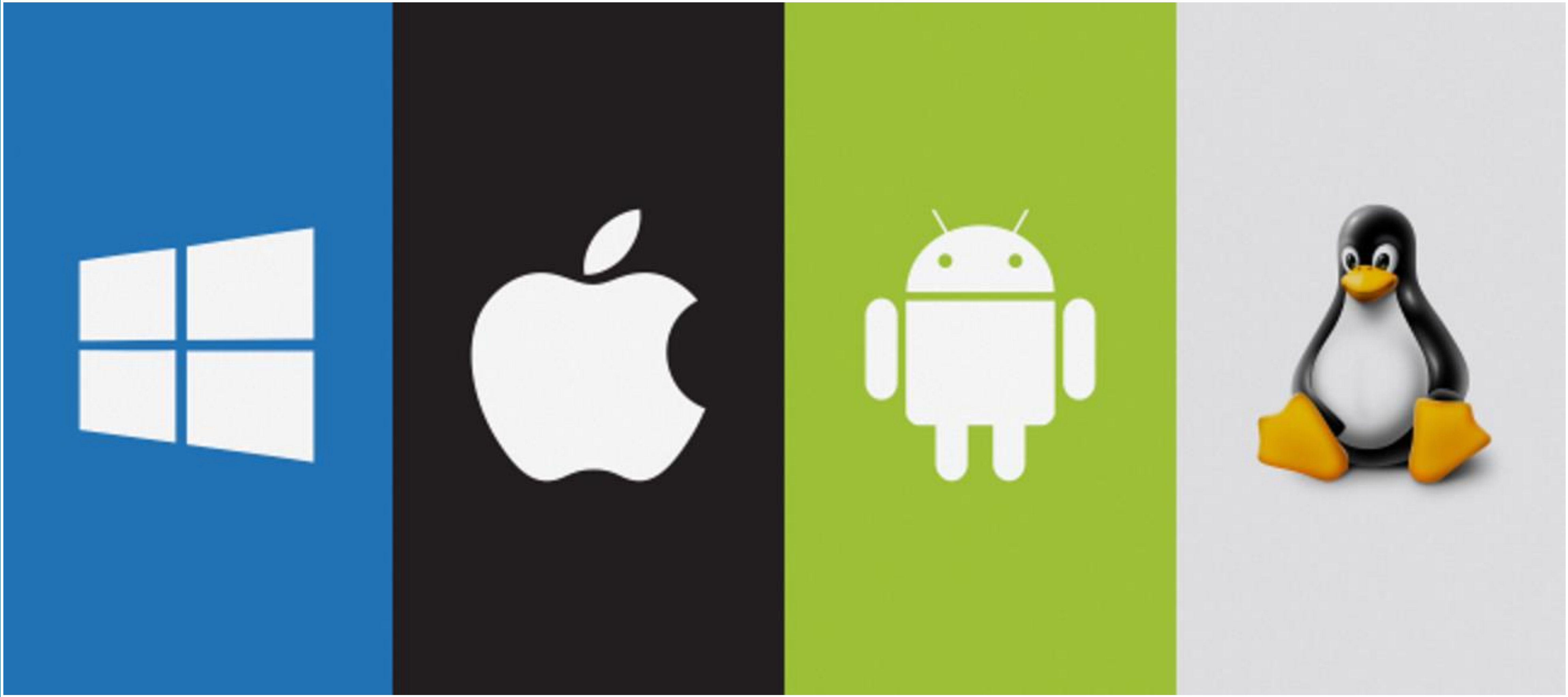
KU LEUVEN

Operating systems

Chapter 1 – Introduction

Bert Lagaisse



What is an operating system ?
What do operating systems do ?

Operating systems

!=

Rocket Science

Well, it actually is ;-)

Operating system in C

A 64-bit floating point number

- the horizontal velocity of the rocket with respect to the platform

converted to a 16 bit signed integer.

- larger than 32,767, the largest integer storable in a 16 bit signed integer,
- Hence the conversion failed.

Triggered system diagnostics code that dumped data into memory of control software of rocket’s motors

boom.

Course details

Course structure
“Operating system concepts”

Lectures

- **Slides** summarize the lectures
- **Book** “Operating system concepts”
- **Explains concepts**
  - Processes
  - Scheduling of processes
  - Main memory management
  - …
- **Example code in C and Java**
  - Explains API
  - C API in both Win32 and POSIX (Linux)
  - Small programming excercises
- Not a **C** course ! Not a **Linux** course !

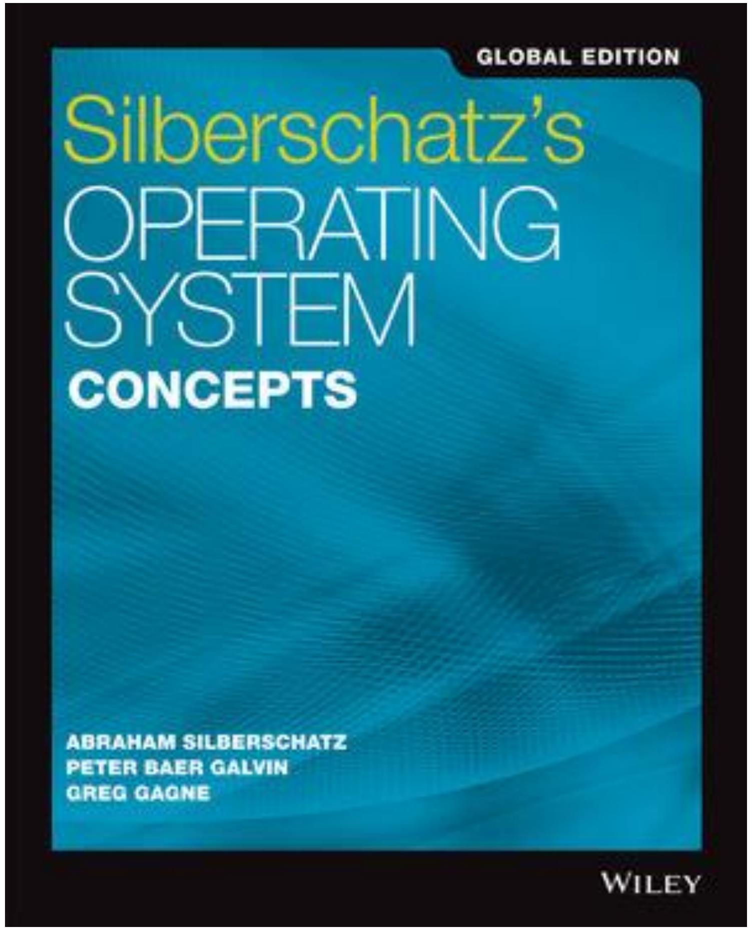
Practical sessions

- **Excercises** with OS concepts
  - Using C, on Linux
  - Apply studied concepts
- Weekly exercise sessions of 2h.
  - **Intro and demo during lectures**
  - <u>Attend to be well prepared !</u>
  - 10 lab sessions to support you!
- **Integrated and incremental** approach
  - Step-wise build-up of project
  - <u>Follow along to pass the course</u>

Book by Silberschatz et al.
"Operating system concepts, 10th edition"

Hard copy (global edition)

- Many practical excercises !
- Can be ordered via ACCO / Industria



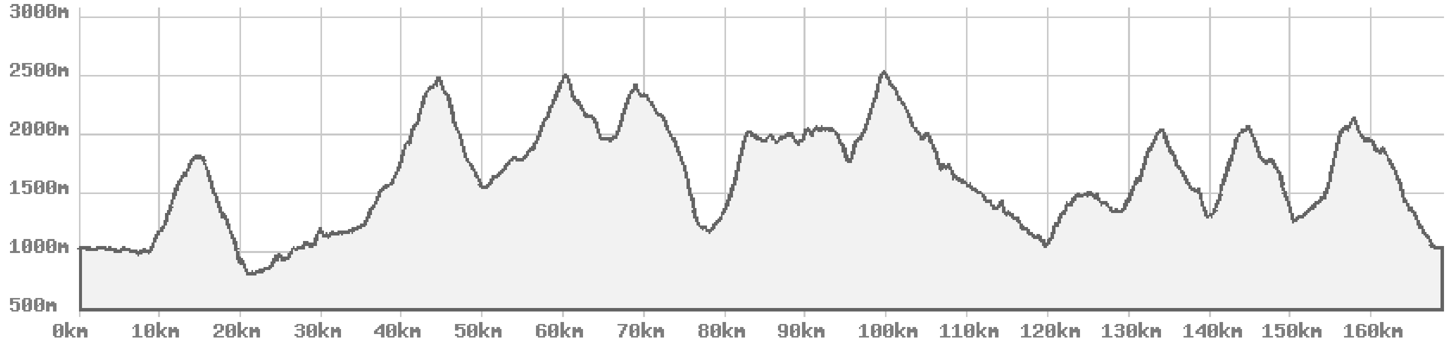
Weekly lab sessions:

Basic labs (first 6 weeks)

- **2 home works** (2 weeks)
  - Setting up a linux environment
  - Setting up your dev tools for C
- **3 basic C labs** (4 weeks)
  - Hand in milestone 1 at end

Project labs (last 7 weeks)

- **4 learning labs (4 weeks)**
  - 4 key OS concepts needed for project
    - File I/O, Inter-process communication
    - Threads, Thread synchronization
  - 4 subsystems of the project
    - = working on your exam lab
    - 2 milestones to hand in
- **Final project (3 weeks)**
    - Composition / integration of above
    - Final submission as exam
    - Final, straight flat to the finish.



KU LEUVEN

Exam ?

Written exam

- Closed book
- Multiple choice
- Written exercises
- C programs: What is the output ?
- Open Questions: Explain concepts

Integrated project

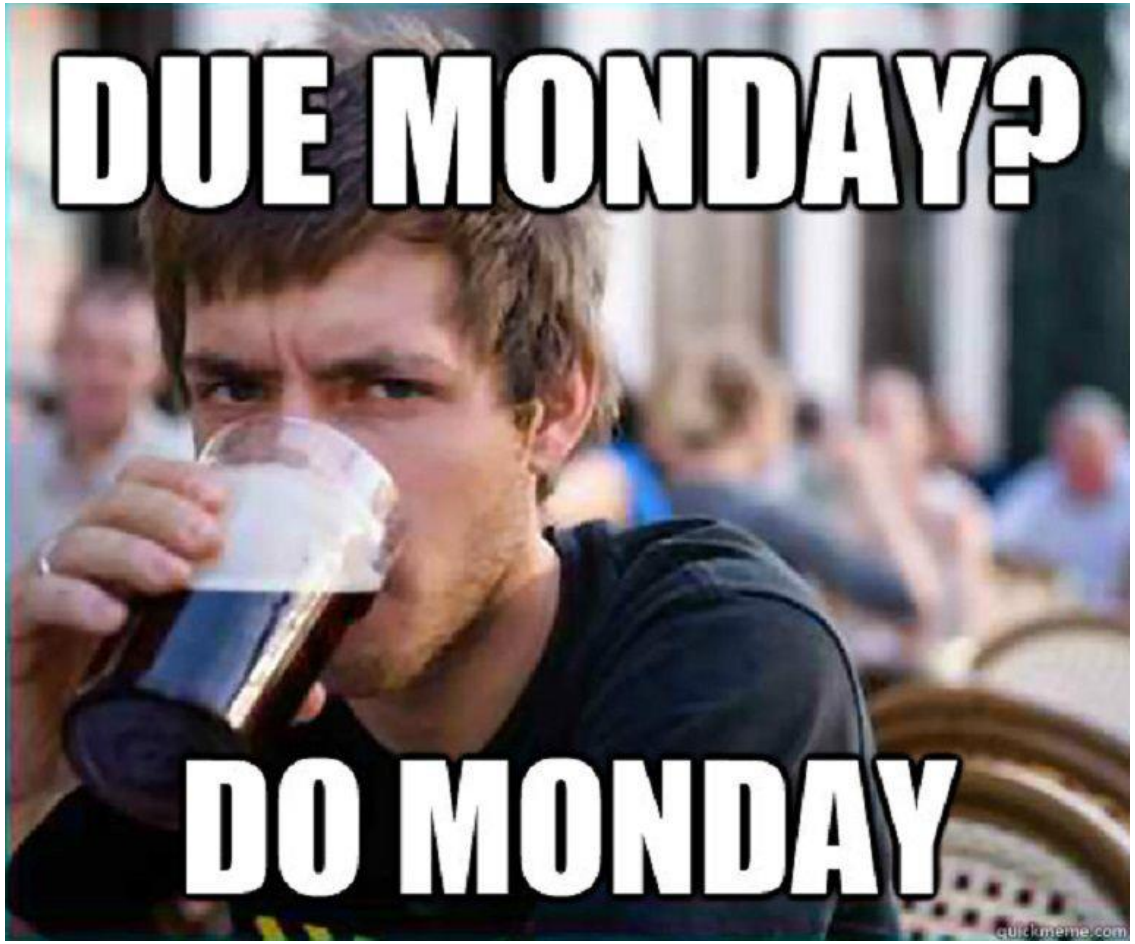
- **Automated tests on your solution**
  - We give you the scripts to build the code and test the basics
- **Bonus point for milestones**.
  - Bonus points = insurance
- **Required / recommended**
  - Attending the practical sessions
  - Submitting the milestones
  - Following along with the rhythm of the sessions in order to complete the integrated project in time

Don’t postpone the project till the last week …

This guy is a software engineer,
you can tell by his awesome
estimation skills

… or even the last day

- We can see the timestamps of your files when you submit your zip ☺
- We check for plagiarism, including all solutions from past years
  - Using 10 lines of code from [ref]
  - Copying 40% .. 70% is never ok.



