interrupt Descriptor Table (IDT)

IDT defines interrupt handlers

Has 256 entries

each giving the %cs and %eip to be used when handling the corresponding interrupt.

Interrupts 0-31 are defined for software exceptions, like divide errors or attempts to access invalid memory addresses.

Xv6 maps the 32 hardware interrupts to the range 32-63

and uses interrupt 64 as the system call interrupt

Interrupt Descriptor Table (IDT) entries

```
// Gate descriptors for interrupts and traps

struct gatedesc {

    uint off_15_0 : 16; // low 16 bits of offset in
segment

    uint cs : 16;      // code segment selector

    uint args : 5;    // # args, 0 for
interrupt/trap gates

    uint rsv1 : 3;    // reserved(should be zero I
guess)

    uint type : 4;    // type(STS_{IG32,TG32})

    uint s : 1;        // must be 0 (system)
```

Setting IDT entries

```
void  
tvinit(void)  
{  
    int i;  
    for(i = 0; i < 256; i++)  
        SETGATE(idt[i], 0, SEG_KCODE<<3, vectors[i],  
0);  
    SETGATE(idt[T_SYSCALL], 1, SEG_KCODE<<3,  
vectors[T_SYSCALL], DPL_USER);  
/* value 1 in second argument --> don't  
 disable interrupts
```

Setting IDT entries

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

Setting IDT entries

Vectors.S

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

```
# handlers
```

```
.globl alltraps
```

```
.globl vector0
```

```
vector0:
```

```
    pushl $0
```

```
    pushl $0
```

```
jmp alltraps
```

trapasm.S

```
#include "mmu.h"
```

```
# vectors.S sends all traps  
here.
```

```
.globl alltraps
```

```
alltraps:
```

```
# Build trap frame.
```

```
    pushl %ds
```

```
    pushl %es
```

```
    pushl %fs
```

```
    pushl %gs
```

How will interrupts be handled?

On intinstruction/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

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

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

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

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

trap()

Has a switch

switch(tf->trapno)

Q: who set this
trapno?

**Depending on the
type of trap**

Call interrupt
handler

Timer

wakeup(&ticks)

IDE: disk interrupt

Ideintr()

KBD

Kbdintr()

COM1

Uatrintr()

If Timer

Call yield() -- calls sched()
()

when trap() returns

#Back in alltraps

call trap

addl \$4, %esp

Return falls
through to trapret...

.globl trapret

trapret:

popal

popl %gs

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

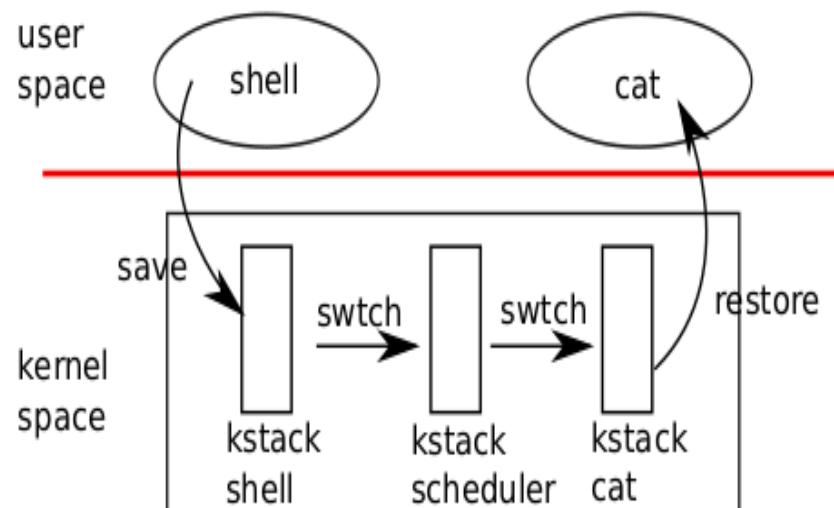
F() { i++; j++; }

P2 was doing

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

**P1 will experience a timer
interrupt after it has done**

4 stacks need to change!



User stack of
process ->
**kernel stack
of process**

Switch to
kernel stack

The normal
sequence on
any interrupt
!

scheduler()

Disable interrupts

Find a RUNNABLE process. Simple round-robin!

c->proc = p

switchuvvm(p) : Save TSS of scheduler's stack 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

pushl %edi

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 timer
calls yield()

sleep() - a process going to wait calls sleep

```
void
```

```
sched(void)
```

```
{
```

```
int intena;
```

```
struct proc *p =  
myproc();
```

```
if(!holding(&ptab  
le.lock))
```

```
panic("sched  
ptable.lock");
```

sched()

get current
process

Error checking
code (ignore as
of now)

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

call to switch

sched() and scheduler()

sched() {

...

swtch(&p->context, myscheduler, sched(), context);

***/**

/* after swtch() call in sched(), the control jumps to Y in scheduler

scheduler(void) {

...

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

Switch from process stack to

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 switchuv(p):

Introduction to Linux Desktop and Command Line

11 Jan 2022

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

Why GNU/Linux ?

Why GNU/Linux ?

1. Programmer's Paradise : most versatile, vast, all pervasive programming environment
2. Free Software (or Open Source?) : Free as in freedom. *Freely* Use, copy, modify, redistribute.
3. Highly Productive : Do more in less time
4. Better quality, more secure, very few crashes

Why Command Line ?

1. Not for everyone ! Absolutely !
2. Those who do it are way more productive than others
3. Makes you think !
4. Portable. Learn once, use everywhere on all Unixes, Linuxes, etc.

