

Systems Programming in C/C++

Mohammed Bahja

School of Computer Science

University of Birmingham

Outline of the module

- Computer architecture: programmer's perspective
- C programming
 - Structure of a C program
 - Pointers and memory management
 - Applications of memory management
- Programming with threads
- Kernel programming ans OS topics

What this module is Not

- Not a ‘Computer Architecture’ module.
- This module is different from non-CS modules on C
- Not a re-run of Software Workshop
- Not a C++ software development module

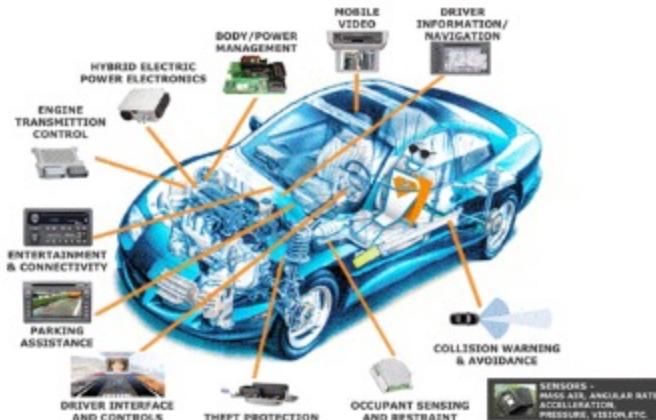
Assessment

- 50% exam, 50% coursework
- Description of coursework and assessment criteria will be made available on Canvas.
- There will be 3 challenging assignments.
(lab sessions will start from the second week).
- Make use of the labs (initially two hours per week) to work on the coursework problems.
- Will use virtual machine for exercises; see Canvas for details.

Study materials

- Recommended Course Books
 - The C Programming Language (2nd Edition) Kernighan and Ritchie
 - OS Concepts (10th Edition) Silberschatz et al.

Computers



Computers are everywhere

Computers: two types

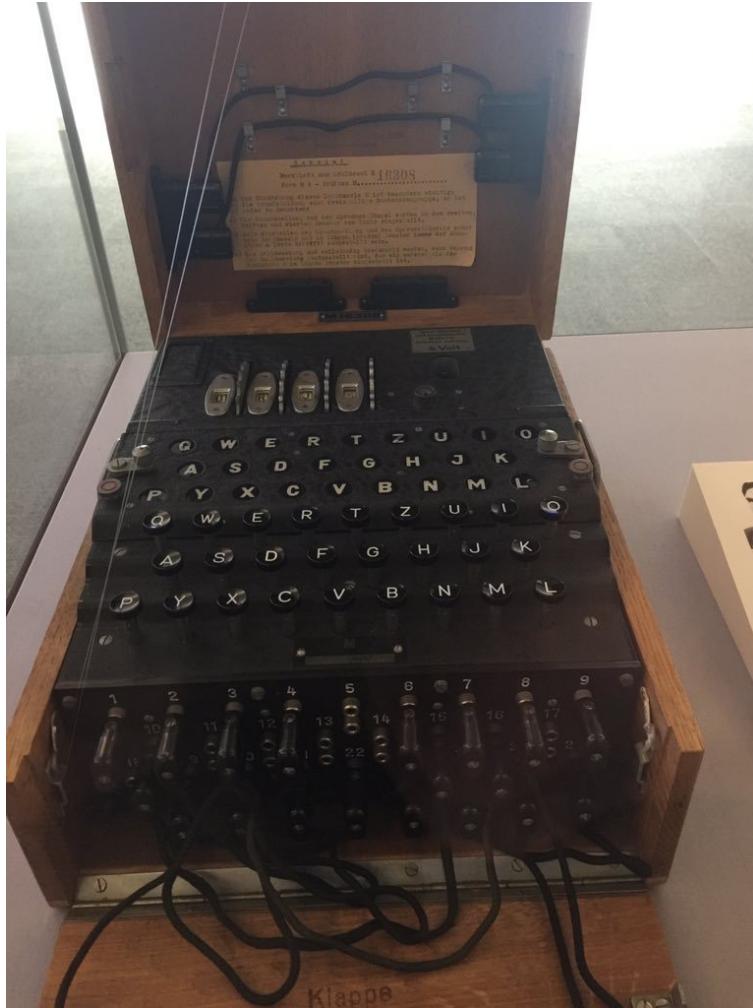
From application point of view, computers can be classified into two major categories.



Can perform specific tasks.
E.g. only simple calculations.
Application Specific Computer

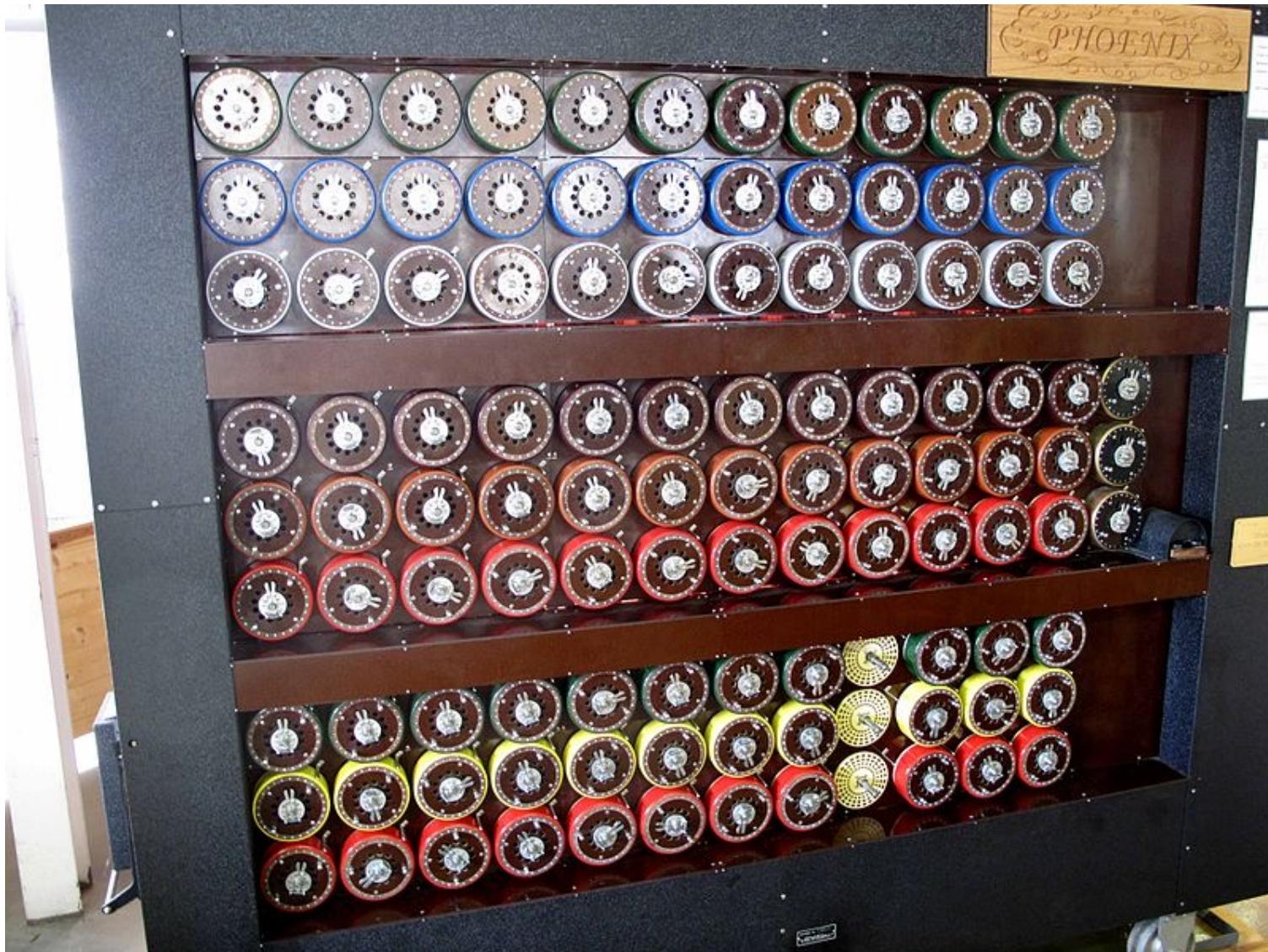


Can perform different tasks.
E.g. calculations, watch movies, play games, browse internet etc.
General Purpose Computer



Enigma machine used during World War II
(Photo of the machine at The Alan Turing Institute, London)

Can perform only encryptions → hence application specific



'British bombe' an electromechanical computer designed by Alan Turing to break codes produced by the Enigma machine

Major bottleneck: Programming required major re-wiring.



Snap from movie “The Imitation Game”.
Benedict Cumberbatch as Alan Turing.

Stored Program Computers

- The invention of stored program computers has been ascribed to John von Neumann.

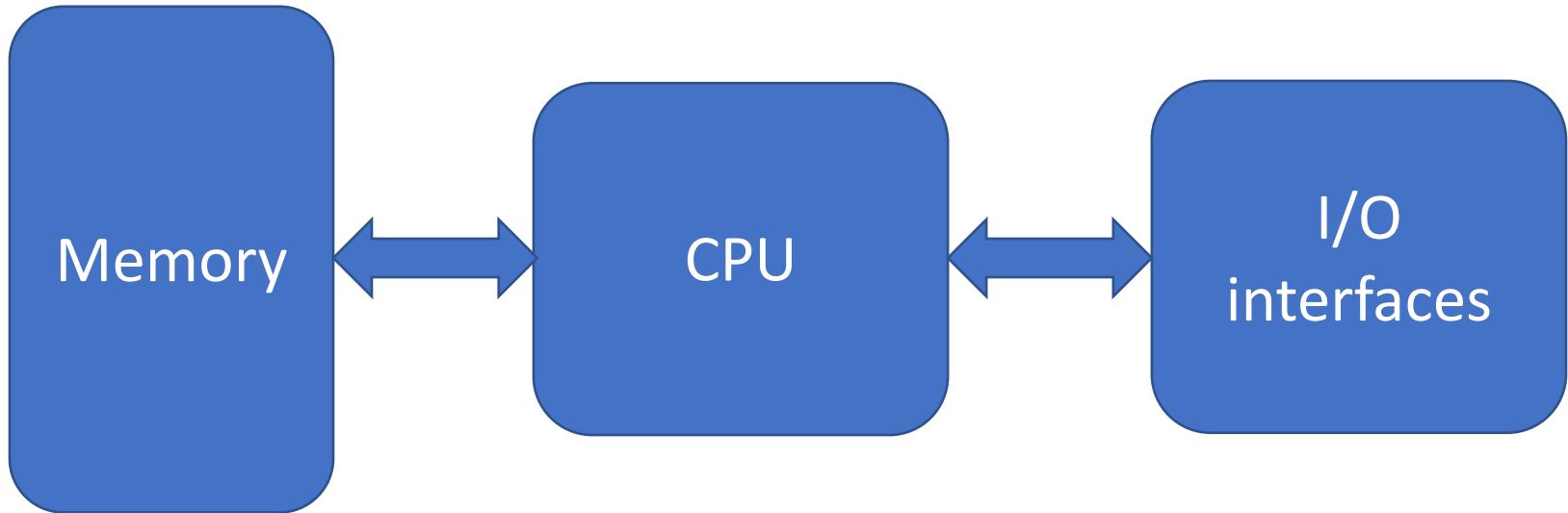


- Stored-program computers have become known as ‘von Neumann Architecture’ systems.

A ‘stored-program computer’ is a computer that stores program instructions in memory.

→ Re-programming does not require any hardware modifications.

The von Neumann Architecture



Consists of three main components

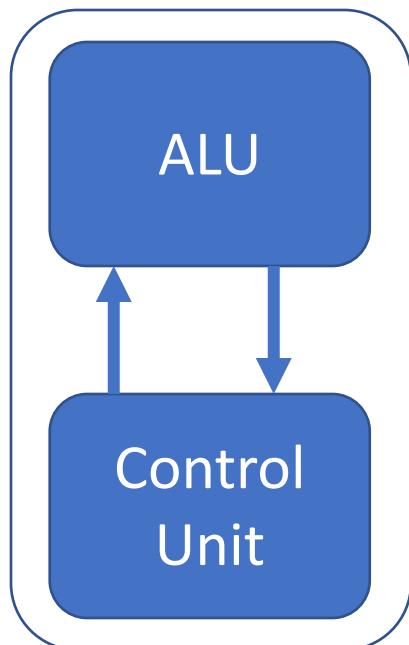
1. Central Processing Unit (CPU)
2. Memory
3. Input/Output (I/O) interfaces.

The CPU

The CPU can be considered the heart of the computing system.

It includes two main components:

1. Control Unit (CU),
2. **Arithmetic and Logic Unit (ALU)**



ALU performs the mathematical or logical operations.

Example:

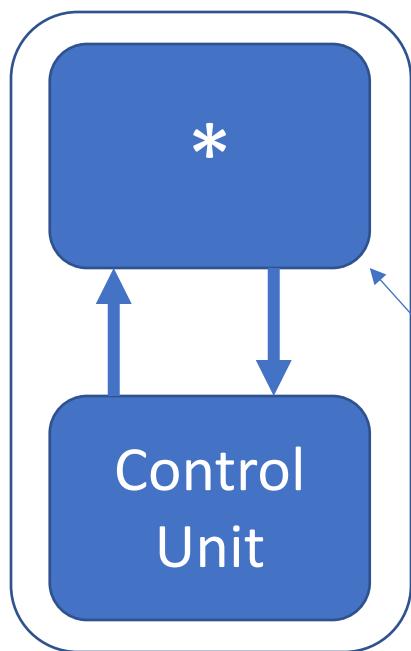
```
main ()  
{  
    int a=5, b=6, c;  
    c = a*b;  
    c = c + b  
}
```

The CPU

The CPU can be considered the heart of the computing system.

It includes two main components:

1. Control Unit (CU),
2. **Arithmetic and Logic Unit (ALU)**



ALU computes multiplication

ALU performs the mathematical or logical operations.

Example:

```
main ()  
{
```

```
    int a=5, b=6, c;  
    c = a * b;  
    c = c + b
```

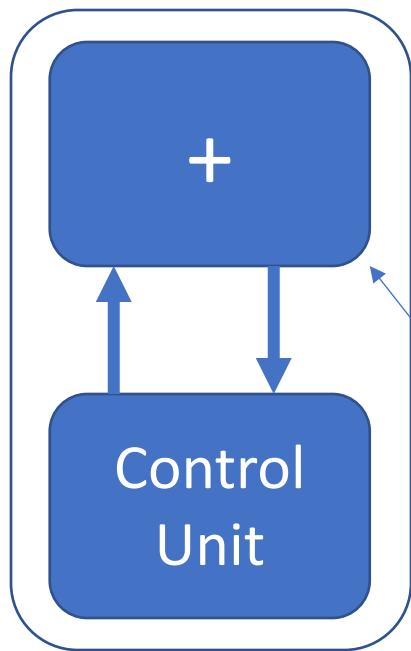
```
}
```

The CPU

The CPU can be considered the heart of the computing system.

It includes two main components:

1. Control Unit (CU),
2. **Arithmetic and Logic Unit (ALU)**



ALU computes addition

ALU performs the mathematical or logical operations.

Example:

main ()

{

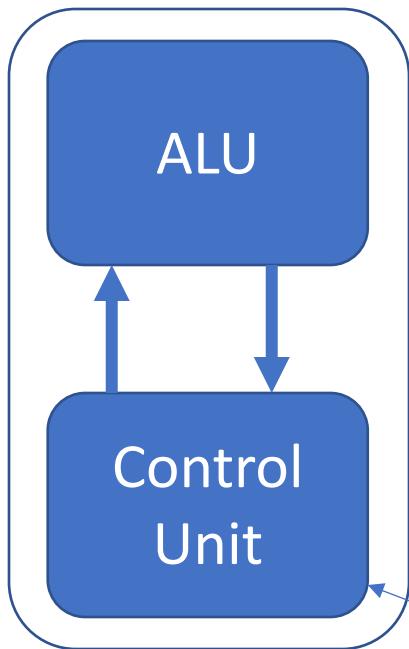
```
int a=5, b=6, c;  
c = a*b;  
c = c + b
```

}

The CPU

The CPU can be considered the heart of the computing system.
It includes two main components:

1. **Control Unit (CU),**
2. Arithmetic and Logic Unit (ALU)



Control Unit determines the order in which instructions should be executed and controls the retrieval of the proper operands.

Example:

main ()

{

int a=5, b=6, c;

c = a*b;

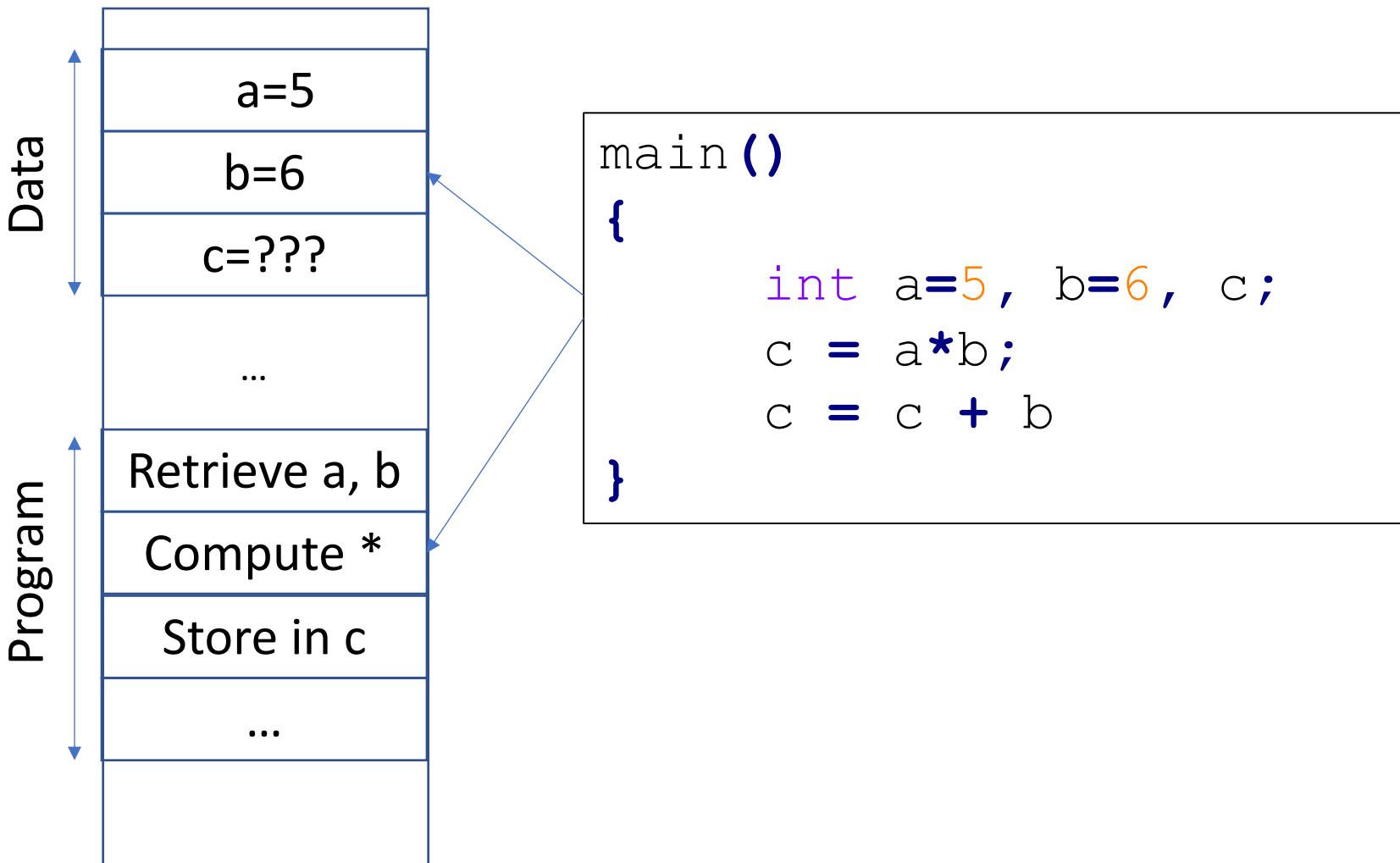
c = c + b

}

1. Retrieves the operands
2. Asks ALU to compute *
3. Stores the result
4. Jumps to the next line

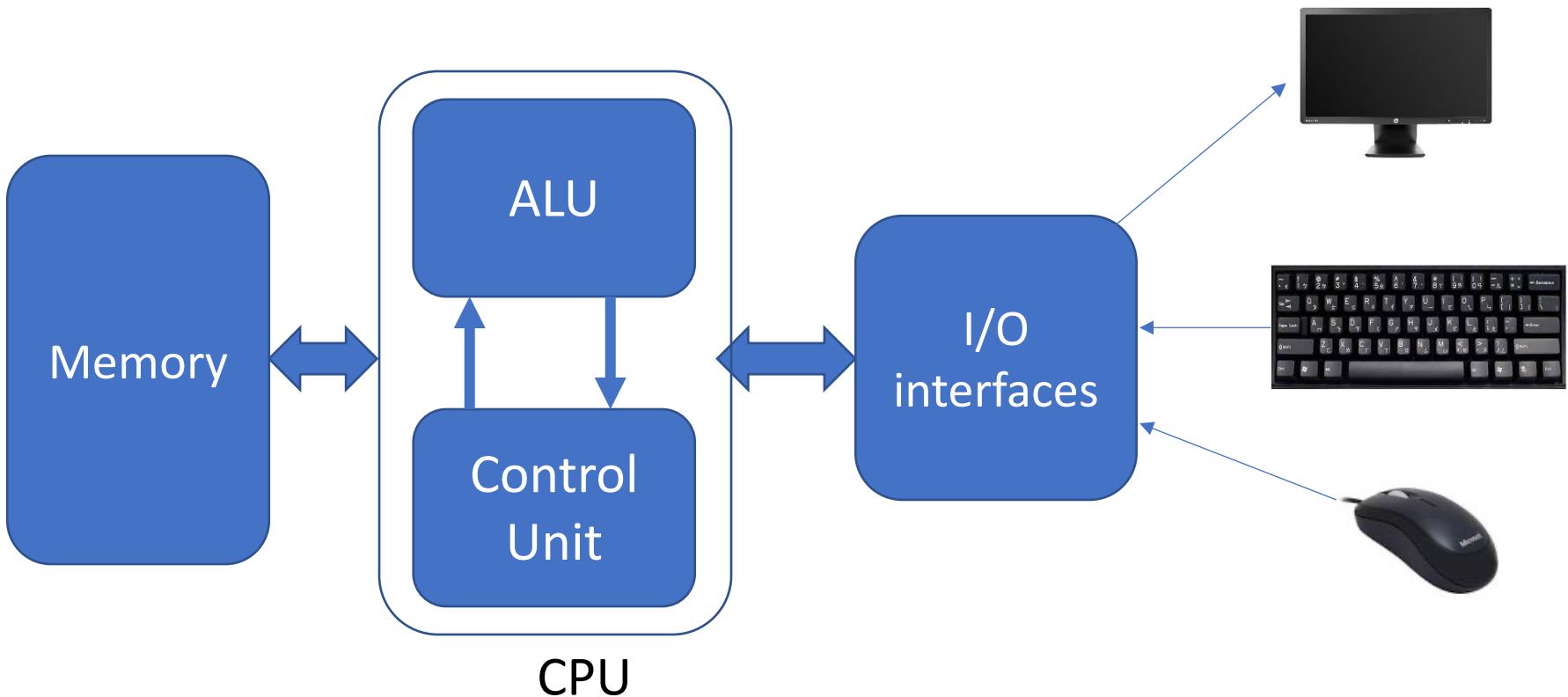
The Memory

The computer's memory is used to store both program instructions and data.



The Input/Output Interfaces

- The I/O interfaces are used to receive or send information from/to connected devices.
- Connected devices are called **peripheral devices**.



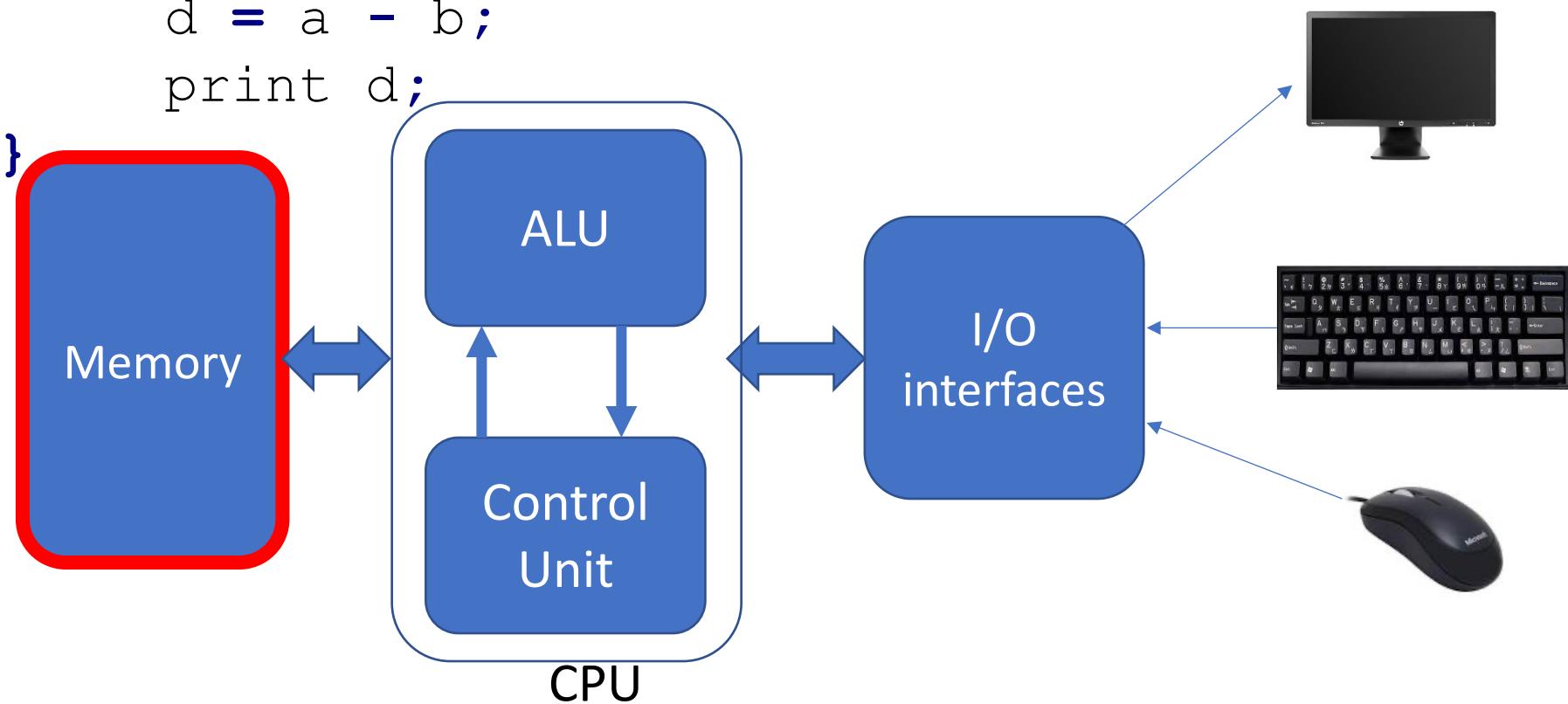
Program execution on von Neumann computer

```
main() {
```

```
    readIO a;  
    readIO b;  
    c = a + b;  
    store c;  
    d = a - b;  
    print d;
```

```
}
```

A program is stored in the memory.

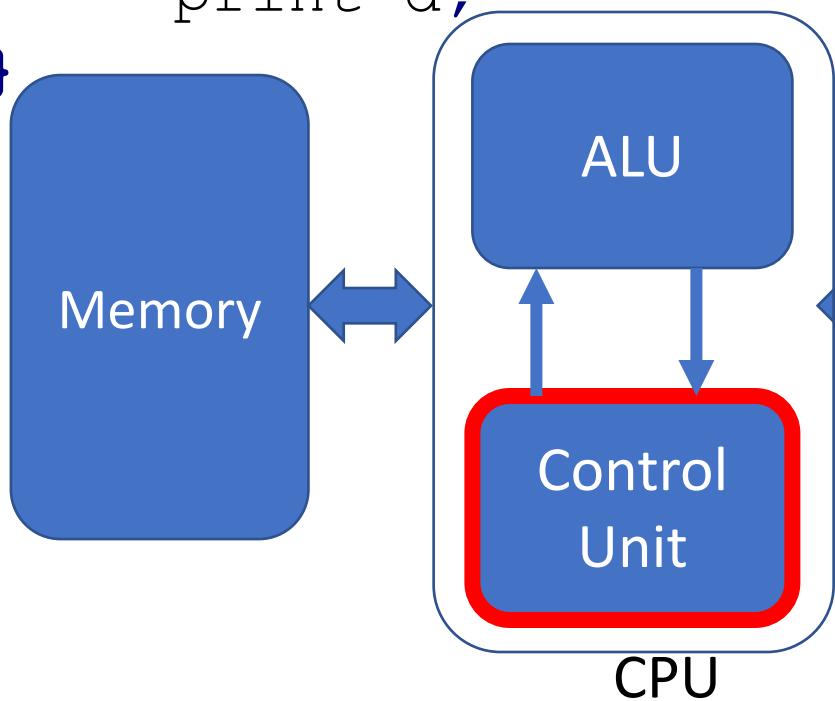


Program execution on von Neumann computer

```
main() {
```

```
    readIO a;  
    readIO b;  
    c = a + b;  
    store c;  
    d = a - b;  
    print d;
```

```
}
```



1. Control Unit understands that data needs to be provided by User.

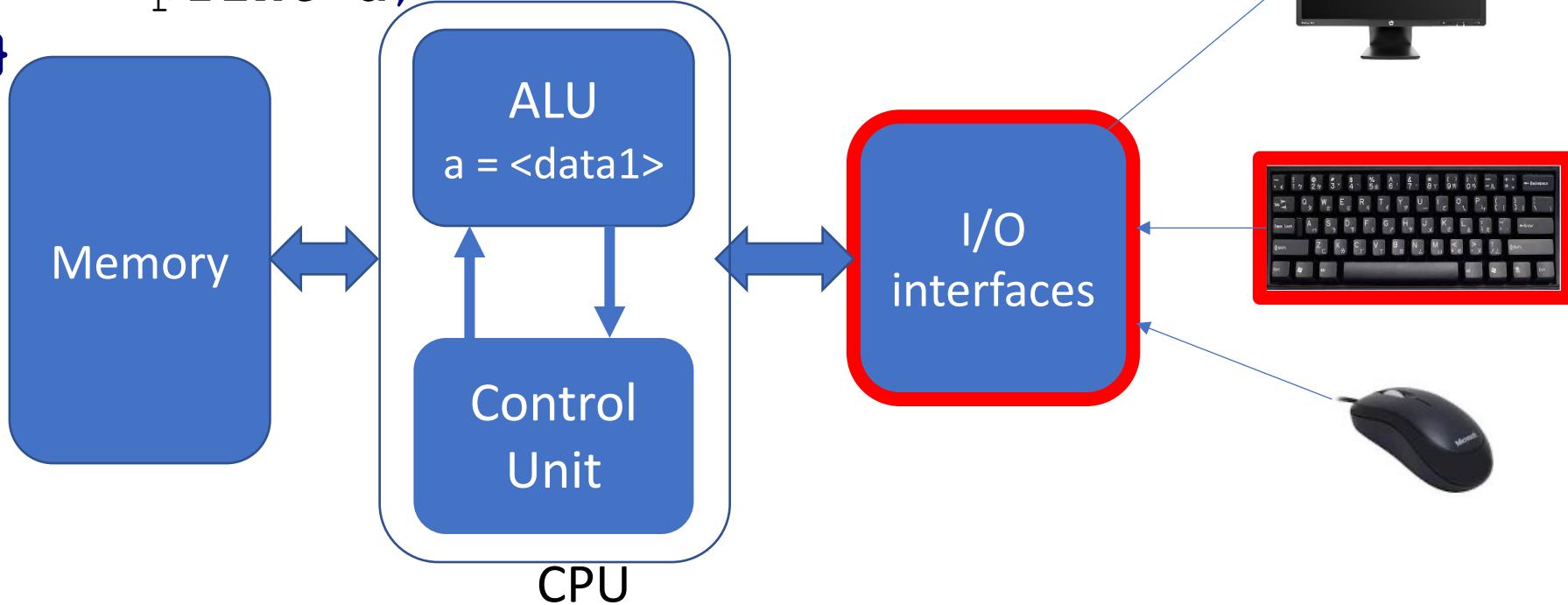
Program execution on von Neumann computer

```
main() {
```

```
    readIO a;  
    readIO b;  
    c = a + b;  
    store c;  
    d = a - b;  
    print d;
```

```
}
```

2. Data is read from input device (e.g. keyboard) and brought to the CPU

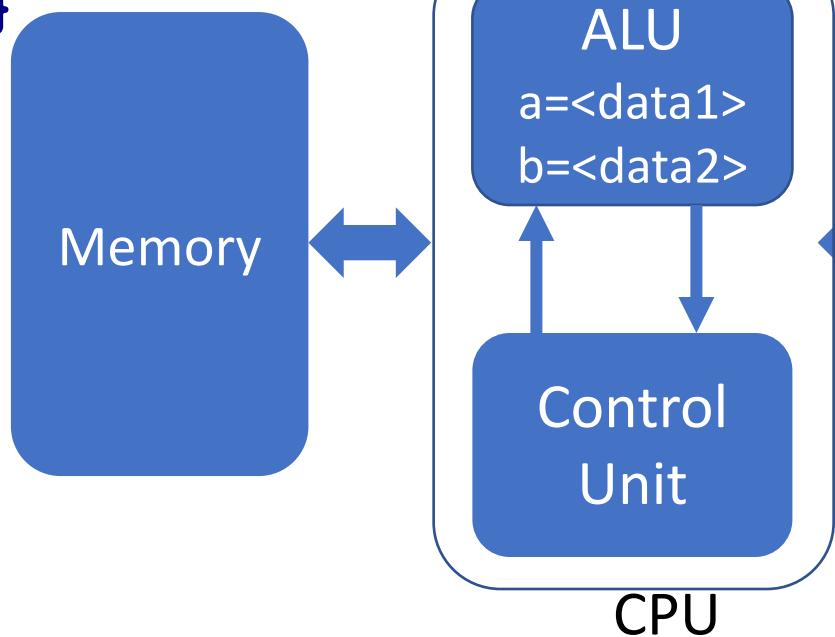


Program execution on von Neumann computer

```
main () {
```

```
    readIO a;  
    readIO b;  
    c = a + b;  
    store c;  
    d = a - b;  
    print d;
```

```
}
```



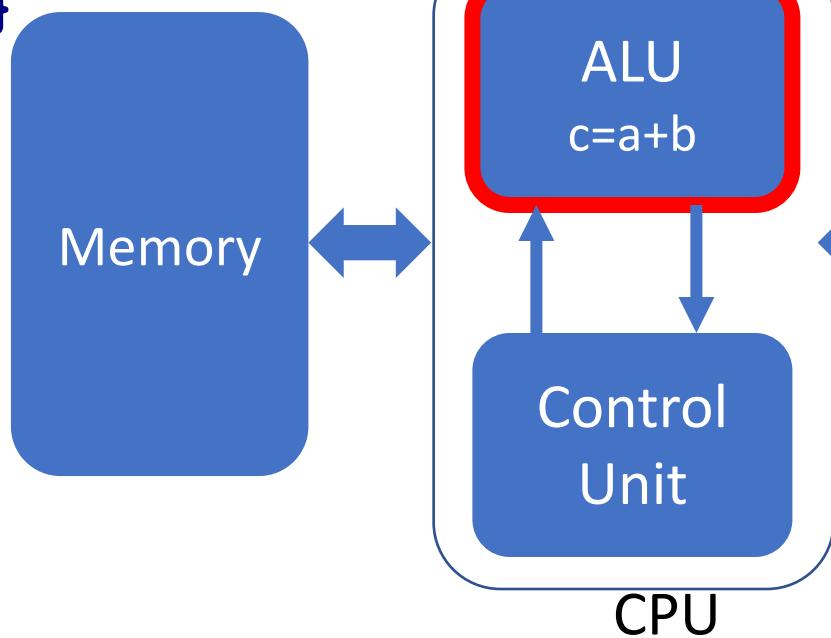
3-4. Similar steps are followed

Program execution on von Neumann computer

```
main() {
```

```
    readIO a;  
    readIO b;  
    c = a + b;  
    store c;  
    d = a - b;  
    print d;
```

```
}
```



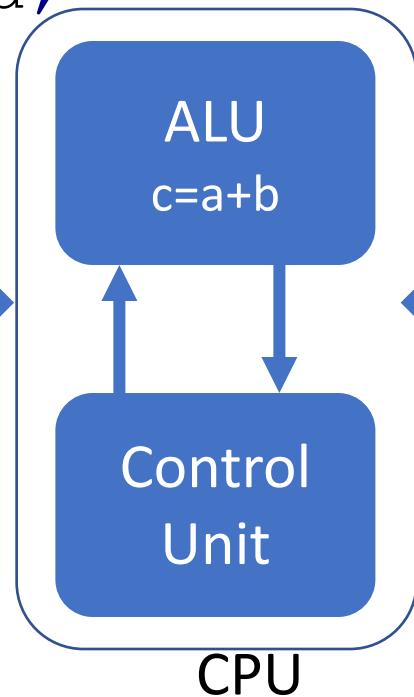
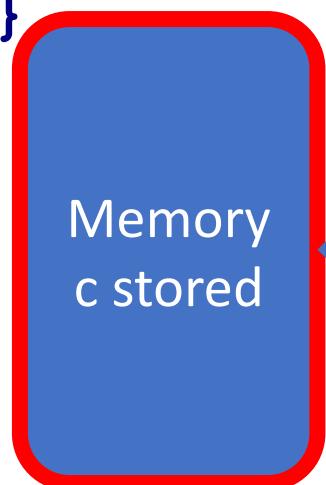
5. Controller commands ALU
to compute addition

Program execution on von Neumann computer

```
main() {
```

```
    readIO a;  
    readIO b;  
    c = a + b;  
    store c;  
    d = a - b;  
    print d;
```

```
}
```



6. Controller copies data from ALU to Memory

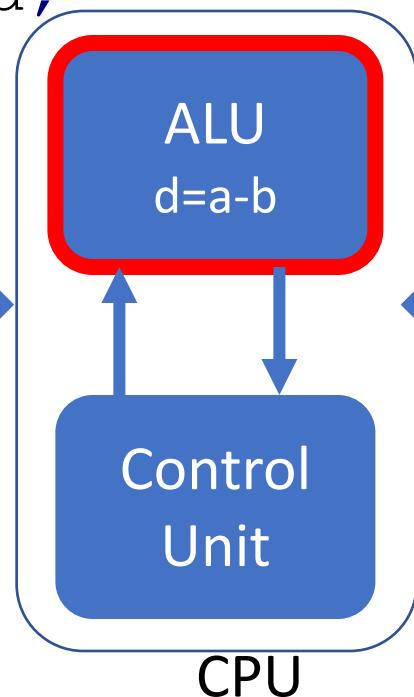
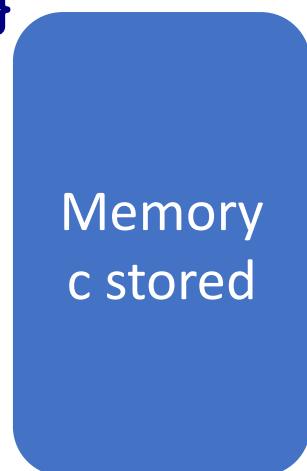


Program execution on von Neumann computer

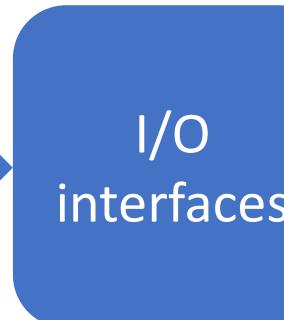
```
main() {
```

```
    readIO a;  
    readIO b;  
    c = a + b;  
    store c;  
    d = a - b;  
    print d;
```

```
}
```



7. Controller commands ALU to compute subtraction

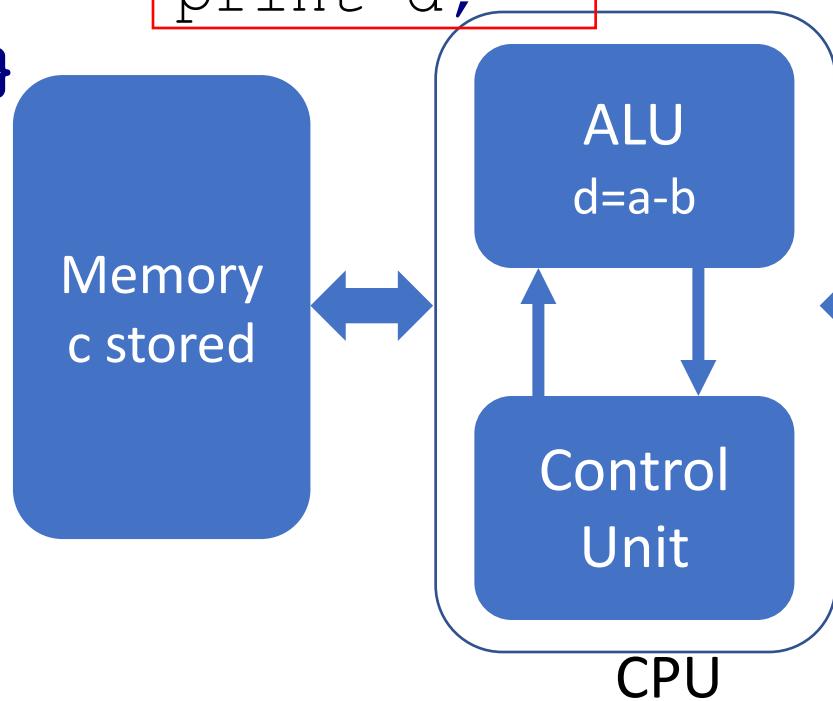


Program execution on von Neumann computer

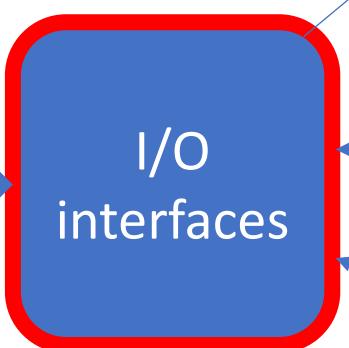
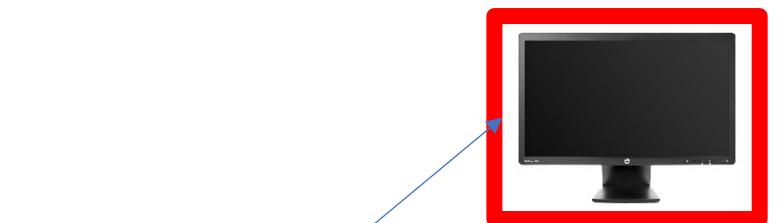
```
main() {
```

```
    readIO a;  
    readIO b;  
    c = a + b;  
    store c;  
    d = a - b;  
    print d;
```

```
}
```



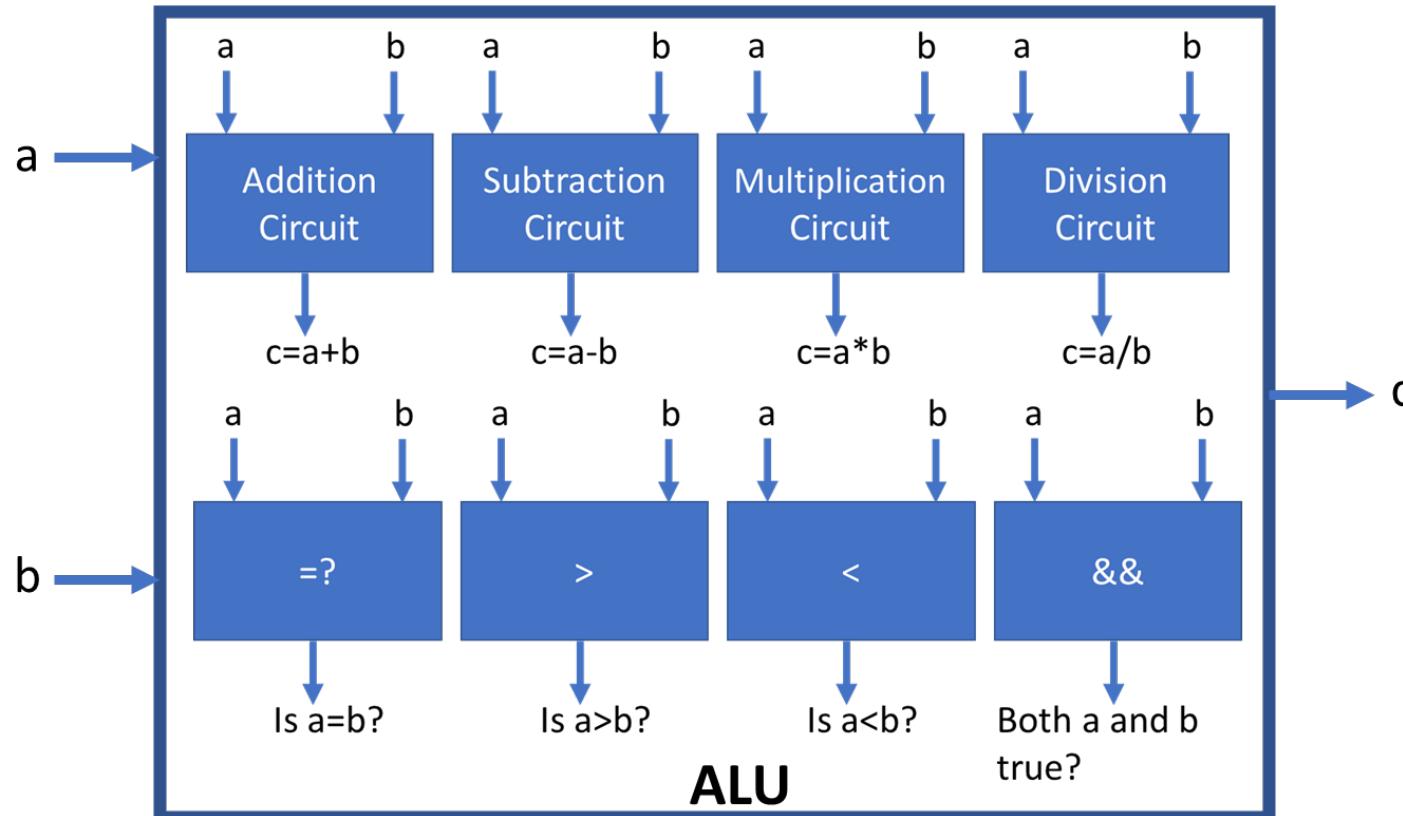
8. Controller sends data from ALU to display



Organization of a CPU

Inside a CPU: Arithmetic and Logic Unit

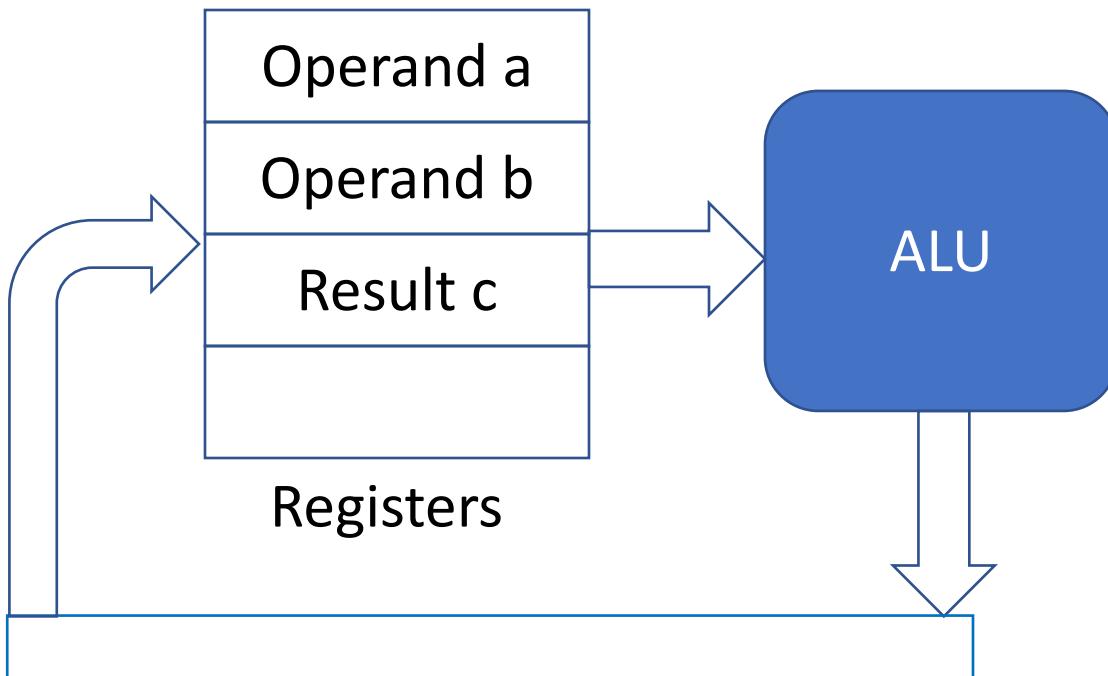
- **Arithmetic and Logic Unit (ALU)** is the union of the circuits for performing arithmetic and logical operations.



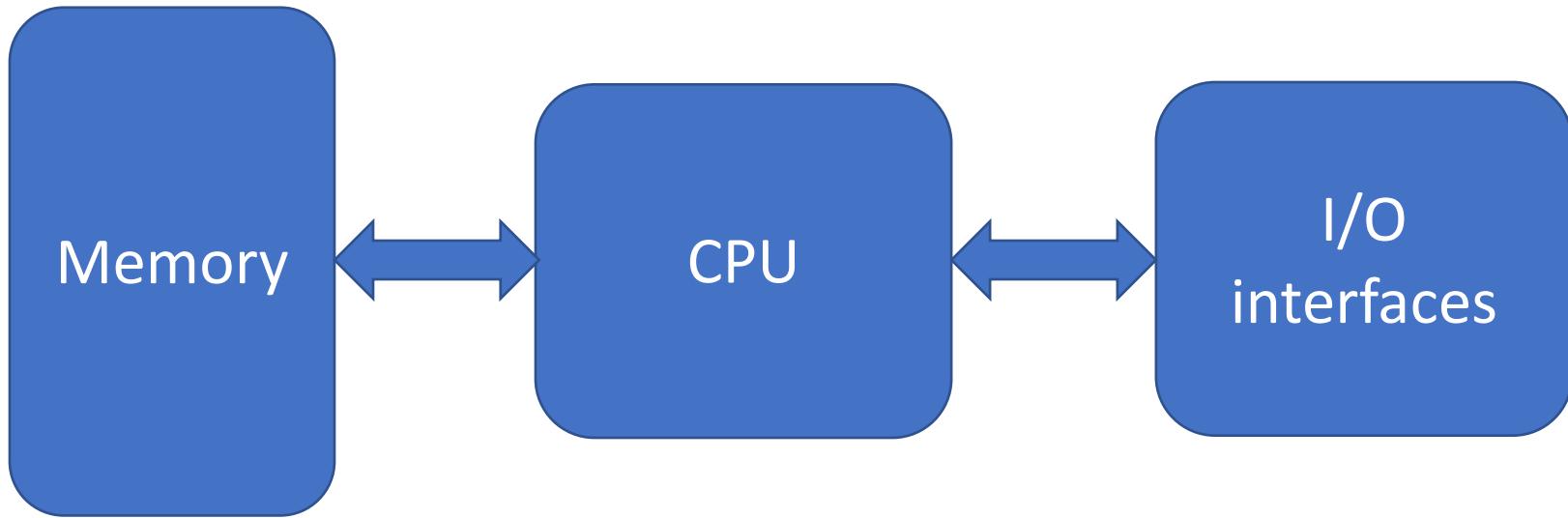
- **Control Unit** is responsible for step-by-step execution of instructions during a program execution.

Inside a CPU: Registers

- Registers are small storage elements that are located very close to the ALU.
- Registers are used as temporary storage during computation.
- ALU can read from and write to registers very fast.



What is the advantage of having registers inside CPU?



von Neumann Architecture has three main components

1. Central Processing Unit (CPU)
2. Memory
3. Input/Output interfaces.

CPU need not have Registers. We can use Memory to store all data.

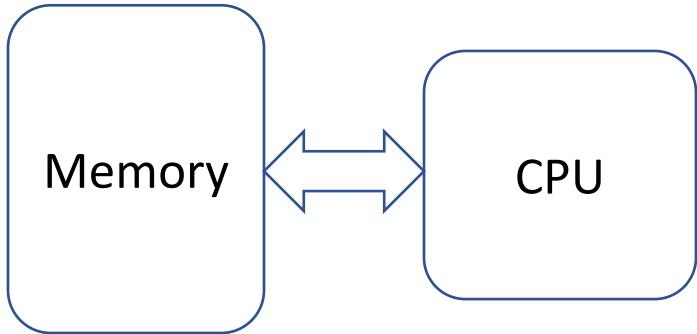
What is the advantage of having registers inside CPU?

Consider this computation: $c = a * b + a + b + (a - b) * a$

What is the advantage of having registers inside CPU?

Consider this computation: $c = a*b + a + b + (a-b)*a$

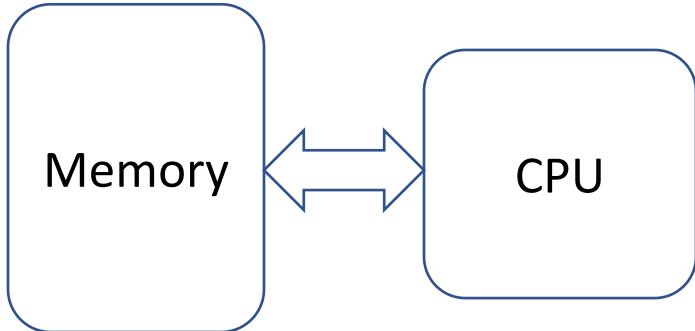
Case 1: without registers



What is the advantage of having registers inside CPU?

Consider this computation: $c = a*b + a + b + (a-b)*a$

Case 1: without registers



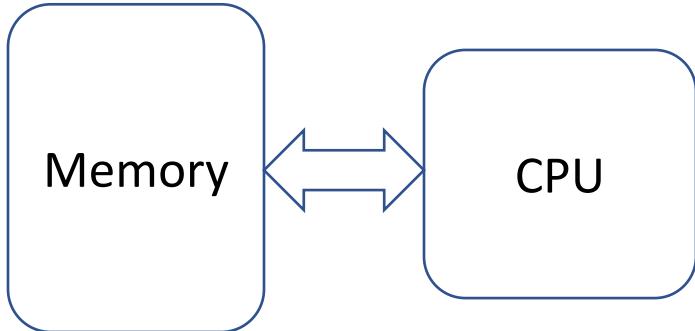
Computation steps

1. read {a,b} from memory

What is the advantage of having registers inside CPU?

Consider this computation: $c = a*b + a + b + (a-b)*a$

Case 1: without registers



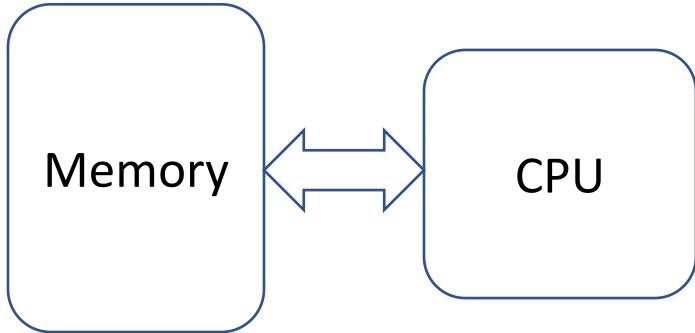
Computation steps

1. read {a,b} from memory
2. compute $c = a*b$

What is the advantage of having registers inside CPU?

Consider this computation: $c = a*b + a + b + (a-b)*a$

Case 1: without registers



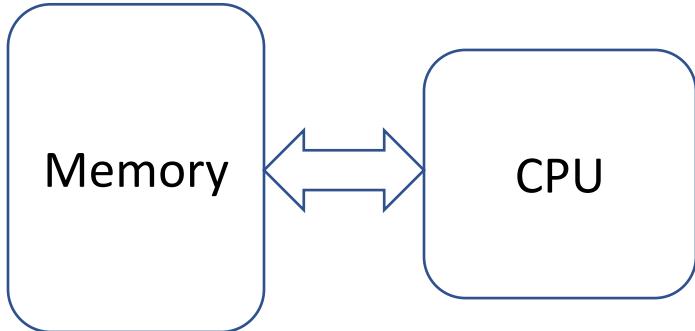
Computation steps

1. read {a,b} from memory
2. compute $c = a*b$
3. store c in memory

What is the advantage of having registers inside CPU?

Consider this computation: $c = a*b + a + b + (a-b)*a$

Case 1: without registers



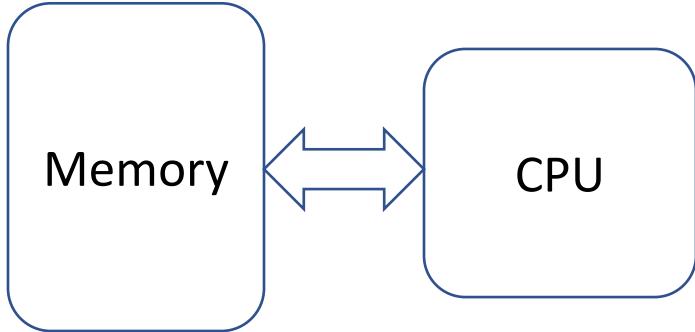
Computation steps

1. read {a,b} from memory
2. compute $c = a*b$
3. store c in memory
4. read {a,c} from memory
5. compute $c = c + a$
6. store c

What is the advantage of having registers inside CPU?

Consider this computation: $c = a*b + a + b + (a-b)*a$

Case 1: without registers



Performance of a computer is measured in 'time requirement' for a task. Assume that

- Each memory read/write takes 4 milliseconds
- Each arithmetic takes 1 millisecond.

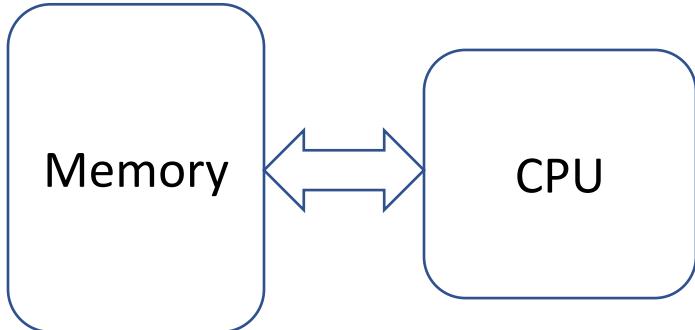
Computation steps (all)

1. read {a,b} from memory
2. compute $c = a*b$
3. store c in memory
4. read {a,c} from memory
5. compute $c = c + a$
6. store c
7. read {b,c} from memory
8. compute $c = c + b$
9. store c
10. read {a,b} from memory
11. compute $c' = a-b$
12. store c' in memory
13. read { c' ,a} from memory
14. compute $c' = c'*a$
15. store c' in memory
16. read {c, c' } from memory
17. compute $c = c+c'$
18. store c in memory

What is the advantage of having registers inside CPU?

Consider this computation: $c = a*b + a + b + (a-b)*a$

Case 1: without registers



In this particular computation,
No. memory read/write = 12
No. arithmetic = 6

Hence, total time = $12*4 + 6*1$
= 54 milliseconds

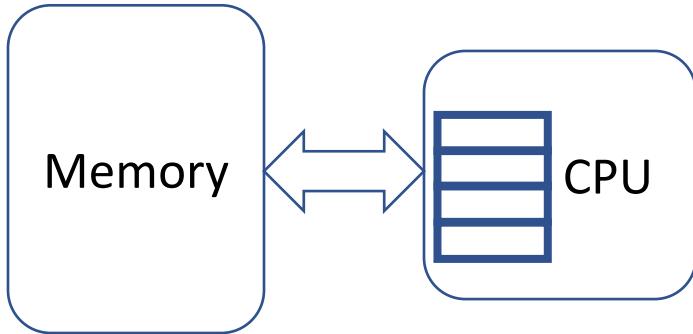
Computation steps (all)

1. read {a,b} from memory
2. compute $c = a*b$
3. store c in memory
4. read {a,c} from memory
5. compute $c = c + a$
6. store c
7. read {b,c} from memory
8. compute $c = c + b$
9. store c
10. read {a,b} from memory
11. compute $c' = a-b$
12. store c' in memory
13. read { c' ,a} from memory
14. compute $c' = c'*a$
15. store c' in memory
16. read {c, c' } from memory
17. compute $c = c+c'$
18. store c in memory

What is the advantage of having registers inside CPU?

Consider this computation: $c = a*b + a + b + (a-b)*a$

Case 2: with registers



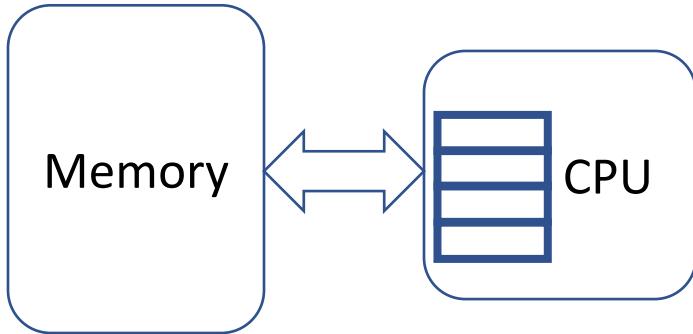
Two assumptions:

1. Initially, a and b are in Memory
2. ALU can read/write registers in 0 time overhead

What is the advantage of having registers inside CPU?

Consider this computation: $c = a*b + a + b + (a-b)*a$

Case 2: with registers



Two assumptions:

1. Initially, a and b are in Memory
2. ALU can read/write registers in 0 time overhead

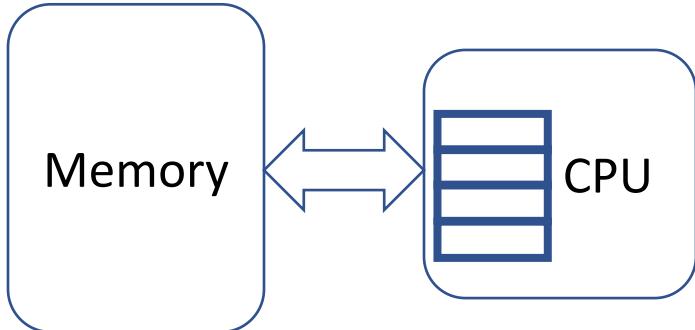
Idea:

We read $\{a, b\}$ from memory **only once** and then use the Registers to store all intermediate data.

What is the advantage of having registers inside CPU?

Consider this computation: $c = a*b + a + b + (a-b)*a$

Case 2: with registers

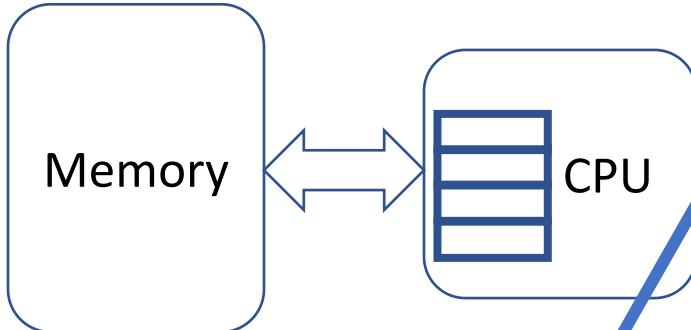


1. Read {a, b} from memory and load them in {Register1, Register2}
2. Read {Register1, Register2}, compute $c=a*b$ and store c in Register3
3. Read {Register3, Register1}, compute $c=c+a$ and store c in Register3
4. Read {Register3, Register2}, compute $c=c+b$ and store c in Register3
5. Read {Register1, Register2}, compute $c'=a-b$ and store c' in Register4
6. Read {Register1, Register4}, compute $c'=a*c'$ and store c' in Register4
7. Read {Register3, Register4}, compute $c=c+c'$ and store c in Register3
8. Finally, copy c from Register3 to Memory

What is the advantage of having registers inside CPU?

Consider this computation: $c = a*b + a + b + (a-b)*a$

Case 2: with registers



Memory

CPU

Pure arithmetic entirely
within CPU.

Register reads or writes have
zero-time overhead.

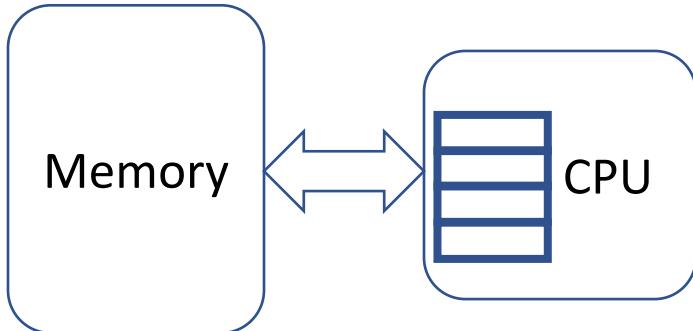
Only two memory read/write

1. Read {a, b} from memory and load them in {Register1, Register2}
2. Read {Register1, Register2}, compute $c=a*b$ and store c in Register3
3. Read {Register3, Register1}, compute $c=c+a$ and store c in Register3
4. Read {Register3, Register2}, compute $c=c+b$ and store c in Register3
5. Read {Register1, Register2}, compute $c'=a-b$ and store c' in Register4
6. Read {Register1, Register4}, compute $c'=a*c'$ and store c' in Register4
7. Read {Register3, Register4}, compute $c=c+c'$ and store c in Register3
8. Finally, copy c from Register3 to Memory

What is the advantage of having registers inside CPU?

Consider this computation: $c = a*b + a + b + (a-b)*a$

Case 2: with registers



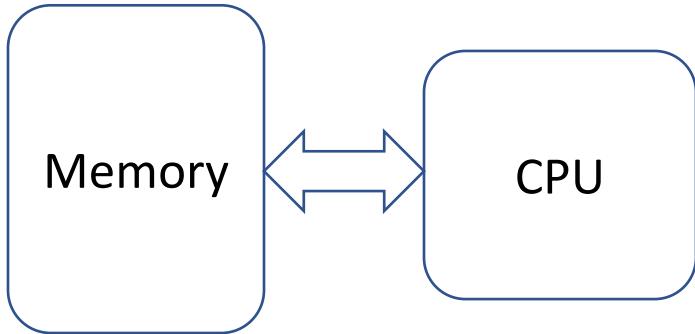
Total time requirement =
 $2*4 + 6*1 = 14$ milliseconds

1. Read {a, b} from memory and load them in {Register1, Register2}
2. Read {Register1, Register2}, compute $c=a*b$ and store c in Register3
3. Read {Register3, Register1}, compute $c=c+a$ and store c in Register3
4. Read {Register3, Register2}, compute $c=c+b$ and store c in Register3
5. Read {Register1, Register2}, compute $c'=a-b$ and store c' in Register4
6. Read {Register1, Register4}, compute $c'=a*c'$ and store c' in Register4
7. Read {Register3, Register4}, compute $c=c+c'$ and store c in Register3
8. Finally, copy c from Register3 to Memory

What is the advantage of having registers inside CPU?

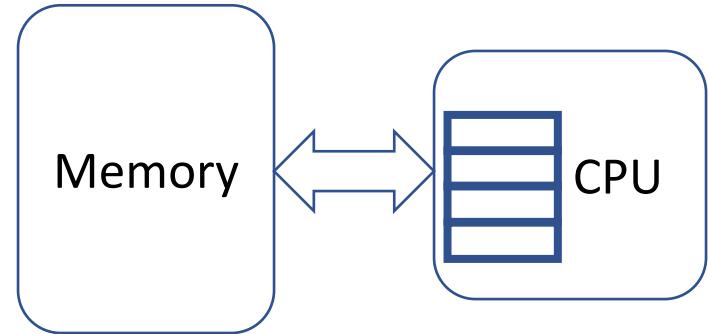
Consider this computation: $c = a*b + a + b + (a-b)*a$

Case 1: without registers



Total time requirement =
 $12*4 + 6*1 = 54$ milliseconds

Case 2: with registers



Total time requirement =
 $2*4 + 6*1 = 14$ milliseconds

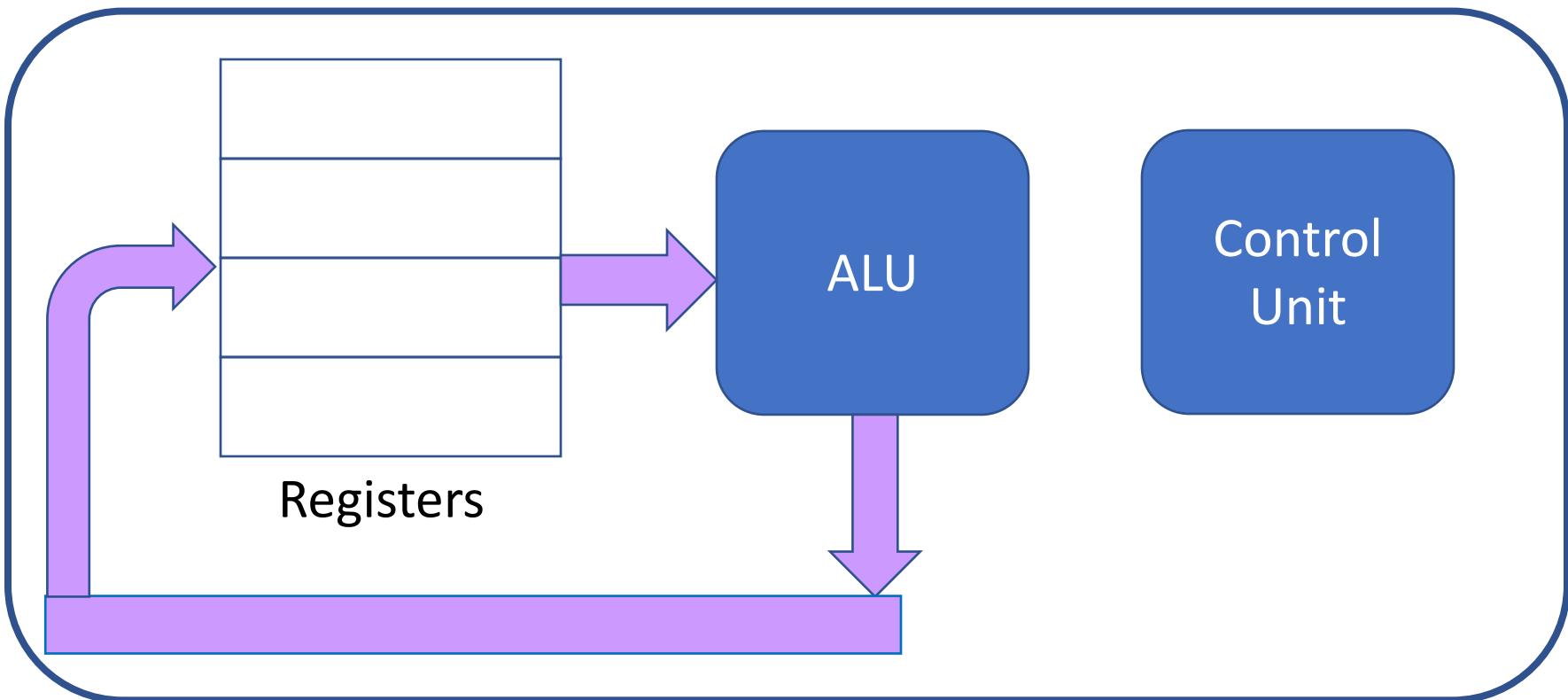
Conclusion: Registers improve processing speed

All present-day computers have Registers inside CPUs

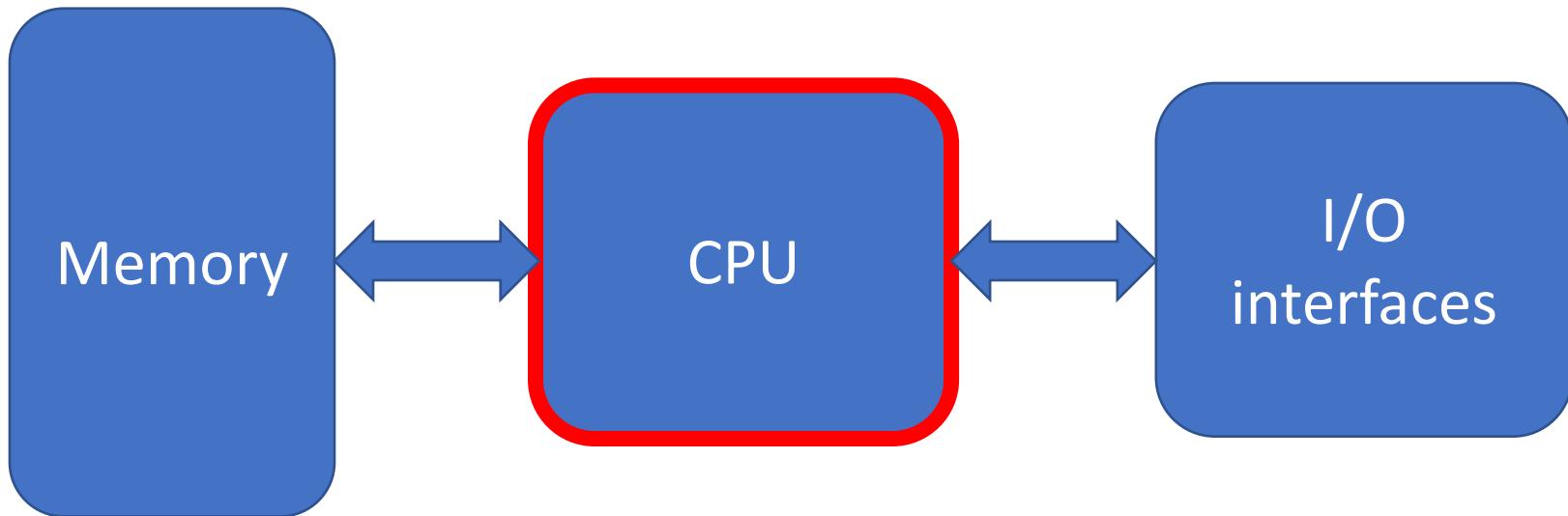
Updated: Inside a CPU

So far, we have the following components inside a CPU

- Arithmetic and Logic Unit (ALU)
- Registers (new component for improving performance)
- Control Unit



Recap: The von Neumann Architecture

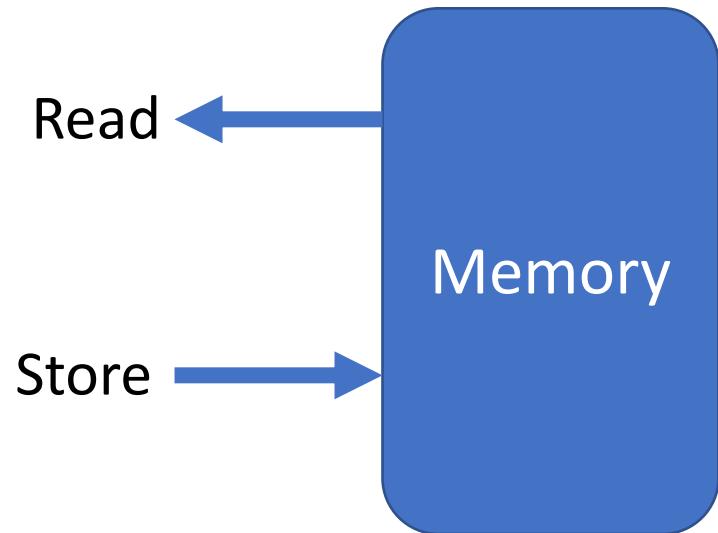


Consists of three main components

1. **Central Processing Unit (CPU)** (we have covered the CPU)
2. Memory (our next topic)
3. Input/Output interfaces.

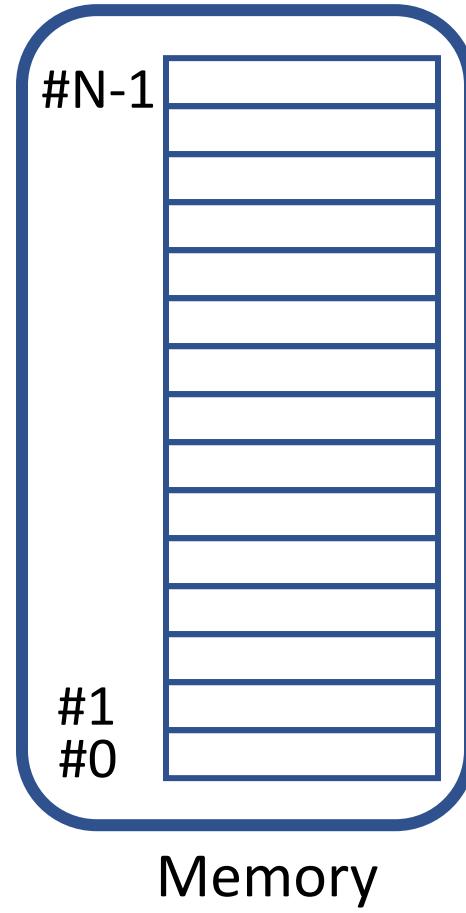
Organization of Memory

Memory



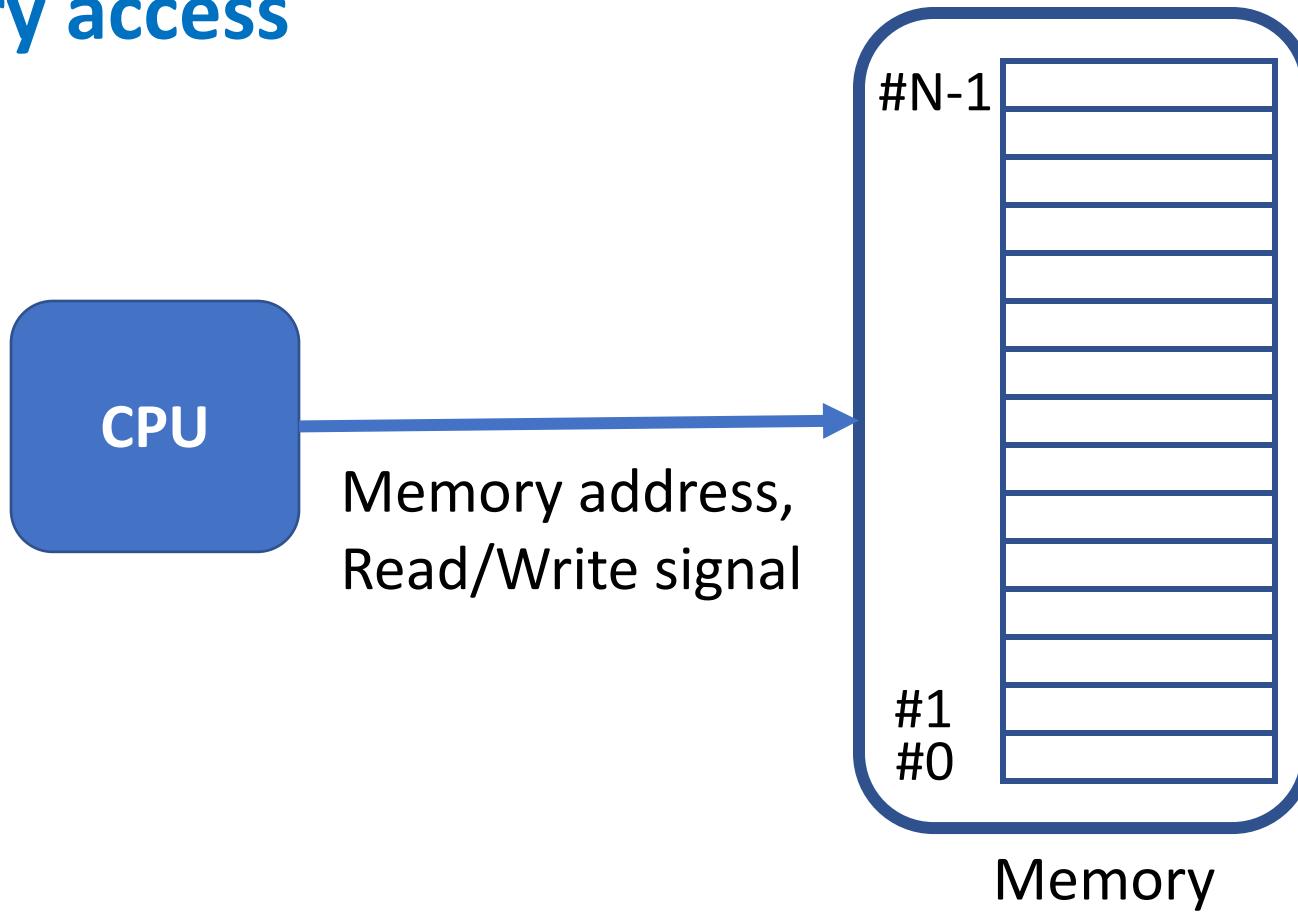
Programmer sees memory as a storage element.

Programmer's View: Memory as an addressable storage



- Memory consists of small 'cells'
- Each cell can store a small piece of data
- The cells have addresses. E.g. 0 to N-1

Memory access



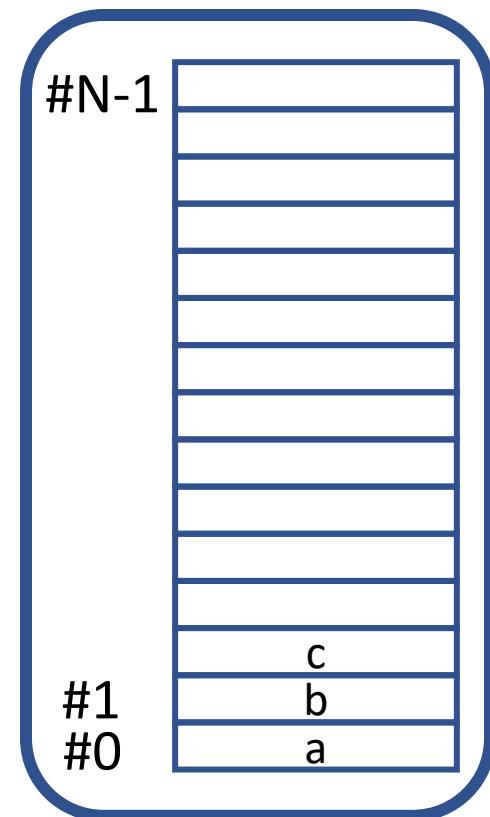
- During Read and Store operations, CPU generates memory addresses.
- Memory Management Unit (MMU) reads or writes from/to the requested memory-location.

Example: Memory access during program exe.

```
main () {  
    read a;  
    read b;  
    c = a + b;  
    store c;  
}
```

High-level code

Assume that programmer or compiler has allocated variables ‘a’, ‘b’ and ‘c’ in the memory locations with address #0, #1 and #2.

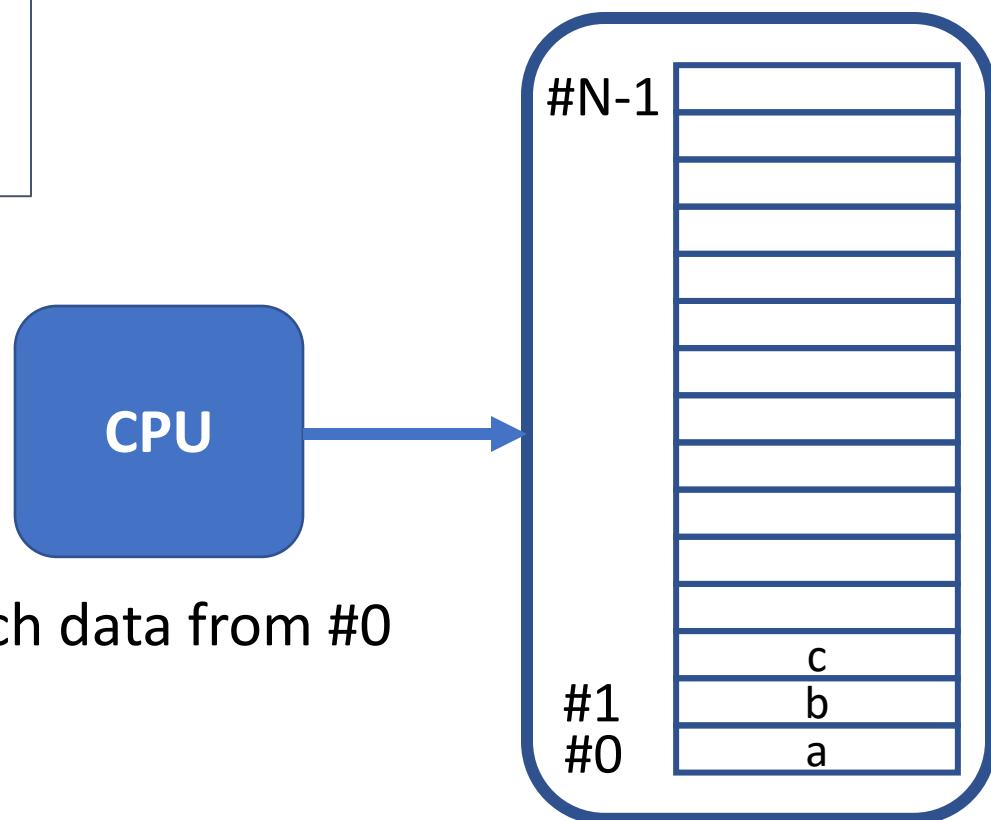


Example: Memory access during program exe.

```
main () {  
    read a;  
    read b;  
    c = a + b;  
    store c;  
}
```

High-level code

Assume that programmer or compiler has allocated variables 'a', 'b' and 'c' in the memory locations with address #0, #1 and #2.



Steps:

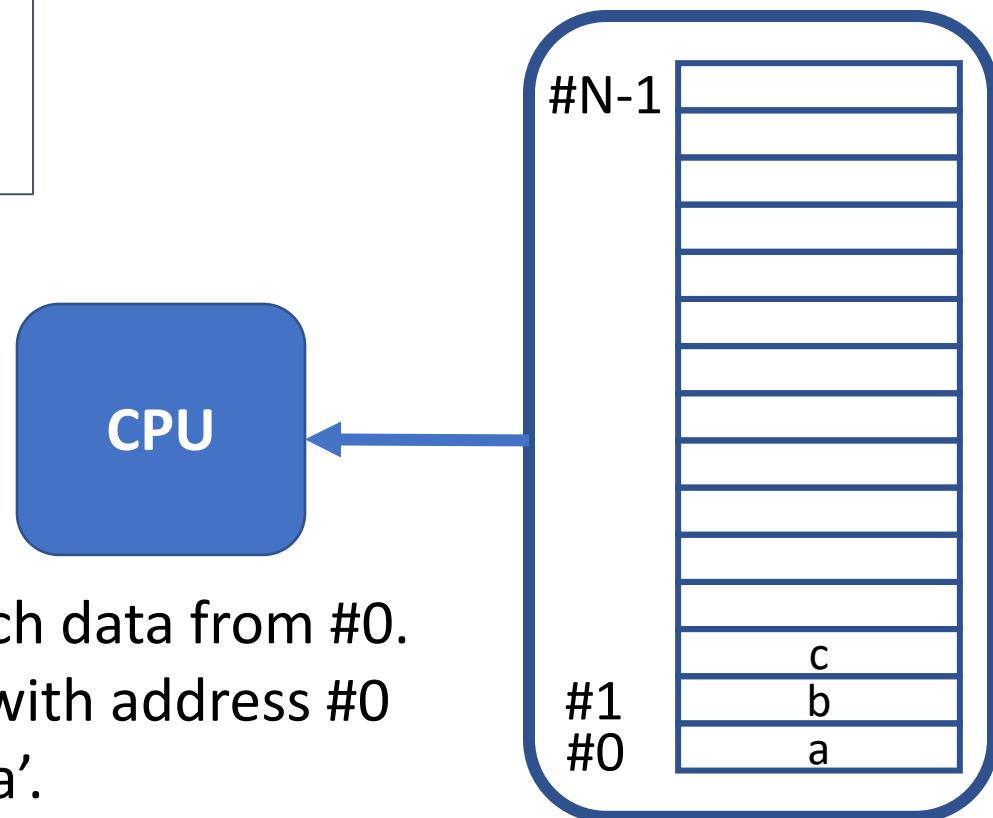
1. CPU asks MMU to fetch data from #0

Example: Memory access during program exe.

```
main () {  
    read a;  
    read b;  
    c = a + b;  
    store c;  
}
```

High-level code

Assume that programmer or compiler has allocated variables ‘a’, ‘b’ and ‘c’ in the memory locations with address #0, #1 and #2.