Administrative

Team

- Ludo Bruynseels
- Toon Dehaene

- Dario Nucibella
- Belgin Ayvat

- Bert Lagaisse

Communication via Toledo

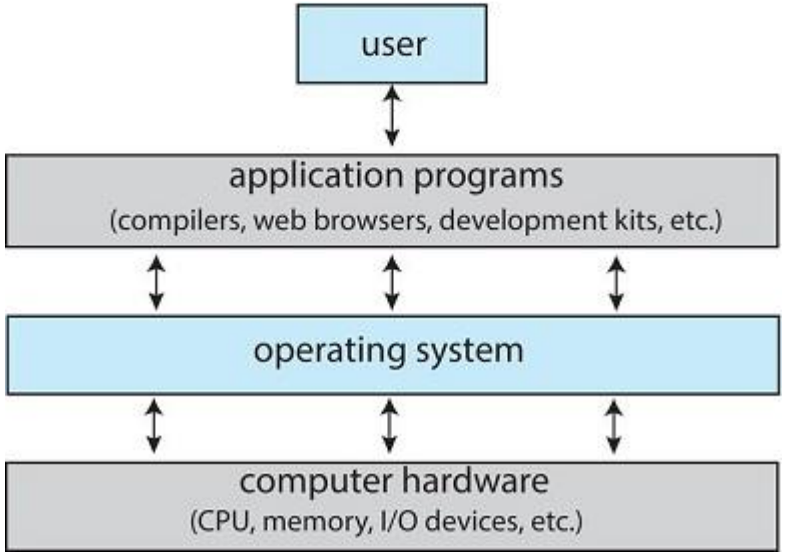
- Slides
- Labs:
  - Assignments
  - Handing in milestones
- Forum
  - Students helping students
  - No code sharing ;-)

Chapter 1
Introduction

KU LEUVEN

1.1 What is an operating system ?

- Software
  - That manages the computer's hardware
  - Basis for application programs
    - Hides and coordinates hardware complexity
- Allocates resources to programs
  - CPU
  - Memory
  - I/O devices
  - Storage
- Positioned in the middle between
  - Application programs
  - Computer hardware



Different point of views

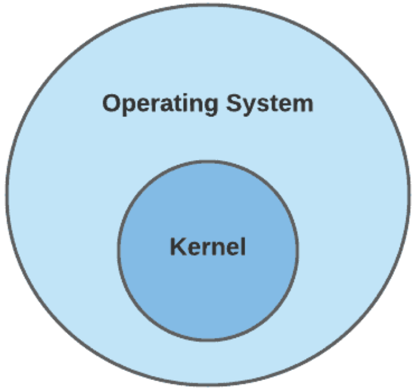
1.1.1 User view

- Human computer interaction
  - Mouse and keyboard
  - Touchscreen
  - Voice recognition (siri)
  - **Command line**
- OR: no user view
  - Embedded computers
  - Home devices, cars

1.1.2 System view

- Intimately involved with the hardware
- OS = resource allocator. Manages
  - Memory
  - I/O
  - Storage
  - Cpu
- OS = Controller. Control program for
  - Execution of user programs
  - Operation and control of I/O

1.1.3 Hard to pinpoint
What is part of the OS ?



All bloatware on your laptop ?

- All the software put on your laptop?
- Gigabytes of tools
- Graphical windowing systems
- A browser! (2001 antitrust lawsuit US vs MS)

The essence ?

- The essence = The **kernel**
  - The one program (set of instructions) that is always running on your computer.
  - Core operations:
    - cpu scheduling, memory management
- In addition
  - **System programs**: part of OS
  - **Application programs**: not part of the operation of the system
- Middleware
  - Software frameworks with additional services for developers
  - Android: databases, multimedia libs

Quiz

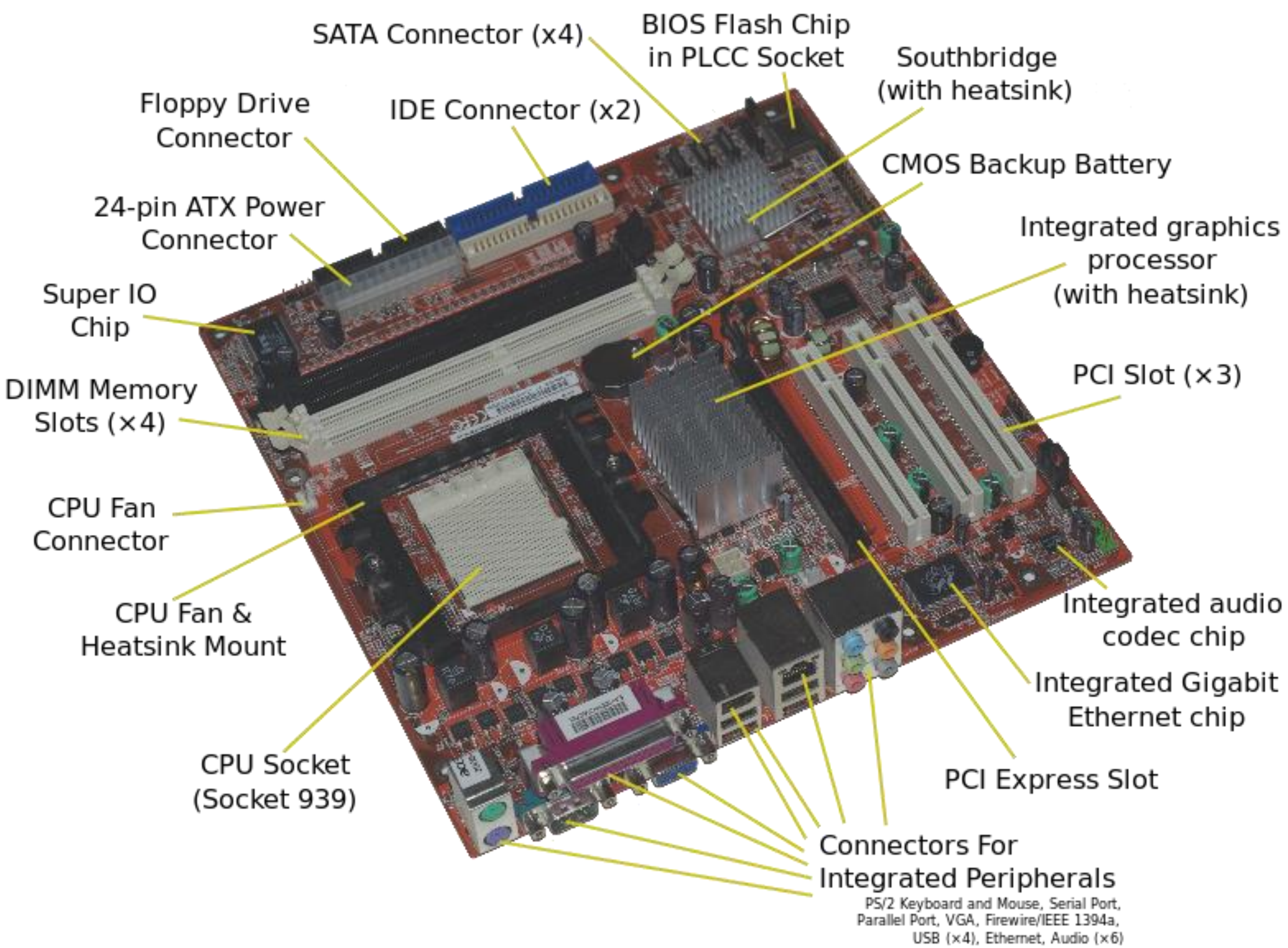
All computer systems have a kind of user interaction

- True
- False

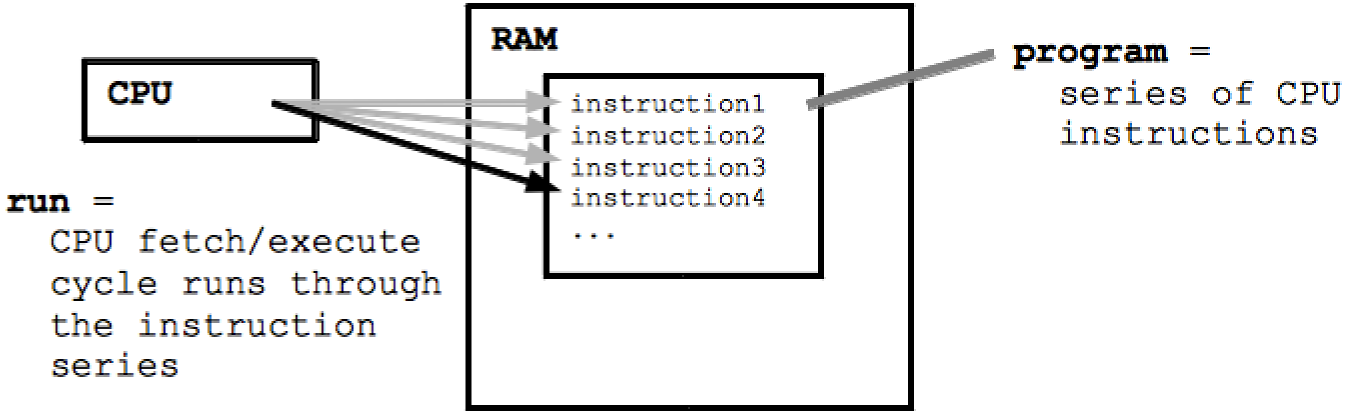
OS kernel =
system programs + application programs

- True
- False

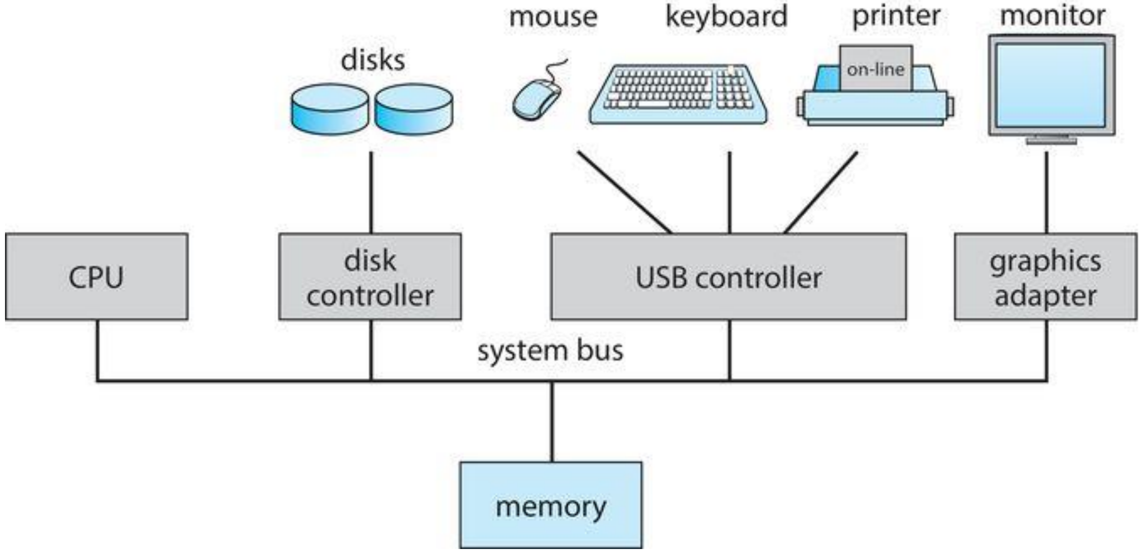
1.2 Computer-system organization
1.3 computer-system architecture
1.4 operating system operations
1.5 resource management
1.6 security and protection
1.7 virtualization