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 ifles

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 working directory

. current directory

.. parent directory

./Desktop/xyz/..../p.c

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/`

`cd`

`cd /home/student/Desktop`

notation: ~

`cd ~`

`cd ~/Desktop`

`ls ~/Desktop`

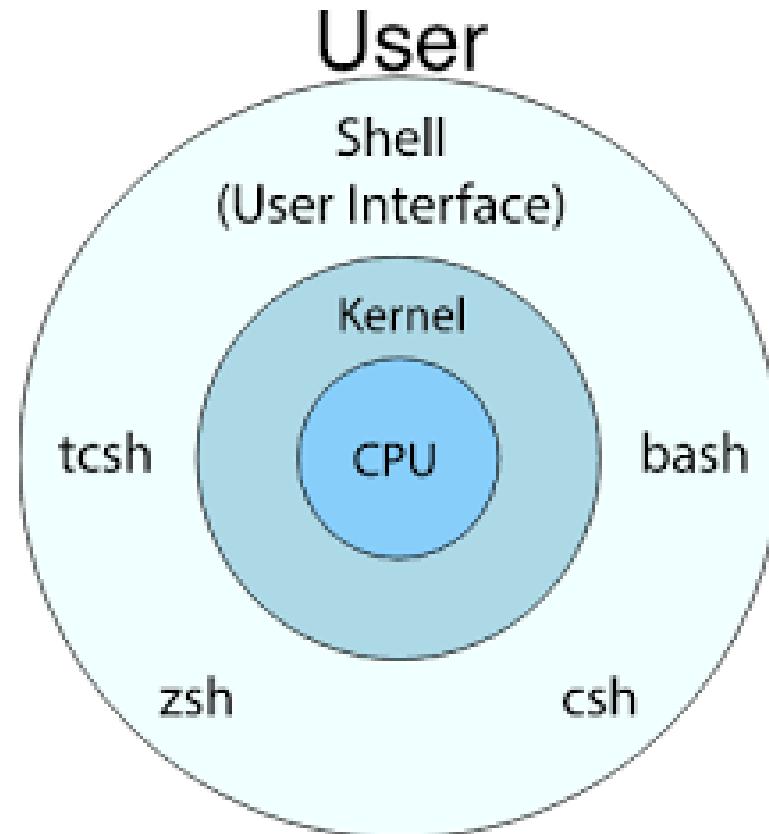
Map these
commands to
navigation using a
graphical file
browser

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);
        }
    }
}
```

File Permissions on Linux

Two types of users

root and non-root

Users can grouped into 'groups'

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 uses these notations

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

3 sets = user (owner), group, others

last-modification

3 permissions = read, write, execute
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

**x on a file: can ask the os to run the file
as an executable program**

r on a directory: can do 'ls'

**w on a directory: can add/remove files
from that directory (even without 'r'!)**

**x on a directory: can 'cd' to that
directory**

Access rights examples

-rw-r--r--

Readable and writable for file 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...

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

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	
/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
recover

/media
removable media:

/media/cdrom

/mnt/
mounted filesystems
/opt/
sysadmin

used instead

/proc/

/proc/version ...

/root/
/sbin/
/sys/

Corrupt files the system tried to

Mount points for

/media/usbdisk,

Mount points for temporarily

Specific tools installed by the

/usr/local/ often

Access to system information

/proc/cpuinfo,

root user home directory

Administrator-only commands

System and device controls

(cpu frequency,

GNU / Linux filesystem structure

/tmp/
/usr/
to the system)

/usr/sbin...
/usr/local/
sysadmin

/opt/)
/var/
system servers

/var/spool/mail (**incoming
mail**), /var/spool/lpd (**print jobs**)...

Temporary files
Regular user tools (not essential

/usr/bin/, /usr/lib/,

Specific software installed by the

(often preferred to

Data used by the system or

/var/log/,

Files: cut, copy, paste, remove,

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

cp -r ./folder1 /tmp/

cp -r ./folder1

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

rmdir /tmp/a /tmp/b

Useful Commands

echo

echo hi

echo hi there

echo "hi there"

j=5; echo \$j

sort

sort

sort < /etc/passwd

firefox

libreoffice

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

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

`df`

`df -h`

`du`

`du -hs .`

`bc`

`time`

`date`

`diff`

`wc`

Network Related Commands

ifconfig

ping

ssh

w

scp

last

telnet

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

Shell Wildcards

? (question mark)

Any one character

ls a?c

ls ??c

*

any number of
characters

ls *

ls d*

echo *

[]

Matches a range

ls a[1-3].c

{}

ls pic[1-3].{txt,jpg}

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

~/.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`

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.

Special devices (3)

`/dev/full`

Mimics a full device.

Useful to check that your application properly handles
this kind of situation.

See man `full` for details.

Files names and inodes

Hard Links Vs Soft Links

**EMBEDDED OLE
OBJECT**

Shell Programming

Shell Programming

is “programming”

**Any programming: Use existing tool to
create new utilities**

**Shell Programming: Use shell facilities
to create better utilities**

Know “commands” --> More tools

Know how to combine them

Shell Variables

Shell supports variables

Try:

```
j=5; echo $j
```

No space in j=5

Try

```
set
```

Shows all set variables

Try

```
a=10; b=20; echo $a$b
```

What did you learn?

Shell's predefined variables

USER

Name of current user

HOME

Home directory of
current \$USER

PS1

The prompt

LINES

No. of lines of the
screen

HOSTNAME

Name of the
computer

OLDPWD

Previous working
directory

PATH

List of locations to
search for a
command's
executable file

\$?

Redirection

cmd > filename

Redirects the output to a file

Try:

```
ls > /tmp/xyz
```

```
cat /tmp/xyz
```

```
echo hi > /tmp/abc
```

```
cat /tmp/abc
```

cmd < filename

Reads the input from a file instead of keyboard

Think of a command now!

Pipes

Try