Steps:

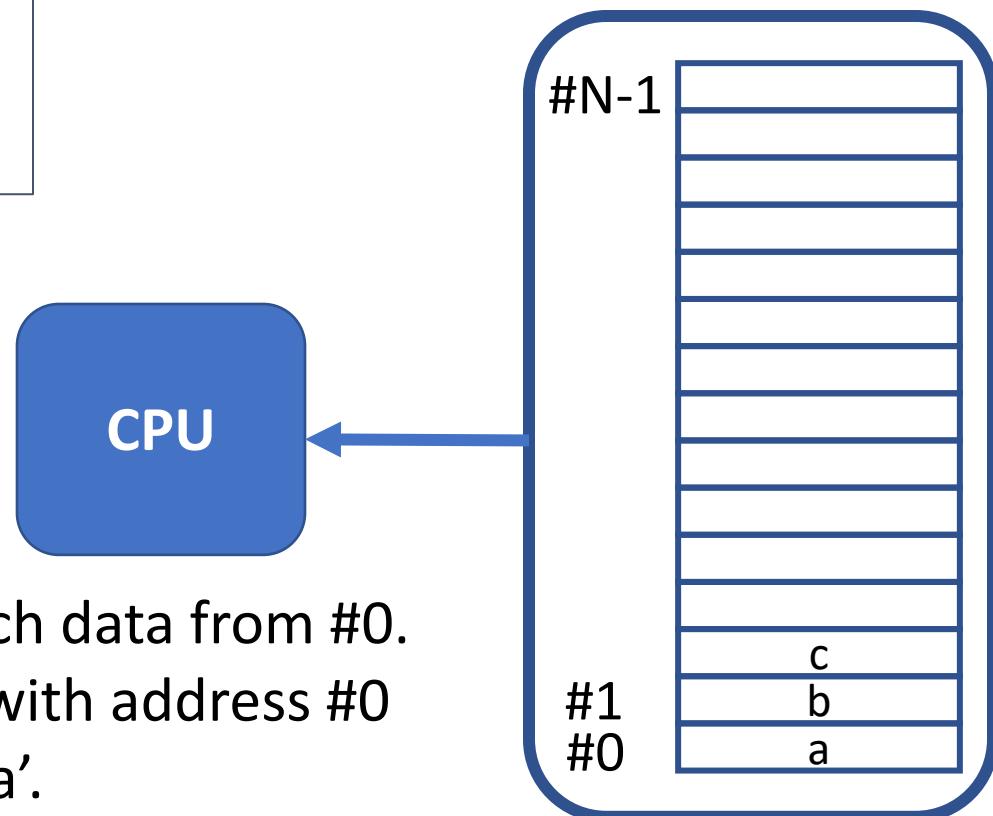
1. CPU asks MMU to fetch data from #0.
2. MMU reads location with address #0 and returns value of ‘a’.

Example: Memory access during program exe.

```
main () {  
    read a;  
    read b;  
    c = a + b;  
    store c;  
}
```

High-level code

Assume that programmer or compiler has allocated variables 'a', 'b' and 'c' in the memory locations with address #0, #1 and #2.



Steps:

1. CPU asks MMU to fetch data from #0.
2. MMU reads location with address #0 and returns value of 'a'.
3. Similar steps for 'read b'

Example: Memory access during program exe.

```
main () {  
    read a;  
    read b;  
    c = a + b;  
    store c;  
}
```

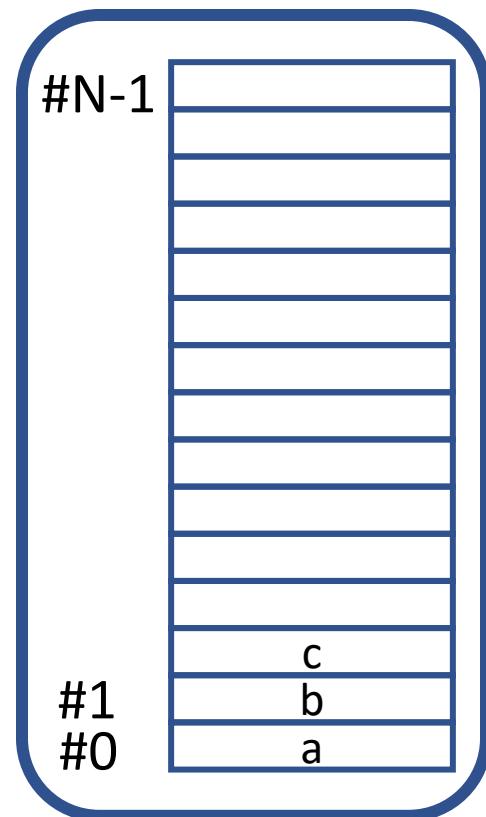
High-level code



Steps:

4. CPU computes sum 'c'

Assume that programmer or compiler has allocated variables 'a', 'b' and 'c' in the memory locations with address #0, #1 and #2.

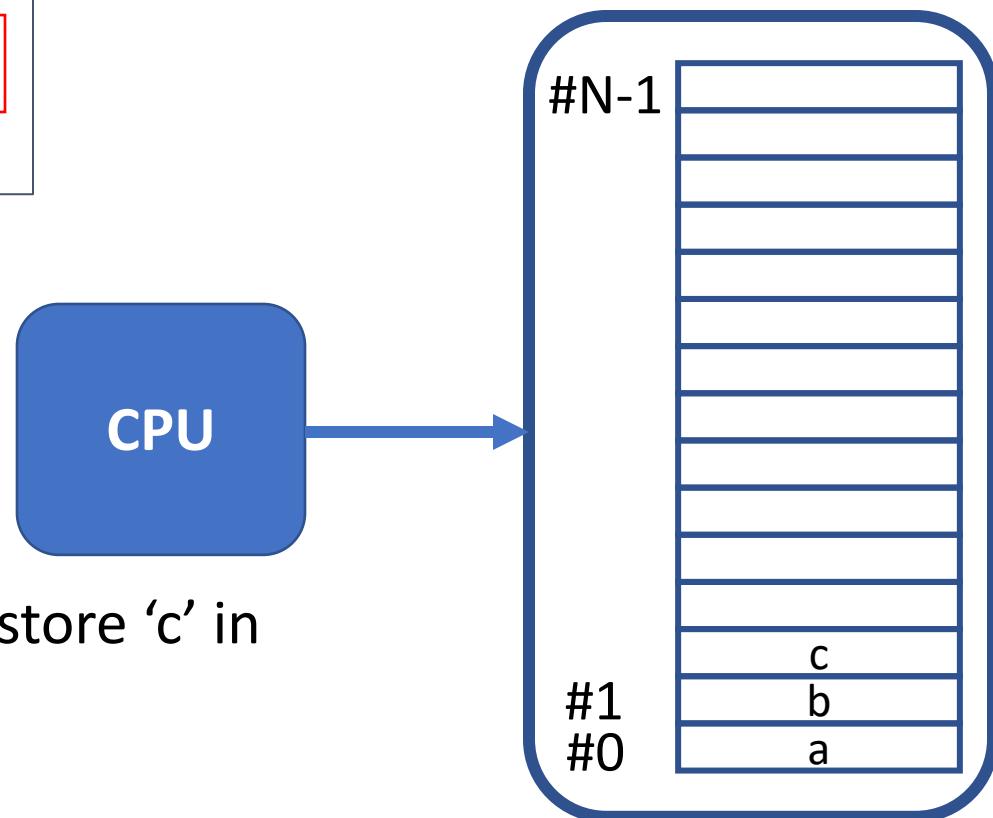


Example: Memory access during program exe.

```
main () {  
    read a;  
    read b;  
    c = a + b;  
    store c;  
}
```

High-level code

Assume that programmer or compiler has allocated variables 'a', 'b' and 'c' in the memory locations with address #0, #1 and #2.



Steps:

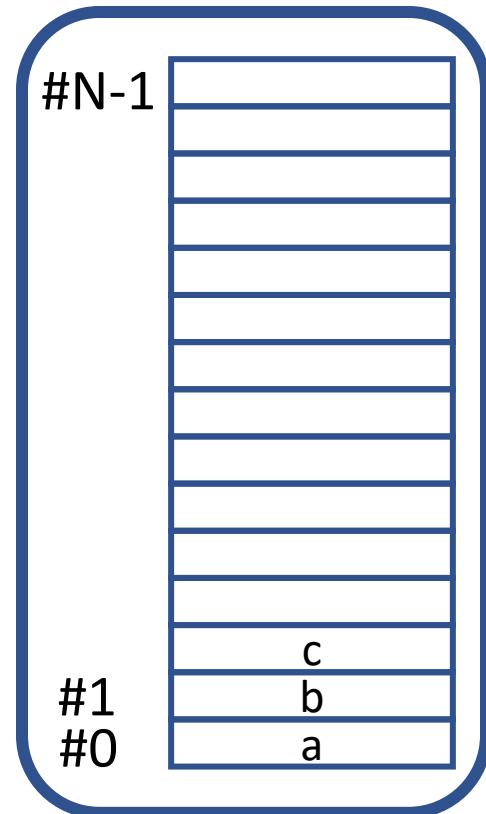
5. CPU instructs MMU to store 'c' in location with address #2.
6. MMU writes 'c' at #2

Example: Memory access during program exe.

```
main () {  
    read a;  
    read b;  
    c = a + b;  
    store c;  
}
```

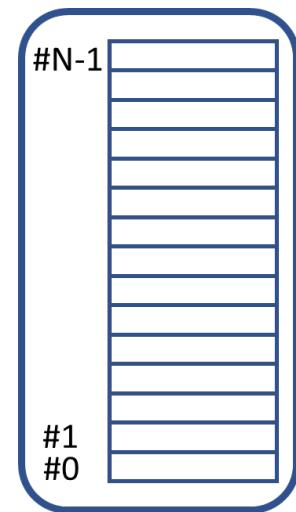
High-level code

In C programming, you will learn how to work with memory addresses using **Pointers**.



Conclusions

- We have studied von Neumann architecture.
- Programmer sees memory as a storage element.
 - Memory consists of small 'cells'
 - Each cell can store a small piece of data
 - The cells have addresses. E.g. 0 to N-1



General Structure of a C Program

Mohammed Bahja

School of Computer Science

University of Birmingham

General structure of a C program

```
#include <header file>
foo1() {    // User-defined function
            // Declaration of variables
            // Arithmetic and Logical expressions
}
main() {
        // Declaration of variables
        // Arithmetic and Logical expressions
        foo1(); // function call
        // More arithmetic and Logical expressions
}
```

- Program must contain a **main()** function
- Program execution starts from main()
- Header file contains pre-defined library functions

Built-in data types in C

C Basic Data Types	32-bit CPU		64-bit CPU	
	Size (bytes)	Range	Size (bytes)	Range
char	1	-128 to 127	1	-128 to 127
short	2	-32,768 to 32,767	2	-32,768 to 32,767
int	4	-2,147,483,648 to 2,147,483,647	4	-2,147,483,648 to 2,147,483,647
long	4	-2,147,483,648 to 2,147,483,647	8	-9,223,372,036,854,775,808-9,223,372,036,854,775,807
long long	8	-9,223,372,036,854,775,808-9,223,372,036,854,775,807	8	-9,223,372,036,854,775,808-9,223,372,036,854,775,807
float	4	3.4E +/- 38	4	3.4E +/- 38
double	8	1.7E +/- 308	8	1.7E +/- 308

Built-in data types in C

When we deal with **positive numbers only**, we can add **unsigned** before the type and get the range doubled.

```
int a;  
// Range is -2,147,483,648 to 2,147,483,647  
  
unsigned int b;  
// Range is 0 to 4,294,967,295
```

Immediate initialization of built-in variable

```
int i = 5, j =6;  
  
char c = 'A', d;  
  
float f = 2.33333333;  
  
unsigned int n = 4294967295;
```

All variables except ‘d’ have been initialized during declaration.

Un-initialized variable contains a ‘garbage’ value initially.

Constant data

- Constants can be declared as `const`

```
const float PI = 3.13;
```

- A constant is stored in the read-only segment of the memory
- Any effort to change a `const` variable will result in compilation error

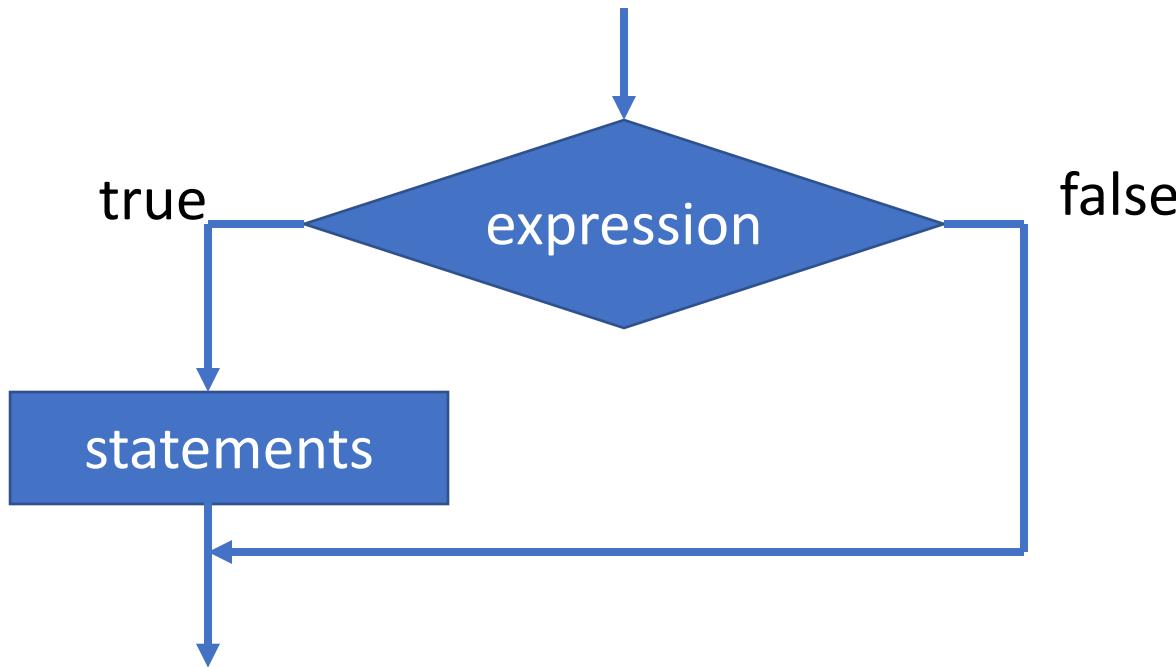
```
const float PI = 3.13;  
// ... Some code ...  
PI = PI + 1;
```

There will be compilation error.

if Statement

if statement is for branching

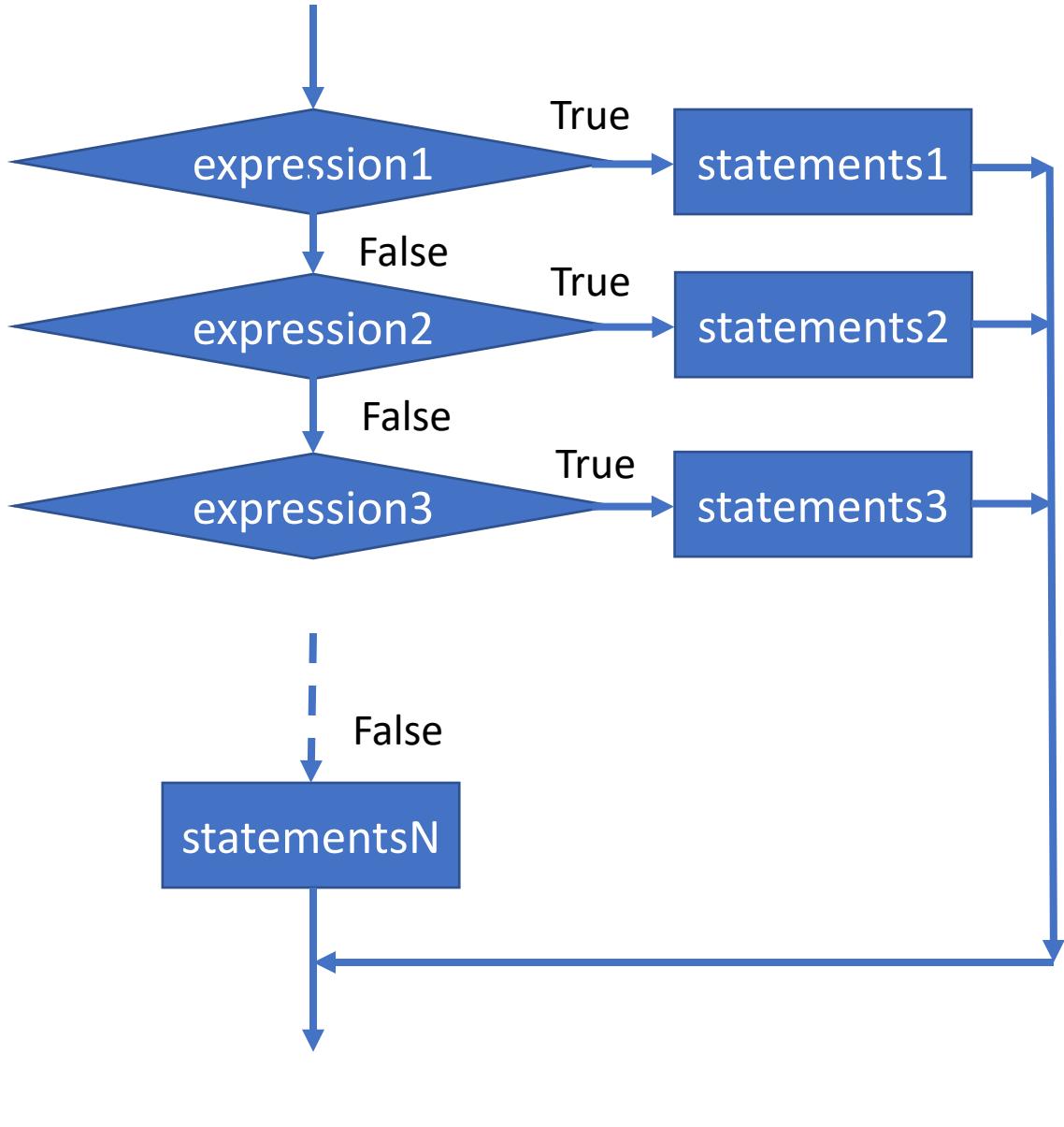
```
if (expression)  
    statements  
}
```



if-else Statement

Multiple branching

```
if (expression1)  
    statements1  
}  
else if (expression2)  
    statements2  
}  
else if (expression3)  
    statements3  
}  
...  
else {  
    statementsN  
}
```



if statement

```
if (value < 0) {  
    sign = '-';  
}
```

Used for conditional computation

if-else statements

```
if (testscore >= 90) {  
    grade = 'A';  
}  
else if (testscore >= 80) {  
    grade = 'B';  
}  
else if (testscore >= 70) {  
    grade = 'C';  
}  
else if (testscore >= 60) {  
    grade = 'D';  
}  
else {  
    grade = 'F';  
}
```

Used for multiple branching

switch Statement

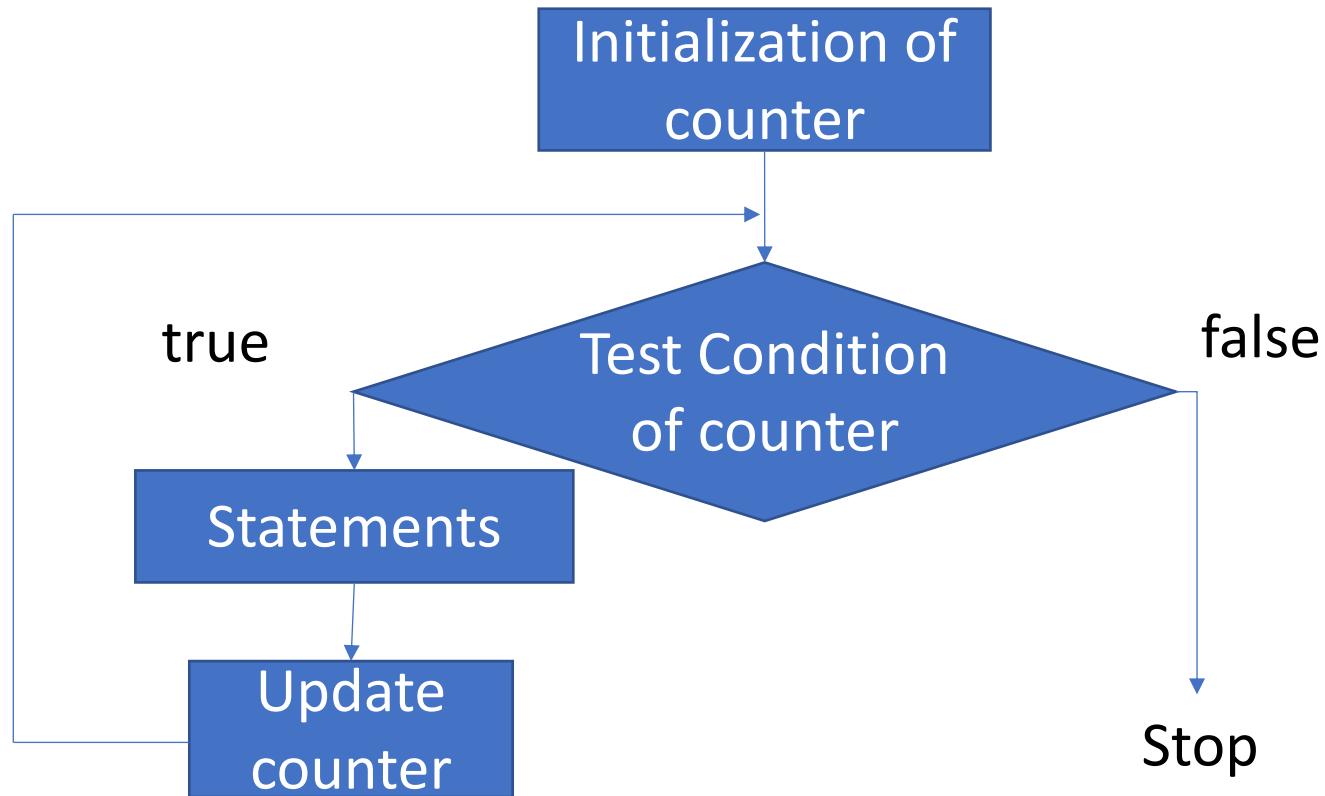
```
switch(expression) {  
    case constant-expression1:  
        Code Block1;  
        break;  
    case constant-expression1:  
        Code Block2;  
        break;  
    // ... More case blocks  
    case constant-expressionN:  
        Code BlockN;  
        break;  
    default:  
        Code BlockDefault;  
        break;  
}
```

- *expression* is first evaluated
- It is compared with *constant-expression1*, *constant-expression2*, ...
- If it matches anyone, then all statements inside that case are executed
- Statements in the **default** case are executed if no match is found

for loop

Syntax is

```
for(init; condition test; increment or decrement)  
{  
    //Statements to be executed in loop  
}
```



for loop

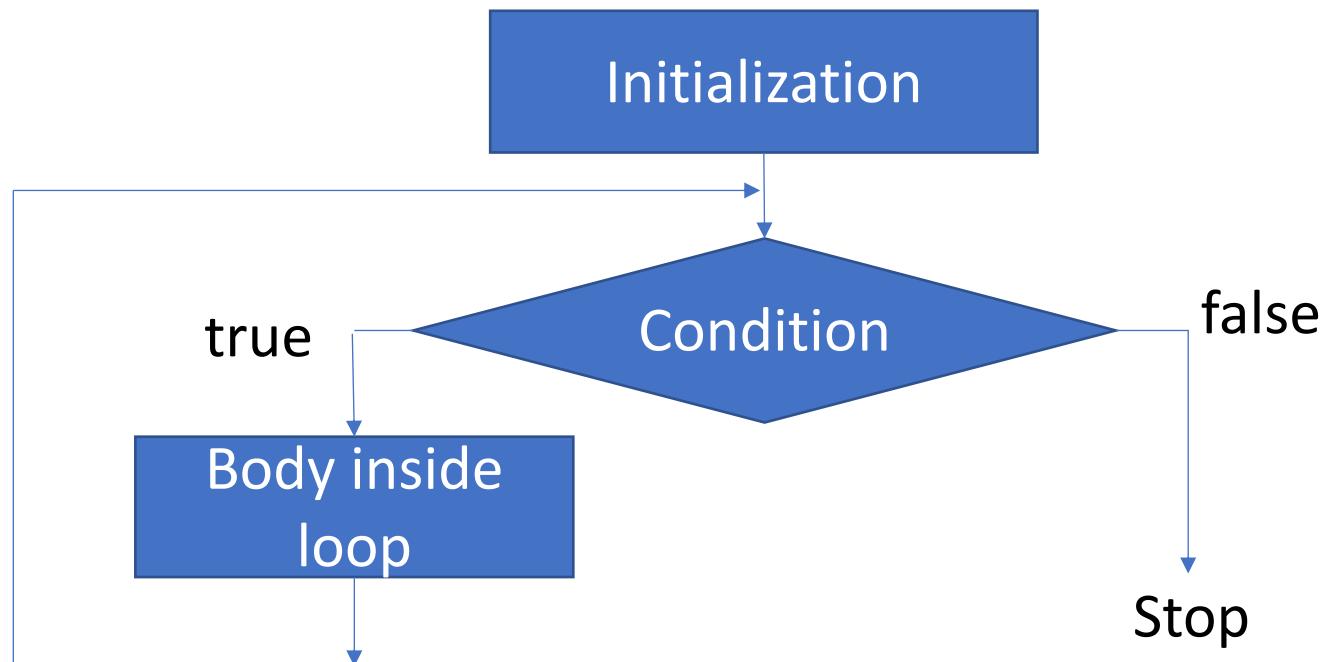
Example: for-loop for computing the sum of N natural numbers.

```
int sum=0, i;  
  
for(i=1; i<=N; i++) {  
    sum = sum + i;  
}
```

while loop

Syntax is

```
while (condition test)
{
    // Statements to be executed in loop
    // Update 'condition'
}
```



while loop

Example: while-loop for computing the sum of N natural numbers.

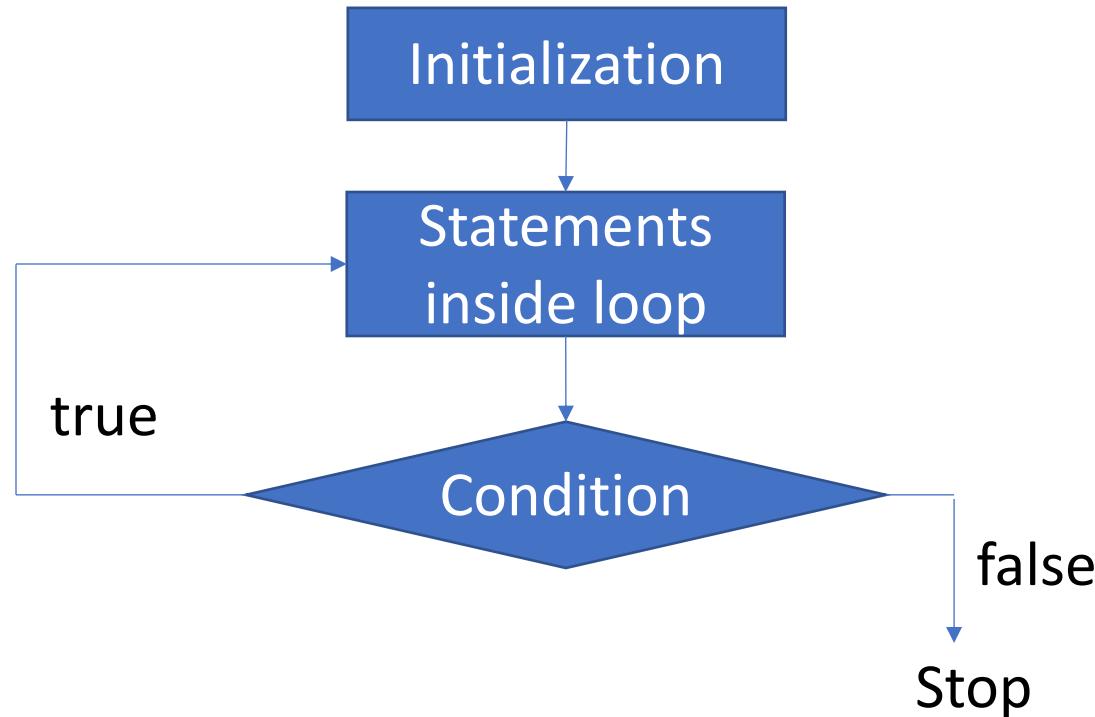
```
int sum=0, i=1;

while(i<=N) {
    sum = sum + i;
    i++;
}
```

do-while loop

Syntax is

```
do
{
    // Statements to be executed in loop
    // Update 'condition'
}while(condition test);
```



do-while loop

Example: do-while-loop for computing the sum of N natural numbers.

```
int sum=0, i=1;  
  
do{  
    sum = sum + i;  
    i++;  
}while(i<N);
```

continue statement

- **continue** is used inside a loop
- When a **continue** statement is encountered inside a loop, control skips the statements inside the loop for the current iteration
- and jumps to the beginning of the next iteration

```
sum = 0;  
for(i=1; i<=5; i++) {  
    if(i==3)  
        continue;  
    sum = sum + i;  
}
```

Computes $1 + 2 + 4 + 5$ (addition of 3 is skipped)

break statement

- **break** is used to come out of the loop instantly.
- After **break** control directly comes out of loop and the loop gets terminated.

```
sum = 0;  
for(i=1; i<=5; i++) {  
    if(i==3)  
        break;  
    sum = sum + i;  
}
```

Computes $1 + 2$

Loop terminates at $i=3$

Arrays in C

Array is a data structure

1. stores a fixed-size
2. sequential collection of elements
3. of the same type.

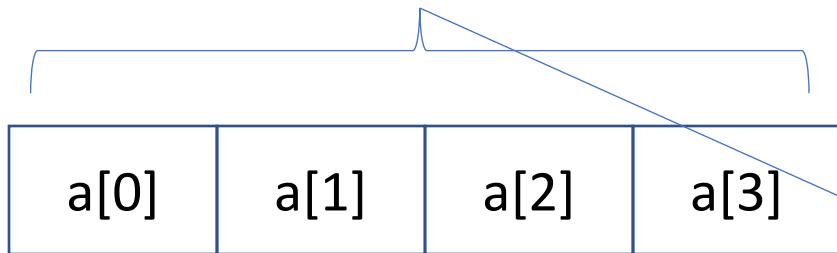
```
int    a[4] = {2, 5, 3, 7};  
float  b[3] = {2.34, 11.2, 0.12};  
char   c[8] = {'h','e','l','l','o'};
```

- Array `a[]` is of length 4 and contains only `int`
- Index of an array starts from 0, and then increments by 1
 - `a[0]` is the first element in `a[]` and it is 2
 - `a[3]` is the last element in `a[]` and it is 7

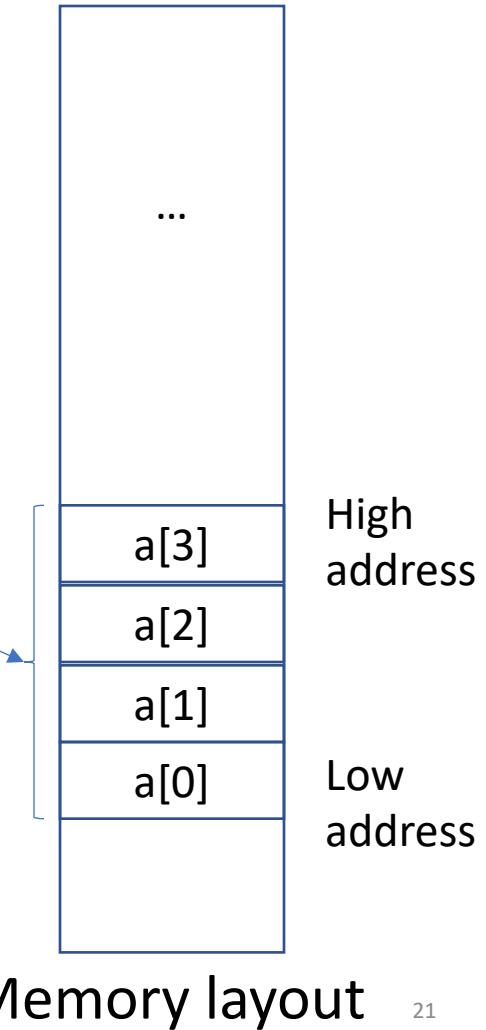
Memory layout of `a []`

Array is a data structure

1. stores a fixed-size
2. **sequential collection** of elements
3. of the same type.



Logical view of array `a[4]`



Array and Loop

Example of computing the sum of an array

```
// Compute sum of an int array
int a[4] = {2, 5, 3, 7};
int sum=0;

for(i=0; i<4; i++)
    sum = sum + a[i];
```

Arrays in C: common mistakes(1)

Array is a data structure

1. stores a **fixed-size**
2. sequential collection of elements
3. of the same type.

```
int array_size; //user input  
...  
int a[array_size];
```

This code causes malfunction.

Array size must be a known constant.

Example: `int array_size = 4;`

Arrays in C: common mistakes(2)

- C compiler does not check array limits
- If memory protection is violated, then the program crashes due to segmentation fault

```
// Compute sum of an int array
int a[4] = {2, 5, 3, 7};
int sum=0;

for(i=0; i<500; i++) // Beyond array limit
    sum = sum + a[i];
```



Program crashes

Two-Dimensional Array

```
// Declaration of array  
// with 3 rows and 4 columns  
int a[3][4];
```

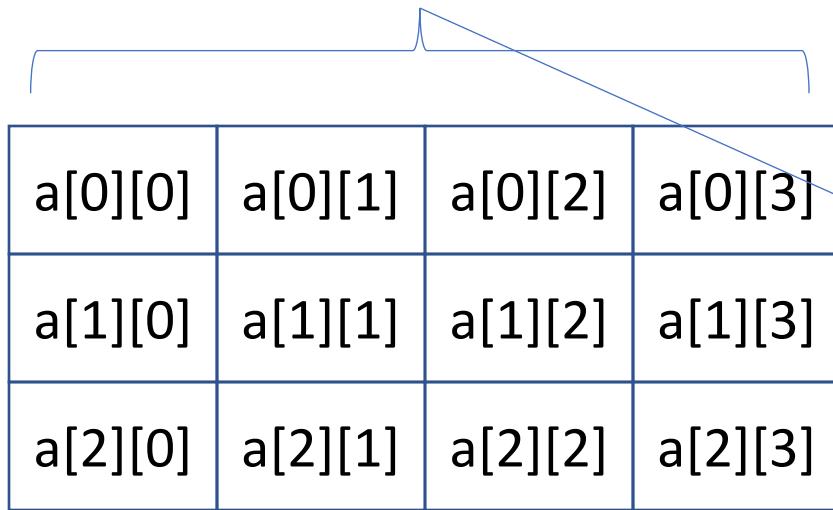
Logical view of a [3] [4]

a[0][0]	a[0][1]	a[0][2]	a[0][3]
a[1][0]	a[1][1]	a[1][2]	a[1][3]
a[2][0]	a[2][1]	a[2][2]	a[2][3]

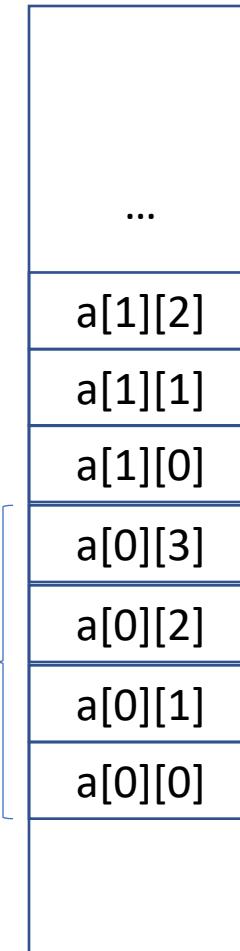
Memory layout of two-dimensional array

C compiler stores 2D array in **row-major** order

- All elements of Row #0 are stored
- then all elements of Row #1 are stored
- and so on



Logical view of array a[3][4]



Memory layout

Functions in C

- A function is a block of statements that together perform a task.
- Function definition in C has
 - a name,
 - a list of arguments (optional)
 - type of the value it returns (if any),
 - local variable declarations (if any), and
 - a sequence of statements (if any)

```
return_type function_name (argument list)
{
    // Local variables
    // Statements
}
```

Example of function call

```
// max() function computes the maximum of two
// input integers and returns the maximum
int max(int num1, int num2) {
    int temp;
    if(num1>num2)
        temp = num1;
    else
        temp = num2;
    return temp;
}

int main(){
    int a=5, b=11, c;
    c = max(a, b); // c gets the maximum of a and b
    // [some code here]
    return 0;
}
```

scanf() function

scanf() function can be used to receive inputs from keyboard.

It is defined in `stdio.h` library and prototype is:

```
scanf ("%format", &variable_name);
```

`format` is a place holder which depends on data-type.

Examples

```
scanf ("%d", &var1); // var1 is an int  
scanf ("%c", &var2); // var2 is a char  
scanf ("%f", &var3); // var3 is a float
```

```
// To read var1, var2 and var3 together:  
scanf ("%d%c%f", &var1, &var2, &var3);
```

Data types and formats

Data Type	Format
signed char	%c
unsigned char	%c
short signed int	%d
short unsigned int	%u
signed int	%d
unsigned int	%u
long signed int	%ld
long unsigned int	%lu
float	%f
double	%lf
long double	%Lf

printf() function

printf() function can be used to print outputs.

It is defined in `stdio.h` library and prototype is:

```
printf ("%format", variable_name);
```

format is a place holder which depends on data-type.

Examples

```
printf ("%d", var1); // var1 is an int
```

```
printf ("The value is = %d", var1);
```

// To print multiple variables together:

```
printf ("%d %c %f", var1, var2, var3);
```

Factorial program in Java

```
class Factorial {  
    static int factorial(int n){  
        if (n == 0)  
            return 1;  
        else  
            return(n * factorial(n-1));  
    }  
    public static void main(String args[]){  
        int i,fact=1;  
        int number=4;  
        fact = factorial(number);  
        System.out.println("Factorial is: "+fact);  
    }  
}
```

We will port it to C

Demo: Factorial program in C

Strings

- A string of characters is a 1D array of characters
- ASCII code (1 byte) of each character element is stored in consecutive memory locations
- String is terminated by the null character '\0' (ASCII value 0).
- The null string (length zero) is the null character only

```
#include <stdio.h>
int main()
{
    char name[] = "Comp Sc";
    printf("%s\n", name);
}
```



Length is 7, but the array is [0 ... 7]

Reading and printing strings

- A string can be read using `scanf()`
 - Format conversion specifier for a sequence of non-white-space characters is `%s`.

E.g. `scanf("%s", a)` will store only “Comp” for user input
“Comp Sc”

➤ And `%[^\\n]` for strings with white-space included

E.g. `scanf("%[^\\n]", a)` will store only “Comp Sc” for user input
“Comp Sc”

- A string can be printed using `printf()` with `%s`

Library `string.h`

- Library functions are available to manipulate strings.
- A list of commonly used string manipulation functions:
 - `strcpy()` - copy a string
 - `strcat()` - concatenate two strings
 - `strlen()` - get string length
 - `strcmp()` - compare two strings

strcpy - copy a string

```
#include <stdio.h>
#include <string.h>
int main () {
    char src[40];
    char dest[100];

    printf("Enter source string:");
    scanf("%[^\\n]", src);
    strcpy(dest, src);

    printf("Destination string : %s\\n", dest);

    return(0);
}
```

```
Enter source string:How are you?
Destination string : How are you?
```

strcat - concatenate two strings

- `strcat` joins two strings together.

strcat(string1, string2)

- `string2` is appended to `string1`
- `string2` remains unchanged

```
#include <stdio.h>
#include <string.h>
int main () {
    char str[100] = "Hello ";
    strcat(str, "World!");

    printf("New string : %s\n", str);

    return(0);
}
```

New string : Hello World!

strlen – length of a string

- Returns the number of characters in the input string
 $n = \text{strlen}(\text{string})$

```
#include <stdio.h>
#include <string.h>
int main () {
    char str[100];
    int n;

    printf("Enter source string:");
    scanf("%[^\\n]", str);
    n = strlen(str);
    printf("Length is: %d\\n", n);

    return(0);
}
```

```
Enter source string:Hello World
Length is: 11
```

strcmp – compares two strings

$n = \text{strcmp}(\text{string1}, \text{string2})$

- If the strings are equal then returns 0
- Otherwise, returns the numeric difference between the first non matching characters

```
int main () {
    char str1[100] = "Hello World!";
    char str2[100] = "Hello World!";
    char str3[100] = "Hello Wosld!";

    int n;

    n = strcmp(str1, str2);
    printf("Difference is: %d\n", n);

    n = strcmp(str1, str3);
    printf("Difference is: %d\n", n);

    return(0);
}
```

```
Difference is: 0
Difference is: -1
```

Representation of numbers: Decimal system

- Decimal number system is a base-10 system.
- The digits are 0,1,2,3,4,5,6,7,8,9
- For example, 957 is
$$957 = 9*10^2 + 5*10^1 + 7$$

Representation of numbers: Binary system

- Binary number system is a base-2 system.
- The digits are 0, 1
- A digit is called ‘bit’
- Example, 1110 is a binary number.
- Its value is
$$\begin{aligned}1 * 2^3 + 1 * 2^2 + 1 * 2^1 + 0 \\= 13 \text{ in decimal}\end{aligned}$$
- Collection of 8 bits is called a ‘byte’
Example: 11010100

Representation of numbers: Hexadecimal system

- Hexadecimal number system is a base-16 system.
- The digits are 0,1,2,3,4,5,6,7,8,9,A,B,C,D,E,F
- Example, A3B is a Hex number.
- Its value is

$$\begin{aligned} & A * 16^2 + 3 * 16^1 + B \\ & = 10 * 16^2 + 3 * 16^1 + 11 \\ & = 2114 \text{ in decimal} \end{aligned}$$

- A byte consists of two Hex digits
- Example: Let the byte be 11010100

- Hex equivalent is D4

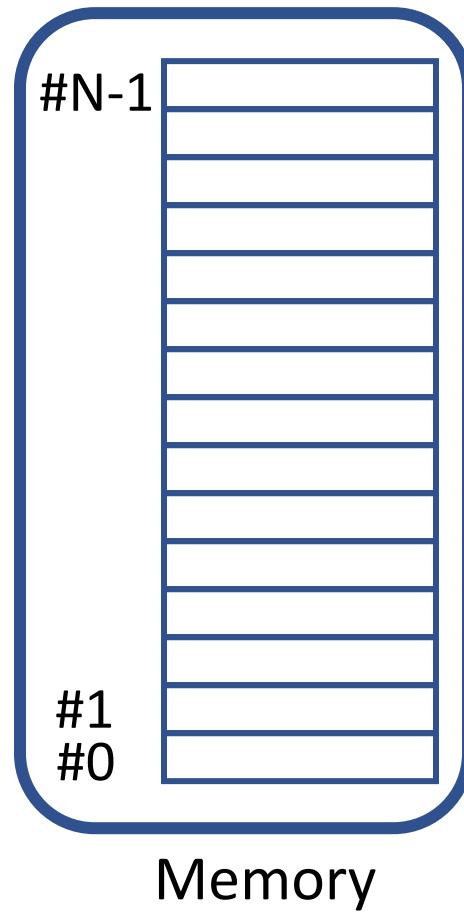
Pointers

Mohammed Bahja

School of Computer Science

University of Birmingham

Programmer's View: Memory as an addressable storage

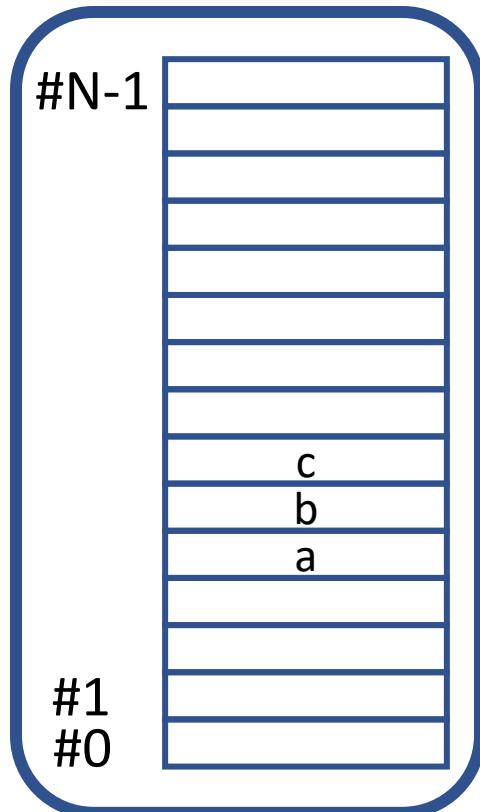
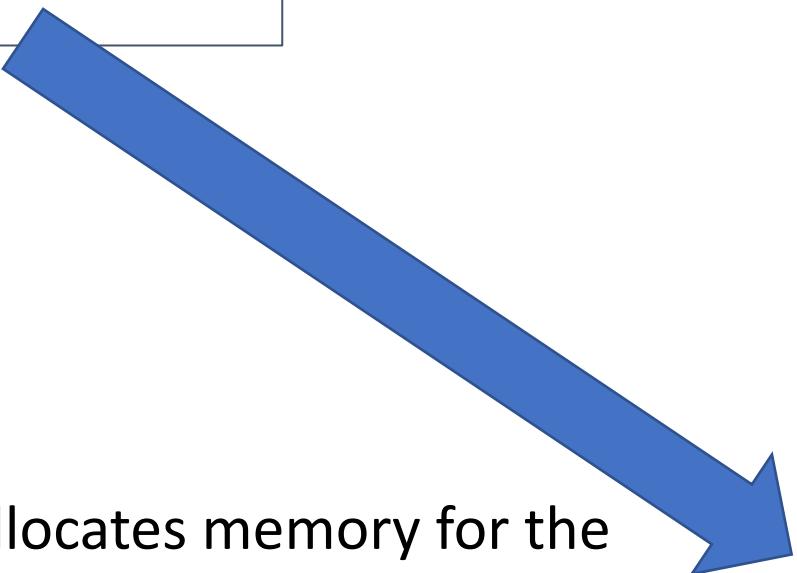


- Memory consists of small 'locations'
- Each location can store a small information

From C program to Memory

```
int a=4, b=5, c;  
c = a*b;
```

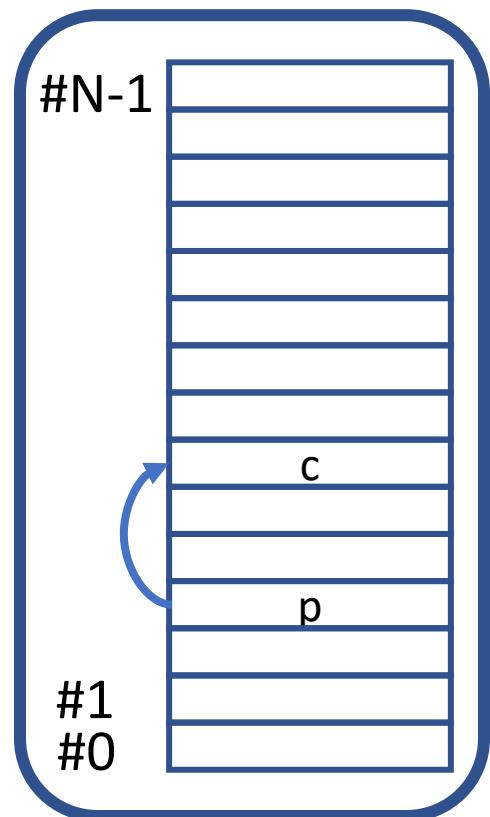
- Compiler allocates memory for the variables.
- Each variable has a unique address.



Pointer

Definition: A pointer is a variable that contains the **address** of a variable.

If 'p' is a pointer to a variable 'c', then the situation will be like this.



A pointer variable is also stored in the memory.

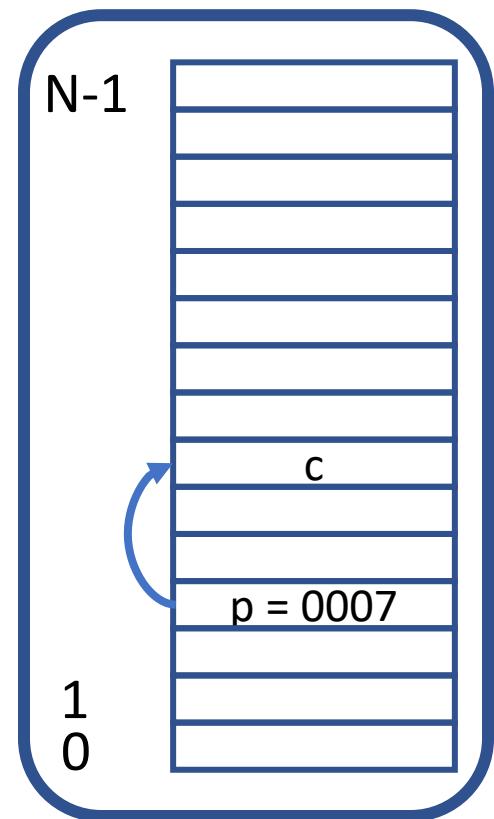
Pointer

Definition: A pointer is a variable that contains the **address** of a variable.

If 'p' is a pointer to a variable 'c', then the situation will be like this.

Example:

If 'c' is present in the memory location with address, say 0007, then the value of p will be 0007.



A pointer variable is also stored in the memory.

Unary operator &

The unary operator ‘&’ is the ‘**address-of**’ operator.
It gives the address of an object.

So,

```
p = &c ;
```

assigns the address of c to p, and p is said to "point to" c.

Unary operator *

The unary operator '*' is called **indirection** or **dereferencing** operator. It is applied to a pointer to accesses the object the pointer points to.

Example:

If p is a pointer to an integer object, say c=5, then

$*p$

will give the value of c, i.e. 5

So,

$p = \&c;$

$c = *p;$

Declaration of a Pointer variable

- A pointer variable is declared as follows.

```
T *p;
```

where **T** is a placeholder for an appropriate data-type.

- p can point to a variable of type **T**.

Example:

```
int *p; // p points to int variable
```

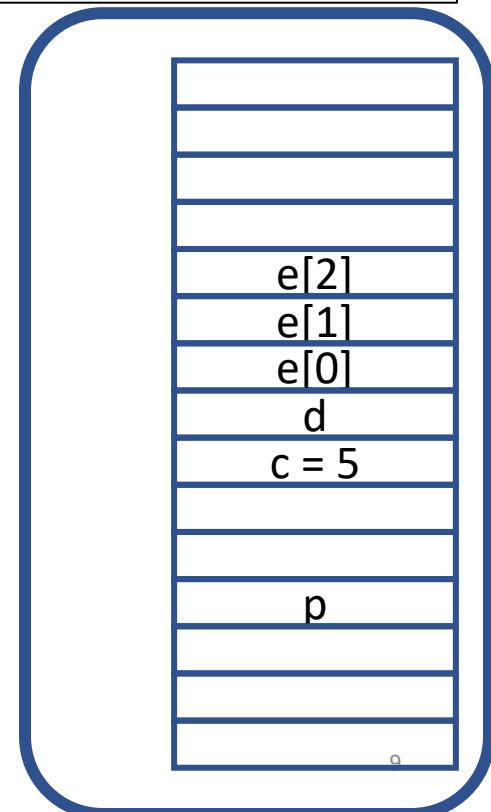


This means, the expression '***p**' is an **int**

Example: Use of pointers

```
int c = 5, d, e[3];
int *p;           // Declared pointer p of type int

p = &c;          // p now points to c
d = *p;          // d is now 5
p = &e[0];        // p now points to e[0]
```

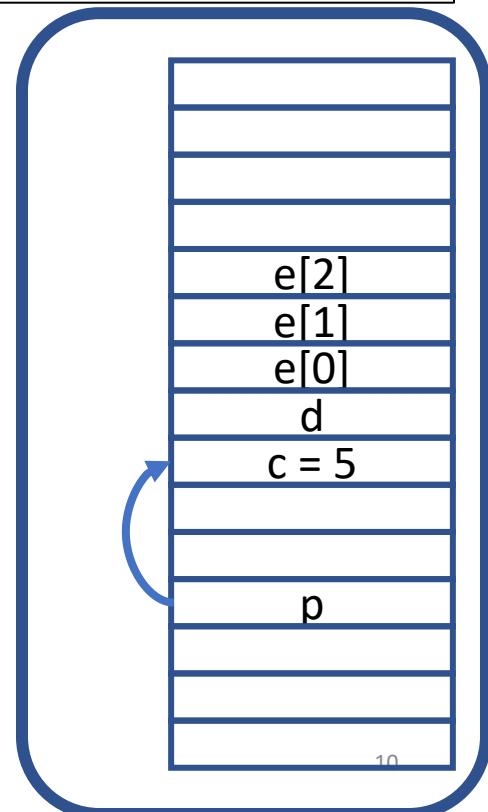


Example: Use of pointers

```
int c = 5, d, e[3];
int *p; // Declared pointer p of type int

p = &c; // p now points to c
d = *p; // d is now 5
p = &e[0]; // p now points to e[0]
```

p contains the address of the memory location where c is residing.

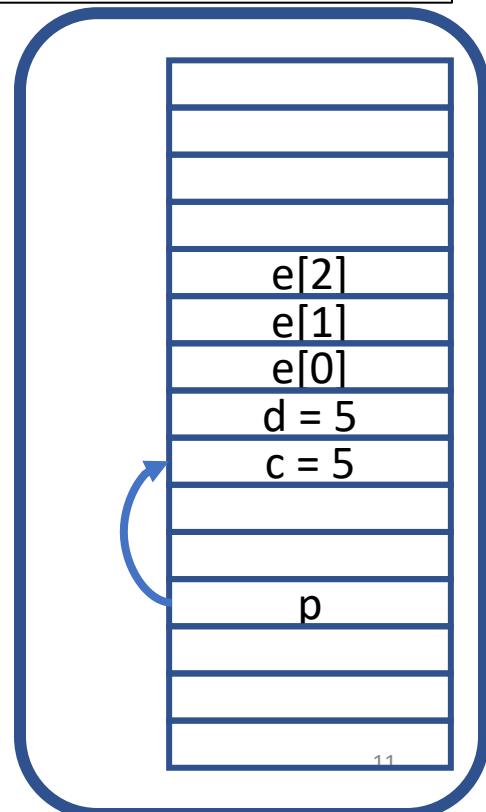


Example: Use of pointers

```
int c = 5, d, e[3];
int *p; // Declared pointer p of type int

p = &c; // p now points to c
d = *p; // d is now 5
p = &e[0]; // p now points to e[0]
```

Dereferencing operator * gives the object pointed by p.
So, d gets the value of c.

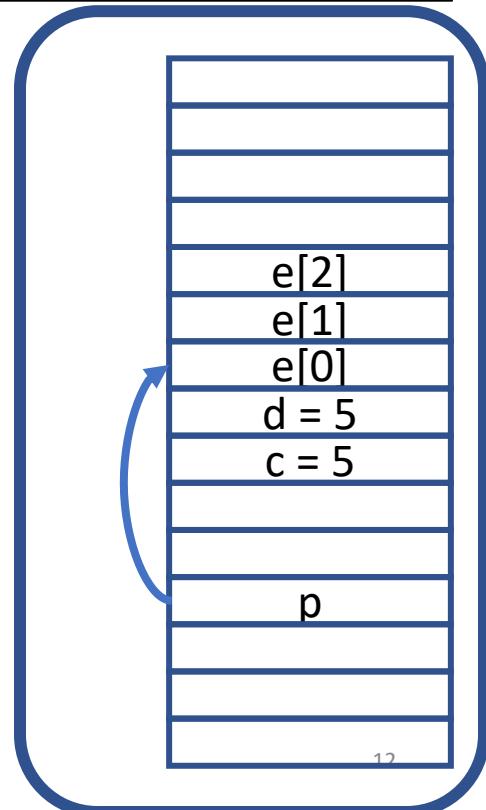


Example: Use of pointers

```
int c = 5, d, e[3];
int *p; // Declared pointer p of type int

p = &c; // p now points to c
d = *p; // d is now 5
p = &e[0]; // p now points to e[0]
```

p now points the first element of array e[].
So, p contains the address of the memory location
where e[0] is residing.



Pointer to an item of different type

- A pointer should point to an object of the same type.
- There are implications if a pointer points to a different type.

```
int i;  
char c;  
int *p;    // p can point to int  
p = &i;    // Correct  
p = &c;    // Can result in wrong calculations
```

Example: Implications of pointing to different type

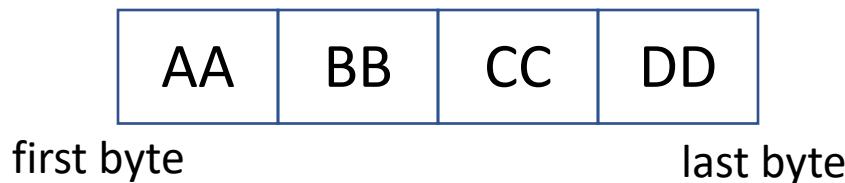
- `int` pointer ‘thinks’ the object it is pointing to is `int` and is of 4 bytes size.
- `char` pointer ‘thinks’ the data it is pointing to is `char` and is of 1 byte size.
- `long long` pointer ‘thinks’ the data it is pointing to is `long long` and is of 8 bytes size.
- [Similar logic for other types ...]

Example: `int i = 0xAABBCCDD;`

Example: Implications of pointing to different type

- `int` pointer ‘thinks’ the object it is pointing to is `int` and is of 4 bytes size.
- `char` pointer ‘thinks’ the data it is pointing to is `char` and is of 1 byte size.
- `long long` pointer ‘thinks’ the data it is pointing to is `long long` and is of 8 bytes size.
- [Similar logic for other types ...]

Example: `int i = 0xAABBCCDD;`

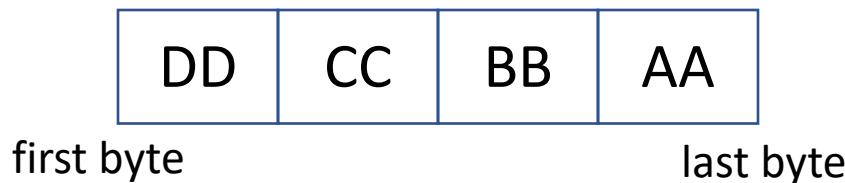


Storage of `i` on big-endian computer

Example: Implications of pointing to different type

- `int` pointer ‘thinks’ the object it is pointing to is `int` and is of 4 bytes size.
- `char` pointer ‘thinks’ the data it is pointing to is `char` and is of 1 byte size.
- `long long` pointer ‘thinks’ the data it is pointing to is `long long` and is of 8 bytes size.
- [Similar logic for other types ...]

Example: `int i = 0xAABBCCDD;`

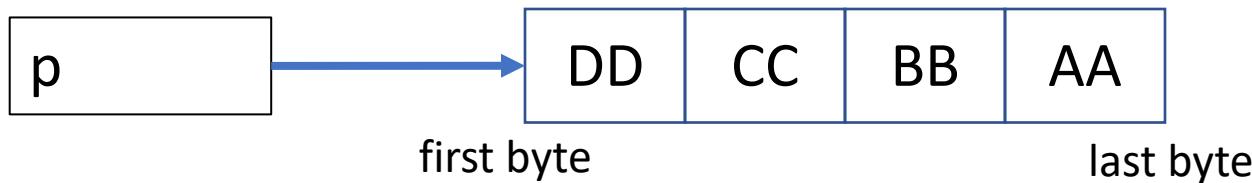


Storage of `i` on little-endian computer

Example: Implications of pointing to different type

- `int` pointer ‘thinks’ the object it is pointing to is `int` and is of 4 bytes size.
- `char` pointer ‘thinks’ the data it is pointing to is `char` and is of 1 byte size.
- `long long` pointer ‘thinks’ the data it is pointing to is `long long` and is of 8 bytes size.
- [Similar logic for other types ...]

Example: `int i = 0xAABBCCDD;` [Consider little-endian]
`type *p = &i;`

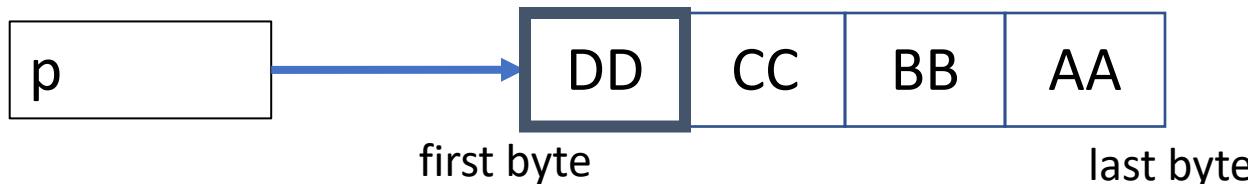


Fact: Pointer always gets the address of the first byte.

Example: Implications of pointing to different type

- `int` pointer ‘thinks’ the object it is pointing to is `int` and is of 4 bytes size.
- `char` pointer ‘thinks’ the data it is pointing to is `char` and is of 1 byte size.
- `long long` pointer ‘thinks’ the data it is pointing to is `long long` and is of 8 bytes size.
- [Similar logic for other types ...]

Example: `int i = 0xAABBCCDD;` [Consider little-endian]
`char *p = &i;`

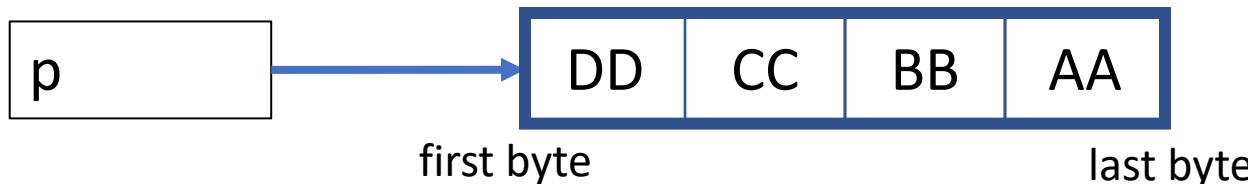


Since `p` is `char` pointer, `*p` has value `0xDD`

Example: Implications of pointing to different type

- `int` pointer ‘thinks’ the object it is pointing to is `int` and is of 4 bytes size.
- `char` pointer ‘thinks’ the data it is pointing to is `char` and is of 1 byte size.
- `long long` pointer ‘thinks’ the data it is pointing to is `long long` and is of 8 bytes size.
- [Similar logic for other types ...]

Example: `int i = 0xAABBCCDD;` [Consider little-endian]
`int *p = &i;`

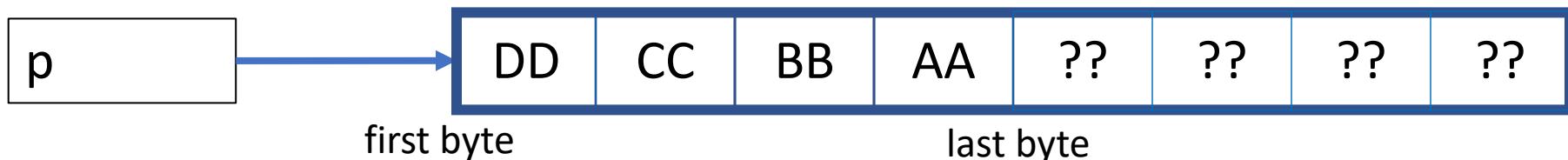


Since `p` is `int` pointer, `*p` has value `0xAABBCCDD`

Example: Implications of pointing to different type

- `int` pointer ‘thinks’ the object it is pointing to is `int` and is of 4 bytes size.
- `char` pointer ‘thinks’ the data it is pointing to is `char` and is of 1 byte size.
- `long long` pointer ‘thinks’ the data it is pointing to is `long long` and is of 8 bytes size.
- [Similar logic for other types ...]

Example: `int i = 0xAABBCCDD;` [Consider little-endian]
`long long *p = &i;`



Since `p` is `long long` pointer, `*p` has value `0xAABBCCDD????????`

Never assign an absolute address to a pointer!

```
int *p = 100; // illegal assignment to pointer
```

Use of pointer in expressions (1)

- Pointer variables can appear in expressions
- If ‘p’ points to the object ‘x’, then *p can occur in any context where x could.

```
int x;  
...  
x = x + 10;
```

is equivalent to

```
int x;  
int *p = &x;  
...  
*p = *p + 10;
```

- Unary operators '*' and '&' have higher precedence than arithmetic operators.

$*p = *p + 10;$

is

$(*p) = (*p) + 10;$

Use of pointer in expressions (2)

$*p = *p + 1;$ is $++*p;$

But, $*p = *p + 1;$ is not $*p++;$

Explanation:

Prefix ++ and -- operators have same precedence as unary * and &

Postfix ++ and -- operators have higher precedence than unary * and &

So, parenthesis is needed.

$*p = *p + 1;$ is $(*p)++;$

Precedence	Operator	Description	Associativity
1	<code>++ --</code> <code>()</code> <code>[]</code> <code>.</code> <code>-></code> <code>(type){ list}</code>	Suffix/postfix increment and decrement Function call Array subscripting Structure and union member access Structure and union member access through pointer Compound literal(C99)	Left-to-right
2	<code>++ --</code> <code>+ -</code> <code>! ~</code> <code>(type)</code> <code>*</code> <code>&</code> <code>sizeof</code> <code>_Alignof</code>	Prefix increment and decrement Unary plus and minus Logical NOT and bitwise NOT Type cast Indirection (dereference) Address-of Size-of Alignment requirement(C11)	Right-to-left
3	<code>* / %</code>	Multiplication, division, and remainder	Left-to-right
4	<code>+ -</code>	Addition and subtraction	
5	<code><< >></code>	Bitwise left shift and right shift	
6	<code>< <=</code> <code>> >=</code>	For relational operators <code><</code> and <code>≤</code> respectively For relational operators <code>></code> and <code>≥</code> respectively	
7	<code>== !=</code>	For relational <code>=</code> and <code>≠</code> respectively	
8	<code>&</code>	Bitwise AND	
9	<code>^</code>	Bitwise XOR (exclusive or)	
10	<code> </code>	Bitwise OR (inclusive or)	
11	<code>&&</code>	Logical AND	
12	<code> </code>	Logical OR	
13	<code>? :</code>	Ternary conditional	Right-to-Left
14	<code>=</code> <code>+= -=</code> <code>*= /= %=</code> <code><<= >>=</code> <code>&= ^= =</code>	Simple assignment Assignment by sum and difference Assignment by product, quotient, and remainder Assignment by bitwise left shift and right shift Assignment by bitwise AND, XOR, and OR	
15	<code>,</code>	Comma	Left-to-right

Use of pointer in expressions (3)

- A pointer variable can be assigned to another pointer variable of the same type.

```
int x;  
int *p1 = &x;  
int *p2;  
int *p3 = p1;  
p2 = p1;
```

x

Use of pointer in expressions (3)

- A pointer variable can be assigned to another pointer variable of the same type.

```
int x;  
int *p1 = &x;  
int *p2;  
int *p3 = p1;  
p2 = p1;
```



Use of pointer in expressions (3)

- A pointer variable can be assigned to another pointer variable of the same type.

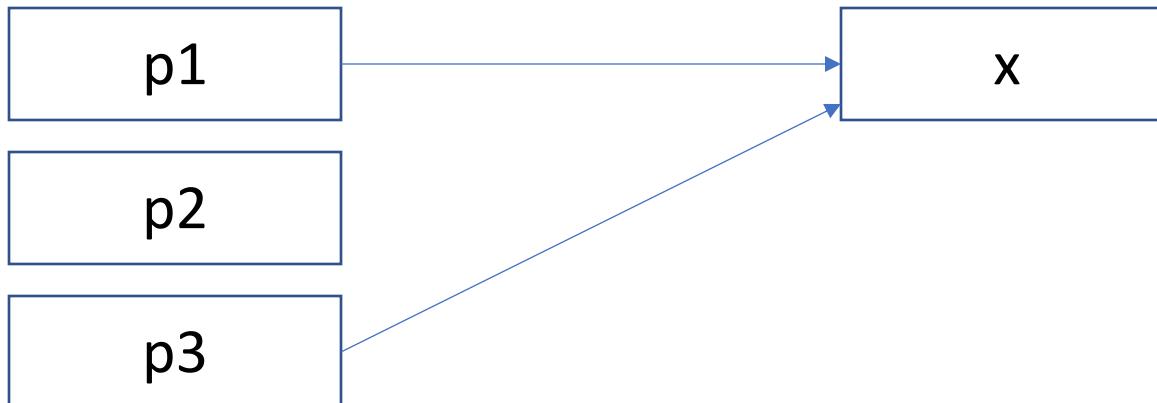
```
int x;  
int *p1 = &x;  
int *p2;  
int *p3 = p1;  
p2 = p1;
```



Use of pointer in expressions (3)

- A pointer variable can be assigned to another pointer variable of the same type.

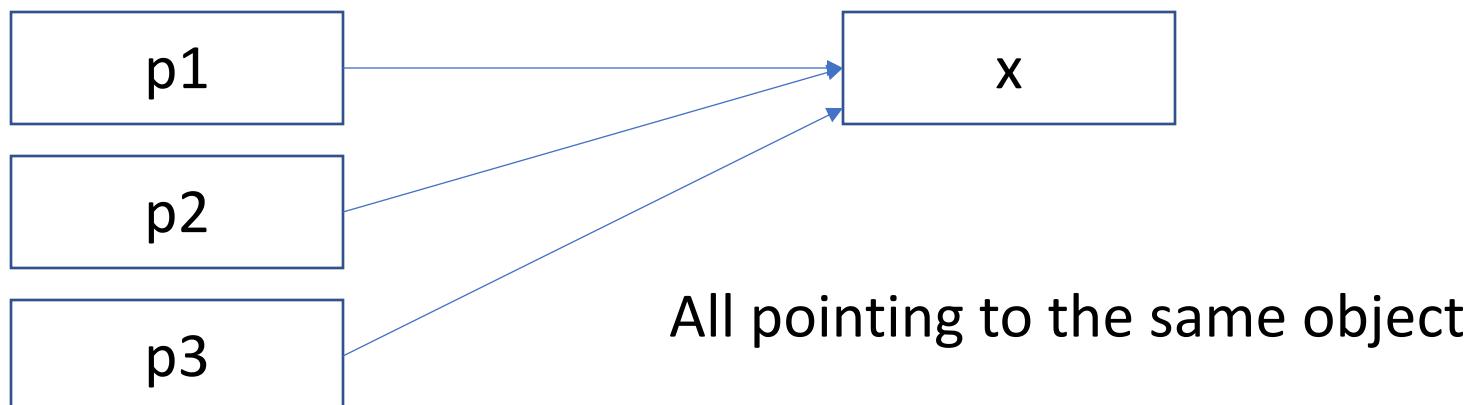
```
int x;  
int *p1 = &x;  
int *p2;  
int *p3 = p1;  
p2 = p1;
```



Use of pointer in expressions (3)

- A pointer variable can be assigned to another pointer variable of the same type.

```
int x;  
int *p1 = &x;  
int *p2;  
int *p3 = p1;  
p2 = p1;
```



Scale factor in pointer expression

- If `p` is a pointer then `p++` or `p+1`
 - points to the next object of the same type. So,
 - value of `p` increments by 4 when `p` is an `int` pointer
 - value of `p` increments by 4 when `p` is a `float` pointer
 - value of `p` increments by 1 when `p` is a `char` pointer
- Scale factor is the number of bytes used by a data-type
- To know scale factor for a data-type, use the `sizeof()` function
- Syntax is: **`sizeof(data_type)`**

Size of different data types

```
int main(){
    printf("Size of int in bytes %d\n", sizeof(int));
    printf("Size of char in bytes %d\n", sizeof(char));
    printf("Size of float in bytes %d\n", sizeof(float));

    printf("Size of int* in bytes %d\n", sizeof(int*));
    printf("Size of char* in bytes %d\n", sizeof(char*));
    printf("Size of float* in bytes %d\n", sizeof(float*));
}
```

Program prints:

Size of int in bytes 4

Size of char in bytes 1

Size of float in bytes 4

Size of int* in bytes 8

Size of char* in bytes 8

Size of float* in bytes 8



See, pointer variables take always 8 bytes, independent of the type it points to. **Why?**

Pointers and Arrays

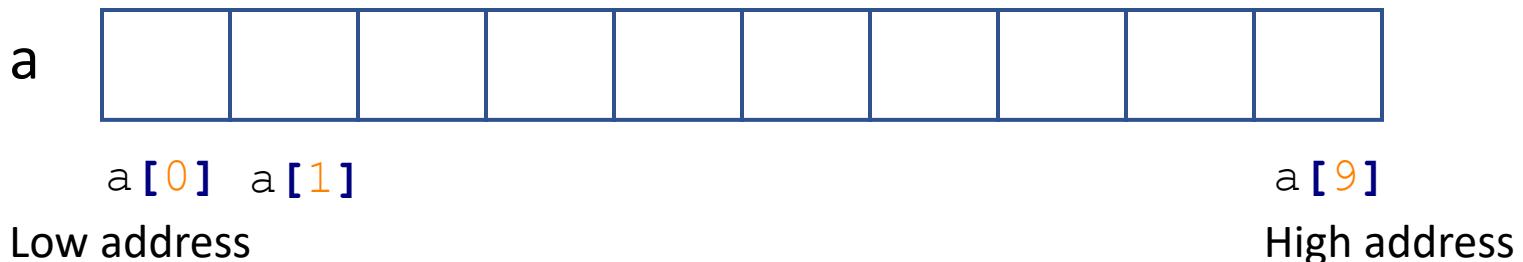
Storage of Array elements

- Array is a **sequential** collection of elements of the same type.

Example:

```
int a[10];
```

is a block of 10 consecutive objects named $a[0], a[1], \dots, a[9]$.

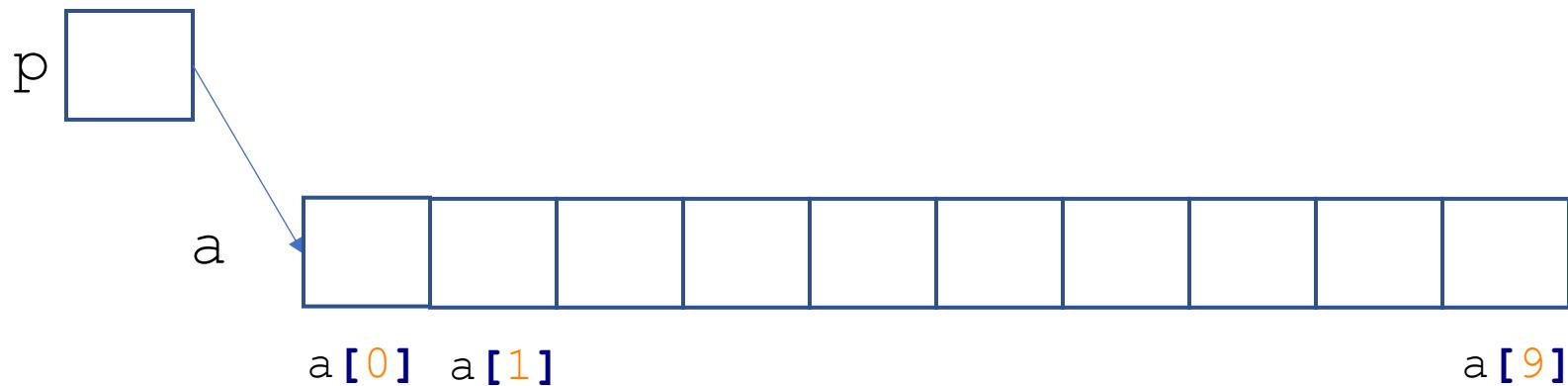


- The notation $a[i]$ refers to the i -th element of the array.

Storage of Array elements

```
int a[10];  
int *p = &a[0];
```

p is a pointer to the first element of array a.



Now

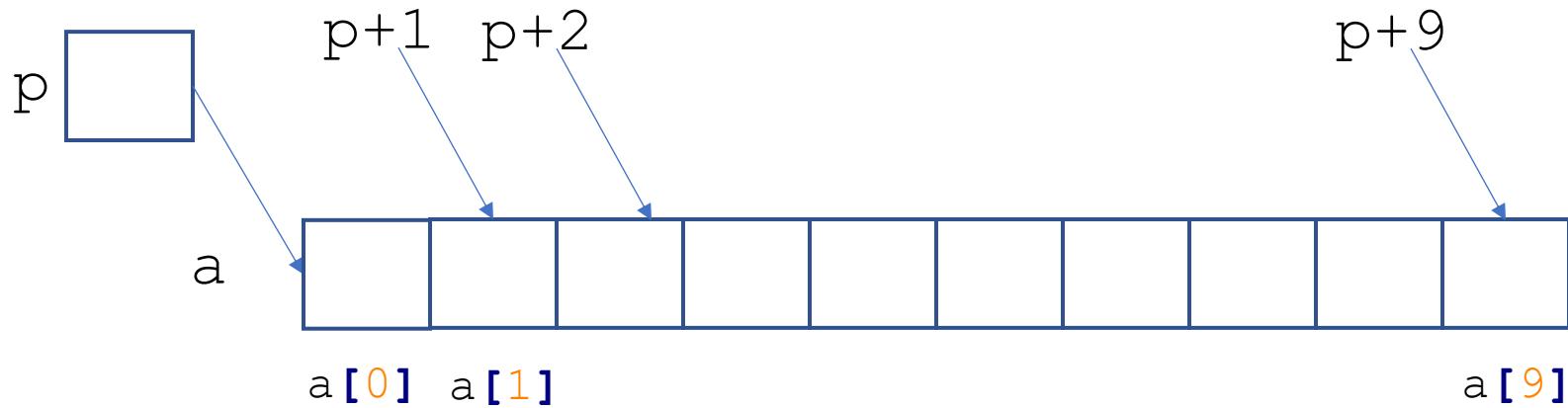
```
int b = *p;
```

will copy value of `a[0]` into `b`.

Accessing Array elements using Pointer

If p points to the first element $a[0]$, then

- $p+1$ points to $a[1]$
- $p+i$ points to $a[i]$



So, $*(p+i)$ refers to the content of $a[i]$

Example: Accessing Array elements using Pointer

Compute the sum of the integer array

```
int a[] = {2,4,5,7,0,1,9,4,8,3,11};
```

```
int sum=0, i;  
for(i=0; i<5; i++)  
    sum = sum + a[i];
```

Using array indexing

```
int *p = &a[0];  
int sum=0, i;  
for(i=0; i<5; i++)  
    sum = sum + *(p+i);
```

Using pointer

Array ‘name’ is an address

Array-name is a synonym for the location of the initial element.
So,

```
int *p = &a[0];
```

can also be written as

```
int *p = a;
```

→ You can think of array-name ‘a’ as pointer to the array a[].

```
int *p = &a[0];
int sum=0, i;
for(i=0; i<5; i++)
    sum = sum + *(p+i);
```

Sum calculation using pointer.

```
int sum=0, i;
for(i=0; i<5; i++)
    sum = sum + *(a+i);
```

Sum calculation: treating array name as pointer.

Array ‘name’ is an address: pitfalls

An array-name is a constant expression, not a variable.

So,

```
int a[10];
int *p = &a[0];
p++; // legal since p is a variable
p=p+1; // legal
```

```
a++; // illegal since a is not a variable
a=a+1; // illegal
```

Pointers to 2D Arrays

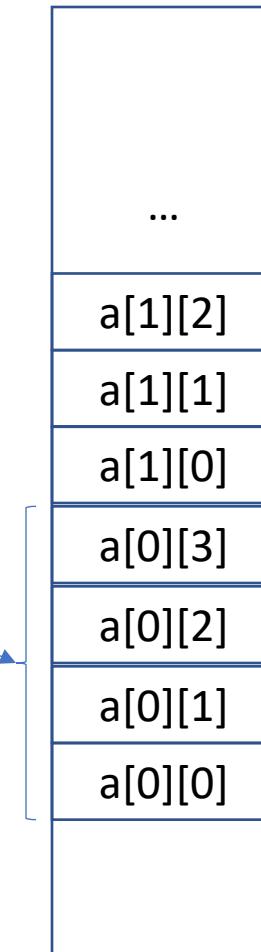
Recap: Memory layout of two-dimensional array

C compiler stores 2D array in **row-major** order

- All elements of Row #0 are stored
- then all elements of Row #1 are stored
- and so on

a[0][0]	a[0][1]	a[0][2]	a[0][3]
a[1][0]	a[1][1]	a[1][2]	a[1][3]
a[2][0]	a[2][1]	a[2][2]	a[2][3]

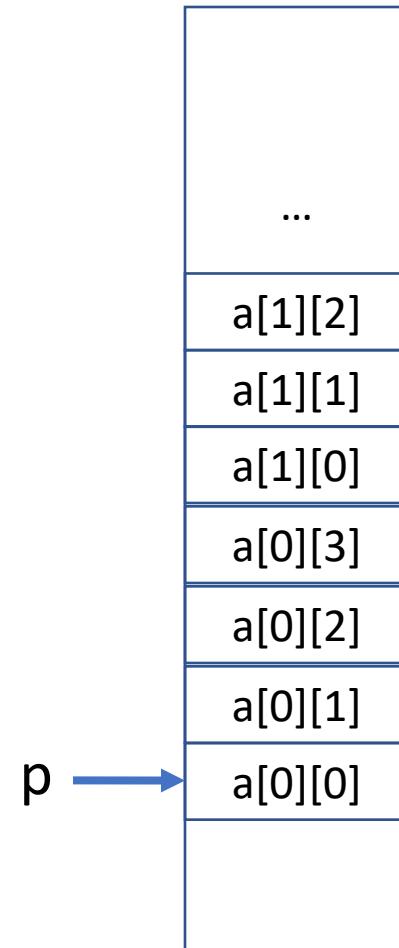
Logical view of array a[3][4]



Memory layout

Accessing 2D array elements using pointer

```
#define ROW 3
#define COL 4
int main(){
    int a[ROW][COL] = {{1,2,3,4},
                        {5,6,7,8},
                        {9,10,11,12}};
    int *p = &a[0][0];
    int i;
    for(i=0; i<ROW*COL; i++)
        printf("%d\n", *(p+i));
    return 0;
}
```



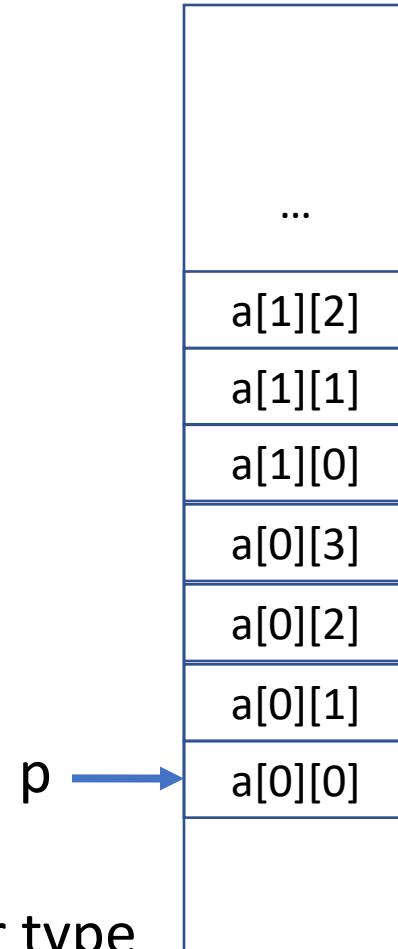
Prints the entire 2D array in row-major order

Accessing 2D array elements using pointer

```
#define ROW 3
#define COL 4
int main(){
    int a[ROW][COL] = {{1,2,3,4},
                        {5,6,7,8},
                        {9,10,11,12}};

    //int *p = &a[0][0];
    int *p = a;
    int i;
    for(i=0; i<ROW*COL; i++)
        printf("%d\n", *(p+i));

    return 0;
}
```



Note: Compiler warns about incompatible pointer type
(explained in the next slide)

Remember: Array names of 1D and 2D arrays

- For an 1D array `a[]`
 - the array name `a` is a constant expression whose value is the address of `a[0]`
 - `a+i` is the address of `a[i]`
- For a 2D array `a[][]`
 - The array name `a` is a constant expression whose value is the address of the 0th **row**
 - `a+i` is the address of the ith **row**

```
int a[ROW][COL] = {{1,2,3,4},  
                    {5,6,7,8},  
                    {9,10,11,12}};  
  
//int *p = &a[0][0];  
int *p = a;
```

In this example

- `p` should point to an `int`
- However, array name ‘`a`’ is the address of 0th **row** (not an `int`). Hence, C compiler warns about incompatible

Pointer and string of characters

- A string of characters is a 1D array of characters
- ASCII code (1 byte) of each character element is stored in consecutive memory locations
- String is terminated by the null character '\0' (ASCII value 0).
- The null string (length zero) is the null character only

```
char name[ ] = "Comp Sc";
```



String access using pointer

```
int main(){
    char name[ ] = "Comp Sc";
    char *ptr = &name[0];

    // print char-by-char
    while(*ptr != '\0'){
        printf("%c", *ptr);
        ptr++;
    }
    return 0;
}
```

\0 is used to indicate termination of a string



Length is 7, but the array is [0 ... 7]

String access using pointer: direct initialization

```
int main(){
    //char name[] = "Comp Sc";
    //char *ptr = &name[0];
    char *ptr = "Comp Sc";

    // print char-by-char
    while(*ptr != '\0'){
        printf("%c", *ptr);
        ptr++;
    }
    return 0;
}
```

With such a declaration,
the string gets stored in the
'read-only' section of the
program code.

String access using pointer: direct initialization

```
int main(){
    //char name[] = "Comp Sc";
    //char *ptr = &name[0];
    char *ptr = "Comp Sc";
    *ptr = 'c';

    // print char-by-char
    while(*ptr != '\0'){
        printf("%c", *ptr);
        ptr++;
    }
    return 0;
}
```

With such a declaration,
the string gets stored in the
'read-only' section of the
program code.

Any attempt to modify
read-only data will cause
'segmentation' fault and the
program will crash.

Outline of this week

General C/Java
programming concepts



C programmer
gets familiar
with comp arch

- Memory layout of a C program
- Dynamic memory management
- Application1: Data structure using dynamic memory management
- Memory hierarchy
- Application2: Exploiting memory hierarchy

Recap: Pointers

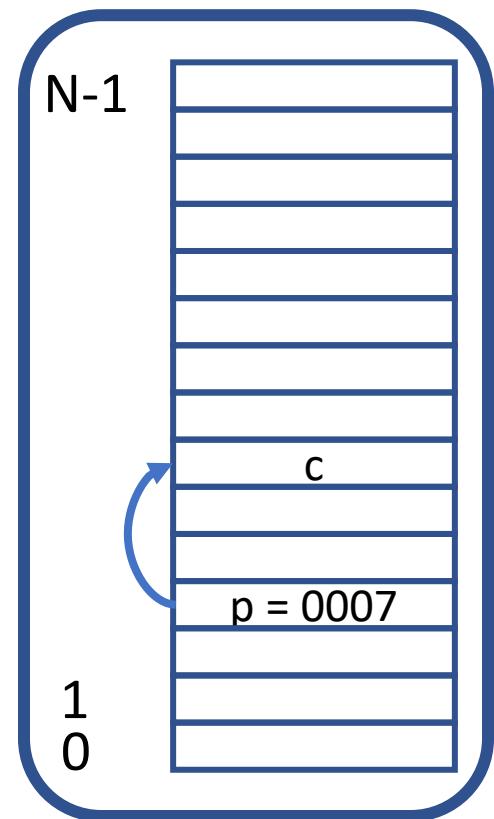
Pointer

Definition: A pointer is a variable that contains the **address** of a variable.

If 'p' is a pointer to a variable 'c', then the situation will be like this.

Example:

If 'c' is present in the memory location with address, say 0007, then the value of p will be 0007.



A pointer variable is also stored in the memory.

Unary operators & *

The unary operator ‘&’ is the ‘**address-of**’ operator.
It gives the address of an object.

The unary operator ‘*’ is called **indirection** or **dereferencing** operator.
It is applied to a pointer to accesses the object the pointer points to.