SATA Connector (x4)
BIOS Flash Chip in PLCC Socket
Southbridge (with heatsink)
Floppy Drive Connector
IDE Connector (x2)
CMOS Backup Battery
24-pin ATX Power Connector
Integrated graphics processor (with heatsink)
Super IO Chip
DIMM Memory Slots (×4)
PCI Slot (×3)
CPU Fan Connector
CPU Fan & Heatsink Mount
Integrated audio codec chip
Integrated Gigabit Ethernet chip
CPU Socket (Socket 939)
PCI Express Slot
Connectors For Integrated Peripherals
PS/2 Keyboard and Mouse, Serial Port, Parallel Port, VGA, Firewire/IEEE 1394a, USB (×4), Ethernet, Audio (×6)

General-purpose computer system



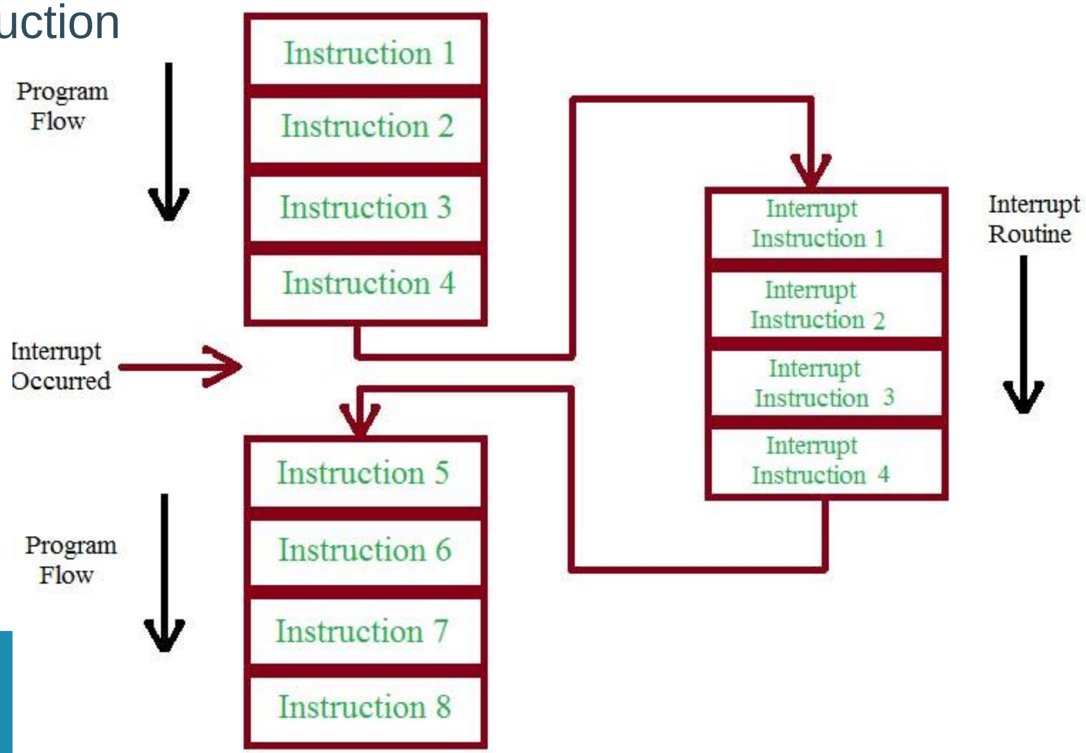
- CPU
- Device controllers
- Connected trough Common bus: access between
  - Components
  - Shared memory
- Operating system: Device driver
  - **for each device controller**
  - Understands the device protocol
  - Abstraction for rest of OS. Uniform interface to device
- CPU and device controller execute in parallel, competing for mem cycles
  - **Memory controller** synchronizes access



1.2.1 Interrupts

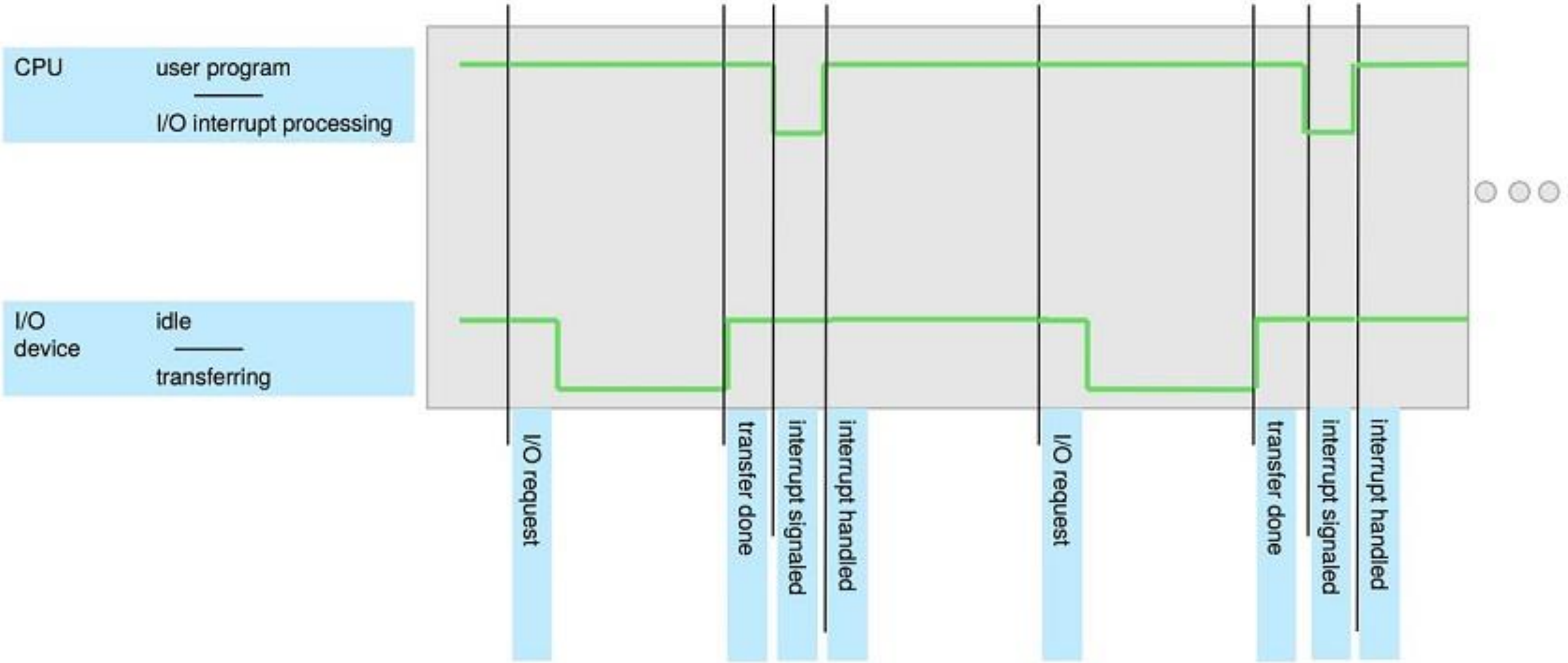
- **Start I/O operation**:
  - Device driver of OS loads registers of device controller with action to take
  - (e.g. read key from keyboard)
- **Device controller** examines contents of the registers
  - Determine which action to take
- **Controller transfers** data from device to its local buffer
- **Controller returns control** back to driver via interrupt

- Hardware triggers interrupt via signal to CPU over bus
- CPU transfers execution to appropriate interrupt routine
- After routine completes, CPU resumes previous instruction



E.g: interrupt timeline for program doing output

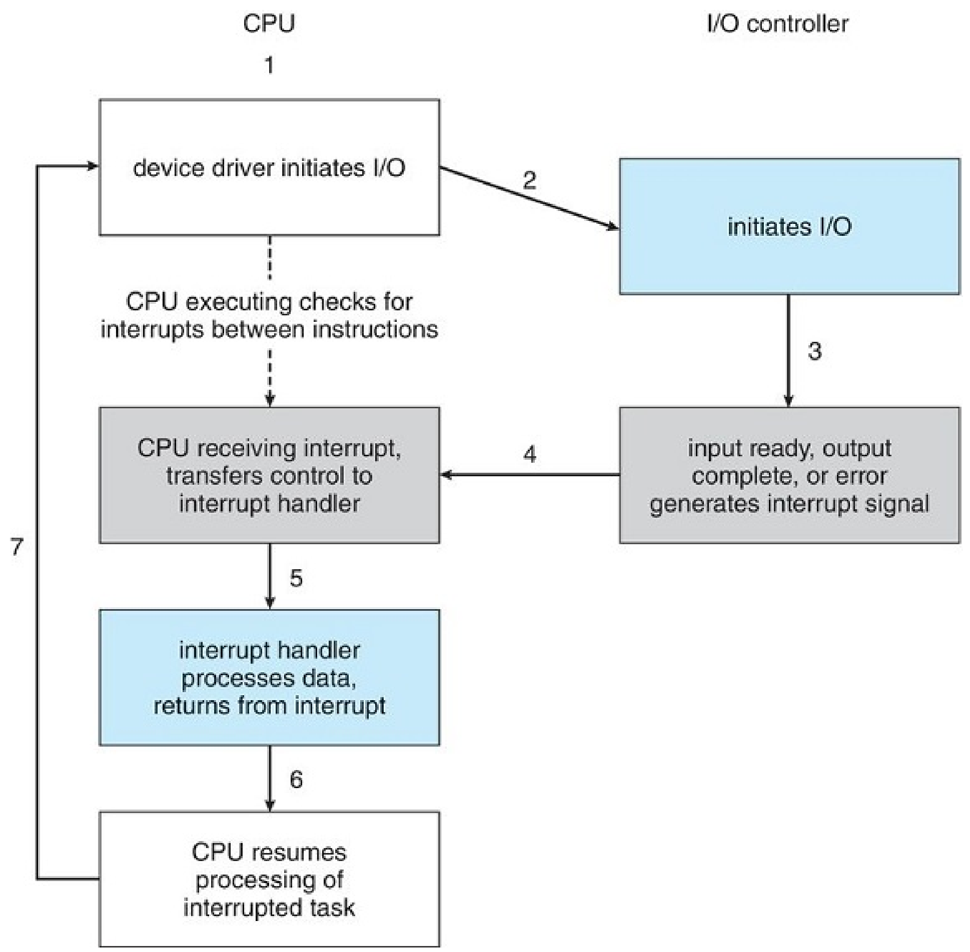
- Hardware may trigger interrupt at any time
  - by sending signal to CPU
  - On the system bus
- Interrupt vector:
  - A table of pointers to interrupt routines
  - Array of addresses
  - Index = number of interrupt request



KU LEUVEN

1.2.1.2 Implementation: interrupt mechanism

- **Interrupt-request line**
  - Wire in the CPU hardware
  - Sensed after each instruction
  - When triggered, cpu reads interrupt number
- **Interrupt-handler routine**
  - Interrupt number -> index to interrupt vector with routines per number
  - Stores state of previous instruction
  - Processes interrupt
  - Restores state and executes *return* instruction
- **More sophistication needed in reality**
  - Defer interrupt handling
  - Multi-level interrupts based on importance
- **Solution: 2 interrupt lines**
  - nonmaskable interrupts: for urgent errors
  - Maskable interrupts: can be turned off



e.g. Intel processor event-vector table

0-31: non-maskable

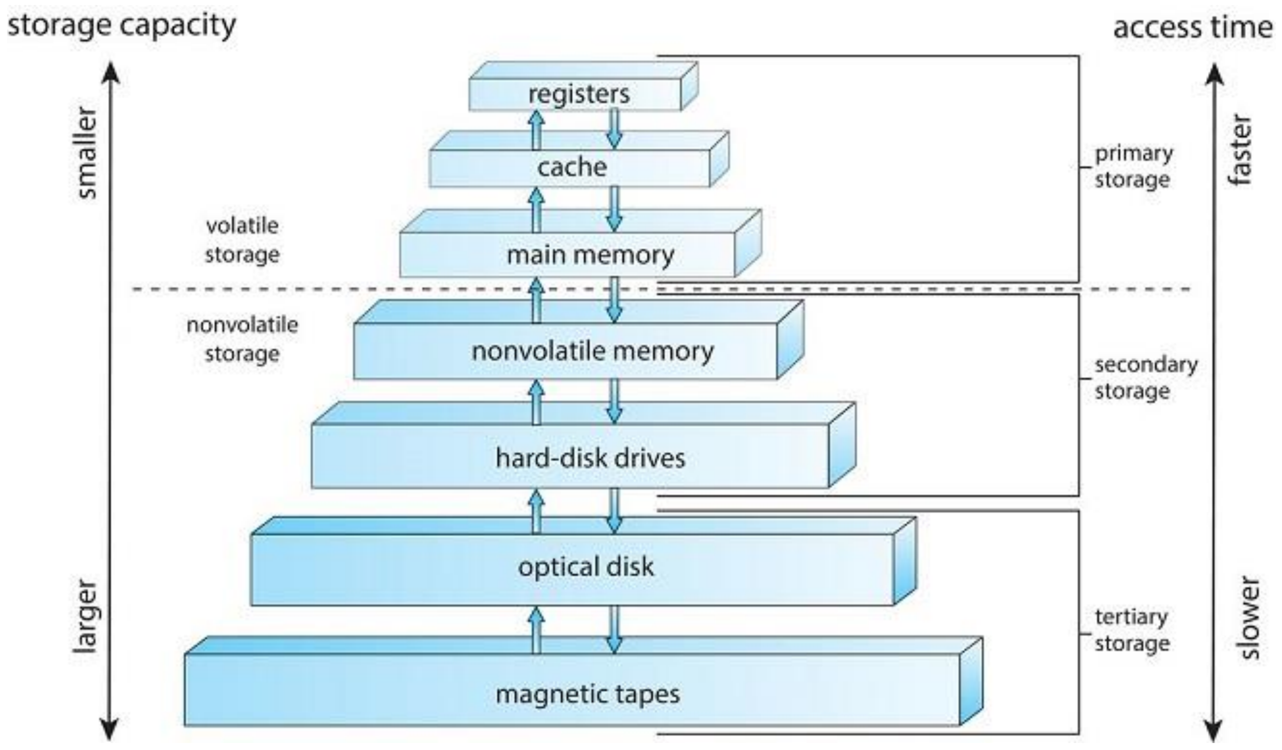
vector number	description
0	divide error
1	debug exception
2	null interrupt
3	breakpoint
4	INTO-detected overflow
5	bound range exception
6	invalid opcode
7	device not available
8	double fault
9	coprocessor segment overrun (reserved)
10	invalid task state segment
11	segment not present
12	stack fault
13	general protection
14	page fault
15	(Intel reserved, do not use)
16	floating-point error
17	alignment check
18	machine check
19–31	(Intel reserved, do not use)
32–255	maskable interrupts