```
last | less
```

```
grep bash /etc/passwd | head -1
```

```
grep bash /etc/passwd | head -2 | tail -1
```

Connect the output of LHS command as input of RHS command

Concept of *filters* – programs which read input only from *stdin* (keyboard, e.g. *scanf*, *getchar*), and write output to *stdout* (e.g. *printf*, *putchar*)

Programs can be connected using pipes if they are filters

The *test* command

test

test 10 -eq 10

test "10" == "10"

test 10 -eq 9

test 10 -gt 9

test "10" >= "9"

test -f /etc/passwd

test -d ~/desktop

...

**Shortcut notation
for calling test**

[]

[10 -eq 10]

**Note the space
after '[' and before**

The *expr* command and backticks

expr

```
expr 1 + 2
```

```
a=2; expr $a + 2
```

```
a=2; b=3; expr $a + $b
```

```
a=2;b=3; expr $a \* $b
```

```
a=2;b=3; expr $a | $b
```

backticks ``

```
j=`ls`; echo $j
```

```
j=`expr 1 + 2`; echo $j
```

Used for mathematical calculations

if then else

```
if [ $a -lt $b ]  
then  
echo $a  
else  
echo $b  
fi
```

```
if [ $a -lt $b ];then  
echo $a; else echo  
$b; fi
```

0 TRUE

Nonzero FALSE

while do done