Example:

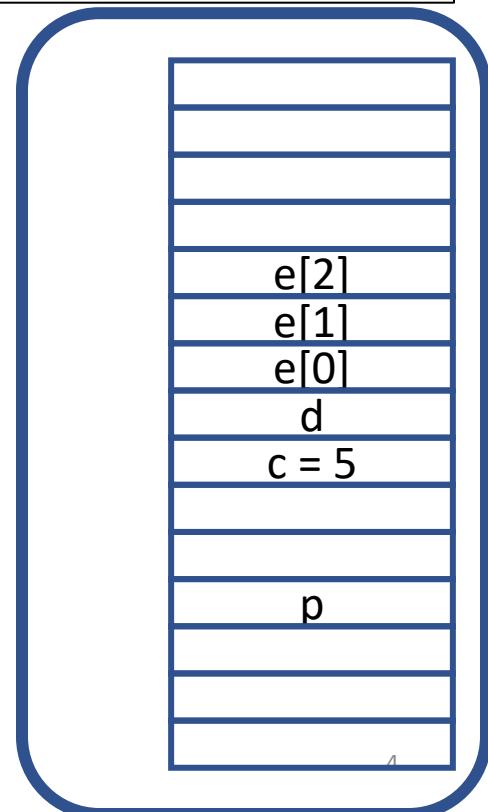
If p is a pointer to an integer object, say c=5, then

```
p = &c;  
c = *p;
```

Example: Use of pointers

```
int c = 5, d, e[3];
int *p; // Declared pointer p of type int

p = &c; // p now points to c
d = *p; // d is now 5
p = &e[0]; // p now points to e[0]
```

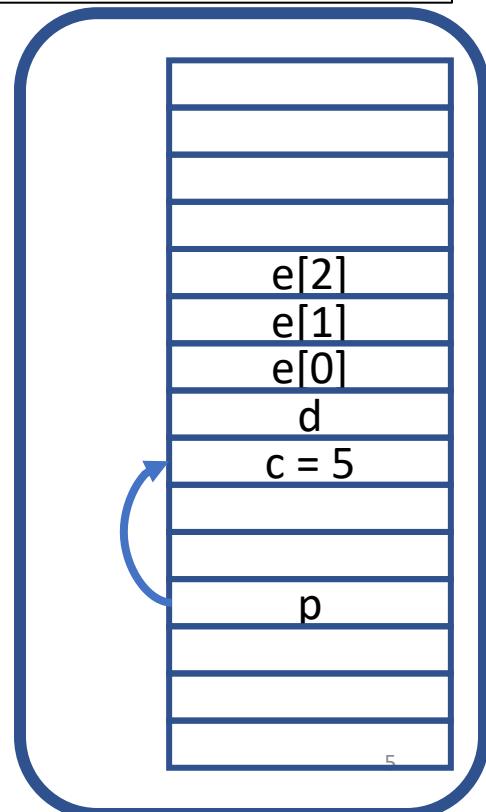


Example: Use of pointers

```
int c = 5, d, e[3];
int *p; // Declared pointer p of type int

p = &c; // p now points to c
d = *p; // d is now 5
p = &e[0]; // p now points to e[0]
```

p contains the address of the memory location where c is residing.

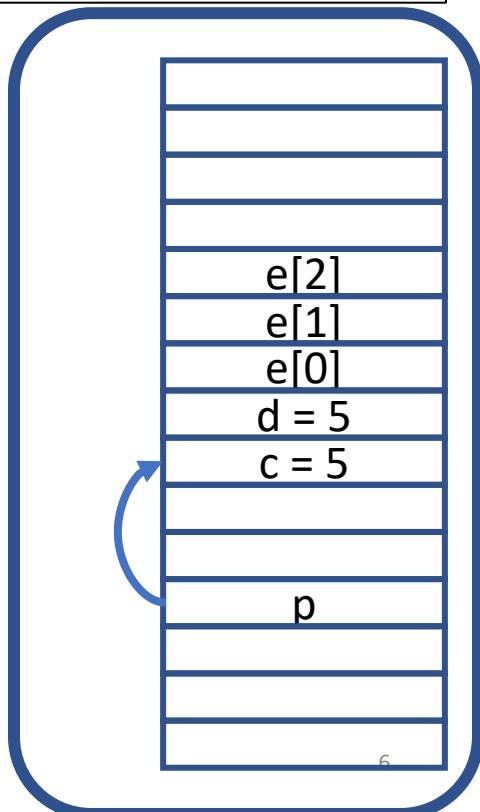


Example: Use of pointers

```
int c = 5, d, e[3];
int *p; // Declared pointer p of type int

p = &c; // p now points to c
d = *p; // d is now 5
p = &e[0]; // p now points to e[0]
```

Dereferencing operator * gives the object pointed by p.
So, d gets the value of c.

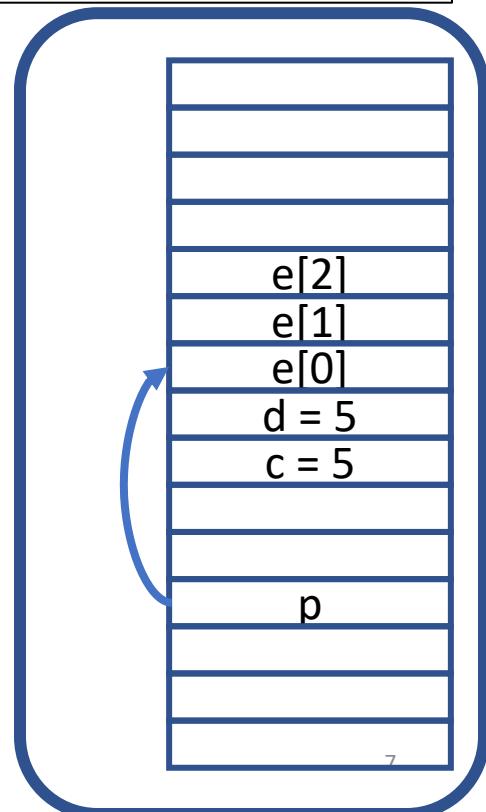


Example: Use of pointers

```
int c = 5, d, e[3];
int *p; // Declared pointer p of type int

p = &c; // p now points to c
d = *p; // d is now 5
p = &e[0]; // p now points to e[0]
```

p now points the first element of array e[].
So, p contains the address of the memory location
where e[0] is residing.



We also covered

- Pointer to 1D array

```
int a[10];
int *p = &a[0];
```

- Pointer to 2D array

```
int a[3][4] = {{1,2,3,4}, {5,6,7,8}, {9,10,11,12}};
int *p = &a[0][0];
```

- Similarly, pointer to string of char
- Pointer expressions. Example: sum of the elements of an array

```
int *p = &a[0];
int sum=0, i;
for(i=0; i<5; i++)
    sum = sum + *(p+i);
```

Memory Layout of a C Program

Mohammed Bahja

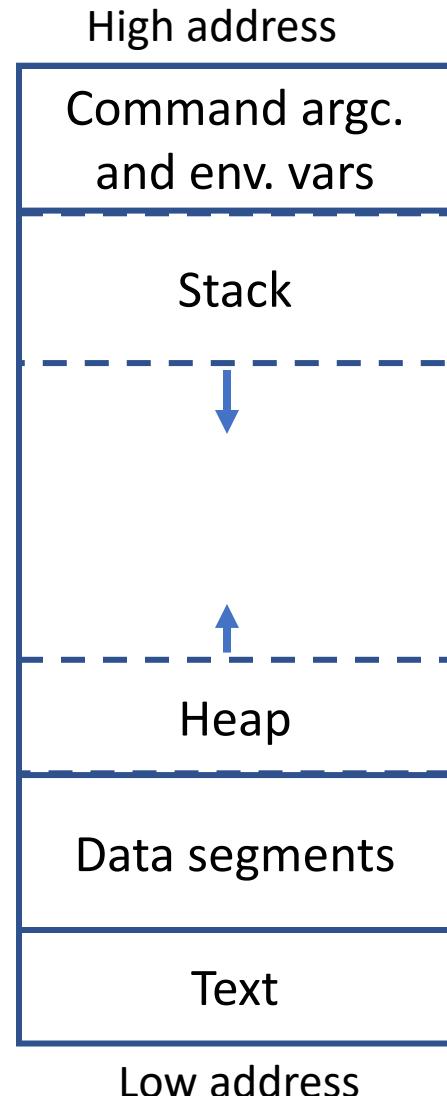
School of Computer Science

University of Birmingham

Memory layout of a C program

Typical memory layout of C program has the following sections:

1. Text or code segment
2. Data segments
3. Stack segment
4. Heap segment

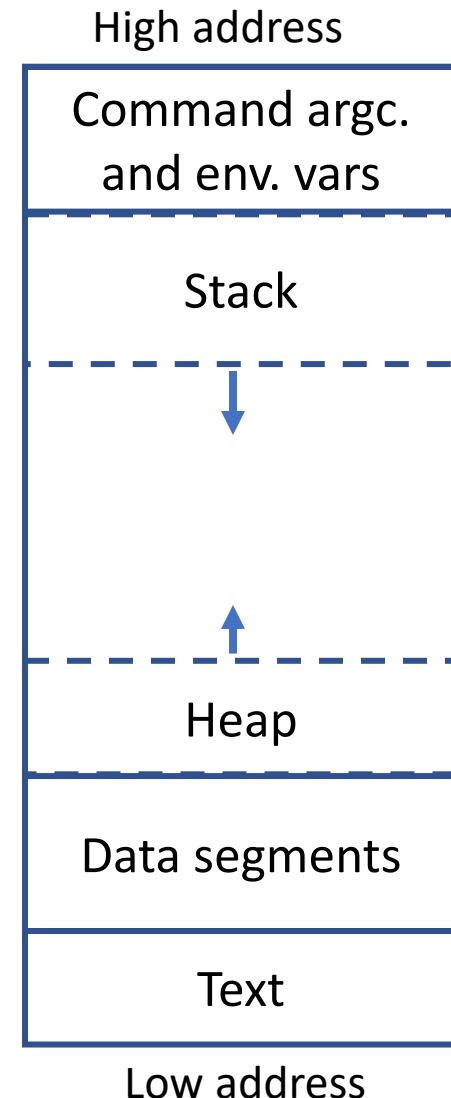


Memory layout of a C program

Typical memory layout of C program has the following sections:

1. **Text or code segment**
2. Data segments
3. Stack segment
4. Heap segment

Text segment contains the the program i.e. the executable instructions.

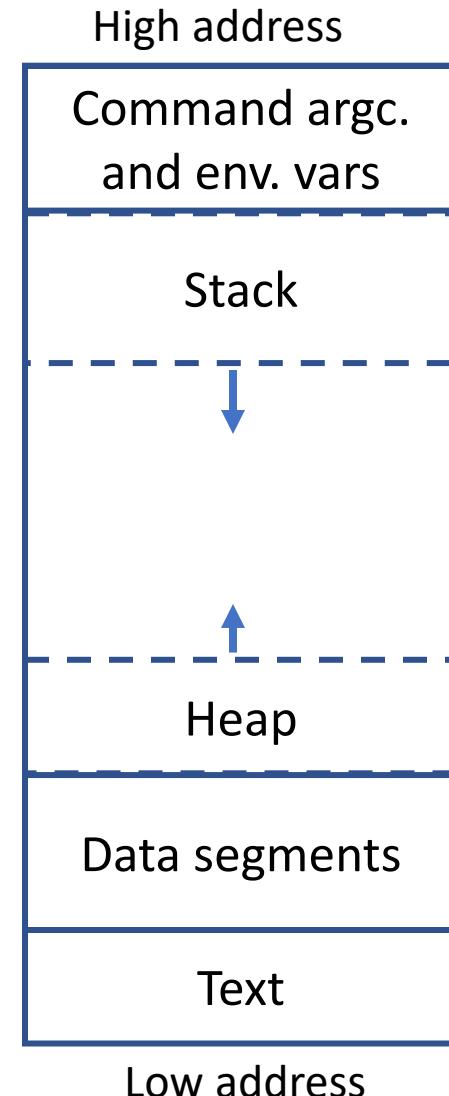


Memory layout of a C program

Typical memory layout of C program has the following sections:

1. Text or code segment
2. **Data segments**
3. Stack segment
4. Heap segment

Two data segments contain initialized and uninitialized global and static variables respectively.

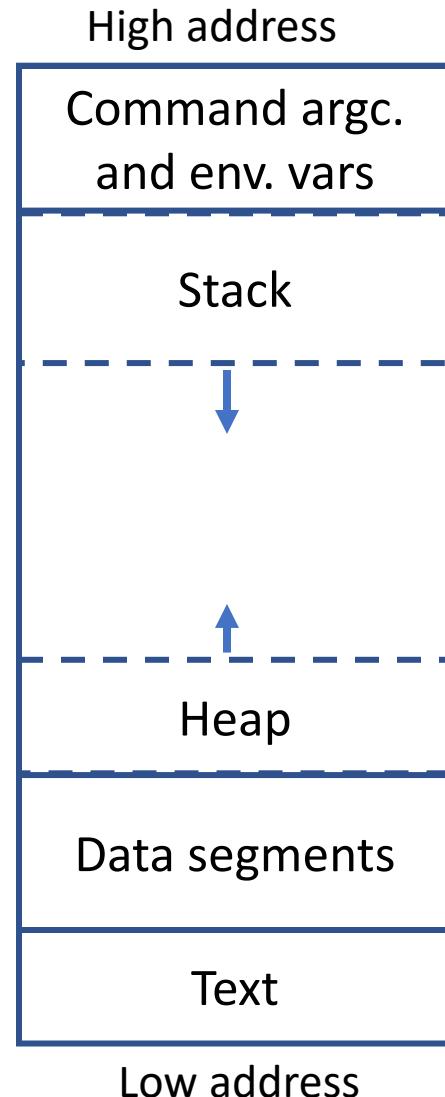


Memory layout of a C program

Typical memory layout of C program has the following sections:

1. Text or code segment
2. Data segments
3. **Stack segment**
4. Heap segment

Stack segment is used to store all local or automatic variables. When we pass arguments to a function, they are kept in stack.

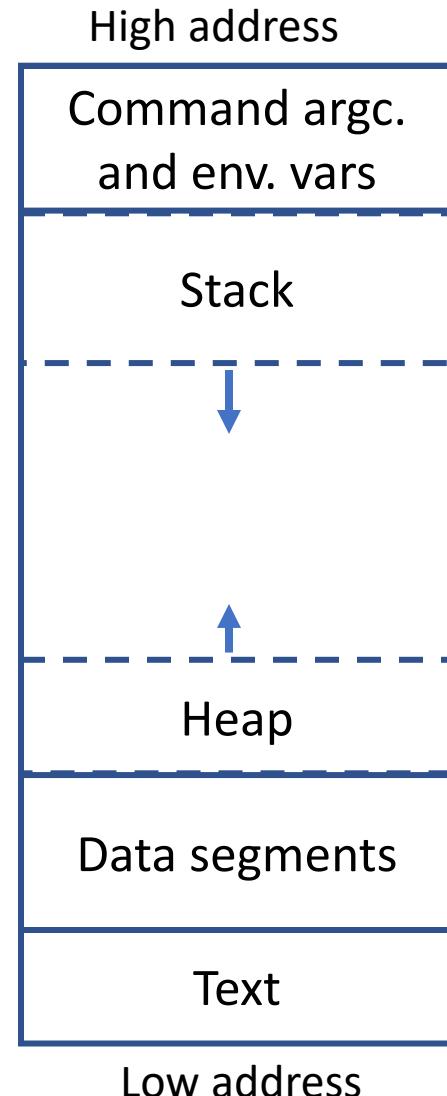


Memory layout of a C program

Typical memory layout of C program has the following sections:

1. Text or code segment
2. Data segments
3. Stack segment
4. **Heap segment**

Heap segment is used to store dynamically allocated variables are stored.



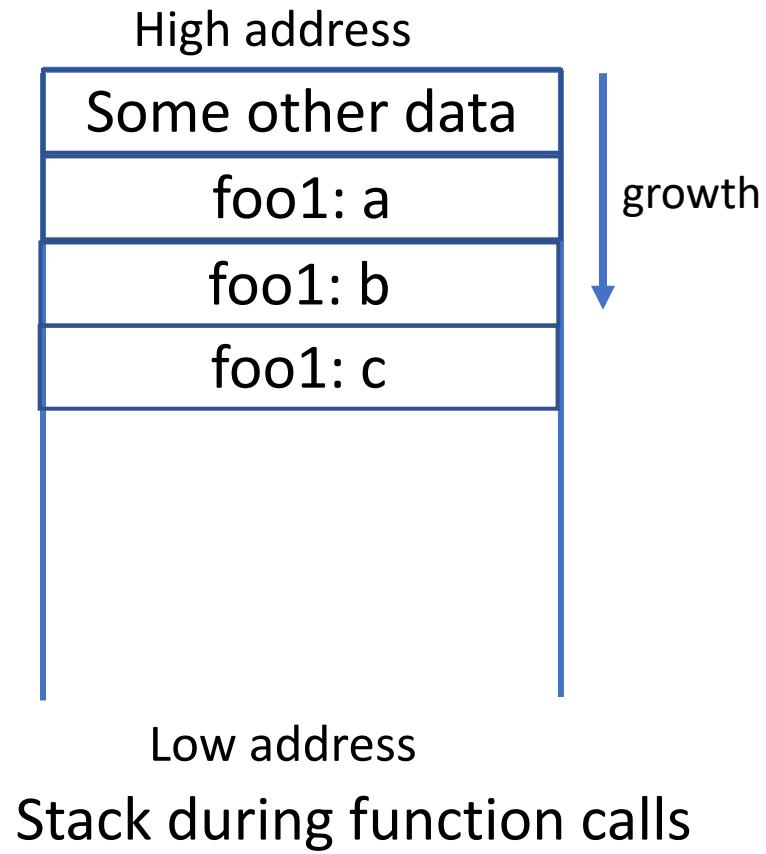
The use of Stack and Heap will be discussed in detail.

Function call: low-level view

- For each function call, a stack frame (portion of stack) is allocated
- Stack grows from high address to low address

1. Before foo2() is called: Stack has data of foo1()

```
T foo2 (T a, T b) {  
    T d;  
    ...  
    ...  
    return d;  
}  
  
foo1 () {  
    T a, b, c;  
    c = foo2 (a, b);  
    ...  
}
```



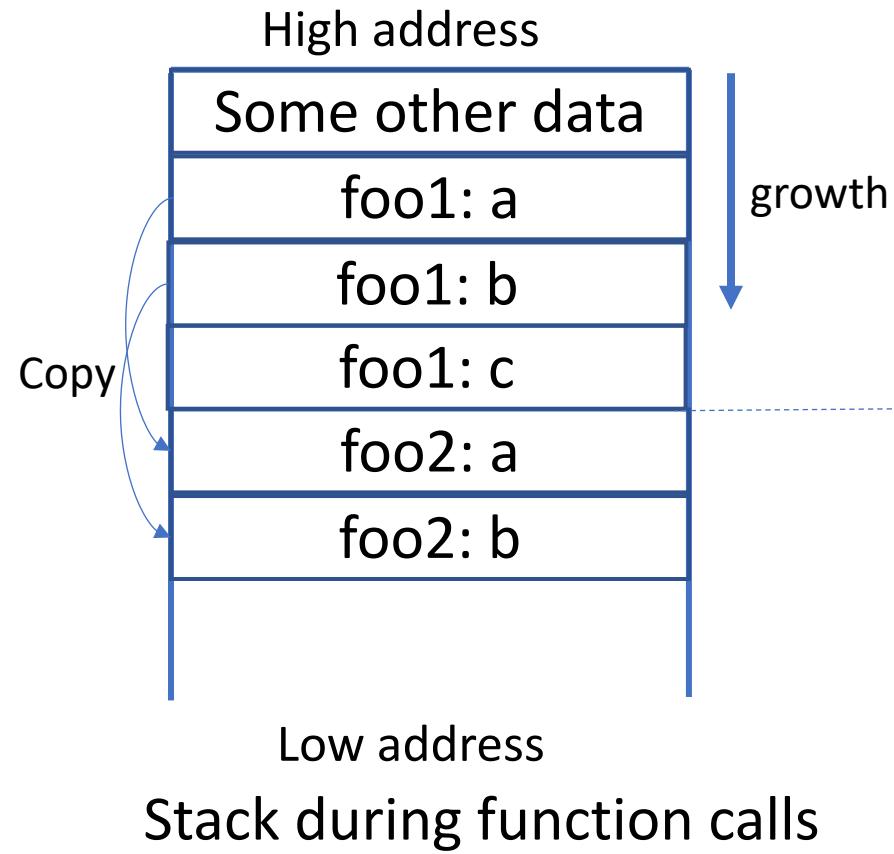
Function call: low-level view

- For each function call, a stack frame (portion of stack) is allocated
- Stack grows from high address to low address

2. When `foo2()` is called: New stack frame for `foo2()` created

3. Function arguments are copied

```
T foo2 (T a, T b) {  
    T d;  
    ...  
    ...  
    return d;  
}  
  
foo1 () {  
    T a, b, c;  
    c = foo2 (a, b);  
    ...  
}
```

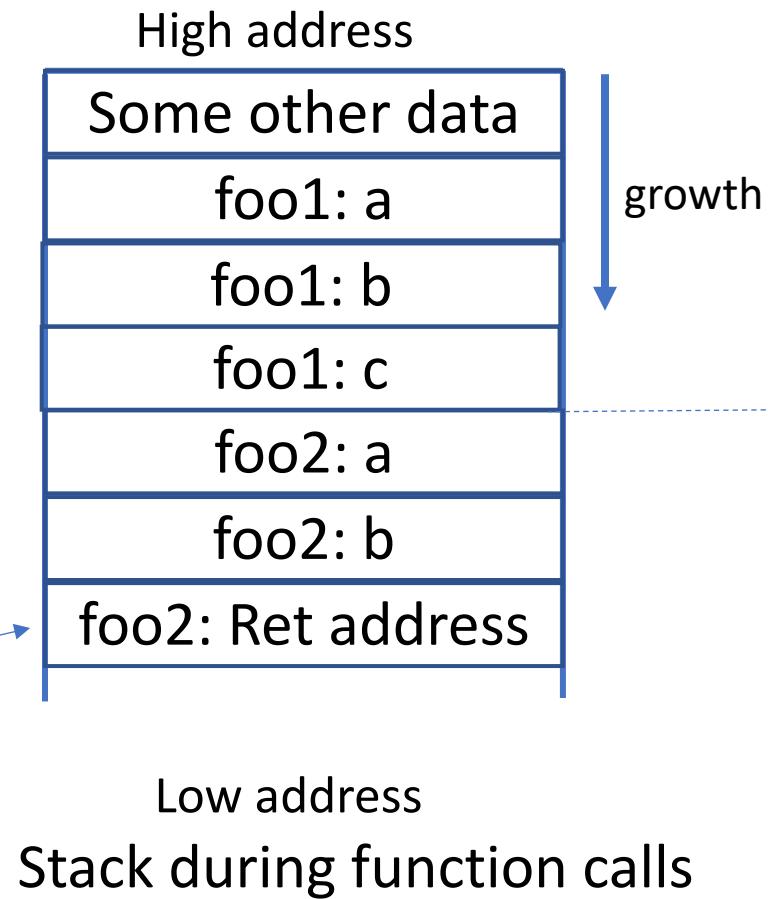


Function call: low-level view

- For each function call, a stack frame (portion of stack) is allocated
- Stack grows from high address to low address

4. The address of the instruction that will be executed from foo1() just after foo2() finishes, is copied. This is called ‘Return address’.

```
T foo2 (T a, T b) {  
    T d;  
    ...  
    ...  
    return d;  
}  
  
foo1 () {  
    T a, b, c;  
    c = foo2 (a, b);  
    ...  
}
```

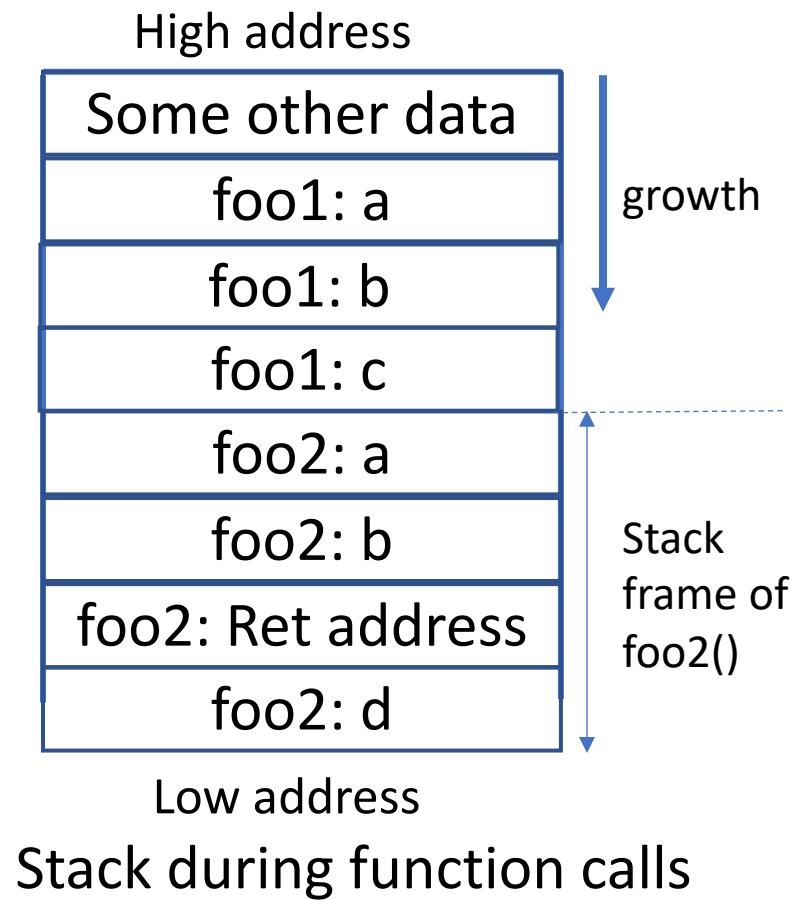


Function call: low-level view

- For each function call, a stack frame (portion of stack) is allocated
- Stack grows from high address to low address

5. Variables within `foo2()`, which are called ‘local variables’ are stored.

```
T foo2 (T a, T b) {  
    T d;  
    ...  
    ...  
    return d;  
}  
  
foo1 () {  
    T a, b, c;  
    c = foo2 (a, b);  
    ...  
}
```

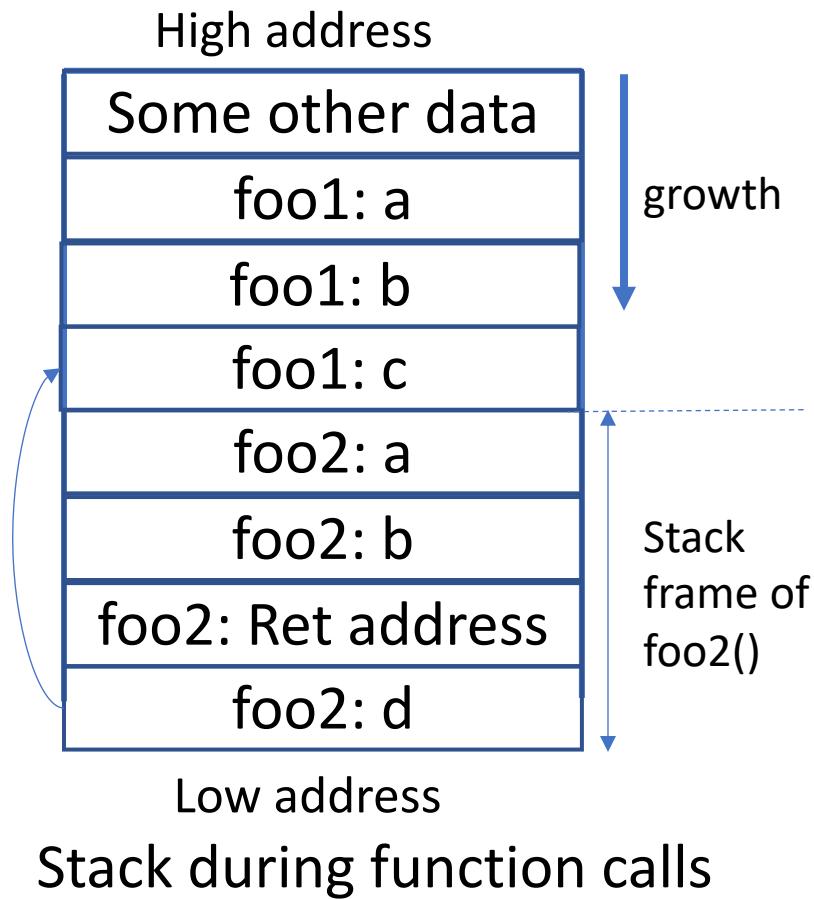


Function call: low-level view

- For each function call, a stack frame (portion of stack) is allocated
- Stack grows from high address to low address

6. After foo2() finishes, local variable d is copied into c of foo1().

```
T foo2 (T a, T b) {  
    T d;  
    ...  
    ...  
    return d;  
}  
  
foo1 () {  
    T a, b, c;  
    c = foo2 (a, b);  
    ...  
}
```



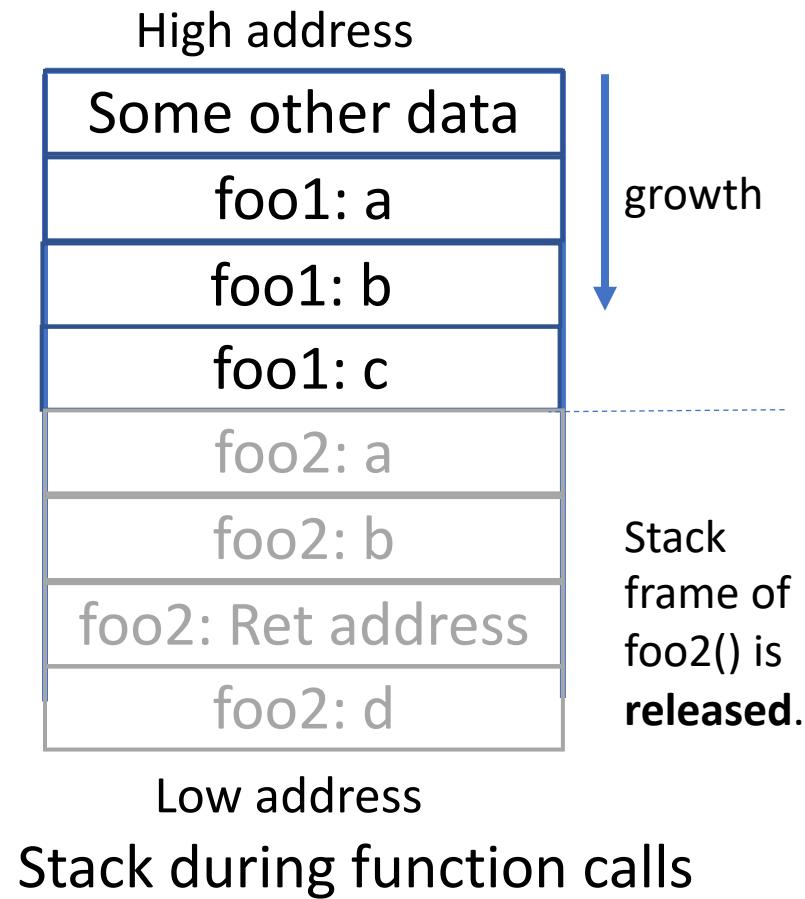
Function call: low-level view

- For each function call, a stack frame (portion of stack) is allocated
- Stack grows from high address to low address

7. Following the return address, control-flow returns to `foo1()`.

8. Stack frame for `foo2()` is released. Hence, all local variables die.

```
T foo2 (T a, T b) {  
    T d;  
    ...  
    ...  
    return d;  
}  
  
foo1 () {  
    T a, b, c;  
    c = foo2 (a, b);  
    ...  
}
```



Stack during function call: Scope

- Scope: part of the program where a variable can be used (or seen)
- Local variables within a function: scope is the function.
They are allocated in the stack-frame of the function. After the function call, the stack-frame is released.

```
T foo2 (T a, T b) {  
    T d; // scope of is foo2  
    ...  
    ...  
    return d;  
}  
foo1 () {  
    T a, b, c; // scope of is foo1  
    c = foo2 (a, b);  
    ...  
}
```

Static variables

- Static variables are stored in the data segment, **not in the stack**.
- The data segment is active during the entire life-time of program
- So, static variables preserve their values even after they are out of their scope.

```
foo(int b) {  
    int a=0;  
    a = a+b;  
    printf("a=%d", a);  
}  
  
main () {  
    foo (1);  
    foo (1);  
}
```

Scope of 'a' is foo().
Outputs will be 1 and 1
both times.

```
foo(int b) {  
    static int a=0;  
    a = a+b;  
    printf("a=%d", a);  
}  
  
main () {  
    foo (1);  
    foo (1);  
}
```

Static 'a' is preserved.
Outputs will be 1 and 2.

Global variables

- A global variable is declared outside all functions.
- It can be read or updated by all functions.
- Careful: Do not name any local var in the name of a global var.

```
int A_g = 5; // Global variable

int foo(int b) {
    int a=0; // Local variable
    a = a+b+A_g;
    return a;
}

main() {
    int a, b=1; // Local variable
    a=foo(1);
}
```

```
void swap(int x, int y) {  
    int temp;  
    temp = x;  
    x = y;  
    y = temp;  
}  
int main() {  
    int a=4, b=5;  
    swap(a, b);  
    printf("a=%d b=%d", a, b);  
    return 0;  
}
```

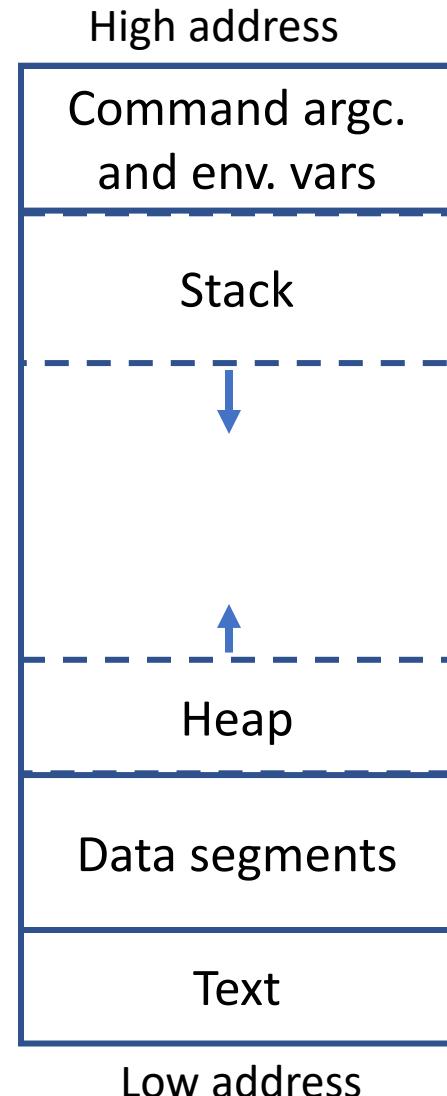
What values of a and b will be printed?

```
void swap(int x, int y) {  
    int temp;  
    temp = x;      Changes are local,  
    x = y;          not visible from main().  
    y = temp;  
}  
int main() {  
    int a=4, b=5;  
    swap(a, b);  
    printf("a=%d b=%d", a, b);  
    return 0;  
}
```

The program will print a=4 and b=5.

Conclusions so far

- Local variables use Stack ‘temporarily’. They are active only within their functions.
- Static variables are stored in the Data segments. They preserve their values.
- Global variables are stored in the Data segments. They are accessible to all functions of the C program.



Use of Heap will be covered in Dynamic Memory Management.

Passing Pointers to a Function

Mohammed Bahja

School of Computer Science

University of Birmingham

Pass-by-reference and Pass-by-value

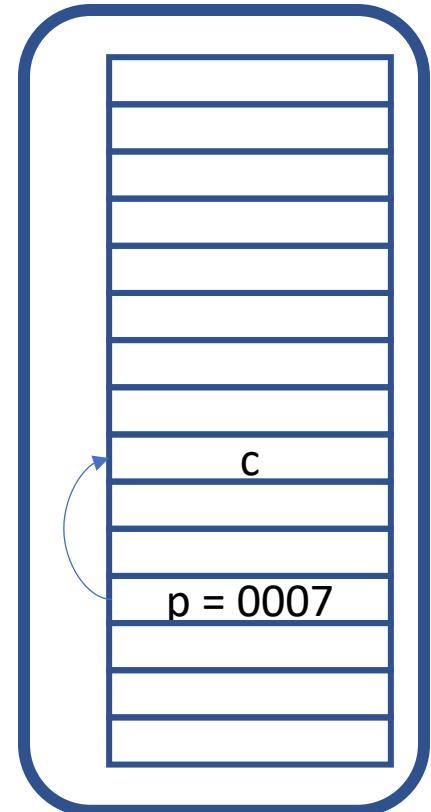
- We have seen how to pass data objects to a function as arguments. This technique is called ‘pass-by-value’.
- We can pass pointers to a function as arguments.
- This is known as ‘pass-by-reference’.

```
foo(int c) {  
    c=c*5;  
    ...  
}  
  
int main() {  
    int c=5;  
    foo(c);  
}
```

Passing object to foo().

```
foo(int *p) {  
    *p=*p*5;  
    ...  
}  
  
int main() {  
    int c=5;  
    int *p = &c;  
    foo(p);  
}
```

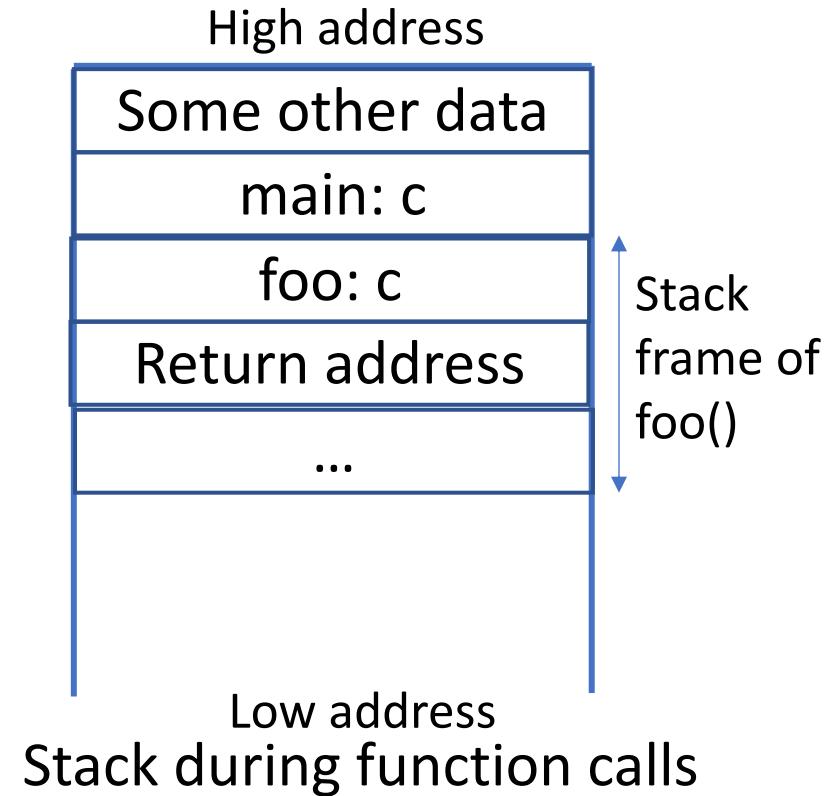
Passing pointer to foo()



Pass-by-reference vs Pass-by-value: difference

```
foo(int c) {  
    c=c*5; // Scope is foo  
    ...  
}  
int main() {  
    int c=5;  
    foo(c);  
}
```

Example of pass-by-value



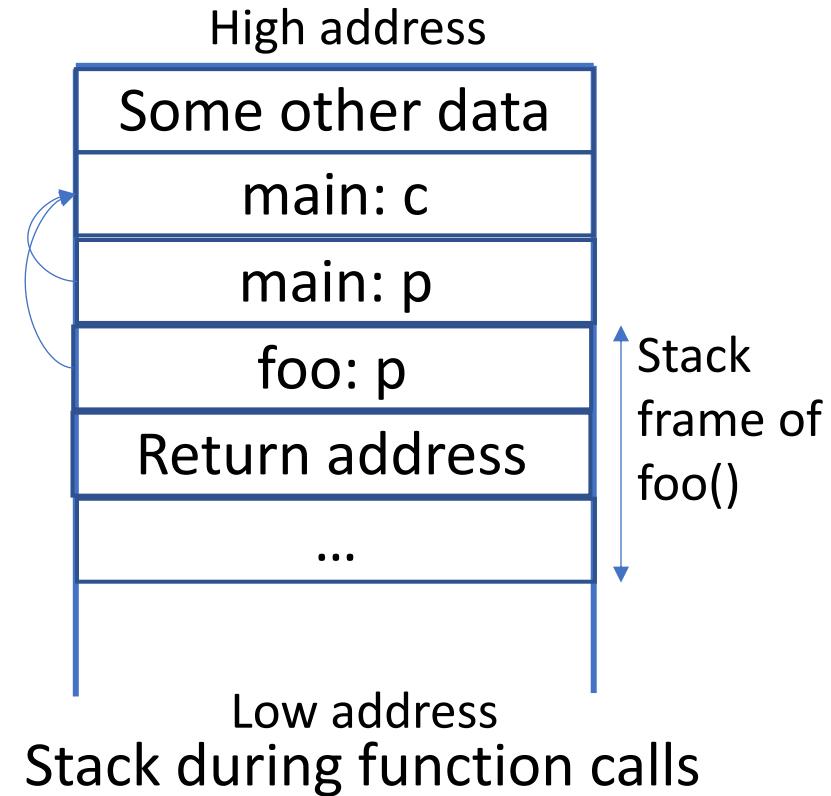
Consequences:

- foo() gets a local copy of c.
So, $c=c*5=25$ happens only within foo().
- main() still sees $c=5$.

Pass-by-reference vs Pass-by-value: difference

```
foo(int *p) {  
    *p=*p*5;  
    ...  
}  
  
int main() {  
    int c=5;  
    int *p = &c;  
    foo(p);  
}
```

Example of pass-by-reference



Consequences:

- `foo()` gets a local copy of `p` which contains the address of `c`.
So, $*p = *p * 5 = 25$ updates the memory location where `c` is stored.
- Both `foo()` and `main()` see `c=25`.

Example: swapping two integers

```
void swap(int x, int y) {  
    int temp;  
    temp = x;      Changes are local,  
    x = y;          not visible from main().  
    y = temp;  
}  
int main() {  
    int a=4, b=5;  
    swap(a, b);  
    printf("a=%d b=%d", a, b);  
    return 0;  
}
```

The program will print a=4 and b=5.

Example: swapping two integers

```
void swap(int *x, int *y) {  
    int temp;  
    temp = *x;  
    *x = *y;  
    *y = temp;  
}  
  
int main() {  
    int a=4, b=5;             
    swap(&a, &b);  
    printf("a=%d b=%d", a, b);  
    return 0;  
}
```

The program will print swapped values, i.e. a=5 and b=4.

Returning pointer from function

- A function can return a pointer.

```
int *foo(...) // Returns pointer to an int
```

```
char *foo(...) // Returns pointer to a char
```

```
float *foo(...) // Returns pointer to a float
```

Returning pointer from function

- A function can return a pointer.

Example: Find the maximum value and return the pointer.

```
int *max(int *a, int *b) {
    if(*a > *b) return a;
    else return b;
}

int main() {
    int a=4, b=5;
    int *c;
    c=max(&a, &b);
    printf("Max value=%d", *c);
    return 0;
}
```

Returning pointer from function: pitfalls

Careful: Never return pointer to a local variable.

```
int *max(int *a, int *b) {  
    int temp;  
    if(*a > *b) temp=*a;  
    else temp=*b;  
    return &temp  
}
```

```
int main () {  
    int a=4, b=5;  
    int *c;  
    c=max (&a, &b);  
    printf ("Max value=%d", *c);  
    return 0;  
}
```

temp is a local object.

After function call, temp doesn't exist.

But c points to temp.

So, c points to an object which does not exist.

Dynamic Memory Management

Mohammed Bahja

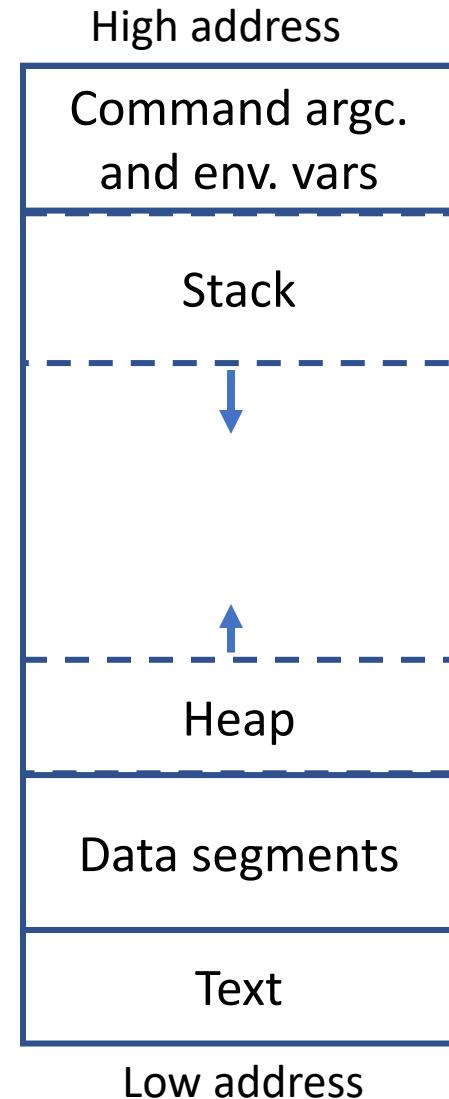
School of Computer Science

University of Birmingham

Recap: Memory layout of C program

- Local variables use Stack ‘temporarily’. They are active only within their functions.
- Static variables are stored in the Data segments. They preserve their values.
- Global variables are stored in the Data segments. They are accessible to all functions of the C program.

[This figure is symbolic. Heap section need not start from the address just after data segment]



Use of Heap will be covered in Dynamic Memory Management.

Application scenario

Consider an application where the volume of data is not known beforehand

Example: Store info of the people you meet during office hour

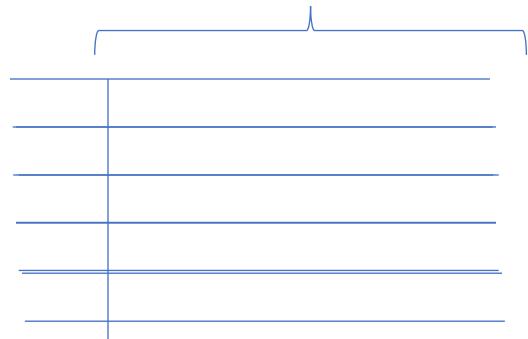
- The number of people is not known
- Each person may provide different amount of info

Solution 1: Allocate a large array

```
char info[10000][2000];
```

10K rows

2K char of info.



Application scenario

Consider an application where the volume of data is not known beforehand

Example: Store info of the people you meet during office hour

- The number of people is not known
- Each person may provide different amount of info

Solution 1: Allocate a large array

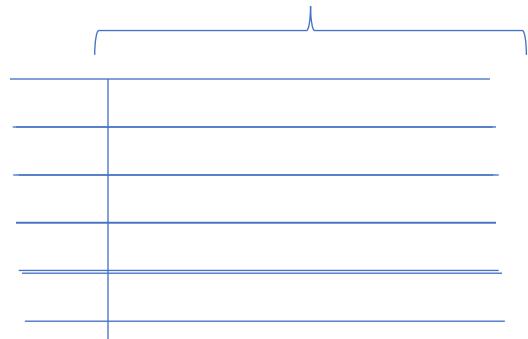
```
char info[10000][2000];
```

Problems:

1. May be wasteful
2. May be insufficient
3. Tiny computers can't afford large amount of space

10K rows

2K char of info.



Application scenario

Consider an application where the volume of data is not known beforehand

Example: Store info of the people you meet during office hour

- The number of people is not known
- Each person may provide different amount of info

Best solution: Allocate memory at **runtime** as per **demand**

Advantages: Optimal memory consumption depending on the current state of the application

Dynamic memory allocation in C

Dynamic memory allocation functions

C provides dynamic memory management functions in stdlib.h

- `malloc()`
 1. Allocates requested number of bytes of **contiguous** memory from the **Heap**.
 2. Returns a pointer to the first byte of the allocated space.
 3. If memory allocation fails, then `malloc()` returns NULL pointer.

Syntax:

`T *p;`

`p = (T *) malloc(number of bytes);`

Dynamic memory allocation functions

C provides dynamic memory management functions in stdlib.h

- `malloc()`
 1. Allocates requested number of bytes of **contiguous** memory from the **Heap**.
 2. Returns a pointer to the first byte of the allocated space.
 3. If memory allocation fails, then `malloc()` returns NULL pointer.

Example: Allocate space for 3 `int`

```
int *p;  
p = (int *) malloc(3*4); // int is 4 bytes
```



Allocates 12 bytes of contiguous memory in Heap.
p points to the first byte

Dynamic memory allocation functions

C provides dynamic memory management functions in stdlib.h

- `malloc()`
 1. Allocates requested number of bytes of **contiguous** memory from the **Heap**.
 2. Returns a pointer to the first byte of the allocated space.
 3. If memory allocation fails, then `malloc()` returns NULL pointer.

Example: Allocate space for 3 `int`

```
int *p;  
p = (int *) malloc(3*sizeof(int));  
// Allocates memory for 3 integers
```

Use `sizeof()` for convenience.

It returns the size of a `data_type` in bytes.

Dynamic memory allocation functions

C provides dynamic memory management functions in stdlib.h

- `malloc()`
 1. Allocates requested number of bytes of **contiguous** memory from the **Heap**.
 2. Returns a pointer to the first byte of the allocated space.
 3. If memory allocation fails, then `malloc()` returns NULL pointer.

Example: Allocate space for 3 int

```
int *p;  
if((p= (int *) malloc(3*sizeof(int)))==NULL) {  
    printf("Allocation failed");  
    exit(-1);  
}
```

Good practice: check if a NULL pointer is returned by `malloc()`

Releasing allocated memory using free()

- Dynamically allocated block of memory can be released using `free()`.
- After `free()`, the block of memory is returned to the Heap.

```
int *p;
// Block of memory is allocated
if((p= (int *) malloc(3*sizeof(int))))==NULL){
    printf("Allocation failed");
    exit(-1);
}

// [Some statements involving p]

// Block of memory pointed by p is released
free(p);
```

Example: Array of variable length

```
#include<stdio.h>
#include<stdlib.h>
int main() {
    int N, i;
    int *p;
    printf("Provide array size:");
    scanf("%d", &N);
    // Allocate memory in Heap for
    // array pointed by p
    if((p= (int *) malloc(N*sizeof(int)))==NULL) {
        printf("Allocation failed");
        exit(-1);
    }
    printf("Provide %d integers\n", N);
    for(i=0; i<N; i++)
        scanf("%d", p+i);

    free(p);
    return 0;
}
```

Memory leak

If memory allocated using `malloc()` is not `free()`-ed, then the system will “leak memory”

- Block of memory is allocated, but not returned to Heap
- The program therefore grows larger over time
- In C, it is **your responsibility** to prevent memory leak

Example: Memory leak

```
int main() {
    int N, i, *p;

    while(1) {
        printf("Provide array size:");
        scanf("%d", &N);
        if((p= (int *) malloc(N*sizeof(int))))==NULL) {
            printf("Allocation failed");
            exit(-1);
        }
        printf("Provide %d integers\n", N);
        for(i=0; i<N; i++)
            scanf("%d", p+i);

        //free(p);
    }
    return 0;
}
```

Each iteration of while loop allocates memory. But that memory is never freed. This causes memory leak.

Valgrind: a tool for memory leak detection

- We will use the valgrind tool to detect memory leaks
- Quick start info: <http://valgrind.org/docs/manual/quick-start.html>

Steps to follow:

1. Compile your C code using gcc
2. Then use valgrind to run the compiled code:
`valgrind --leak-check=full ./a.out`

Example: valgrind output

```
int main()
{
    int *pt = malloc(1*sizeof(int));
    scanf("%d", pt);
    printf("%d\n", *pt);
    return 0;
}
```

Output of valgrind --leak-check=full ./a.out

```
==1057== Memcheck, a memory error detector
==1057== Copyright (C) 2002-2017, and GNU GPL'd, by Julian Seward et al.
==1057== Using Valgrind-3.13.0 and LibVEX; rerun with -h for copyright info
==1057== Command: ./a.out
==1057==
```

... continued

```
==1057== HEAP SUMMARY: ←
==1057==     in use at exit: 4 bytes in 1 blocks
==1057==   total heap usage: 1 allocs, 0 frees, 4 bytes allocated
==1057==
==1057== LEAK SUMMARY: ←
==1057==     definitely lost: 0 bytes in 0 blocks
==1057==     indirectly lost: 0 bytes in 0 blocks
==1057==     possibly lost: 0 bytes in 0 blocks
==1057==     still reachable: 4 bytes in 1 blocks
==1057==           suppressed: 0 bytes in 0 blocks
```

Valgrind reports a memory leak of 4 bytes. 😞

Example: valgrind output

```
int main()
{
    int *pt = malloc(1*sizeof(int));
    scanf("%d", pt);
    printf("%d\n", *pt);
    free(pt); → Allocated memory is freed here
    return 0;
}
```

Output of valgrind --leak-check=full ./a.out

```
==2308== Memcheck, a memory error detector
==2308== Copyright (C) 2002-2017, and GNU GPL'd, by Julian Seward et al.
==2308== Using Valgrind-3.13.0 and LibVEX; rerun with -h for copyright info
==2308== Command: ./a.out
==2308==
5
5
==2308==
==2308== HEAP SUMMARY:
==2308==     in use at exit: 0 bytes in 0 blocks
==2308==     total heap usage: 1 allocs, 1 frees, 4 bytes allocated
==2308==
==2308== All heap blocks were freed -- no leaks are possible
==2308==
```

No memory leak reported! ☺

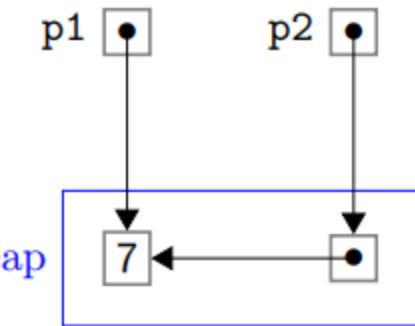
Consequences of memory leak

- Memory leaks slowdown system performance by reducing the amount of available memory.
- In modern operating systems, memory used by an application is released when the application terminates
 - If a program (with leak) runs for a short duration, then no serious problem
 - Will be a serious problem when a leaking program runs for long duration
- Memory leak will cause serious issues on resource-constrained embedded devices

Overall, your program should not contain memory leaks

Another memory leak example(1)

```
int main() {
    int *p1, **p2;
    p1 = malloc(sizeof(int));
    *p1=7;
    p2 = malloc(sizeof(int*));
    *p2=p1;
    return 0;
}
```

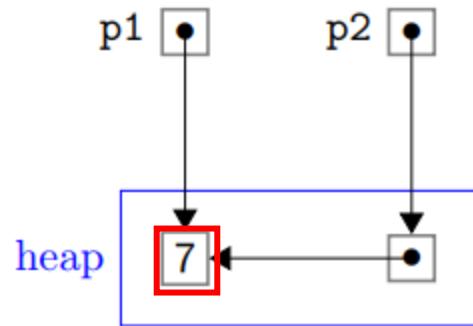


p2 is a pointer to a pointer variable

```
==3600== HEAP SUMMARY:
==3600==     in use at exit: 12 bytes in 2 blocks
==3600==   total heap usage: 2 allocs, 0 frees, 12 bytes allocated
==3600==
==3600== 12 (8 direct, 4 indirect) bytes in 1 blocks are definitely lost in loss record 2 of 2
==3600==    at 0x4C29BC3: malloc (vg_replace_malloc.c:299)
==3600==    by 0x400546: main (in /users/cosic/ssinharo/dynamicmem/a.out)
==3600==
==3600== LEAK SUMMARY:
==3600==    definitely lost: 8 bytes in 1 blocks
==3600==    indirectly lost: 4 bytes in 1 blocks
==3600==    possibly lost: 0 bytes in 0 blocks
==3600==    still reachable: 0 bytes in 0 blocks
==3600==          suppressed: 0 bytes in 0 blocks
==3600==
```

Another memory leak example(2)

```
int main() {
    int *p1, **p2;
    p1 = malloc(sizeof(int));
    *p1=7;
    p2 = malloc(sizeof(int*));
    *p2=p1;
    free(p1);
    return 0;
}
```



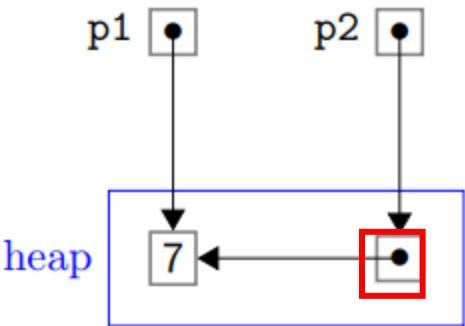
p2 is a pointer to a pointer variable

```
==3847== LEAK SUMMARY:
==3847==     definitely lost: 8 bytes in 1 blocks
==3847==     indirectly lost: 0 bytes in 0 blocks
==3847==     possibly lost: 0 bytes in 0 blocks
==3847==     still reachable: 0 bytes in 0 blocks
==3847==           suppressed: 0 bytes in 0 blocks
==3847==
```

Still 8 bytes are leaked!

Another memory leak example(3)

```
int main() {
    int *p1, **p2;
    p1 = malloc(sizeof(int));
    *p1=7;
    p2 = malloc(sizeof(int*));
    *p2=p1;
    free(p2);
    return 0;
}
```



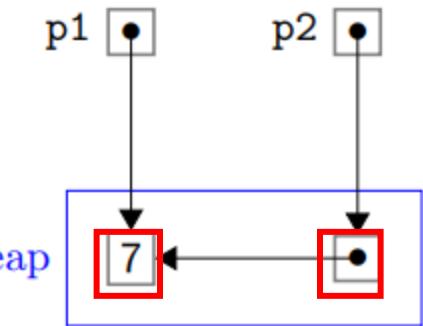
p2 is a pointer to a pointer variable

```
==6994== LEAK SUMMARY:
==6994==    definitely lost: 4 bytes in 1 blocks
==6994==    indirectly lost: 0 bytes in 0 blocks
==6994==    possibly lost: 0 bytes in 0 blocks
==6994==    still reachable: 0 bytes in 0 blocks
==6994==          suppressed: 0 bytes in 0 blocks
```

4 bytes are leaked!

Another memory leak example(4)

```
int main() {  
    int *p1, **p2;  
    p1 = malloc(sizeof(int));  
    *p1=7;  
    p2 = malloc(sizeof(int*));  
    *p2=p1;  
    free(p1);  
    free(p2);  
    return 0;  
}
```



p2 is a pointer to a pointer variable

No memory leak!

Accessing memory after free()

- After `free(p)`, the memory is no longer owned by the program.



➤ Some other program may be using the memory! **Data is lost**

- Re-accessing the memory pointed by p causes **undefined behaviour**

```
int main() {
    int sum, *p;
    p = (int *) malloc(4*sizeof(int));
    // [some code here]
    free(p);
    sum = sum + *p; //Use after free issue
    // [some code here]
}
```

Memory pointed by p was freed and then used.
Program will cause undefined behaviour.
Valgrind reports memory error.

Don't free something that did not come from malloc()

```
foo(int a) {  
    int b;  
    int c []={1,2,3};  
    free(&a);  
    free(&b);  
    free(c);  
}
```

- Only, dynamically created objects are stored in heap
- gcc gives warnings: “attempt to free a non-heap object”
- However, program crashes with segmentation fault

Valgrind reports memory errors

Double free problem

- Double free errors occur when `free()` is called more than once with the same memory address as an argument.

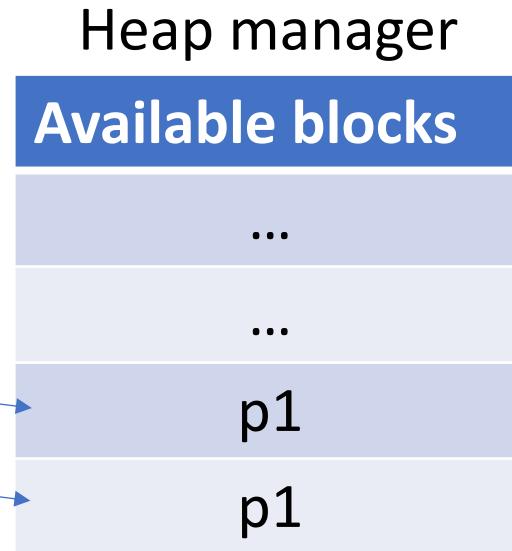
```
int main () {
    char *p1=malloc (1);
    char *p2=malloc (1);
    free (p1);
    free (p1);
}
```

- When `free()` is called twice with the same argument, the program's memory management data structures become corrupted
- Consequences: Program malfunction including security vulnerabilities

At runtime, modern OS detects double-free operation.
Program is aborted.

How to think about double free

```
int main() {  
    char *p1=malloc(1);  
    char *p2=malloc(1);  
    free(p1);  
    free(p1);  
}
```



- Every time `free()` is called, the address is added in the list of available memory blocks
- The last address added to the list can be used by the next `malloc()` call
- Since p1 is present twice in the list, next two memory allocations will have the same address (**which is a problem**)
- Valgrind reports memory error

Application of memory management in C: Implementation of Linked List

Mohammed Bahja

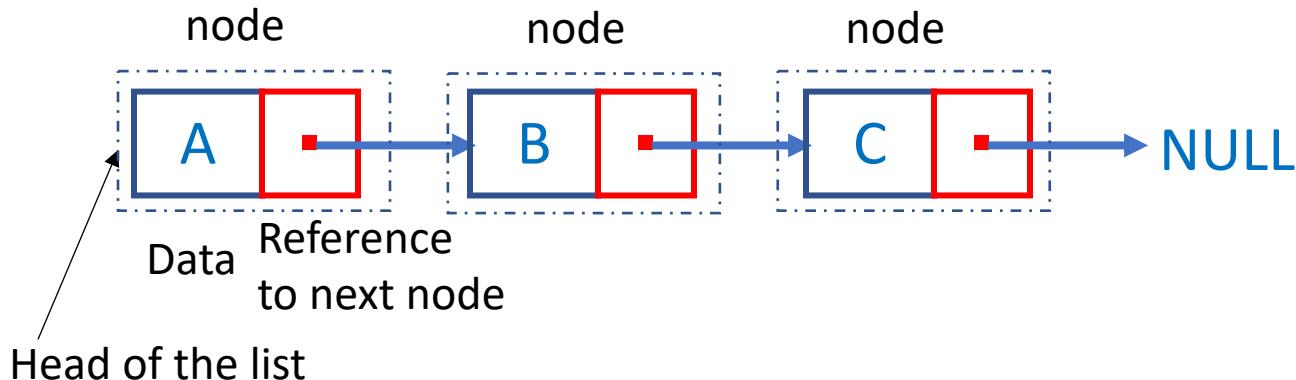
School of Computer Science

University of Birmingham

Linked List (Recap from Data Structure module)

A ‘linked list’ is a

- linear collection of data elements called ‘nodes’
- each node points to the next node in the list
- unlike arrays, linked list nodes are not stored at contiguous locations; they are linked using pointers as shown below.



Implementation of singly Linked List in C

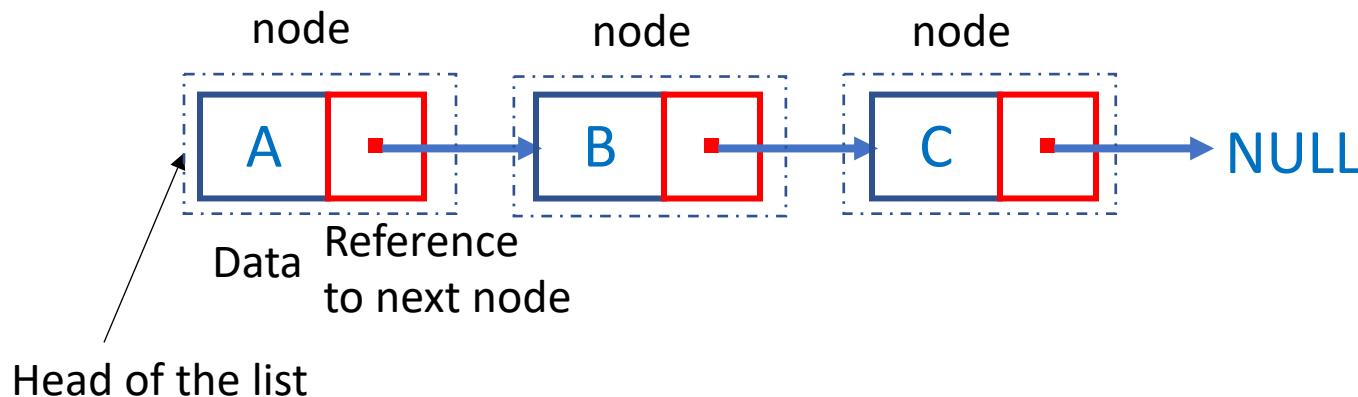
Our goal is to implement the following operations:

1. Append at the end of list
2. Search an element in the list
3. Print entire list from the start
4. Free memory occupied by a list

Defining ‘Node’ of a list in C

A ‘node’ is a composite data-type which consists of

- a data
- and a reference to the next node



How to define a composite data-type in C?

Example of Java class

```
class LinkedList {  
    Node head; // head of list  
  
    /* Linked list Node */  
    class Node {  
        int data;  
        Node next;  
  
        // Constructor to create a new node  
        // Next is by default initialized  
        // as null  
        Node(int d) { data = d; }  
    }  
}
```

A Java class has multiple members and member functions

Creating composite data types in C

- C programming language does not have ‘class’
- Note: C++ supports OOP
- In C language, we can create user-defined composite data types as **structures**.

Structures in C

- A structure is a user defined composite data type in C
- A structure is used to group items of possibly different types into a single type
- Unlike Java/C++ classes, structures do not have member functions.
- Syntax is

```
struct tag_name {  
    T1 member1;  
    T2 member2;  
    /* declare as many members as desired,  
       but the entire structure size must be  
       known to the compiler. */  
};
```

Example1: Points with x and y coordinates

- A point has two coordinates: ‘x’ and ‘y’
- In C we can create a new data-type ‘Point’ as

```
struct Point  
{  
    int x, y;  
};
```

- Objects of type ‘Point’ can be created as

```
int main()  
{  
    struct Point p1; // p1 is an object of type Point  
}
```

Example1: Points with x and y coordinates

- There is a shortcut for ‘`struct Point p1`’

```
typedef struct Point  
{  
    int x, y;  
} Point;
```

add `typedef` before `struct`

- Now, objects of type ‘Point’ can be created as

```
int main()  
{  
    Point p1; // p1 is an object of type Point  
}
```

Accessing the members of a structure

- Structure members are accessed using dot (.) operator.

```
Point p1;  
// For p1(2,3)  
p1.x = 2;  
p1.y = 3;
```

Pointer to a structure object (1)

- We can create pointers to point to structure objects.
- If we have a pointer to structure, members are accessed using arrow (->) operator.
- Example: pointer to the point objects

```
Point p1;  
// For p1(2,3)  
p1.x = 2;  
p1.y = 3;  
printf("%d %d", p1.x, p1.y); // (.) used to access members
```

```
Point *p2;  
p2 = &p1; // p2 points to p1  
printf("%d %d", p2->x, p2->y); // (->) used to access members
```

Pointer to a structure object (2)

- We can dynamically allocate memory for structure objects
- Example:

```
Point *p;  
p = (Point *) malloc(sizeof(point));  
  
// Now assign coordinates (5,6) to *p  
p->x = 5;  
p->y = 6;
```

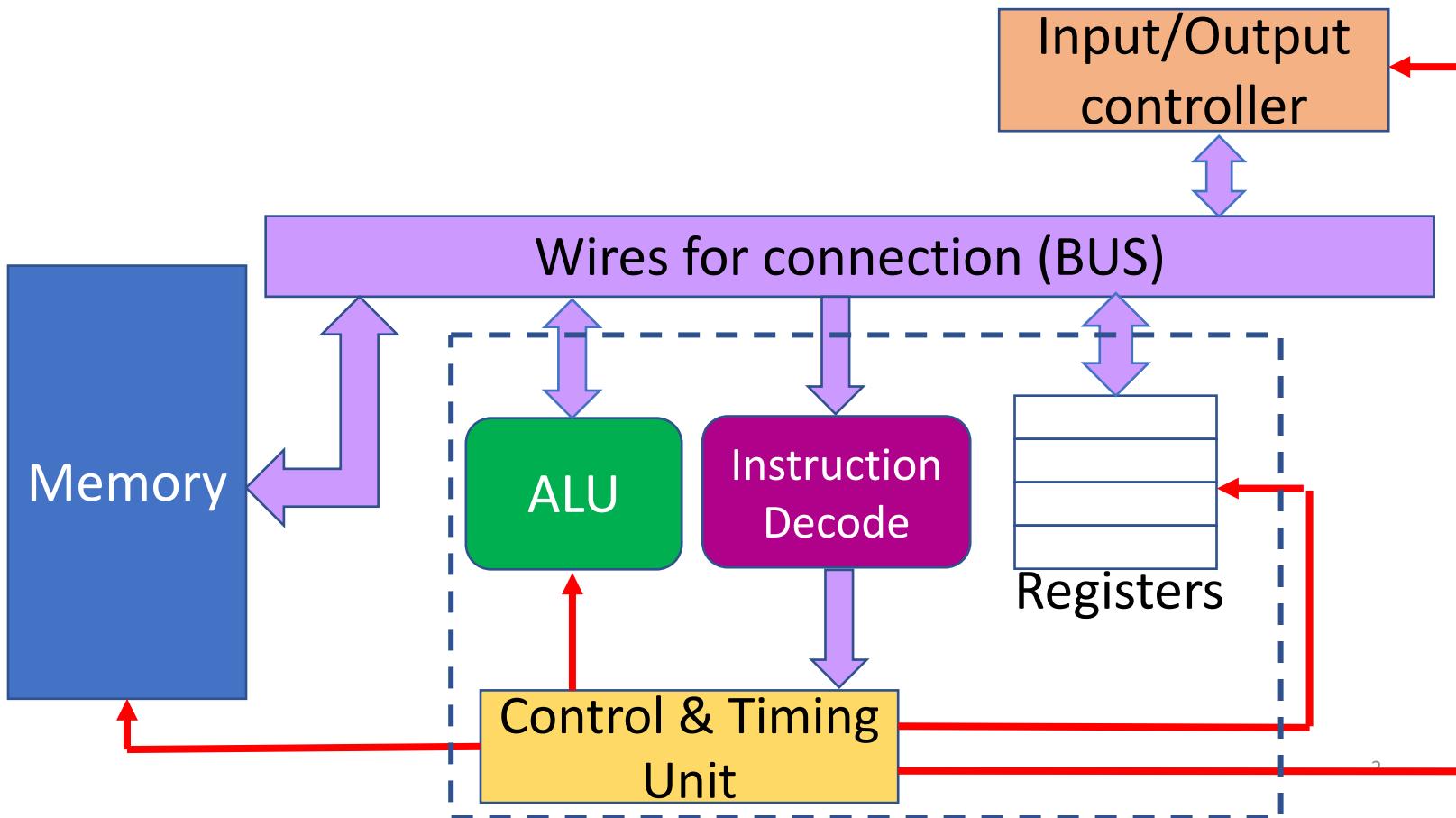
Computer Architecture: Memory Hierarchy

Mohammed Bahja

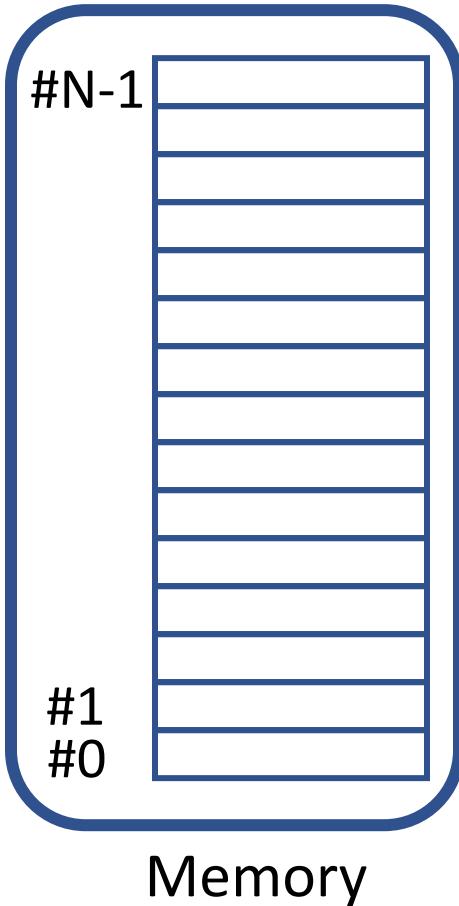
School of Computer Science

University of Birmingham

Recap: Computer organization



Memory: Programmer's perspective

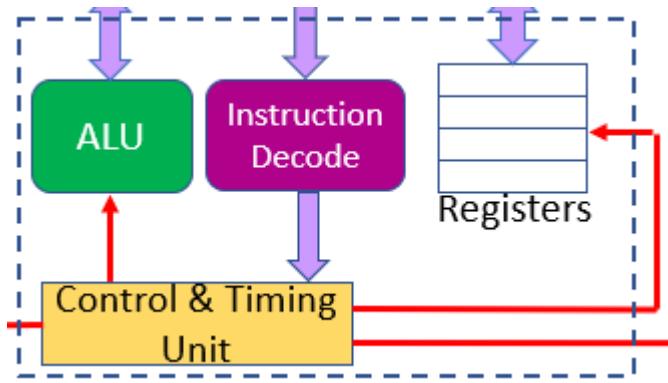


This shows a simplified functional view of Memory.

Our computers have more complex memory system!

- Memory consists of small 'locations'
- Each location can store a small information

Different types of memory



Registers inside a Processor



DRAM (we call it 'RAM')



Solid State Disk



Hard Disk

Different types of memory

Memory Tech.	Capacity	Data speed (time)	Cost/GB
Registers	~1000s bits	10 ps	£fffffff
SRAM	~10 KB-10 MB	1-10 ns	~£1000
DRAM	~10 GB	100 ns	~£10
SSD	~100 GB	100 us	~£1
Hard Disk	~1 TB	10 ms	~£0.10

Different memory technologies have different **tradeoffs**.

- Fast memory elements have small storage capacity.
- Large memory elements have slow data rate.

Given these **tradeoffs**:

- Fast memory elements have small storage capacity.
- Large memory elements have slow data rate.

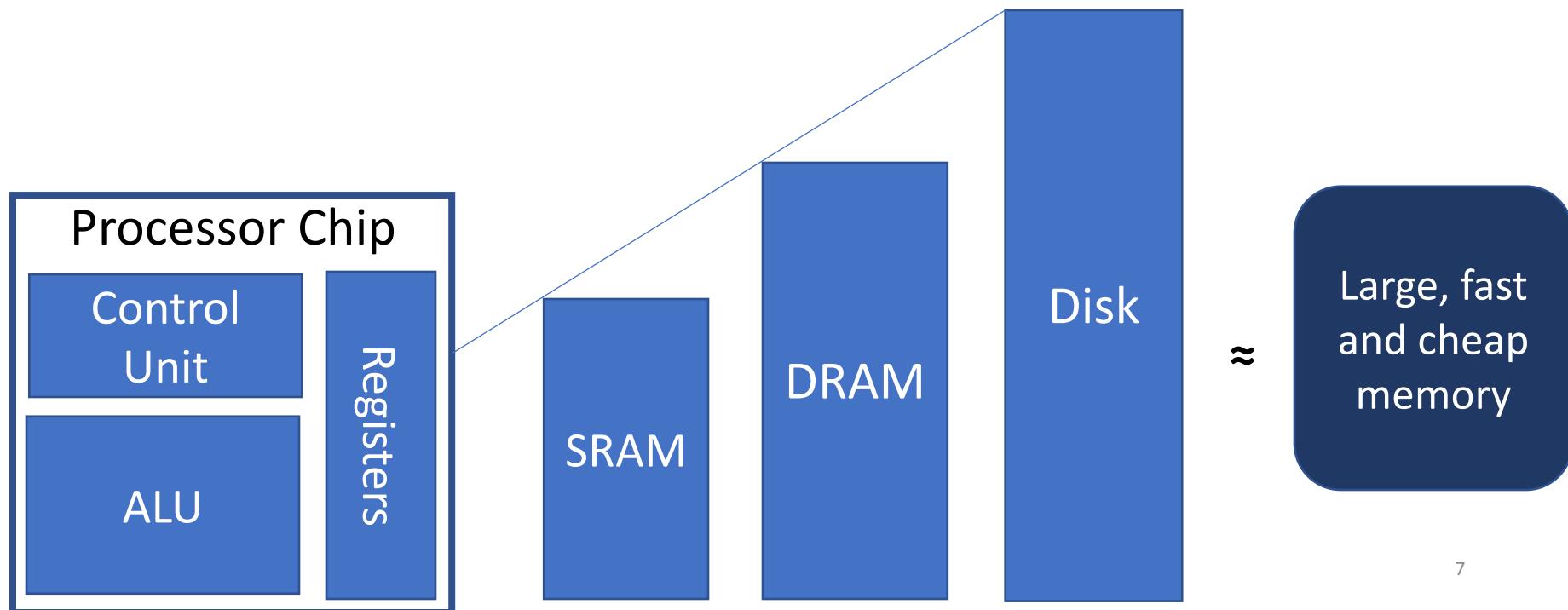
Can we have a computer which has '*large, fast and cheap*' memory?

Given these **tradeoffs**:

- Fast memory elements have small storage capacity.
- Large memory elements have slow data rate.

Can we have a computer which has '*large, fast and cheap*' memory?

Idea: use hierarchy of memory elements to emulate a '*large, fast and cheap*' memory.



The next cartoon shows how to think of memory hierarchy.

I have study leave
before exam.
At home, I want to
prepare computer
architecture.





I have study leave
before exam.
At home, I want to
prepare computer
architecture.

Let's collect the
architecture book
by Patterson and
Hennessy from
our Library.

Library (large storage of books)



Computer architecture books

There are several books on computer architecture. So, why not collect a few more of them? May be later, I find some of them useful!

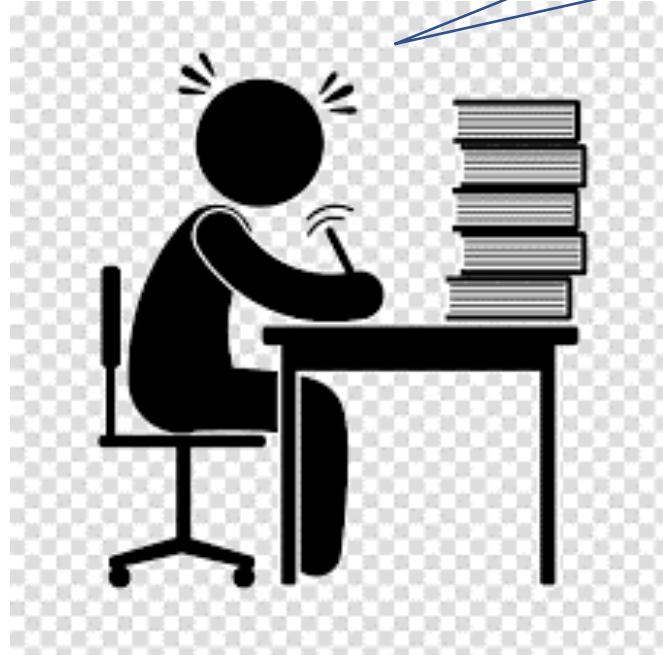


home sweet home



Luckily, I had brought these other books on Comp Architecture. Otherwise I would have required to visit the library again!

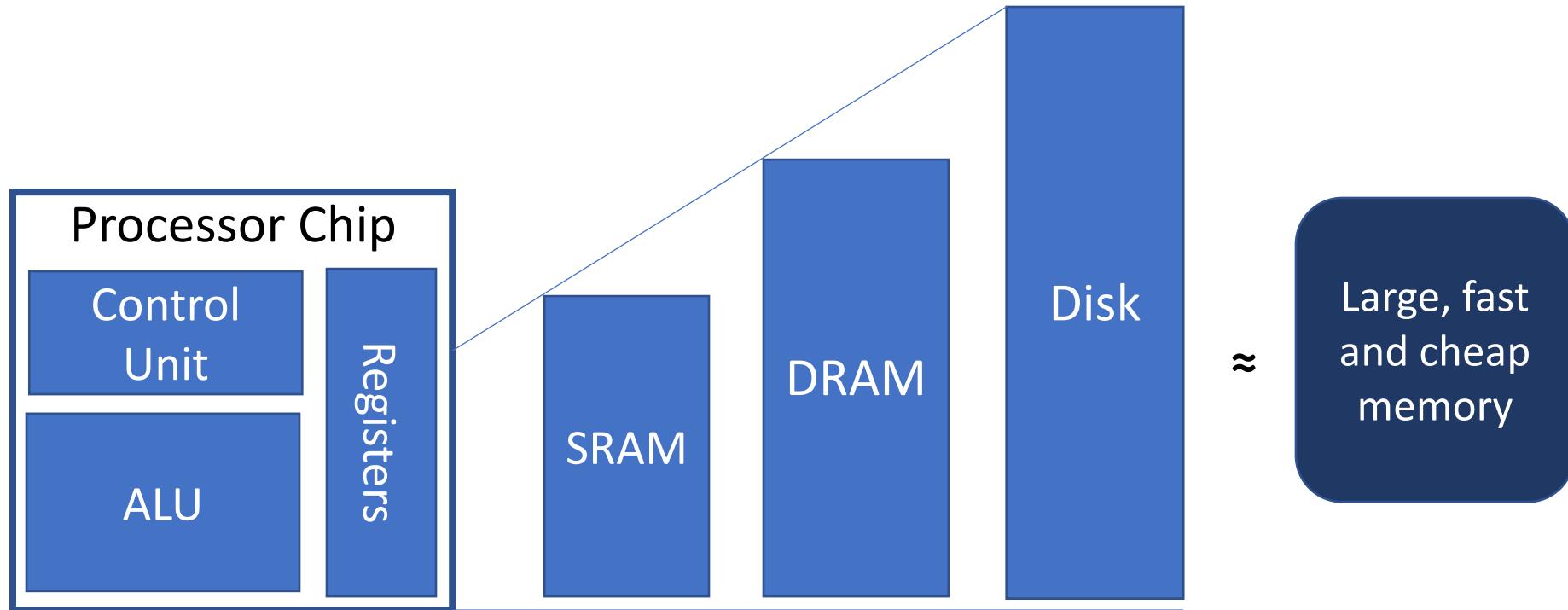
home sweet home



Luckily, I had brought these other books on Comp Architecture. Otherwise I would have required to visit the library again!

Idea: Keep the book that you wanted to study as well as a few extra books that might be useful, on your study desk.

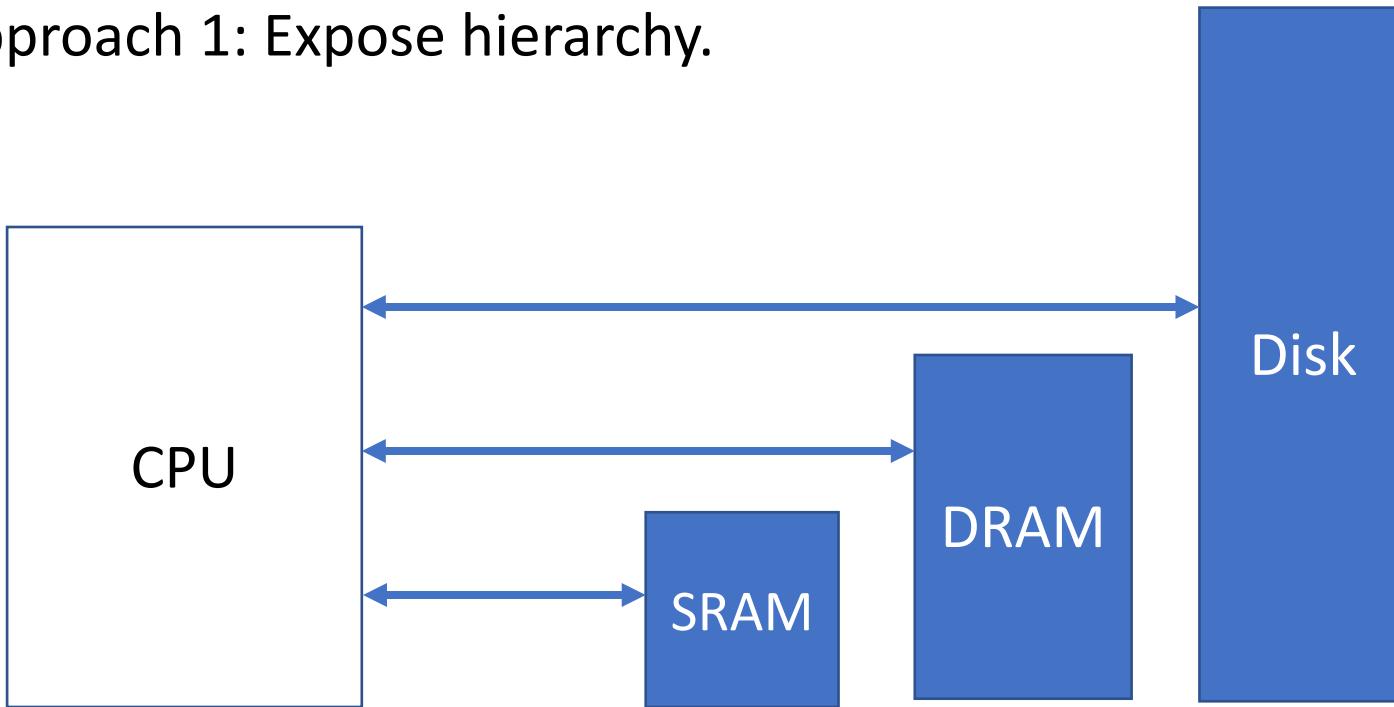
The same idea can be applied to a computer!



Idea: Keep the piece of data that the processor requires now, as well as some extra data that might be useful next, close to the processor in the fast memory.

How to interface memory hierarchy?

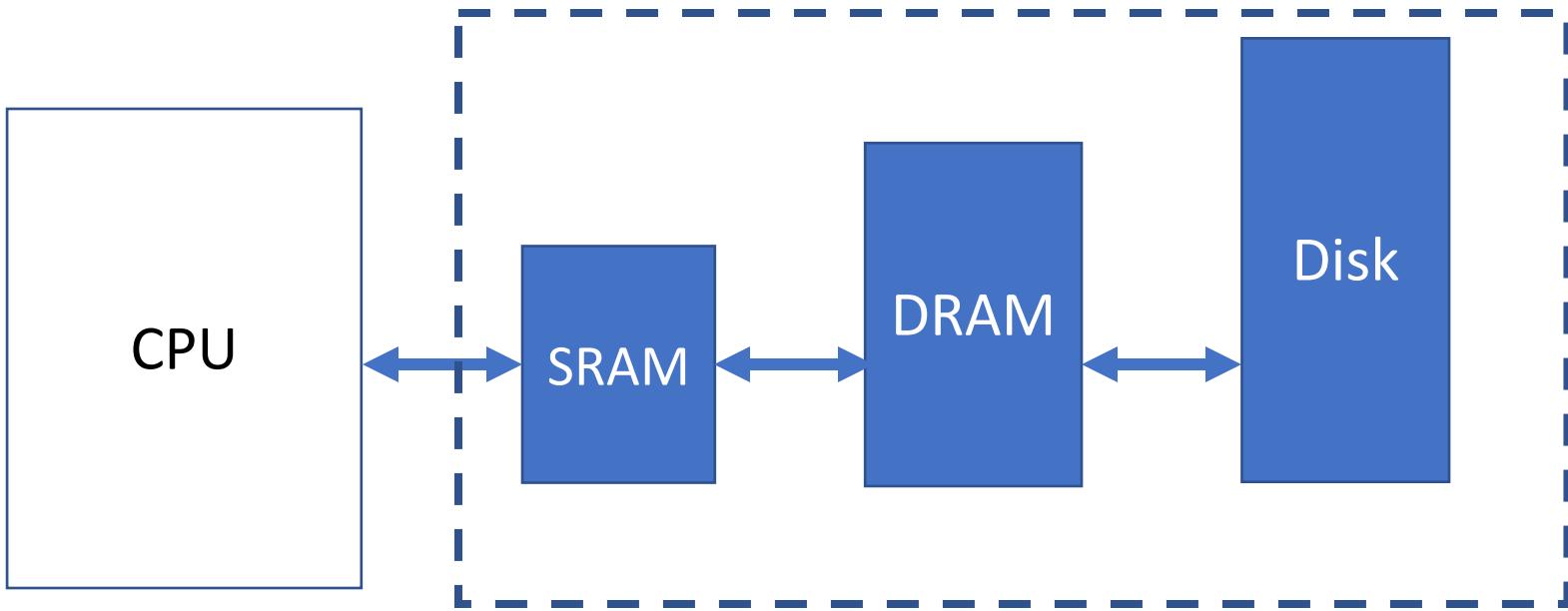
Approach 1: Expose hierarchy.



- CPU can access any level in the memory hierarchy directly
- Programmer should know the 'details' of the hierarchy and should write code 'cleverly'.

How to interface memory hierarchy?

Approach 2: Hide hierarchy.



- Programmer's perspective: a single memory with single address space.
- Hardware takes the responsibility of storing data in fast or slow memory, depending on usage patterns.
- 'Clever programmer' writes code in such a way that the CPU gets its data from fast memory with high probability. [will learn this]

How does a computer decide which data to keep in fast memory and which data to keep in slow memory?

Answer: computer uses locality of reference

The locality of reference

- **Locality of reference**, also known as the principle of locality, is the tendency of a processor to access the same set of memory locations repetitively over a short period of time.
- Two kinds of locality: temporal and spatial locality.
- Temporal locality refers to the reuse of specific data, and/or resources, within a relatively small time duration.
- Spatial locality refers to the use of data elements within relatively close storage locations.