1.2.2 storage structure

CPU can only load instructions from <u>main memory</u>

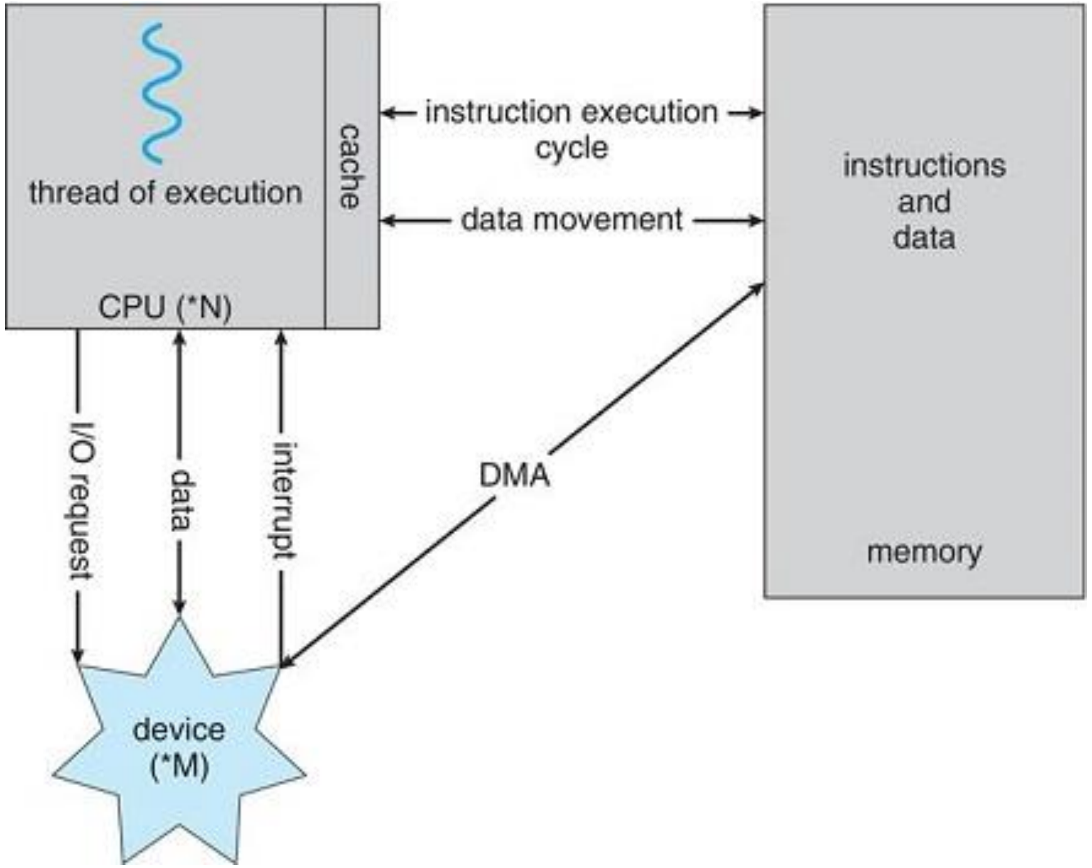
- = Rewritable memory
- **= Random-access memory**
  - Implemented using semi-conductor technology
  - e.g. DRAM: Dynamic RAM.
- **= array of bytes**
  - <u>Each byte has an address</u>
  - <u>Load/store instructions to access</u>
- **However**
  - Too <u>small</u> to store all running programs
  - <u>Volatile</u>: loses content when powered off
- **Secondary storage needed**
  - <u>Hard disk drives</u> (mechanical)
  - <u>Non-volatile memory</u> (electrical) e.g. SSD

Storage device hierarchy



1.2.3 I/O Structure
Need for Direct Memory Access (DMA)

- **Interrupt-driven I/O**
  - Fine for small amounts of data
  - High overhead for bulk data
- **DMA**
  - Device controller transfers entire block of data
  - directly to or from device and main memory
- **Interrupts**
  - Per block
  - Instead of per byte



Quiz

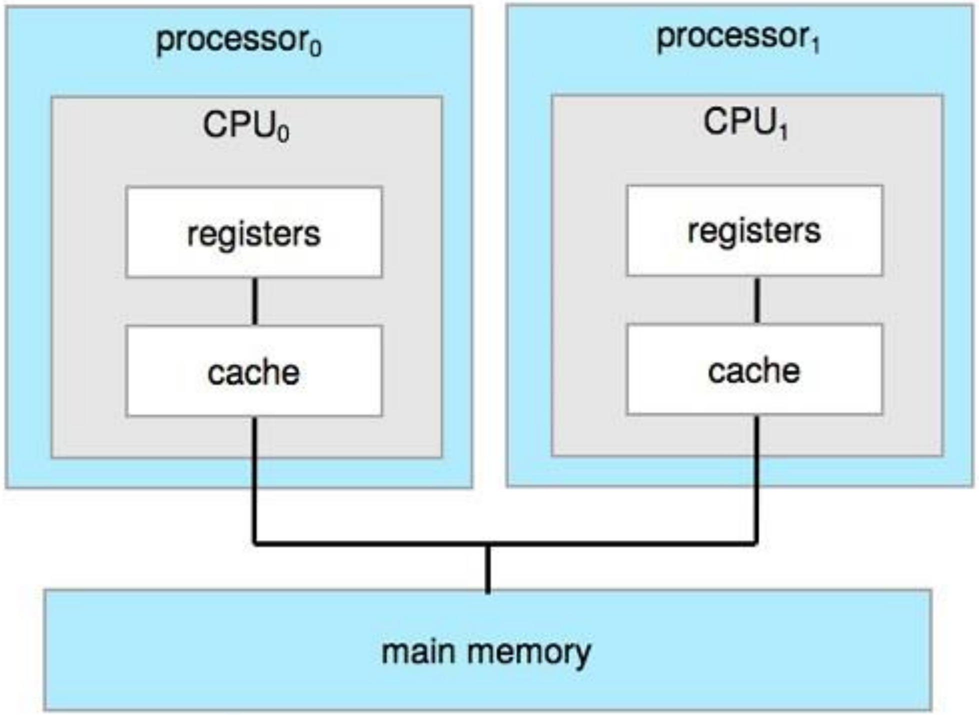
- Order the following storage mediums based on speed.

- CPU register
- Flash memory
- Hard disk drive
- DRAM

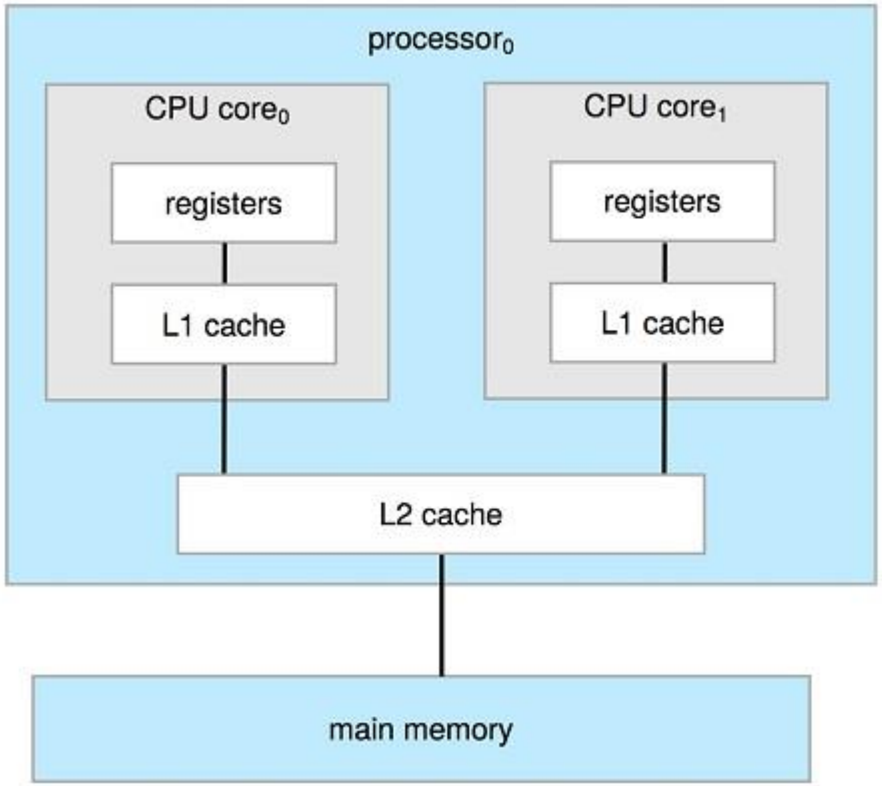
1.3 Computer-system Architecture

Evolution to dominant multiprocessor systems: on mobile devices and servers

Multiprocessor systems: multiple single-core CPUs on the same system bus

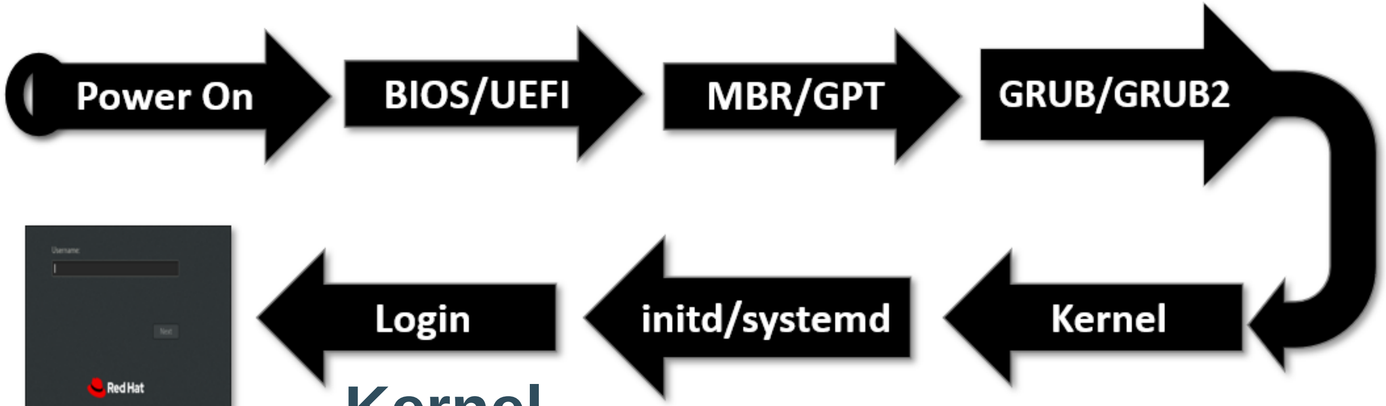


Multi-core system: 1 cpu, multiple cores



1.4 Operating-system operations

Getting started …



Bootstrap program

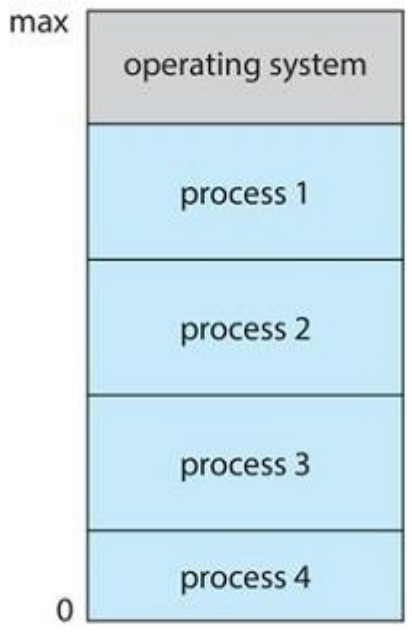
- Loaded after reboot/at start-up
- From hardware / firmware
- Initializes system
  - Cpu registers
  - Device controllers
  - Memory contents
- Loads OS kernel into memory from disk