```
while [ $a -lt $b ]
```

```
do
```

```
echo $a
```

```
a=`expr $a + 1`
```

```
done
```

```
while [ $a -lt $b ]
```

```
do
```

```
echo $a
```

```
a=$((a + 1))
```

```
done
```

```
while [ $a -lt $b ];
```

```
do echo $a;
```

```
a=`expr $a + 1`;
```

```
done
```

for x in ... do done

```
for i in {1..10}  
do  
echo $i  
done
```

```
for i in *; do echo  
$i;  
done
```

```
for i in *  
do  
echo $i  
done
```

read space

case \$space in

[1-6]*)

Message="one to 6"

;;

[7-8]*)

Message="7 or 8"

;;

9[1-8])

**Message="9 with a
number"**

;;

***)**

case ... esac

Syntax

Try these things

Print 3rd line from /etc/passwd, which contains the word bash

Print numbers from 1 to 1000

Create files named like this: file1, file2, file3, ... filen where n is read from user

Read i%5th file from /etc/passwd and store it in file/
file/i

Find all files ending in .c or .h and create a .tar.gz file of these files

The Golden Mantra

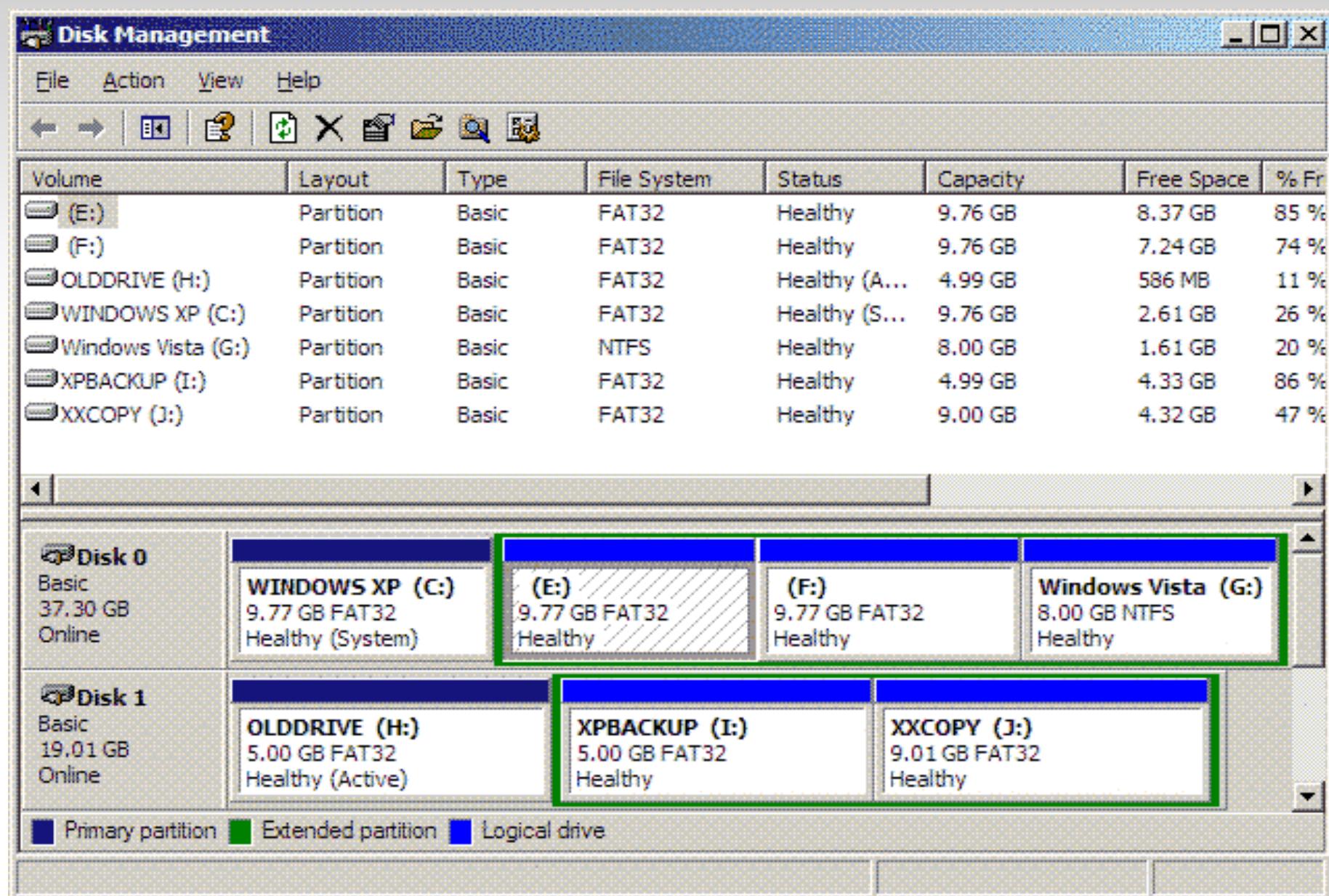
Everything can be done from command line !

Command line is far more powerful than
graphical interface

Command line makes you a better
programmer

Mounting

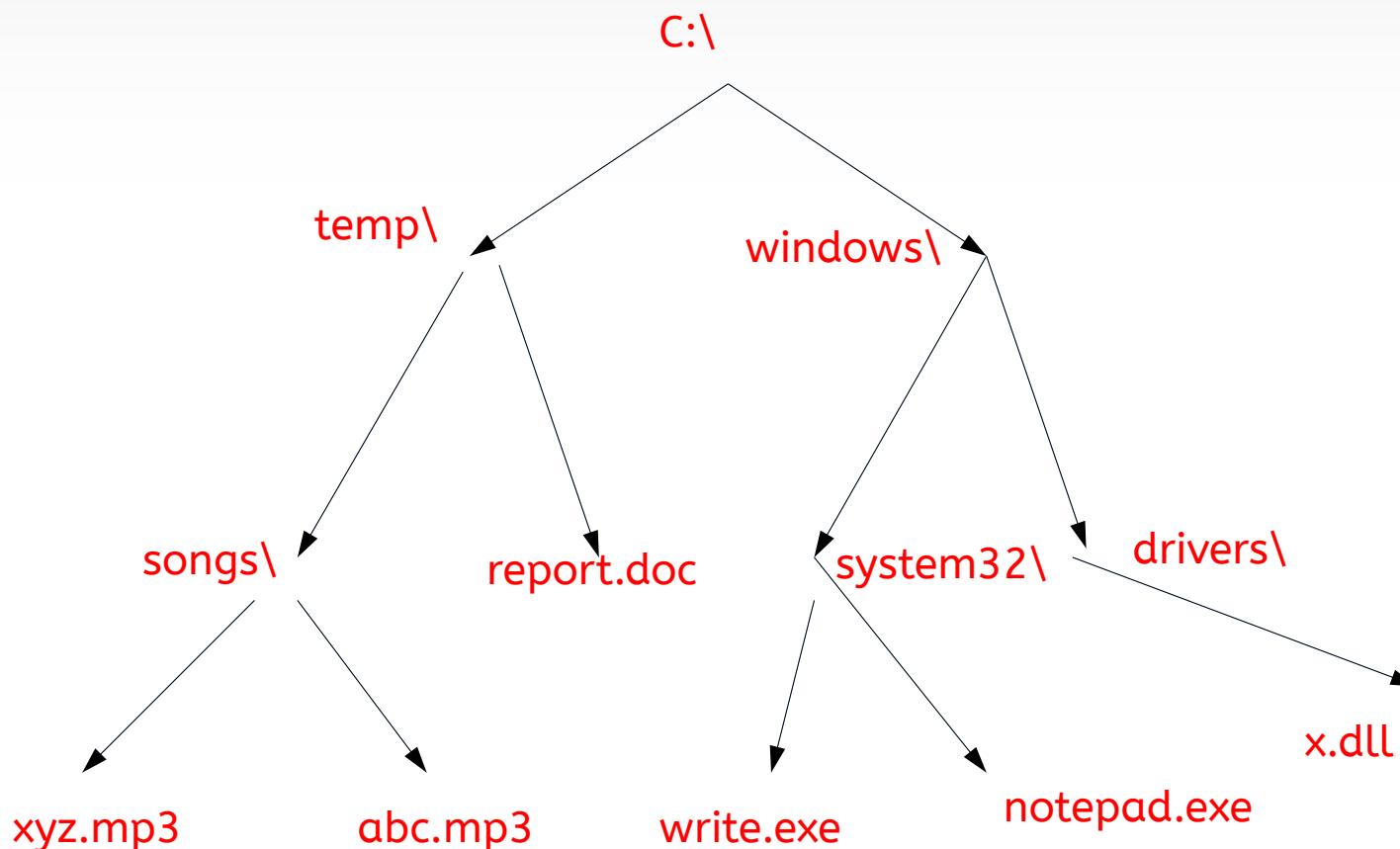
Partitions



Windows Namespace

c:\temp\songs\xyz.mp3

Root is C:\ or D:\ etc
Separator is also “\”



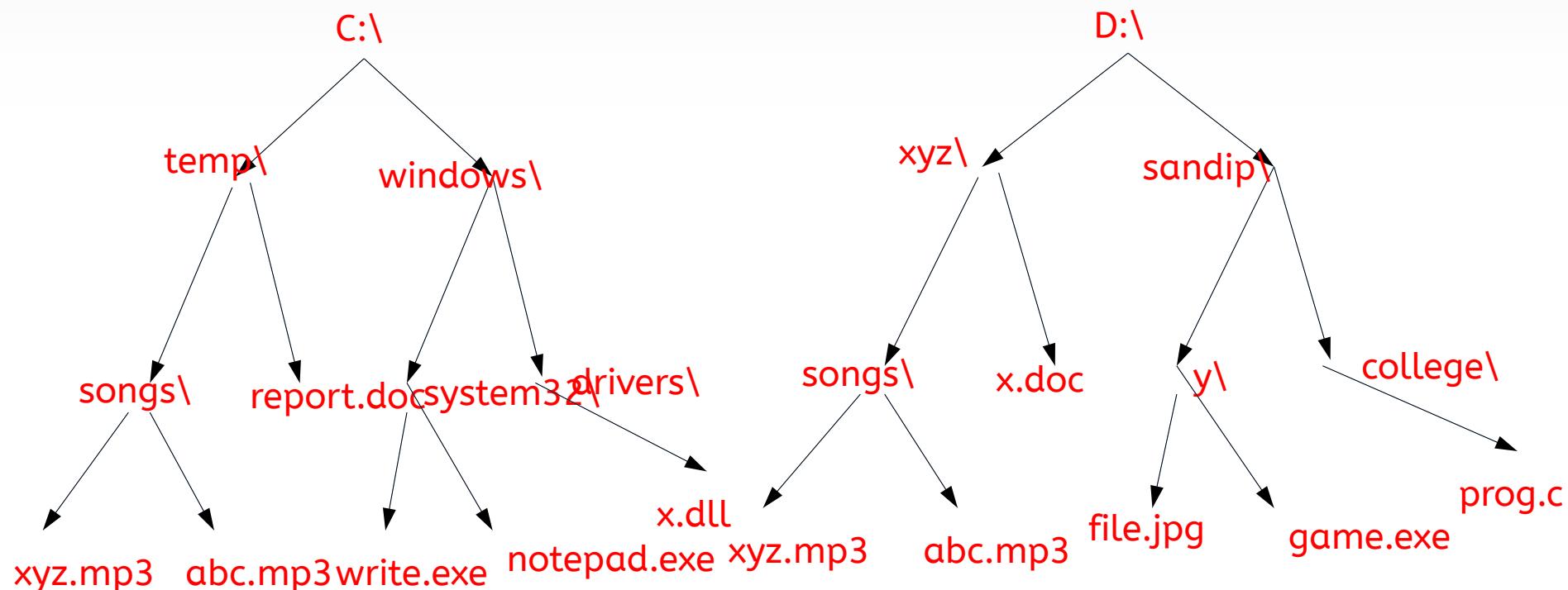
Windows Namespace

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”



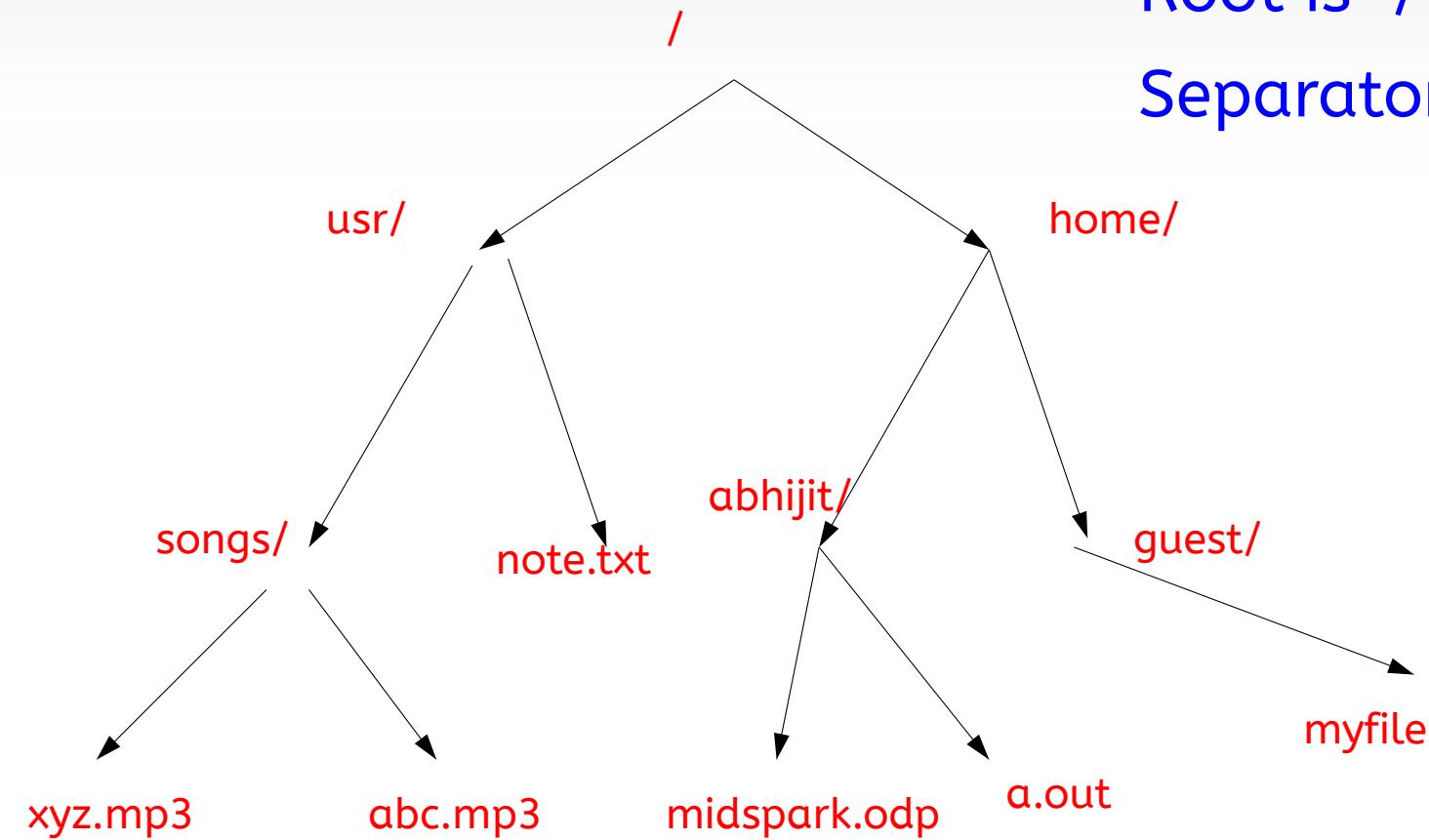
Linux Namespace: On a partition

/usr/songs/xyz.mp3

On every partition:

Root is “/”

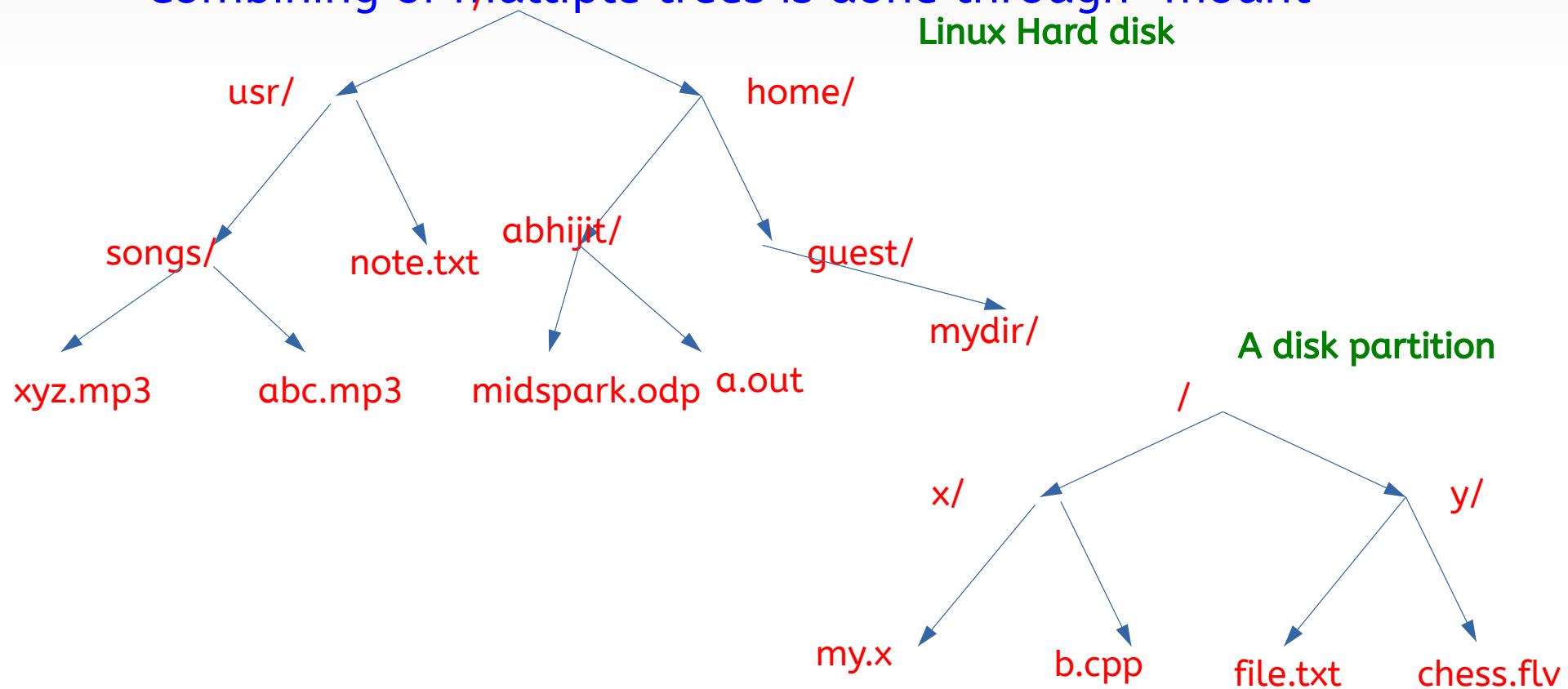
Separator is also “/”



Linux namespace: Mount

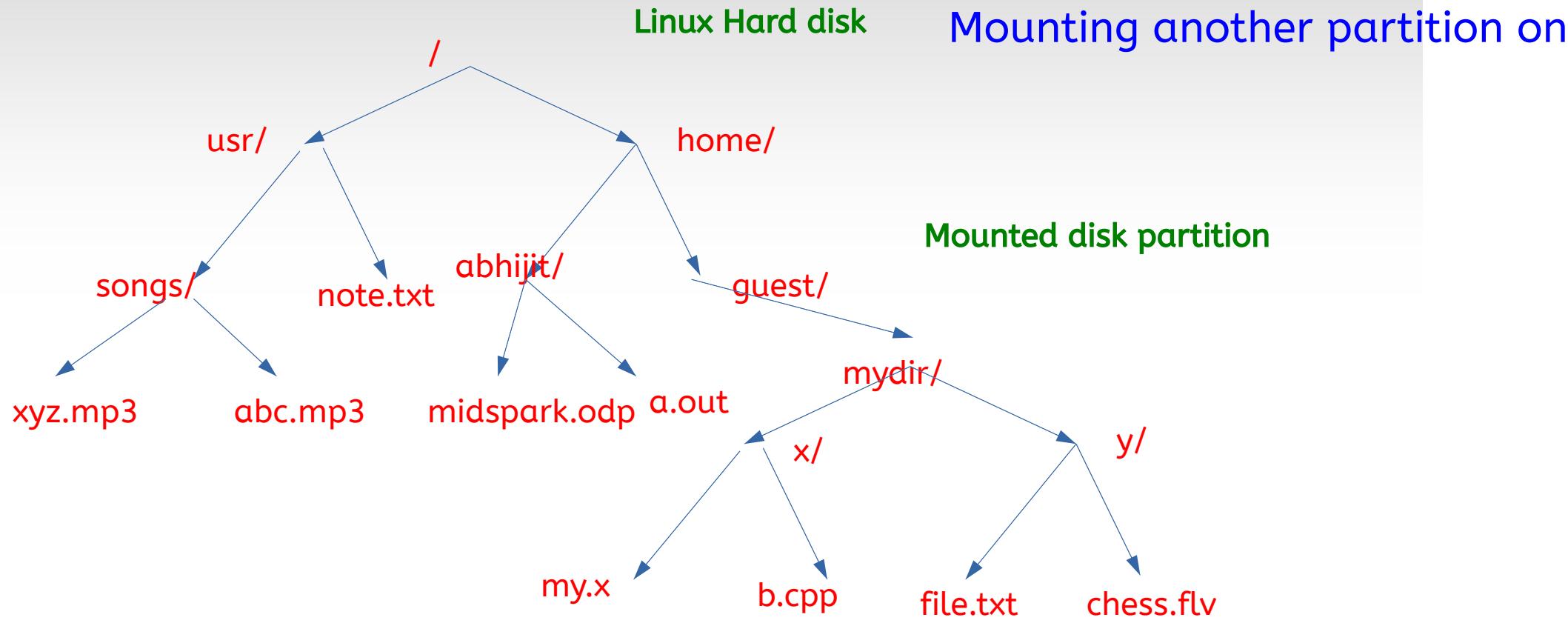
Linux namespace is a single “tree” and not a “forest” like Windows

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

Let's go for a live installation Demo !

Some Shell Gimmicks

Terminal Tricks

Ctrl + n : same as Down arrow.

Ctrl + p : same as Up arrow.

Ctrl + r : begins a backward search through command history.(keep pressing Ctrl + r to move backward)

Ctrl + s : to stop output to terminal.

Ctrl + q : to resume output to terminal after Ctrl + s.

Terminal Tricks

Ctrl + a : move to the beginning of line.

Ctrl + e : move to the end of line.

Ctrl + d : if you've type something, Ctrl + d deletes the character under the cursor, else, it escapes the current shell.

Ctrl + k : delete all text from the cursor to the end of line.

Ctrl + t : transpose the character before the cursor with the one under the cursor

Terminal Tricks

**Ctrl + w : cut the word before the cursor; then Ctrl + y
paste it**

**Ctrl + u : cut the line before the cursor; then Ctrl + y
paste it**

Ctrl + _ : undo typing.

Ctrl + l : equivalent to clear.

**Ctrl + x + Ctrl + e : launch editor defined by \$EDITOR
to input your command.**

Run from history

First: What's history?

Ans: Run 'history

\$ history

\$!53

\$!!

\$!cat

\$!c

Math