The locality of reference: example

Example of computing the sum of an array

```
// Compute sum of an int array
int a[N] = {2, 5, 3, 7, ...};
int sum=0;

for(i=0; i<N; i++)
    sum = sum + a[i];
```

Do you see any locality in the program?

The locality of reference: example

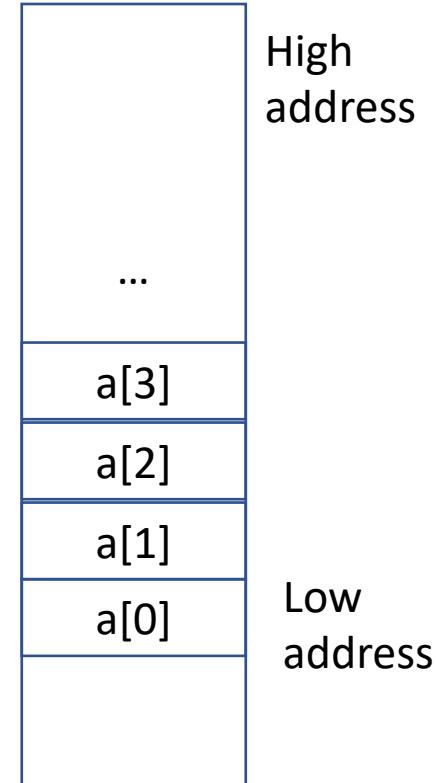
Example of computing the sum of an array

```
// Compute sum of an int array
int a[N] = {2, 5, 3, 7, ...};
int sum=0;

for(i=0; i<N; i++)
    sum = sum + a[i];
```

Locality of reference in the above program:

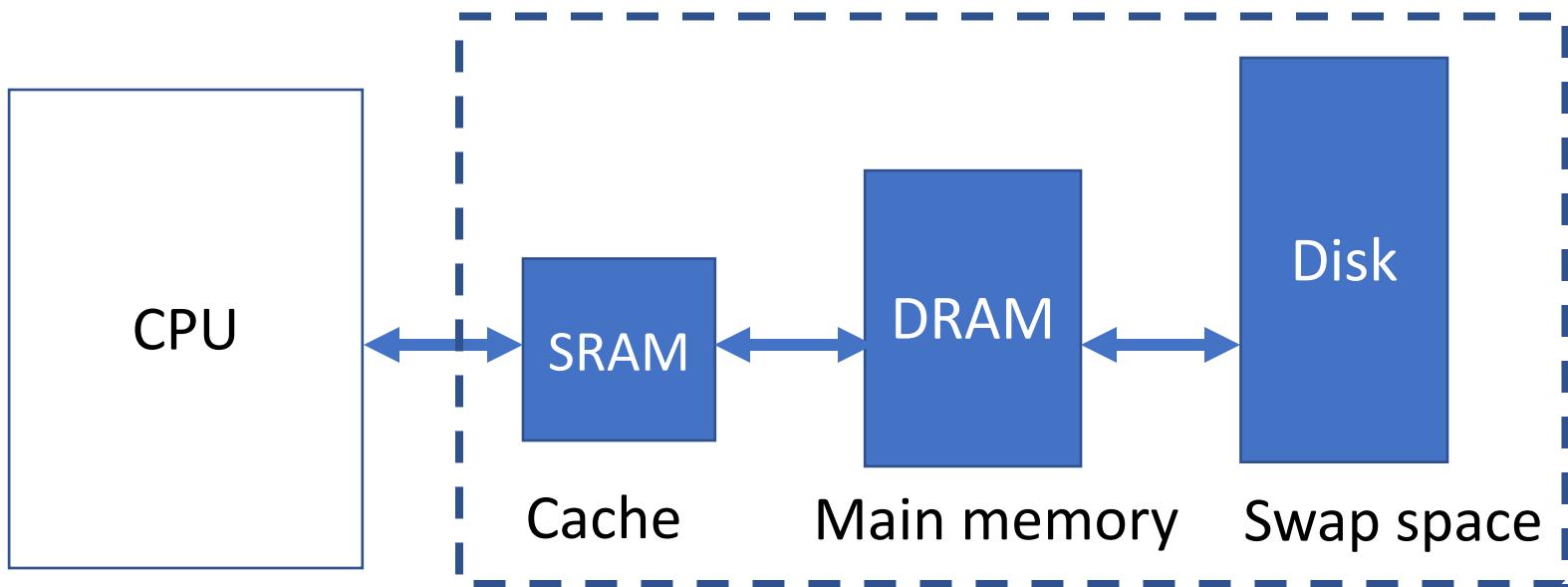
- The object 'sum' satisfies temporal locality.
It is used again and again.
- Array elements satisfy spatial locality.
→ If $a[i]$ is used, then use of $a[i+1], a[i-1]$ etc. is highly probable.



Computer tries to increase locality of reference

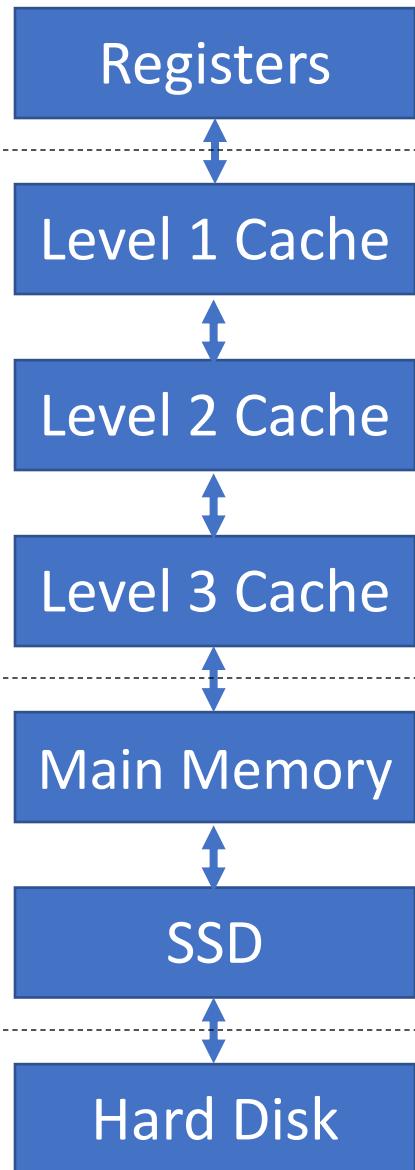
Computer tries to:

- Keep the most-frequently used data in a fast (but small) SRAM. This SRAM is called 'cache' as it transparently retains (caches) data from recently accessed memory locations in DRAM.
- Refer to large (but slower) DRAM for data which is not present in cache. This DRAM is called 'main memory'
- Swap space is used only when data cannot be fit in main memory.



A typical memory hierarchy

Attached to ALU



Different levels of caches are present inside the processor chip.

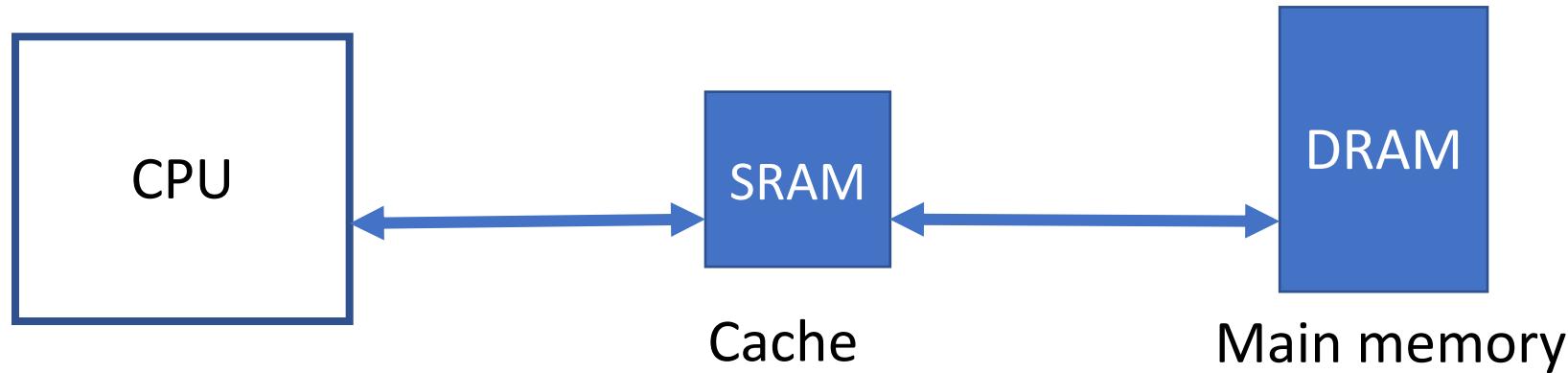
Present outside the processor chip.

Slow mechanical device

Access time	Capacity
1 cycle	1000s of bits
4 cycles	32 KB
10 cycles	256 KB
40 cycles	10 MB
200 cycles	8-16 GB
10-100 us	256 GB
10 ms	1 TB

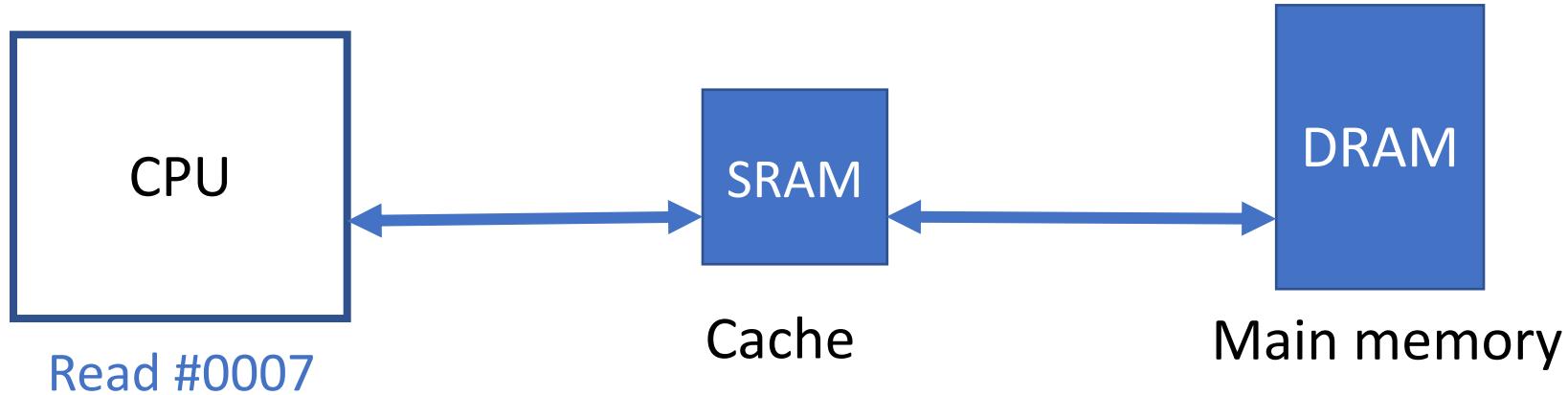
Cache access

Example: (Simplified) Computer with only Level 1 Cache and Main memory



Cache access

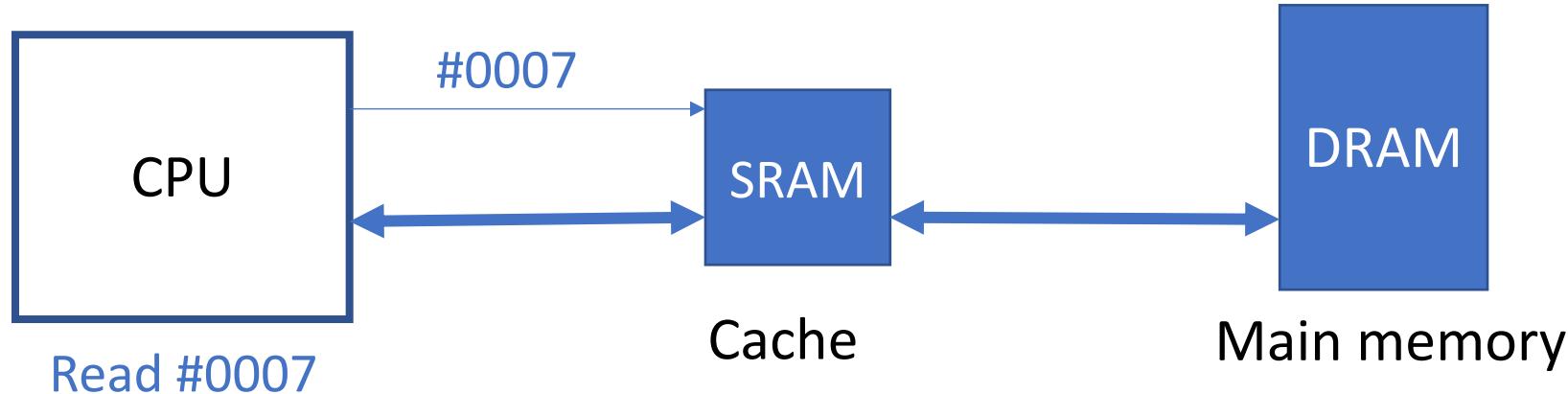
Example: (Simplified) Computer with only Level 1 Cache and Main memory



1. Suppose CPU wants to read from address #0007

Cache access

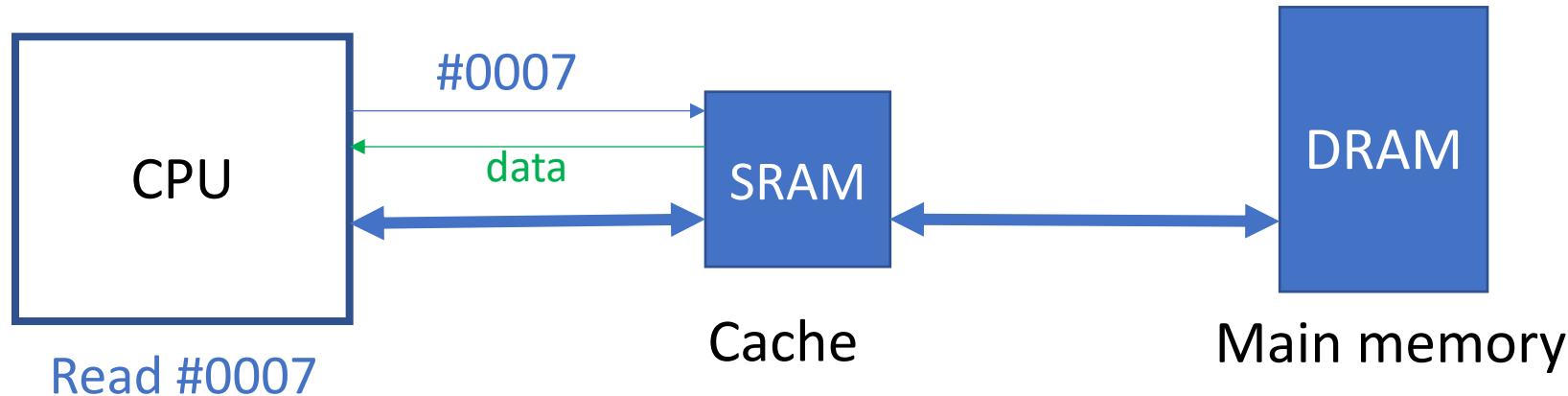
Example: (Simplified) Computer with only Level 1 Cache and Main memory



1. Suppose CPU wants to read from address #0007
2. Processor sends the address to Cache.

Cache access

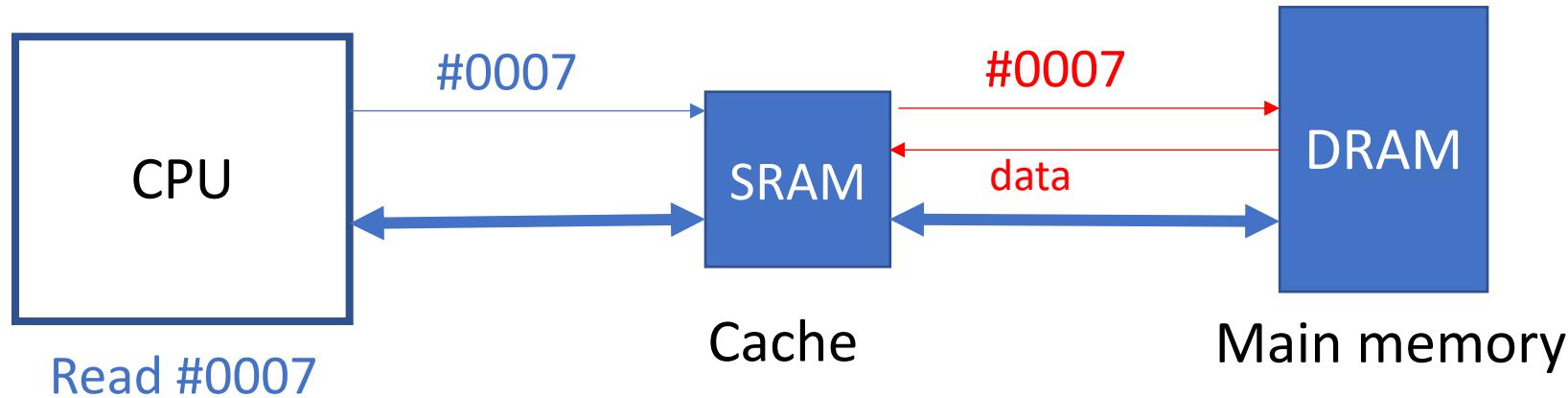
Example: (Simplified) Computer with only Level 1 Cache and Main memory



1. Suppose CPU wants to read from address #0007
2. Processor sends the address to Cache.
3. Two situations can happen.
 - **Cache hit:** The required data is present in Cache.
So, the data is returned quickly from Cache.

Cache access

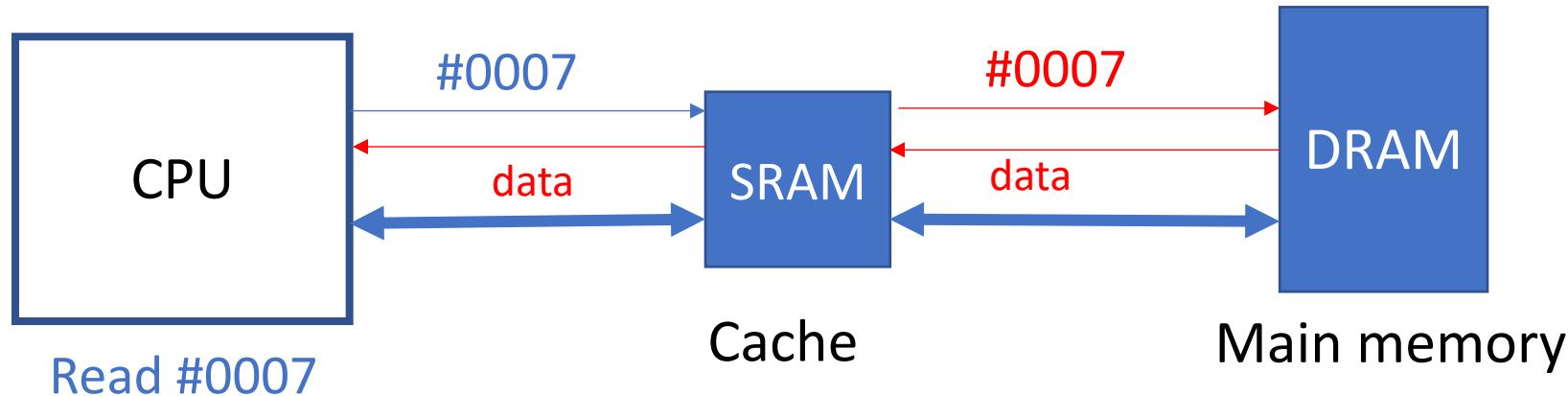
Example: (Simplified) Computer with only Level 1 Cache and Main memory



1. Suppose CPU wants to read from address #0007
2. Processor sends the address to Cache.
3. Two situations can happen.
 - **Cache hit:** The required data is present in Cache.
So, the data is returned quickly from Cache.
 - **Cache miss:** The data is not in cache. So, get it from Main Memory and bring it to Cache

Cache access

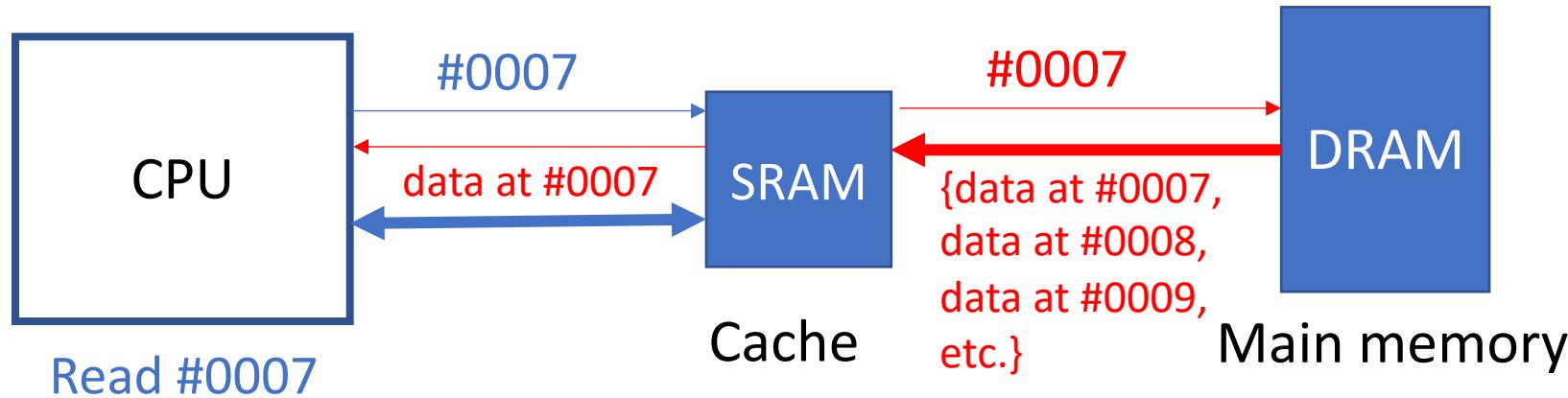
Example: (Simplified) Computer with only Level 1 Cache and Main memory



1. Suppose CPU wants to read from address #0007
2. Processor sends the address to Cache.
3. Two situations can happen.
 - **Cache hit:** The required data is present in Cache.
So, the data is returned quickly from Cache.
 - **Cache miss:** The data is not in cache. So, get it from Main Memory and bring it to Cache and finally provide it to CPU.
→ There is a performance penalty ☹

Cache access: Spatial locality

Example: (Simplified) Computer with only Level 1 Cache and Main memory



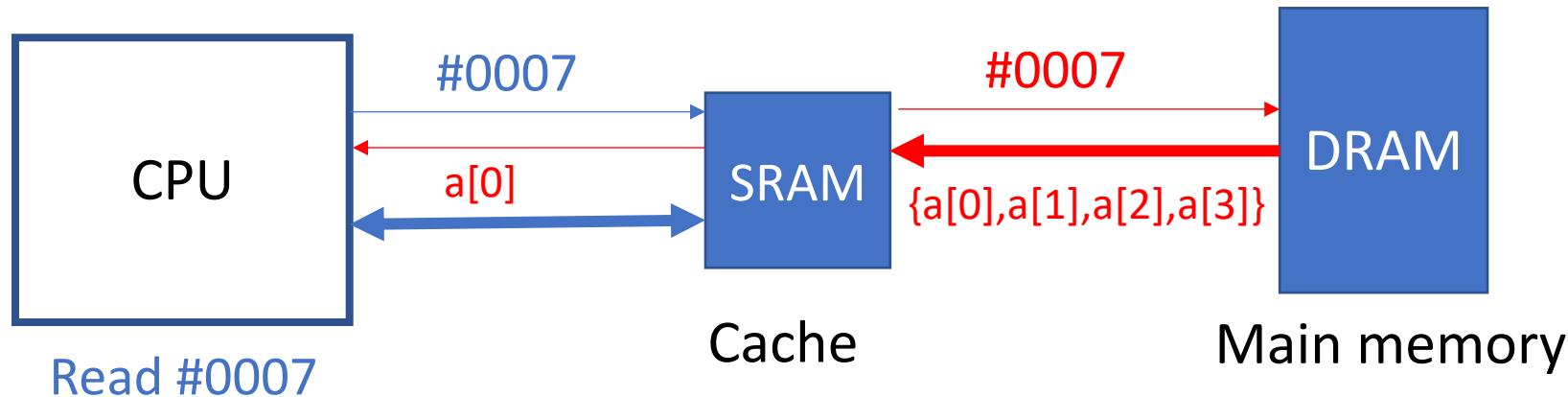
When the required piece of data is loaded from Main Memory to Cache, other nearby data blocks are also copied into the Cache.

Example: The data requested by CPU is at #0007 in Main memory. When it is copied into Cache, data blocks from #0008, #0009, etc. are also copied into Cache.

This increases the chances of ‘Cache Hit’ in the future.

Cache access: Spatial locality: Example

Example: (Simplified) Computer with only Level 1 Cache and Main memory



```
// Compute sum of an int array
int a[N] = {2, 5, 3, 7, ...};
int sum=0;

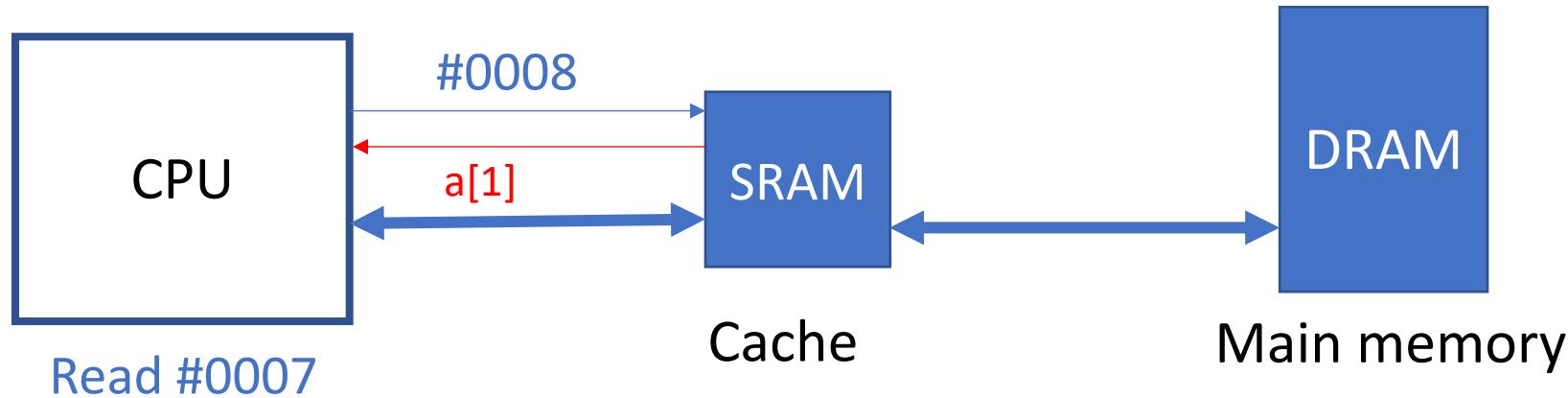
for(i=0; i<N; i++)
    sum = sum + a[i];
```

Assume the array starts from #0007. Initially all data is in DRAM.

CPU wants a[0]: Data is initially not in Cache. So, a[0] along with a[1], a[2], a[3] are fetched from Main memory and brought to Cache.

Cache access: Spatial locality : Example

Example: (Simplified) Computer with only Level 1 Cache and Main memory



```
// Compute sum of an int array
int a[N] = {2, 5, 3, 7, ...};
int sum=0;

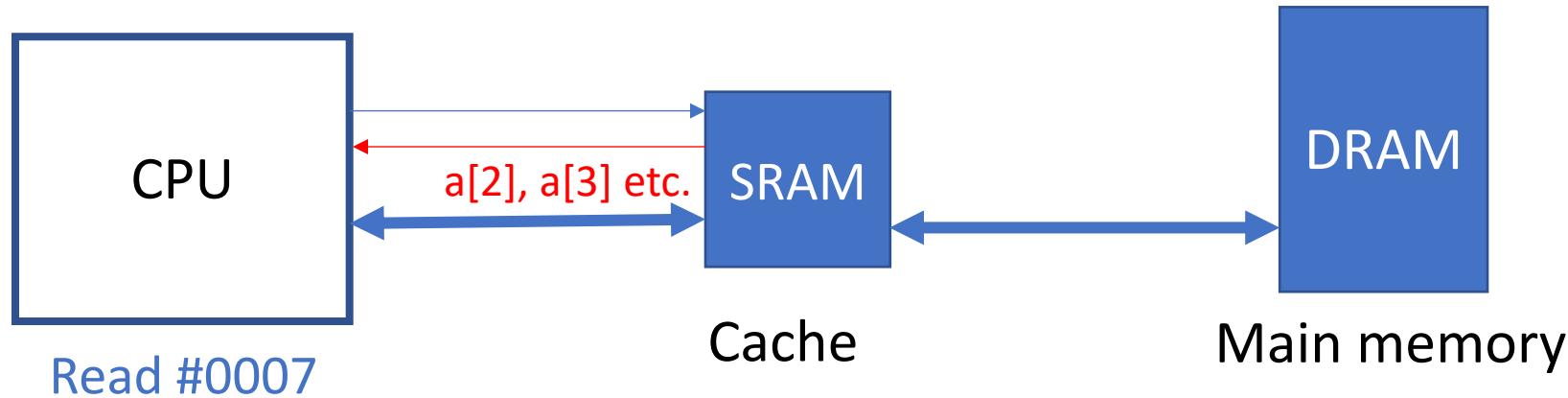
for(i=0; i<N; i++)
    sum = sum + a[i];
```

Assume the array starts from #0007.

CPU wants a[1]: Since a[1] is in Cache, there is a Cache Hit.
Hence, a[1] is provided to CPU quickly.

Cache access: Spatial locality: Example

Example: (Simplified) Computer with only Level 1 Cache and Main memory



```
// Compute sum of an int array
int a[N] = {2, 5, 3, 7, ...};
int sum=0;

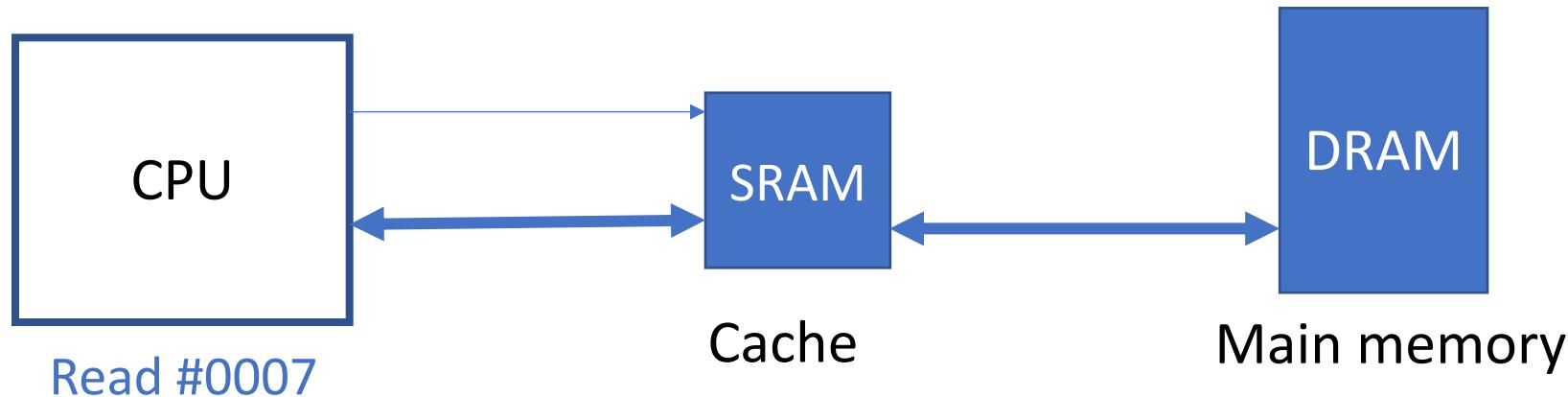
for(i=0; i<N; i++)
    sum = sum + a[i];
```

Assume the array starts from #0007.

Similarly, a[2] and a[3] are provided to the CPU from Cache.

Cache access: Spatial locality: Example

Example: (Simplified) Computer with only Level 1 Cache and Main memory



```
// Compute sum of an int array
int a[N] = {2, 5, 3, 7, ...};
int sum=0;

for(i=0; i<N; i++)
    sum = sum + a[i];
```

Assume the array starts from #0007.

If there is not enough space left in Cache, then some old data is deleted from Cache to create space for new data.

Locality example

Both functions compute the sum of the elements of an input 2D matrix.
Which one has better locality?

```
int sum_2d_array1(int a[N][M]){
    int i, j, sum=0;
    for(i=0; i<N; i++)
        for(j=0; j<M; j++)
            sum = sum + a[i][j];
    return sum;
}
```

Computes the sum
over a row.
 $a[0][0]+a[0][1]+\dots$
 $+a[1][0]+a[1][1]+\dots$

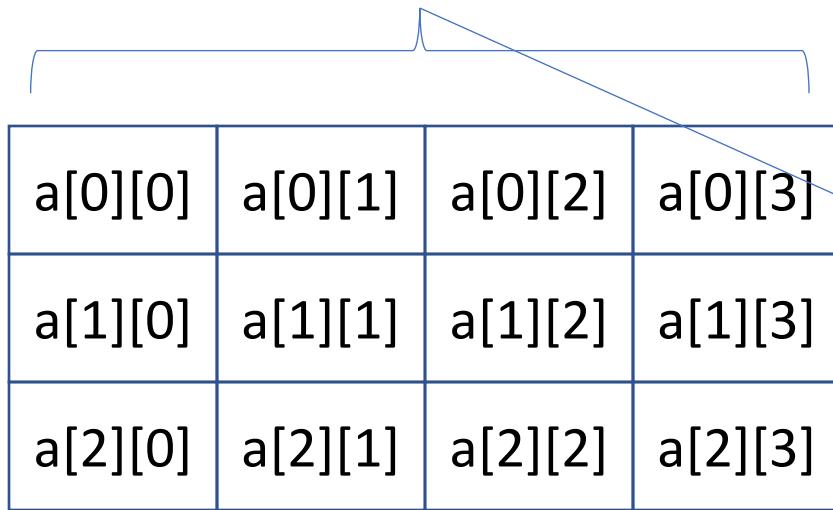
```
int sum_2d_array2(int a[N][M]){
    int i, j, sum=0;
    for(i=0; i<M; i++)
        for(j=0; j<N; j++)
            sum = sum + a[j][i];
    return sum;
}
```

Computes the sum
over a column.
 $a[0][0]+a[1][0]+\dots$
 $+a[0][1]+a[1][1]+\dots$

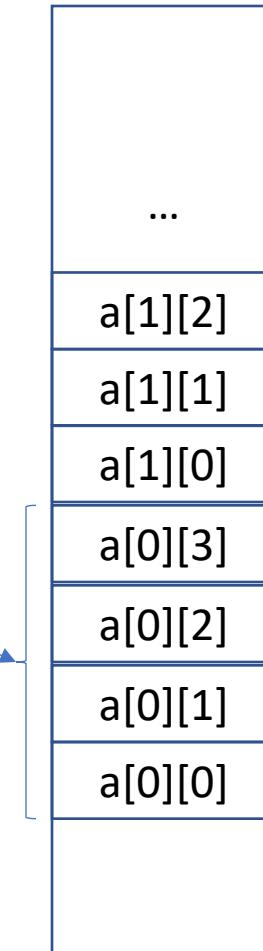
Recap: Memory layout of two-dimensional array

C compiler stores 2D array in **row-major** order

- All elements of Row #0 are stored
- then all elements of Row #1 are stored
- and so on



Logical view of array $a[3][4]$



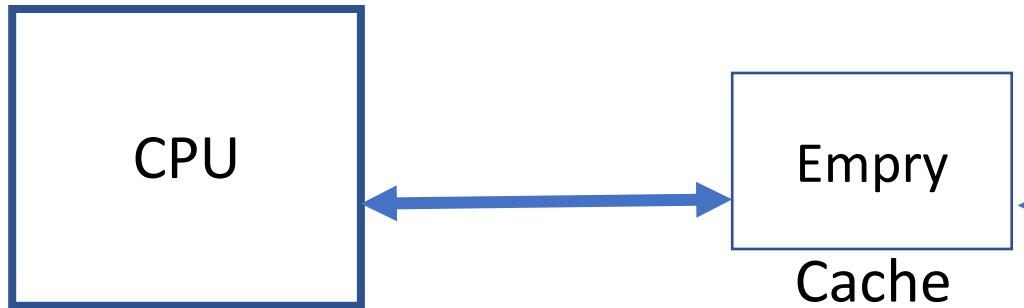
Memory layout

Locality example: the first case

```
int sum_2d_array1(int a[N][M]){
    int i, j, sum=0;
    for(i=0; i<N; i++)
        for(j=0; j<M; j++)
            sum = sum + a[i][j];
    return sum;
}
```

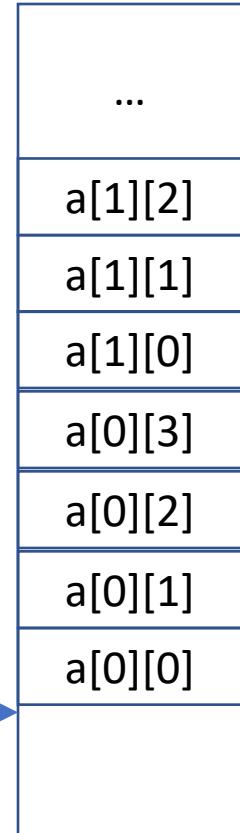
Computes the sum over a row.

$$a[0][0] + a[0][1] + \dots \\ + a[1][0] + a[1][1] + \dots$$



1. CPU requires $a[0][0]$ to compute $\text{sum} = \text{sum} + a[0][0]$

Array elements are initially in the Main Memory.



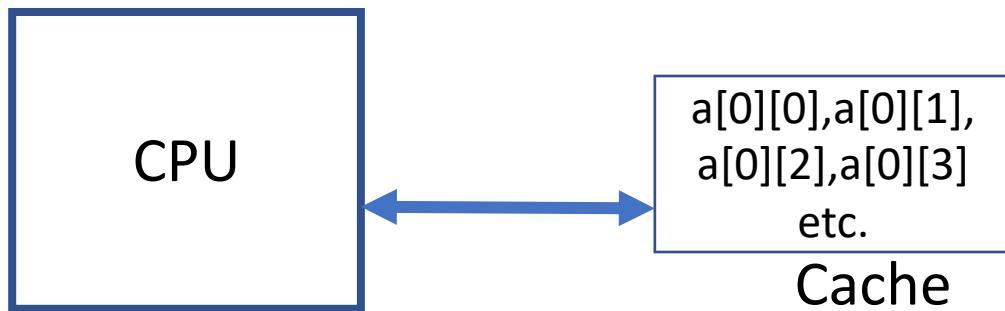
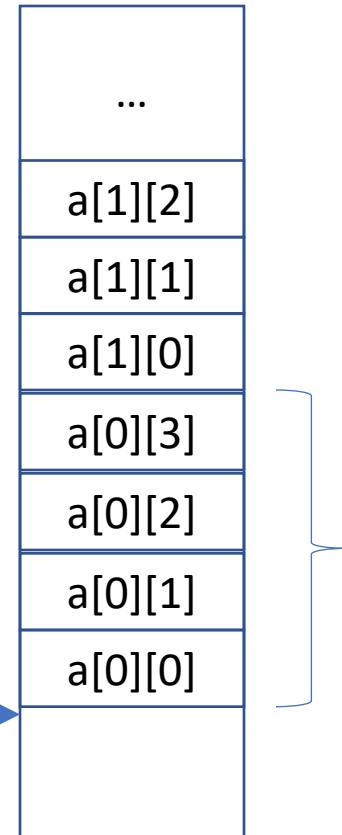
Locality example: the first case

```
int sum_2d_array1(int a[N][M]){
    int i, j, sum=0;
    for(i=0; i<N; i++)
        for(j=0; j<M; j++)
            sum = sum + a[i][j];
    return sum;
}
```

Computes the sum over a row.

$$a[0][0]+a[0][1]+\dots \\ +a[1][0]+a[1][1]+\dots$$

Array elements are initially in the Main Memory.



1. CPU requires a[0][0] to compute $\text{sum} = \text{sum} + a[0][0]$
2. Due to 'spatial locality', say a[0][0], a[0][1], a[0][2] etc. are loaded from Main Memory to Cache. [100 cycles are spent to access Main Memory]

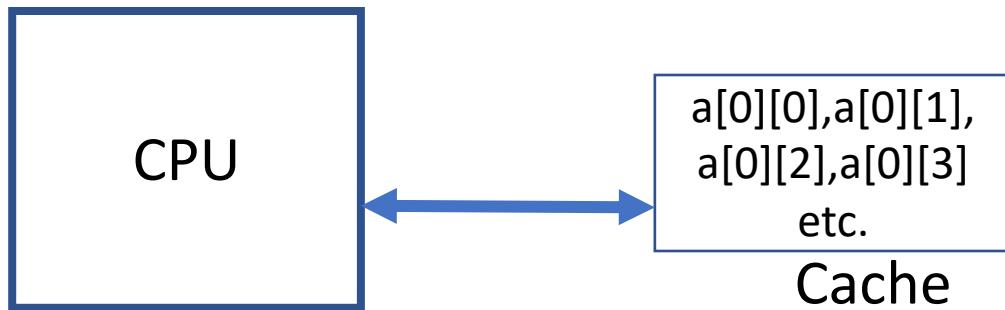
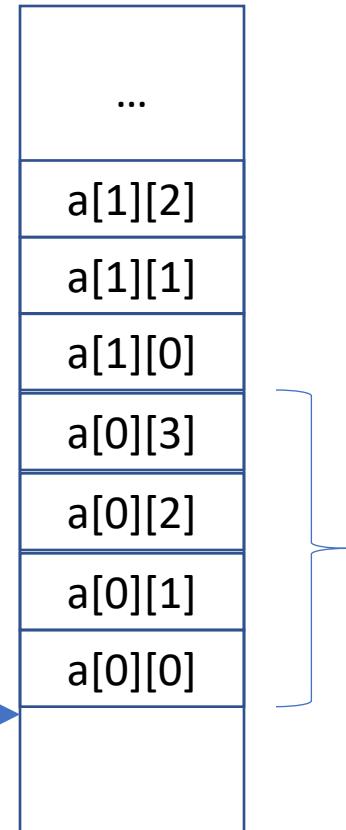
Locality example: the first case

```
int sum_2d_array1(int a[N][M]){
    int i, j, sum=0;
    for(i=0; i<N; i++)
        for(j=0; j<M; j++)
            sum = sum + a[i][j];
    return sum;
}
```

Computes the sum over a row.

$$\begin{aligned} &a[0][0]+a[0][1]+\dots \\ &+a[1][0]+a[1][1]+\dots \end{aligned}$$

Array elements are initially in the Main Memory.



3. CPU computes $\text{sum}=a[0][0]+a[0][1]+a[0][2] \dots$ by reading the elements from Cache. [Each access takes 4 cycles]
4. Advantage: Cache hit happens most of the times.

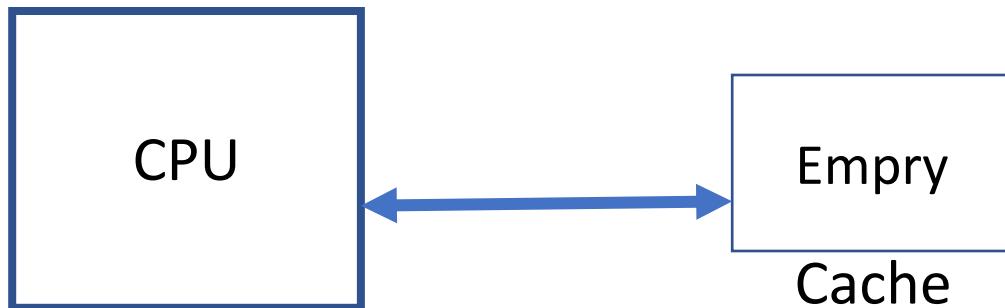
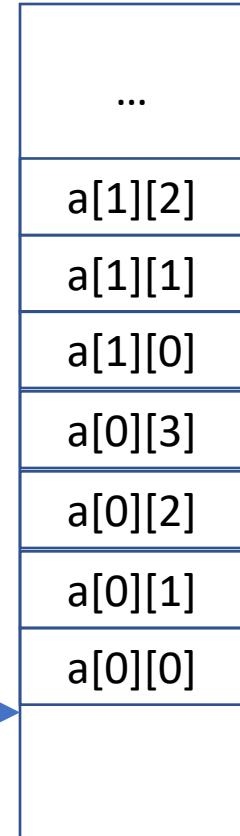
Locality example: the second case

```
int sum_2d_array2(int a[N][M]){
    int i, j, sum=0;
    for(i=0; i<M; i++)
        for(j=0; j<N; j++)
            sum = sum + a[j][i];
    return sum;
}
```

Computes the sum over a column.

$$\begin{aligned} &a[0][0]+a[1][0]+\dots \\ &+a[0][1]+a[1][1]+\dots \end{aligned}$$

Array elements are initially in the Main Memory.



1. CPU requires a[0][0] to compute sum = sum + a[0][0]

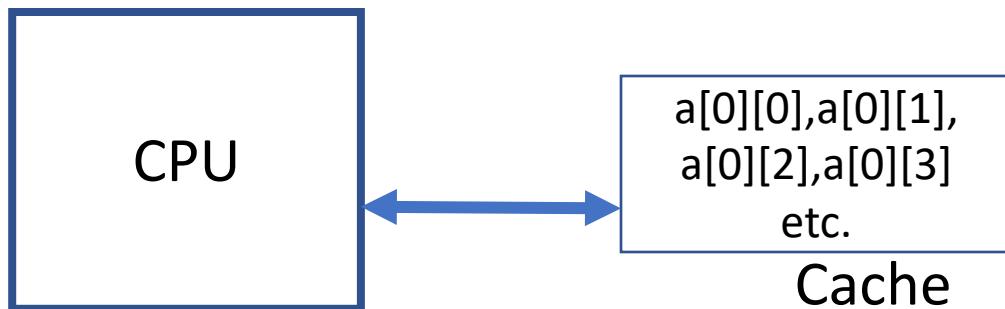
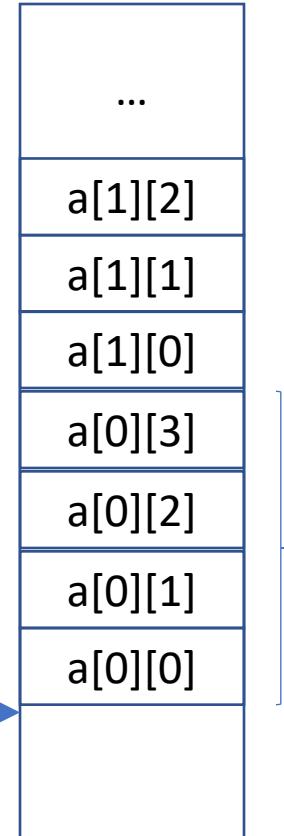
Locality example: the second case

```
int sum_2d_array2(int a[N][M]){
    int i, j, sum=0;
    for(i=0; i<M; i++)
        for(j=0; j<N; j++)
            sum = sum + a[j][i];
    return sum;
}
```

Computes the sum over a column.

$$\begin{aligned} &a[0][0]+a[1][0]+\dots \\ &+a[0][1]+a[1][1]+\dots \end{aligned}$$

Array elements are initially in the Main Memory.



1. CPU requires a[0][0] to compute sum = sum + a[0][0]
2. Due to 'spatial locality', say a[0][0], a[0][1], a[0][2] etc. are loaded from Main Memory to Cache. [100 cycles are spent to access Main Memory]

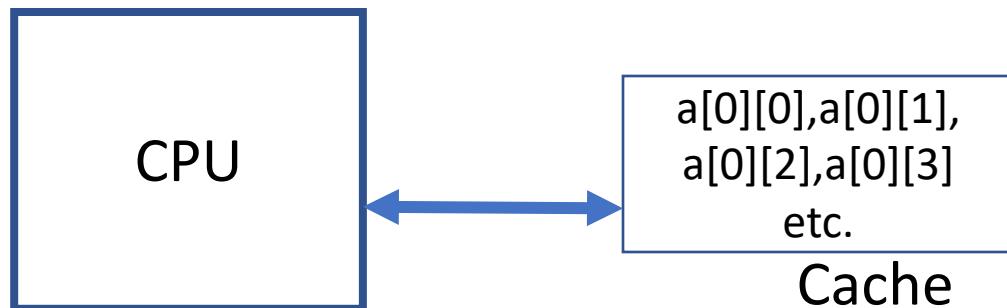
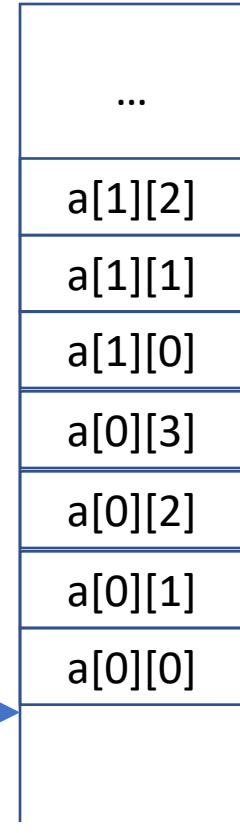
Locality example: the second case

```
int sum_2d_array2(int a[N][M]){
    int i, j, sum=0;
    for(i=0; i<M; i++)
        for(j=0; j<N; j++)
            sum = sum + a[j][i];
    return sum;
}
```

Computes the sum over a column.

$$\begin{aligned} &a[0][0]+a[1][0]+\dots \\ &+a[0][1]+a[1][1]+\dots \end{aligned}$$

Array elements are initially in the Main Memory.



3. However, CPU computes $\text{sum}=\text{sum}+a[0][0]+a[1][0]+a[2][0]+\dots$
4. Since, none of { $a[1][0]$, $a[2][0]$, ...} are in the Cache. Hence, Main Memory is accessed for each of them. [100 cycles for every access]

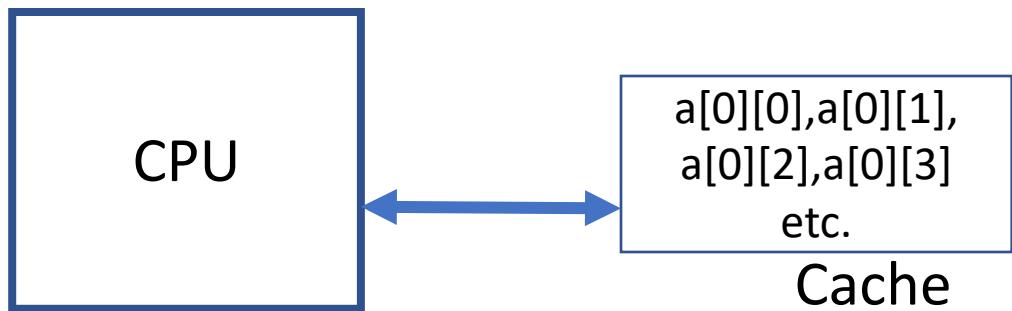
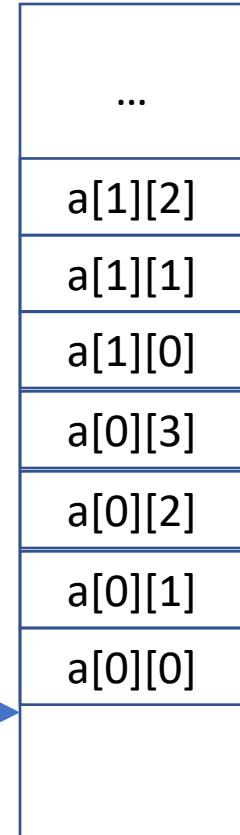
Locality example: the second case

```
int sum_2d_array2(int a[N][M]){
    int i, j, sum=0;
    for(i=0; i<M; i++)
        for(j=0; j<N; j++)
            sum = sum + a[j][i];
    return sum;
}
```

Computes the sum over a column.

$$\begin{aligned} &a[0][0]+a[1][0]+\dots \\ &+a[0][1]+a[1][1]+\dots \end{aligned}$$

Array elements are initially in the Main Memory.



Disadvantage: Cache miss happens always

Locality example: conclusions

```
int sum_2d_array1(int a[N][M]){
    int i, j, sum=0;
    for(i=0; i<N; i++)
        for(j=0; j<M; j++)
            sum = sum + a[i][j];
    return sum;
}
```

High Cache hit rate!
Hence, much
faster execution ☺

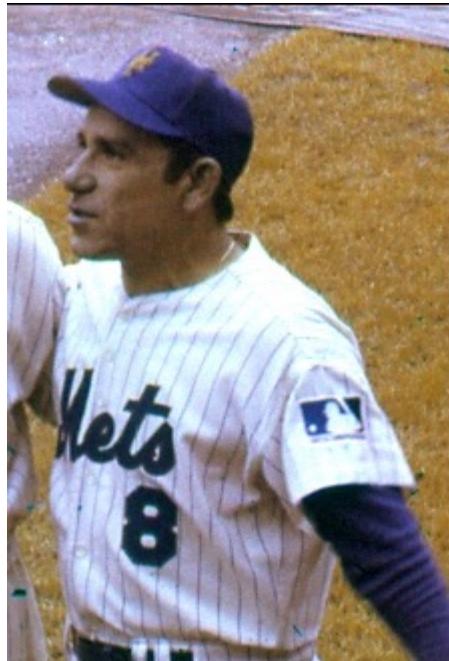
```
int sum_2d_array2(int a[N][M]){
    int i, j, sum=0;
    for(i=0; i<M; i++)
        for(j=0; j<N; j++)
            sum = sum + a[j][i];
    return sum;
}
```

Always Cache miss.
Order of magnitude
slower execution ☹

This is where the difference between a Java programmer and a C programmer becomes apparent.

Theory vs practice

*“In theory there is no difference between theory and practice.
But in practice there is.” - Yogi Berra*



We also see ‘theory vs practice’ when we run algorithms.

Why should I use pointers when I can access objects directly?

Why should I use pointers when I can access objects directly?

- We have seen applications of pointers in memory management

Why should I use pointers when I can access objects directly?

The following example will explain this part.

```
typedef struct pair{  
    int x[512];  
    int y[512];  
} pair;
```

Consider a large compound data type ‘pair’

```
...
pair add1(pair a, pair b){
    pair temp;
    int i;
    for(i=0; i<512; i++){
        temp.x[i] = a.x[i]+b.x[i];
        temp.y[i] = a.y[i]+b.y[i];
    }
    return temp;
}
int main(){
    int i;
    pair a, b, c;
    ...
    //approach1
    c = add1(a, b);
    ...
}
```

Approach 1:
Compute c by passing
objects

```
...
void add2(pair *p0, pair *p1, pair *p2){
    int i;
    for(i=0; i<512; i++){
        p2->x[i] = p0->x[i] + p1->x[i];
        p2->y[i] = p0->y[i] + p1->y[i];
    }
    return;
}
int main(){
    int i;
    pair a, b, c;
    pair *p0, *p1, *p2;
    p0 = &a; p1=&b; p2=&c;
    ...
    add2(p0, p1, p2);
    ...
}
```

Approach 2:
Compute c by passing
pointers

Both approaches compute
the same result.

Question:
Which one would be better
for a system?

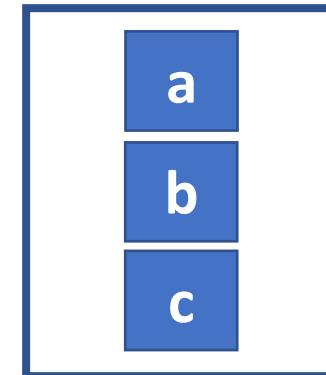
```

...
pair add1(pair a, pair b){
    pair temp;
    int i;
    for(i=0; i<512; i++){
        temp.x[i] = a.x[i]+b.x[i];
        temp.y[i] = a.y[i]+b.y[i];
    }
    return temp;
}
int main(){
    int i;
    pair a, b, c;
    ...
    //approach1
    c = add1(a, b);
    ...
}

```

Approach 1:

Compute c by passing objects



Stack
frame of
main()

Initially large objects a, b, c
are in stack frame of main()

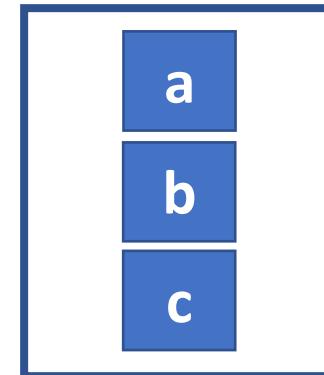
```

...
pair add1(pair a, pair b){
    pair temp;
    int i;
    for(i=0; i<512; i++){
        temp.x[i] = a.x[i]+b.x[i];
        temp.y[i] = a.y[i]+b.y[i];
    }
    return temp;
}
int main(){
    int i;
    pair a, b, c;
    ...
    //approach1
    c = add1(a, b);
    ...
}

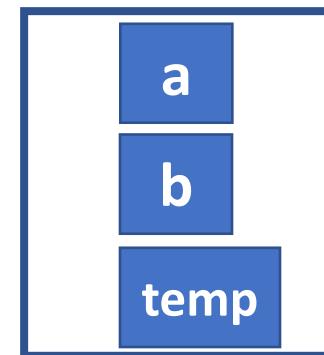
```

Approach 1:

Compute c by passing objects



Stack
frame of
main()



Stack
frame of
add1()

add1() is called and then
large a and b are passed.
→ they are **copied**.

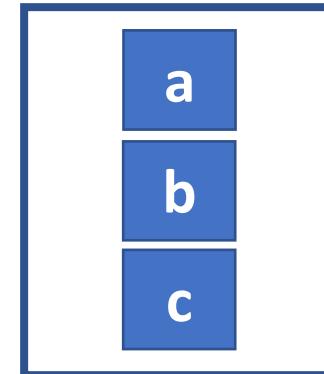
```

...
pair add1(pair a, pair b){
    pair temp;
    int i;
    for(i=0; i<512; i++){
        temp.x[i] = a.x[i]+b.x[i];
        temp.y[i] = a.y[i]+b.y[i];
    }
    return temp;
}
int main(){
    int i;
    pair a, b, c;
    ...
    //approach1
    c = add1(a, b);
    ...
}

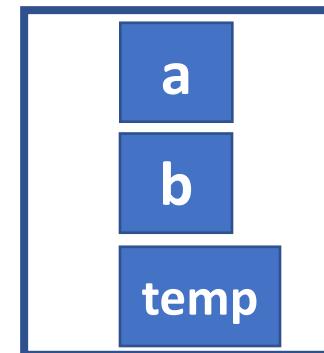
```

Approach 1:

Compute c by passing objects



Stack
frame of
main()



Stack
frame of
add1()

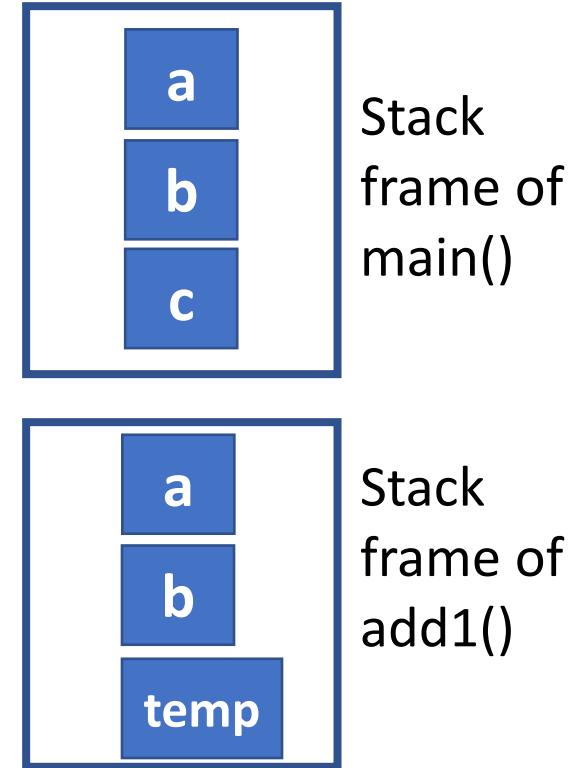
In the end add1() returns
large 'temp'.
→ It is copied into c

```

...
pair add1(pair a, pair b){
    pair temp;
    int i;
    for(i=0; i<512; i++){
        temp.x[i] = a.x[i]+b.x[i];
        temp.y[i] = a.y[i]+b.y[i];
    }
    return temp;
}
int main(){
    int i;
    pair a, b, c;
    ...
    //approach1
    c = add1(a, b);
    ...
}

```

Approach 1:
Compute c by passing objects



Lots of big-data copy happen.

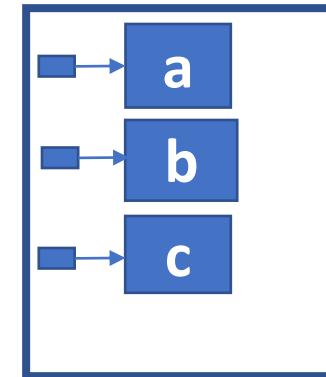
In the end add1() returns large 'temp'.
→ It is copied into c

```

...
void add2(pair *p0, pair *p1, pair *p2){
    int i;
    for(i=0; i<512; i++){
        p2->x[i] = p0->x[i] + p1->x[i];
        p2->y[i] = p0->y[i] + p1->y[i];
    }
    return;
}
int main(){
    int i;
    pair a, b, c;
    pair *p0, *p1, *p2;
    p0 = &a; p1=&b; p2=&c;
    ...
    add2(p0, p1, p2);
    ...
}

```

Approach 2:
Compute c by passing
pointers



Stack
frame of
main()

Initially large objects a, b, c
and 8-byte pointers
are in stack frame of main()
11

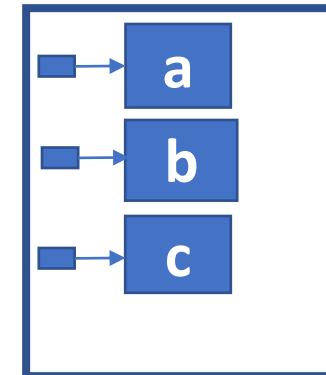
```

...
void add2(pair *p0, pair *p1, pair *p2){
    int i;
    for(i=0; i<512; i++){
        p2->x[i] = p0->x[i] + p1->x[i];
        p2->y[i] = p0->y[i] + p1->y[i];
    }
    return;
}

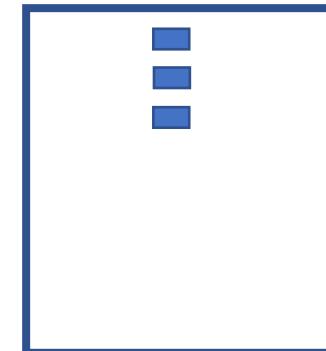
int main(){
    int i;
    pair a, b, c;
    pair *p0, *p1, *p2;
    p0 = &a; p1=&b; p2=&c;
    ...
    add2(p0, p1, p2);
    ...
}

```

Approach 2:
Compute c by passing
pointers



Stack
frame of
main()



Stack
frame of
add2()

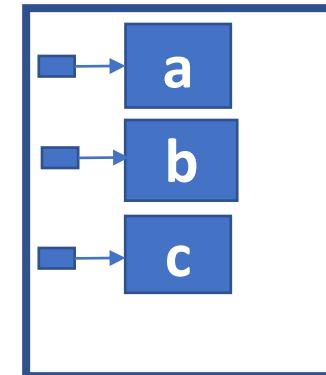
add2() is called and then
pointers are passed.
→ **Small 8-byte pointers**
are **copied**

```

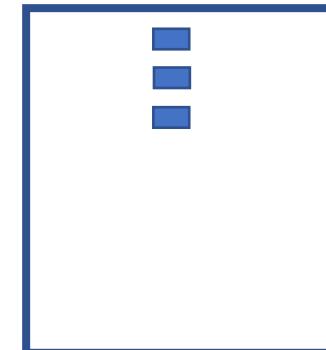
...
void add2(pair *p0, pair *p1, pair *p2){
    int i;
    for(i=0; i<512; i++){
        p2->x[i] = p0->x[i] + p1->x[i];
        p2->y[i] = p0->y[i] + p1->y[i];
    }
    return;
}
int main(){
    int i;
    pair a, b, c;
    pair *p0, *p1, *p2;
    p0 = &a; p1=&b; p2=&c;
    ...
    add2(p0, p1, p2);
    ...
}

```

Approach 2:
Compute c by passing
pointers



Stack
frame of
main()



Stack
frame of
add2()

add2() updates c directly

So, overall only 3 pointers
are copied!

Conclusions: pass-by-value vs pass-by-pointer

- Pass-by-value copies objects from one stack frame to other
- Pass-by-pointer copies only pointers

Thus, pass-by-pointer is more efficient for large data objects

Application of memory management in C: Cache-efficient algorithms

Sujoy Sinha Roy

School of Computer Science

University of Birmingham

Locality example

Both functions compute the sum of the elements of an input 2D matrix.
Which one has better locality?

```
int sum_2d_array1(int a[N][M]){
    int i, j, sum=0;
    for(i=0; i<N; i++)
        for(j=0; j<M; j++)
            sum = sum + a[i][j];
    return sum;
}
```

Computes the sum over a row.
 $a[0][0]+a[0][1]+\dots+a[1][0]+a[1][1]+\dots$

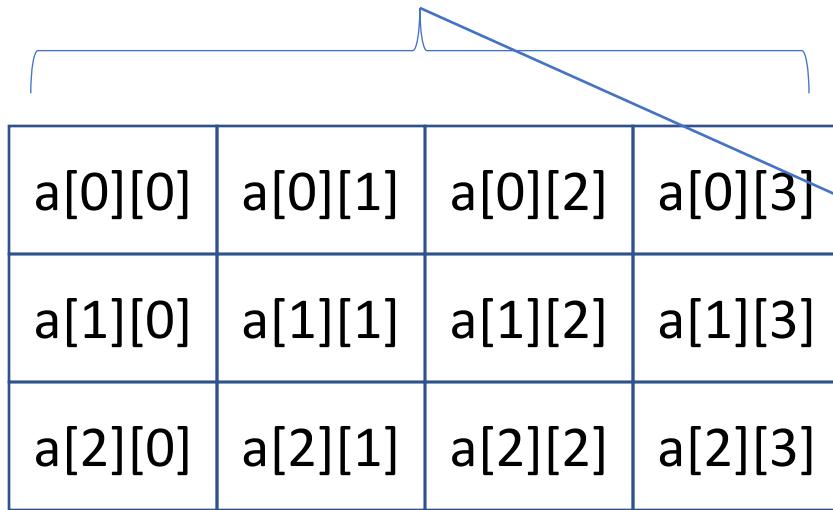
```
int sum_2d_array2(int a[N][M]){
    int i, j, sum=0;
    for(i=0; i<M; i++)
        for(j=0; j<N; j++)
            sum = sum + a[j][i];
    return sum;
}
```

Computes the sum over a column.
 $a[0][0]+a[1][0]+\dots+a[0][1]+a[1][1]+\dots$

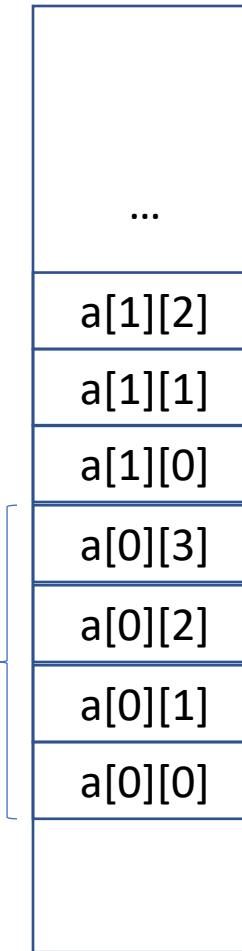
Recap: Memory layout of two-dimensional array

C compiler stores 2D array in **row-major** order

- All elements of Row #0 are stored
- then all elements of Row #1 are stored
- and so on



Logical view of array $a[3][4]$



Memory layout

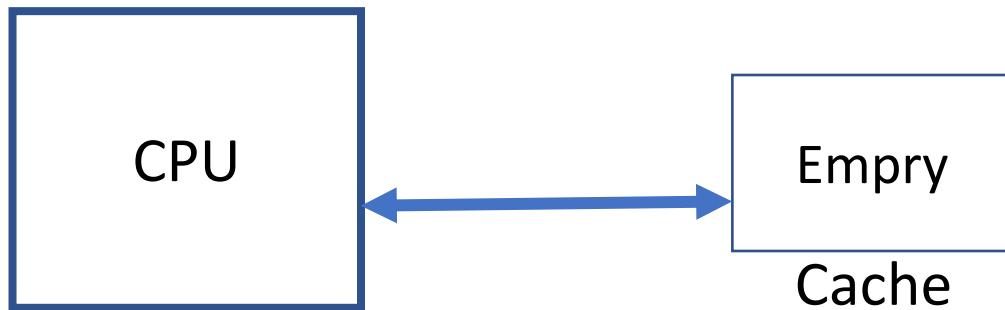
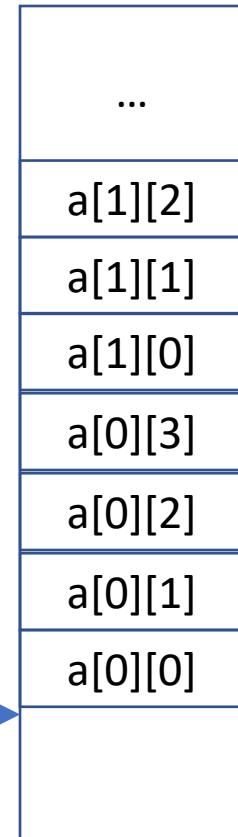
Locality example: the first case

```
int sum_2d_array1(int a[N][M]){
    int i, j, sum=0;
    for(i=0; i<N; i++)
        for(j=0; j<M; j++)
            sum = sum + a[i][j];
    return sum;
}
```

Computes the sum over a row.

$$\begin{aligned} &a[0][0]+a[0][1]+\dots \\ &+a[1][0]+a[1][1]+\dots \end{aligned}$$

Array elements
are initially in the
Main Memory.



1. CPU requires a[0][0] to compute sum = sum + a[0][0]

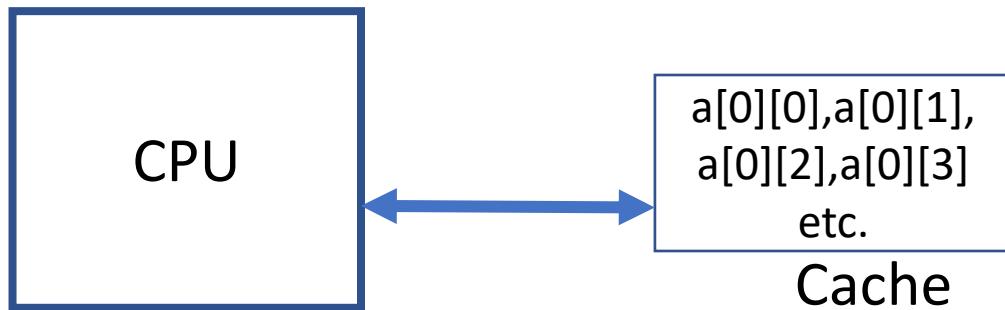
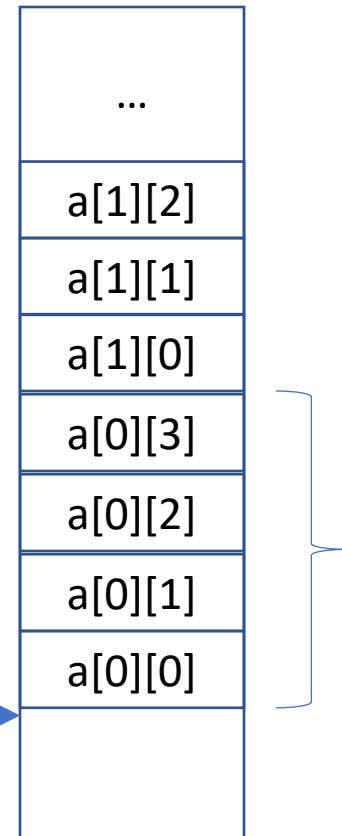
Locality example: the first case

```
int sum_2d_array1(int a[N][M]){
    int i, j, sum=0;
    for(i=0; i<N; i++)
        for(j=0; j<M; j++)
            sum = sum + a[i][j];
    return sum;
}
```

Computes the sum over a row.

$$\begin{aligned} &a[0][0]+a[0][1]+\dots \\ &+a[1][0]+a[1][1]+\dots \end{aligned}$$

Array elements are initially in the Main Memory.



1. CPU requires a[0][0] to compute $\text{sum} = \text{sum} + a[0][0]$
2. Due to 'spatial locality', say a[0][0], a[0][1], a[0][2] etc. are loaded from Main Memory to Cache. [100 cycles are spent to access Main Memory]

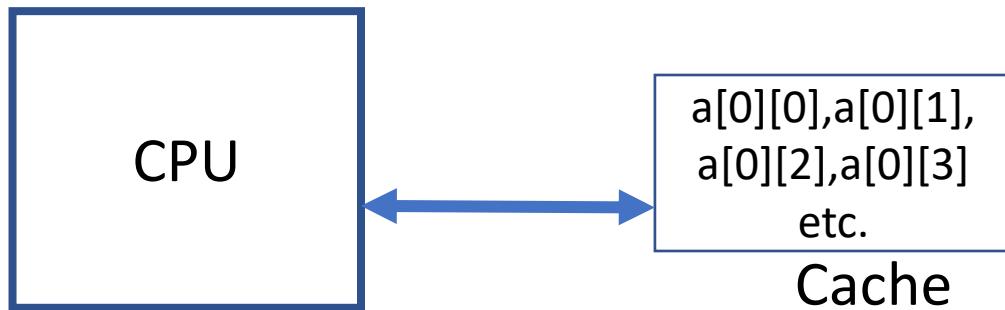
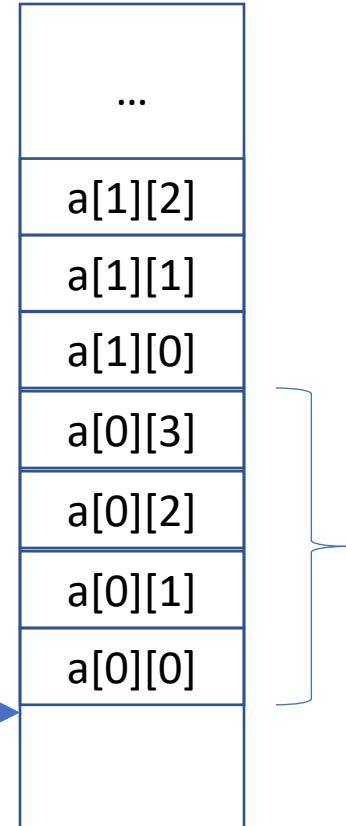
Locality example: the first case

```
int sum_2d_array1(int a[N][M]){
    int i, j, sum=0;
    for(i=0; i<N; i++)
        for(j=0; j<M; j++)
            sum = sum + a[i][j];
    return sum;
}
```

Computes the sum over a row.

$$\begin{aligned} &a[0][0]+a[0][1]+\dots \\ &+a[1][0]+a[1][1]+\dots \end{aligned}$$

Array elements are initially in the Main Memory.



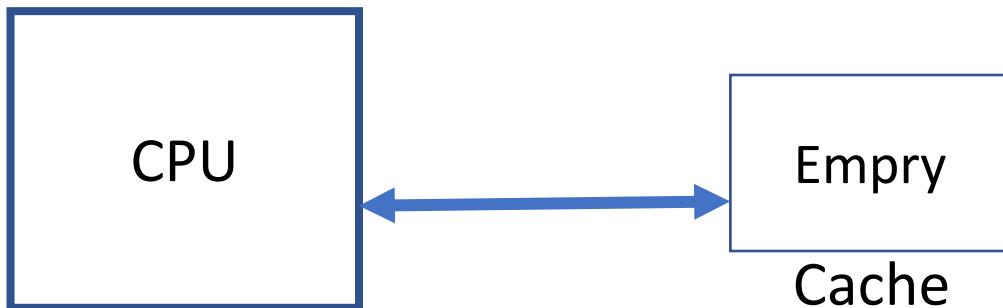
3. CPU computes $\text{sum}=a[0][0]+a[0][1]+a[0][2] \dots$ by reading the elements from Cache. [Each access takes 4 cycles]
4. Advantage: Cache hit happens most of the times.

Locality example: the second case

```
int sum_2d_array2(int a[N][M]){
    int i, j, sum=0;
    for(i=0; i<M; i++)
        for(j=0; j<N; j++)
            sum = sum + a[j][i];
    return sum;
}
```

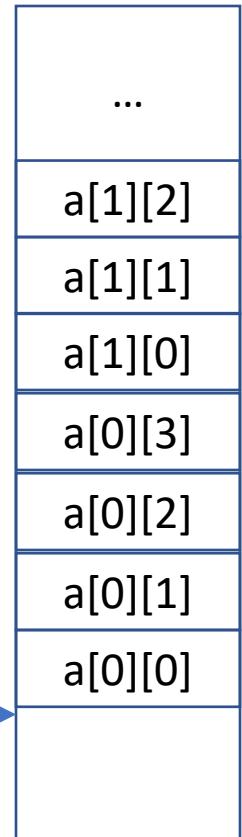
Computes the sum over a column.

$$\begin{aligned} &a[0][0]+a[1][0]+\dots \\ &+a[0][1]+a[1][1]+\dots \end{aligned}$$



1. CPU requires $a[0][0]$ to compute $\text{sum} = \text{sum} + a[0][0]$

Array elements are initially in the Main Memory.



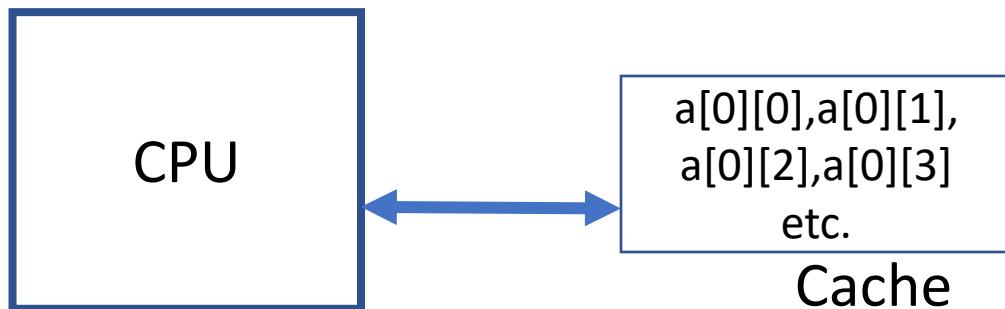
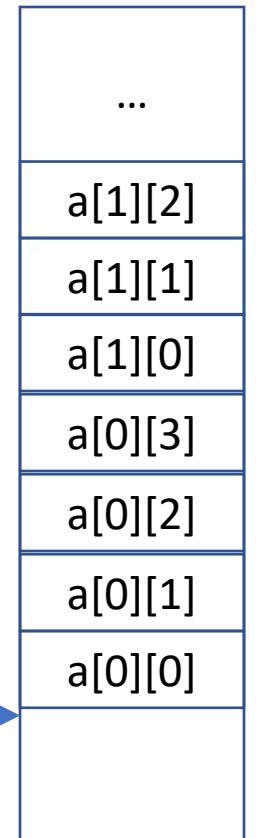
Locality example: the second case

```
int sum_2d_array2(int a[N][M]){
    int i, j, sum=0;
    for(i=0; i<M; i++)
        for(j=0; j<N; j++)
            sum = sum + a[j][i];
    return sum;
}
```

Computes the sum over a column.

$$\begin{aligned} &a[0][0]+a[1][0]+\dots \\ &+a[0][1]+a[1][1]+\dots \end{aligned}$$

Array elements are initially in the Main Memory.



1. CPU requires a[0][0] to compute sum = sum + a[0][0]
2. Due to 'spatial locality', say a[0][0], a[0][1], a[0][2] etc. are loaded from Main Memory to Cache. [100 cycles are spent to access Main Memory]

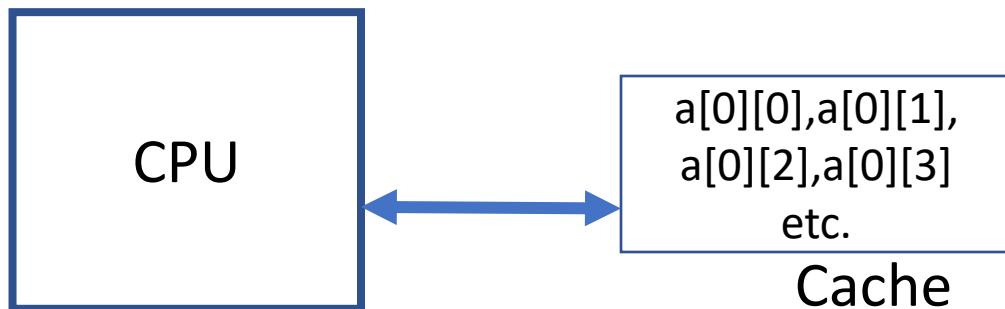
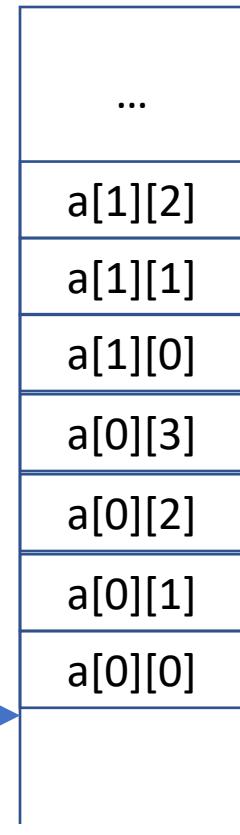
Locality example: the second case

```
int sum_2d_array2(int a[N][M]){
    int i, j, sum=0;
    for(i=0; i<M; i++)
        for(j=0; j<N; j++)
            sum = sum + a[j][i];
    return sum;
}
```

Computes the sum over a column.

$$\begin{aligned} &a[0][0]+a[1][0]+\dots \\ &+a[0][1]+a[1][1]+\dots \end{aligned}$$

Array elements are initially in the Main Memory.



3. However, CPU computes $\text{sum}=\text{sum}+a[0][0]+a[1][0]+a[2][0]+\dots$
4. Since, none of {a[1][0], a[2][0], ...} are in the Cache. Hence, Main Memory is accessed for each of them. [100 cycles for every access]

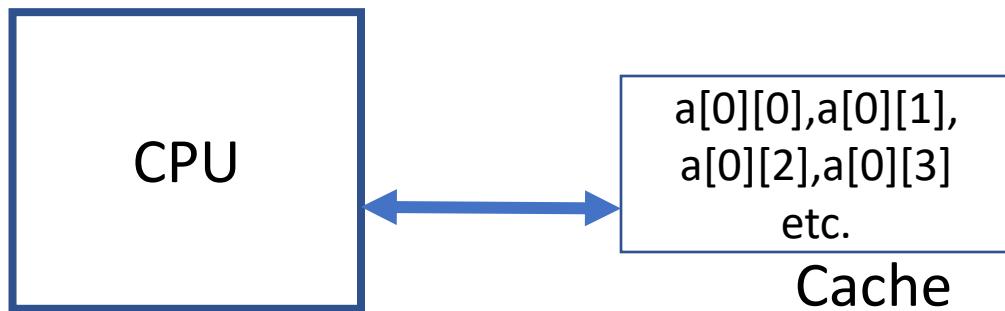
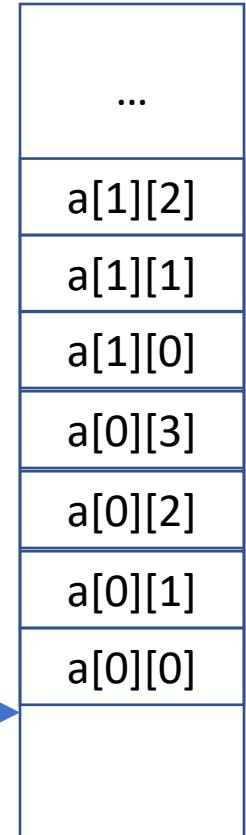
Locality example: the second case

```
int sum_2d_array2(int a[N][M]){
    int i, j, sum=0;
    for(i=0; i<M; i++)
        for(j=0; j<N; j++)
            sum = sum + a[j][i];
    return sum;
}
```

Computes the sum over a column.

$$\begin{aligned} &a[0][0]+a[1][0]+\dots \\ &+a[0][1]+a[1][1]+\dots \end{aligned}$$

Array elements are initially in the Main Memory.



Disadvantage: Cache miss happens always

Locality example: conclusions

```
int sum_2d_array1(int a[N][M]){
    int i, j, sum=0;
    for(i=0; i<N; i++)
        for(j=0; j<M; j++)
            sum = sum + a[i][j];
    return sum;
}
```

High Cache hit rate!
Hence, much
faster execution ☺

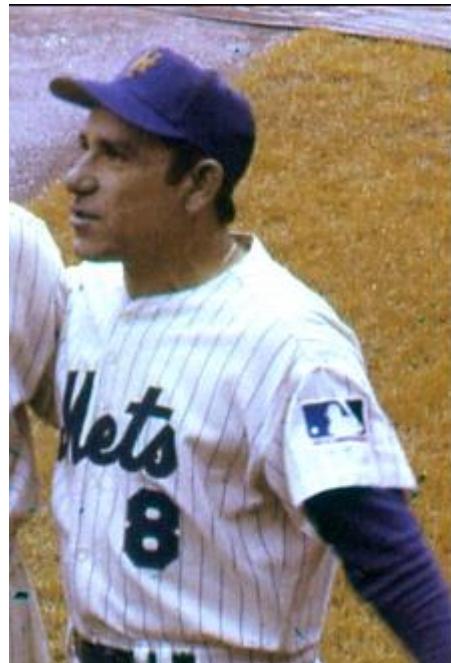
```
int sum_2d_array2(int a[N][M]){
    int i, j, sum=0;
    for(i=0; i<M; i++)
        for(j=0; j<N; j++)
            sum = sum + a[j][i];
    return sum;
}
```

Always Cache miss.
Order of magnitude
slower execution ☹

This is where the difference between a Java programmer and a C programmer becomes apparent.

Theory vs practice

*“In theory there is no difference between theory and practice.
But in practice there is.” - Yogi Berra*



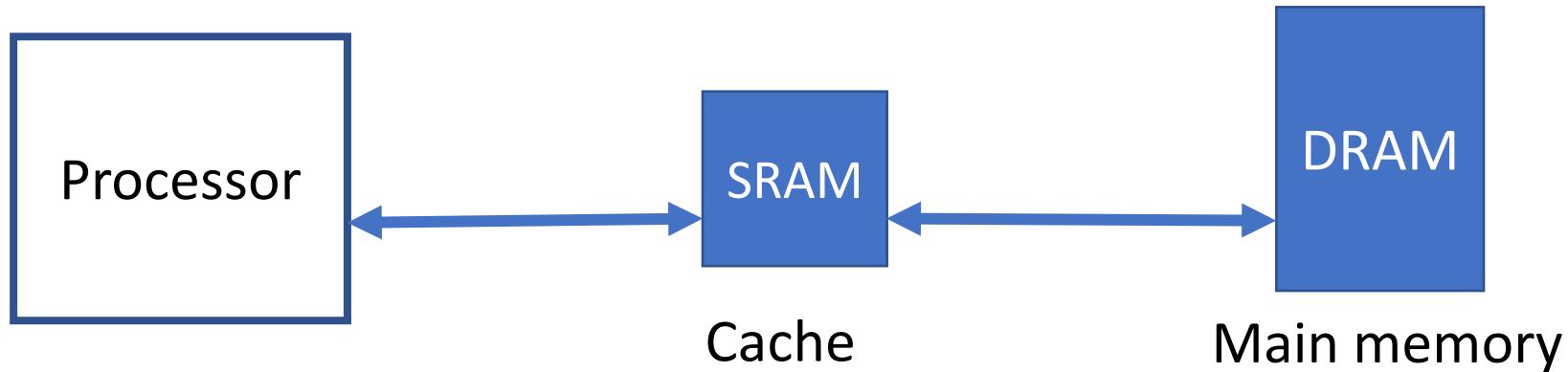
We also see ‘theory vs practice’ when we run algorithms.