Kernel

- **Once loaded, starts providing services**
  - To the system, Users. Applications
- **Some services run next to kernel**
  - System daemons
  - Loaded at boot time
  - On linux: “systemd” first service
- **OS waits when idle** (no i/o, no processes)
  - For interrupts from hardware
  - For traps from software (programs)
    - E.g. exceptions (divide by zero)
    - E.g. system calls from programs (write to file)

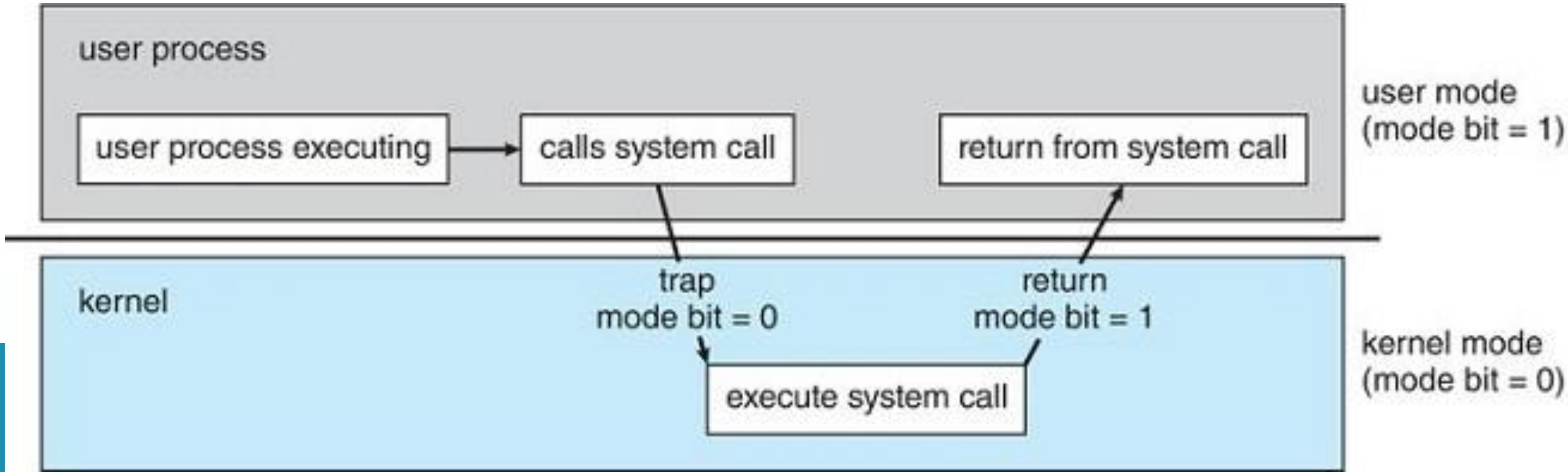
1.4.1 Multiprogramming and multi-tasking

- **Run multiple programs on one OS**
  - Increases CPU utilization
  - Organise programs such that CPU always has one program to execute
  - Process = a program in execution
- **Multi-programming**
  - Switch from one process to other when first process is idle (e.g. I/O)
- **Multi-tasking**
  - Switch more frequently (e.g. time based) for interactive experience.
- **Requires**
  - Appropriate CPU scheduling (Chapter 5)
  - Enough memory for all programs (Virtual Memory)



1.4.2 Dual-mode and Multi-mode operation

- **Protect OS from malicious programs**
- **Distinction** between execution of
  - OS code
  - User-defined code
- **Hardware support:** user mode vs kernel mode
  - Aka supervisor mode, privileged mode
  - Mode bit: kernel (0) or user (1)
- **Privileged instructions**
  - I/O control
  - Timer management
  - Interrupt management
  - Requires kernel mode
- **Attempting privileged instructions in user mode**
  - Not executed. Traps to OS.
- **Beyond 2 modes**
  - Intel: 4 protection rings
  - ARM v8: 7 modes
- **Can be used for virtualization**
  - Virtual machine manager (VMM)
  - OS < …< VMM < …< user process



Quiz!

What is another term for kernel mode?

1. supervisor mode
2. system mode
3. privileged mode
4. All of the above

Quiz!

What statement concerning privileged instructions is considered false?

A) They may cause harm to the system.

B) They can only be executed in kernel mode.

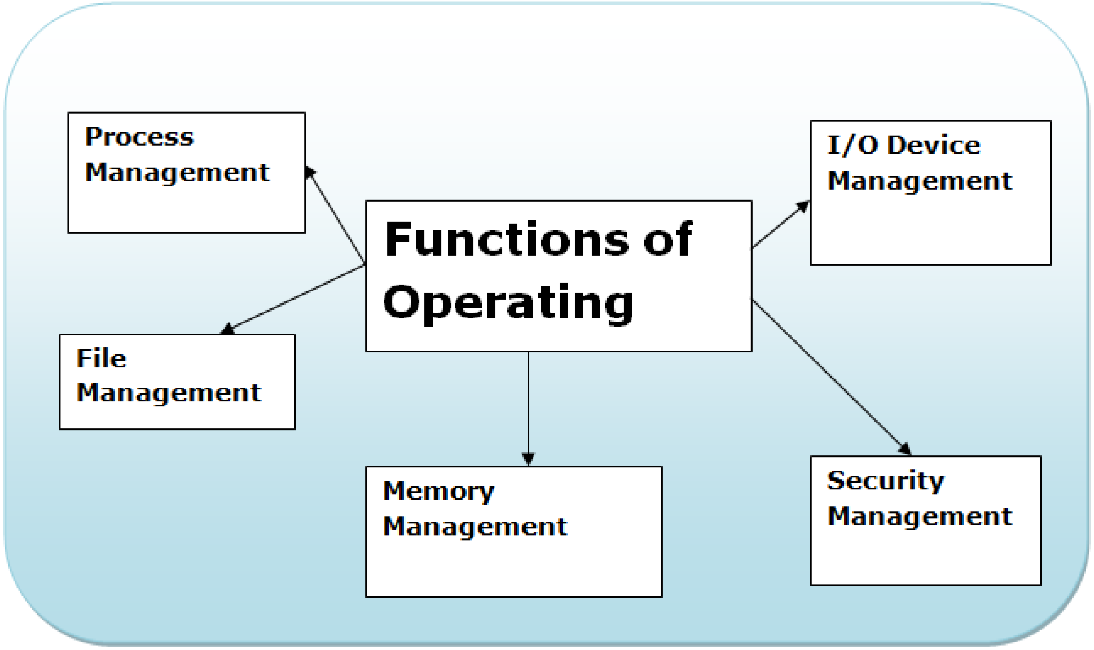
C) They cannot be attempted from user mode.

D) They are used to manage interrupts.

1.5 The operating system as resource manager

Process Management

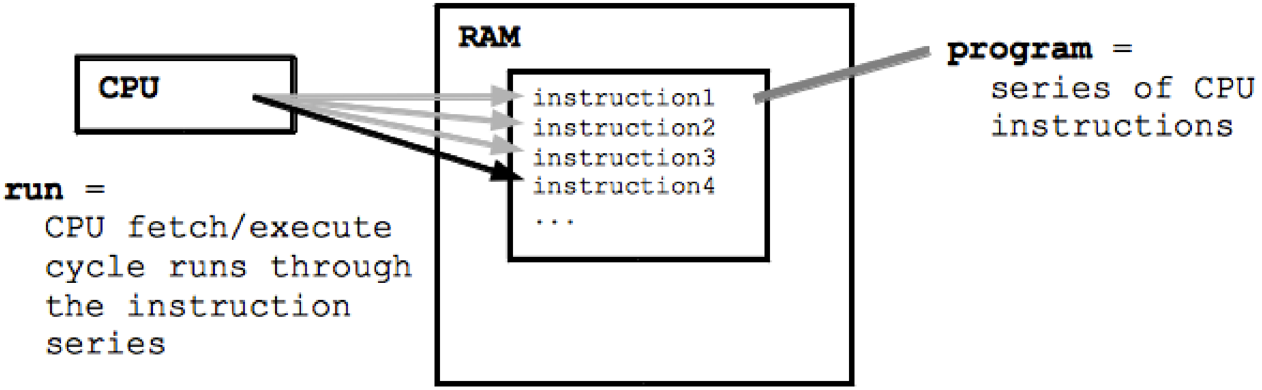
OS as a resource manager



OS responsible for process management

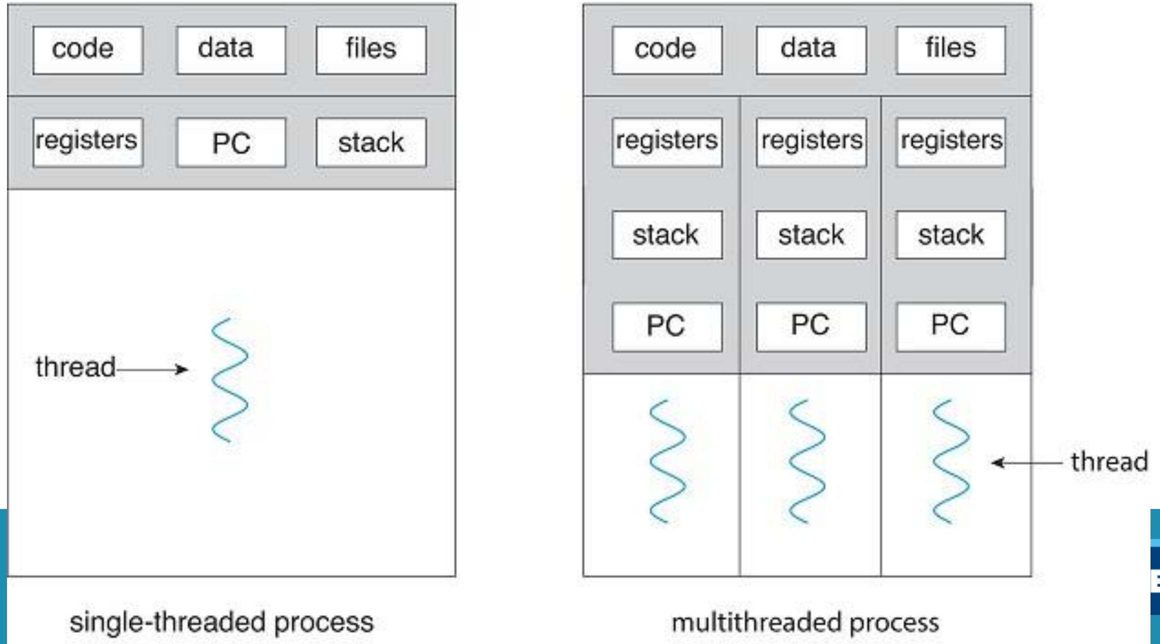
- Creating and deleting processes
  - User and system processes
- Scheduling processes on the CPU
- Suspending and resuming processes
- Mechanisms for process synchronization
  - Exclusive access to memory
- Mechanisms for process communication
  - E.g. shared memory
- Chapter 3-7 of the book

Process management



A process = a program in execution

- **Needs resources to execute**
  - CPU time
  - Memory
  - Files
  - I/O devices
- **Program counter:**
  - next instruction to execute
  - Per process
  - 2 processes of 1 program: 2 counters
- **Multi-threaded program**
  - Multiple threads of execution in 1 process
  - Program counter per thread