```
$ echo $(( 10 + 5 )) #15
```

```
$ x=1
```

```
$ echo $(( x++ )) #1 , notice that it is still 1, since it's  
post-incremen
```

```
$ echo $(( x++ )) #2
```

More Math

```
$ seq 10|paste -sd+|bc
```

```
# How does that work ?
```

Using expr

```
$ expr 10+20 #30
```

```
$ expr 10\*20 #600
```

```
$ expr 30 \> 20 #1
```

More Math

Using bc

\$ bc

obase=16

ibase=16

AA+1

AB

More Math

Using bc

\$ bc

ibase=16

obase=16

AA+1

07 17

what went wrong?

Fun with grep

```
$ grep -Eo '[0-9]{1,3}\.[0-9]{1,3}\.[0-9]{1,3}\.[0-9]{1,3}'
```

above will only search for IP addresses!

```
$ grep -v bbo filename
```

```
$ grep -w 'abhijit' /etc/passwd
```

```
$ grep -v '^#' file.txt
```

```
$ grep -v ^$ file.txt
```

xargs: convert stdin to args

```
$ find . | grep something | xargs rm
```

xargs is highly powerful

rsync

The magic tool to sync your folders!

```
$ rsync -rvupt ~/myfiles /media/abhijit/PD
```

```
$ rsync -rvupt --delete ~/myfiles /media/abhijit/PD
```

find

```
$ find .
```

```
$ find . -type f
```

```
$ find . -type d
```

```
$ find . -name '*.php'
```

```
$ find / -type f -size +4G
```

```
$ find . -type f -empty -delete
```

```
$ find . -type f | wc -l
```

Download : wget, curl

```
$ wget foss.coep.org.in
```

```
$ wget -r foss.coep.org.in
```

```
$ wget -r -convert-links foss.coep.org.in
```

```
$ wget -r -convert-links --no-parent  
foss.coep.org.in/fossmeeet/
```

```
$ curl  
https://raw.githubusercontent.com/onceupon/Bash-Oneliner/master/README.md | pandoc -f  
markdown -t man | man -l -
```

```
# curl is more powerful than wget. Curl can upload.  
Curl supports many protocols, wget only HTTP/FTP.
```

Random data

shuffle numbers from 0-100, then pick 15 of them randomly

\$ shuf -i 0-100 -n 15

Random pick 100 lines from a file

\$ shuf -n 100 filename

#generate 5 password each of length 13

\$ pwgen 13 5

Run commands remotely

\$ ssh administrator@foss.coep.org.in

\$ ssh -X administrator@foss.coep.org.in

\$ ssh -X administrator@foss.coep.org.in firefox

System Information

Show memory usage,. # print 10 times, at 1 second interval

\$ **free -c 10 -mhs 1**

Display CPU and IO statistics for devices and partitions. # refresh every second

\$ **iostat -x -t 1**

Display bandwidth usage on an network interface (e.g. enp175s0f0)

\$ **sudo iftop -i wlo1**

Tell how long the system has been running and

Surf the web

\$ w3m

\$ links

Add a user without commands

Know how to edit the */etc/passwd* and */etc/shadow files*

More tricks

Show 10 Largest Open Files

```
$ lsof / | awk '{ if($7 > 1048576) print $7/1048576 "MB"  
" " $9 " " $1 }' | sort -n -u | tail
```

Generate a sequence of numbers