Application of memory management in C: Cache-efficient algorithms Matrix Multiplication

Sujoy Sinha Roy

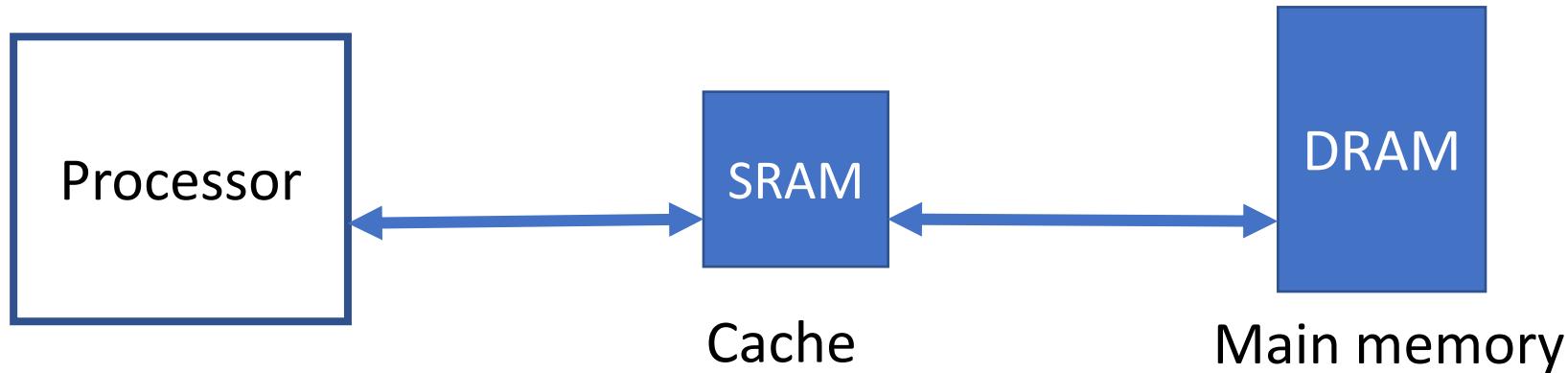
School of Computer Science
University of Birmingham

Hypothetical computer



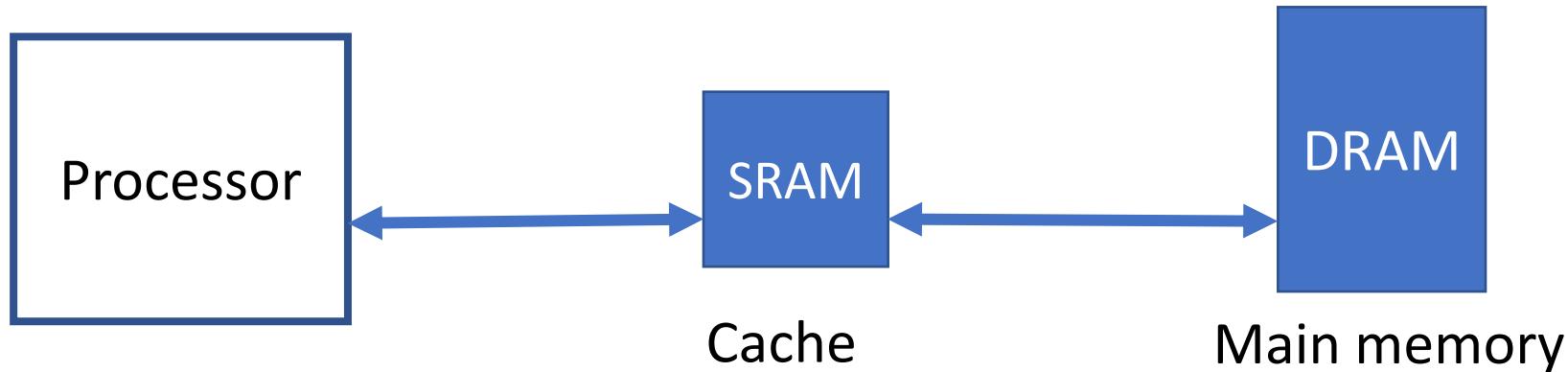
Assumption: processor can access fast cache in negligible time

Simplified model for computation-communication tradeoff



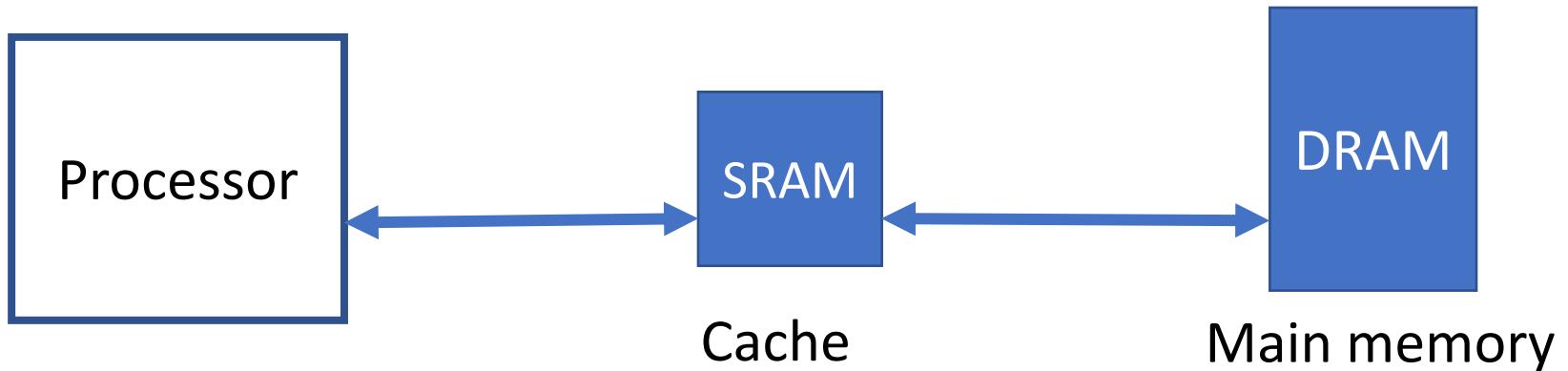
- Assume a generic program which performs
 - m = number of data movements between fast and slow memory
 - f = number of arithmetic operations

Simplified model for computation-communication tradeoff



- Assume a generic program which performs
 - m = number of data movements between fast and slow memory
 - f = number of arithmetic operations
 - t_m = time per data movement
 - t_f = time per arithmetic operation
- Total time for program execution = $f * t_f + m * t_m$

Simplified model for computation-communication tradeoff

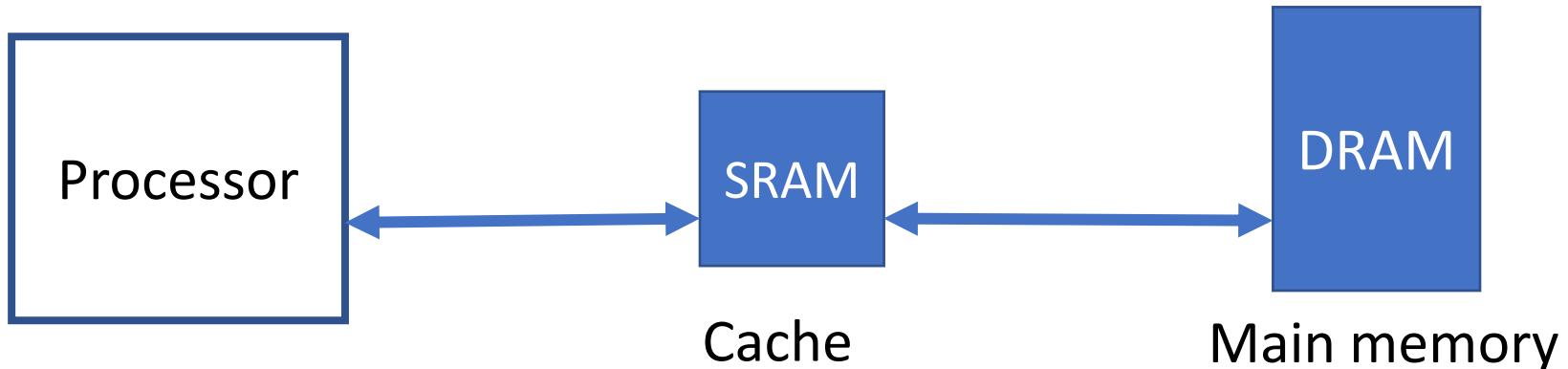


- Assume a generic program which performs
 - m = number of data movements between fast and slow memory
 - f = number of arithmetic operations
 - t_m = time per data movement
 - t_f = time per arithmetic operation
 - $q = f/m$ average number of arithmetic operations per slow memory access

Higher q indicates that the algorithm is ***computation-centric*** and performs less data movement.
So, a higher q results in a better system-performance.



Simplified model for computation-communication tradeoff



- Assume a generic program which performs
 - m = number of data movements between fast and slow memory
 - f = number of arithmetic operations
 - t_m = time per data movement
 - t_f = time per arithmetic operation
 - $q = f/m$ average number of arithmetic operations per slow memory access
- Total time for program execution =
$$\begin{aligned} & f * t_f + m * t_m \\ &= f * t_f * (1 + t_m/t_f * \textcolor{red}{1/q}) \end{aligned}$$

$$\begin{aligned}\text{Total time for program execution} &= f * t_f + m * t_m \\ &= f * t_f * (1 + t_m/t_f * \mathbf{1/q})\end{aligned}$$

where

- t_m = time per data movement (speed of memory)
- t_f = time per arithmetic operation (speed of processor)

Hardware constants
(programmer cannot control)

Goal: Cache-efficient algorithms increase \mathbf{q} by reducing the number of slow main-memory access → improves system performance.

Case study: Arithmetic on dynamically allocated matrices

Assume we have three matrices:

$$A = \begin{vmatrix} A_{00} & A_{01} & A_{02} & A_{03} \\ A_{10} & A_{11} & A_{12} & A_{13} \\ A_{20} & A_{21} & A_{22} & A_{23} \\ A_{30} & A_{31} & A_{32} & A_{33} \end{vmatrix} \quad B = \begin{vmatrix} B_{00} & B_{01} & B_{02} & B_{03} \\ B_{10} & B_{11} & B_{12} & B_{13} \\ B_{20} & B_{21} & B_{22} & B_{23} \\ B_{30} & B_{31} & B_{32} & B_{33} \end{vmatrix}$$

$$C = \begin{vmatrix} C_{00} & C_{01} & C_{02} & C_{03} \\ C_{10} & C_{11} & C_{12} & C_{13} \\ C_{20} & C_{21} & C_{22} & C_{23} \\ C_{30} & C_{31} & C_{32} & C_{33} \end{vmatrix}$$

We want to compute $[C] = [C] + [A]^*[B]$

Creating a dynamically-allocated matrix

We can create a matrix with n rows and n columns as

```
T *A;
```

```
A = (T *) malloc(n*n*sizeof(T));
```

If we want to store data in row-major order then

```
for (int i = 0; i < n; ++i) {  
    for (int j = 0; j < n; ++j)  
        scanf("%f", p+i*n+j);  
}
```

If we want to store data in column-major order then

```
for (int i = 0; i < n; ++i) {  
    for (int j = 0; j < n; ++j)  
        scanf("%f", p+i+j*n);  
}
```

Case study: Arithmetic on dynamically allocated matrices

Assume that matrices are stored in column-major order.

$$A = \begin{vmatrix} A_{00} & A_{01} & A_{02} & A_{03} \\ A_{10} & A_{11} & A_{12} & A_{13} \\ A_{20} & A_{21} & A_{22} & A_{23} \\ A_{30} & A_{31} & A_{32} & A_{33} \end{vmatrix} \quad B = \begin{vmatrix} B_{00} & B_{01} & B_{02} & B_{03} \\ B_{10} & B_{11} & B_{12} & B_{13} \\ B_{20} & B_{21} & B_{22} & B_{23} \\ B_{30} & B_{31} & B_{32} & B_{33} \end{vmatrix}$$

Logical view

A00	A10	A20	A30	A01	A11	A21	A31	A02	A12	A22	A32	A03	A13	A23	A33
-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----

B00	B10	B20	B30	B01	B11	B21	B31	B02	B12	B22	B32	B03	B13	B23	B33
-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----

Memory layout

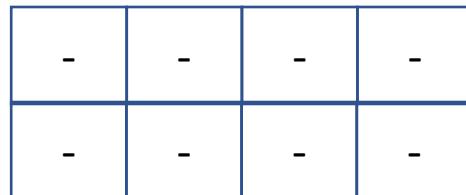
Note: All data is initially in slow Main memory

Compute $C = C + A^*B$

A00 A01 A02 A03	B00 B01 B02 B03	Logical view
A10 A11 A12 A13	B10 B11 B12 B13	
A20 A21 A22 A23	B20 B21 B22 B23	
A30 A31 A32 A33	B30 B31 B32 B33	

A00	A10	A20	A30	A01	A11	A21	A31	A02	A12	A22	A32	A03	A13	A23	A33
B00	B10	B20	B30	B01	B11	B21	B31	B02	B12	B22	B32	B03	B13	B23	B33

A and B are stored
in slow memory in
column-major order



Cache (fast but small)

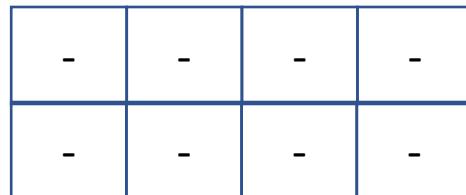
Compute $C = C + A * B$

A00 A01 A02 A03	B00 B01 B02 B03	Logical view
A10 A11 A12 A13	B10 B11 B12 B13	
A20 A21 A22 A23	B20 B21 B22 B23	
A30 A31 A32 A33	B30 B31 B32 B33	

A00	A10	A20	A30	A01	A11	A21	A31	A02	A12	A22	A32	A03	A13	A23	A33
B00	B10	B20	B30	B01	B11	B21	B31	B02	B12	B22	B32	B03	B13	B23	B33

Compute $C_{00} = A_{00}*B_{00}+A_{01}*B_{10}+A_{02}*B_{20}+A_{03}*B_{30}$

A and B are stored
in slow memory in
column-major order



Cache (fast but small)

Compute $C = C + A^*B$

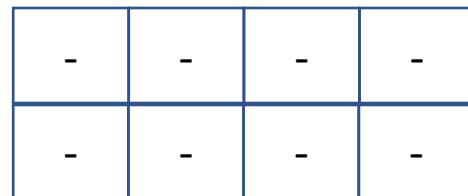
A00	A01	A02	A03	B00	B01	B02	B03	Logical view
A10	A11	A12	A13	B10	B11	B12	B13	
A20	A21	A22	A23	B20	B21	B22	B23	
A30	A31	A32	A33	B30	B31	B32	B33	

A00	A10	A20	A30	A01	A11	A21	A31	A02	A12	A22	A32	A03	A13	A23	A33
B00	B10	B20	B30	B01	B11	B21	B31	B02	B12	B22	B32	B03	B13	B23	B33

Compute $C_{00} = A_{00}*B_{00}+A_{01}*B_{10}+A_{02}*B_{20}+A_{03}*B_{30}$

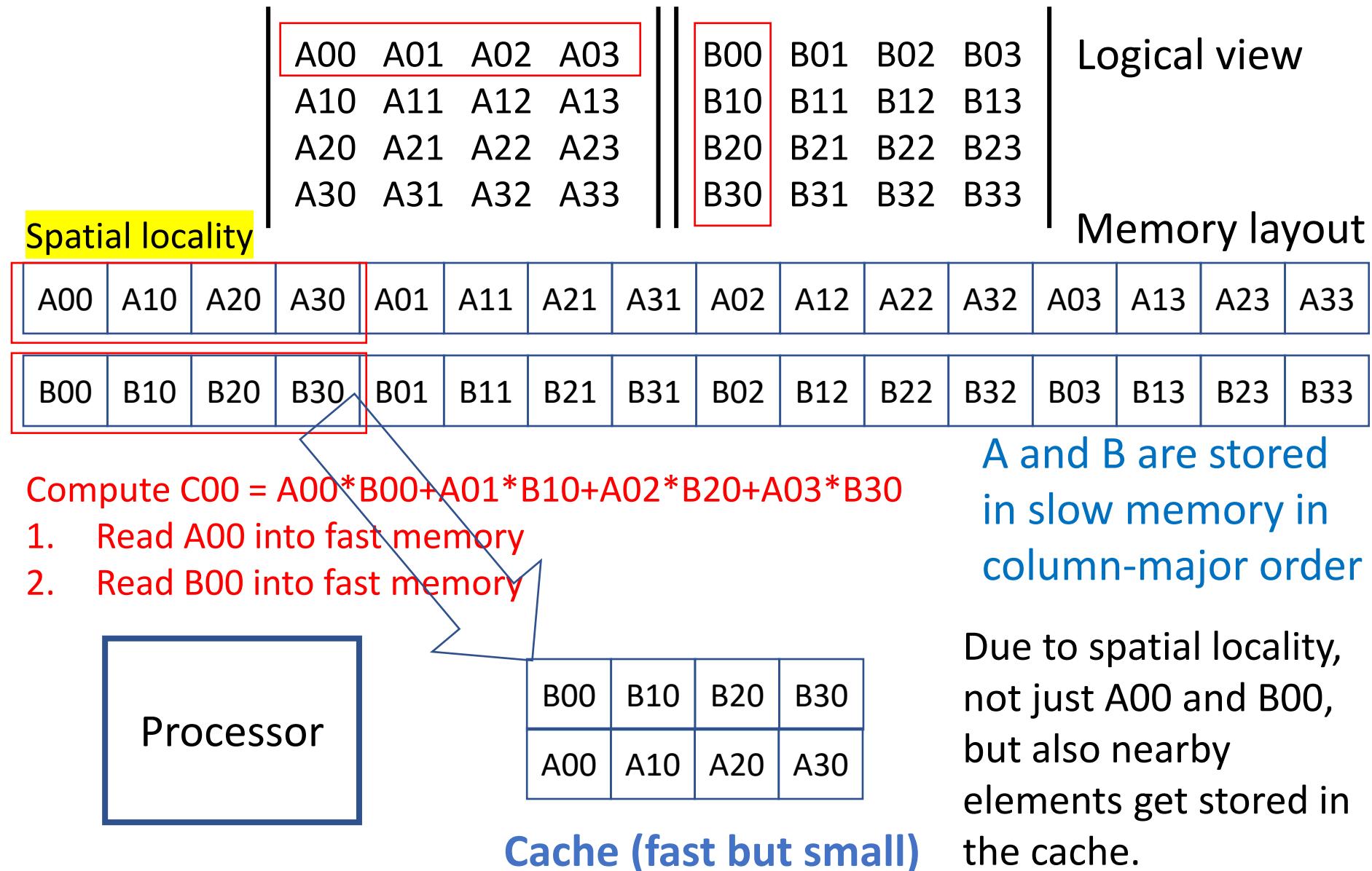
1. Read A00 into fast memory
2. Read B00 into fast memory

A and B are stored in slow memory in column-major order

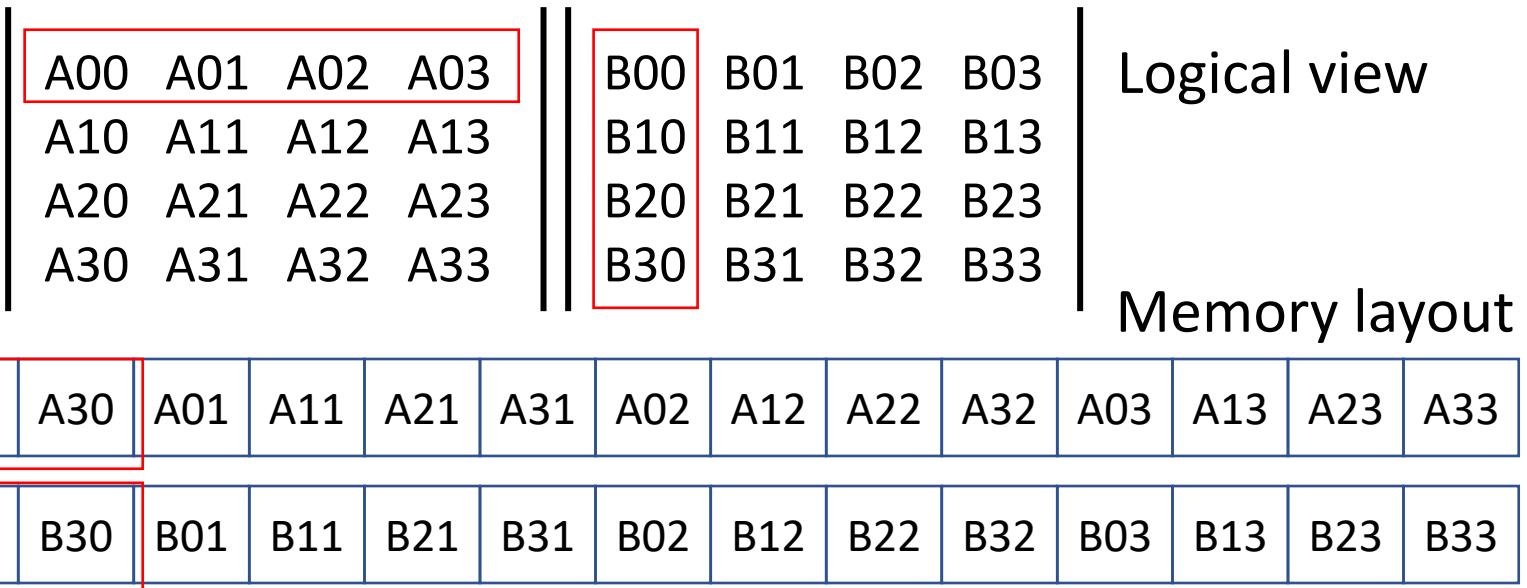


Cache (fast but small)

Compute $C = C + A \cdot B$



Compute $C = C + A * B$



Compute $C_{00} = A_{00} * B_{00} + A_{01} * B_{10} + A_{02} * B_{20} + A_{03} * B_{30}$

A and B are stored in slow memory in column-major order



B00	B10	B20	B30
A00	A10	A20	A30

Processor can read only {A00, B00} from fast mem.
So, $C_{00} = A_{00} * B_{00}$

Cache (fast but small)

Compute $C = C + A^*B$

A00	A01	A02	A03		B00	B01	B02	B03		Logical view		
A10	A11	A12	A13		B10	B11	B12	B13				
A20	A21	A22	A23		B20	B21	B22	B23				
A30	A31	A32	A33		B30	B31	B32	B33				
										Memory layout		
A30	A01	A11	A21	A31	A02	A12	A22	A32	A03	A13	A23	A33
B30	B01	B11	B21	B31	B02	B12	B22	B32	B03	B13	B23	B33

Compute C00 = (A00*B00)+A01*B10+A02*B20+A03*B30

A and B are stored
in slow memory in
column-major order

Processor

B00	B10	B20	B30
A00	A10	A20	A30

Processor can get only B10 from Cache.
A01 is needed from Main memory.

Cache (fast but small)

Compute $C = C + A^*B$

The diagram illustrates the memory layout and logical view for two arrays, A and B.

Logical view:

A00	A01	A02	A03		B00	B01	B02	B03
A10	A11	A12	A13		B10	B11	B12	B13
A20	A21	A22	A23		B20	B21	B22	B23
A30	A31	A32	A33		B30	B31	B32	B33

Memory layout:

A30	A01	A11	A21	A31	A02	A12	A22	A32	A03	A13	A23	A33
B30	B01	B11	B21	B31	B02	B12	B22	B32	B03	B13	B23	B33

The logical view shows two arrays, A and B, each with four elements. Array A elements are A00, A01, A02, A03; A10, A11, A12, A13; A20, A21, A22, A23; and A30, A31, A32, A33. Array B elements are B00, B01, B02, B03; B10, B11, B12, B13; B20, B21, B22, B23; and B30, B31, B32, B33. The memory layout shows the physical memory storage for both arrays. The first row of memory contains elements A30, A01, A11, A21, A31, A02, A12, A22, A32, A03, A13, A23, A33. The second row contains elements B30, B01, B11, B21, B31, B02, B12, B22, B32, B03, B13, B23, B33. Red boxes highlight the first four elements of each array in both the logical view and memory layout, showing they are contiguous in memory.

Compute C00 = (A00*B00)+A01*B10+A02*B20+A03*B30

3. Read A01 into fast memory (cache miss)

A and B are stored
in slow memory in
column-major order

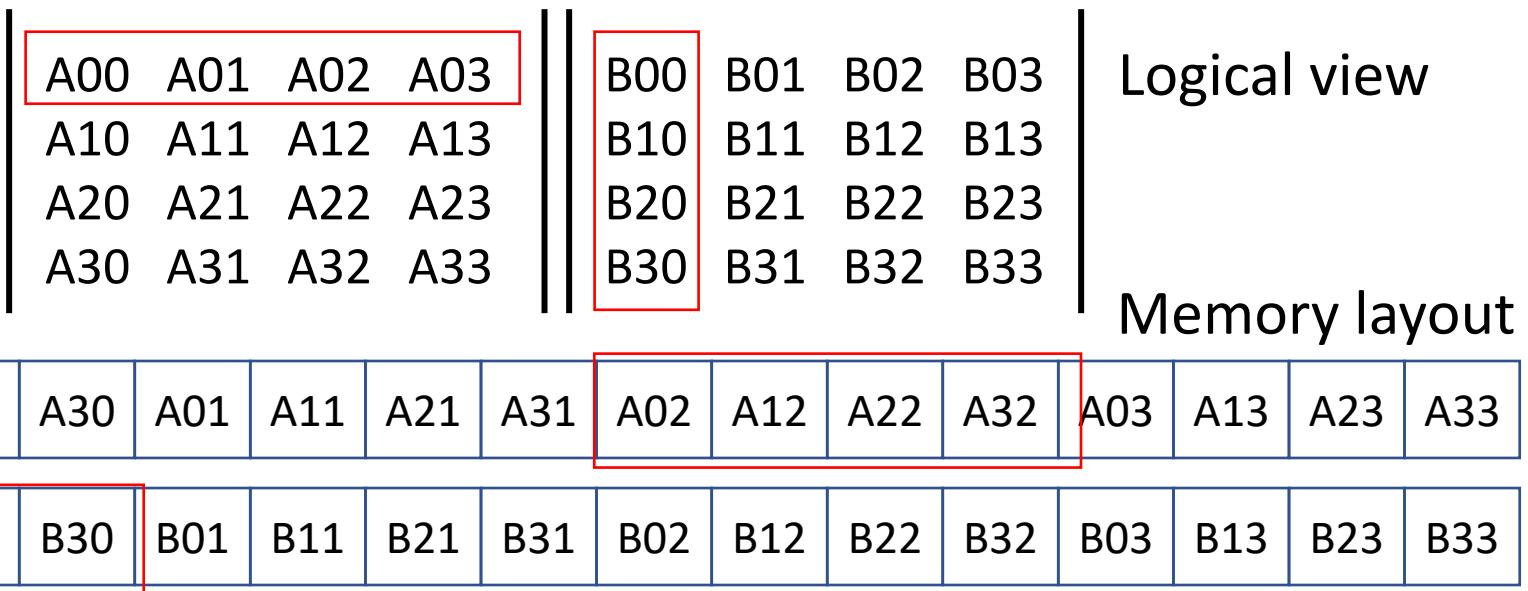
Processor

B00	B10	B20	B30
A01	A11	A21	A31

Cache (fast but small)

Processor can read only
 $\{A_01, B_{10}\}$ from fast mem
 $C_{00}=C_{00}+A_01*B_{10}$

Compute $C = C + A^*B$

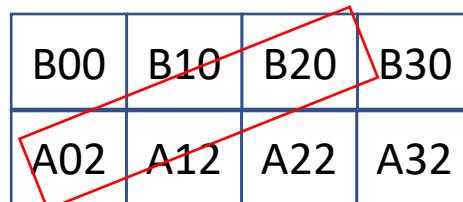


Compute C00 = (A00*B00+A01*B10)+A02*B20+A03*B30

4. Read A02 into fast memory (again cache miss!)

A and B are stored
in slow memory in
column-major order

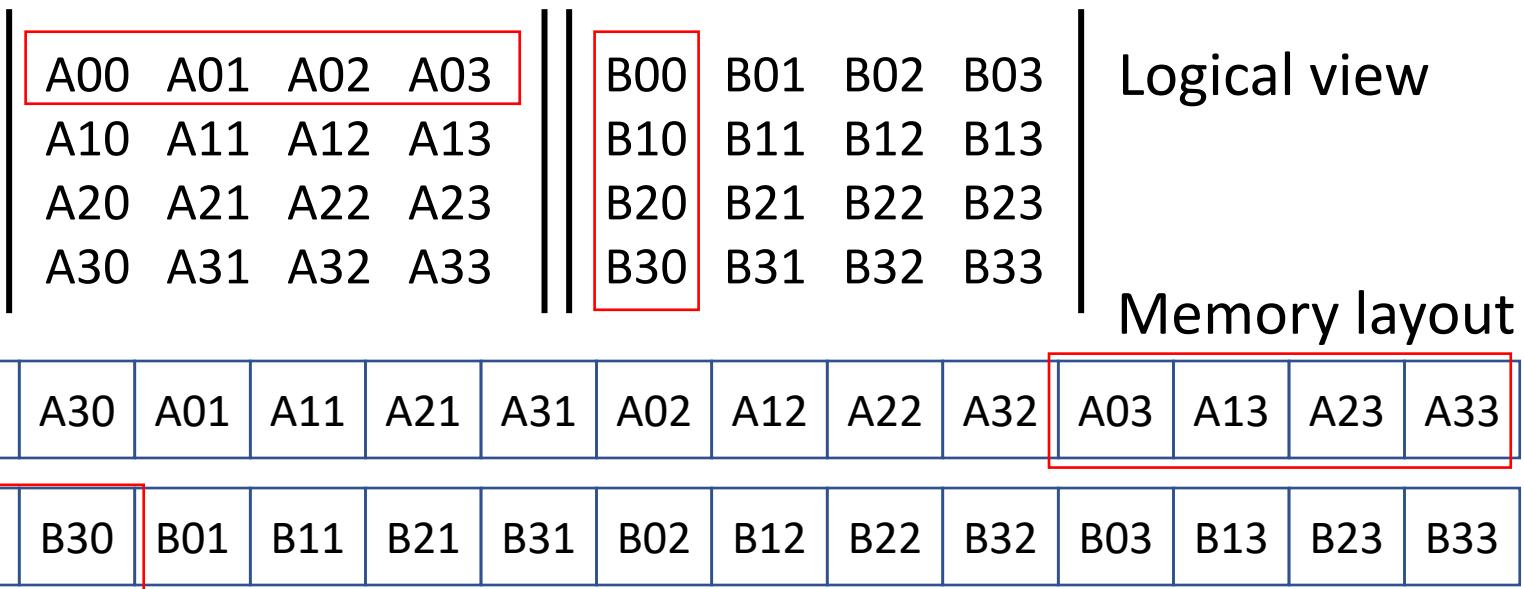
Processor



Processor can read only
 $\{A02, B10\}$ from fast mem
 $C00=C00+A02*B20$

Cache (fast but small)

Compute $C = C + A^*B$



Compute C00 = A00*B00+A01*B10+A02*B20+A03*B30

5. ... and so on

A and B are stored
in slow memory in
column-major order

Processor

B00	B10	B20	B30
A03	A13	A23	A33

Cache (fast but small)

Always cache miss for A[]

Observation:

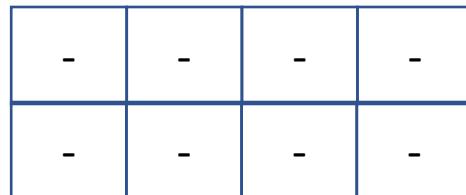
Processor cannot exploit spatial locality

What if A and B are in row-major order?

A00	A01	A02	A03	B00	B01	B02	B03		Logical view						
A10	A11	A12	A13	B10	B11	B12	B13								
A20	A21	A22	A23	B20	B21	B22	B23								
A30	A31	A32	A33	B30	B31	B32	B33								
									Memory layout						
A00	A01	A02	A03	A10	A11	A12	A13	A20	A21	A22	A30	A30	A31	A32	A33
B00	B01	B02	B03	B10	B11	B12	B13	B20	B21	B22	B30	B30	B31	B32	B33

Compute $C00 = A00*B00+A01*B10+A02*B20+A03*B30$

A and B are stored
in slow memory in
row-major order



Can processor exploit
spatial locality?

Cache (fast but small)

What if A and B are in row-major order? (Answer)

A00	A01	A02	A03	B00	B01	B02	B03		Logical view
A10	A11	A12	A13	B10	B11	B12	B13		
A20	A21	A22	A23	B20	B21	B22	B23		
A30	A31	A32	A33	B30	B31	B32	B33		

A00	A01	A02	A03	A10	A11	A12	A13	A20	A21	A22	A30	A30	A31	A32	A33
B00	B01	B02	B03	B10	B11	B12	B13	B20	B21	B22	B30	B30	B31	B32	B33

Compute $C00 = A00*B00+A01*B10+A02*B20+A03*B30$

A and B are stored
in slow memory in
row-major order



-	-	-	-
-	-	-	-

Can processor exploit
spatial locality?
Always cache miss for B[]

Cache (fast but small)

Column-major order: #comp, #comm

Logical view				Memory layout											
A00	A10	A20	A30	A01	A11	A21	A31	A02	A12	A22	A32	A03	A13	A23	A33
B00	B10	B20	B30	B01	B11	B21	B31	B02	B12	B22	B32	B03	B13	B23	B33

To compute $C_{00} = A_{00}*B_{00}+A_{01}*B_{10}+A_{02}*B_{20}+A_{03}*B_{30}$

$$\begin{aligned}
 m_{C00} &= \text{read } B(, 0) \text{ once: } \#n \text{ memory access} \\
 &\quad + \text{read entire } A(,): \#n^2 \text{ memory access (due to cache misses)} \\
 &\quad + \text{read/write } C_{00} \text{ once: } \#2 \text{ memory access} \\
 &= (n^2+n+2)
 \end{aligned}$$

$$f_{C00} = n \text{ multiplications} + n \text{ additions} = 2n$$

There are n^2 elements in matrix multiplication result, so total is

$$m_{\text{total}} = n^2(n^2+n+2) \approx O(n^4) \quad f_{\text{total}} = 2n^3 \rightarrow q = f_{\text{total}}/m_{\text{total}} = 1/n$$

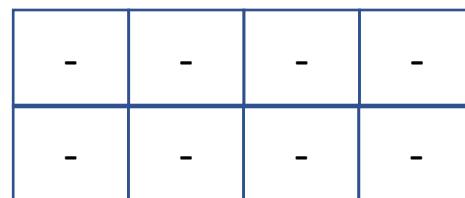
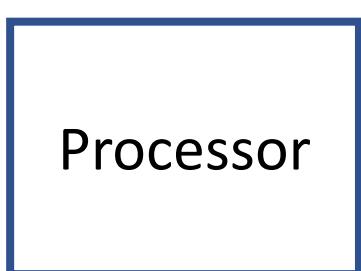
Better approach

A(,) in row-major and B(,) in column-major orders

A00 A01 A02 A03	B00 B01 B02 B03	Logical view
A10 A11 A12 A13	B10 B11 B12 B13	
A20 A21 A22 A23	B20 B21 B22 B23	
A30 A31 A32 A33	B30 B31 B32 B33	

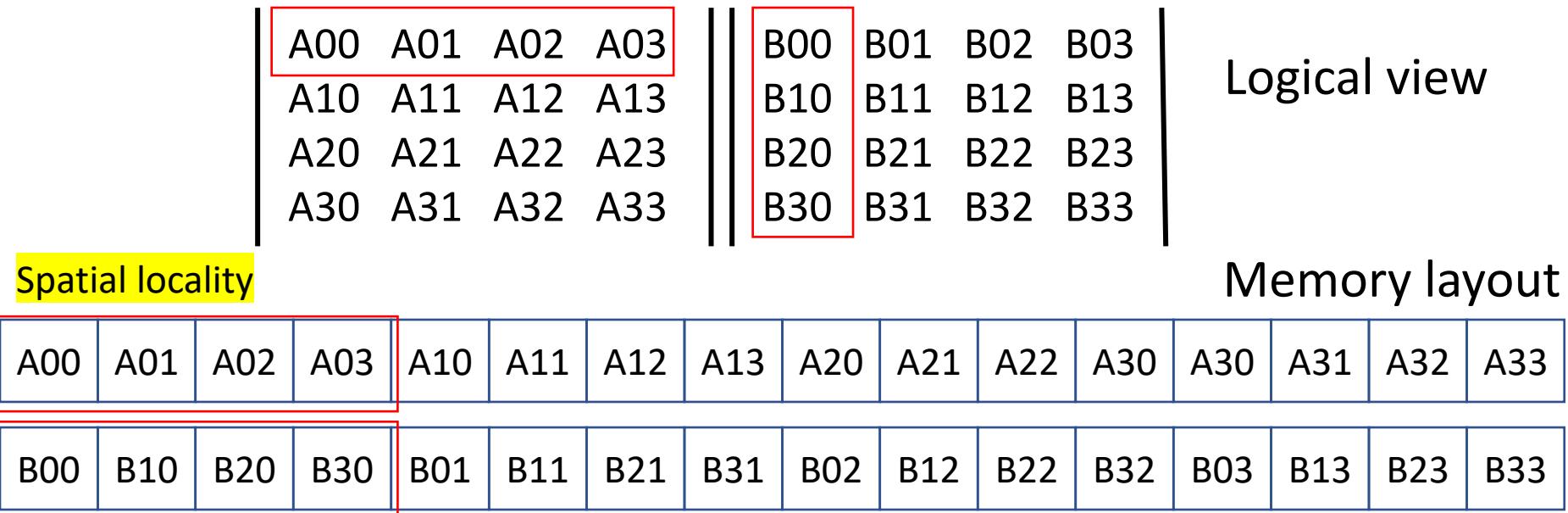
Memory layout

A00	A01	A02	A03	A10	A11	A12	A13	A20	A21	A22	A30	A30	A31	A32	A33
B00	B10	B20	B30	B01	B11	B21	B31	B02	B12	B22	B32	B03	B13	B23	B33



Cache (fast but small)

A(,) in row-major and B(,) in column-major orders



Compute $C00 = A00 * B00 + A01 * B10 + A02 * B20 + A03 * B30$

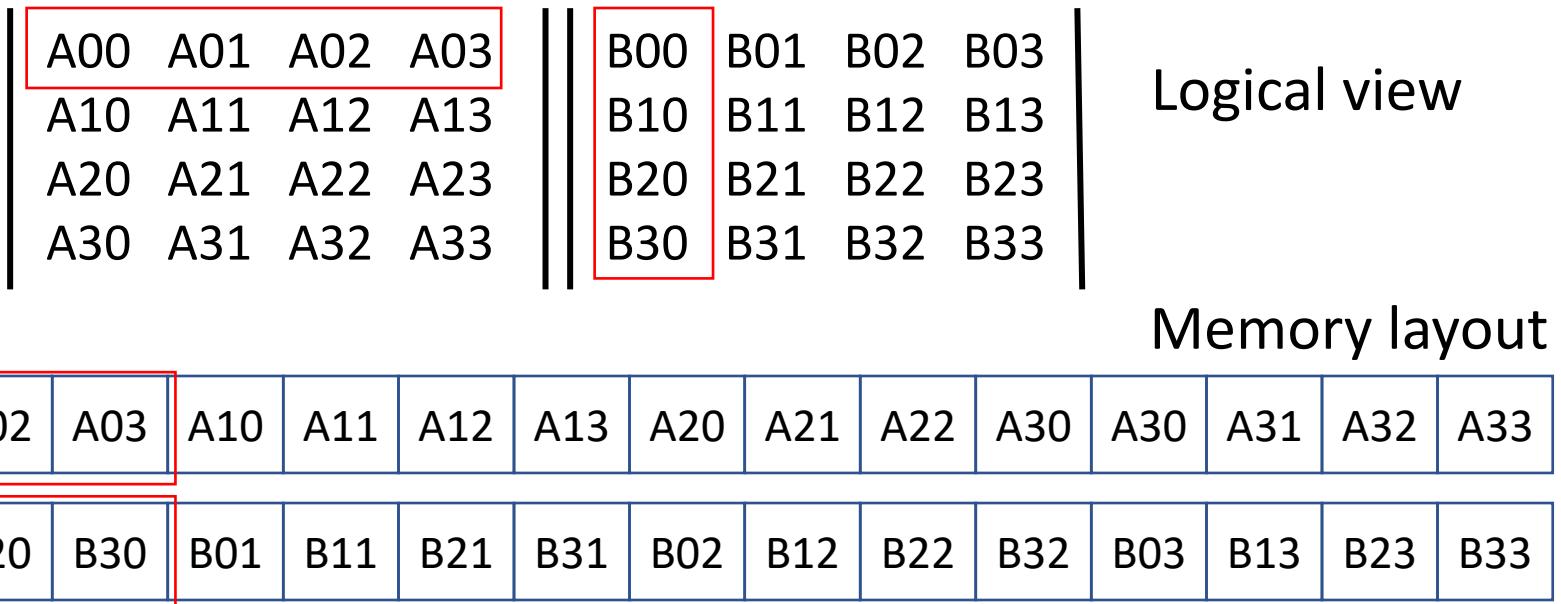
1. Read A00 into fast memory
2. Read B00 into fast memory



B00	B10	B20	B30
A00	A01	A02	A03

Cache (fast but small)

A(,) in row-major and B(,) in column-major orders



Compute $C00 = A00*B00+A01*B10+A02*B20+A03*B30$

3. Compute $C00=A00*B00$
4. Compute $C00=C00+A01*A02$
5. and the rest



B00	B10	B20	B30
A00	A01	A02	A03

Cache (fast but small)

Processor finds all required elements in cache 😊
[Very few cache miss]

[Row]-[Column] major orders: #Comm and #Comp

A00	A01	A02	A03		B00	B01	B02	B03		Logical view
A10	A11	A12	A13		B10	B11	B12	B13		
A20	A21	A22	A23		B20	B21	B22	B23		
A30	A31	A32	A33		B30	B31	B32	B33		Memory layout
02	A03	A10	A11	A12	A13	A20	A21	A22	A30	
20	B30	B01	B11	B21	B31	B02	B12	B22	B32	
						B03	B13	B23	B33	

To compute $C_{00} = A_{00} * B_{00} + A_{01} * B_{10} + A_{02} * B_{20} + A_{03} * B_{30}$

$$\begin{aligned}
 m_{C00} &= \text{read } A(0,) \text{ once: } \#n \text{ memory access} \\
 &\quad + \text{read } B(,0) \text{ once: } \#n \text{ memory access} \\
 &\quad + \text{read/write } C00 \text{ once: } \#2 \text{ memory access} \\
 &= (2n+2)
 \end{aligned}$$

$$f_{Coo} = n \text{ multiplications} + n \text{ additions} = 2n$$

There are n^2 elements in matrix multiplication result, so total is

$$m_{\text{total}} = n^2(2n+2) \approx O(2n^3) \quad f_{\text{total}} = 2n^3 \quad \rightarrow \quad q = f_{\text{total}} / m_{\text{total}} \approx 1$$

+ exploit temporal locality of $A(i, \cdot)$

[Row]-[Column] major orders: #Comm and #Comp

A00	A01	A02	A03	B00	B01	B02	B03
A10	A11	A12	A13	B10	B11	B12	B13
A20	A21	A22	A23	B20	B21	B22	B23
A30	A31	A32	A33	B30	B31	B32	B33

$$\text{Compute } C_{00} = A_{00} * B_{00} + A_{01} * B_{10} + A_{02} * B_{20} + A_{03} * B_{30}$$

$$C_{01} = A_{00} * B_{01} + A_{01} * B_{11} + A_{02} * B_{21} + A_{03} * B_{31}$$

$$C_{02} = A_{00} * B_{02} + A_{01} * B_{12} + A_{02} * B_{22} + A_{03} * B_{32}$$

$$C_{03} = A_{00} * B_{03} + A_{01} * B_{13} + A_{02} * B_{23} + A_{03} * B_{33}$$

Compute one row of C

$$\begin{aligned} m_{C(0, \cdot)} &= \text{read } A(0, \cdot) \text{ once: } \#n \text{ memory access} \\ &\quad + \text{read } B(\cdot, i) \text{ } n \text{ times: } \#n^2 \text{ memory access} \\ &\quad + \text{read/write } C(0, \cdot) \text{ once: } \#2n \text{ memory access} \\ &= (n^2 + 3n) \end{aligned}$$

$$f_{C00} = n^2 \text{ multiplications} + n^2 \text{ additions} = 2n^2$$

Total cost for matrix multiplication

$$m_{\text{total}} = n(n^2 + 3n) \approx O(n^3) \quad f_{\text{total}} = 2n^3 \rightarrow q = f_{\text{total}}/m_{\text{total}} = 2$$

even better!

What assumptions did we make?

A00	A01	A02	A03	B00	B01	B02	B03
A10	A11	A12	A13	B10	B11	B12	B13
A20	A21	A22	A23	B20	B21	B22	B23
A30	A31	A32	A33	B30	B31	B32	B33

1. Cache memory is sufficiently large to store one row of A
2. and one column of B
3. Cache memory access has 0 overhead

In a real system, the advantage will be smaller compared to our hypothetical machine

Achieving $q > 2$

Partition matrix into blocks

$$\begin{vmatrix} A_{00} & A_{01} \\ A_{10} & A_{11} \end{vmatrix} \leftarrow \begin{vmatrix} A_{00} & A_{01} & A_{02} & A_{03} \\ A_{10} & A_{11} & A_{12} & A_{13} \\ A_{20} & A_{21} & A_{22} & A_{23} \\ A_{30} & A_{31} & A_{32} & A_{33} \end{vmatrix}$$

where

$$A_{00} = \begin{vmatrix} A_{00} & A_{01} \\ A_{10} & A_{11} \end{vmatrix}$$

$$A_{01} = \begin{vmatrix} A_{02} & A_{03} \\ A_{12} & A_{13} \end{vmatrix}$$

etc.

Bold font is used to represent a sub-matrix.

Multiplication after partitioning

A00 A01	A02 A03	B00 B01	B02 B03
A10 A11	A12 A13	B10 B11	B12 B13
A20 A21	A22 A23	B20 B21	B22 B23
A30 A31	A32 A33	B30 B31	B32 B33

$$\begin{vmatrix} A00 & A01 \\ A10 & A11 \end{vmatrix}$$

$$\begin{vmatrix} B00 & B01 \\ B10 & B11 \end{vmatrix}$$

Then matrix multiplication result is

$$\begin{vmatrix} C00 & C01 \\ C10 & C11 \end{vmatrix} = \begin{vmatrix} A00*B00+A01*B10 & A00*B01+A01*B11 \\ A10*B00+A11*B10 & A10*B01+A11*B11 \end{vmatrix}$$

Multiplication after partitioning

A00 A01	A02 A03	B00 B01	B02 B03
A10 A11	A12 A13	B10 B11	B12 B13
A20 A21	A22 A23	B20 B21	B22 B23
A30 A31	A32 A33	B30 B31	B32 B33

$$\begin{array}{|c|c|} \hline A00 & A01 \\ \hline A10 & A11 \\ \hline \end{array} \quad \begin{array}{|c|c|} \hline B00 & B01 \\ \hline B10 & B11 \\ \hline \end{array}$$
$$\begin{array}{|c|c|} \hline C00 & C01 \\ \hline C10 & C11 \\ \hline \end{array} = \begin{array}{|c|c|} \hline A00*B00+A01*B10 & A00*B01+A01*B11 \\ \hline A10*B00+A11*B10 & A10*B01+A11*B11 \\ \hline \end{array}$$

Assume we can fit three such sub-matrices in the cache.



Processor

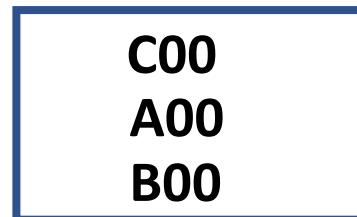


Cache (fast but small)

Multiplication after partitioning

$$\begin{array}{|c|c|c|c|} \hline A_{00} & A_{01} & A_{02} & A_{03} \\ \hline A_{10} & A_{11} & A_{12} & A_{13} \\ \hline A_{20} & A_{21} & A_{22} & A_{23} \\ \hline A_{30} & A_{31} & A_{32} & A_{33} \\ \hline \end{array} \quad \begin{array}{|c|c|c|c|} \hline B_{00} & B_{01} & B_{02} & B_{03} \\ \hline B_{10} & B_{11} & B_{12} & B_{13} \\ \hline B_{20} & B_{21} & B_{22} & B_{23} \\ \hline B_{30} & B_{31} & B_{32} & B_{33} \\ \hline \end{array}$$
$$\begin{array}{|c|c|} \hline A_{00} & A_{01} \\ \hline A_{10} & A_{11} \\ \hline \end{array} \quad \begin{array}{|c|c|} \hline B_{00} & B_{01} \\ \hline B_{10} & B_{11} \\ \hline \end{array}$$
$$\begin{array}{|c|c|} \hline C_{00} & C_{01} \\ \hline C_{10} & C_{11} \\ \hline \end{array} = \begin{array}{|c|c|} \hline A_{00}*B_{00}+A_{01}*B_{10} & A_{00}*B_{01}+A_{01}*B_{11} \\ \hline A_{10}*B_{00}+A_{11}*B_{10} & A_{10}*B_{01}+A_{11}*B_{11} \\ \hline \end{array}$$

Assume we can fit three such sub-matrices in the cache.



Cache (fast but small)

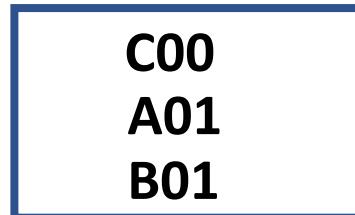
We can compute **entire** sub-matrix operation smoothly!
 $C00 = A00 * B00$

Multiplication after partitioning

A00	A01	A02	A03	B00	B01	B02	B03
A10	A11	A12	A13	B10	B11	B12	B13
A20	A21	A22	A23	B20	B21	B22	B23
A30	A31	A32	A33	B30	B31	B32	B33

$$\begin{array}{|c|c|} \hline A00 & \boxed{A01} \\ \hline A10 & A11 \\ \hline \end{array} \quad \begin{array}{|c|c|} \hline B00 & B01 \\ \hline B10 & \boxed{B11} \\ \hline \end{array}$$
$$\begin{array}{|c|c|} \hline \boxed{C00} & C01 \\ \hline C10 & C11 \\ \hline \end{array} = \begin{array}{|c|c|} \hline A00*B00+A01*B10 & A00*B01+A01*B11 \\ \hline A10*B00+A11*B10 & A10*B01+A11*B11 \\ \hline \end{array}$$

Assume we can fit three such sub-matrices in the cache.



Cache (fast but small)

We can compute **entire** sub-matrix operation smoothly!
 $C00 = C00 + A01 * B10$

Multiplication after partitioning

A00 A01	A02 A03	B00 B01	B02 B03
A10 A11	A12 A13	B10 B11	B12 B13
A20 A21	A22 A23	B20 B21	B22 B23
A30 A31	A32 A33	B30 B31	B32 B33

$$\begin{vmatrix} A00 & A01 \\ A10 & A11 \end{vmatrix}$$

$$\begin{vmatrix} B00 & B01 \\ B10 & B11 \end{vmatrix}$$

$$\begin{vmatrix} C00 & C01 \\ C10 & C11 \end{vmatrix} = \begin{vmatrix} A00*B00+A01*B10 & A00*B01+A01*B11 \\ A10*B00+A11*B10 & A10*B01+A11*B11 \end{vmatrix}$$

What can we do if these sub-matrices do not fit in Cache?

Answer: partition into multiple smaller sub-matrices such that they can be placed in the cache.

(this is the idea behind Tiled Matrix Multiplication)

Tiled matrix multiplication

Consider A,B,C to be N-by-N matrices of b-by-b subblocks where

b=n / N is called the **block size**

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

```
    for(j = 0; j<N; j++)
```

{read block C(i,j) into fast memory}

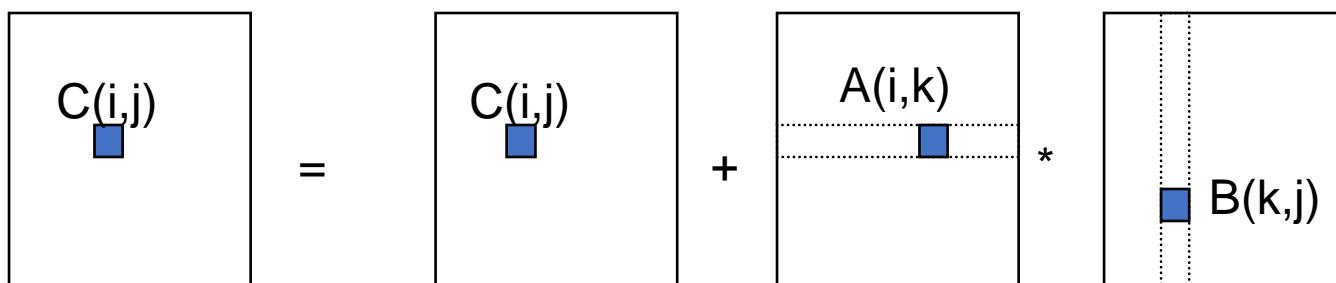
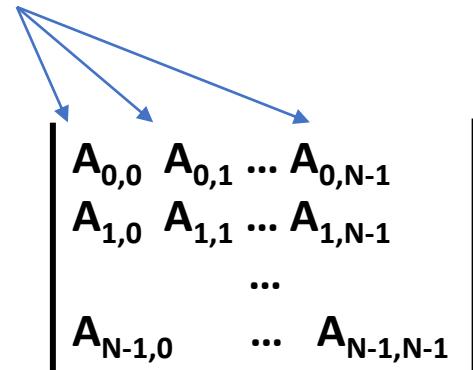
```
    for(k = 1; k<N; k++)
```

{read block A(i,k) into fast memory}

{read block B(k,j) into fast memory}

$$C(i,j) = C(i,j) + A(i,k) * B(k,j) \{ \text{do a matrix multiply on blocks} \}$$

{write block C(i,j) back to slow memory}



Tiled matrix multiplication: #Comm and #Comp

$$\begin{aligned}m &= N^2 n^2 \quad \text{read blocks of } B \ N^3 \text{ times } (N^3 * b^2 = N^3 * (n/N)^2 = N^2 n^2) \\&\quad + N^2 n^2 \quad \text{read blocks of } A \ N^3 \text{ times} \\&\quad + 2n^2 \quad \text{read and write each block of } C \text{ once} \\&= (2N + 2) * n^2\end{aligned}$$

$$f = 2n^3$$

So computational intensity $q = f / m = 2n^3 / ((2N + 2) * n^2)$
 $\approx n / N = b$ for large n

- By choosing $b > 2$ we can achieve $q > 2$
- Additionally q is proportional to block size b

Advantages of tiled matrix multiplication

Computational intensity $q = f / m \approx b$ for large n

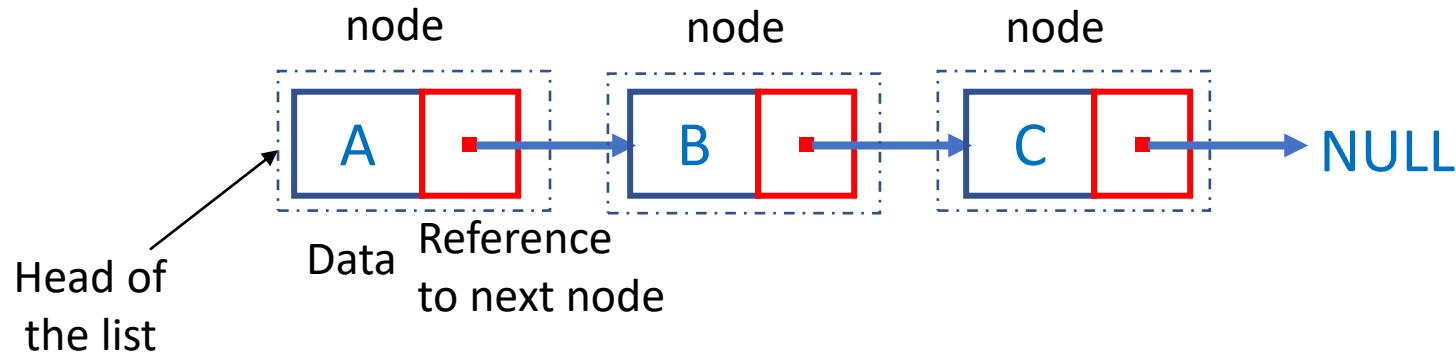
Advantages:

- Enhancement in computational intensity
- Flexibility depending on size of fast memory in machine
(choose block size b according to system cache)

Linked List (Recap from last week)

A ‘linked list’ is a

- linear collection of data elements called ‘nodes’
- each node points to the next node in the list
- unlike arrays, linked list nodes are not stored at contiguous locations; they are linked using pointers as shown below.



**Can we have a cache-efficient linked-list?
(or any other data structures)
Try yourself**

Void Pointer

Sujoy Sinha Roy
School of Computer Science
University of Birmingham

Void pointer

- A void pointer in C has no associated data type.
- It can store the address of any type of object
- '*Generic pointer*'
- It can be type-casted to any types.

Syntax for declaration

```
void *pointer_name;
```

Void pointer and reusability

- Most important feature of the void pointer is reusability.
- We can store the address of **any** object
- Whenever required we can typecast it to a required type

Void pointer example

```
int main()
{
    void *pv;
    int iData = 5;
    char cData = 'C';

    //Pointer to char
    pv = &cData;

    //Dereferencing void pointer with char typecasting
    printf("cData = %c\n\n",*((char*)pv));

    //Pointer to int
    pv = &iData;

    //Dereferencing void pointer with int typecasting
    printf("iData = %d\n\n",*((int *)pv));

    return 0;
}
```

The same pointer is reused for multiple data types.
Type must be specified while dereferencing.

Arithmetic on void pointer

```
#include<stdio.h>

int main()
{
    int a[4] = {1, 5, 13, 4};
    void *pv = &a[0];
    pv = pv + 1;

    printf("Value %d\n", *((int *) pv));
}

return 0;
```

What will be the output?

Arithmetic on void pointer

```
#include<stdio.h>

int main()
{
    int a[4] = {1, 5, 13, 4};
    void *pv = &a[0];
    pv = pv + 1;

    printf("Value %d\n", *((int *) pv) );
}

return 0;
}
```

What will be the output?

It will not print 5

pv+1 does not increment pv by scale_factor=4

Arithmetic on void pointer

Perform proper typecasting on the void pointer before performing arithmetic operation.

```
#include<stdio.h>

int main()
{
    int a[4] = {1, 5, 13, 4};
    void *pv = &a[0];
    pv = (int *) pv + 1;

    printf("Value %d\n", *((int *) pv));
}

return 0;
```

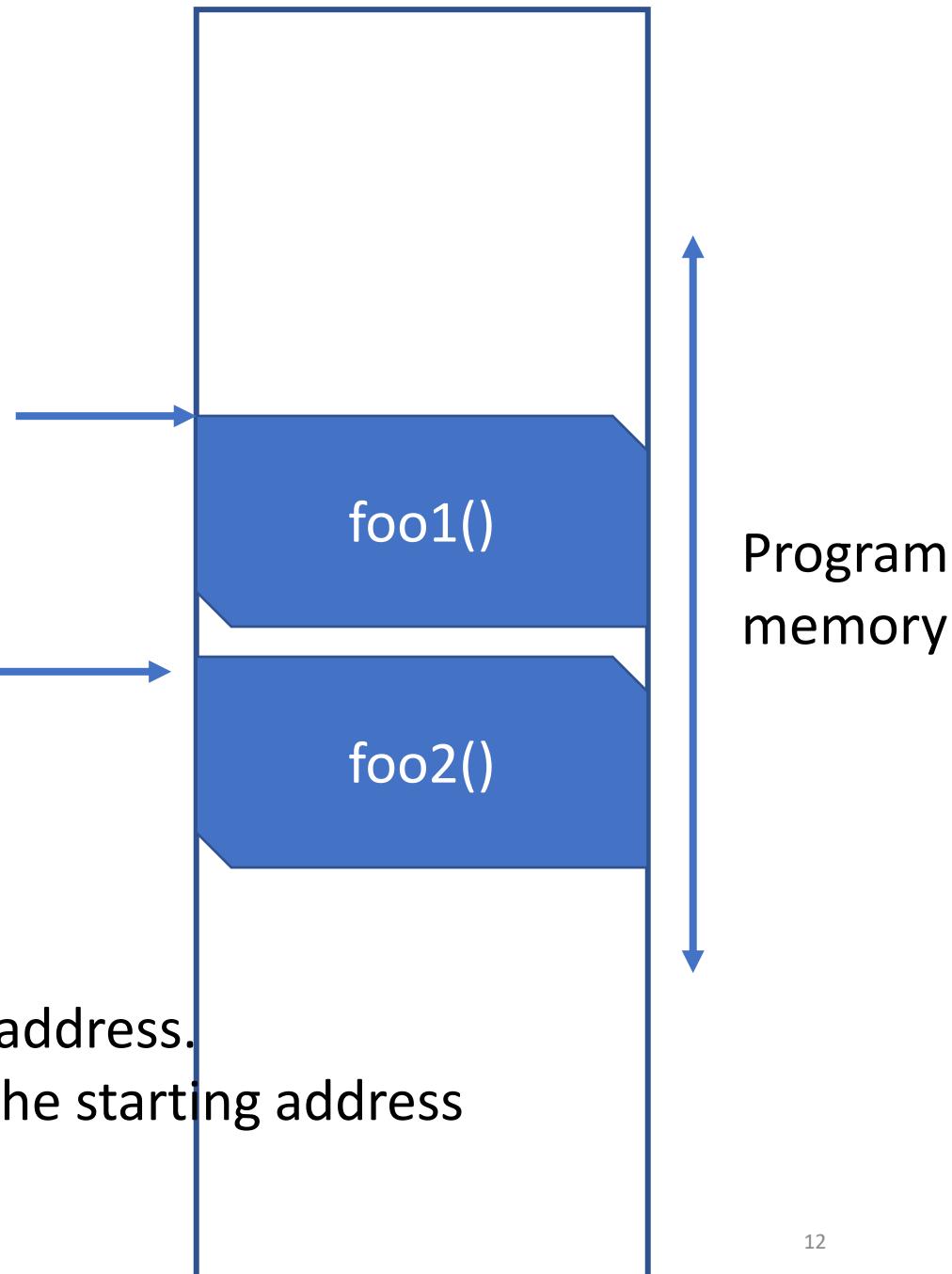
Now it prints 5

During `pv+1` compiler increments `pv` by scale_factor=4

Function pointers

Start address of foo1()

Start address of foo2()



- Every function has a memory address.
- A pointer to a function holds the starting address

Function pointer syntax

Syntax for declaration

```
int (*foo)(int);
```

- foo is a pointer to a function
- Where function takes one int argument and returns int.

foo

```
int negate(int a);
int square(int c);
...
```

Foo can point to any of these functions

Function pointer syntax: careful

Function pointer declaration

```
int (*foo)(int);
```

Here function returns pointer of type int

```
int *foo(int);
```

To declare a function pointer () must be used

Function pointer syntax

What is the meaning of this syntax?

```
int * (*foo) (int);
```

- foo is a pointer to a function
- Where function takes one int argument and **returns pointer to int.**

foo

```
int *negate(int a);
int *square(int c);
...
```

Foo can point to any of these functions

Initialization of function pointer

```
void int_func(int a)
{
    printf("%d\n", a);
}

int main()
{
    void (*foo)(int);

        // & is optional
    foo = &int_func;

    return 0;
}
```

Calling function using function pointer

```
void int_func(int a)
{
    printf("%d\n", a);
}

int main()
{
    void (*foo)(int);

    // & is optional
    foo = &int_func;
    // two ways to call
    foo(2);
    (*foo)(3);
    return 0;
}
```

Other standard input/output functions in C

Standard Input/Output functions in C

So far, we have extensively used two input/output functions

```
scanf ("%format", &variable_name);  
printf ("%format", variable_name);
```

There are several other functions in C to perform input/output operations.

Standard Input/Output functions in C: getchar() and putchar()

- **getchar()** gets a character from standard input.
- **putchar()** writes a character to standard output.

```
int main () {
    char c;

    printf("Enter a character: ");
    c = getchar();

    printf("Character entered: ");
    putchar(c);

    return(0);
}
```

Program output
Enter a character: B
Character entered: B

Standard Input/Output functions in C: gets() and puts()

- **gets ()** reads a newline-terminated string from standard input and writes into the specified string.
- **puts ()** writes the string a trailing newline to standard output.

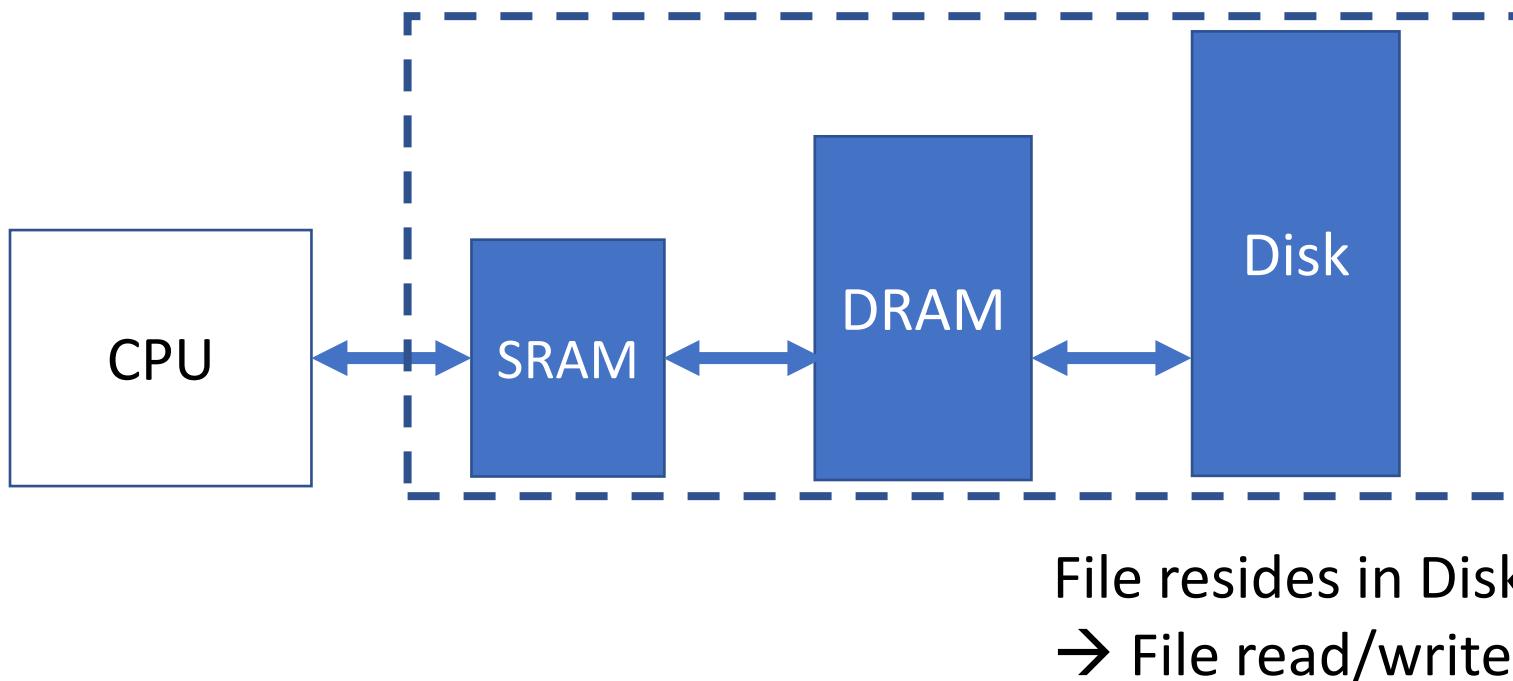
```
int main(){  
    // char array of length 100  
    char str[100];  
    printf("Enter a string: ");  
    gets( str );  
    puts( str );  
    return 0;  
}
```

Program output
Enter a string: ABC DEF
ABC DEF

File handling in C

Use of files in program

- Large data volumes
- E.g. data from statistics, experiments, human genome, population records etc.



Opening file from a C program

- For opening a file, `fopen()` function is used with the required access modes.

```
FILE *fp; /*variable fp is pointer to type FILE*/  
  
fp = fopen("filename", "mode");  
/*opens file with name filename , assigns identifier to fp */
```

`fopen()` returns `NULL`, if it is unable to open the specified file.

- File pointer `fp` points to the ‘file’ resource
 - contains all information about file
 - Communication link between system and program
- An opened file is closed by passing the file pointer to `fclose()`

```
fclose(fp);
```

Different modes

- Reading mode (**r**)
 - if the file already exists then it is opened as *read-only*
 - sets up a pointer which points to the first character in it.
 - else error occurs.
- Writing mode (**w**)
 - if the file already exists then it is *overwritten* by a new file
 - else a new file with specified name created
- Appending mode (**a**)
 - if the file already exists then it is opened
 - else new file created
 - sets up a pointer that points to the last character in it
 - any new content is appended after existing content

Additional modes

- r+ opens file for both reading/writing an *existing* file
 - doesn't delete the content of if the *existing* file
- w+ opens file for reading and writing a *new* file
 - Overwrites if the specified file already exists
- a+ open file for reading and writing from the last character in the file

Functions for reading or writing: getc() and putc()

There are several functions to read from and write to a file.

- `getc()` – read a `char` from a file.
- `putc()` – write a `char` to a file

Functions for reading or writing: getc() and putc()

```
int main(){
    char ch;
    FILE *fp0, *fp1;
    fp0 = fopen("infile.txt", "r");
    fp1 = fopen("outfile.txt", "w");
    if (fp0==NULL || fp1==NULL){
        printf("Cannot open files\n");
        exit(-1);
    }

    ch = getc(fp0); // reads one char from first file
    while (ch != EOF){ // EOF is 'end of file'
        printf("%c", ch); // Displays on screen
        putc(ch, fp1); // writes to second file
        ch = getc(fp0); // reads another char from first file
    }
    fclose(fp0);
    fclose(fp1);
    return 0;
}
```

Functions for reading or writing: `fprintf()` and `fscanf()`

- Similar to `printf()` and `scanf()`
- in addition the file pointer is provided as an input
- Examples:

To read one `int` from a file with file pointer `fp0`

```
int i;  
fp0=fopen("some_file", "r");  
fscanf(fp0, "%d", &i);
```

To write one `int` to a file with file pointer `fp1`

```
int i=4;  
fp1=fopen("some_file", "w");  
fprintf(f1, "%d", i);
```

Functions for reading or writing: fprintf() and fscanf()

```
int main(){
    char ch;
    FILE *fp0, *fp1;
    fp0 = fopen("infile.txt", "r");
    fp1 = fopen("outfile.txt", "w");
    if (fp0==NULL || fp1==NULL){
        printf("Cannot open files\n");
        exit(-1);
    }

    fscanf(fp0, "%c", &ch); // reads one char from first file
    while (ch != EOF){ // EOF is 'end of file'
        printf("%c", ch); // Displays on screen
        fprintf(fp1, "%c", ch); // writes to the second file
        fscanf(fp0, "%c", &ch); // reads another char from first file
    }
    fclose(fp0);
    fclose(fp1);
    return 0;
}
```

Typical file errors

Typically, errors happen when a program

- tries to read beyond end-of-file (EOF)
- tries to use a file that has not been opened
- performs operation on file not permitted by ‘fopen’ mode

Example:

```
fp=fopen("filename", "r");  
...  
fprintf(fp, "%d", i);
```

- opens file with invalid filename
 - writes to write-protected file
- Example: files with read-only permission

Handling these errors

Programmer can perform the following checks

- `feof(fp)` returns a non-zero value when End-of-File is reached, else it returns zero

```
fp = fopen("somefile", "somemode");
...
if(feof(fp)){
    printf("End of file\n");
}
```

- `ferror(fp)` returns nonzero value if error detected else returns zero

```
fp = fopen("somefile", "somemode");
...
if(ferror(fp) !=0)
    printf("An error has occurred\n");
```

Random access to file [Optional slide]

Data in a file is basically a collection of bytes.

We can ***directly*** jump to a target byte-number in a file without reading previous data using fseek()

- Syntax: `fseek(file-pointer, offset, position);`
- position: 0 (beginning), 1 (current), 2 (end)
- offset: number of locations to move from specified position

Examples:

```
fseek(fp, -m, 1); // move back by m bytes from current  
fseek(fp, m, 0); // move to (m+1)th byte in file
```

- `ftell(fp)` returns current byte position in file
- `rewind(fp)` resets position to start of file

Random access to file [Optional slide]

```
int main () {  
    FILE *fp;  
  
    fp = fopen("file.txt","w+");  
    fprintf(fp,"%s", "This is something");  
  
    fseek( fp, 7, 0);  
    fprintf(fp,"%s", " C Language");  
    fclose(fp);  
  
    return(0);  
}
```

The program

1. writes “This is something”
 2. then moves to 7 byte-positions after beginning (i.e., 8th position)
 3. writes " C Language" (overwriting any data that exists)
- Thus, the final content is “This is C Language”

Command line arguments in C

Command line arguments in C

We can modify a program to receive arguments from command line.

Example

```
./a.out string1 string2 string3
```

Syntax is

```
int main(int argc, char *argv[]){
```

```
    ...
```

```
}
```

- argc (Argument Count) is **int** and *automatically* stores the number of command-line arguments passed by the user including name of program.
- argv(Argument Vector) is array of **char** pointers listing all the arguments.
 argv[0] is the name of the program
 argv[1] is the first command-line argument etc.

Command line arguments in C

```
int main(int argc, char *argv[]){
    int i;
    printf("You have entered %d arguments:\n", argc);

    for (int i = 0; i < argc; ++i)
        printf("%s\n", argv[i]);

    return 0;
}
```

```
./a.out how are you?
You have entered 4 arguments:
a.out
how
are
you?
```

Another example: Command line arguments in C

```
#include <stdio.h>
#include <stdlib.h>
#define PI 3.1416
int main (int argc, char *argv[]){
    double r, area, circ;
    char *a = argv[1];
    int diameter = atoi(a);

    if(argc>1)
        printf("You have entered %d\n",diameter);
    else{
        printf("Enter diameter:");
        scanf("%d", &diameter);
    }

    r= diameter/2;
    area = PI*r*r;
    circ= 2*PI*r;

    printf ("Circle with diameter %d\n", diameter);
    printf ("has area of %f\n", area);
    printf ("and circumference of %f\n", circ);

    return (0);
}
```

atoi() converts a string to an integer.

./a.out
Enter diameter:2
Circle with diameter 2
has area of 3.141600
and circumference of 6.283200

./a.out 2
You have entered 2
Circle with diameter 2
has area of 3.141600
and circumference of 6.283200

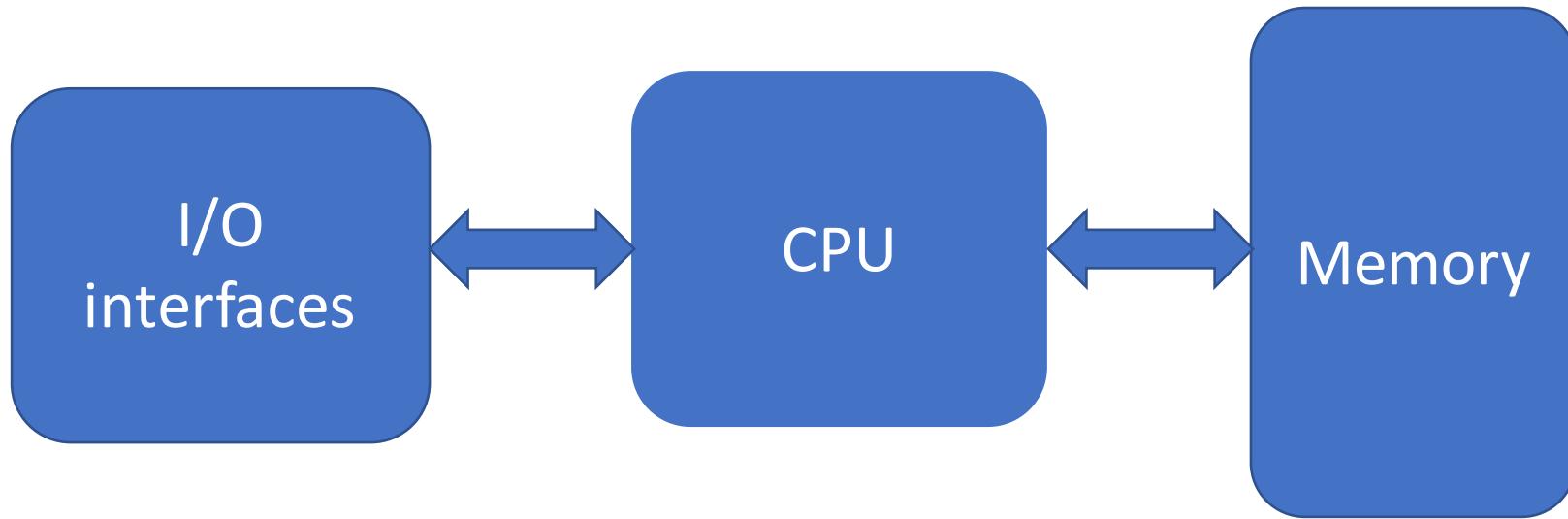
Multicore Systems

Sujoy Sinha Roy

School of Computer Science

University of Birmingham

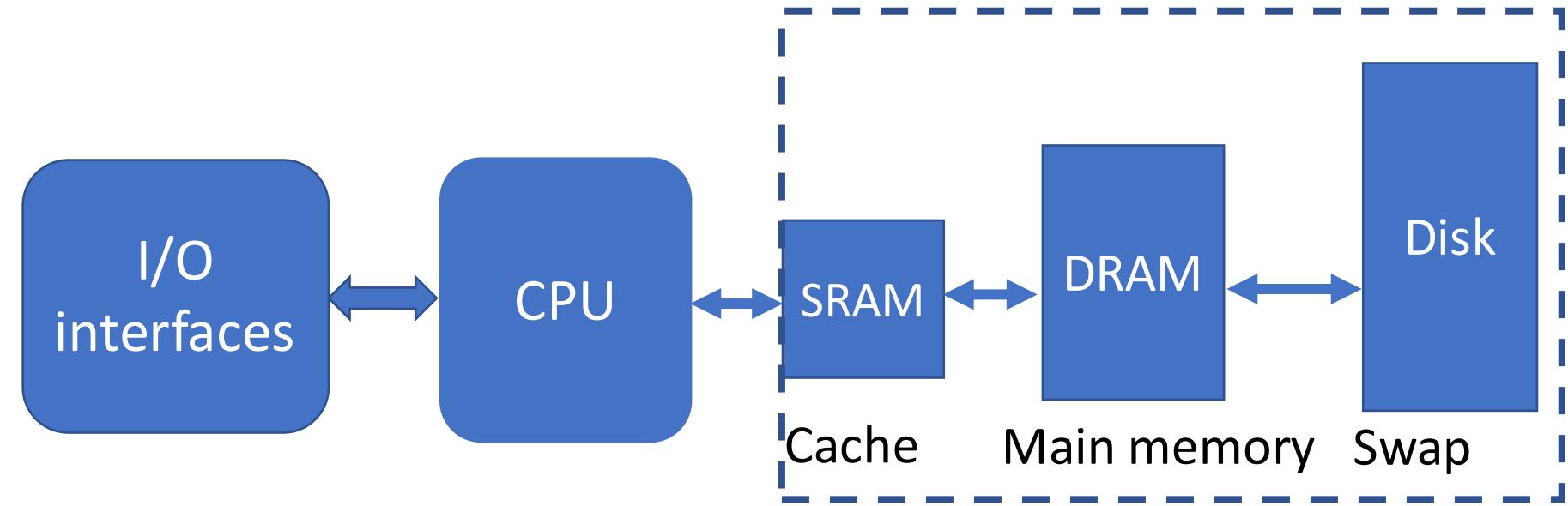
Recap: Simple von Neumann Computer



We started C programming assuming a simple von Neumann computation model.

- We executed program on a single CPU (or processor) step-by-step.
- We stored data and program instructions in the Memory.

Recap: Memory Hierarchy



We studied memory hierarchy from a programmer's perspective
→ Learnt how to write efficient C code to exploit memory hierarchy
→ Programs are still executed on a single CPU (or processor)
step-by-step but more efficiently



Produced	From May 7, 1997 to December 26, 2003 ^[1]
Common manufacturer(s)	Intel
Max. CPU clock rate	233 MHz to 450 MHz
FSB speeds	66 MHz to 100 MHz
Min. feature size	0.35 µm to 0.18 µm
Instruction set	IA-32, MMX



Produced	From February 26, 1999 to May 18, 2007 ^[1]
Common manufacturer(s)	Intel
Max. CPU clock rate	450 MHz to 1.4 GHz
FSB speeds	100 MHz to 133 MHz
Min. feature size	0.25 µm to 0.13 µm
Instruction set	IA-32, MMX, SSE

Clock frequency almost doubled w.r.t Pentium II



Produced	From November 20, 2000 to August 8, 2008
Max. CPU clock rate	1.3 GHz to 3.8 GHz
FSB speeds	400 MT/s to 1066 MT/s
Instruction set	x86 (i386), x86-64 (only some chips), MMX, SSE, SSE2, SSE3

Clock frequency almost doubled w.r.t Pentium III

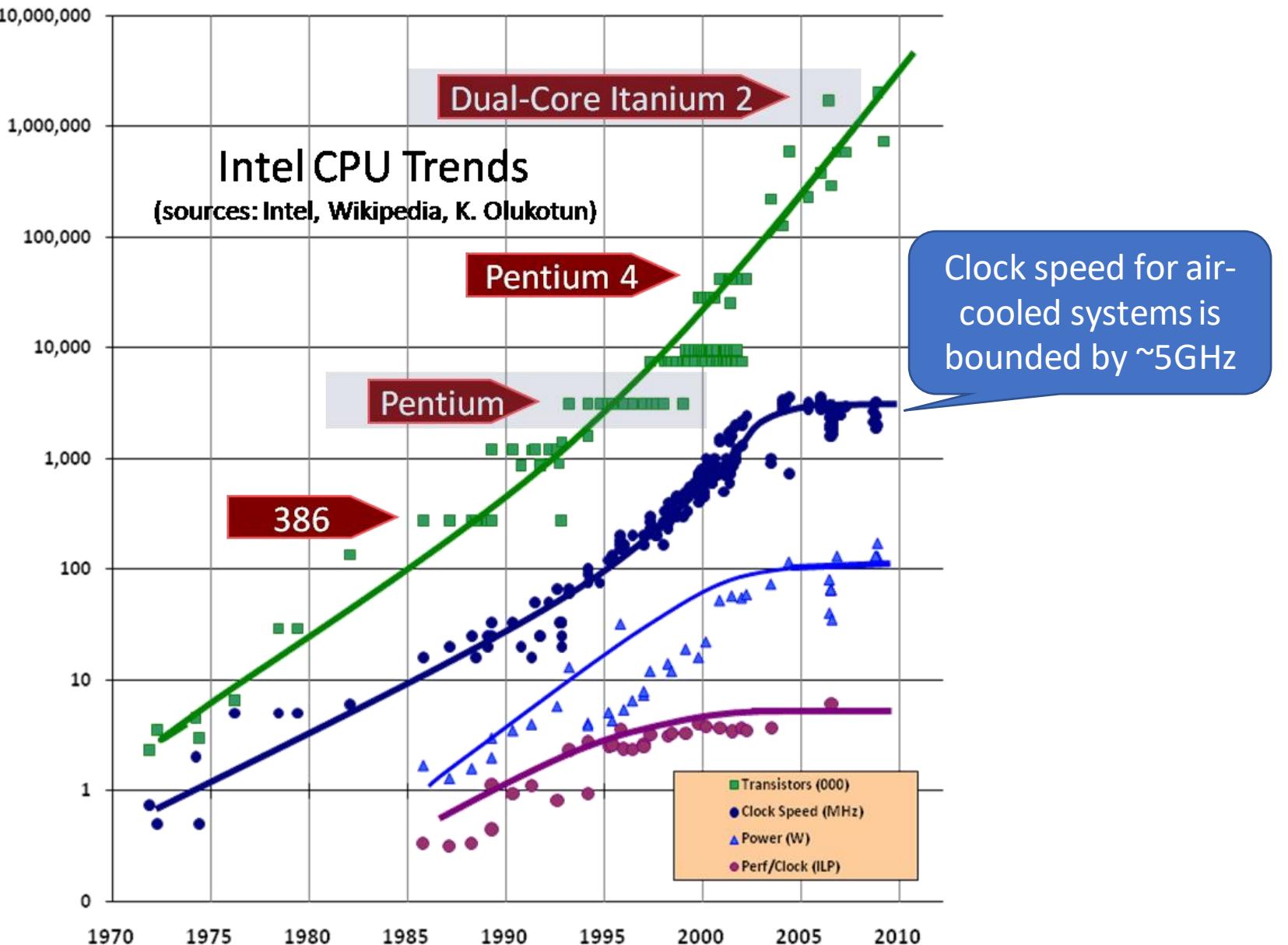
What is the clock frequency in 2020?



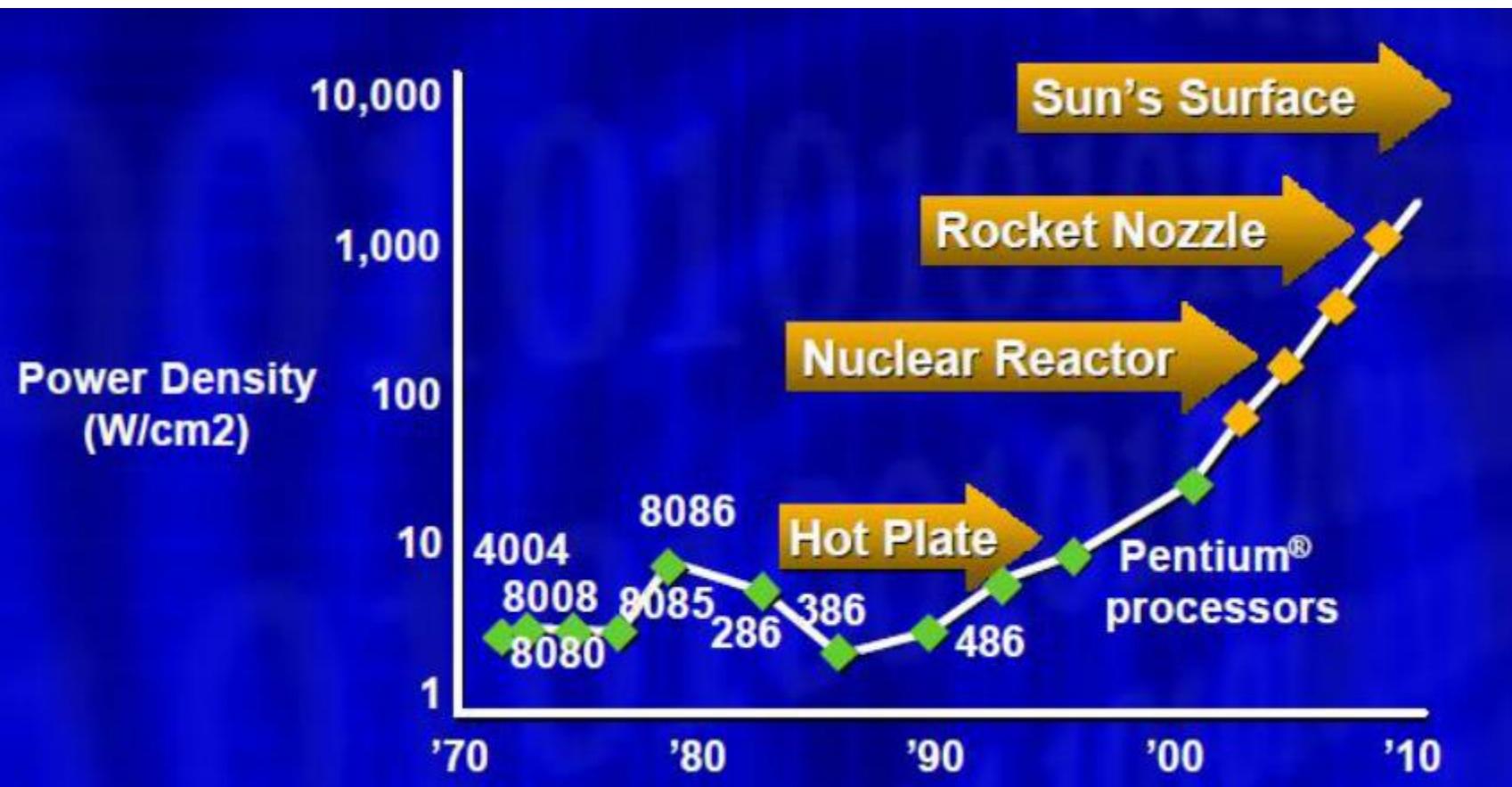
Intel® Core™ i7-10510U Processor (8M Cache, Up to 4.80 GHz)

- 8 MB Intel® Smart Cache Cache
- 4 Cores
- 8 Threads
- 4.90 GHz Max Turbo Frequency
- U - Ultra-low power
- 10th Generation

Little increase in clock frequency after 10 years since Pentium 4



Source: Herb Sutter “The Free Lunch Is Over: A Fundamental Turn Toward Concurrency in Software” available at <http://www.gotw.ca/publications/concurrency-ddj.htm>



Source: Patrick Gelsinger, Intel Developer's Forum, Intel Corporation, 2004.

Applications are demanding more resources!

Alternative solution:

- Put many processing cores on the microprocessor chip.
- The number of cores doubles with each generation.