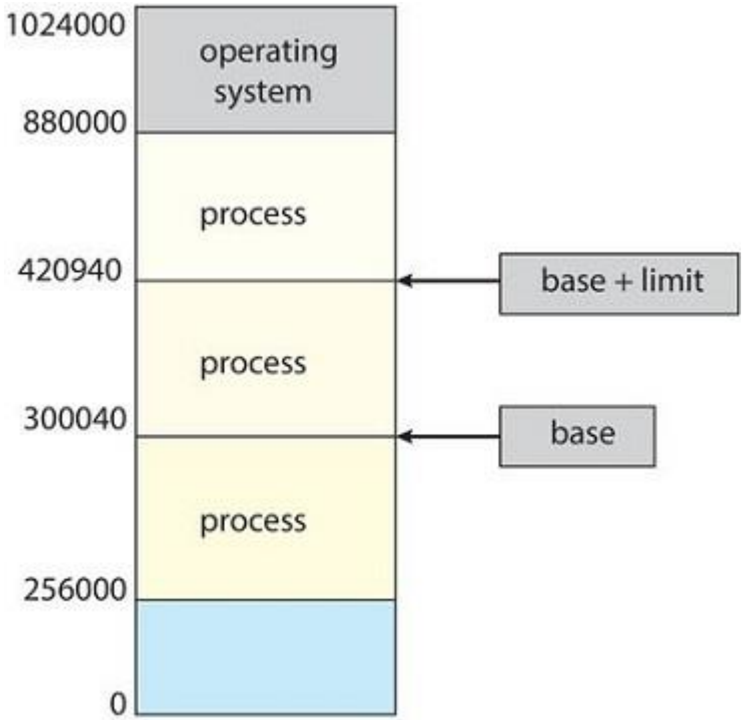


Memory management

Main memory

- **Large array of bytes**
  - Each byte has own address
- **Shared by CPU and I/O devices**
- **To execute a process**, required
  - instructions must be in memory
  - data must be in memory
- **Memory mapping to absolute address**
- **Memory management**
  - Which memory is being used
  - Which processes and data must be in memory and which must be moved out

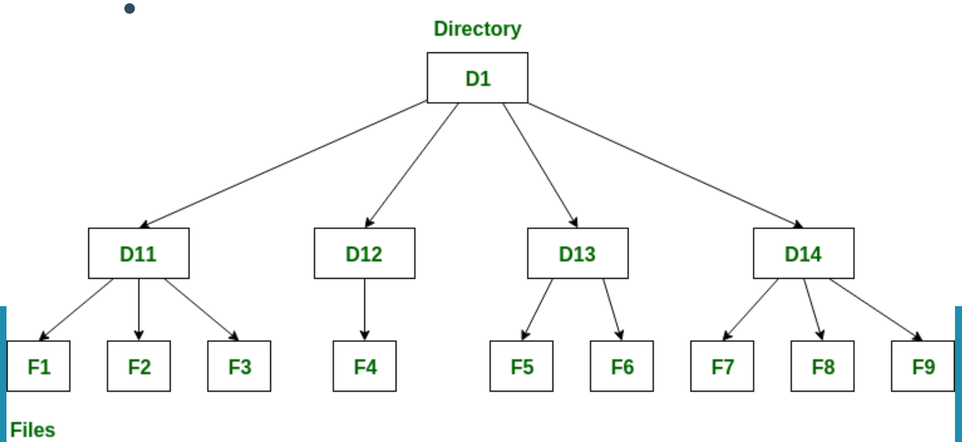
Memory mapping



Persistent storage management

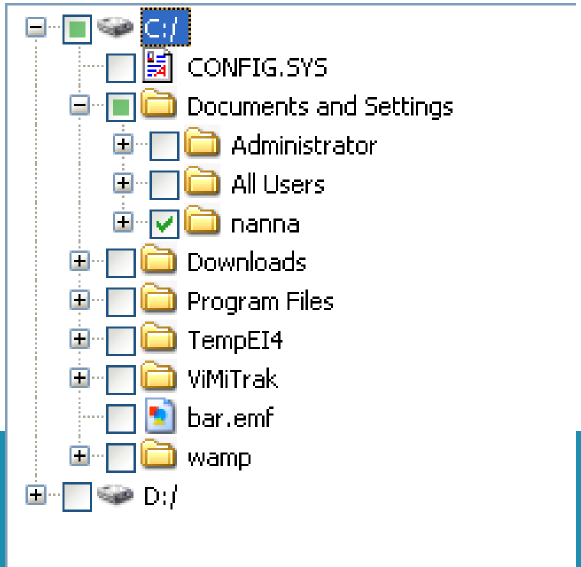
File-system management

- Files
- Directories
- User-based access control
- OS activities
  - Creating and deleting files
  - Creating and deleting directories
  - Mapping files onto mass storage
  -



Mass-storage management

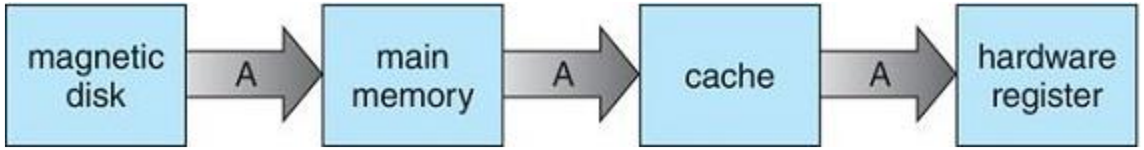
- HDD, NVM (SSD, Flash),…
- OS activities
  - Mounting and unmounting
  - Free-space management
  - Disk scheduling
  - …



Cache management

- **Caching**
  - Copy data to faster storage
  - On a temporary basis
- **Registers** are a high-speed cache for main memory
- **Instruction cache** in CPU
- Copying data →
  - Across hierarchy
  - Across multiple CPUs
  - Most recent value ?

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



Quiz !

There is one program counter…
for each

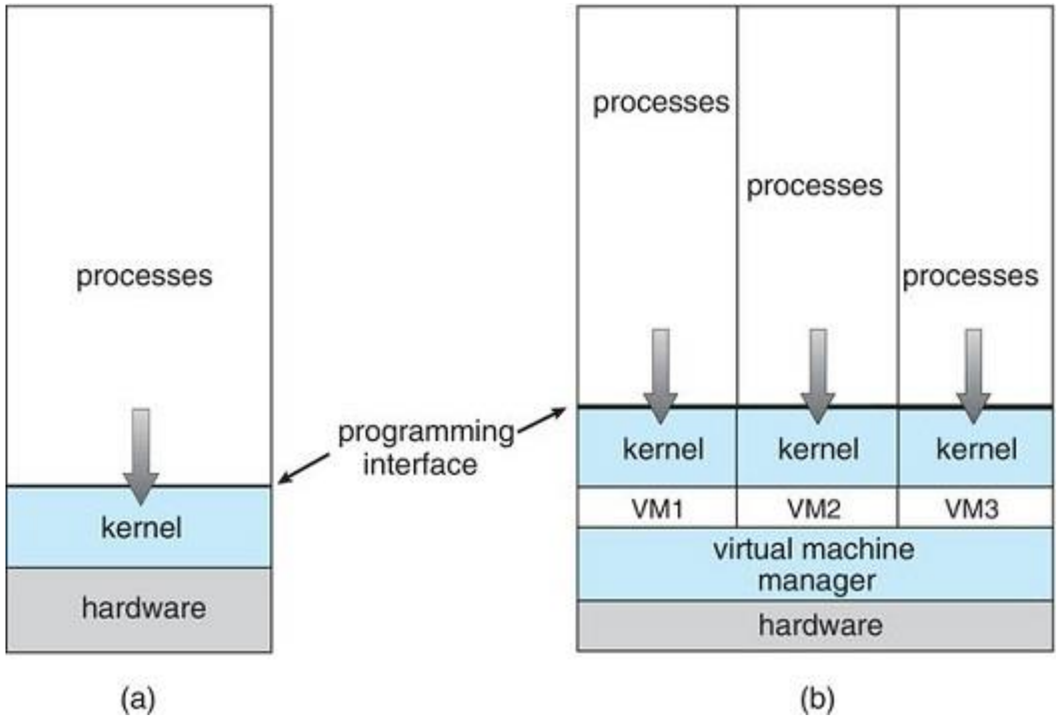
- Program
- Process
- Thread

Program counter =
current instruction to execute

- True
- False

1.7 Virtualization

- **Multiple virtual computers**
  - on 1 single computer
  - Running at the same time
- **Abstract and virtualize the hardware**
  - CPU
  - Memory
  - Disk drive
  - Network interface card
- Multiple virtual machines with their **own OS**
  - Guest OS
  - VMM runs on host OS (e.g. linux)
  - Example VMM: VirtualBox
  - Type 1 Hypervisor = VMM.
- Native VMM: VMM = host OS
  - VMWare ESX, Citrix XenServer
  - Type 2 Hypervisor = baremetal VMM



KU LEUVEN

The world of C: Essentials for OS

```c
#include <stdio.h>

int main() {
   /* my first program in C */
   printf("Hello, World! \n");

   return 0;
}
```

Interesting fact:

“C was invented to write an operating system called UNIX.”

```c
#include <stdio.h>

void f1();
static int globalvar = 10;
int externvar;
extern void externwrite();

int main(int argc, char** argv)
{
    /* comment */
    printf("hello from %s!\n", "TestAppInC");
    while (globalvar--) f1();

    externvar = 100;
    externwrite();

    return 0;
}
void f1() {
    static int localvar = 5;
    localvar++;
    printf("local is %d and global is %d\n",
       localvar, globalvar);
}
```

```c
#include <stdio.h>

extern int externvar;

void externwrite() {
    printf("externvar is %d\n", externvar);
}
```

- main function with args
- global vars, local vars
- types: int, long, float, …
- functions: f1()
    - Returns void or type
    - Declare before you use it.
- static local vars and global vars
- control: while, for(;;), if-then-else
- #include functions/types from libs via header file
- extern: defined outside of file
    - function
    - variables
    - example: main.c and support.c

Type	Storage size	Value range
char	1 byte	-128 to 127 or 0 to 255
unsigned char	1 byte	0 to 255
signed char	1 byte	-128 to 127
int	2 or 4 bytes	-32,768 to 32,767 or -2,147,483,648 to 2,147,483,647
unsigned int	2 or 4 bytes	0 to 65,535 or 0 to 4,294,967,295
short	2 bytes	-32,768 to 32,767
unsigned short	2 bytes	0 to 65,535
long	8 bytes or (4bytes for 32 bit OS)	-9223372036854775808 to 9223372036854775807
unsigned long	8 bytes	0 to 18446744073709551615

Type	Storage size	Value range	Precision
float	4 byte	1.2E-38 to 3.4E+38	6 decimal places
double	8 byte	2.3E-308 to 1.7E+308	15 decimal places
long double	10 byte	3.4E-4932 to 1.1E+4932	19 decimal places

sizeof(type)
literals

- Storage size
  - Of array
  - Of type
  - In bytes

- Literals
  - 'a' for a char
  - 85 /* decimal */
  - 0213 /* octal */
  - 0x4b /* hexadecimal */
  - 30 /* int */
  - 30u /* unsigned int */
  - 30l /* long */
  - 30ul /* unsigned long */

```c
#include <stdio.h>

int main()
{

    char    a       ='A';
    int     b       =120;
    float   c       =123.0f;
    double  d       =1222.90;
    char    str[]   ="Hello";

    printf("\nSize of a: %ld",sizeof(a));
    printf("\nSize of b: %ld",sizeof(b));
    printf("\nSize of c: %ld",sizeof(c));
    printf("\nSize of d: %ld",sizeof(d));
    printf("\nSize of str: %ld\n",sizeof(str));

    return 0;
}
```

Arrays

- Declaring
- Initializing
- Set value at i
- Get value at j

```c
#include <stdio.h>

int main () {

   int n[ 10 ]; /* n is an array of 10 integers */
   int i,j;

   /* initialize elements of array n to 0 */
   for ( i = 0; i < 10; i++ ) {
      n[ i ] = i + 100; /* set element at location i to i + 100 */
   }

   /* output each array element's value */
   for (j = 0; j < 10; j++ ) {
      printf("Element[%d] = %d\n", j, n[j] );
   }

   return 0;
}
```

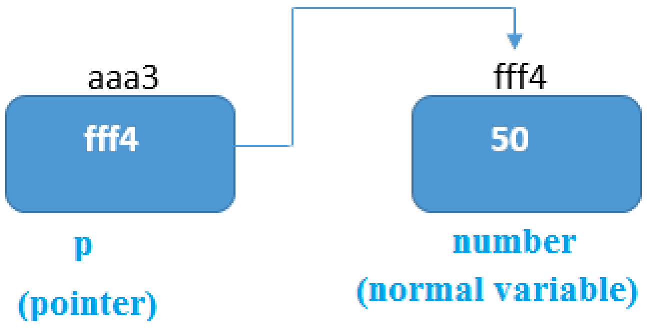
Types and pointers

Types

- char a = ‘c’;
- unsigned char
- signed char
- int
- unsigned int
- short
- unsigned short
- float
- double
- long double

```c
char c1, c2, *p;
c1 = ’c’;
p = &c1;
c2 = *p;
```

Pointers



- Points to
  - A memory location
  - Of a certain type
- Contains address of memory location
- & = address of
- * = content of

NULL Pointers.
Pass by value. Pass by reference.