```
$ echo {01..10}
```

More tricks

```
# Rename all items in a directory to lower case
```

```
$ for i in *; do mv "$i" "${i,,}"; done
```

```
# List IP addresses connected to your server on port  
80
```

```
$ netstat -tn 2>/dev/null | grep :80 | awk '{print $5}' |  
cut -d: -f1 | sort | uniq -c | sort -nr | head
```

Credits: <https://onceupon.github.io/Bash-Oneliner/>
<http://www.bashoneliners.com/>

User Administration

Users and Groups

There is a privileged user called “root”

Can do anything, like “administrator” on Windows

Can't login in graphical mode !

Other users are normal users

**Some users are given “sudo” privileges:
called *sudoers***

Sudo means “do as a **superuser**”

Password is asked, when the otherwise normal user tries to do administrative task

The first user account created on Ubuntu, is by default with sudo privileges

Adding/Deleting/Changing users

System → Administration → Users and Groups

Click on "Add" to add a user

Asks for password !

Provide the details asked for

Verify the user was created, by doing 'switch user'

Try 'deleting' the user created

Groups: Various groups of users meant for different purposes

Every user by default belongs to her own group

Add the user explicitly to other groups

Software installation

Some terms

.deb

The “setup” file. The installer package. Similar to Setup.exe on windows.

Contains all *binary* files and some *shell scripts*

repository

A collection of .deb files, categorized according to type (security, main, etc.)

Software source/ ubuntu mirror

A computer on internet having all .deb files for ubuntu

Software Installation Concept

Online installation

Use the “Ubuntu Software Center” to select the software, click and install !

Software is fetched automatically and installed !

Much easier than Windows !

Offline installation

Collect ALL .deb files for your application

Select all, and install using package manager

Software Installation

When we install using Software Center

.deb files are stored in /var/cache/apt/archives
folder

**One needs to be a *sudoer* to install
software**

**Try installing some software on your
own and try them out !**

Network configuration

Setting up network for a desktop

DHCP

Nothing needs to be done !

Default during installing Linux

Static I/P

System → Preferences → Network connections

System → Administration → Network

System → Administration → Network tools

Network setup for wireless

Just plug and Play !

Network icon shows available wireless network, just click and connect.

Reliance/Tata/Idea USB devices

Each has a different procedure

One needs to search the web for setting it up

Most of the devices work plug and play on Ubuntu 12.04

Some do not work, as fault of the providers, they have not given an installer CD for Linux!

Still Linux community have found ways to work around it !

My Reliance netconnect worked faster and more steadily on Linux than Windows !

Disk Management

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

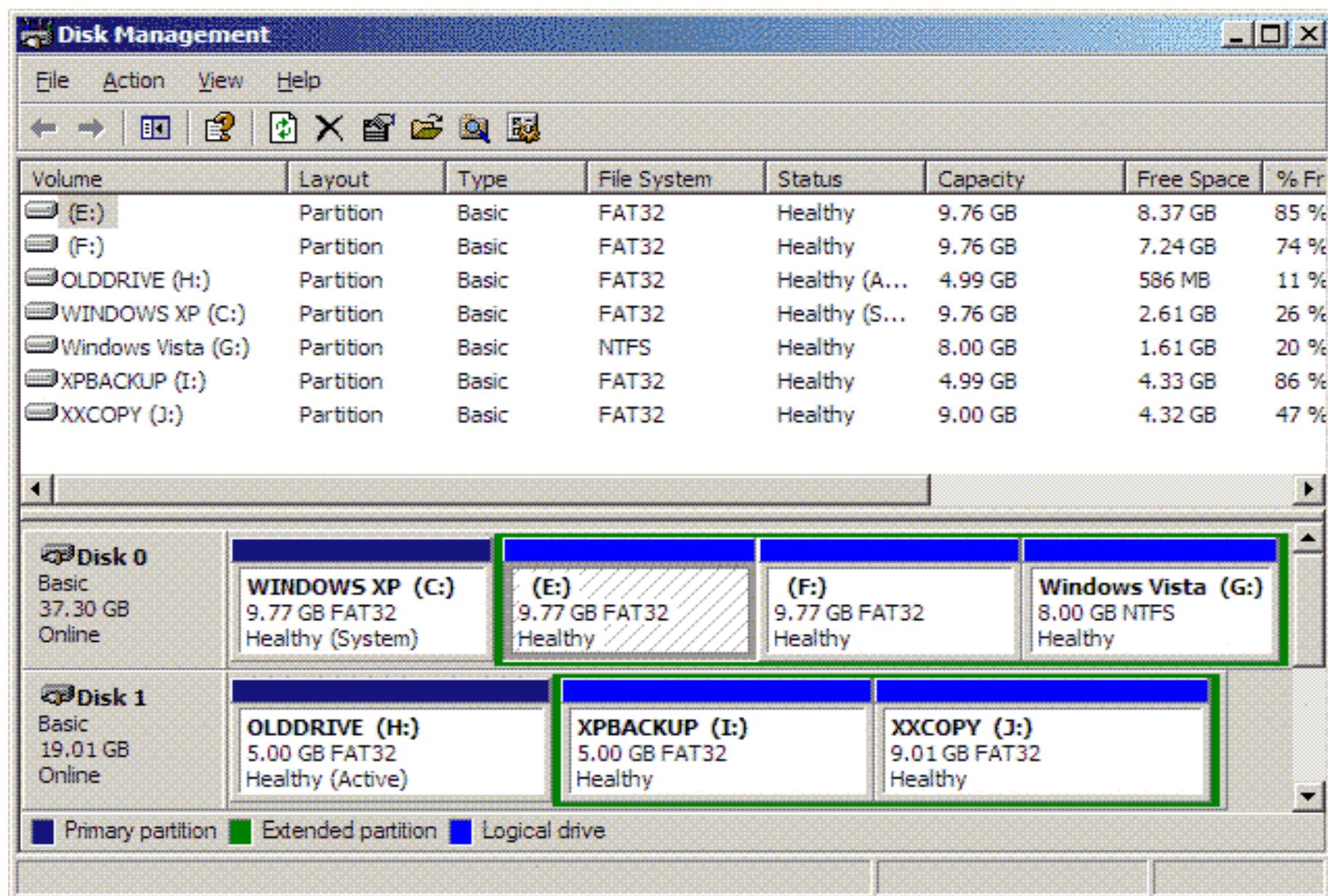
Partition is a “logical division” of the storage

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

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.

Pen drives can also be managed from here

Formatting can also be done from here