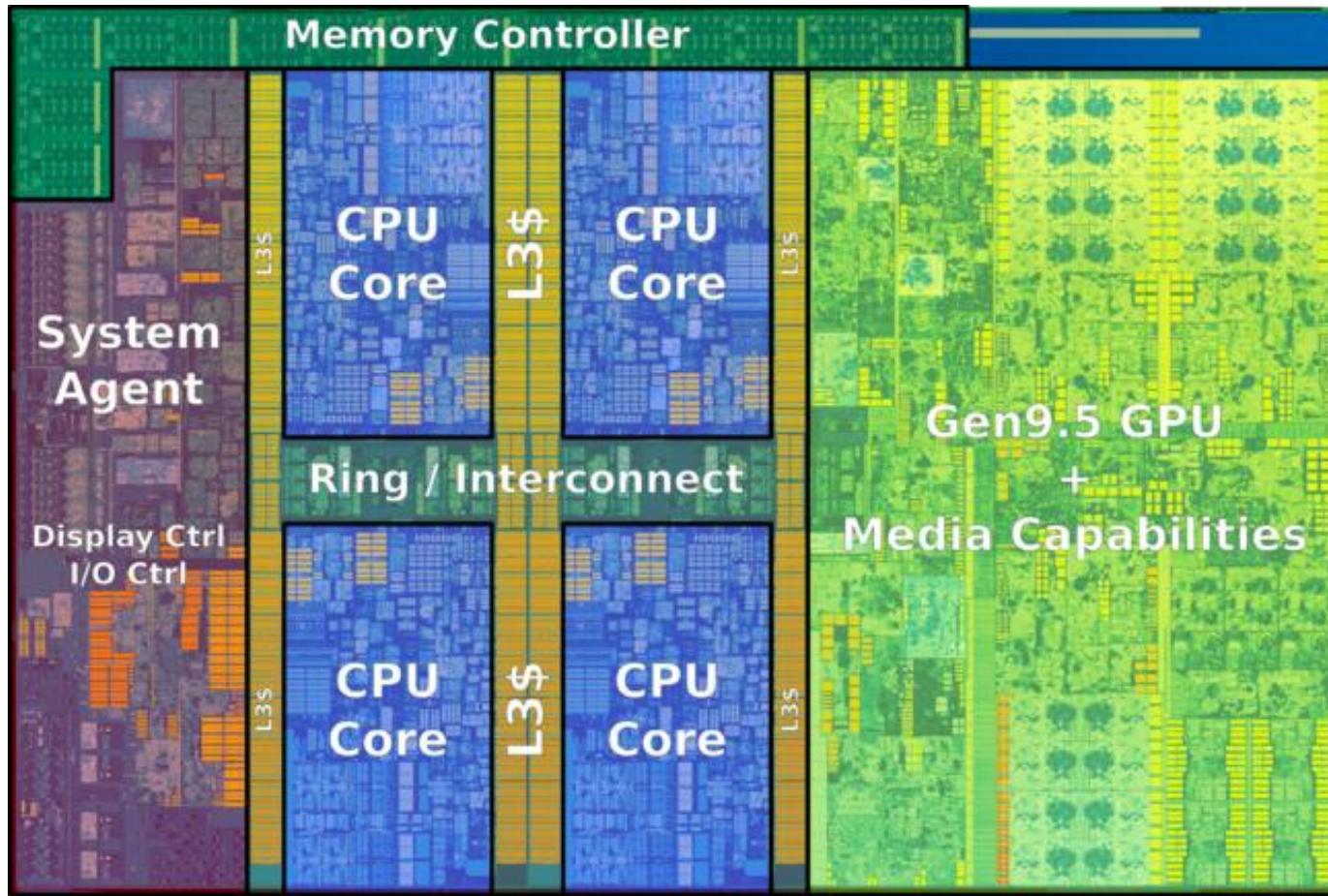


Photo of the quad-core GT2 Kaby Lake processor chip.
Used in mainstream desktop computers.

It has 4 CPU cores to compute in parallel.



Samsung Galaxy S10+

Exynos 9 Series (9820)

6.4" Quad HD+ Curved Dynamic AMOLED (3040x1440)

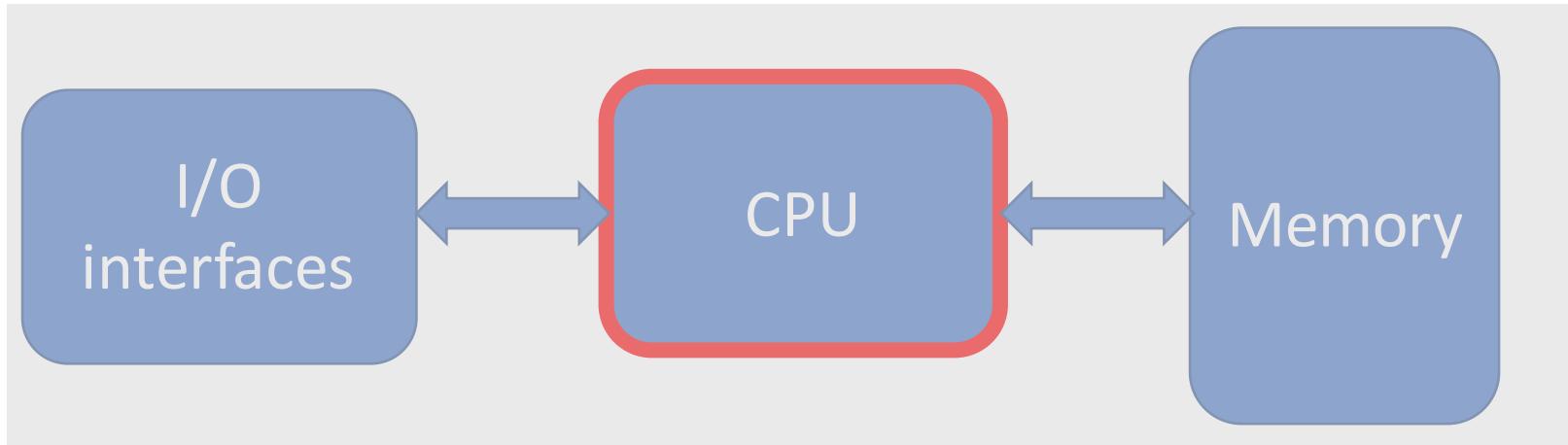
Android 9 (Pie)

8/12GB RAM, 128/256GB/1TB Storage, microSD(up to 512GB)

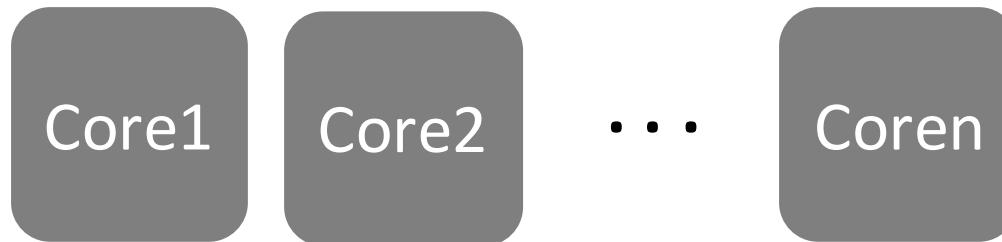
Rear: 12MP AF(Wide)+12MP AF(Tele)+16MP FF(Ultra Wide),
Front: 10MP AF+8MP FF(RGB Depth)

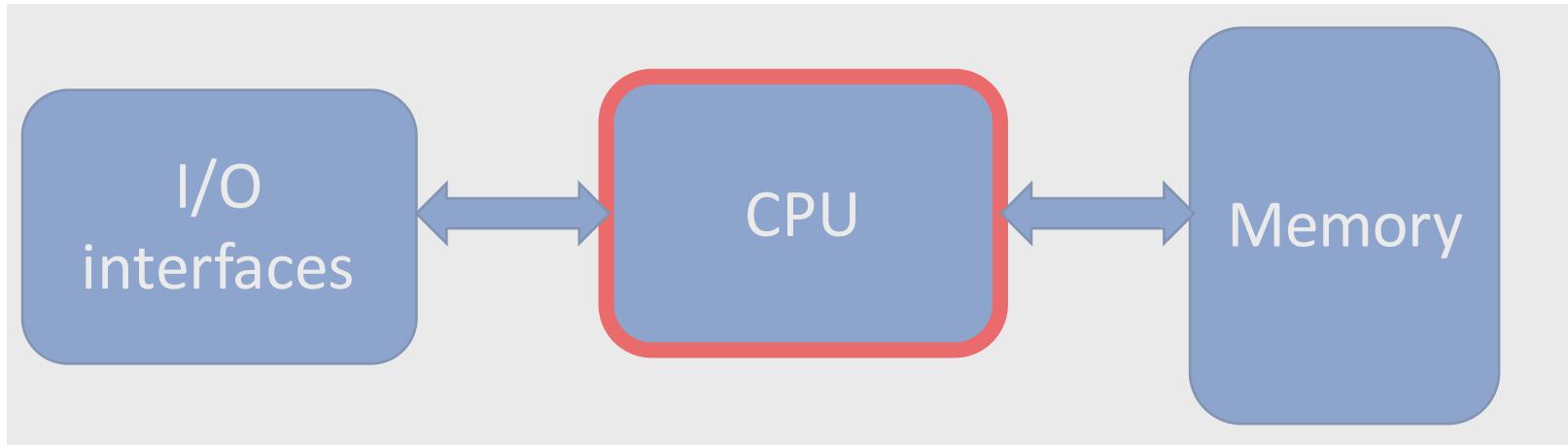
Superior Performance for Seamless Multi-tasking

Featuring a 4th generation custom CPU, the Exynos 9820's innovative tri-cluster architecture delivers premium processing power. The CPU consists of two custom cores for ultimate processing power, two Cortex-A75 cores for optimal performance, and four Cortex-A55 cores for greater efficiency, resulting in superior performance that lasts. Tri-cluster with intelligent task scheduler boosts multi-core performance by 15 percent when compared to the Exynos 9810, while the 4th generation custom CPU with enhanced memory access capability and cutting-edge architecture design improves single core performance by up to 20 percent or boosts power efficiency by up to 40 percent.¹



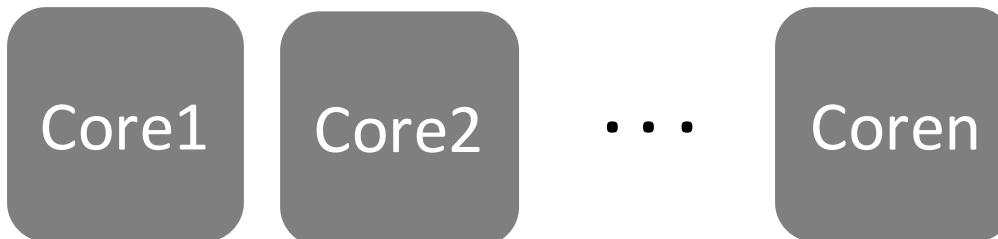
From Single Core
To
Multi Core



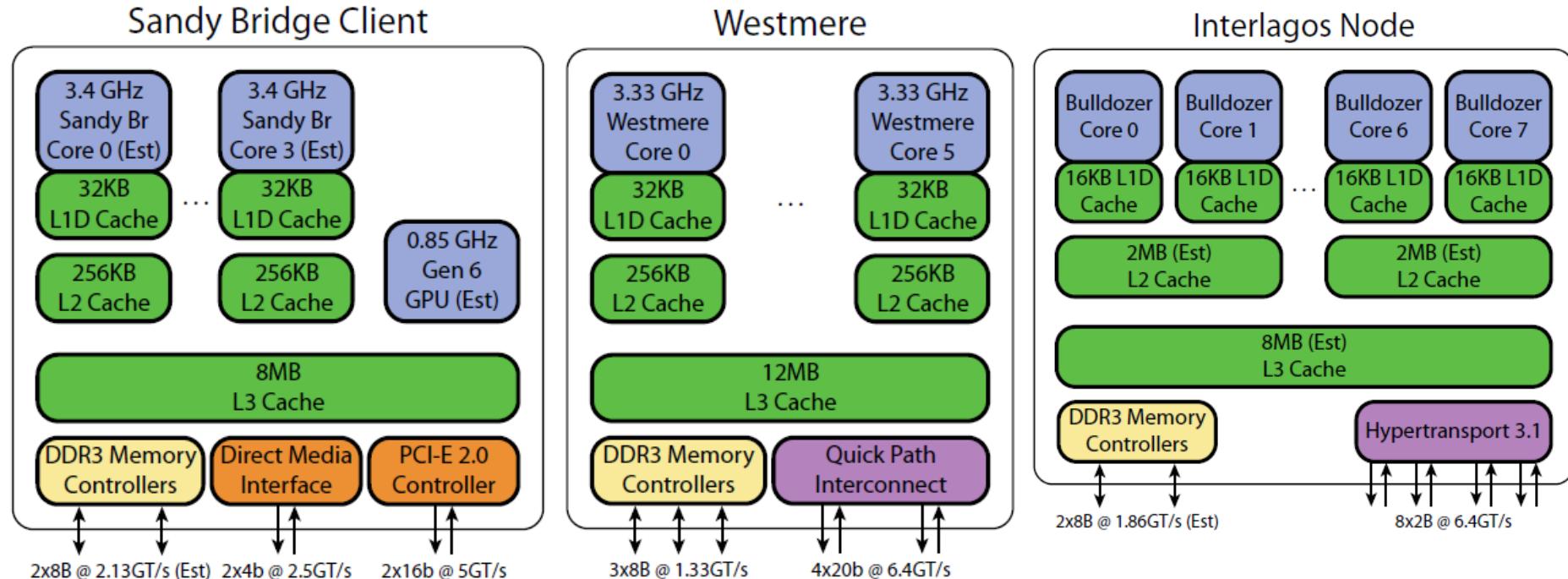


From Single Core
To
Multi Core

What happens to the Memory?
One memory? Multiple memory?

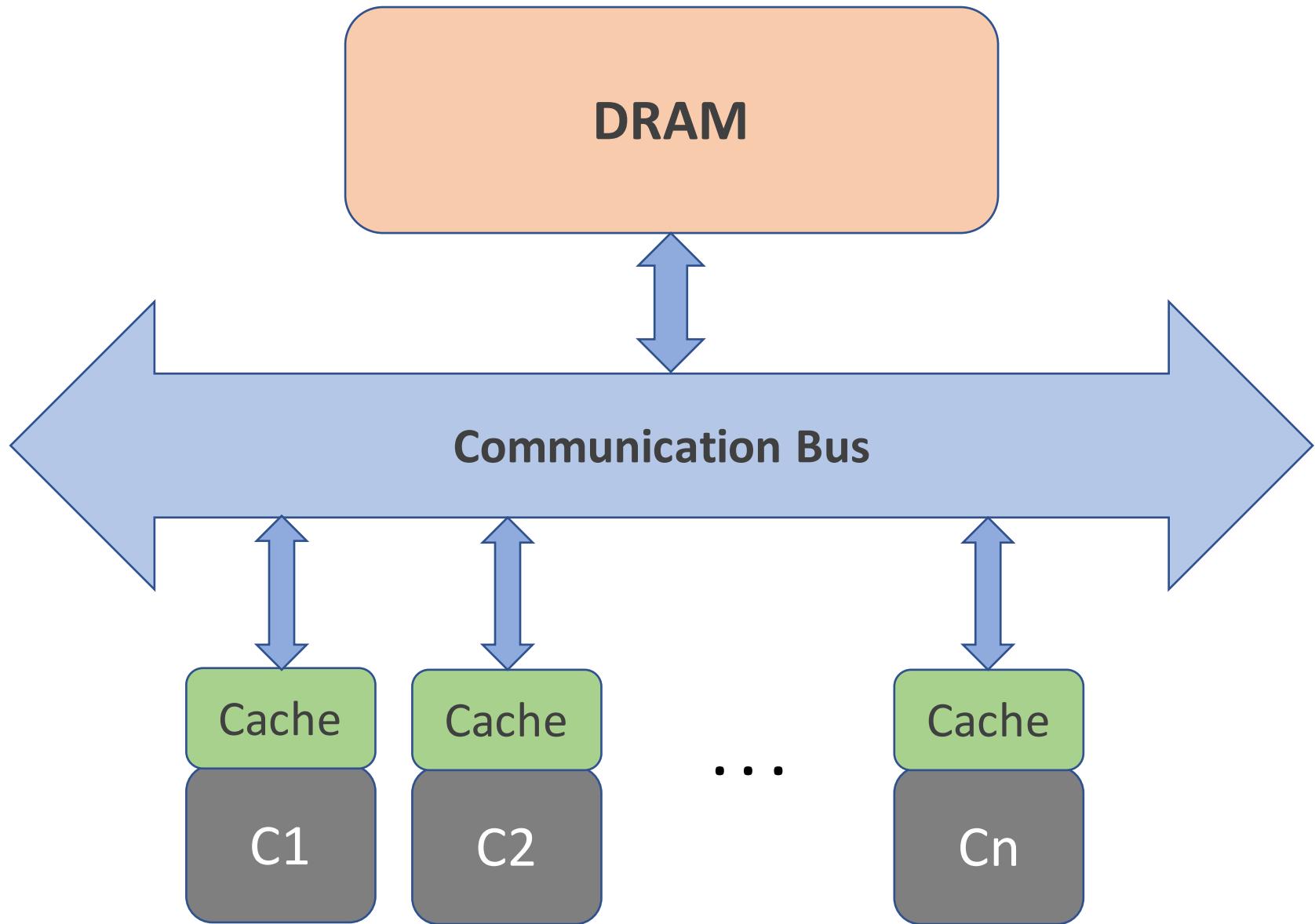


Examples of Intel Multicore Processors

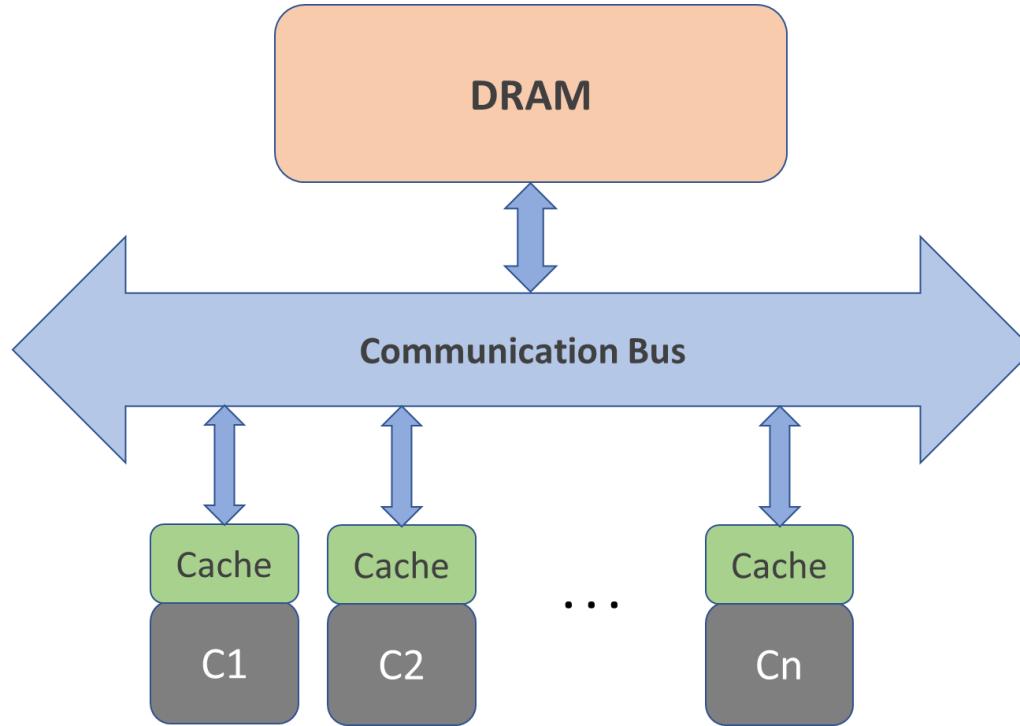


- Each Core has its own L1 Cache
- In some processors, each core has its own L2 Cache
- All cores share one L3 cache
- All cores share one DDR3 DRAM (main memory)

Simplified Multicore Architecture



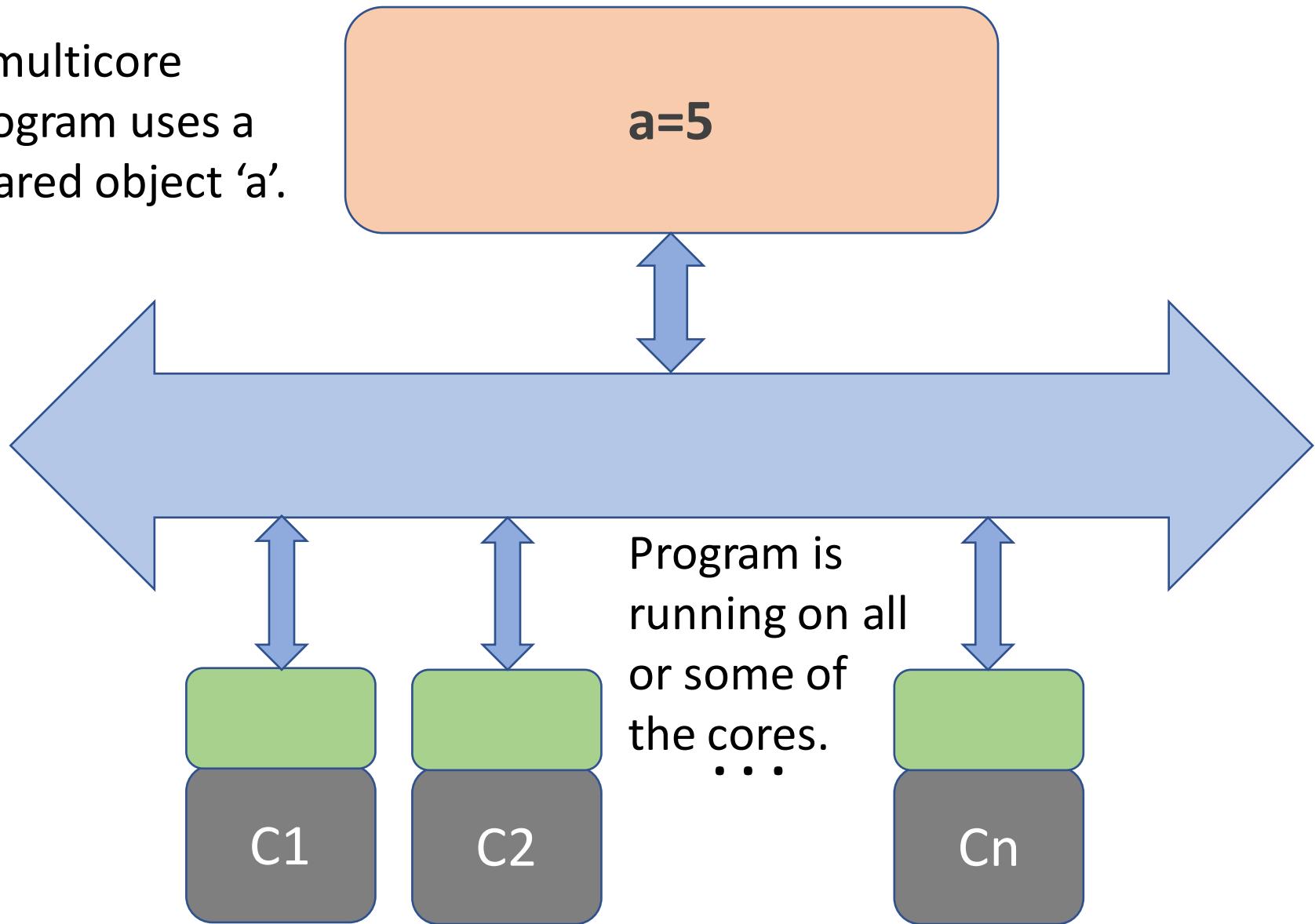
Coherence of data



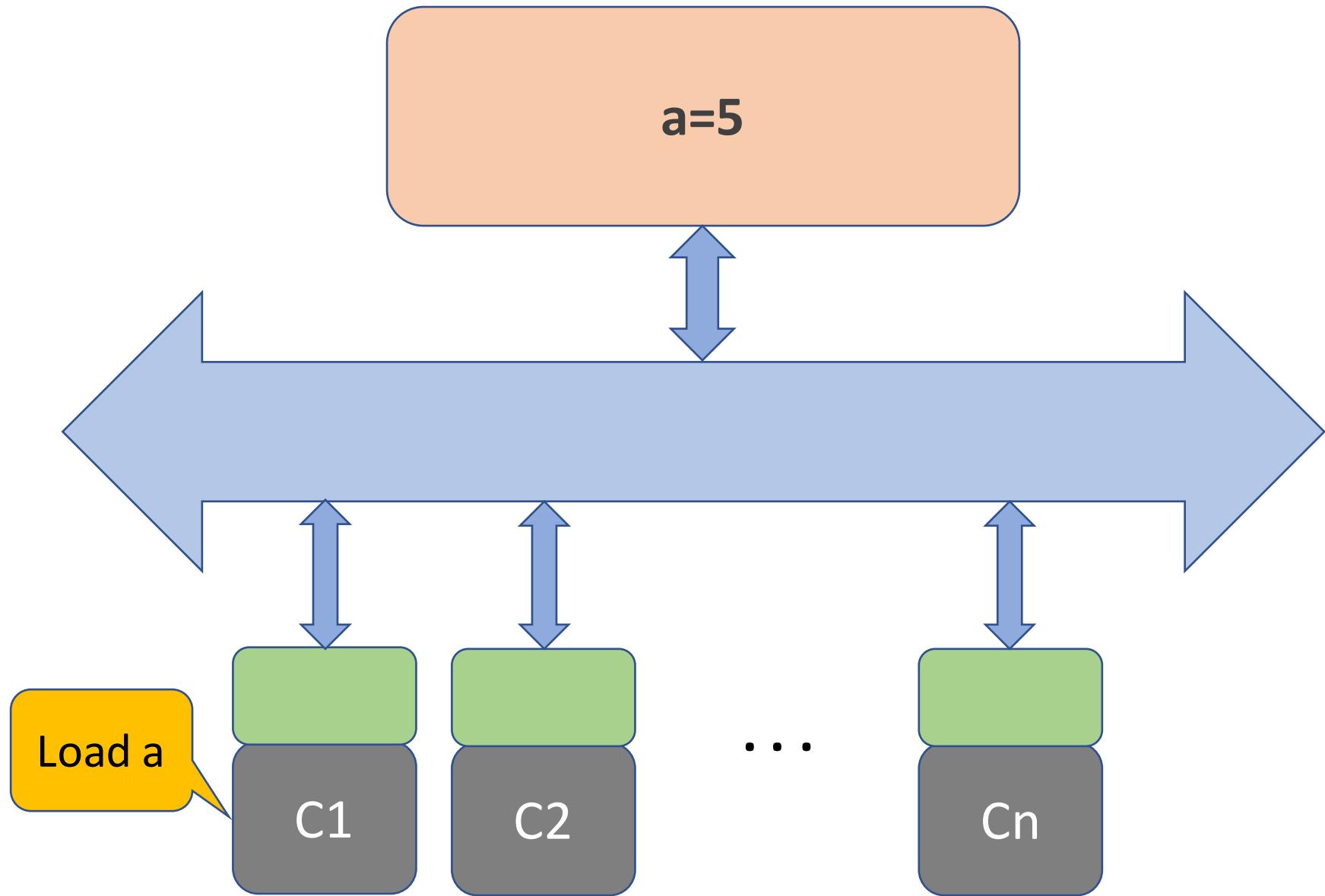
- ‘Coherence’ is the quality of being logical and consistent. During program execution, data must remain coherent.
- On a multicore system, a data variable may reside in multiple caches and might get updated ‘locally’ by its local core
→ this causes coherence problem

Example: Cache coherence problem

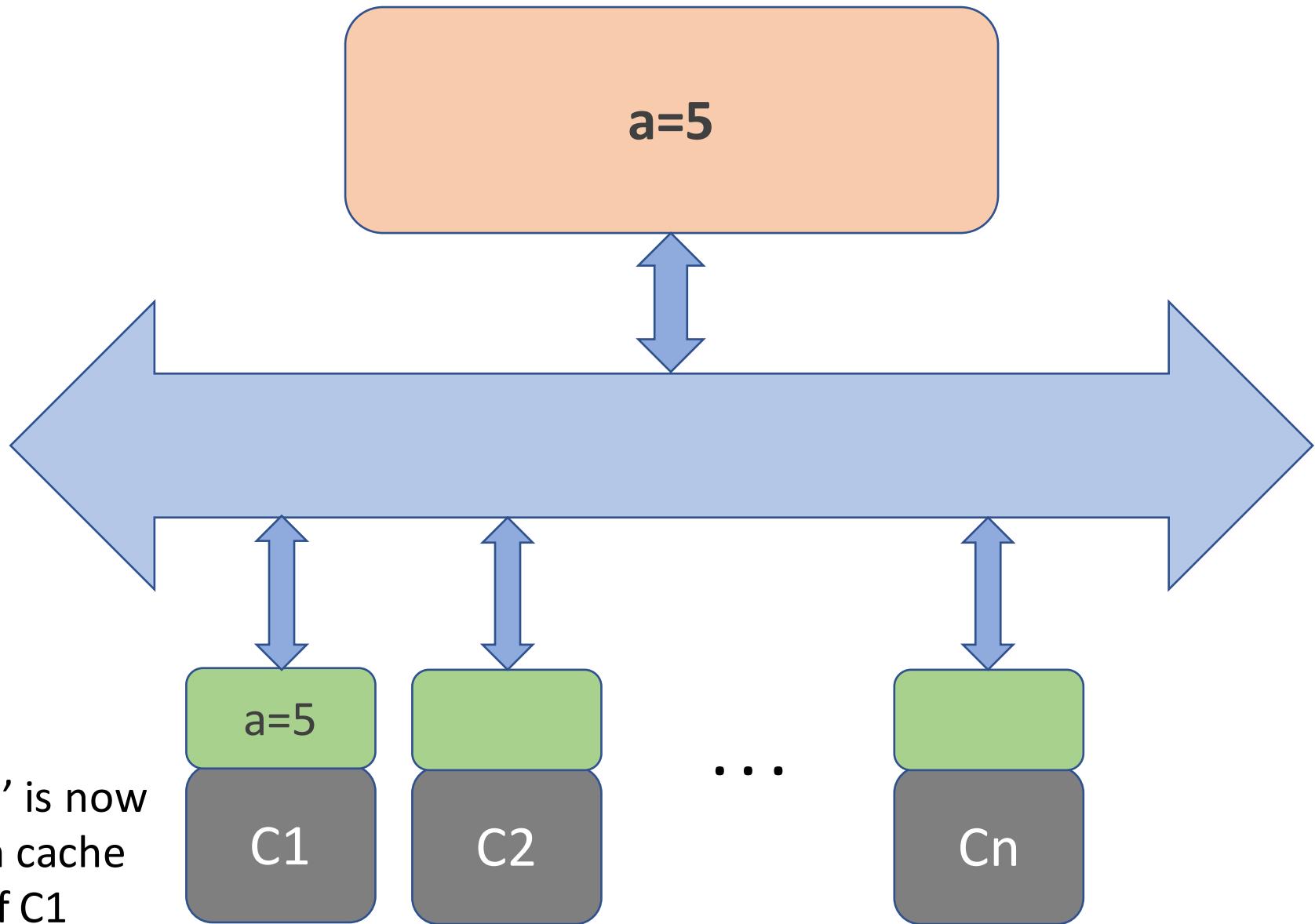
A multicore program uses a shared object 'a'.



Example: Cache coherence problem

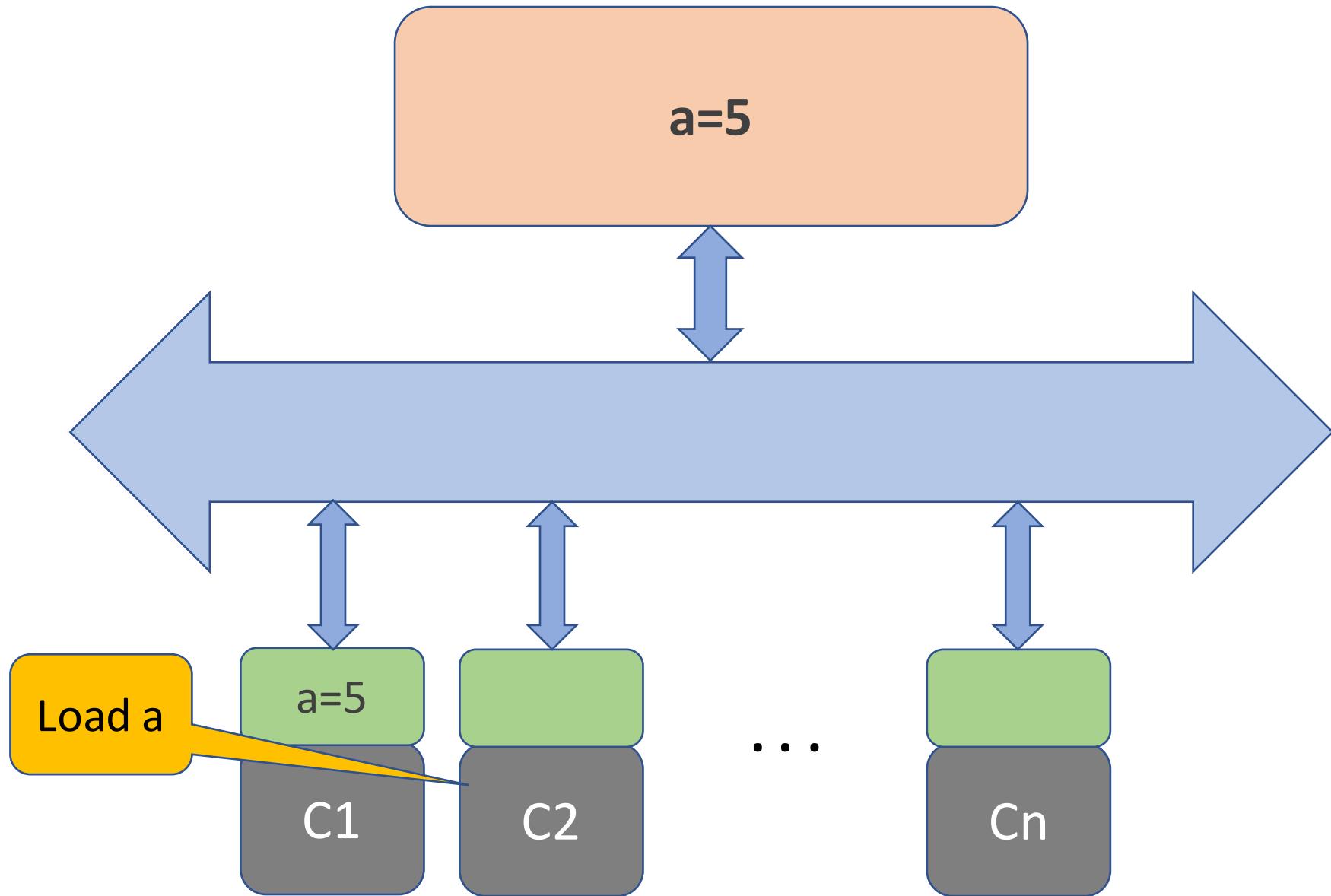


Example: Cache coherence problem

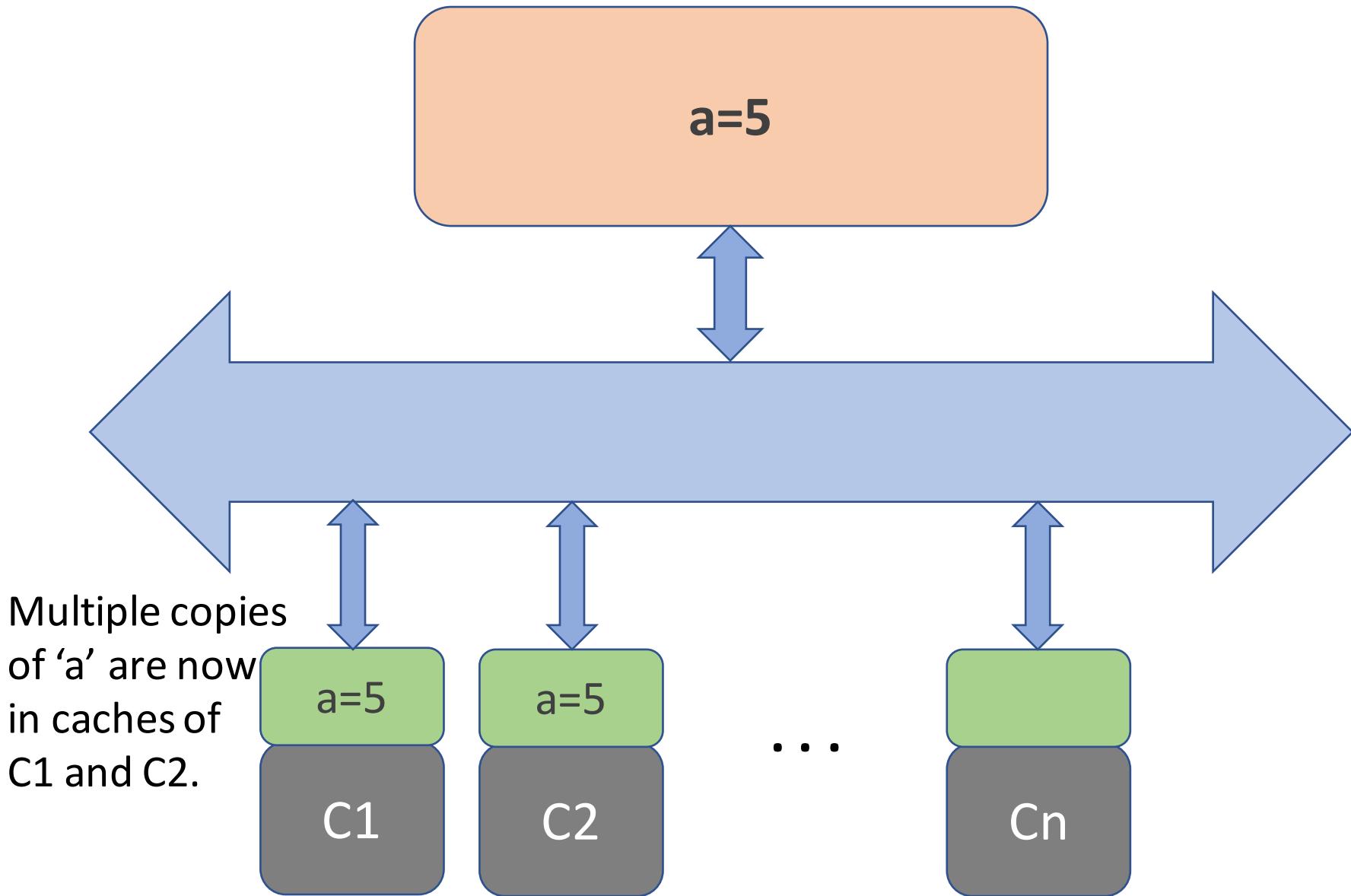


'a' is now
in cache
of $C1$

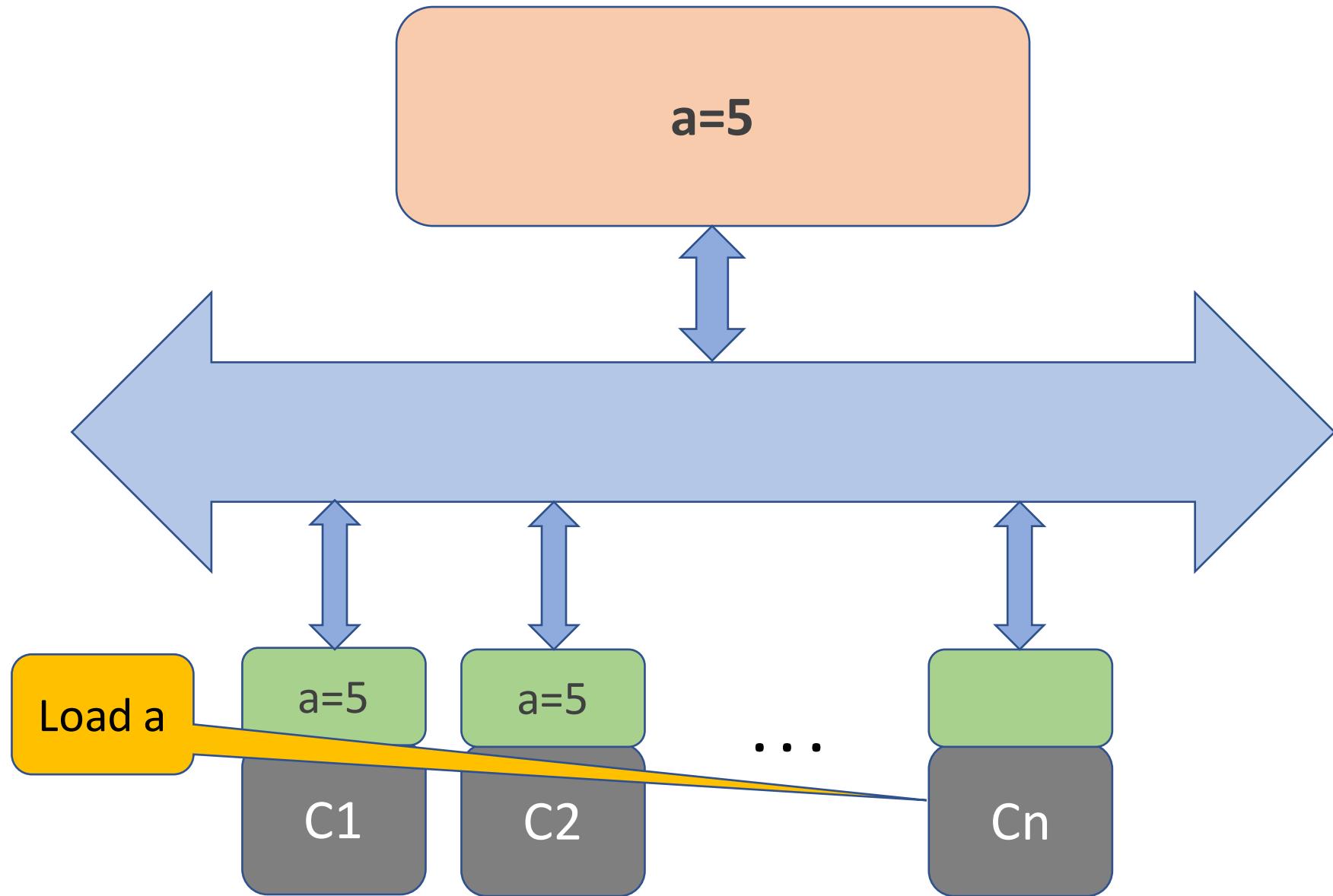
Example: Cache coherence problem



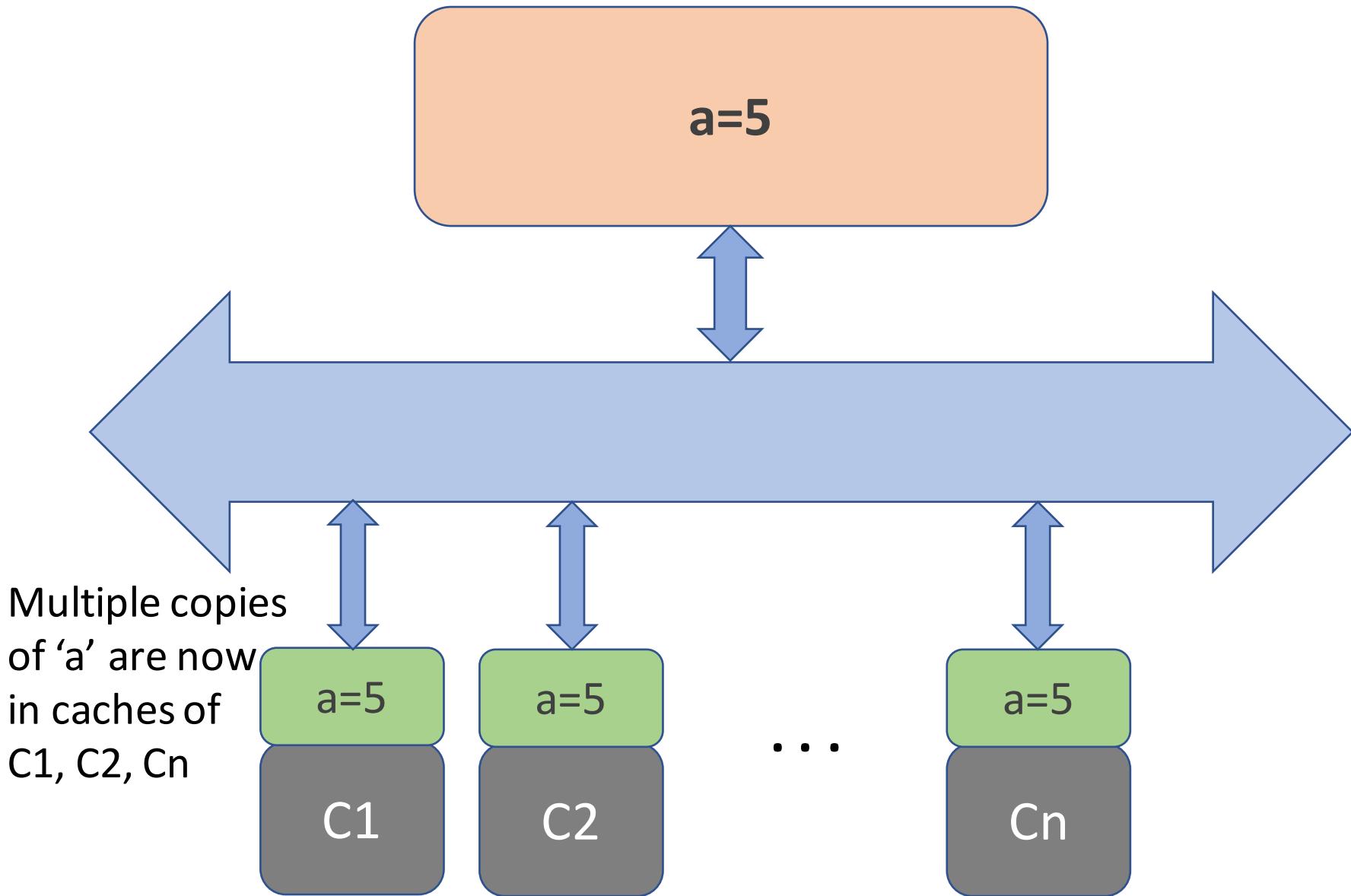
Example: Cache coherence problem



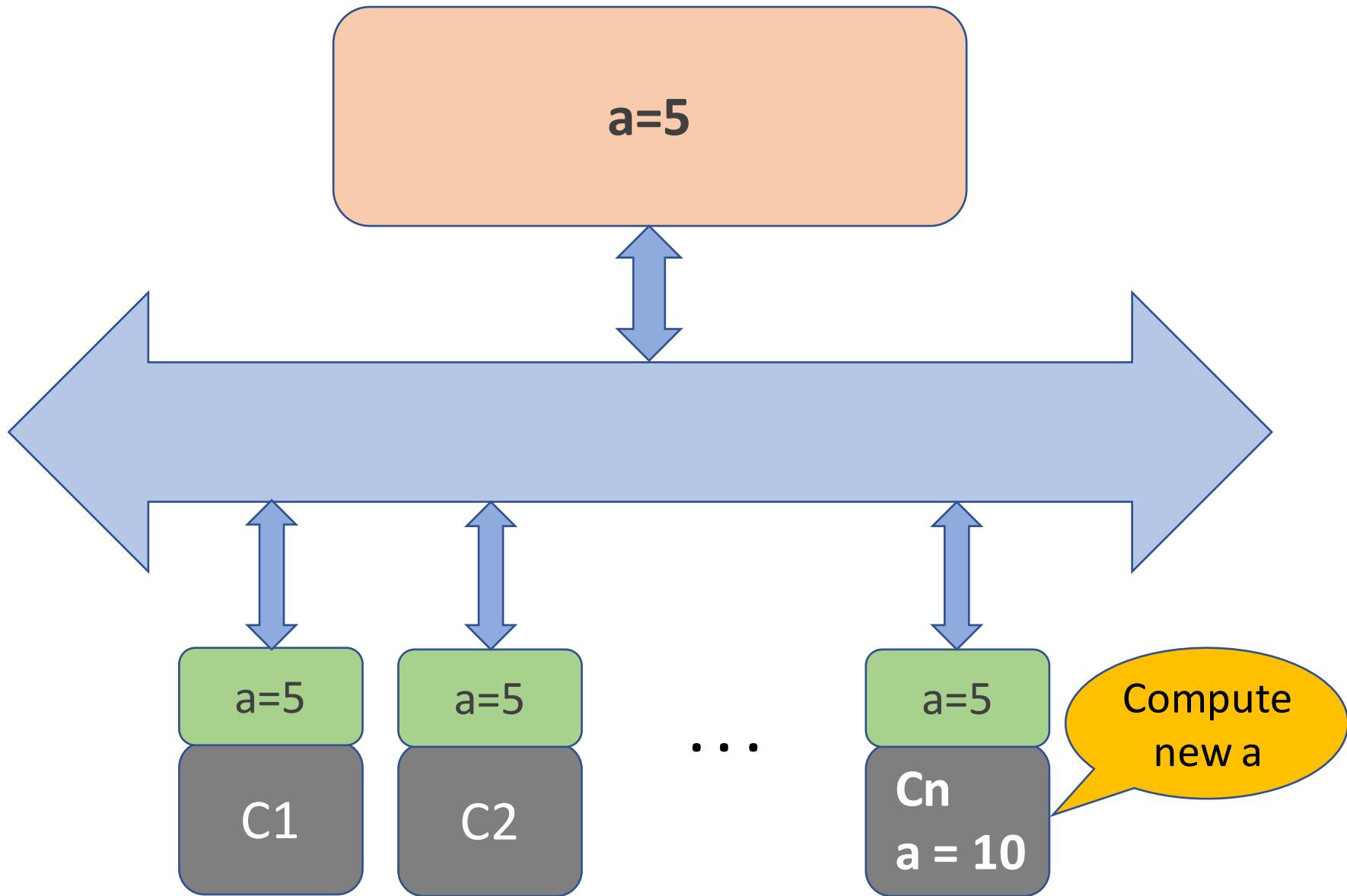
Example: Cache coherence problem



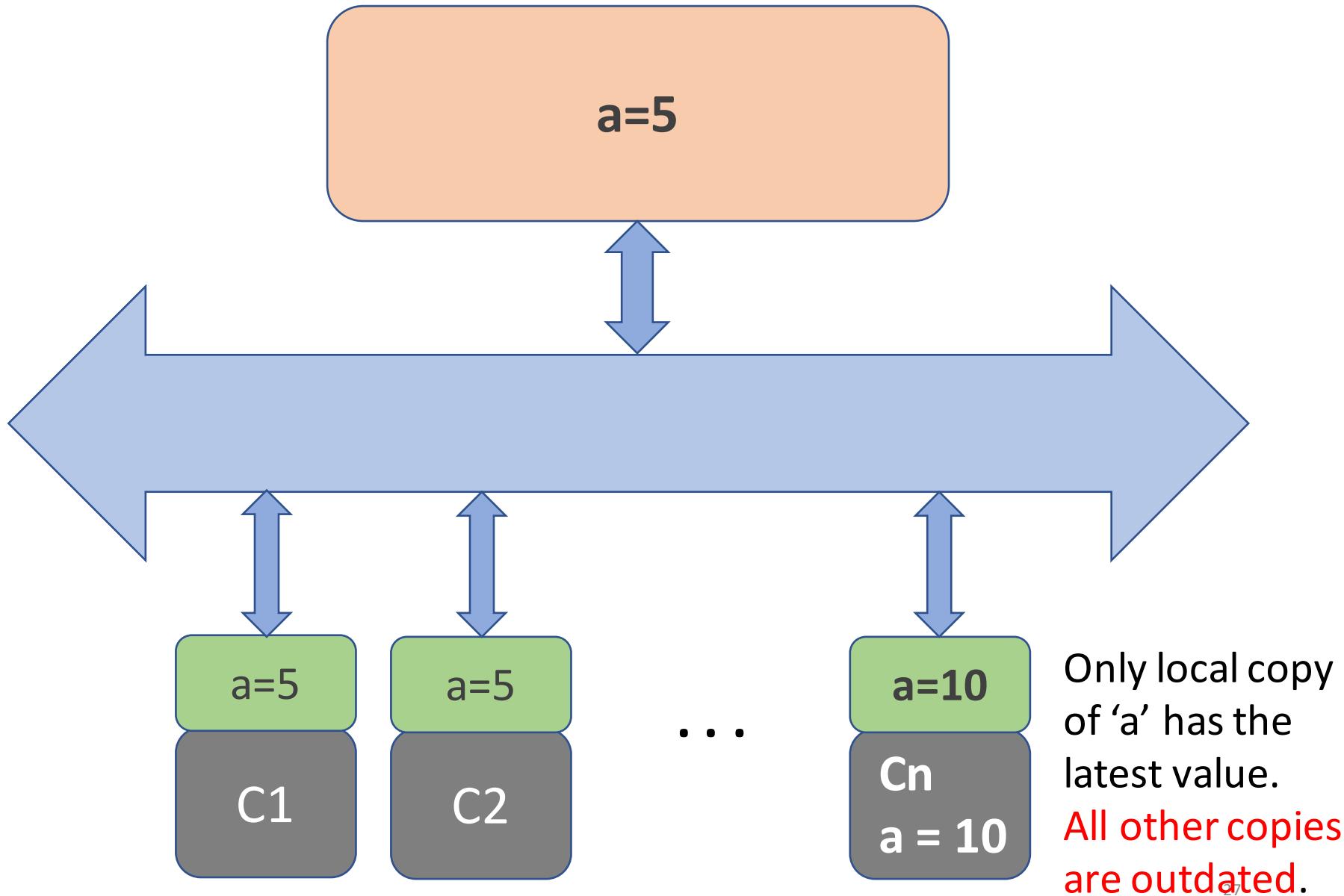
Example: Cache coherence problem



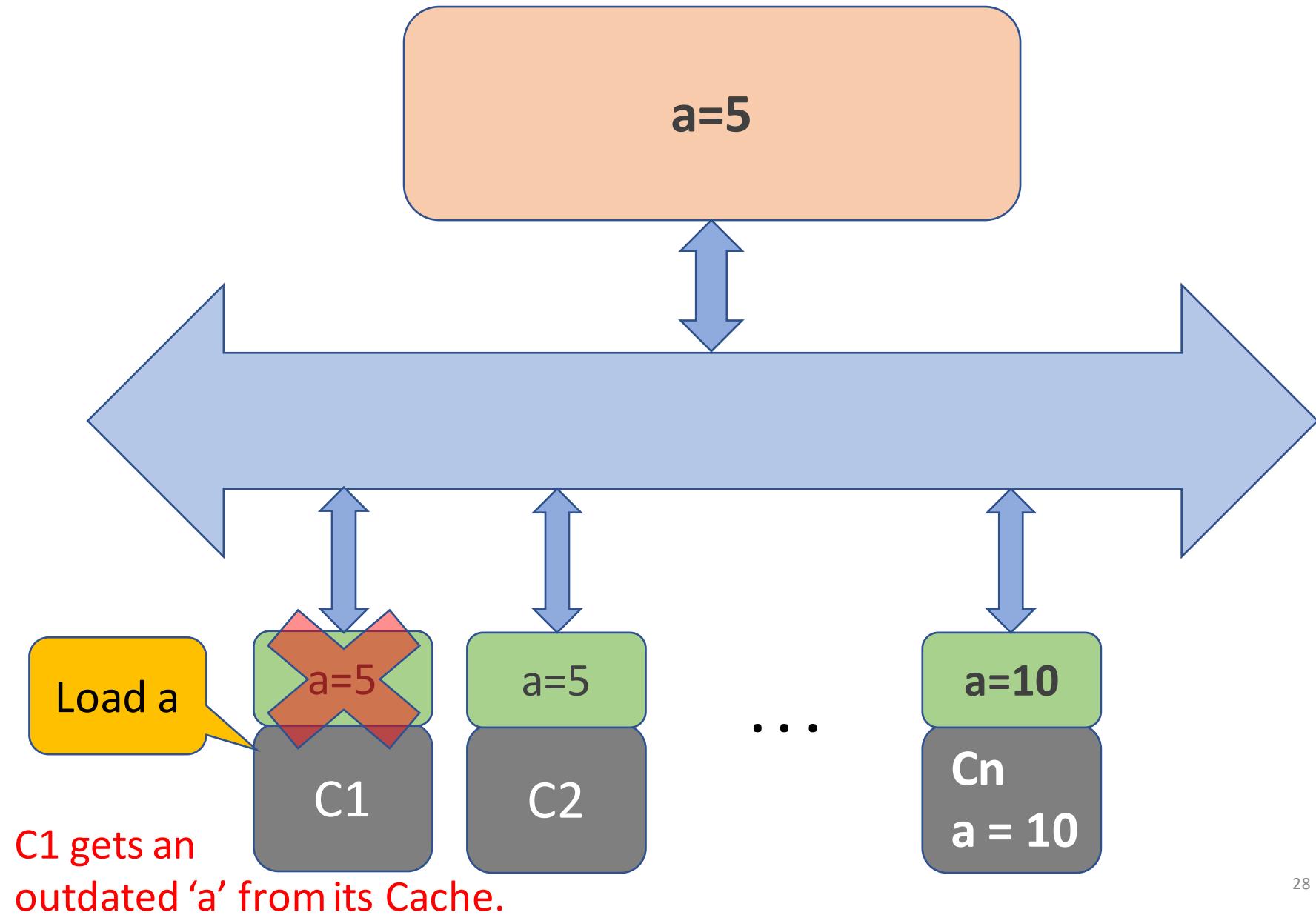
Example: Cache coherence problem



Example: Cache coherence problem



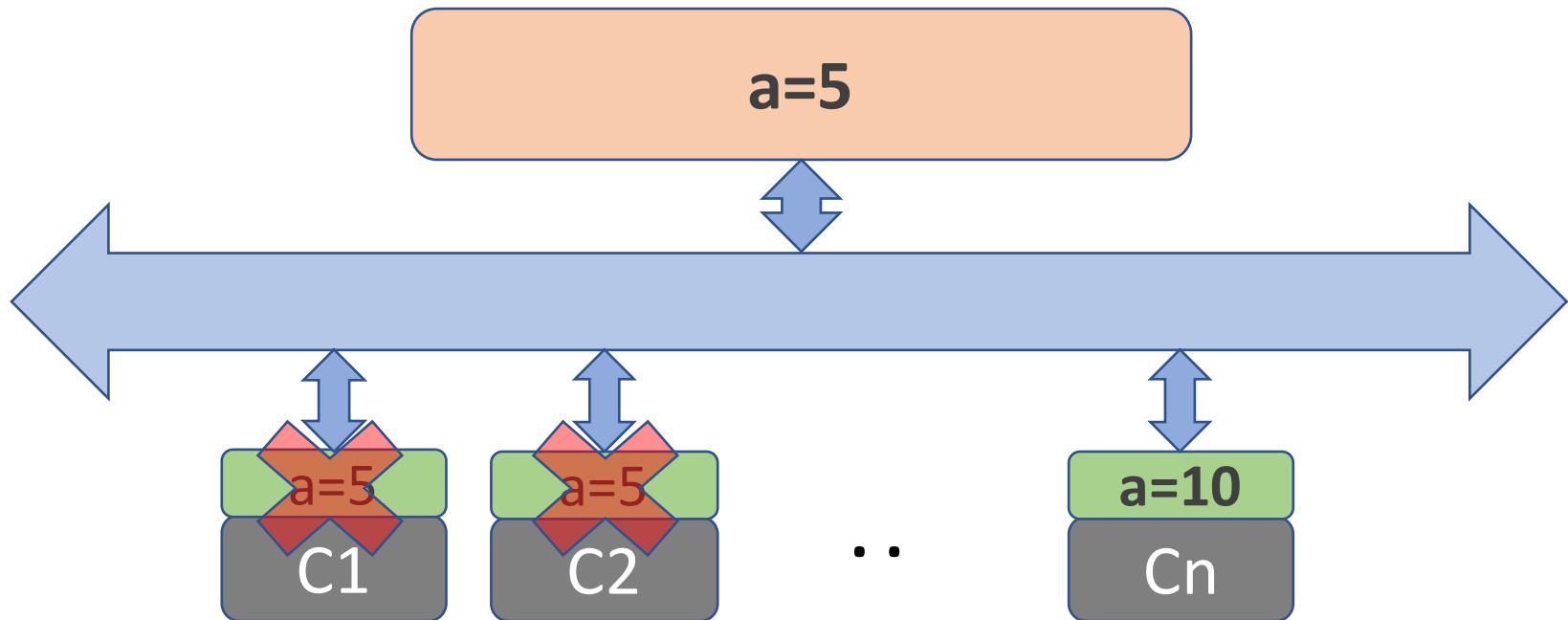
Example: Cache coherence problem



Coherence mechanisms

To achieve coherence:

a **write to a memory location** must cause all other copies of this location to be **removed from the caches** they are in.



Do: As soon as 'a' gets updated by a core, invalidate or delete all other copies of it.

Coherence protocols

- Coherence protocols apply cache coherence in multicore systems.
- Goal is that two cores **must never see different values** for the same shared data.

How this is achieved?

Additional information is stored and passed

- Is this data stored in multiple caches?
- Has this cache line been modified?

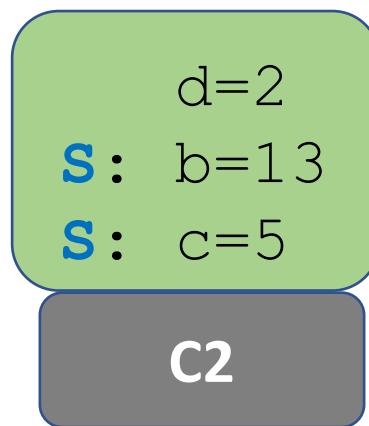
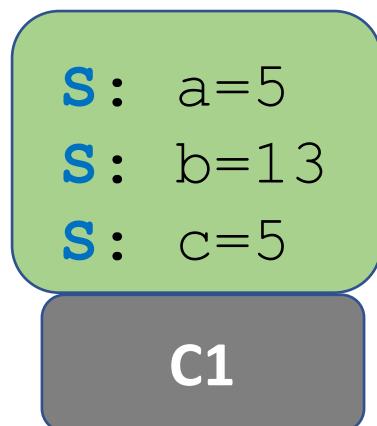
MSI protocols

MSI protocol is a simple cache coherence protocol.

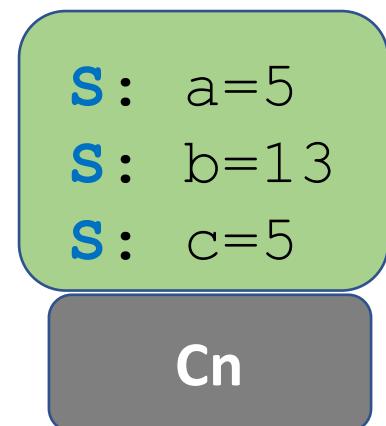
In this protocol, each cache line is labeled with a state:

- **M**: cache block has been modified.
- **S**: other caches may be sharing this block.
- **I**: cache block is invalid

Initial state of caches before computation



...

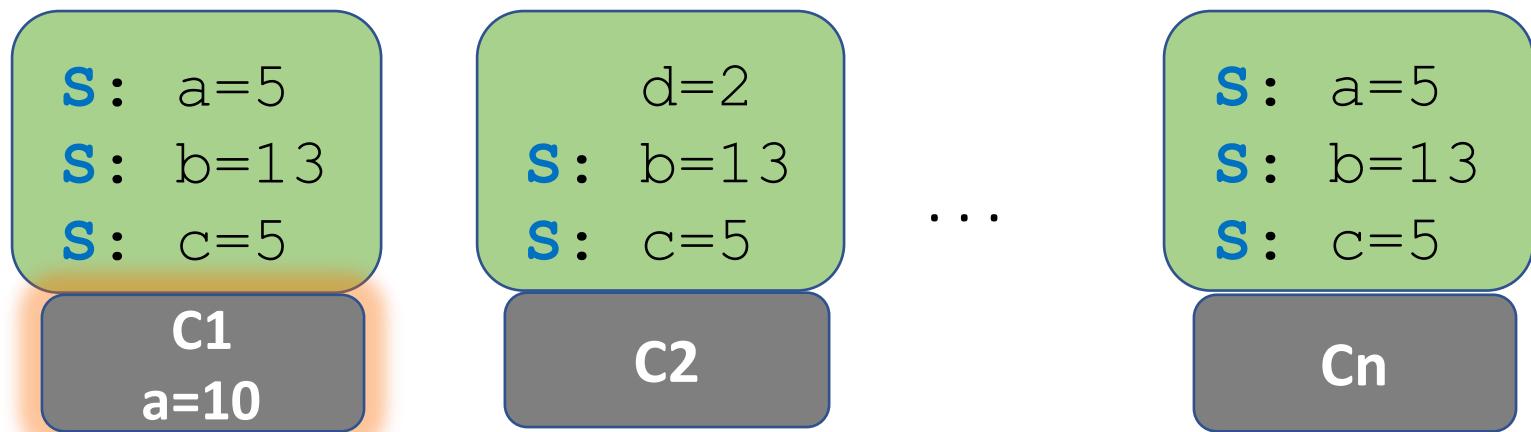


MSI protocols

MSI protocol is a simple cache coherence protocol.
In this protocol, each cache line is labeled with a state:

- **M**: cache block has been modified.
- **S**: other caches may be sharing this block.
- **I**: cache block is invalid

C1 computes new value a=10 in register



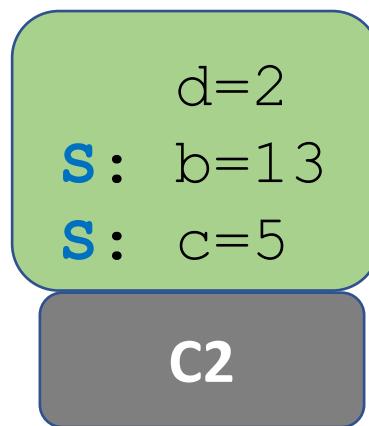
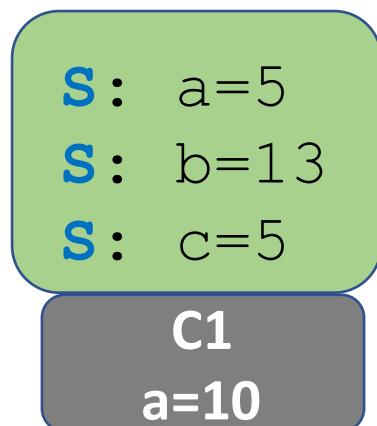
MSI protocols

MSI protocol is a simple cache coherence protocol.

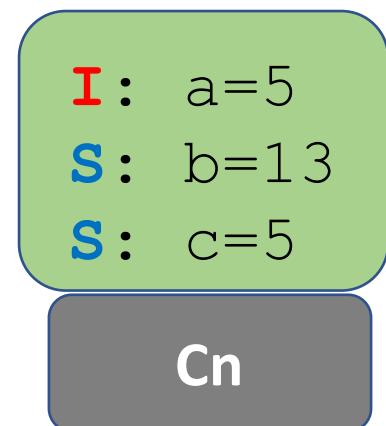
In this protocol, each cache line is labeled with a state:

- **M**: cache block has been modified.
- **S**: other caches may be sharing this block.
- **I**: cache block is invalid

Hardware invalidates all other copies of a



...



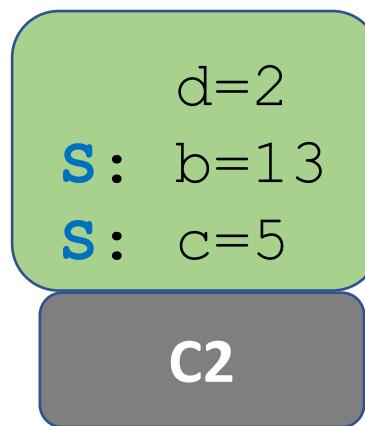
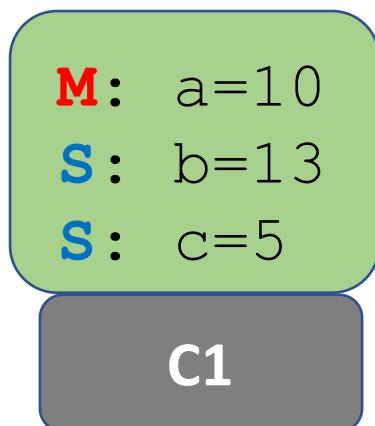
MSI protocols

MSI protocol is a simple cache coherence protocol.

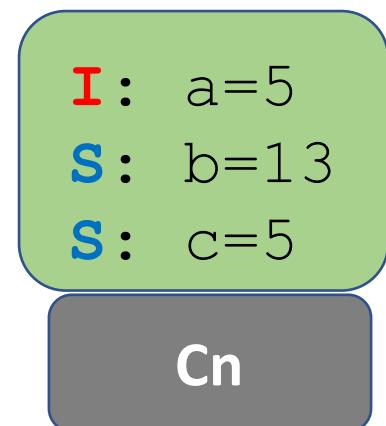
In this protocol, each cache line is labeled with a state:

- **M**: cache block has been modified.
- **S**: other caches may be sharing this block.
- **I**: cache block is invalid

Hardware updates a in cache of C1 and marks it with ‘M’



...



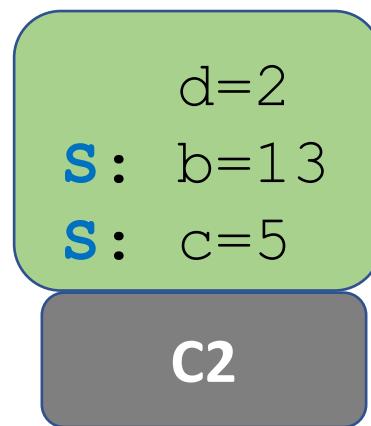
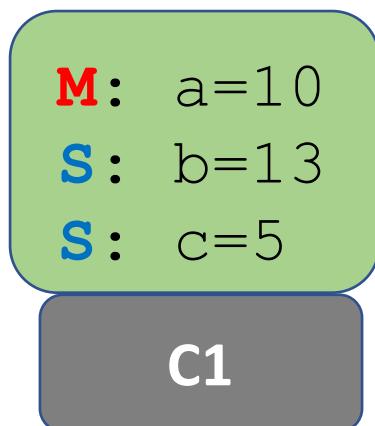
MSI protocols

MSI protocol is a simple cache coherence protocol.

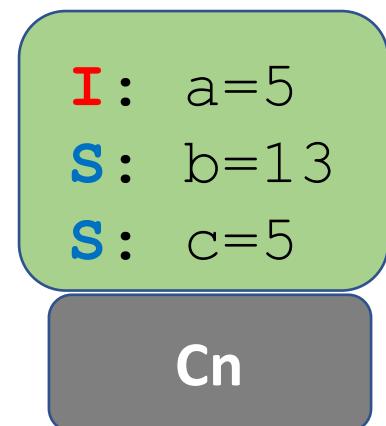
In this protocol, each cache line is labeled with a state:

- **M**: cache block has been modified.
- **S**: other caches may be sharing this block.
- **I**: cache block is invalid

Hardware also updates ‘a’ in the shared memory



...



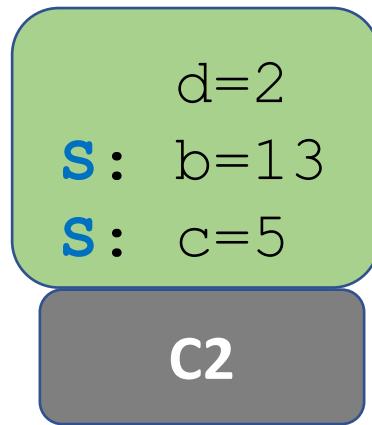
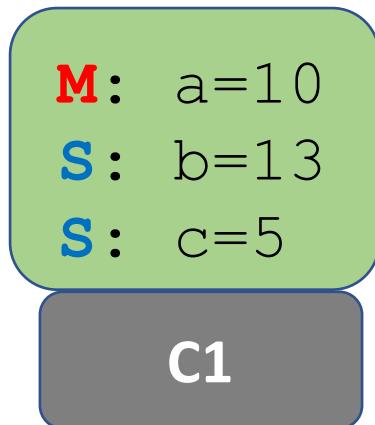
MSI protocols

MSI protocol is a simple cache coherence protocol.

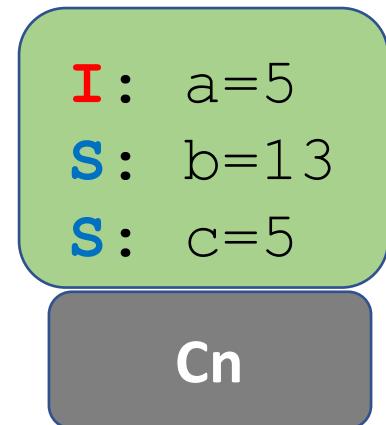
In this protocol, each cache line is labeled with a state:

- **M**: cache block has been modified.
- **S**: other caches may be sharing this block.
- **I**: cache block is invalid

If Cn wants to load 'a' then 'a' will be brought from the shared memory to the local cache of Cn.



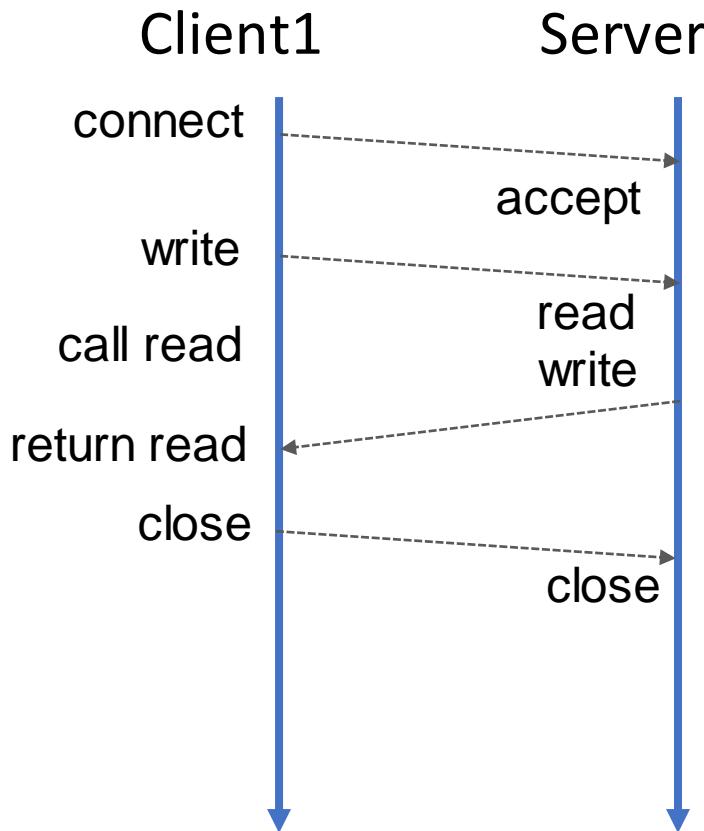
...



Programming multicore platforms

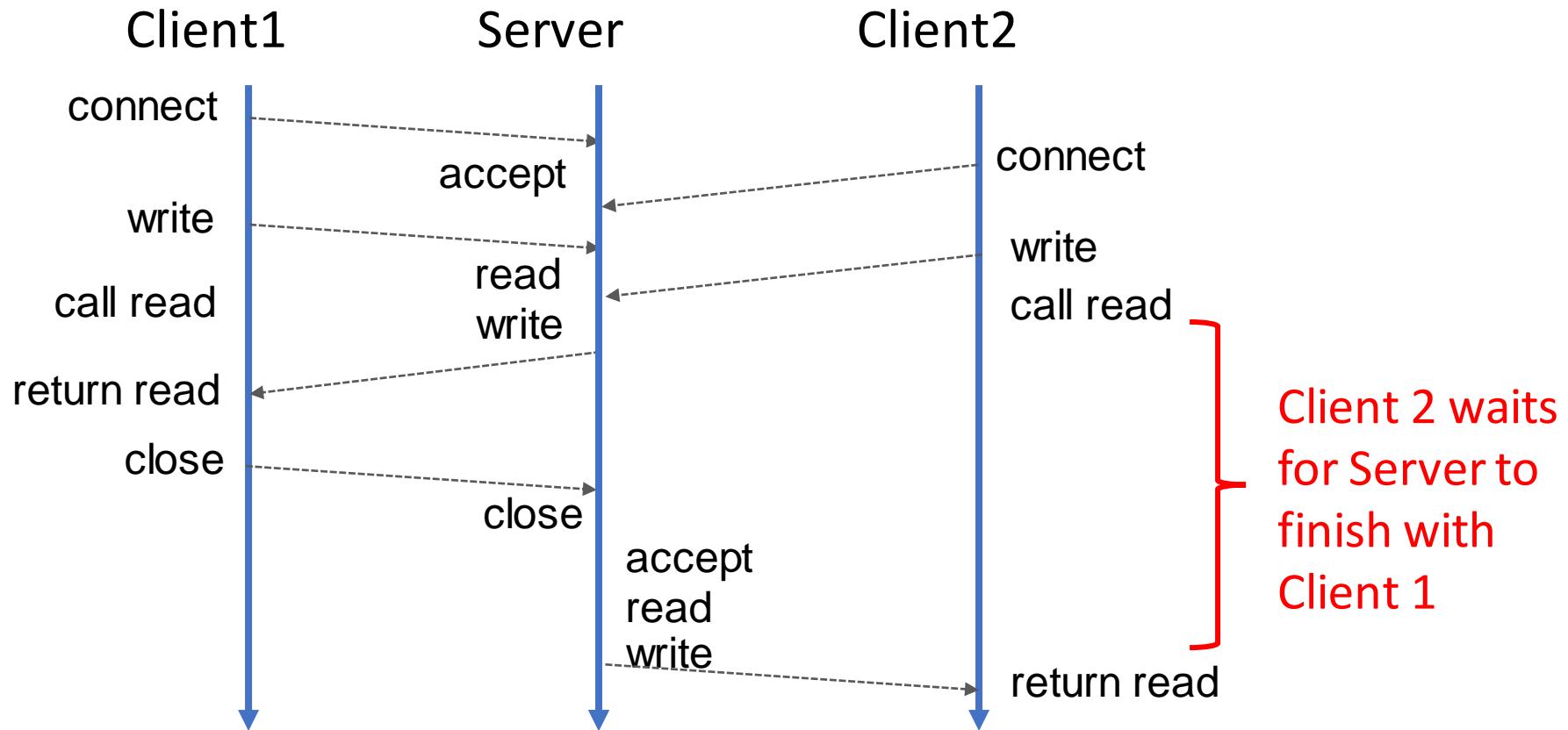
- Option 1: Program directly targeting processor cores
 - Programmer takes care of synchronization
 - Painful and error-prone
- Option 2: Use a *concurrency platform*
 - It abstracts processor cores, handles synchronization and communication protocols, and performs load balancing.
 - Hence offers much easier multicore programming environment
 - Examples:
 - **Pthreads** and WinAPI threads
 - OpenMP

Example of a Sequential Server



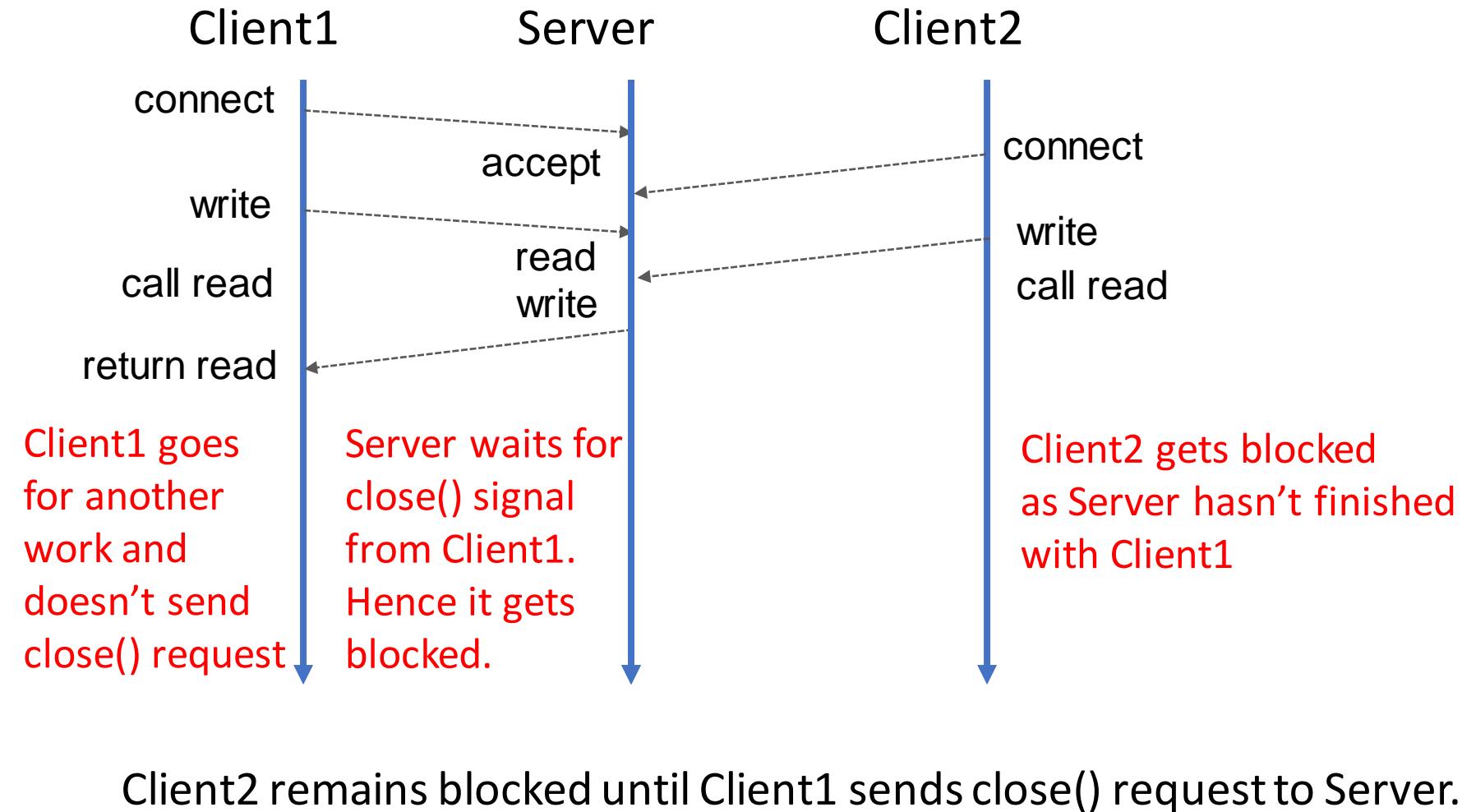
- Sequential server processes one client at a time

Example of a Sequential Server

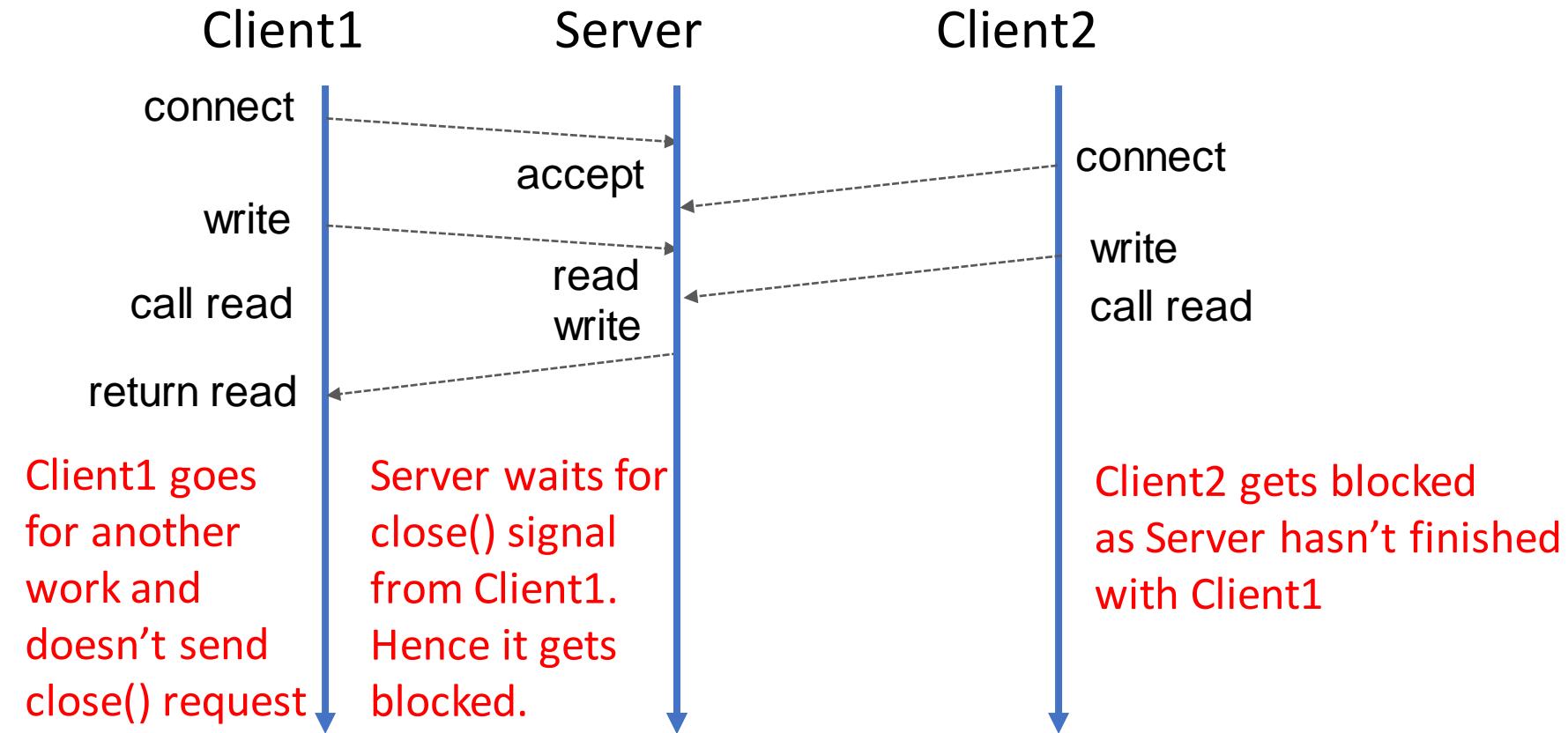


- Sequential server processes one client at a time
- In this example, Client 2 has a long waiting time

Big problem with Sequential Server



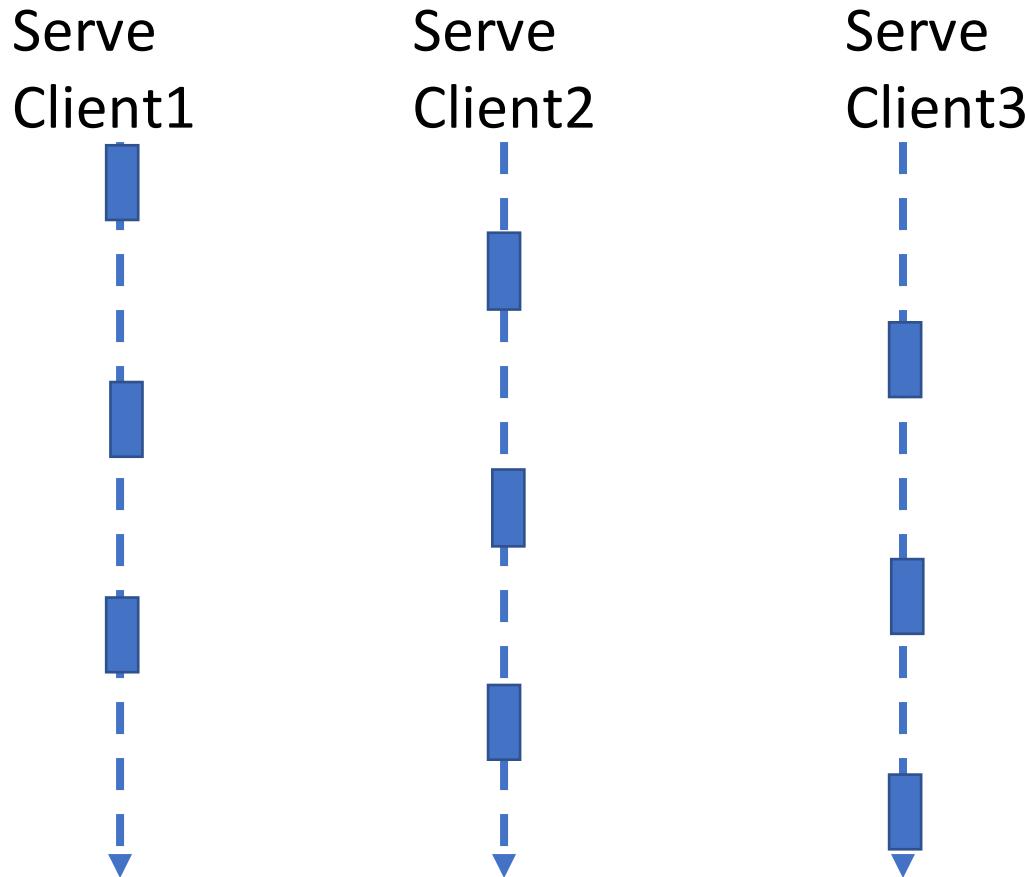
Big problem with Sequential Server



Client2 remains blocked until Client1 sends close() request to Server.