Initialize to NULL pointer

- When exact address is unknown

```
#include <stdio.h>

int main () {

   int  *ptr = NULL;

   printf("The value of ptr is : %p\n", ptr  );

   return 0;
}
```

```
The value of ptr is 0
```

Quiz: output ?

```
# include <stdio.h>
void fun(int v)
{
    v = 30;
}

int main()
{
  int y = 20;
  fun(y);
  printf("%d", y);

  return 0;
}
```

```
# include <stdio.h>
void fun(int *ptr)
{
    *ptr = 30;
}

int main()
{
  int y = 20;
  fun(&y);
  printf("%d", y);

  return 0;
}
```

KU LEUVEN

Arrays and pointers

- Array names are *like* pointers
    - points to first element
- Pointer arithmetic
    - ptr++; → adds 1 * sizeof(int)
    - ptr--;
    - *(ptr + 1)
- QUIZ ME QUICK: OUTPUT ?

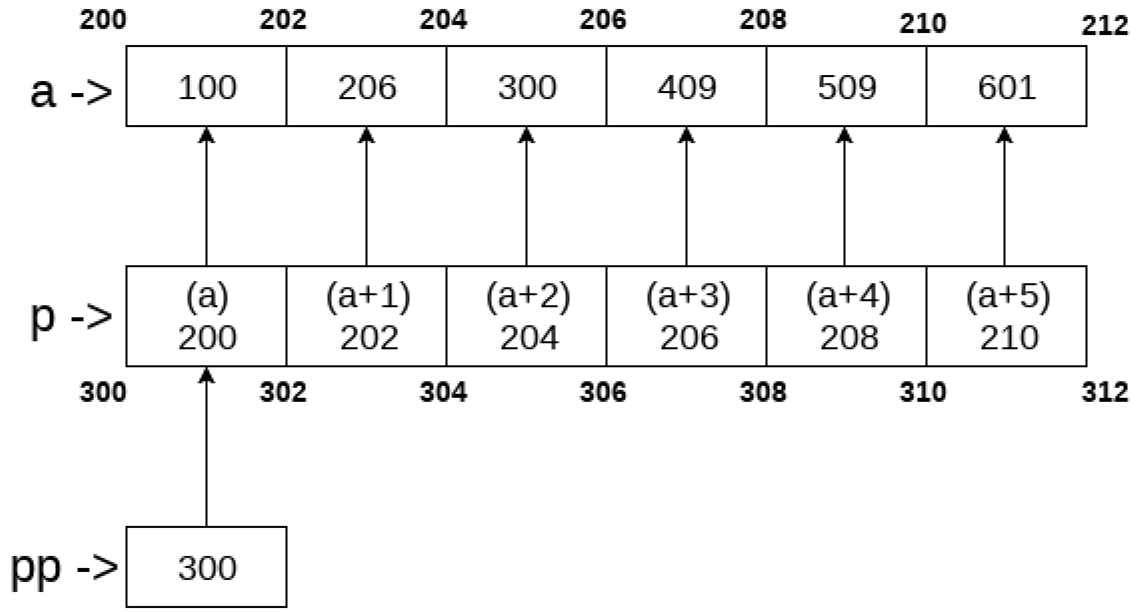
```c
#include <stdio.h>
int main() {

  int x[5] = {1, 2, 3, 4, 5};
  int* ptr;

  // ptr is assigned the address of the third element
  ptr = &x[2];

  printf("*ptr = %d \n", *ptr);
  printf("*(ptr+1) = %d \n", *(ptr+1));
  printf("*(ptr-1) = %d", *(ptr-1));

  return 0;
}
```

Arrays and pointers: Quiz me quick 2

Main args

- What is correct ?
  - int main()
  - int main(int argc, char **argv)
  - int main(int argc, char *argv[])
  - int main(void)

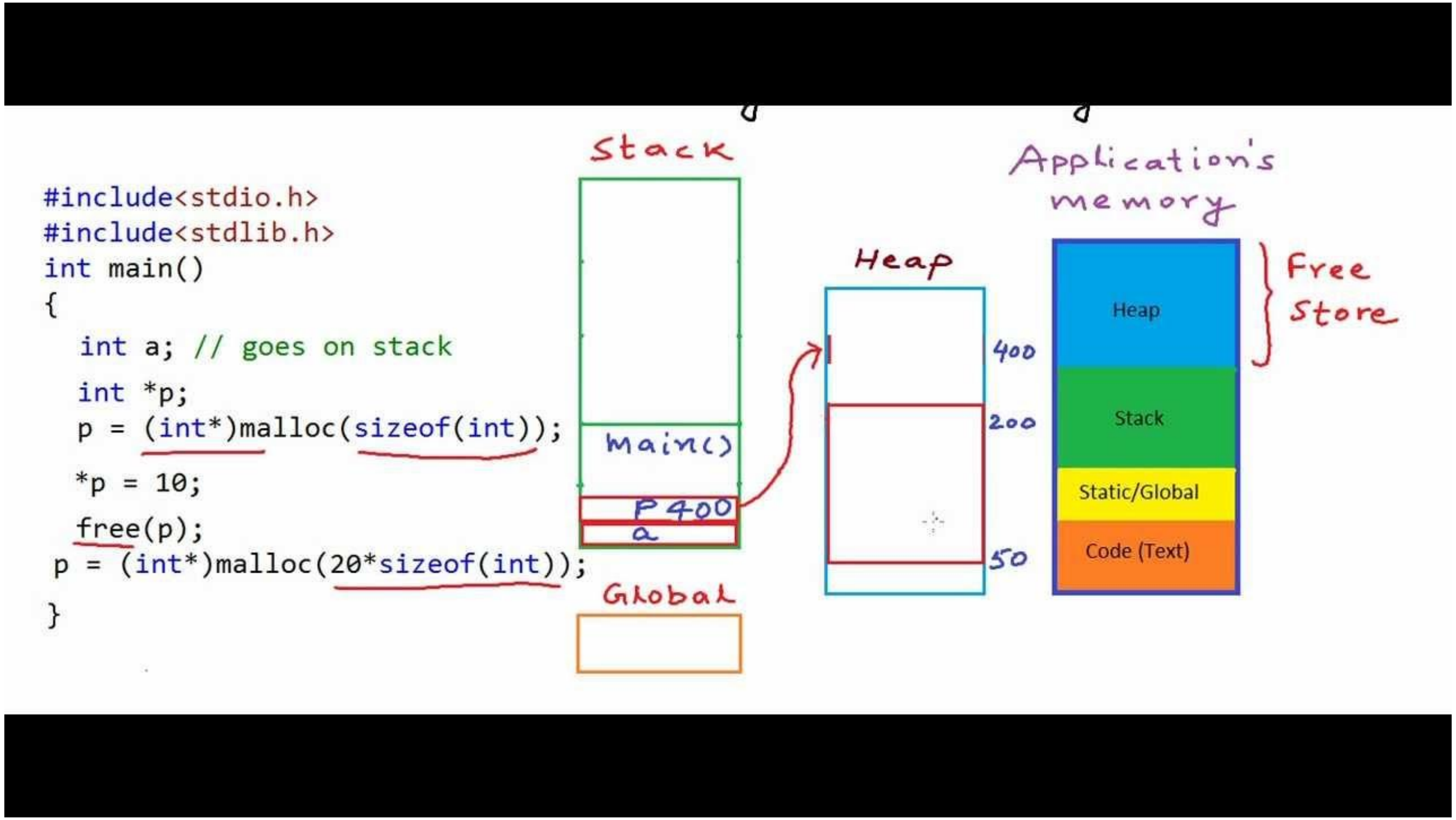
Pointers to pointers ?




```
int a[6];
int p[6];
int *pp;
```

To access a[0] → a[0] = * (a) = *p[0] = **(p+0) = **(pp+0) = 100

Stack vs heap



malloc() and free()

- **Stack vs heap**
  - Stack = temp data storage
    - During function invocation
    - Function parameters, local variables
  - Heap = memory to allocate dynamically
    - During program execution
    - Using malloc()
- **void * malloc(size_t size)**
  - Dynamic memory allocation on heap
  - Given size. Returns void* type.
- **Free()**
  - Deallocate the memory
  - Avoid memory leaks
  - Never reference a free pointer.

```c
#include <stdio.h>
#include <stdlib.h>

int main() {
  int n, i, *ptr, sum = 0;

  printf("Enter number of elements: ");
  scanf("%d", &n);

  ptr = (int*) malloc(n * sizeof(int));

  printf("Enter elements: ");
  for(i = 0; i < n; ++i) {
    scanf("%d", ptr + i);
    sum += *(ptr + i);
  }
  printf("Sum = %d", sum);

  // deallocating the memory
  free(ptr);

  return 0;
}
```

Typedef and structs

typedef

- Just a new name for a type
- E.g. more natural name
- Pure symbolic

```c
typedef unsigned char BYTE;
…
BYTE b1, b2;
```

struct

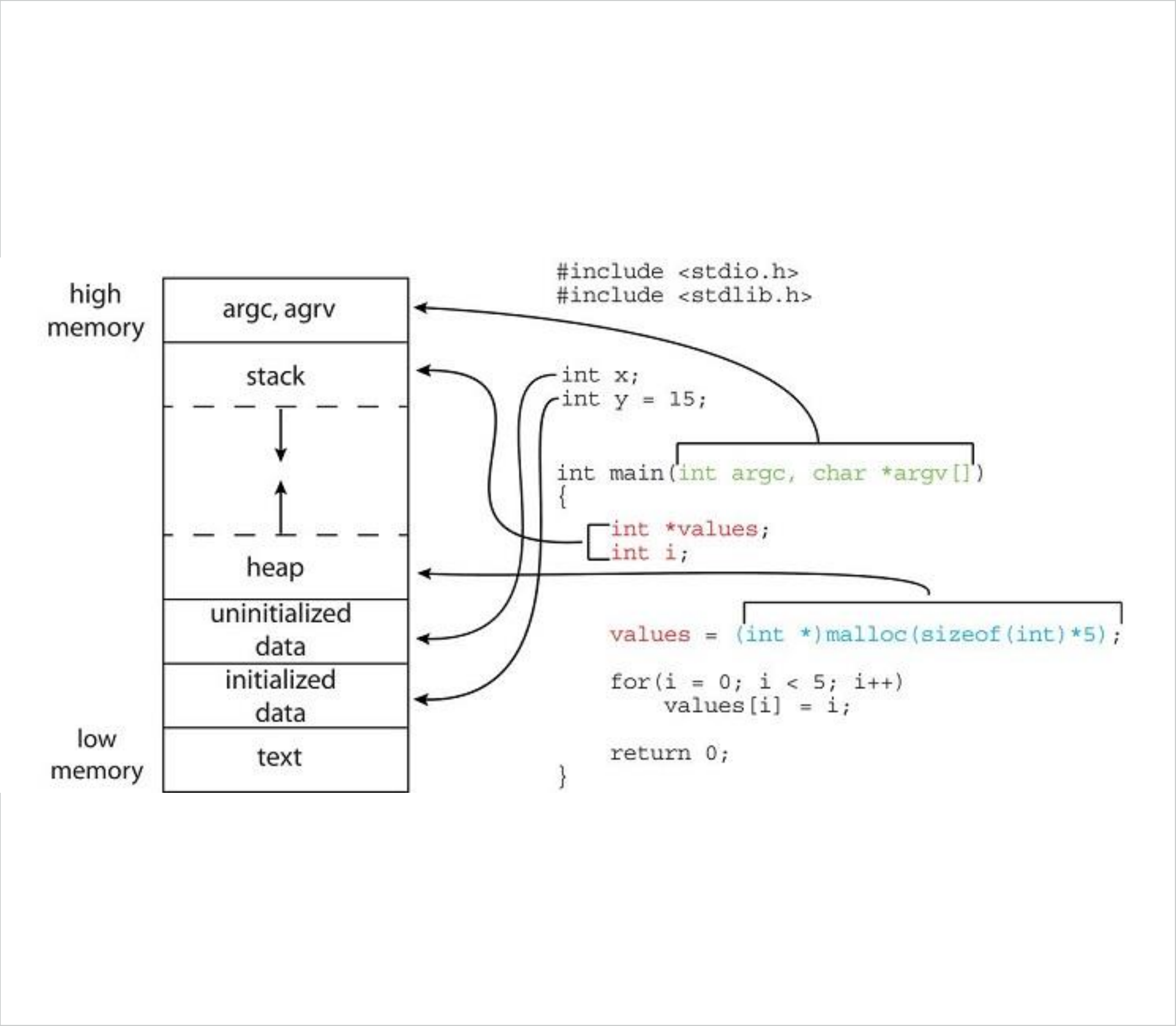
- Composite data type
- Often used together with typedef

```c
#include <stdio.h>
#include <string.h>

struct Book {
    char title[50];
    int book_id;
};

int main( ) {
    struct Book book;
    strcpy( book.title, "C Programming");
    printf( "Book title : %s\n", book.title);
}
```

```c
…
typedef struct Book {
    char title[50];
    int book_id;
} Book;

int main( ) {
    Book book;
    strcpy( book.title, "C Programming");
    printf( "Book title : %s\n", book.title);
}
```

Memory layout

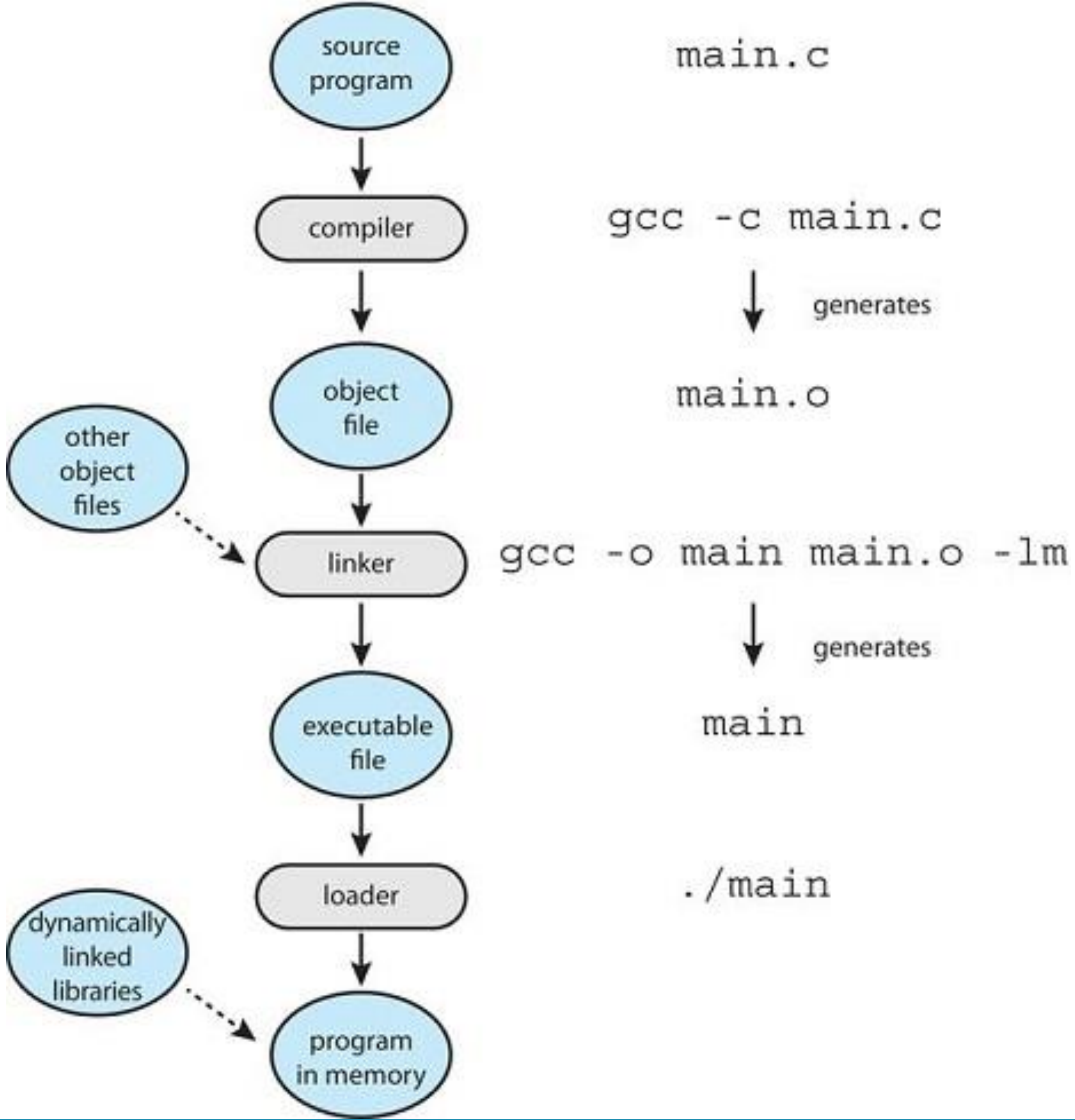
- Main memory = array of bytes
- Linearization of conceptual data structures in C program
  - Text
  - Initialized data
  - Uninitialized data
  - Heap
  - Stack
  - argc, argv



From C program to executable code

Steps (main.c includes math.h)

- Compiler
  - Produces relocatable object file (.o)
  - Can be loaded into memory
- Linker
  - Links in static libraries, object files
  - -l<library>
  - Produces one executable
- Loader:
  - Load and execute executable
- In practice
  - gcc to compile and link into executable
  - ./ to execute



Preprocessor directives… = pure text manipulation

- Preprocessor commands start with #
  - #define
  - #include
  - #ifdef
  - #if
  - #else
  - #endif
  - …
- Macro’s:
  - Parameters are not replaced in strings
  - Stringizing: #a #b

```c
#include <stdio.h>
#include "myheader.h"

#define MAX_ARRAY_LENGTH 20

#undef  FILE_SIZE
#define FILE_SIZE 42

#ifndef MESSAGE
   #define MESSAGE "You wish!"
#endif

#ifdef DEBUG
   /* Your debugging statements here */
#endif

#define square(x) ((x) * (x))

#define  message_for(a, b)  \
   printf(#a " and " #b ": Hello!\n")

int main(void) {
   message_for(Carole, Debra); //not a function call !
   return 0;
}
```

Header files and inclusion

- header.h

```
char *test (void);
```

- program.c

```
int x;
#include "header.h"

int main (void) {
   puts (test ());
}
```

- preprocessed program

```
int x;
char *test (void);

int main (void) {
   puts (test ());
}
```

```
#ifndef MY_HEADER_FILE
#define MY_HEADER_FILE

the entire header file file

#endif
```

- Avoiding double inclusion: Include guards

```
//grandfather.h
…



//father.h
#include “grandfather.h”
…



//child.h
#include “father.h”
#include “grandfather.h”
…
```

```
//grandfather.h
#ifndef GRANDFATHER_H
#define GRANDFATHER_H
..
#endif

//father.h
#ifndef FATHER_H
#define FATHER_H
..
#endif

//child.h
#include “father.h”
#include “grandfather.h”
…
```

1.9 Data Structures

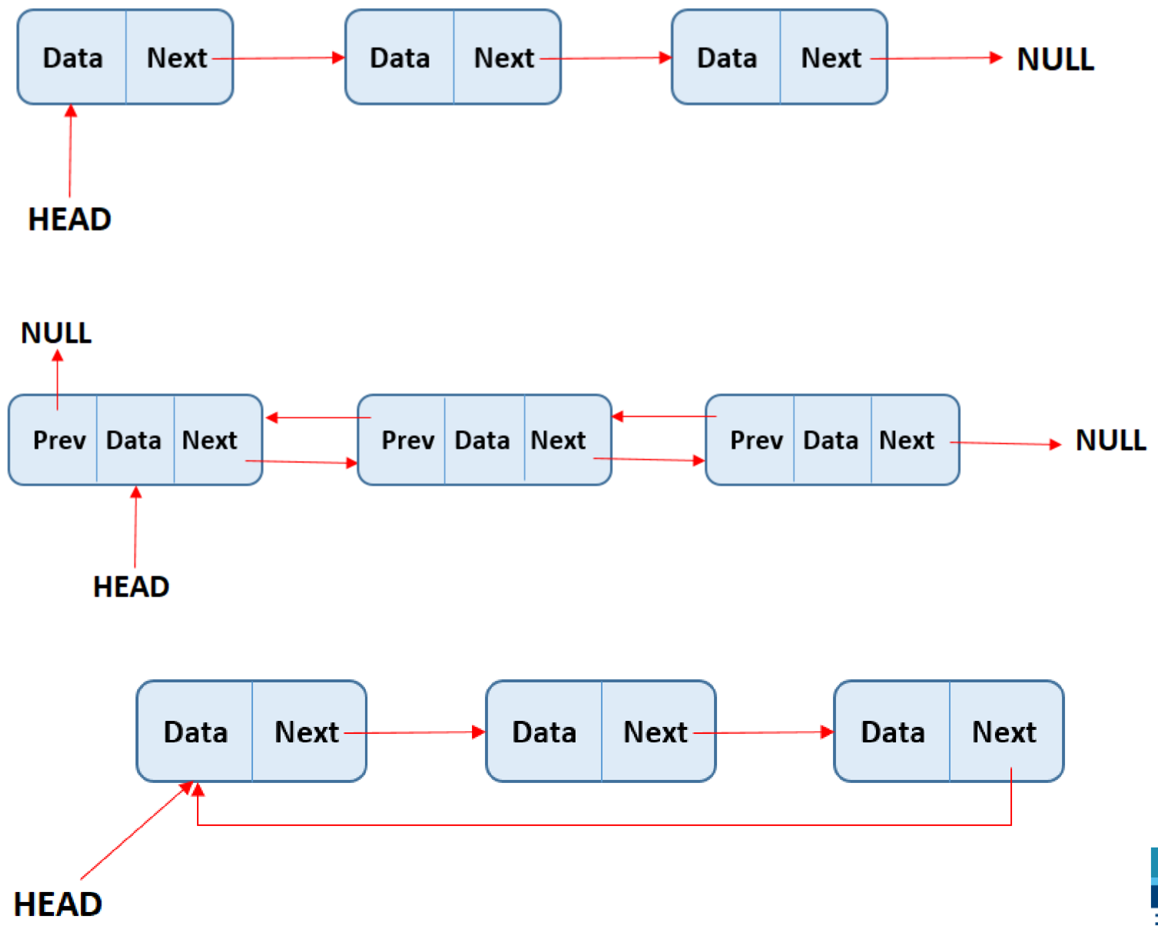
(used in OS's)
(in Kernels)
(in C)

1.9.1 Lists, stacks and queues

Beyond arrays

- **Array = simple data structure**
  - Fixed size when allocated
  - Each element accessible directly
- **Linked list:**
  - Singly linked list: each item points to successor
  - Doubly linked list: item refers to predecessor and/or successor
  - Circularly linked list: last element refers to first element, rather than to null
- **Stack**
- **Queue**

Linked lists

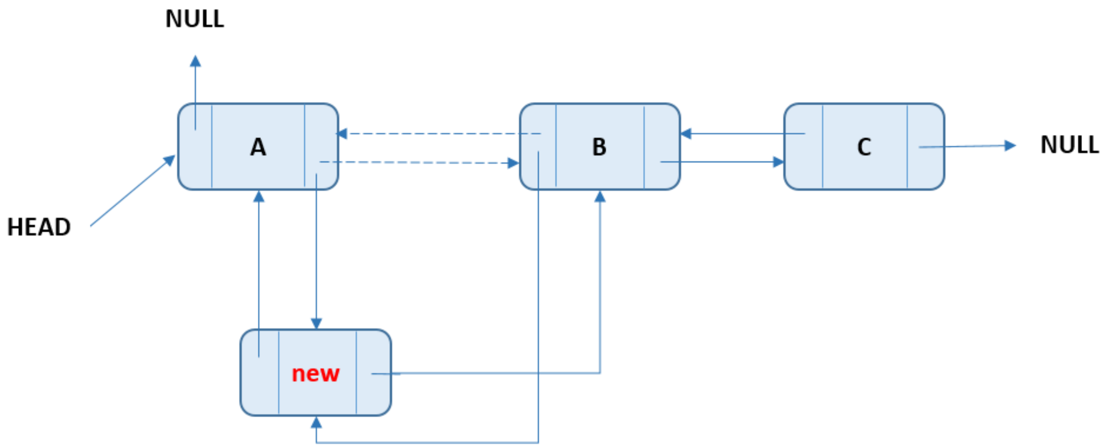


Inserting data into a doubly linked list

Steps

- Create a new node
- Check if position is > 0
- If position == 1 create new head
- Else
    - create temporary node
    - Traverse to node before position

Make a drawing !



Stack

```c
#include<stdio.h>
#include<stdlib.h>
// Struct to hold the data and the pointer to the next element.
struct element{
    char data;
    struct element* next;
};
void push(char data, struct element** stack){
    struct element* Element = (struct element*)malloc(sizeof(struct element));
    Element -> data = data;
    Element -> next = *stack;
    (*stack) = Element;
}
void pop(struct element** stack){
    if(*stack != NULL){
        printf("Element popped: %c\n",(*stack) -> data);
        struct element* tempPtr = *stack;
        *stack = (*stack) -> next;
        free(tempPtr);
    }
    else{
        printf("The stack is empty.\n");
    }
}
int main() {
    struct element* root = NULL;
 …
```

Conclusion

Programming with C takes time

- Practice pointers !
- Try some of the exercises in the book yourself while studying
- Small steps, each week.

Lectures and practical sessions !

- Attend both !
- Lectures and *plenary practicals*
  - Learn and understand the concepts
  - Bootstrap yourself for the practical sessions
- Practical sessions
  - Maximize their utility and advantages
  - Discuss with the session coach and co-students
  - Don’t copy !