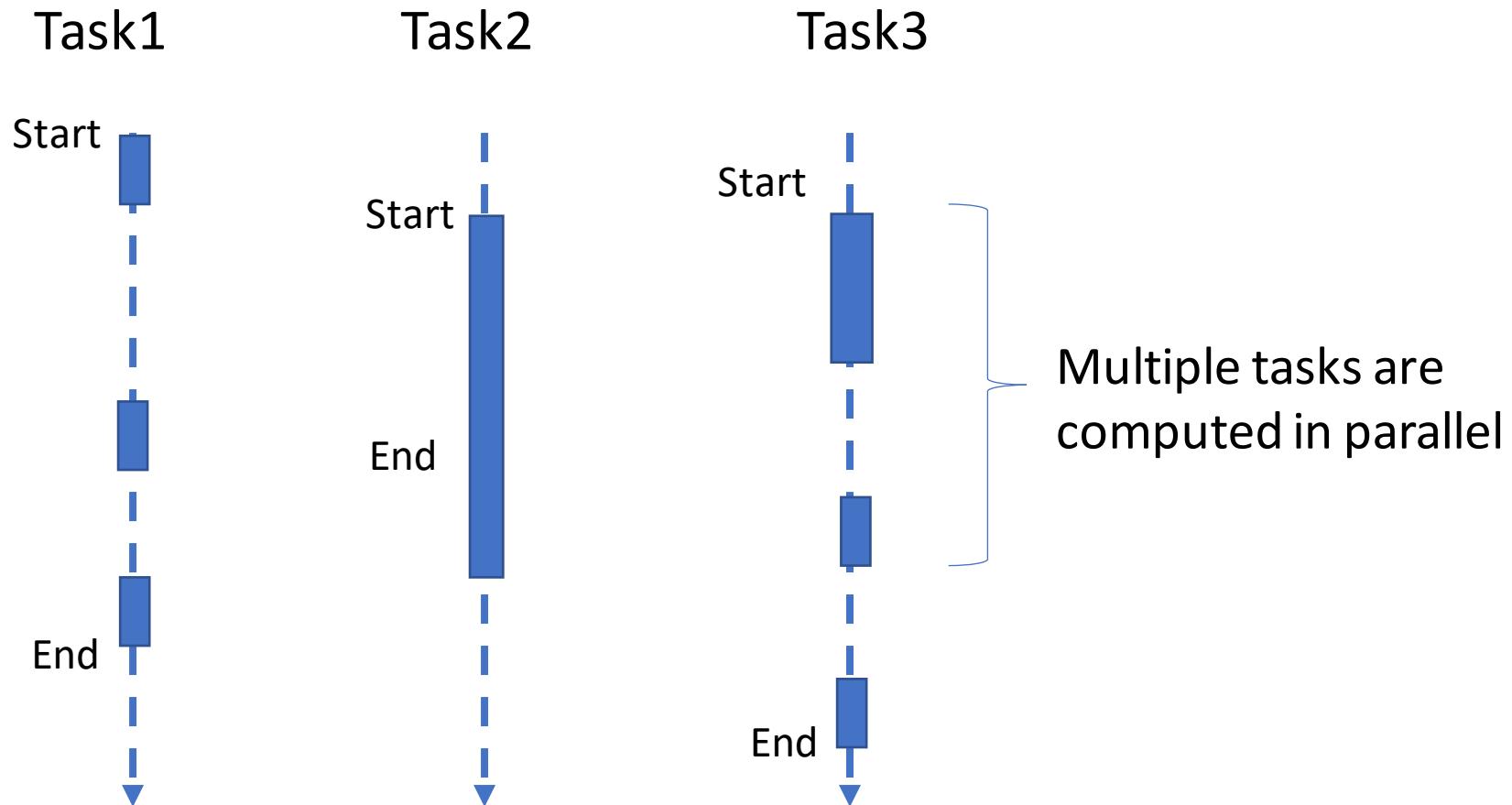
Solution: Use a concurrent server to serve multiple clients concurrently. Thus, a client cannot block another client.

Concurrent Server



- All clients get served by the server.
- Even if one client causes blocking, the other clients do not have to wait.

Concurrent vs Parallel



- Task2 is **parallel** to Task1 and Task3
- Parallel tasks are always concurrent.
- Concurrent tasks may not be parallel (Task1 and Task3)
- So, 'concurrency' is a more general term

Concurrency using Threads

Program execution: Sequential perspective

```
#include <stdio.h>
void do_one_thing(int *);
void do_another_thing(int *);
void do_wrap_up(int, int);

int r1 = 0, r2 = 0;
int main(void)
{
    do_one_thing(&r1);
    do_another_thing(&r2);
    do_wrap_up(r1, r2);
    return 0;
}
```

In a serial system, we see this program as a *sequence of instructions* which are executed one-by-one.

- It starts from main()
- then it executes the instructions that are inside do_one_thing()
- after that do_another_thing()
- and finally do_wrap_up()

Program execution: Concurrent perspective

```
#include <stdio.h>
void do_one_thing(int *);
void do_another_thing(int *);
void do_wrap_up(int, int);

int r1 = 0, r2 = 0;
int main(void)
{
    do_one_thing(&r1);
    do_another_thing(&r2);
    do_wrap_up(r1, r2);
    return 0;
}
```

We look at our program (our big task) as a collection of subtasks.

Example:

Subtask1 is `do_one_thing()`

Subtask2 is `do_another_thing()`

Subtask3 is `do_wrap_up()`

IDEA: If it is possible to execute some of these subtasks at the same time with no change in final result (i.e., correctness), then we can **reduce overall time** by executing these subtasks **concurrently**.

$$C = A + B$$

$$D = C - E$$

$$F = D - K$$

$$C = A + B$$

$$D = A - B$$

$$F = D - K$$

Program execution: Concurrent perspective

```
#include <stdio.h>
void do_one_thing(int *);
void do_another_thing(int *);
void do_wrap_up(int, int);

int r1 = 0, r2 = 0;
int main(void)
{
    do_one_thing(&r1);
    do_another_thing(&r2);
    do_wrap_up(r1, r2);
    return 0;
}
```

We look at our program (our big task) as a collection of subtasks.

Example:

Subtask1 is `do_one_thing()`

Subtask2 is `do_another_thing()`

Subtask3 is `do_wrap_up()`

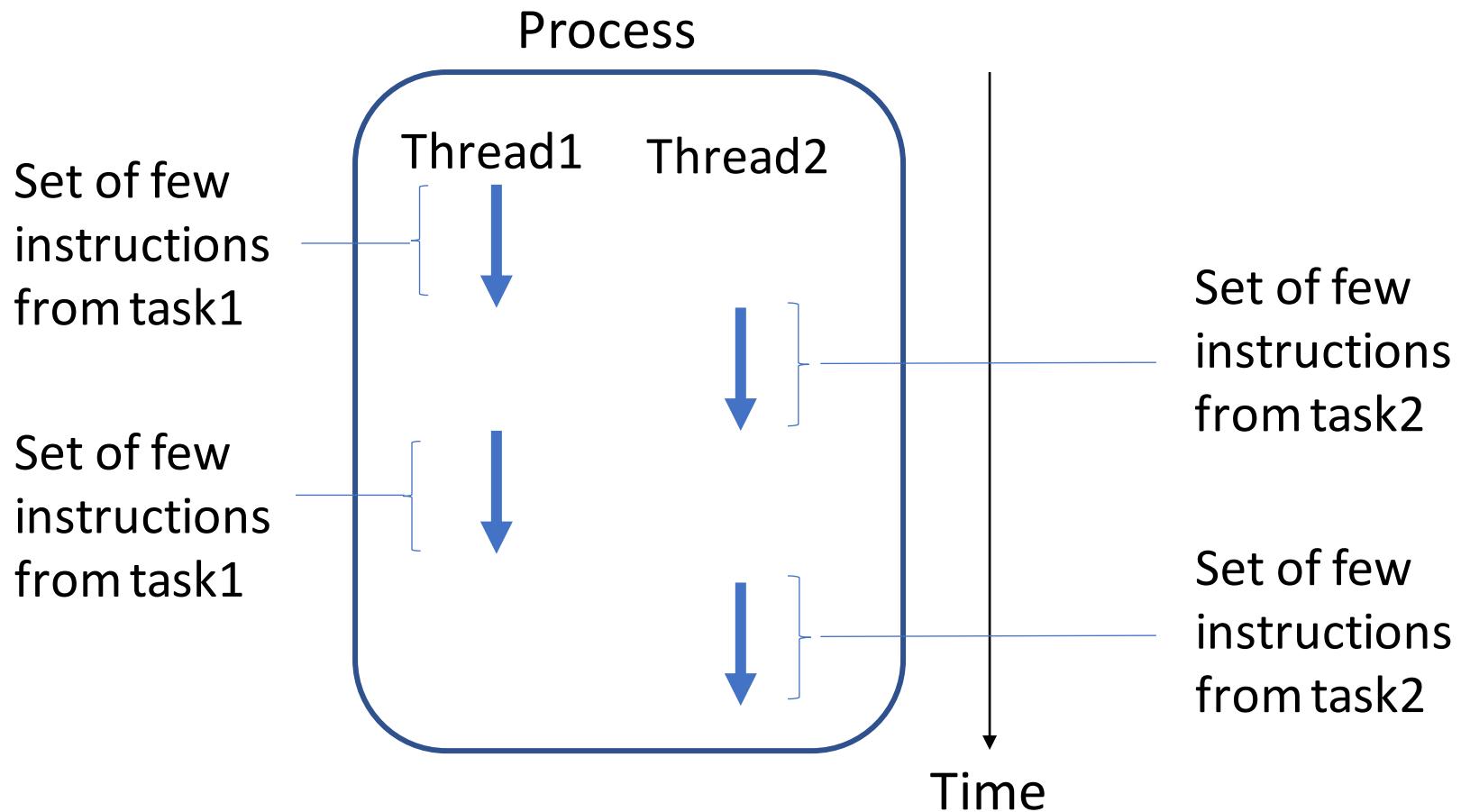


These subtasks can be executed as ‘threads’ within a single program

IDEA: If it is possible to execute some of these subtasks at the same time with no change in final result (i.e., correctness), then we can **reduce overall time** by executing these subtasks **concurrently**.

What is a thread?

A thread of execution is the smallest sequence of programmed instructions that can be managed independently by a scheduler, which is typically a part of the operating system.



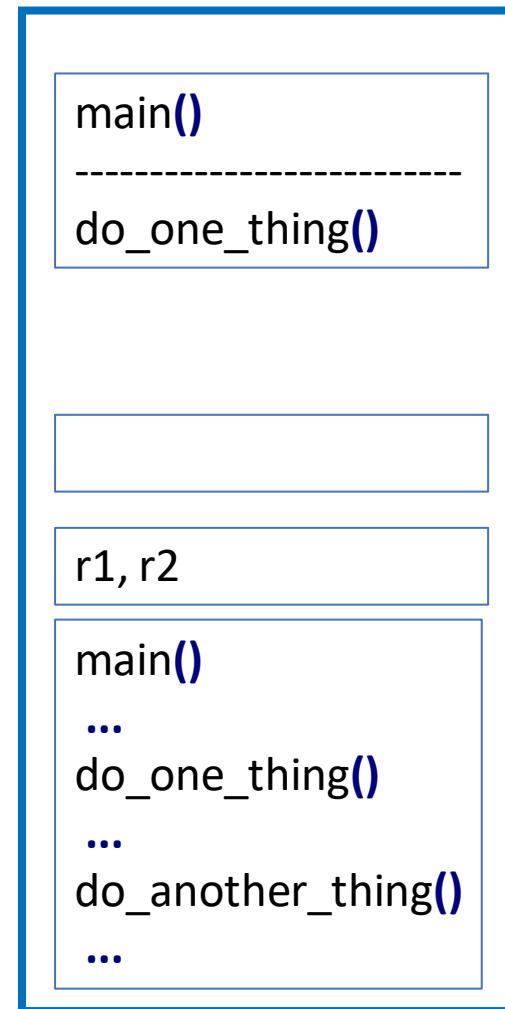
Program as a single process with no threads

Stack related info

Process Identity

Info about other resources. e.g., files, network sockets, locks etc.

High address



Low address

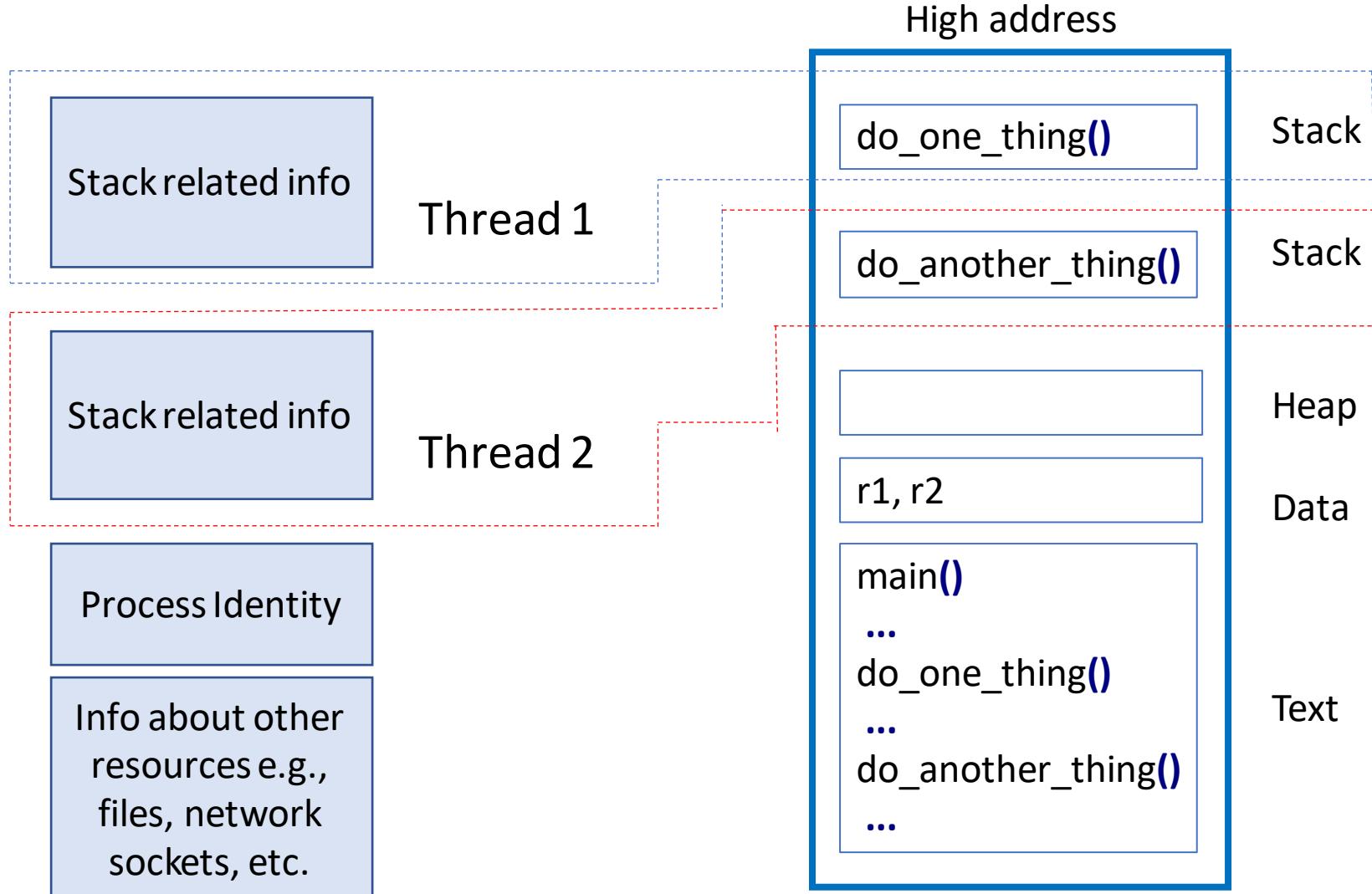
Stack

Heap

Data

Text

Program as a single process with two threads



C program as a process with two threads, each executing a function (or task) concurrently in one of the two stack regions. Each thread has its own copy of stack related info. Both threads can refer to common Heap, global Data and other resources such as opened files, sockets etc.

Using Pthreads library in C

- In your C program you need to `#include <pthread.h>`
- There are many functions related to pthreads.
- On a UNIX-based system, a list of these functions can typically be obtained with the command `man -k pthread`
- To know about a specific function see the man-page.
- When linking, you must link to the pthread library using compilation flag `-lpthread`
Example: `gcc -lpthread file.c`
or
`gcc file.c -lpthread`

Spawning a thread using pthread_create()

- A thread is created using pthread_create()

The syntax is:

```
int pthread_create(  
    pthread_t *thread_id, // ID number for thread  
    const pthread_attr_t *attr, // controls thread attributes  
    void *(*function)(void *), // function to be executed  
    void *arg // argument of function  
)
```

- pthread_create() returns 0 if thread creation is successful
- Otherwise it returns a nonzero value to indicate an error.

Spawning a thread using `pthread_create()`

- We will use default attributes set by the system
So, `*attr` gets replaced by **NULL**
The easier syntax is:

```
int pthread_create(  
    pthread_t *thread_id, // ID number for thread  
    NULL, // we set it to default thread attributes  
    void *(*function)(void *), // function to be executed  
    void *arg // argument of function  
);
```

Spawning a thread using pthread_create()

- A thread is created using pthread_create()

The syntax is:

```
int pthread_create(  
    pthread_t *thread_id, // ID number for thread  
    const pthread_attr_t *attr, // controls thread attributes  
    void *(*function)(void *), // function to be executed  
    void *arg // argument of function  
,
```

Example of functions that can be passed to pthread_create()

void *foo1();	✓
void foo2(int *);	✓
int *foo3(int *);	✓
int *foo4(int *, int*);	✗

Note: functions with multiple arguments

- Consider the function with multiple arguments

```
T *foo (T *, T *);
```

where T represents any data type.

- Solution:** Pack all arguments into a compound object and create a wrapper function which takes the compound argument and then unpacks inside before passing the unpacked arguments to foo()

```
typedef struct Compound{  
    T *a, *b;  
} Compound;
```

```
T * foo_wrapper(Compound *c){// This can be passed to pthread_create  
    T *d;  
    d=foo(c->a, c->b);  
}
```

Example: program with threads

File pthread0.c

```
void do_one_thing()
{
    int i, j, x;

    for (i = 0; i < 200; i++) {
        printf("doing one thing\n");
    }
}

void do_another_thing()
{
    int i, j, x;

    for (i = 0; i < 200; i++) {
        printf("doing another \n");
    }
}
```

```
int main(void)
{
    pthread_t thread1, thread2;

    pthread_create(&thread1,
                  NULL,
                  (void *) do_one_thing,
                  NULL);

    pthread_create(&thread2,
                  NULL,
                  (void *) do_another_thing,
                  NULL);
    sleep(1); // sleeps 1s
    return 0;
}
```

Program flow thread0.c

Main thread



```
int main(void)
{
    pthread_t thread1, thread2;

    pthread_create(&thread1,
                  NULL,
                  (void *) do_one_thing,
                  NULL);

    pthread_create(&thread2,
                  NULL,
                  (void *) do_another_thing,
                  NULL);
    sleep(1); // sleeps 1s
    return 0;
}
```

1. Initially, only the main thread is present.

Program flow thread0.c

Main thread

Main thread spawns thread1

thread1

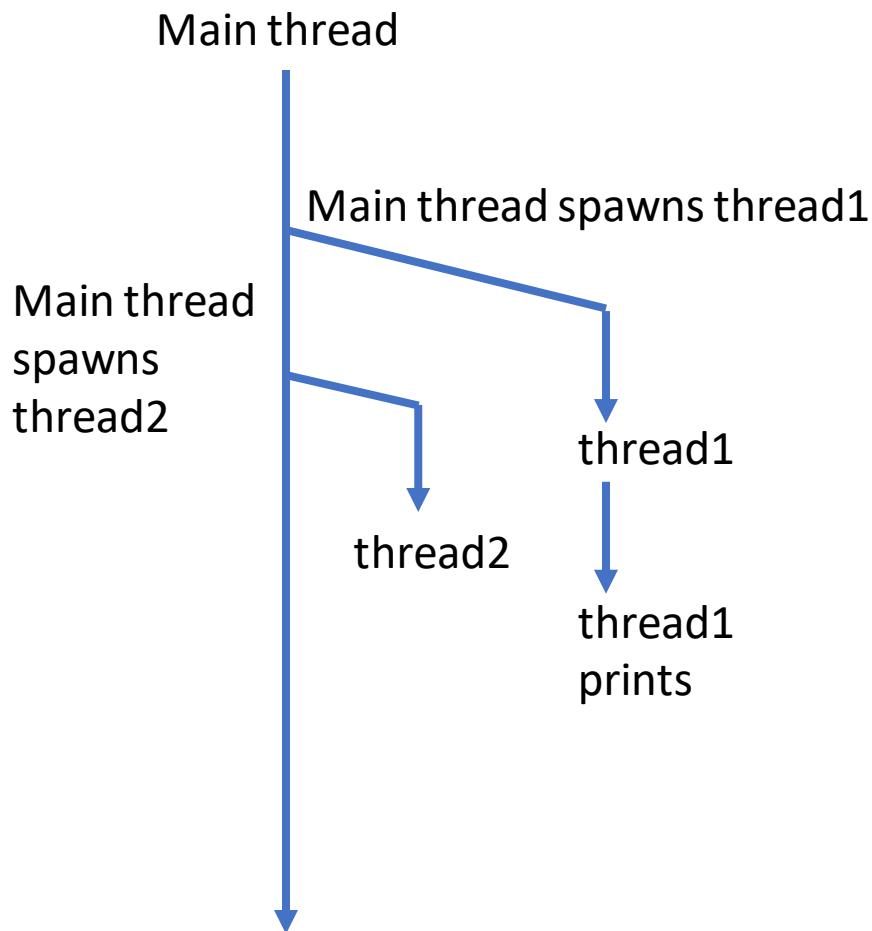
2. Main thread spawns thread1

```
int main(void)
{
    pthread_t thread1, thread2;

    pthread_create(&thread1,
                  NULL,
                  (void *) do_one_thing,
                  NULL);

    pthread_create(&thread2,
                  NULL,
                  (void *) do_another_thing,
                  NULL);
    sleep(1); // sleeps 1s
    return 0;
}
```

Program flow thread0.c



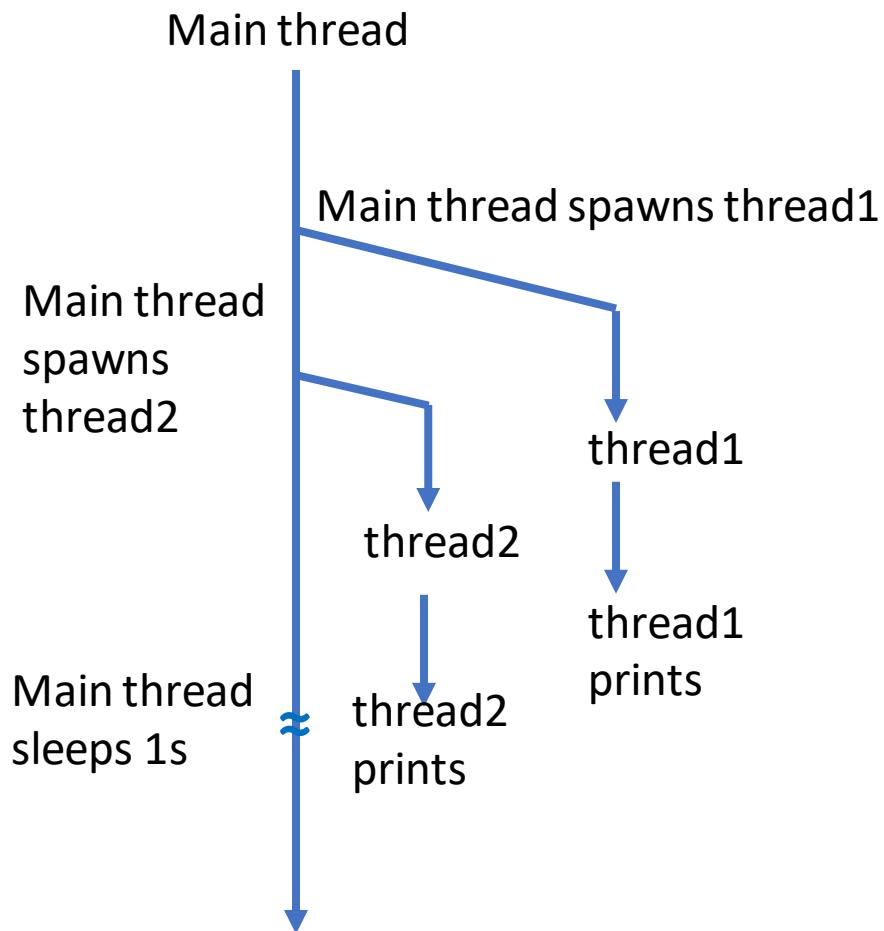
3. Main thread spawns thread2.
4. It might happen that thread1 has started printing

```
int main(void)
{
    pthread_t thread1, thread2;

    pthread_create(&thread1,
                  NULL,
                  (void *) do_one_thing,
                  NULL);

    pthread_create(&thread2,
                  NULL,
                  (void *) do_another_thing,
                  NULL);
    sleep(1); // sleeps 1s
    return 0;
}
```

Program flow thread0.c



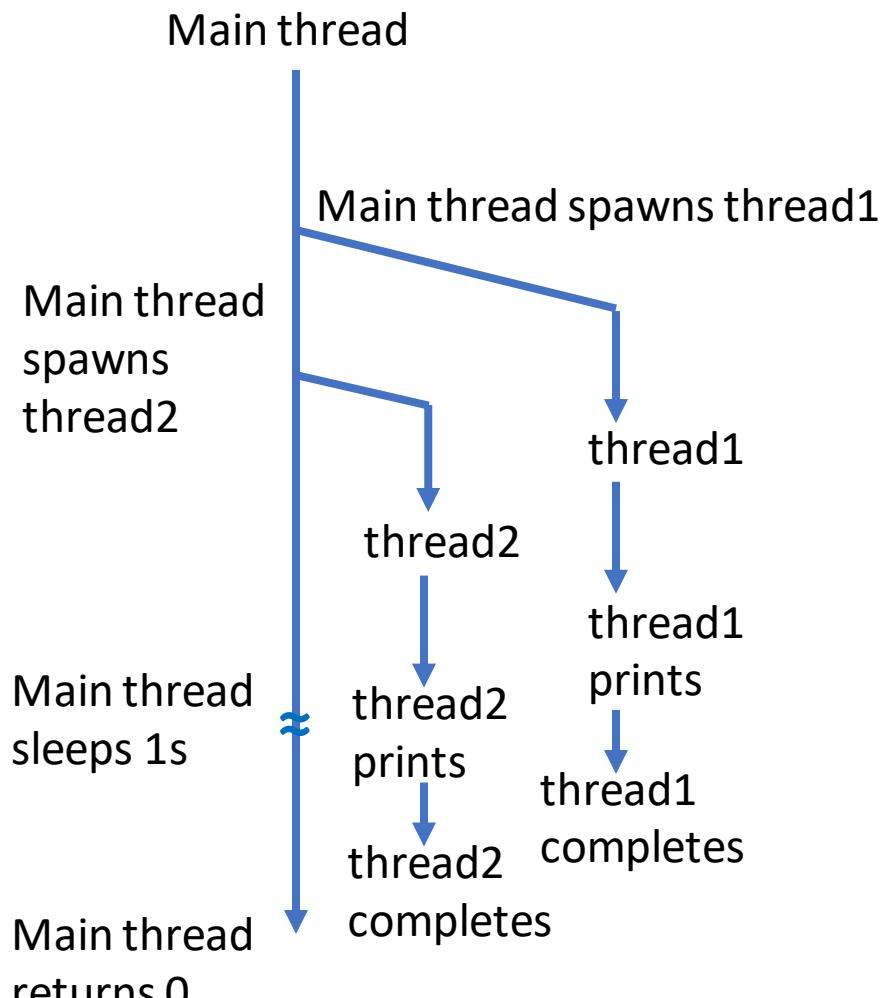
```
int main(void)
{
    pthread_t thread1, thread2;

    pthread_create(&thread1,
                  NULL,
                  (void *) do_one_thing,
                  NULL);

    pthread_create(&thread2,
                  NULL,
                  (void *) do_another_thing,
                  NULL);
    sleep(1); // sleeps 1s
    return 0;
}
```

5. While main thread goes for a sleep of 1s,
the other two threads continue with printing and finally completes.

Program flow thread0.c



```
int main(void)
{
    pthread_t thread1, thread2;

    pthread_create(&thread1,
                  NULL,
                  (void *) do_one_thing,
                  NULL);

    pthread_create(&thread2,
                  NULL,
                  (void *) do_another_thing,
                  NULL);
    sleep(1); // sleeps 1s
    return 0;
}
```

6. Finally the main thread finishes and returns.

Program flow thread0.c with // sleep(1)

```
int main(void)
{
    pthread_t thread1, thread2;

    pthread_create(&thread1,
                  NULL,
                  (void *) do_one_thing,
                  NULL);

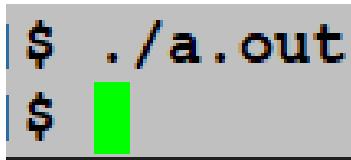
    pthread_create(&thread2,
                  NULL,
                  (void *) do_another_thing,
                  NULL);
    //sleep(1); // sleeps 1s
    return 0;
}
```

Program flow thread0.c with // sleep(1)

```
int main(void)
{
    pthread_t thread1, thread2;

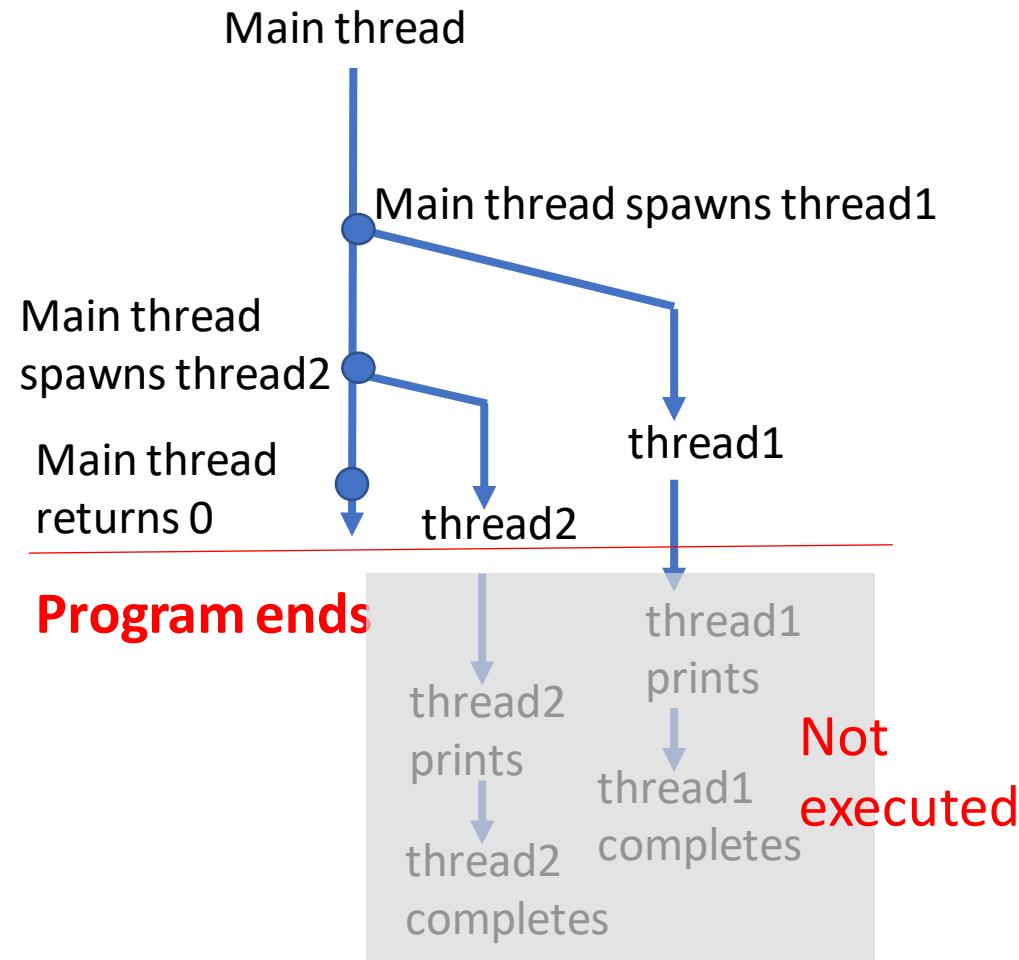
    pthread_create(&thread1,
                  NULL,
                  (void *) do_one_thing,
                  NULL);

    pthread_create(&thread2,
                  NULL,
                  (void *) do_another_thing,
                  NULL);
    //sleep(1); // sleeps 1s
    return 0;
}
```



The program does not print anything!

Program flow thread0.c with // sleep(1)

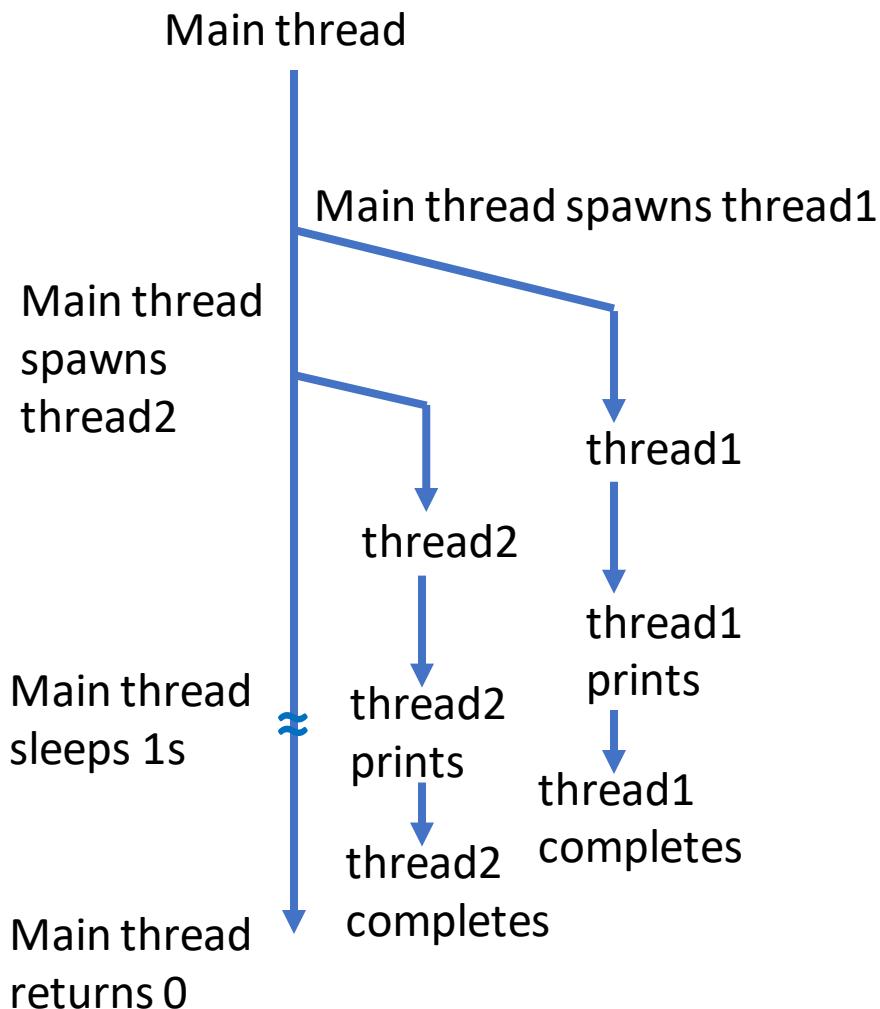


```
int main(void)
{
    pthread_t thread1, thread2;

    pthread_create(&thread1,
                  NULL,
                  (void *) do_one_thing,
                  NULL);

    pthread_create(&thread2,
                  NULL,
                  (void *) do_another_thing,
                  NULL);
    //sleep(1); // sleeps 1s
    return 0;
}
```

Program flow thread0.c



```
int main(void)
{
    pthread_t thread1, thread2;

    pthread_create(&thread1,
                  NULL,
                  (void *) do_one_thing,
                  NULL);

    pthread_create(&thread2,
                  NULL,
                  (void *) do_another_thing,
                  NULL);
    sleep(1); // sleeps 1s
    return 0;
}
```

We inserted a `sleep(1)` in main-thread and ‘hoped’ that Thread1 and Thread2 will finish by the time main-thread wakes up.

Shared data objects in a concurrent system

Cooperation between concurrent threads leads to the sharing of

- Global data objects,
- Heap objects
- Files, etc.

Lack of synchronization leads to chaos and wrong calculations.

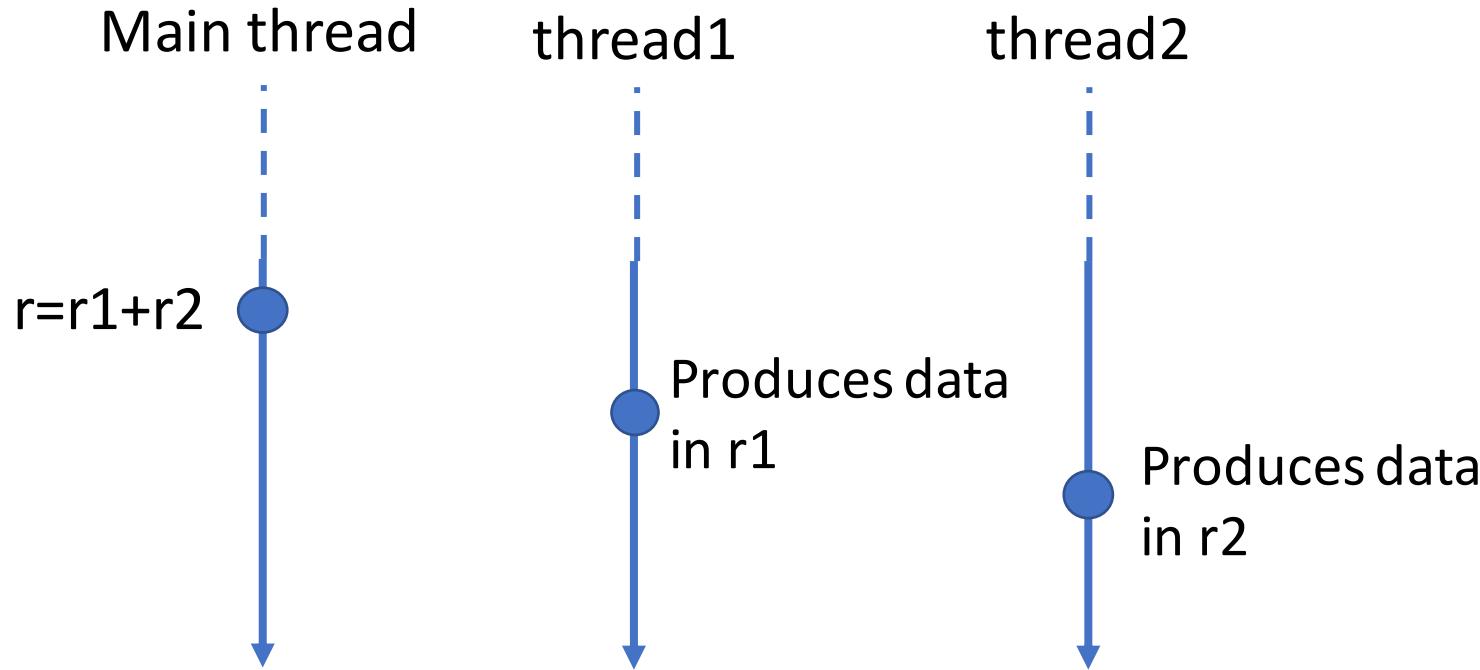
```
// global variables r1 and r2
int r1 = 0, r2 = 0;
main(){  // main thread
    int r;
    ...
    Call do_one_thing() in thread1;
    Call do_another_thing() in thread2;
    r = r1 + r2;
    ...
}
```

```
do_one_thing(...){ // in thread1
    ...
    Compute result in r1;
    ...
}

do_another_thing(...){ // in thread2
    ...
    Compute result in r2;
    ...
}
```

[Assume that only thread1 uses r1, only thread2 uses r2 and only main-thread writes to r.]

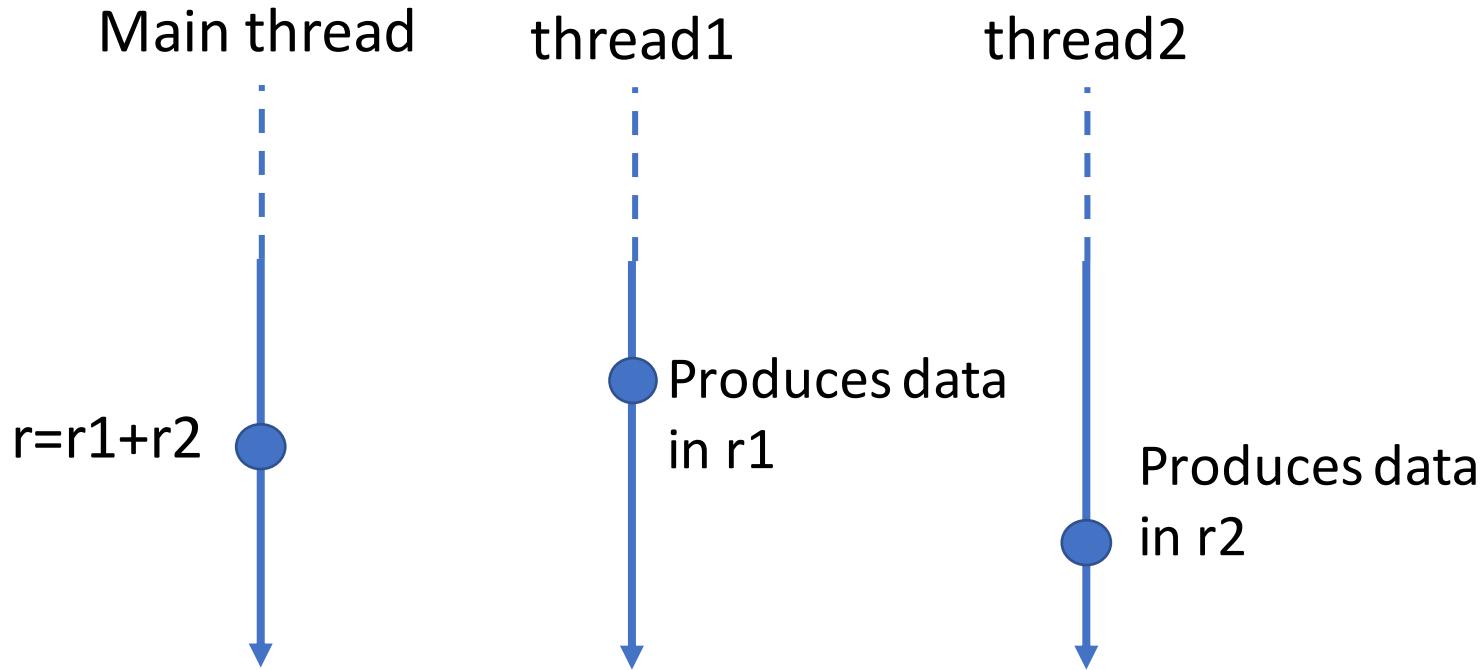
What can go wrong?



Scenario 1:

Main thread computes r_1+r_2 before thread1 and thread2 produces the results.
Both r_1 and r_2 will be wrong.

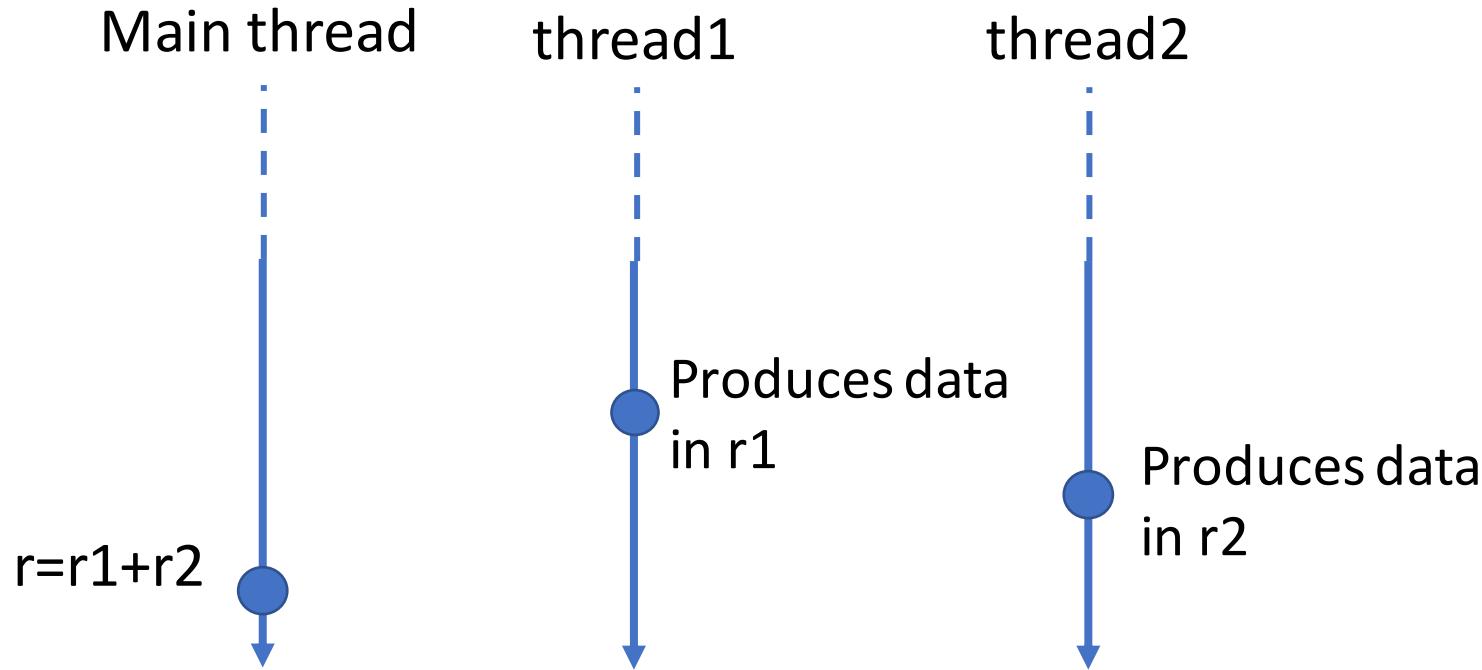
What can go wrong?



Scenario 2:

Main thread computes r_1+r_2 after thread1 but before thread2 produces.
So, r_1 will have correct but r_2 will have wrong values.
Thus r will be wrong.

What can go wrong?



Scenario 3:

Luckily, main thread computes $r1+r2$ after thread1 and thread2 produce.
Luckily, r will be correct.

Synchronization of threads

Synchronization in threads programming is used to make sure that some events happen in order.



Image source <https://theclassicalnovice.com/glossary/ensembles/>

Synchronization mechanism in Pthreads

Pthreads library provides three synchronization mechanisms:

- Joins
- Mutual exclusions
- Condition variables

Synchronizing threads using pthread_join()

- `pthread_join()` is a blocking function

The syntax is:

```
int pthread_join(  
    pthread_t thread_id, // ID of thread to "join"  
    void **value_pntr // address of function's return value  
)
```

- Again, we will set `value_pntr` to **NULL**

The syntax we will use is:

```
int pthread_join(  
    pthread_t thread_id, // ID of thread to "join"  
    NULL  
)
```

Example: Synchronizing threads using pthread_join()

```
int r1 = 0, r2 = 0;

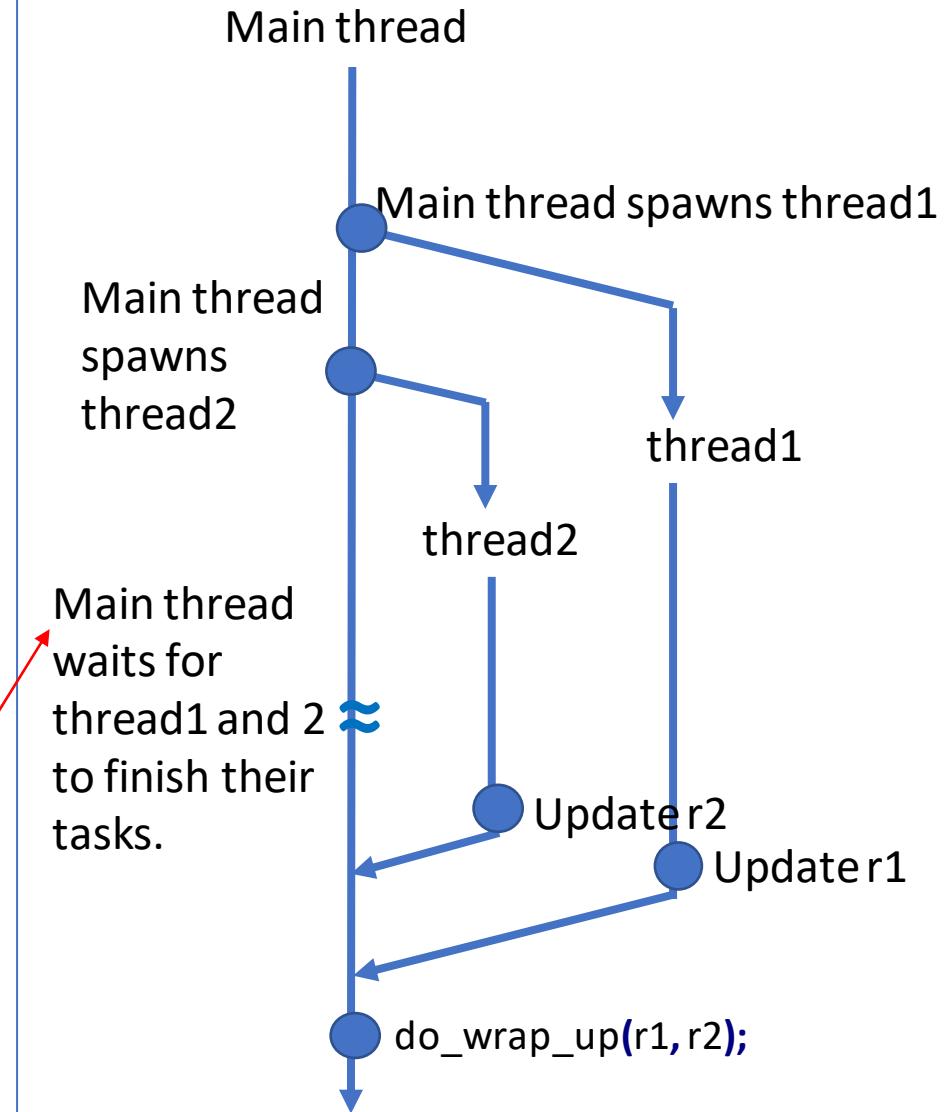
int main(void){
    pthread_t thread1, thread2;

    pthread_create(&thread1,
                  NULL,
                  (void *) do_one_thing,
                  (void *) &r1);

    pthread_create(&thread2,
                  NULL,
                  (void *) do_another_thing,
                  (void *) &r2);

    pthread_join(thread1, NULL);
    pthread_join(thread2, NULL);

    do_wrap_up(r1, r2);
    return 0;
}
```



Example: Synchronizing threads using pthread_join()

```
int r1 = 0, r2 = 0;

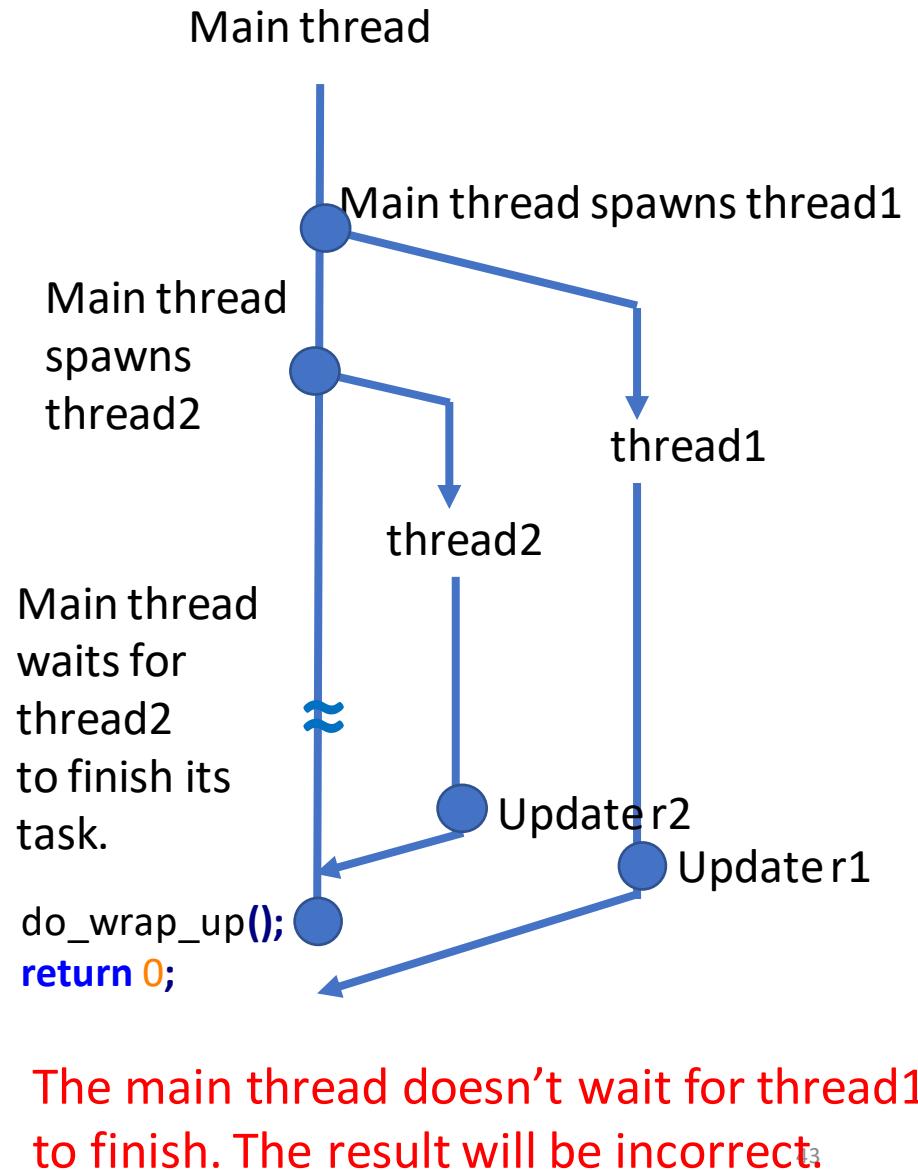
int main(void){
    pthread_t thread1, thread2;

    pthread_create(&thread1,
                  NULL,
                  (void *) do_one_thing,
                  (void *) &r1);

    pthread_create(&thread2,
                  NULL,
                  (void *) do_another_thing,
                  (void *) &r2);

    //pthread_join(thread1, NULL);
    pthread_join(thread2, NULL);

    do_wrap_up(&r1, &r2);
    return 0;
}
```



What did we assume in the previous synchronization example?

```
// global var r1 and r2  
int r1 = 0, r2 = 0;  
main(){ // main thread  
    int r;  
    ...  
    Call do_one_thing() in thread1;  
    Call do_another_thing() in thread2;  
    r = r1 + r2;  
    ...  
}
```

```
do_one_thing(...){ // in thread1  
    ...  
    Compute result in r1;  
    ...  
}  
  
do_another_thing(...){ // in thread2  
    ...  
    Compute result in r2;  
    ...  
}
```

We **assumed** that only thread1 uses r1, only thread2 uses r2 and only main-thread writes to r after thread1 and 2 finish.

What will happen if several threads try to read/write a shared variable?

Example: Several threads update a shared data

```
void *functionC();
int counter = 0;

main(){
    int rc1, rc2;
    pthread_t thread1, thread2;

    // Two threads execute functionC() concurrently
    pthread_create(&thread1, NULL, &functionC, NULL);
    pthread_create(&thread2, NULL, &functionC, NULL);

    pthread_join(thread1, NULL);
    pthread_join(thread2, NULL);
    return 0;
}

void *functionC(){
    counter++;
    printf("Counter value: %d\n", counter);
}
```

We expect the shared data ‘counter’ to be updated by one thread at a time and thus expect the program to print:

1
2

What can go wrong?

Example: Several threads update a shared data

```
void *functionC();  
int counter = 0;  
  
main(){  
    int rc1, rc2;  
    pthread_t thread1, thread2;  
  
    // Two threads execute functionC() concurrently  
    pthread_create(&thread1, NULL, &functionC, NULL);  
    pthread_create(&thread2, NULL, &functionC, NULL);  
  
    pthread_join(thread1, NULL);  
    pthread_join(thread2, NULL);  
    return 0;  
}  
  
void *functionC(){  
    counter++;  
    printf("Counter value: %d\n", counter);  
}
```

Scenario 1:

counter=0

thread1

counter++;

counter=1

thread2

counter++;

counter=2

Program prints:

1

2

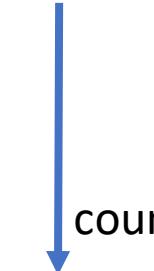
Example: Several threads update a shared data

```
void *functionC();  
int counter = 0;  
  
main(){  
    int rc1, rc2;  
    pthread_t thread1, thread2;  
  
    // Two threads execute functionC() concurrently  
    pthread_create(&thread1, NULL, &functionC, NULL);  
    pthread_create(&thread2, NULL, &functionC, NULL);  
  
    pthread_join(thread1, NULL);  
    pthread_join(thread2, NULL);  
    return 0;  
}  
  
void *functionC(){  
    counter++;  
    printf("Counter value: %d\n", counter);  
}
```

Scenario 2:

counter=0

thread1



thread2

counter++;

counter=1

counter=2

Program prints:

1

2

Example: Several threads update a shared data

```
void *functionC();  
int counter = 0;  
  
main(){  
    int rc1, rc2;  
    pthread_t thread1, thread2;  
  
    // Two threads execute functionC() concurrently  
    pthread_create(&thread1, NULL, &functionC, NULL);  
    pthread_create(&thread2, NULL, &functionC, NULL);  
  
    pthread_join(thread1, NULL);  
    pthread_join(thread2, NULL);  
    return 0;  
}  
  
void *functionC(){  
    counter++;  
    printf("Counter value: %d\n", counter);  
}
```

Scenario 3:

counter=0

thread1

counter++;

counter=1

thread2

counter++;

counter=1

Both threads race to update the shared data at the same time.

Program prints:

1

1

Data inconsistencies due to race conditions

Race condition and its prevention

- A race condition often occurs when two or more threads need to perform operations on the same data, but the results of computations depend on the order in which these operations are performed.
- The problem can be solved if we can enforce **mutual exclusion**
→ Threads get exclusive access to the shared resource in turn
- The Pthreads library offers ‘**mutex**’ objects to enforce exclusive access by a thread to a variable or a set of variables.

Mutual exclusion in Pthread: mutex

- Syntax for declaration and initialization of a mutex object is

```
pthread_mutex_t mutex1 = PTHREAD_MUTEX_INITIALIZER;
```

This statement will create a mutex object ‘mutex1’ and initialize it with the default characteristics.

- Generally mutex objects are declared as global.
- The following shows the use of mutex for serializing access to a shared resource.

```
...
pthread_mutex_lock( &mutex1 );
counter++;
pthread_mutex_unlock( &mutex1 );
...
```

Threads access ‘counter’ serially one after another.

Several threads update a shared data using mutex lock/unlock

```
pthread_mutex_t mutex1 = PTHREAD_MUTEX_INITIALIZER;  
void *functionC();  
int counter = 0;  
main(){  
    int rc1, rc2;  
    pthread_t thread1,thread2;  
  
    // Two threads execute functionC() concurrently  
    pthread_create(&thread1,NULL,&functionC,NULL);  
    pthread_create(&thread2,NULL,&functionC,NULL);  
  
    pthread_join(thread1,NULL);  
    pthread_join(thread2,NULL);  
    return 0;  
}  
void *functionC(){  
    pthread_mutex_lock(&mutex1);  
    counter++;  
    printf("Counter value:%d\n",counter);  
    pthread_mutex_unlock(&mutex1);  
}
```

counter=0

Thread1 obtains
the mutex lock



counter++;

counter=1

Thread1 releases
the mutex lock



Thread2 obtains
the mutex lock

counter++;

counter=2

Thread2 releases
the mutex lock



Several threads update a shared data using mutex lock/unlock

```
pthread_mutex_t mutex1 = PTHREAD_MUTEX_INITIALIZER;  
void *functionC();  
int counter = 0;  
main(){  
    int rc1, rc2;  
    pthread_t thread1,thread2;  
  
    // Two threads execute functionC() concurrently  
    pthread_create(&thread1,NULL,&functionC,NULL);  
    pthread_create(&thread2,NULL,&functionC,NULL);  
  
    pthread_join(thread1,NULL);  
    pthread_join(thread2,NULL);  
    return 0;  
}  
void *functionC(){  
    pthread_mutex_lock(&mutex1);  
    counter++;  
    printf("Counter value:%d\n",counter);  
    pthread_mutex_unlock(&mutex1);  
}
```

counter=0

Thread2 obtains
the mutex lock



counter++;

counter=1

Thread2 releases
the mutex lock



Thread1 obtains
the mutex lock

counter++;

counter=2

Thread1 releases
the mutex lock



The order may
vary with time

Exclusive access to a set of shared objects

We can also serialize access to a set of shared objects.

```
pthread_mutex_lock( &mutex1 );
Access shared object1;
Access shared object2;
...
pthread_mutex_unlock( &mutex1 );
```

Critical region

Threads will access {object1, object2, ...} serially.

The code segment that resides between mutex_lock() and mutex_unlock() is called ‘critical region’.

Critical region is executed serially by the threads.

Mutual exclusion: Pitfalls

```
//Thread1
void *do_one_thing(){
    ...
    pthread_mutex_lock(&mutex1);
    pthread_mutex_lock(&mutex2);
    ...
    ...
    pthread_mutex_lock(&mutex2);
    pthread_mutex_lock(&mutex1);
    ...
}
}
```

```
//Thread2
void *do_another_thing(){
    ...
    pthread_mutex_lock(&mutex2);
    pthread_mutex_lock(&mutex1);
    ...
    ...
    pthread_mutex_lock(&mutex1);
    pthread_mutex_lock(&mutex2);
    ...
}
}
```

Thread1 obtains mutex1

Thread2 obtains mutex2

Thread1 waits to obtain mutex2

Thread2 waits to obtain mutex1

Both threads get stalled indefinitely
→ Situation is known as Deadlock

pthread_mutex_trylock()

Syntax is:

```
int pthread_mutex_trylock(pthread_mutex_t *mutex);
```

- Tries to lock a mutex object.
- If the mutex object is available, then it is locked and 0 is returned.
- Otherwise, the function call returns nonzero. It **will not wait** for the object to be freed.

Avoiding mutex deadlock

- Thread tries to acquire mutex2
- and if it fails, then it releases mutex1 to avoid deadlock

```
...
pthread_mutex_lock(&mutex1);
// Now test if already locked
while ( pthread_mutex_trylock(&mutex2) ) {
    // unlock resource to avoid deadlock
    pthread_mutex_unlock(&mutex1);
    ...
    // wait here for some time
    ...
    pthread_mutex_lock(&mutex1);
}
count++;
pthread_mutex_unlock(&mutex1);
pthread_mutex_unlock(&mutex2);
...
```

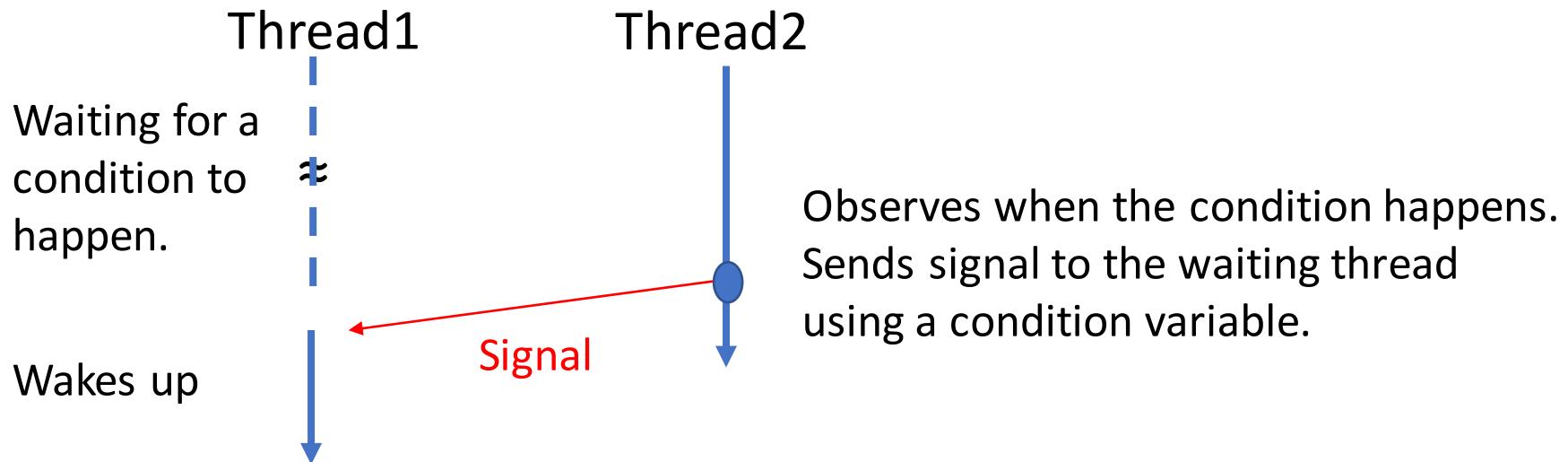
Pthread Synchronization so far

We have used the following synchronization functions

- **Join:** It is used to block the calling thread until the specified thread finishes.
- **Mutual Exclusion:** It is used to serialize access to shared data

Condition variables

- A condition variable is used to synchronize threads based on the value of data (i.e., a condition).
- One thread waits until data reaches a particular value or until a certain event occurs.
- Then another active thread sends a signal when the event occurs.
- Receiving the signal, the waiting thread wakes up and becomes active.



Condition variables: Syntax

- A condition variable is a variable of type pthread_cond_t
- Creation and initialization

```
pthread_cond_t condition_cond = PTHREAD_COND_INITIALIZER;
```

- A thread goes to waiting state based on ‘condition to happen’ by:

```
pthread_cond_wait( &condition_cond, &condition_mutex );
```

Function takes two arguments: the condition variable and a mutex variable associated with the condition variable.

- Waking thread based on condition

```
pthread_cond_signal( &condition_cond );
```

```
int count = 0;
#define COUNT_DONE 10
#define COUNT_HALT1 3
#define COUNT_HALT2 6

...
void *functionCount1(){
    for(;;){
        pthread_mutex_lock( &count_mutex );
        count++;
        printf("Counter value functionCount1: %d\n",count);
        pthread_mutex_unlock( &count_mutex );

        if(count >= COUNT_DONE) return(NULL);
    }
}
```

```
...
void *functionCount2(){
    for(;;){
        pthread_mutex_lock( &count_mutex );
        count++;
        printf("Counter value functionCount1: %d\n",count);
        pthread_mutex_unlock( &count_mutex );

        if(count >= COUNT_DONE) return(NULL);
    }
}
```

- `functionCount1()` and `functionCount2()` are run concurrently.
- They increment the shared variable ‘count’ serially
- The order in which the two functions increment ‘count’ is somewhat random.

Suppose, we want that only `functionCount2()` increments ‘count’ during the following condition

```
while( count >= COUNT_HALT1 && count <= COUNT_HALT2 )
```

```

pthread_mutex_t condition_mutex = PTHREAD_MUTEX_INITIALIZER;
void *functionCount1(){
    for(;;){
        pthread_mutex_lock( &condition_mutex );
        while( count >= COUNT_HALT1 && count <= COUNT_HALT2 ){
            pthread_cond_wait( &condition_cond, &condition_mutex );
        }
        pthread_mutex_unlock( &condition_mutex );

        pthread_mutex_lock( &count_mutex );
        count++;
        printf("Counter value functionCount1: %d\n",count);
        pthread_mutex_unlock( &count_mutex );

        if(count >= COUNT_DONE) return(NULL);
    }
}

```

- When `functionCount1()` sees for the first time that ‘count’ is in the mentioned range, it goes to wait state.
- `pthread_cond_wait()` releases the condition mutex.
- So, the condition variable can be used by the other thread now.

```

void *functionCount2(){
    for(;;){
        pthread_mutex_lock( &condition_mutex );
        if(count > COUNT_HALT2 ){
            pthread_cond_signal( &condition_cond );
        }
        pthread_mutex_unlock( &condition_mutex );

        pthread_mutex_lock( &count_mutex );
        count++;
        printf("Counter value functionCount2: %d\n",count);
        pthread_mutex_unlock( &count_mutex );

        if(count >= COUNT_DONE) return(NULL);
    }
}

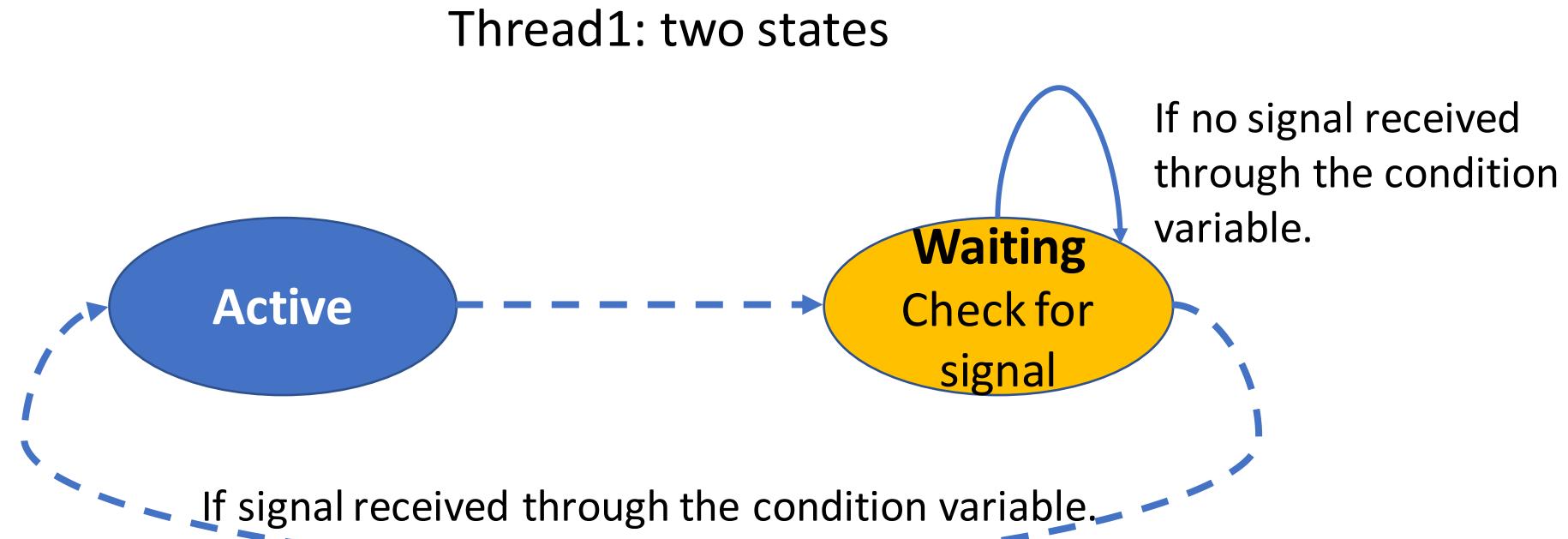
```

Only `functionCount2()` increments ‘count’ from `COUNT_HALT1` to `COUNT_HALT2`

After that, it ‘signals’ the waiting thread to wake up using the condition variable.

`functionCount2()` releases the condition mutex as the other thread will get it after waking up.

Why do we need a mutex with a condition variable?



- Presence of a signal is checked only in the ‘waiting’ state.
- If the signal arrives before Thread1 moves to the ‘waiting’ state, then the Thread1 will miss that
→ Consequence: Thread1 will wait indefinitely

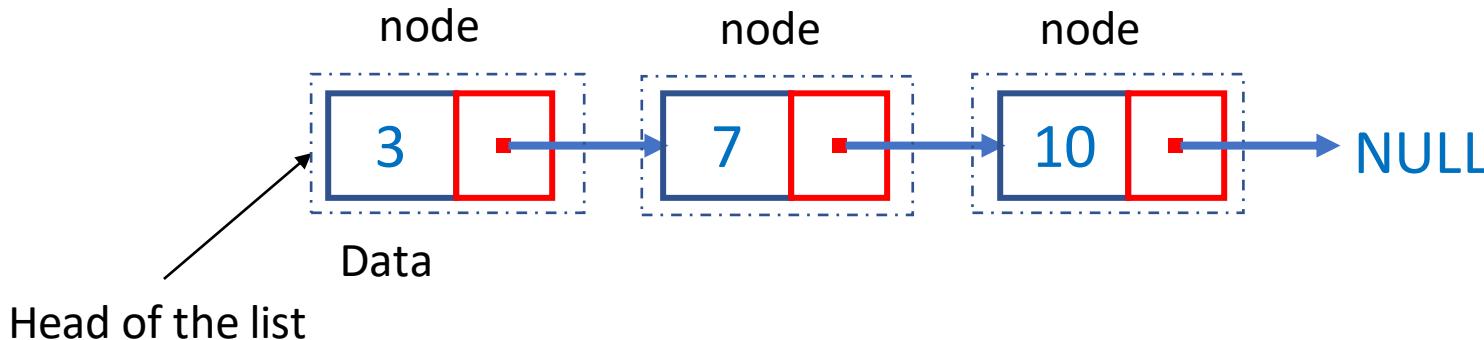
Condition mutex is used to ‘serialize’ the access of the condition variable in a proper way.

Application: Concurrent operations on a shared Linked List

Application: Concurrent operations on a shared Linked List

Consider a sorted linked list of integers with the following operations:

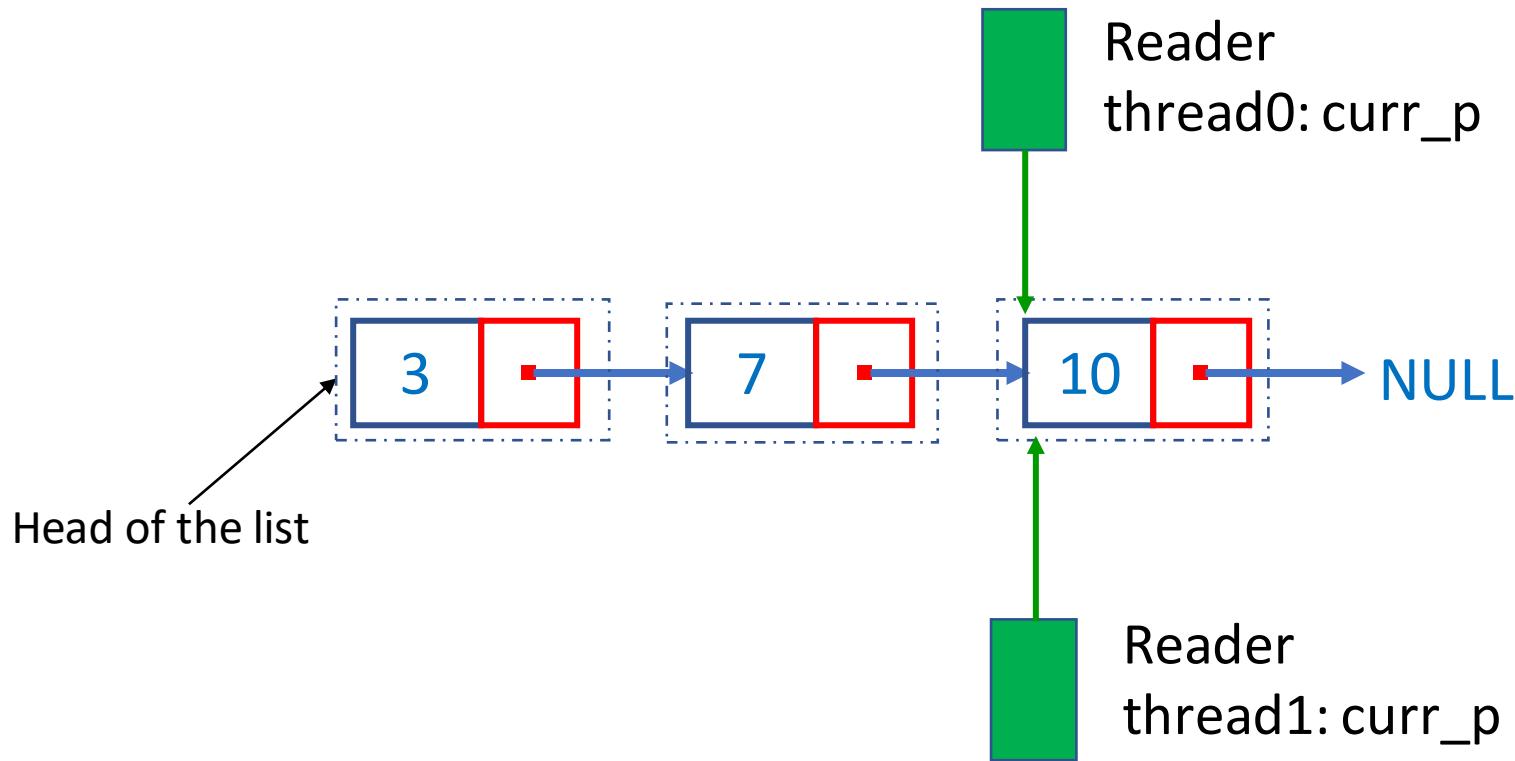
- Insert: inserts a new node maintaining the sorted order.
- Delete: deletes an existing node.
- Member: returns true/false depending on node present/absent.



Challenge: Multiple concurrent threads perform these operations on a shared linked list.

Simultaneous access by two threads

Two reader threads in operation.

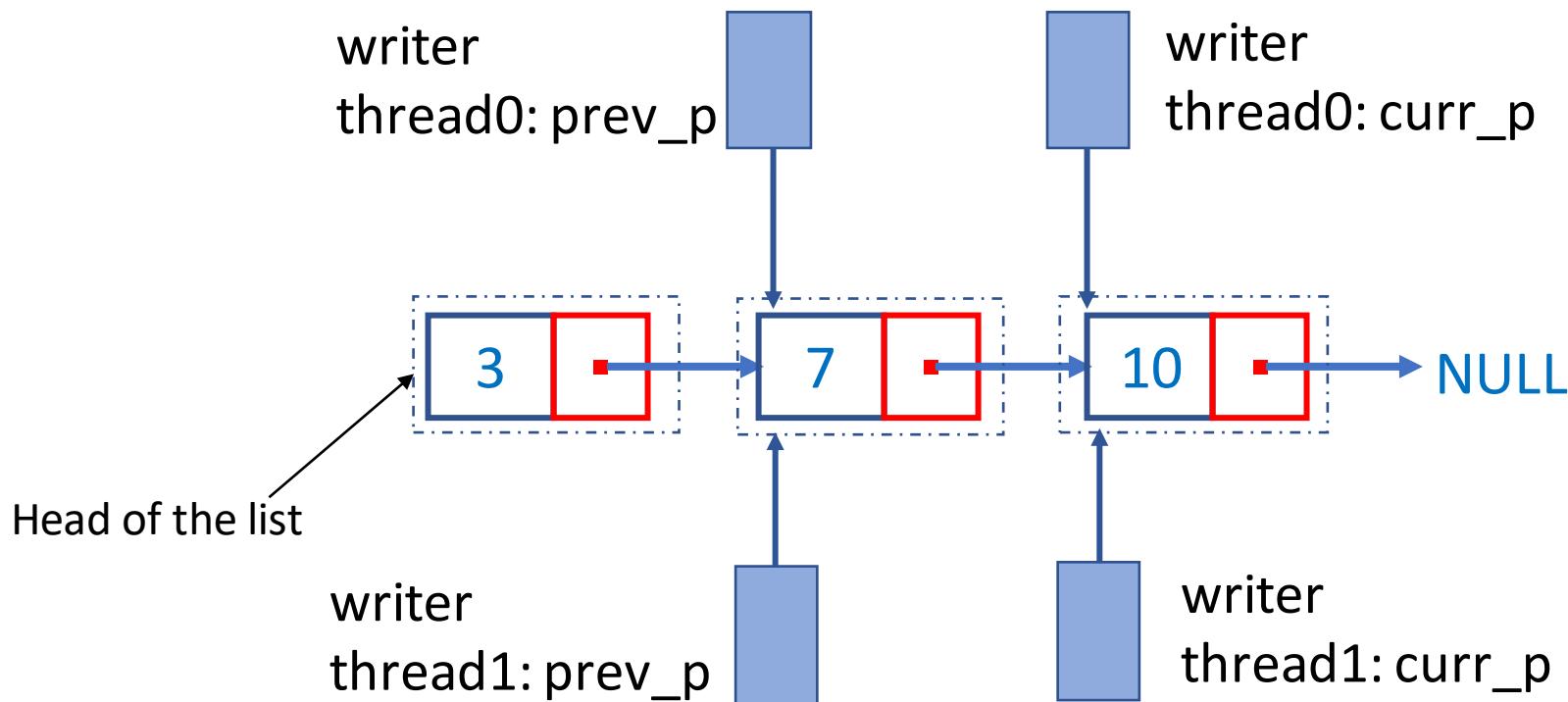


Multiple threads **can read** concurrently.

Simultaneous access by two threads

Two concurrent operations:

- Thread0 wants to insert node value 8 in the list.
- Thread1 wants to insert node value 9 in the list.

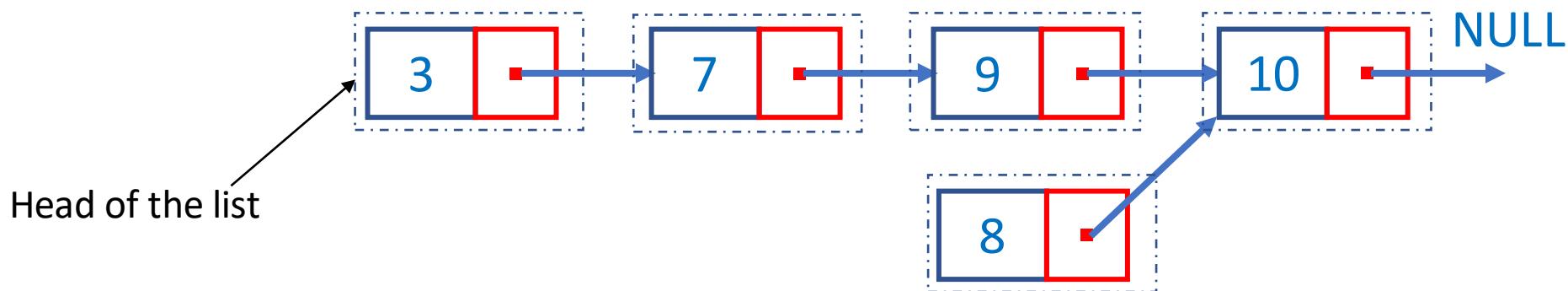


Multiple threads **cannot write** concurrently.

Simultaneous access by two threads

Two concurrent operations:

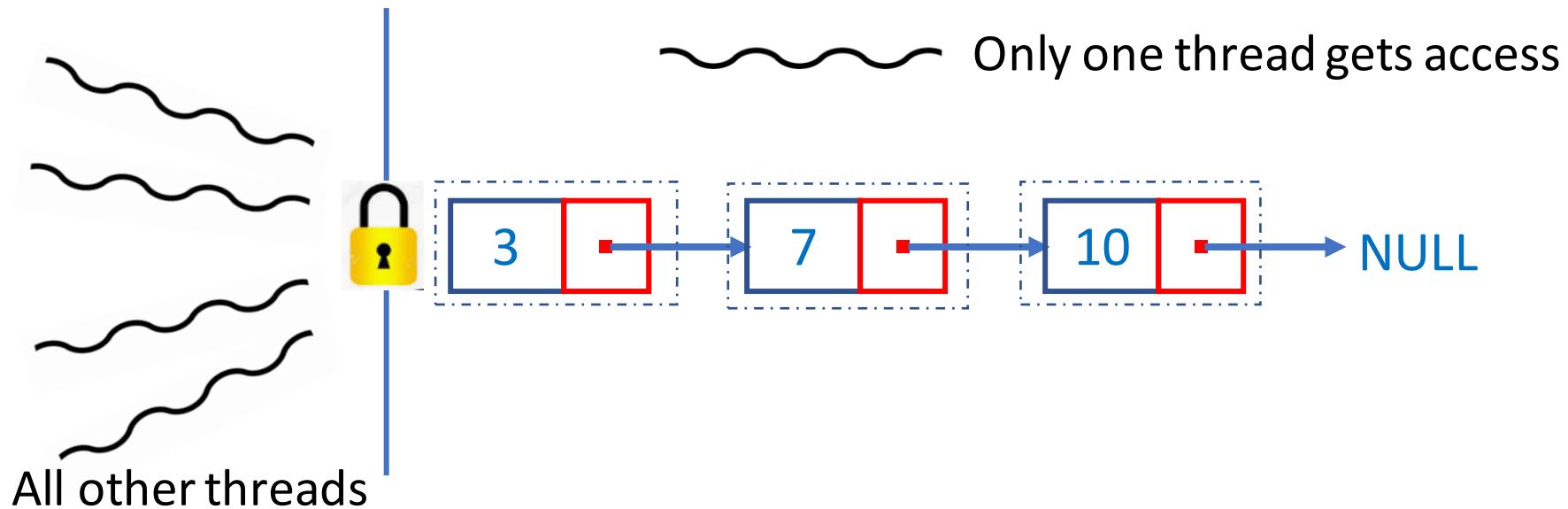
- Thread0 wants to insert node value 8 in the list.
- Thread1 wants to insert node value 9 in the list.



Multiple threads **cannot write** concurrently.

Solution1: Only one thread can access the list

- A naive solution is to simply lock the entire list to serialize access to the list.
- Serialization of access can be implemented using a mutex.



```
...
pthread_mutex_lock(&mutex1);
member(value0);
pthread_mutex_unlock(&mutex1);
```

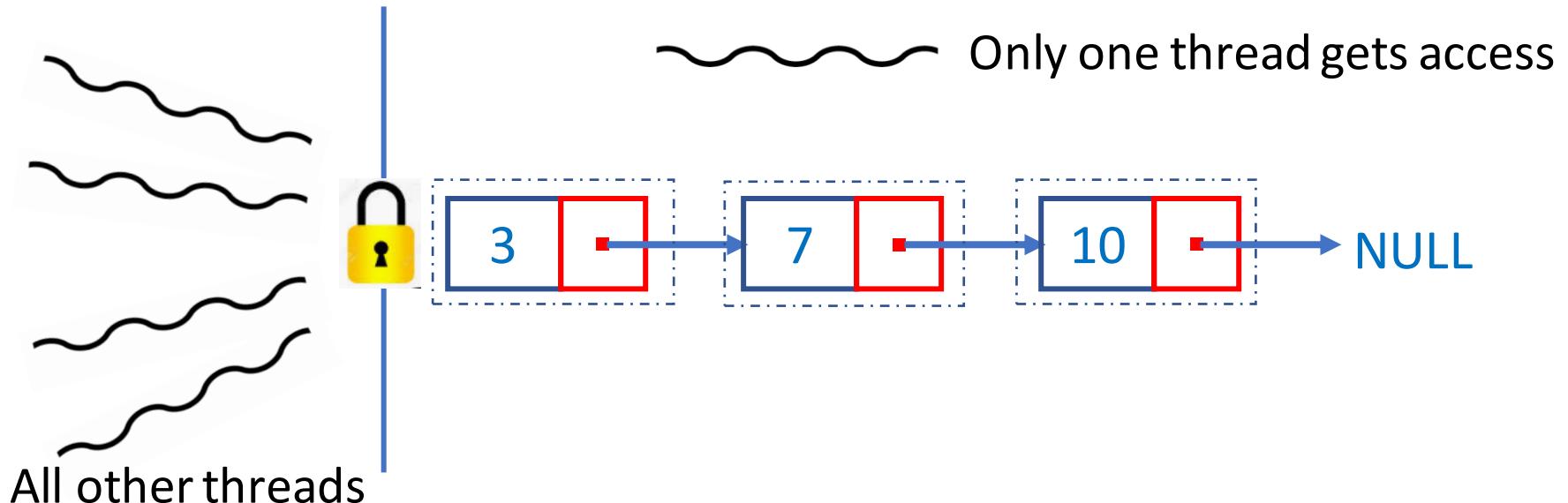
Thread0

```
...
pthread_mutex_lock(&mutex1);
insert(value1);
pthread_mutex_unlock(&mutex1);
...
```

Thread1

Issues with Solution1

- Only one thread gets access to the list.
- If vast majority of operations are ‘read’, then this approach fails to exploit parallelism.
- On the other hand, if most of the operations are ‘write’ then this approach may be the best and easy solution.

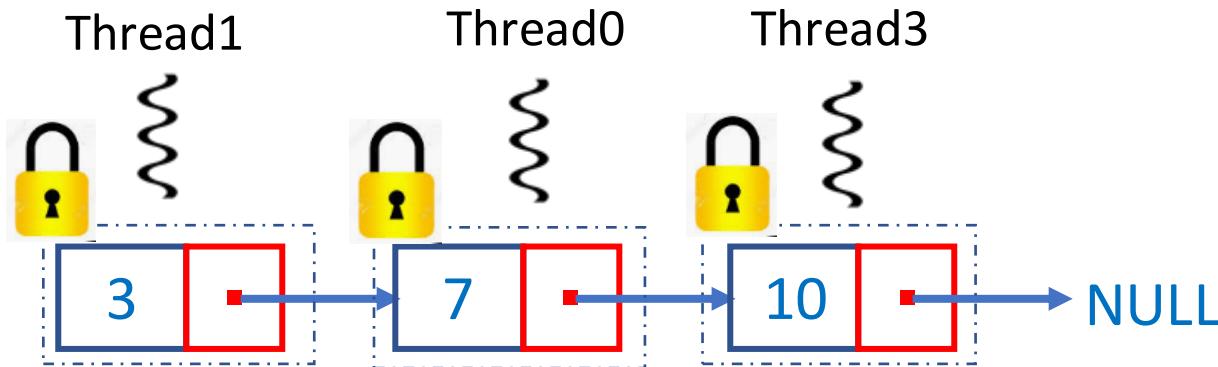


Solution2: Granular access to individual nodes

- Instead of locking the entire list, lock individual nodes.

- This gives granular access to the nodes.

Example: Thread-M accesses one node while Thread-N accesses another node.



- Implementation requires one mutex lock per node.

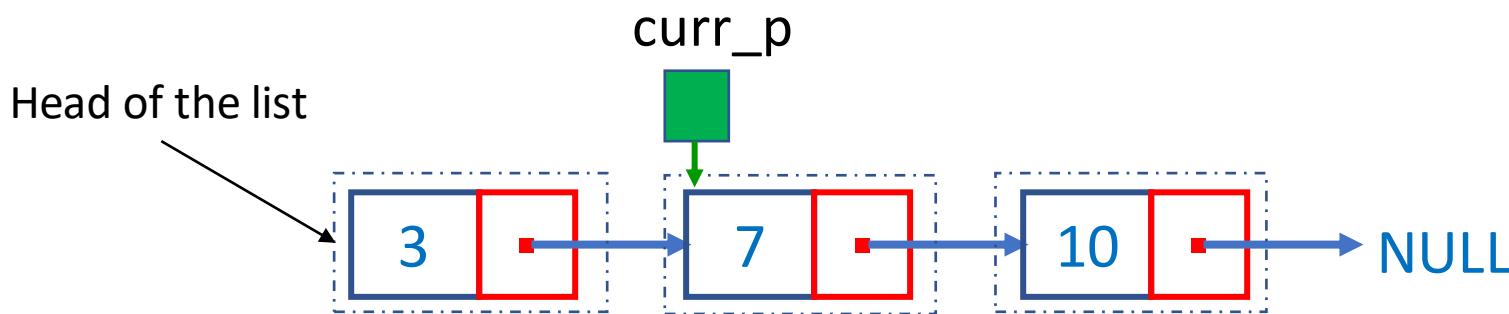
```
typedef struct Node{  
    int data;  
    struct Node *next;  
    pthread_mutex_t mutex;  
} Node;
```

With Solution2, code becomes a lot more complicated

Non-threaded Linked list: function Member()

Member() returns 1 if a node with the input ‘value’ is present in the list. Otherwise, it returns 0;

```
int Member(list *l, int value){  
    Node *curr_p = l->head;  
    while(curr_p!=NULL && curr_p->data < value)  
        curr_p = curr_p->next;  
    if(curr_p==NULL || curr_p->data > value)  
        return 0;  
    else  
        return 1;  
}
```



Implementation of Member() for Solution2

```
int Member(int value){  
    Node *curr_p;  
  
    pthread_mutex_lock(&head_mutex);  
    curr_p = head;  
    while(curr_p!=NULL&& curr_p->data < value){  
        if(curr_p->next != NULL)  
            pthread_mutex_lock(&(curr_p->next->mutex));  
        if(curr_p == head)  
            pthread_mutex_unlock(&head_mutex);  
  
        pthread_mutex_unlock(&(curr_p->mutex));  
        curr_p = curr_p->next;  
    }  
}
```

// Remaining part in the next slide

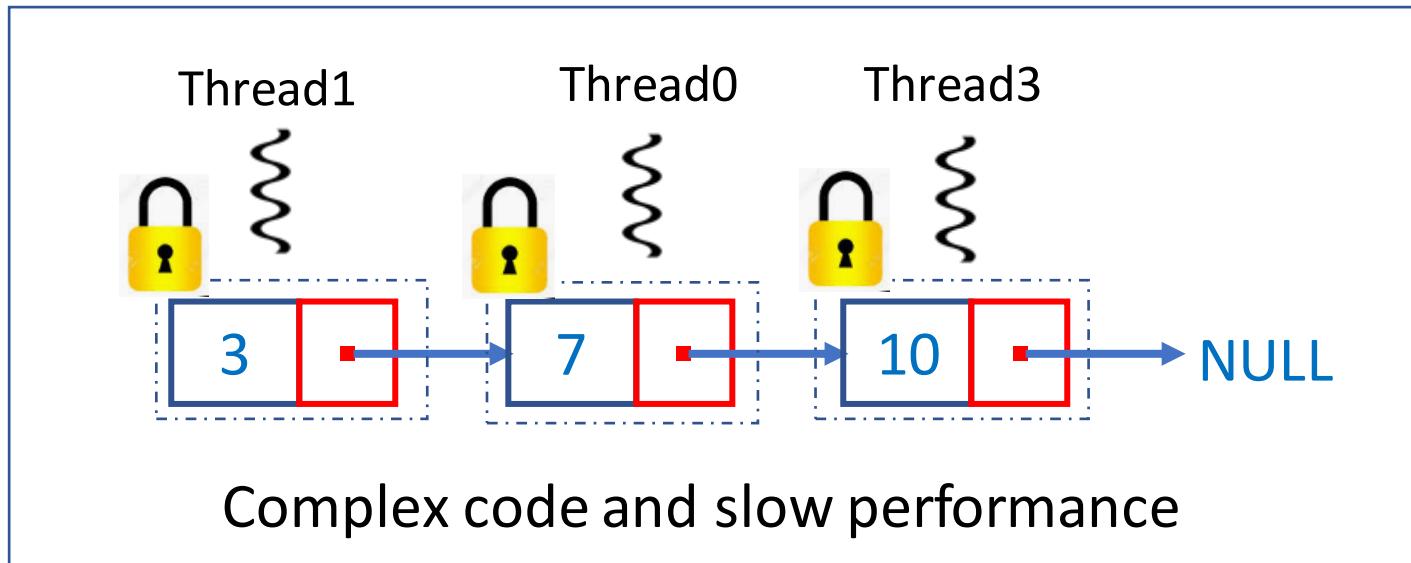
Implementation of Member() for Solution2

```
// continuation from the previous slide
if(curr_p == NULL || curr_p->data > value){
    if(curr_p == head)
        pthread_mutex_unlock(&head_mutex);
    if(curr_p != NULL)
        pthread_mutex_unlock(&(curr_p->mutex));
    return 0;
}
else{
    if(curr_p == head)
        pthread_mutex_unlock(&head_mutex);
    pthread_mutex_unlock(&(curr_p->mutex));
    return 1;
}
```

Source code of threaded linked list with one mutex per node is available at
https://www.csee.umbc.edu/~tsimo1/CMSC483/cs220/code/pth-rw/pth_linked_list_mult_mut.c

Issues with Solution2: Granular access to individual nodes

- With Solution2, simple Member() function becomes rather 'complex'.
- Every time a thread tries to access a node, it needs to
 - check if a mutex lock is available
 - then locking and unlocking of the mutex lock etc.
- Performance will be much slower.



Solution1 and Solution2: summary

- The first solution only allows one thread to access the entire list at any instant.
→ defeats the purpose of multi-threading
- The second only allows one thread to access any given node at any instant.
→ major performance problem and complicated code.

Can we have a simpler and more efficient multi-threaded linked list?

Pthreads provide another kind of lock known as ‘read-write lock’.

Read-write locks

- A read-write lock is declared and initialized as

```
pthread_rwlock_t lock = PTHREAD_RWLOCK_INITIALIZER;
```

- A read-write lock is somewhat like a mutex except that it provides two lock functions.

- for just reading

```
pthread_rwlock_rdlock(&lock);
```

- for read-write access

```
pthread_rwlock_wrlock(&lock);
```

- There is only one unlock function

```
pthread_rwlock_unlock(&lock);
```

Rules that read-write locks follow

Goal: allow multiple threads to read,
but allow only one thread to write.

```
pthread_rwlock_rdlock(){
```

If no other thread holds the lock, then get the lock.

Else if other threads hold the read-lock, then get the lock.

Else if another thread holds the write-lock, then wait.

```
}
```

```
pthread_rwlock_wrlock(){
```

If no other threads hold the read or write lock, then get the lock.

Else, wait for the lock.

```
}
```

Application of read-write lock to Linked list

```
...
pthread_rwlock_rdlock(&lock);
Member(value1);
pthread_rwlock_unlock(&lock);

pthread_rwlock_wrlock(&lock);
Insert(value2);
pthread_rwlock_unlock(&lock);
...
```

One thread

```
...
pthread_rwlock_wrlock(&lock);
Delete(value3);
pthread_rwlock_unlock(&lock);

pthread_rwlock_rdlock(&lock);
Member(value3);
pthread_rwlock_unlock(&lock);
...
```

Another concurrent thread

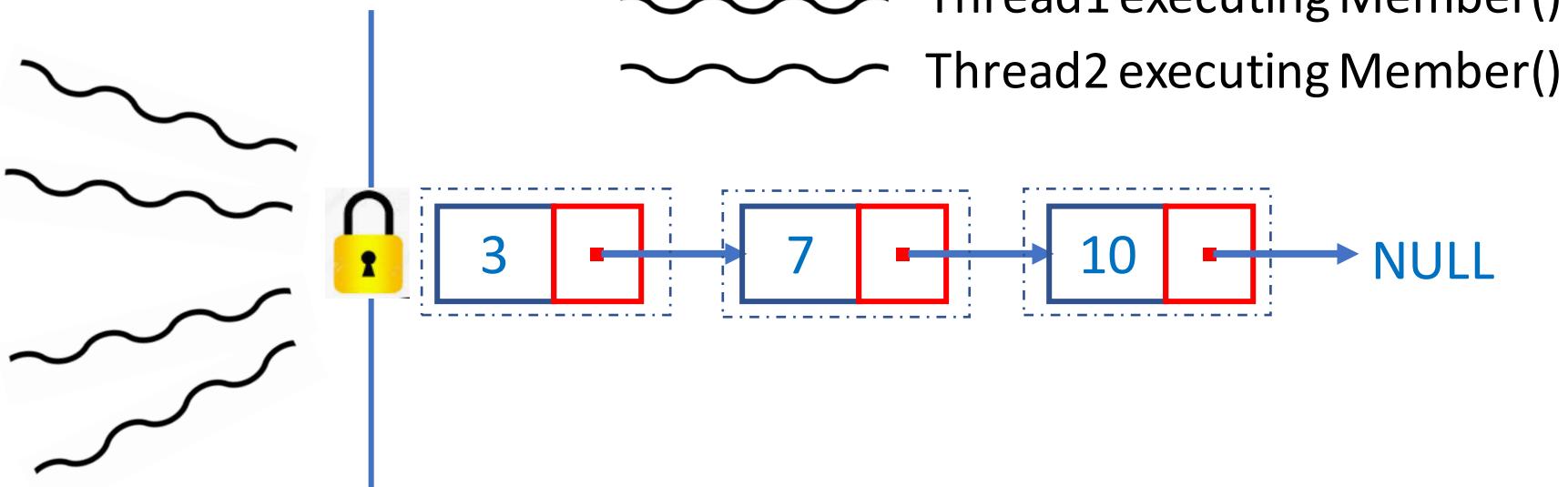
Multiple concurrent threads perform operations on the linked list.

- Read lock is used for functions that do not modify the list.
- Write lock is used for functions that modify the list.

Application of read-write lock to Linked list

Writer threads
wait for reader
threads to finish.

Multiple reader threads get concurrent access.



Application of read-write lock to Linked list

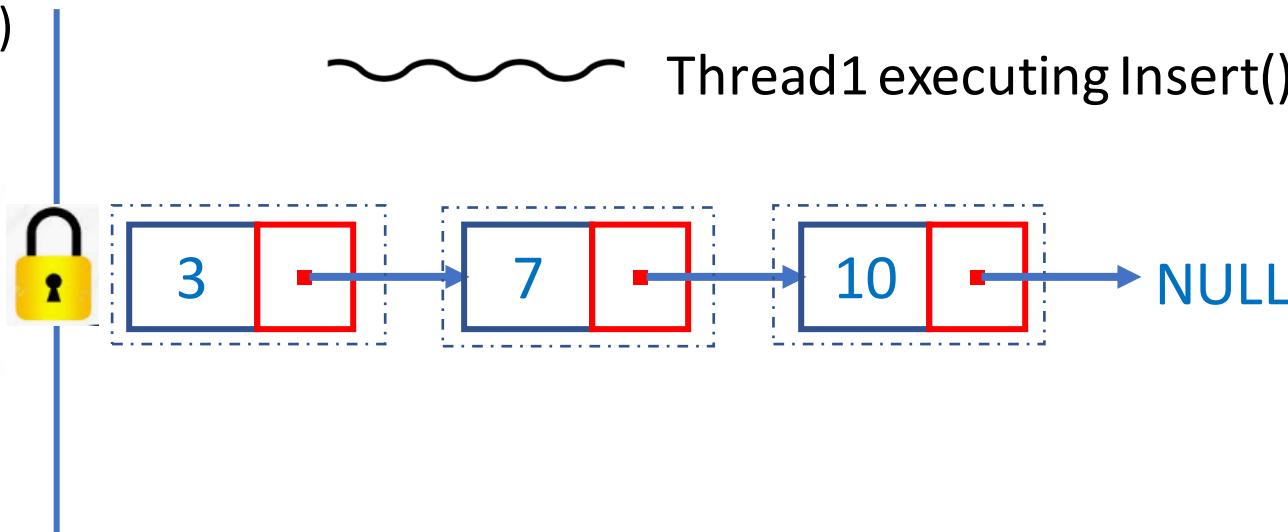
Other threads wait.

Only one writer thread gets exclusive access.

Thread2: Member()

Thread3: Insert()

Thread4: Delete()





Mario has life 10,000

Mario attacks using

- Fireball (attack point 5,000)
- Iceball (attack point 2,500)

Tortoise has life 8,000
It attacks Mario.



FireDragon has life 25,000
It attacks Mario.

Fireballs strengthen it.





```
...
short int fireballs_in_stock= <some value>;
short int iceballs_in_stock= <some value>;
...
throw_fireball(){
    fireballs_in_stock = fireballs_in_stock - 5000;
}
throw_iceball(){
    iceballs_in_stock = iceballs_in_stock - 2500;
}
```



```
short int life_tortoise = 8000;  
attack_mario(){  
    life_mario = life_mario - 2500;  
}  
attack_by_mario(){  
    if(fireball)  
        life_tortoise = life_tortoise - 5000;  
    else  
        life_tortoise = life_tortoise - 2500;  
}
```



```
short int life_firedragon = 25000;  
attack_mario(){  
    life_mario = life_mario - 5000;  
}  
attack_by_mario(){  
    if(fireball)  
        life_firedragon = life_firedragon + 5000;  
    else  
        life_firedragon = life_firedragon - 2500;  
}
```

If Mario attacks Firedragon with fireballs, then life of Firedragon improves

Mario faces Tortoise



Mario must throw 2 fireballs or 4 iceballs before Tortoise reaches Mario

```
short int life_tortoise = 8000;  
attack_mario(){  
    life_mario = life_mario - 3000;  
}  
attack_by_mario(){  
    if(fireball)  
        life_tortoise = life_tortoise - 5000;  
    else  
        life_tortoise = life_tortoise - 2500;  
}
```

Mario faces Firedragon



Mario must throw 10 iceballs before Firedragon kills Mario.

```
short int life_firedragon = 25000;  
attack_mario(){  
    life_mario = life_mario - 5000;  
}  
attack_by_mario(){  
    if(fireball)  
        life_firedragon = life_firedragon + 5000;  
    else  
        life_firedragon = life_firedragon - 2500;  
}
```

Mario faces Firedragon



Mario must throw 10 iceballs before Firedragon kills Mario.

**Can Mario survive till he throws 10 iceballs?
Is there a ‘cheat’?**

Signed integer

Let us consider 16-bit signed short integer

short int a=3;

0000 0000 0000 0011

How it is stored in computer

short int a=1;

0000 0000 0000 0001

How it is stored in computer

Signed integer

Let us consider 16-bit signed short integer

short int a=3;

0000 0000 0000 0011

How it is stored in computer

short int a=1;

0000 0000 0000 0001

How it is stored in computer

short int a=-1;

1111 1111 1111 1111

How it is stored in computer

short int a=-3;

1111 1111 1111 1101

How it is stored in computer

All negative integers have most significant bit = 1

Mario faces Firedragon



Initially



`short int life_firedragon = 25000;`

0110000110101000

Mario faces Firedragon



Initially



`short int life_firedragon = 25000;`

0110000110101000

Mario throws one fireball

`short int life_firedragon = 30000;`

0111010100110000

Mario faces Firedragon



Initially



`short int life_firedragon = 25000;`

0110000110101000

Mario throws one fireball

`short int life_firedragon = 30000;`

0111010100110000

Mario throws another fireball

`short int life_firedragon = 35000;`

1000100010111000

Mario faces Firedragon



Initially



`short int life_firedragon = 25000;`

0110000110101000

Mario throws one fireball

`short int life_firedragon = 30000;`

0111010100110000

Mario throws another fireball

`short int life_firedragon = 35000;`

1000100010111000

Most significant bit has become 1.

Recap: negative numbers have most significant bit 1.

→ Life of FireDragon is negative, it dies 😊

Operating Systems

Second part of Operating Systems and Systems Programming
Module

University of Birmingham

Eike Ritter

Overview

Operating Systems

Second part of the course: Operating Systems

Outline of lecture:

- What is an operating system?
- How to interact with the operating system
- Possible operating systems architectures
- How to write programs in the kernel
- Virtual machines

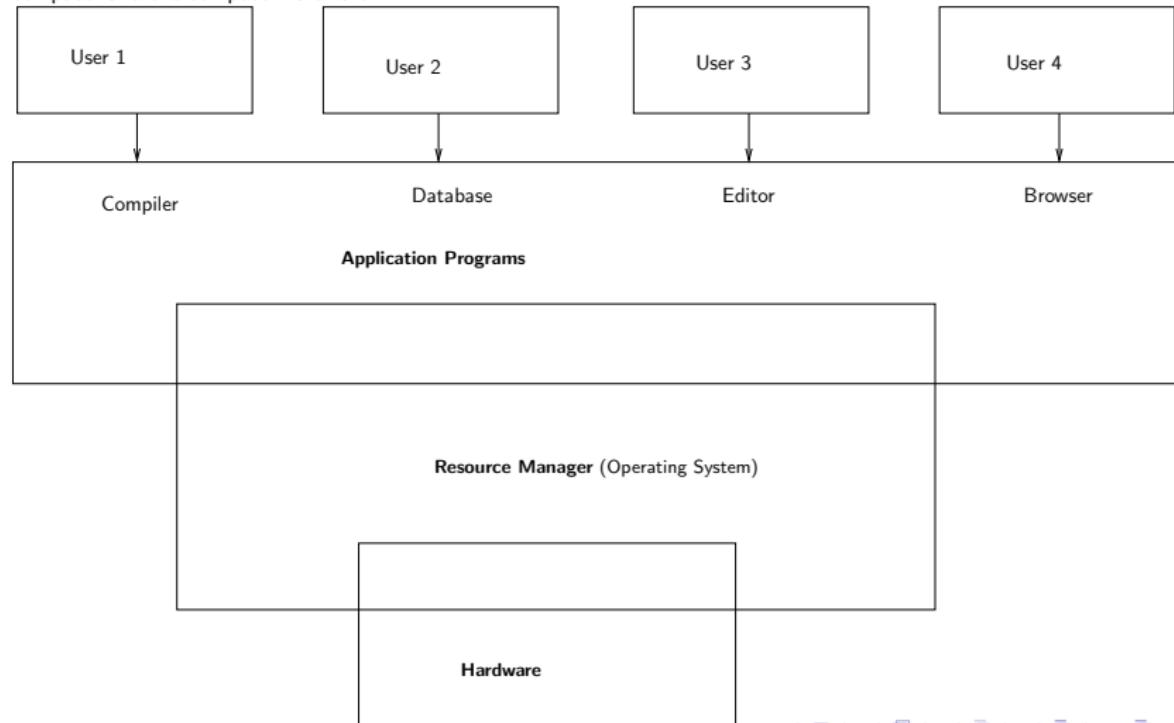
Recommended books

Recommended books for this part of the course:

- Operating Systems Concepts Silberschatz *et al.*
- Linux Kernel Development. Robert Love.
- Linux Programming Interface, Michael Kerrisk
- The C Programming Language (2nd Edition) Kernighan and Ritchie

What is an Operating System?

A program that acts as an intermediary between a user of a computer and the computer hardware



Main functions of an Operating Systems:

OS as a resource allocator:

Manages all resources (eg hardware) and decides between conflicting requests for efficient and fair resource use (e.g. accessing disk or other devices)

OS as a control system:

Controls execution of programs to prevent errors and improper use of the computer (e.g. protects one user process from crashing another)

Examples of Operating Systems

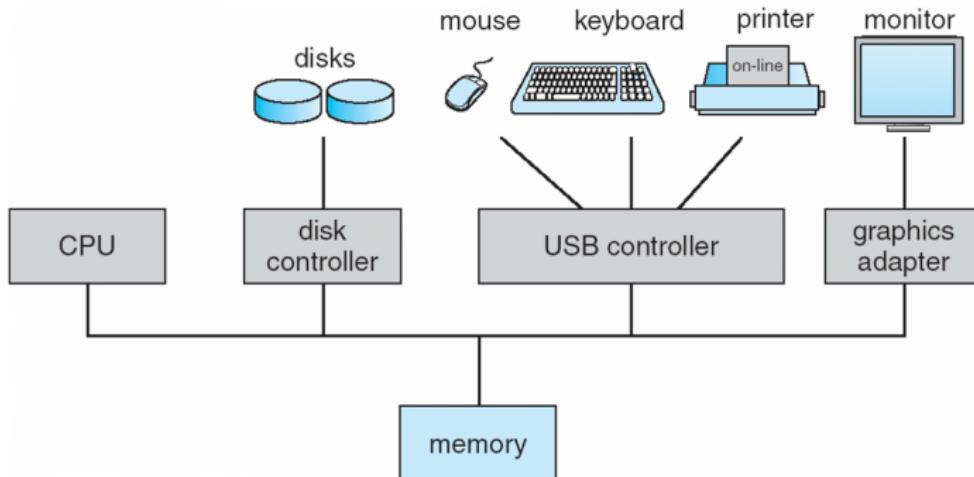
- Windows: current Windows OS based on Windows NT
- Mac OS: based on BSD Unix
- Linux: Based on Unix. Used also for Android.
- Newer operating systems for realtime applications and small resource usage (eg for Internet of Things)

Operating System Topics

Bootstrapping of the OS

- Small bootstrap program is loaded at power-up or reboot
 - Typically stored in ROM or EPROM, generally known as firmware (e.g. BIOS)
- Initializes all aspects of the system (e.g. detects connected devices, checks memory for errors, etc.)
- Loads operating system kernel and starts its execution

Computer System Organisation



- One or more CPUs, device controllers connect through common bus providing access to shared memory
- CPU(s) and devices compete for memory cycles (*i.e.* to read and write memory addresses)

Computer System Operation

- I/O devices and the CPU can execute concurrently
- Each device controller (e.g. controller chip) is in charge of a particular device type
- Each device controller has a local buffer (*i.e.* memory store for general data and/or control registers)
- CPU moves data from/to main memory to/from controller buffers (e.g. write this data to the screen, read coordinates from the mouse, *etc.*)
- I/O is from the device to local buffer of controller
- Device controller informs CPU that it has finished its operation by causing an *interrupt*

Interrupts

- Interrupt transfers control to the interrupt service routine generally, through the interrupt vector, which contains the addresses of all the service routines
- Interrupt architecture must save the address of the interrupted instruction so original processing may be resumed
- Incoming interrupts are disabled while another interrupt is being processed to prevent a lost interrupt
- A trap is a software-generated interrupt caused either by an error or a user request

Storage Structure

- Main memory - only large storage media that the CPU can access directly
- Secondary storage - provides large non-volatile storage capacity
- Important example: magnetic disks - rigid metal or glass platters covered with magnetic recording material
 - Disk surface is logically divided into tracks, which are subdivided into sectors
 - The disk controller determines the logical interaction between the device and the computer
- Today also often flash memory
However, same logical division into tracks and sectors still used

OS Services

How do Users and Processes interact with the Operating System?

- **Users** interact indirectly through a collection of system programs that make up the operating system interface. The interface could be:
 - A GUI, with icons and windows, etc.
 - A command-line interface for running processes and scripts, browsing files in directories, etc.
 - Or, back in the olden days, a non-interactive batch system that takes a collection of jobs, which it proceeds to churn through (e.g. payroll calculations, market predictions, etc.)
- **Processes** interact by making *system calls* into the operating system proper (*i.e.* the *kernel*).
 - Though we will see that, for stability, such calls are not direct calls to kernel functions.

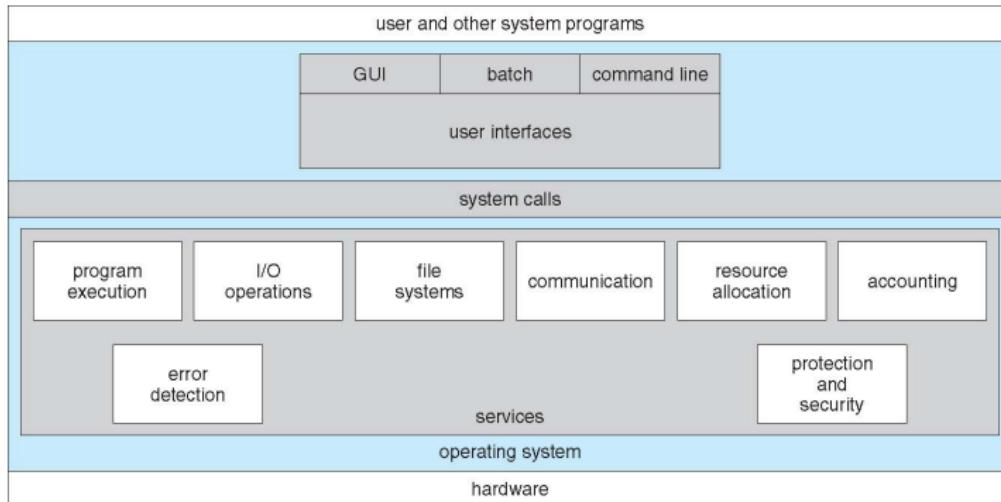
Services for Processes

- Typically, operating systems will offer the following services to processes:
 - **Program execution:** The system must be able to load a program into memory and to run that program, end execution, either normally or abnormally (indicating error)
 - **I/O operations:** A running program may require I/O, which may involve a file or an I/O device
 - **File-system manipulation:** Programs need to read and write files and directories, create and delete them, search them, list file information, permission management.
 - **Interprocess Communication (IPC):** Allowing processes to share data through message passing or shared memory

Services for the OS Itself

- Typically, operating systems will offer the following internal services:
 - **Error handling:** what if our process attempts a divide by zero or tries to access a protected region of memory, or if a device fails?
 - **Resource allocation:** Processes may compete for resources such as the CPU, memory, and I/O devices.
 - **Accounting:** e.g. how much disk space is this or that user using? how much network bandwidth are we using?
 - **Protection and Security:** The owners of information stored in a multi-user or networked computer system may want to control use of that information, and concurrent processes should not interfere with each other

OS Structure with Services



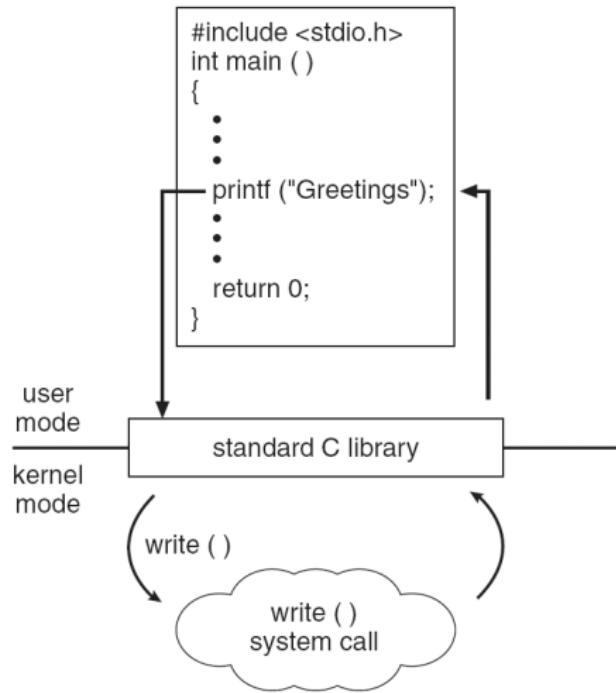
System Calls

- Programming interface to the services provided by the OS (e.g. open file, read file, etc.)
- Typically written in a high-level language (C or C++)
- Mostly accessed by programs via a high-level Application Program Interface (API) rather than direct system call.
- Three most common APIs are Win32 API for Windows, POSIX API for UNIX-based systems (including virtually all versions of UNIX, Linux, and Mac OS X)
- So why use APIs in user processes rather than system calls directly?
 - Since system calls result in execution of privileged kernel code, and since it would be crazy to let the user process switch the CPU to privileged mode, we must make use of the low-level hardware trap instruction, which is cumbersome for user-land programmers.
 - The user process runs the trap instruction, which will switch CPU to privileged mode and jump to a kernel pre-defined address of a generic system call function, hence the transition is controlled by the kernel.
 - Also, APIs can allow for backward compatibility if system calls change with the release of a OSS

System calls provided by Windows and Linux

	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() shmget() mmap()
Protection	SetFileSecurity() InitializeSecurityDescriptor() SetSecurityDescriptorGroup()	chmod() umask() chown()

An example of a System Call



System calls for file operations

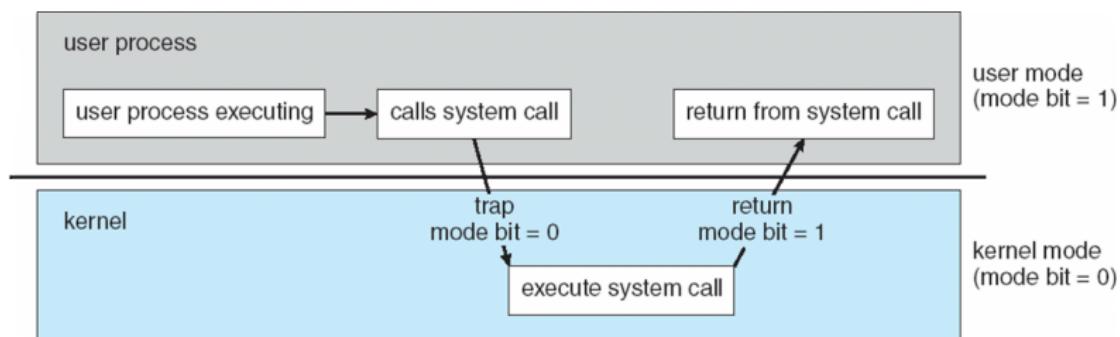
Have the following operations for files:

- open** Register the file with the operating system. Must be called before any operation on the file. Returns an integer called the file descriptor - an index into the list of open files maintained by the OS.
- read** Read data from the file. Returns number of bytes read or 0 for end of file.
- write** Write data to a file. Returns number of bytes written.
- close** De-registers the file with the operating system. No further operations on the file are possible.

These system calls return a negative number on error. Can use `perror`-function to display error.

Trapping to the Kernel

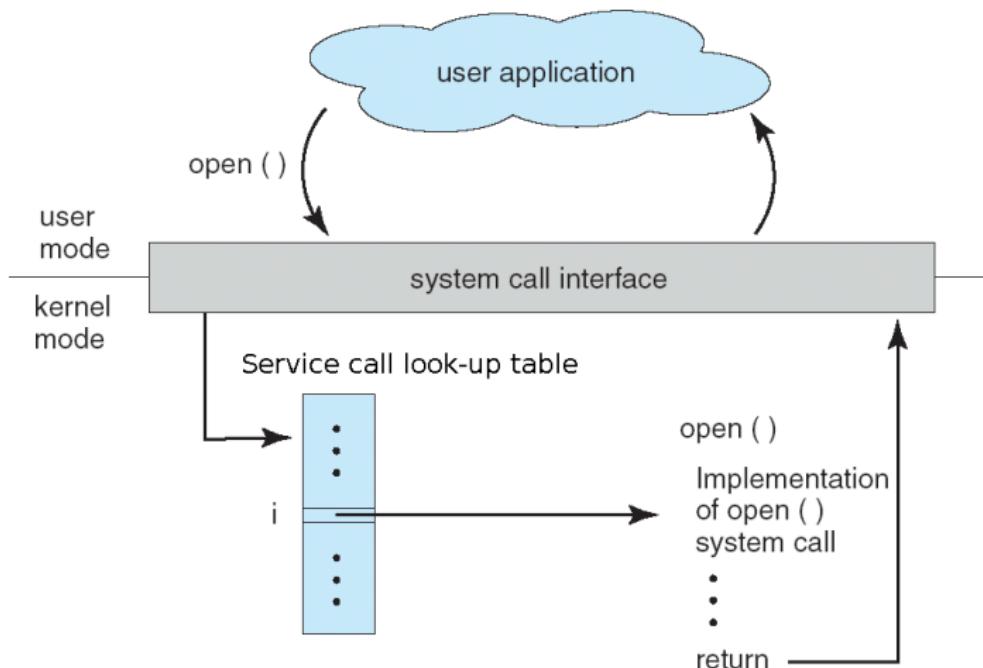
- The user process calls the system call wrapper function from the standard C library
- The wrapper function issues a low-level *trap* instruction (in assembly) to switch from user mode to kernel mode



Trapping to the Kernel

- To get around the problem that no call can directly be made from user space to a specific function in kernel space:
 - Before issuing the trap instruction, an index is stored in a well known location (e.g. CPU register, the stack, etc.).
 - Then, once switched into kernel space, the index is used to look up the desired kernel service function, which is then called.
- Some function calls may take arguments, which may be passed as pointers to structures via registers.

Trapping to the Kernel



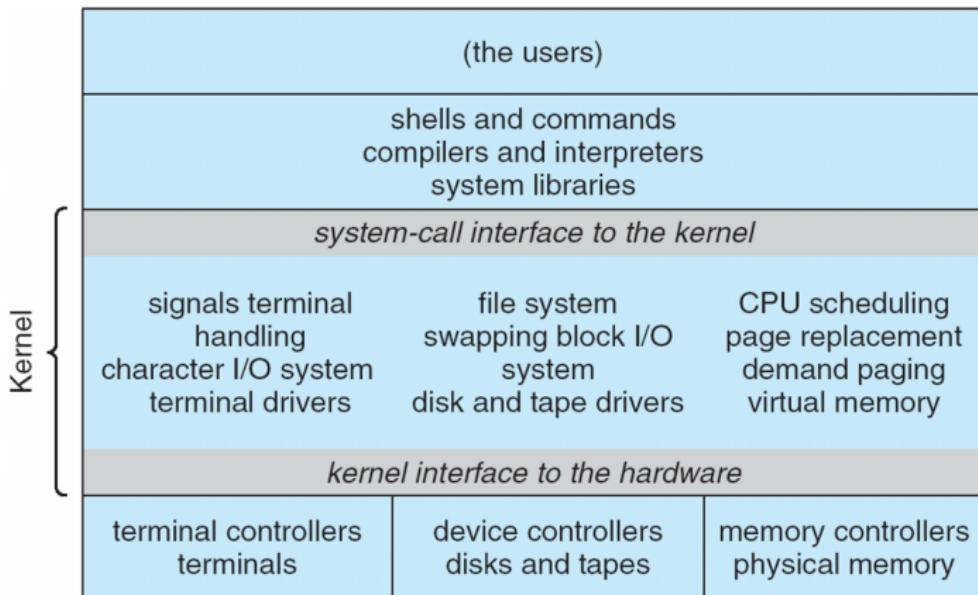
OS Architecture

Traditional UNIX

UNIX - one big kernel

- Consists of everything below the system-call interface and above the physical hardware
- Provides the file system, CPU scheduling, memory management, and other operating-system functions; a large number of functions for one level
- Limited to hardware support compiled into the kernel.

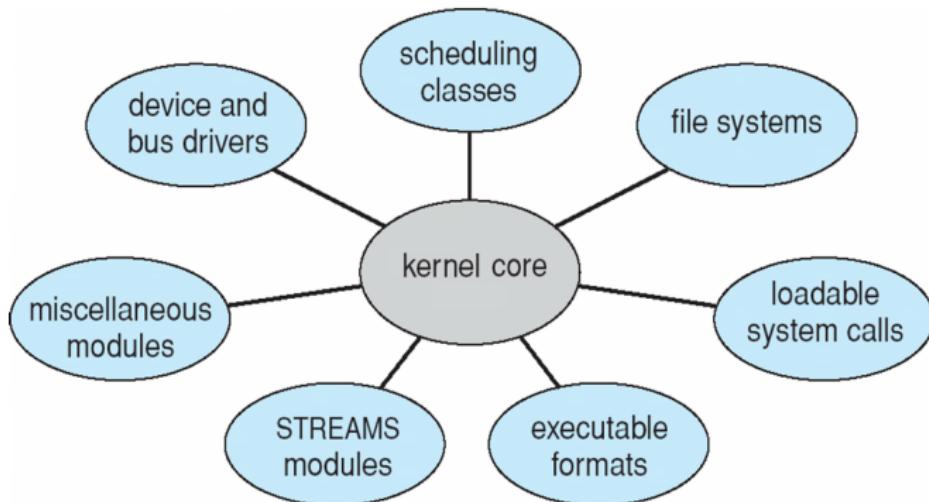
Traditional UNIX



Modular Kernel

- Most modern operating systems implement kernel modules
 - Uses object-oriented-like approach
 - Each core component is separate
 - Each talks to the others over known interfaces
 - Each is loadable as needed within the kernel, so you could download a new device driver for your OS and load it at run-time, or perhaps when a device is plugged in
- Overall, similar to layered architecture but with more flexibility, since all require drivers or kernel functionality need not be compiled into the kernel binary.
- Note that the separation of the modules is still only logical, since all kernel code (including dynamically loaded modules) runs in the same privileged address space (a design now referred to as monolithic), so I could write a module that wipes out the operating system no problem.
 - This leads to the benefits of micro-kernel architecture, which we will look at soon

Modular Kernel



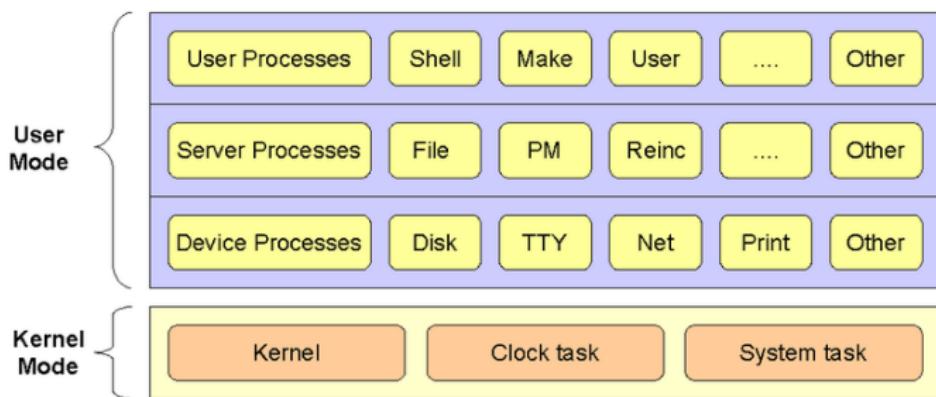
Microkernel

- Moves as much as possible from the kernel into less privileged “user” space (e.g. file system, device drivers, etc.)
- Communication takes place between user modules using message passing
 - The device driver for, say, a hard disk device can run all logic in user space (e.g. decided when to switch on and off the motor, queuing which sectors to read next, etc.)
 - But when it needs to talk directly to hardware using privileged I/O port instructions, it must pass a message requesting such to the kernel.

Microkernel

- Benefits:
 - Easier to develop microkernel extensions
 - Easier to port the operating system to new architectures
 - More reliable (less code is running in kernel mode) - if a device driver fails, it can be re-loaded
 - More secure, since kernel is less-complex and therefore less likely to have security holes.
 - The system can recover from a failed device driver, which would usually cause “a blue screen of death” in Windows or a “kernel panic” in linux.
- Drawbacks:
 - Performance overhead of user space to kernel space communication
 - The Minix OS and L3/L4 are examples of microkernel architecture

Microkernel: MINIX



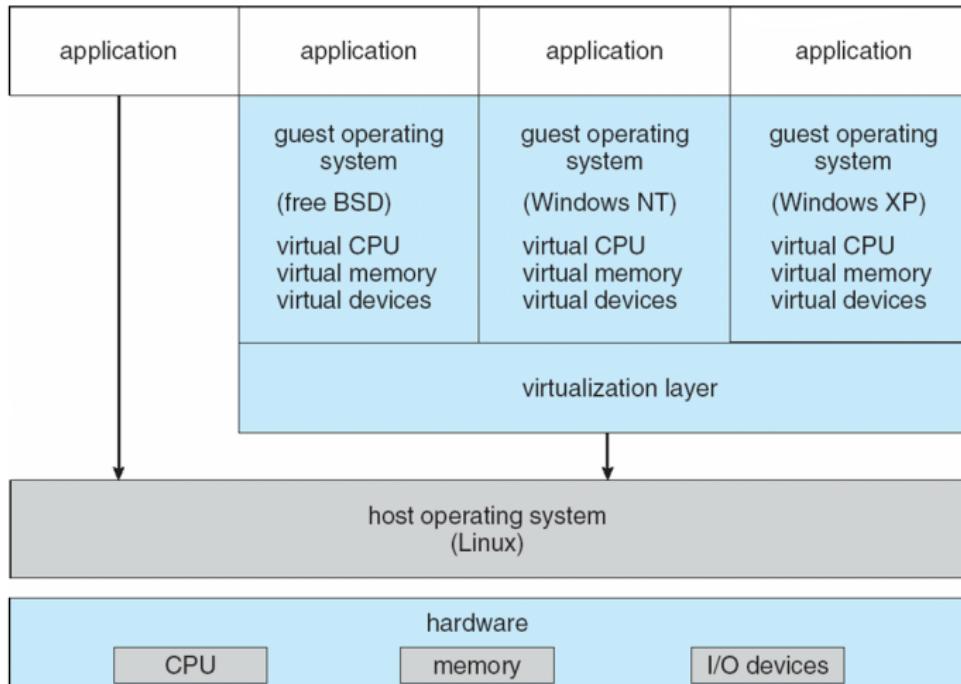
The MINIX 3 Microkernel Architecture

Virtual Machines

Virtual Machines

- A virtual machine allows to run one operating system (the guest) on another operating system (the host)
- A virtual machine provides an interface identical to the underlying bare hardware
- The operating system host creates the illusion that a process has its own processor and (virtual memory)
- Each guest is provided with a (virtual) copy of underlying computer, so it is possible to install, say, Windows 10 as a guest operating system on Linux.

VM Architecture



Virtual Machines: History and Benefits

- First appeared commercially in IBM mainframes in 1972
- Fundamentally, multiple execution environments (different operating systems) can share the same hardware
- Protected from one another, so no interference
 - Some sharing of files can be permitted, controlled
 - Communicate with one another other and with other physical systems via networking
- Useful for development, testing, especially OS development, where it is trivial to revert an accidentally destroyed OS back to a previous stable snapshot.

Virtual Machines: History and Benefits

- Consolidation of many low-resource use systems onto fewer busier systems
- “Open Virtualization Format” (OVF): standard format of virtual machines, allows a VM to run within many different virtual machine (host) platforms.
- Not to be confused with emulation, where guest instructions are run within a process that pretends to be the CPU (e.g. Bochs and QEMU). In virtualisation, the goal is to run guest instructions directly on the host CPU, meaning that the guest OS must run on the CPU architecture of the host.

Para-virtualisation

- Presents guest with system similar but not identical to hardware (e.g. Xen Hypervisor)
- Guest OS must be modified to run on paravirtualized 'hardware'
 - For example, the kernel is recompiled with all code that uses privileged instructions replaced by hooks into the virtualisation layer
 - After an OS has been successfully modified, para-virtualisation is very efficient, and is often used for providing low-cost rented Internet servers (e.g. Amazon EC2, Rackspace)

VMWare Architecture

- VMWare implements full virtualisation, such that guest operating systems do not require modification to run upon the virtualised machine.
- The virtual machine and guest operating system run as a user-mode process on the host operating system

VMWare Architecture

- As such, the virtual machine must get around some tricky problems to convince the guest operating system that it is running in privileged CPU mode when in fact it is not.
 - Consider a scenario where a process of the guest operating system raises a divide-by-zero error.
 - Without special intervention, this would cause the host operating system immediately to halt the virtual machine process rather than the just offending process of the guest OS.
 - So VMWare must look out for troublesome instructions and replace them at run-time with alternatives that achieve the same effect within user space, albeit with less efficiency
 - But since usually these instructions occur only occasionally, many instructions of the guest operating system can run unmodified on the host CPU.

Linux kernel programming

Structure of kernel

Simplified structure of kernel:

```
initialise data structures at boot time;  
while (true) {  
    while (timer not gone off) {  
        assign CPU to suitable process;  
        execute process;  
    }  
    select next suitable process;  
}
```

Kernel programming

Kernel has access to **all** resources

Kernel programs not subject to any constraints for memory access
or hardware access

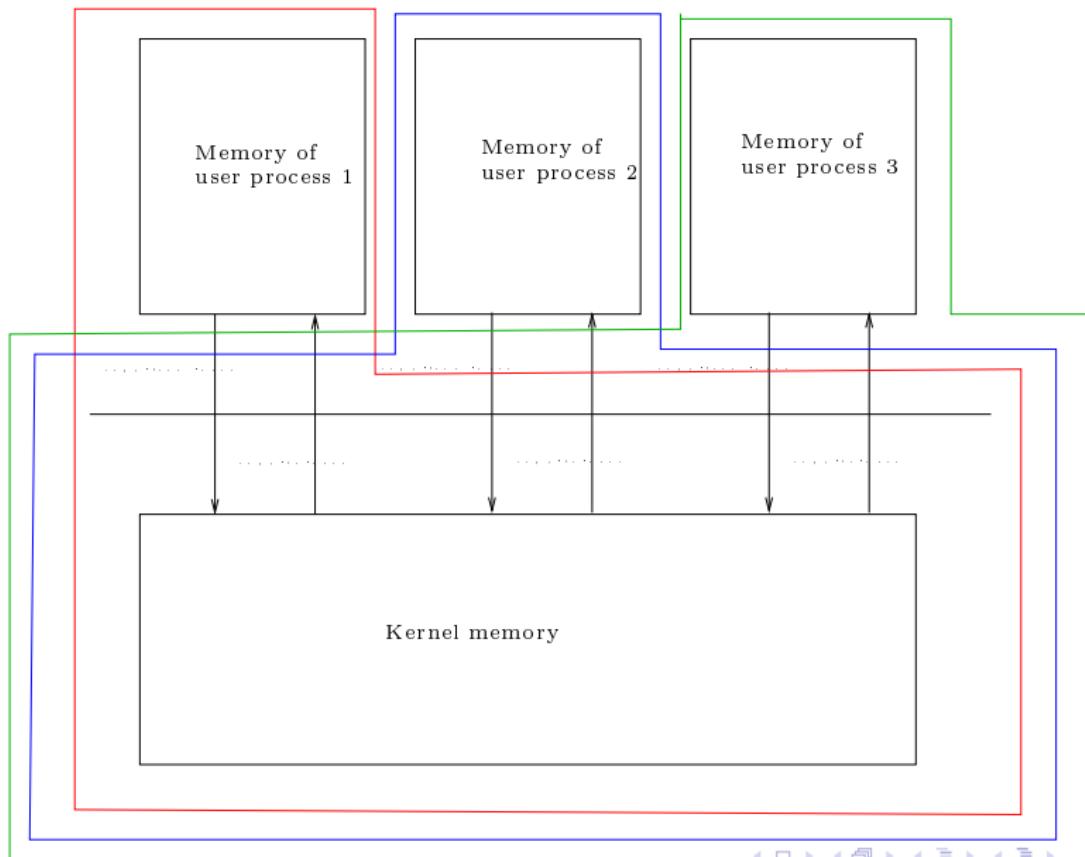
⇒ faulty kernel programs can cause system crash

Interaction between kernel and user programs

Kernel provides its functions only via special functions, called
system calls

standard C-library provides them

Have strict separation of kernel data and data for user programs
⇒ need explicit copying between user program and kernel
(`copy_to_user()`, `copy_from_user()`)



In addition, have **interrupts**:

kernel asks HW to perform certain action

HW sends interrupt to kernel which performs desired action

interrupts must be processed quickly

⇒ any code called from interrupts must not sleep

Linux kernel modes

Structure of kernel gives rise to two main modes for kernel code:

- **process context**: kernel code working for user programs by executing a system call
- **interrupt context**: kernel code handling an interrupt (eg by a device)

have access to user data only in process context

Any code running in process context may be pre-empted at any time by an interrupt

Interrupts have priority levels

Interrupt of lower priority are pre-empted by interrupts of higher priority

Kernel modules

can add code to running kernel

useful for providing device drivers which are required only if hardware present

`modprobe` inserts module into running kernel

`rmmmod` removes module from running kernel (if unused)

`lsmod` lists currently running modules

Concurrency issues in the kernel

Correct handling concurrency in the kernel important:

Manipulation of data structures which are shared between

- code running in process mode and code running in interrupt mode
- code running in interrupt mode

must happen only within critical regions

In multi-processor system even manipulation of data structures shared between code running in process context must happen only within critical sections

Achieving mutual exclusion

Two ways:

- **Semaphores/Mutex:** when entering critical section fails, current process is put to sleep until critical region is available
⇒ only usable if **all** critical regions are in process context
Functions: `DEFINE_MUTEX()`, `mutex_lock()`,
`mutex_unlock()`
- **Spinlocks:** processor tries repeatedly to enter critical section
Usable anywhere
Disadvantage: Have busy waiting
Functions: `spin_lock_init()`, `spin_lock()`,
`spin_unlock()`

Use of semaphores

Have two kinds of semaphores:

- Normal semaphores
- Read-Write semaphores: useful if some critical regions only read shared data structures, and this happens often

Programming data transfer between userspace and kernel

Linux maintains a directory called proc as interface between user space and kernel

Files in this directory do not exist on disk

Read-and write-operations on these files translated into kernel operations, together with data transfer between user space and kernel

Useful mechanism for information exchange between kernel and user space

A tour of the Linux kernel

Major parts of the kernel:

- Device drivers: in the subdirectory `drivers`, sorted according to category
- file systems: in the subdirectory `fs`
- scheduling and process management: in the subdirectory `kernel`
- memory management: in the subdirectory `mm`
- networking code: in the subdirectory `net`
- architecture specific low-level code (including assembly code): in the subdirectory `arch`
- include-files: in the subdirectory `include`

Summary

- Operating System as Resource manager, in particular for hardware
- OS has access to all resources
- Have API to interact with OS from programs (system calls)
- Have monolithic kernel (one big program) (eg Unix, Linux) or microkernels (only direct interaction with HW part of kernel)
- In Linux, can modify kernel via kernel modules - separate programs which are still part of the kernel
- Virtual machines make it possible to run one operating system on top of another one. Need to be careful with privileged OS instructions.

Linux Device Drivers

Device drivers

View from user space:

Have special file in `/dev` associated with it, together with five systems calls:

- `open`: make device available
- `read`: read from device
- `write`: write to device
- `ioctl`: Perform operations on device (optional)
- `close`: make device unavailable

Kernel side

Each file may have functions associated with it which are called when corresponding system calls are made

`linux/fs.h` lists all available operations on files

Device driver implements at least functions for `open`, `read`,
`write` and `close`.

Categorising devices

Kernel also keeps track of

- **Physical dependencies** between devices. Example:
devices connected to a USB-hub
- **Buses**: Channels between processor and one or more devices.
Can be either physical (eg pci, usb), or logical
- **Classes**: Sets of devices of the same type, eg keyboards, mice

Handling Interrupts in Device Drivers

Normal cycle of interrupt handling for devices:

- Device sends interrupt
- CPU selects appropriate interrupt handler
- Interrupt handler processes interrupt

Two important tasks to be done:

- Data to be transferred to/from device
- Waking up processes which wait for data transfer to be finished
- Interrupt handler clears interrupt bit of device
Necessary for next interrupt to arrive

Interrupt processing time must be as short as possible

Data transfer fast, rest of processing slow

⇒ Separate interrupt processing in two halves:

- **Top Half** is called directly by interrupt handler
 - Only transfers data between device and appropriate kernel buffer and schedules software interrupt to start Bottom half
- **Bottom half** still runs in interrupt context and does the rest of the processing (eg working through the protocol stack, and waking up processes)

Memory Management

Memory Management

Management of a **limited resource**:

(Memory hunger of applications increases with capacity!)

⇒ **Sophisticated algorithms needed**, together with support from HW and from compiler and loader.

Key point: program's view memory is set of memory cells starting at addresss 0x0 and finishing at some value (**logical address**)

Hardware: have set of memory cells starting at address 0x0 and finishing at some value (**physical address**)

Want to be able to store memory of several programs in main memory at the same time

need suitable mapping from logical addresses to physical addresses:

- at **compile time**: absolute references are generated (eg MS-DOS .com-files)
- at **load time**: can be done by **special program**
- at **execution time**: needs **HW support**

Address mapping can be taken one step further:

dynamic linking: use only **one copy of system library**

⇒ OS has to help: same code accessible to more than one process

Swapping

If memory demand is too high, memory of some processes is transferred to disk

Usually combined with scheduling: low priority processes are swapped out

Problems:

- Big transfer time
- What to do with pending I/O?

First point reason why swapping is not principal memory management technique

Fragmentation

Swapping raises two problems:

- over time, many **small holes** appear in memory (**external fragmentation**)
- programs only a little smaller than hole \Rightarrow **leftover too small to qualify as hole** (**internal fragmentation**)

Strategies for choosing holes:

- **First-fit**: Start from beginning and use first available hole
- **Rotating first fit**: start after last assigned part of memory
- **Best fit**: find smallest usable space
- **Buddy system**: Free holes are administered according to tree structure; smallest possible chunk used

Paging

Alternative approach: Assign **memory of a fixed size (page)**
⇒ avoids **external fragmentation**

Translation of logical address to physical address **done via page table**

Hardware support mandatory for paging:

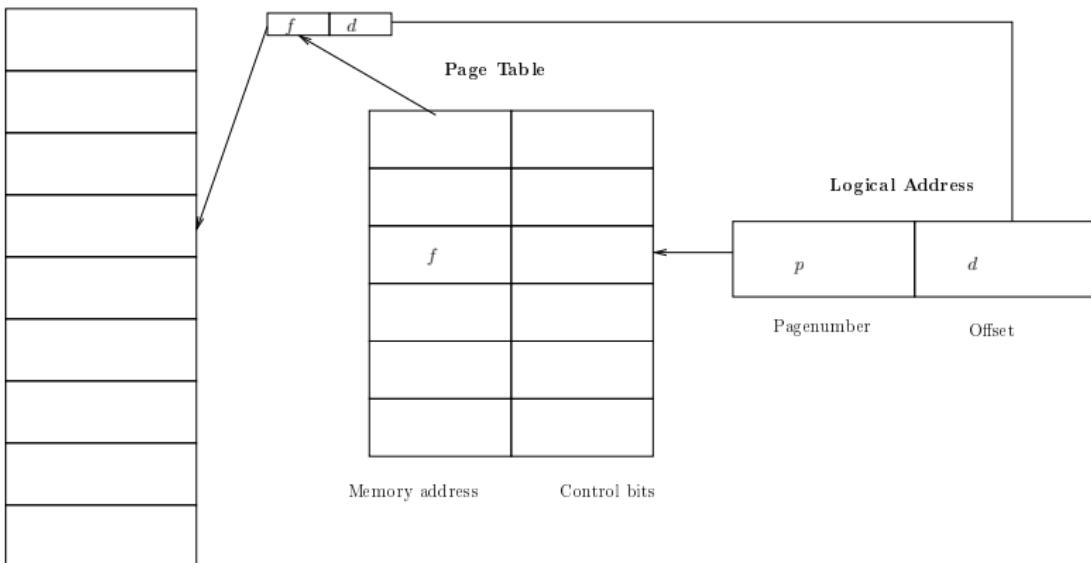
If page table **small**, use **fast registers**. Store large page tables in main memory, but **cache most recently used entries**

Instance of a general principle:

Whenever **large lookup** tables are required, **use cache (small but fast storage)** to store most recently used entries

Memory protection easily added to paging:
protection information **stored in page table**

Main Memory



Segmentation

Idea: Divide memory according to its usage by programs:

- Data: mutable, different for each instance
- Program Code: immutable, same for each instance
- Symbol Table: immutable, same for each instance, only necessary for debugging

Requires again HW support

can use same principle as for paging, but have to do overflow check

Paging motivated by ease of allocation, segmentation by use of memory

⇒ combination of both works well (eg 80386)

Main Memory

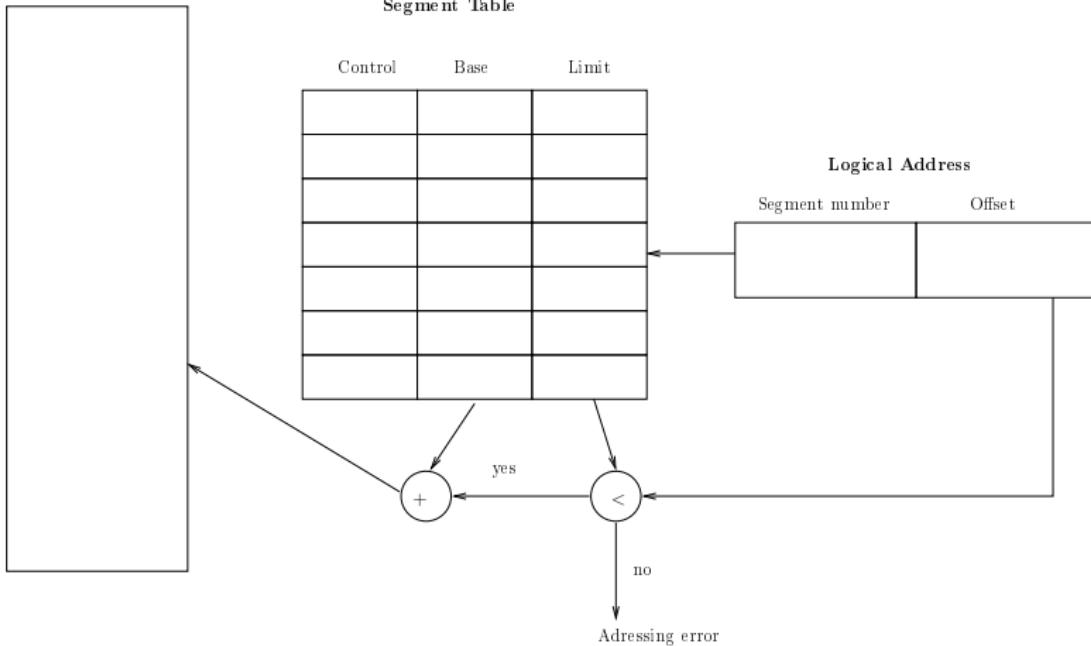
Segment Table

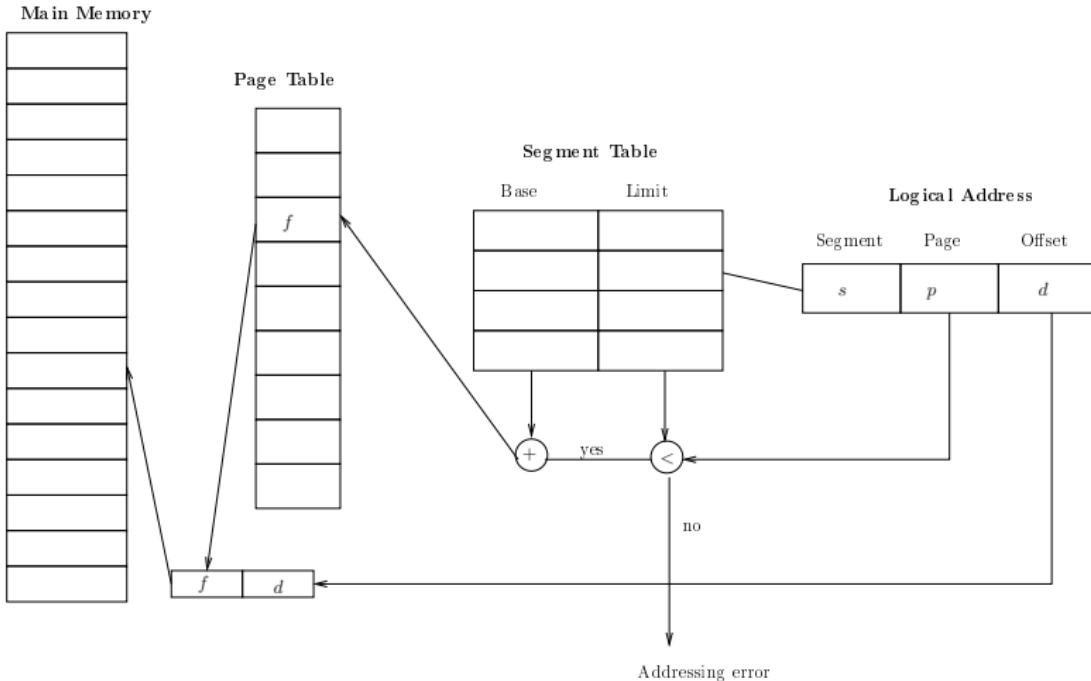
Control Base Limit

Control	Base	Limit

Logical Address

Segment number Offset





Virtual memory

Idea: complete separation of logical and physical memory

⇒ Program can have extremely large amount of virtual memory

works because most programs use only small fraction of memory intensively.

Efficient implementation tricky

Reason: Enormous difference between

- memory access speed (ca. 60ns)
- disk access speed (ca. 6ms)

Factor 100,000 !!

Demand Paging

Virtual memory implemented as demand paging: memory divided into **units of same length (pages)**, together with valid/invalid bit

Two strategic decisions to be made:

- Which process to “**swap out**” (move whole memory to disk and block process): swapper
- **which pages to move to disk** when additional page is required: pager

Minimisation of rate of page faults (page has to be fetched from memory) **crucial**

If we want 10% slowdown due to page fault, require fault rate
 $p < 10^{-6}!!$

Page replacement algorithms

1.) FIFO:

easy to implement, but does not take locality into account

Further problem: Increase in number of frames can cause increase in number of page faults (Belady's anomaly)

2.) Optimal algorithm:

select page which will be re-used at the latest time (or not at all)

⇒ not implementable, but good for comparisons

3.) Least-recently used:

use past as guide for future and replace page which has been unused for the longest time

Problem: Requires a lot of HW support

Possibilities:

-Stack in microcode

-Approximation using reference bit: HW sets bit to 1 when page is referenced.

Now use FIFO algorithm, but skip pages with reference bit 1, resetting it to 0

⇒ Second-chance algorithm

Thrashing

If process lacks frames it uses constantly, page-fault rate very high.
⇒ CPU-throughput decreases dramatically.
⇒ Disastrous effect on performance.

Two solutions:

1.) Working-set model (based on locality):

Define working set as set of pages used in the most recent Δ page references

keep only working set in main memory

⇒ Achieves high CPU-utilisation and prevents thrashing

Difficulty: Determine the working set!

Approximation: use reference bits; copy them each 10,000 references and define working set as pages with reference bit set.

2.) Page-Fault Frequency:

takes direct approach:

- give process additional frames if page frequency rate high
- remove frame from process if page fault rate low

Memory Management in the Linux Kernel

Have only four segments in total:

- Kernel Code
- Kernel Data
- User Code
- User Data

Paging used as described earlier

Have elaborate permission system for pages

Kernel memory and user memory

Have separate logical addresses for kernel and user memory

For 32-bit architectures (4 GB virtual memory):

- kernel space address are the upper 1 GB of address space ($\geq 0xC0000000$)
- user space addresses are 0x0 to 0xFFFFFFFF (lower 3 GB)

kernel memory always mapped but protected against access by user processes

Kernel memory and user memory

For 64-bit architectures:

- kernel space addresses are the upper half of address space ($\geq 0x8000\ 0000\ 0000\ 0000$)
- user space addresses are 0x0 to 0x7fff ffff ffff, starting from above.

kernel memory always mapped but protected against access by user processes

Page caches

Experience shows: have repeated cycles of allocation and freeing same kind of objects (eg inodes, dentries)
can have pool of pages used as cache for these objects (so-called slab cache)
cache maintained by application (eg file system)
`kmalloc` uses slab caches for commonly used sizes

Summary

Two topics:

Device drivers

implement open, read, write and close with common structure

Memory management

- Management of a limited resource ⇒ serious effort required
- Need to isolate memory for each process
- If memory demand is too high, need to swap out part of the memory of a process
- Looked at paging and segmentation to achieve this
- Requires hardware support

The Critical-Section Problem

Background

- Concurrent access to shared data may result in data inconsistency
- Maintaining data consistency requires mechanisms to ensure the orderly execution of cooperating processes

Classic Problem: Finite buffer shared by producer and consumer

- Producer produces a stream of items and puts them into buffer
- Consumer takes out stream of items

Have to deal with producers waiting for consumers when buffer is full, and consumers waiting for producers when buffer is empty.

Producer

```
while (true) {  
    /* produce an item and put in nextProduced */  
    while (count == BUFFER_SIZE); // wait if buffer full  
    buffer [in] = nextProduced; // store new item  
    in = (in + 1) % BUFFER_SIZE; // increment IN pointer.  
    count++; // Increment counter  
}
```

Consumer

```
while (true) {  
    while (count == 0) ; // do nothing  
    nextConsumed = buffer[out];  
    out = (out + 1) % BUFFER_SIZE;  
    count--;  
    /* consume the item in nextConsumed */  
}
```

Race Condition

There is something wrong with this code!

- `count++` could be compiled as a sequence of CPU instructions:
 - `register1 = count`
 - `register1 = register1 + 1`
 - `count = register1`
- `count--` could be compiled likewise:
 - `register2 = count`
 - `register2 = register2 - 1`
 - `count = register2`

Race Condition

- Consider that the producer and consumer execute the `count++` and `count--` around the same time, such that the CPU instructions are interleaved as follows (with `count = 5` initially):
 - Prod.: `register1 = count {register1 → 5}`
 - Prod.: `register1 = register1 + 1 {register1 → 6}`
 - Cons.: `register2 = count {register2 → 5}`
 - Cons.: `register2 = register2 - 1 {register2 → 4}`
 - Prod.: `count = register1 {count → 6}`
 - Cons.: `count = register2 {count → 4}`
- With an increment and a decrement, in whatever order, we would expect no change to the original value (5), but here we have ended up with 4?!?!

How to implement synchronization primitives?

Solution Criteria to Critical-Section Problem

- Solution to protect against concurrent modification of data in the *critical section* with the following criteria:
 - **Mutual Exclusion** - If process P_i is in its critical section (*i.e.* where shared variables could be altered inconsistently), then no other processes can be in this critical section.
 - **Progress** - no process outside of the critical section (*i.e.* in the *remainder section*) should block a process waiting to enter.
 - **Bounded Waiting** - A bound must exist on the number of times that other processes are allowed to enter the critical section after a process has made a request to enter its critical section and before that request is granted (*i.e.* it must be fair, so one poor process is not always waiting behind the others).
- Assume that each process executes at a nonzero speed.
- No assumption concerning relative speed of the N processes.

Peterson's Solution

- Two process solution.
- Assume that the CPU's register LOAD and STORE instructions are atomic (*not realistic* on modern CPUs, educational example).
- The two processes share two variables:
 - `int turn;`
 - `Boolean wants_in[2];`
- The variable `turn` indicates whose turn it is to enter the critical section.
- The `wants_in` array is used to indicate if a process is ready to enter the critical section. `wants_in[i] = true` implies that process P_i is ready.

A Naïve Algorithm for Process P_i

```
is_in[i] = FALSE; // I'm not in the section
do {
    while (is_in[j]); // Wait whilst j is in the critical section
    is_in[i] = TRUE; // I'm in the critical section now.
    [critical section]
    is_in[i] = FALSE; // I've finished in the critical section now.
    [remainder section]
} while (TRUE);
```

What's wrong with this?

Peterson's Algorithm for Process P_i

```
do {
    wants_in[i] = TRUE; // I want access...
    turn = j; // but, please, you go first
    while (wants_in[j] && turn == j); // if you are waiting and it is
                                    // your turn, I will wait.
    [critical section]
    wants_in[i] = FALSE; // I no longer want access.
    [remainder section]
} while (TRUE);
```

When both processes are interested, they achieve fairness through the `turn` variable, which causes their access to alternate.

Peterson's Algorithm for Process P_i

Interesting, but:

- Aside from the question of CPU instruction atomicity, how can we support more than two competing processes?
- Also, if we were being pedantic, we could argue that a process may be left to wait unnecessarily if a context switch occurs only after another process leaves the remainder section but before it politely offers the turn to our first waiting process.

Synchronisation Hardware

Synchronisation Hardware

- Many systems provide hardware support for critical section
- Uniprocessors - could disable interrupts
 - Currently running code would execute without preemption
 - Generally too inefficient on multiprocessor systems
 - Delay in one processor telling others to disable their interrupts
- Modern machines provide the special atomic hardware instructions (Atomic = non-interruptible) **TestAndSet** or **Swap** which achieve the same goal:
 - **TestAndSet**: Test memory address (*i.e.* read it) and set it in one instruction
 - **Swap**: Swap contents of two memory addresses in one instruction
- We can use these to implement simple locks to realise mutual exclusion

Solution to Critical-section Problem Using Locks

- The general pattern for using locks is:

```
do {
    [acquire lock]
    [critical section]
    [release lock]
    [remainder section]
} while (TRUE);
```

TestAndSet Instruction

- High-level definition of the atomic CPU instruction:

```
boolean TestAndSet (boolean *target) {  
    boolean original = *target; // Store the original value  
    *target = TRUE; // Set the variable to TRUE  
    return original; // Return the original value  
}
```

- In a nutshell: this single CPU instruction sets a variable to TRUE and returns the original value.
- This is useful because, if it returns FALSE, we know that only our thread has changed the value from FALSE to TRUE; if it returns TRUE, we know we haven't changed the value.

Solution using TestAndSet

- Shared boolean variable `lock`, initialized to false.
- Solution:

```
do {
    while (TestAndSet(&lock)) ; // wait until we successfully
                                // change lock from false to true
    [critical section]
    lock = FALSE; // Release lock
    [remainder section]
} while (TRUE);
```

- Note, though, that this achieves mutual exclusion but not *bounded waiting* - one process could potentially wait for ever due to the unpredictability of context switching.

Bounded-waiting Mutual Exclusion with TestAndSet()

- All data structures are initialised to FALSE.
- `wants_in[]` is an array of waiting flags, one for each process.
- `lock` is a boolean variable used to lock the critical section.

Bounded-waiting Mutual Exclusion with TestAndSet()

```
boolean wants_in[], key = FALSE, lock = FALSE; //all false to begin with

do {
    wants_in[i] = TRUE; // I (process i) am waiting to get in.
    key = TRUE; // Assume another has the key.
    while (wants_in[i] && key) { // Wait until I get the lock
        // (key will become false)
        key = TestAndSet(&lock); // Try to get the lock
    }
    wants_in[i] = FALSE; // I am no longer waiting: I'm in now

    [*** critical section ***]

    // Next 2 lines: get ID of next waiting process, if any.
    j = (i + 1) % NO_PROCESSES; // Set j to my right-hand process.
    while ((j != i) && !wants_in[j]) { j = (j + 1) % NO_PROCESSES };

    if (j == i) { // If no other process is waiting...
        lock = FALSE; // Release the lock
    } else { // else ....
        wants_in[j] = FALSE; // Allow j to slip in through the 'back door'
    }
    [*** remainder section ***]
} while (TRUE);
```

Inefficient Spinning

But There's a Bit of a Problem...

- Consider the simple mutual exclusion mechanism we just saw:

```
do {
    while (TestAndSet(&lock)) ; // wait until we successfully
                                // change lock from false to true
    [critical section]
    lock = FALSE; // Release lock
    [remainder section]
} while (TRUE);
```

- This guarantees mutual exclusion but with a high cost: that while loop spins constantly (using up CPU cycles) until it manages to enter the critical section.
- This is a huge waste of CPU cycles and is not acceptable, particularly for user processes where the critical section may be occupied for some time.

Sleep and Wakeup

- Rather than having a process spin around and around, checking if it can proceed into the critical section, suppose we implement some mechanism whereby it sends itself to sleep and then is awoken only when there is a chance it can proceed
- Functions such as `sleep()` and `wakeup()` are often available via a threading library or as kernel service calls.
- Let's explore this idea

Mutual Exclusion with sleep(): A first Attempt

- If constant spinning is a problem, suppose a process puts itself to sleep in the body of the spin.
- Then whoever won the competition (and therefore entered the critical section) will wake all processes before releasing the lock - so they can all try again.

```
do {  
    while (TestAndSet(&lock)) { // If we can't get the lock, sleep.  
        sleep();  
    }  
    [critical section]  
    wake_up(all); // Wake all sleeping threads.  
    lock = FALSE; // Release lock  
    [remainder section]  
} while (TRUE);
```

Towards Solving The Missing Wakeup Problem

- Somehow, we need to make sure that when a process decides that it will go to sleep (if it failed to get the lock) it actually goes to sleep without interruption, so the wake-up signal is not missed by the not-yet-sleeping process
- In other words, we need to make the check-if-need-to-sleep and go-to-sleep operations happen atomically with respect to other threads in the process.
- Perhaps we could do this by making use of another lock variable, say `deciding_to_sleep`.
- Since we now have two locks, for clarity, let's rename `lock` to `mutex_lock`.

A Possible Solution?

```
while (True) {
    // Spinning entry loop.
    while (True) {
        // Get this lock, so a wake signal cannot be raised before we actually sleep.
        while(TestAndSet(&deciding_to_sleep));

        // Now decide whether or not to sleep on the lock.
        if (TestAndSet(&mutex_lock)) {
            sleep();
        } else {
            // We are going in to critical section.
            deciding_to_sleep = False;
            break;
        }
        deciding_to_sleep = False; // Release the sleep mutex for next attempt.
    }
    [critical section]
    while(TestAndSet(&deciding_to_sleep)); // Don't issue 'wake' if a thread is
                                                // deciding whether or not to sleep.
    mutex_lock = False; // Open up the lock for the next guy.
    wake_up(all); // Wake all sleeping threads so they may compete for entry.
    deciding_to_sleep = False;
    [remainder section]
}
```

Have we, perhaps, overlooked something here?

Sleeping with the Lock

- We've encountered an ugly problem — in fact, there are several problems with the previous example that are all related.
 - As we said before, we need to decide to sleep and then sleep in an atomic action (with respect to other threads/processes)
 - But in the previous example, when a thread goes to sleep it keeps hold of the `deciding_to_sleep` lock which sooner or later will result in deadlock!
 - And if we release the `deciding_to_sleep` lock immediately before sleeping, then we have not solved the original problem.
- What to do, what to do...

Sleeping with the Lock

- The solution to this problem implemented in modern operating systems, such as Linux, is to release the lock *during* the kernel service call `sleep()`, such that it is released prior to the context switch, with a guarantee there will be no interruption.
- The lock is then reacquired upon wakeup, prior to the return from the `sleep()` call.
- Since we have seen how this can all get very complicated when we introduce the idea of sleeping (to avoid wasteful spinning), kernels often implement a sleeping lock, called a *semaphore*, which hides the gore from us.
- See <http://www.makelinux.net/ldd3/chp-5-sect-3>

Semaphores

Semaphores

- Synchronisation tool, proposed by E. W. Dijkstra (1965), that
 - Simplifies synchronisation for the programmer
 - Does not require (much) busy waiting: We don't busy-wait for the critical section, usually only to achieve atomicity to check if we need to sleep or not, etc.
 - Can guarantee *bounded waiting time* and *progress*.
- Consists of:
 - A semaphore type **S**, that records a list of waiting processes and an integer
 - Two standard atomic (\leftarrow very important) operations by which to modify **S: wait()** and **signal()**
 - Originally called **P()** and **V()** based on the equivalent Dutch words

Semaphores

- Works like this:
 - The semaphore is initialised with a count value of the maximum number of processes allowed in the critical section at the same time.
 - When a process calls `wait()`, if count is zero, it adds itself to the list of sleepers and blocks, else it decrements count and enters the critical section
 - When a process exits the critical section it calls `signal()`, which increments count and issues a wakeup call to the process at the head of the list, if there is such a process
 - It is the use of ordered wake-ups (e.g. FIFO) that makes semaphores support bounded (*i.e.* fair) waiting.

Semaphore as General Synchronisation Tool

- We can describe a particular semaphore as:
 - A Counting semaphore - integer value can range over an unrestricted domain (e.g. allow at most N threads to read a database, etc.)
 - A Binary semaphore - integer value can range only between 0 and 1
 - Also known as mutex locks, since ensure mutual exclusion.
 - Basically, it is a counting semaphore initialised to 1

Critical Section Solution with Semaphore

```
Semaphore mutex; // Initialized to 1
do {
    wait(mutex); // Unlike the pure spin-lock, we are blocking here.
    [critical section]
    signal(mutex);
    [remainder section]
} while (TRUE);
```

Semaphore Implementation: State and Wait

We can implement a semaphore within the kernel as follows (note that the functions must be atomic, which our kernel must ensure):

```
typedef struct {
    int count;
    process_list; // Hold a list of waiting processes/threads
} Semaphore;

void wait(Semaphore *S) {
    S->count--;
    if (S->count < 0) {
        add process to S->process_list;
        sleep();
    }
}
```

Semaphore Implementation: State and Wait

- Note that, here, we decrement the wait counter before blocking (unlike the previous description).
- This does not alter functionality but has the useful side-effect that the negative count value reveals how many processes are currently blocked on the semaphore.

Semaphore Implementation: Signal

```
void signal(Semaphore *S) {  
    S->count++;  
    if (S->count <= 0) { // If at least one waiting process, let him in.  
        remove next process, P, from S->process_list  
        wakeup(P);  
    }  
}
```

But we have to be Careful: Deadlock and Priority Inversion

- Deadlock: Two or more processes are waiting indefinitely for an event that can be caused by only one of the waiting processes
- Let **S** and **Q** be two semaphores initialized to 1

Process 1

```
wait(S);  
wait(Q);  
. . .  
signal(S);  
signal(Q);
```

Process 2

```
wait(Q);  
wait(S);  
. . .  
signal(Q);  
signal(S);
```

- Priority Inversion: Scheduling problem when lower-priority process holds a lock needed by higher-priority process: Good account of this from NASA programme:

http://research.microsoft.com/en-us/um/people/mbj/mars_pathfinder/authoritative_account.html

Semaphore Examples

Classical Problems of Synchronisation and Semaphore Solutions

- Bounded-Buffer Problem
- Readers and Writers Problem
- Boring as it may be to visit (and teach about) the same problems over and over again, a solid understanding of these seemingly-insignificant problems will directly help you to spot and solve many related real-life synchronisation issues.

Our Old Friend: The Bounded-Buffer Problem

- The solution set-up:
 - A buffer to hold N items, each can hold one item
 - Semaphore `mutex` initialized to the value 1
 - Semaphore `full_slots` initialized to the value 0
 - Semaphore `empty_slots` initialized to the value N .

Producer

```
while(True) {  
    wait(empty_slots); // Wait for, then claim, an empty slot.  
    wait(mutex);  
    // add item to buffer  
    signal(mutex);  
    signal(full_slots); // Signal that there is one more full slot.  
}
```

Consumer

```
while(True) {  
    wait(full_slots); // Wait for, then claim, a full slot  
    wait(mutex);  
    // consumer item from buffer  
    signal(mutex);  
    signal(empty_slots); // Signal that now there is one more empty slot.  
}
```

Readers-Writers Problem

- A data set is shared among a number of concurrent processes
 - Readers - only read the data set; they do not perform any updates
 - Writers - can both read and write
- Problem:
 - Allow multiple readers to read at the same time if there is no writer in there.
 - Only one writer can access the shared data at the same time

Readers-Writers Problem

- The solution set-up:
 - Semaphore `mutex` initialized to 1
 - Semaphore `wrt` initialized to 1
 - Integer `readcount` initialized to 0

Writer

```
while(True) {  
    wait(wrt); // Wait for write lock.  
    // perform writing  
    signal(wrt); // Release write lock.  
}
```

Reader

```
while(True) {  
    wait(mutex) // Wait for mutex to change read_count.  
    read_count++;  
    if (read_count == 1) // If we are first reader, lock out writers.  
        wait(wrt)  
    signal(mutex) // Release mutex so other readers can enter.  
  
    // perform reading  
  
    wait(mutex) // Decrement read_count as we leave.  
    read_count--;  
    if (read_count == 0)  
        signal(wrt) // If we are the last reader to  
    signal(mutex) // leave, release write lock  
}
```

Linux kernel representation of semaphores

- `wait(mutex)` is `mutex_lock` in the kernel
`signal(mutex)` is `mutex_unlock` in the kernel
This is a binary semaphore
- `down_read` and `down_write` in the kernel are read-write binary semaphores
- Have also counting semaphores in the linux kernel

Summary

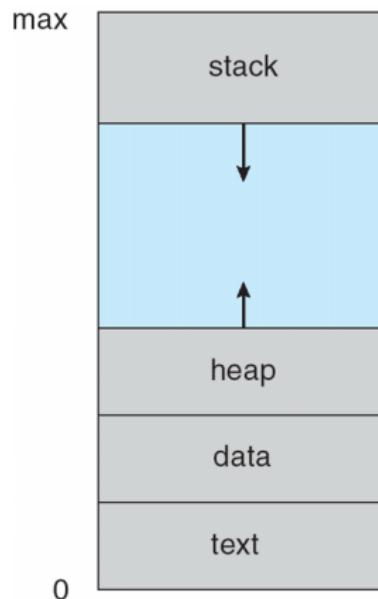
- Need to ensure that certain parts of the code (critical sections) are executed in a specified order
- Software solutions exist, but complexity very high
- Need atomic test-and-set operation, supported by hardware
- Can build synchronisation primitives (semaphores, locks) on top of test-and-set operation
- Linux kernel implements these primitives

Process Concept

Process Concept

- An operating system executes a variety of programs:
 - Batch system - jobs
 - Time-shared systems - user programs or tasks
- Process - a program in execution; process execution must progress in sequential fashion
- A process includes:
 - program (text) and program counter (PC)
 - stack
 - data section

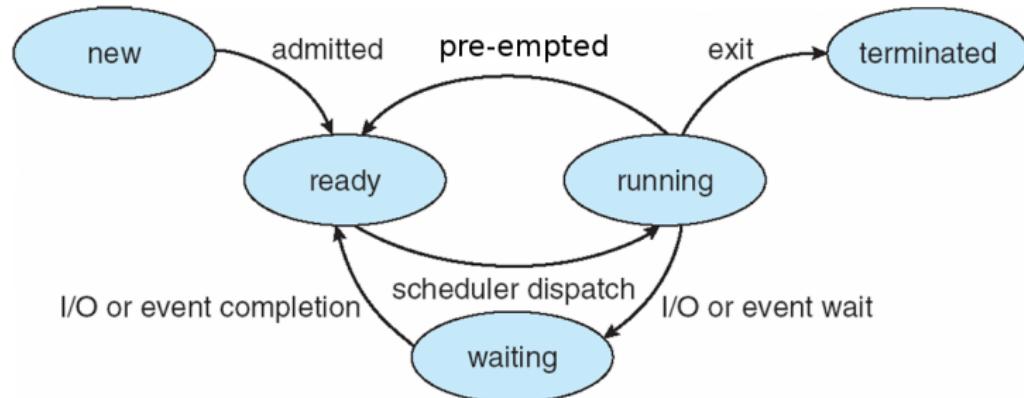
Process in Memory



Process States

- As a process executes, it changes state
 - new: The process is being created
 - running: Instructions are being executed
 - waiting: The process is waiting for some event to occur
 - ready: The process is waiting to be assigned to a processor
 - terminated: The process has finished execution

Process States



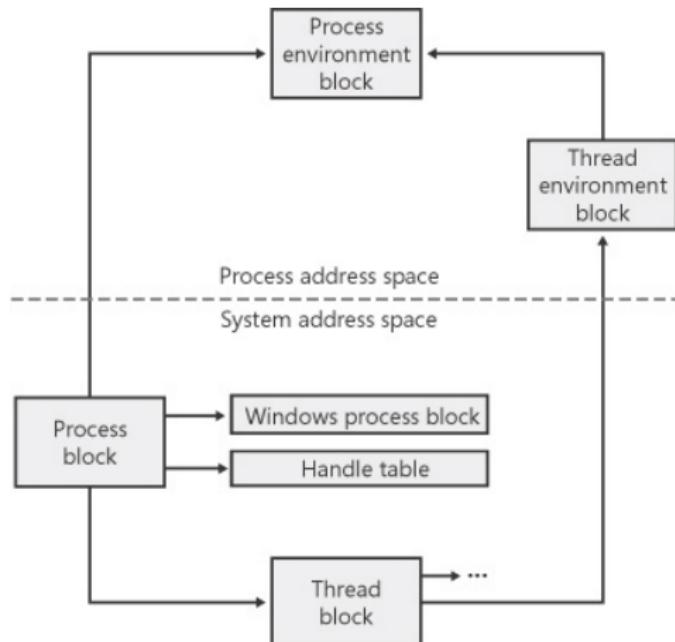
Process Control Block

- Information associated with each process, which is stored as various fields within a kernel data structure:
 - Process state
 - Program counter
 - CPU registers
 - CPU scheduling information
 - Memory-management information
 - Accounting information
 - I/O status information

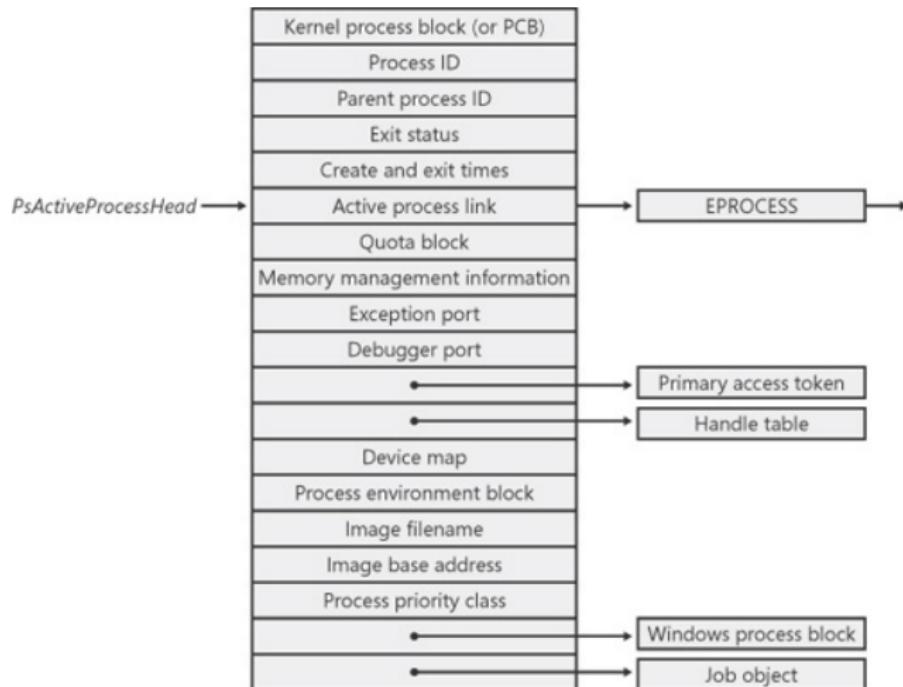
Process Control Block



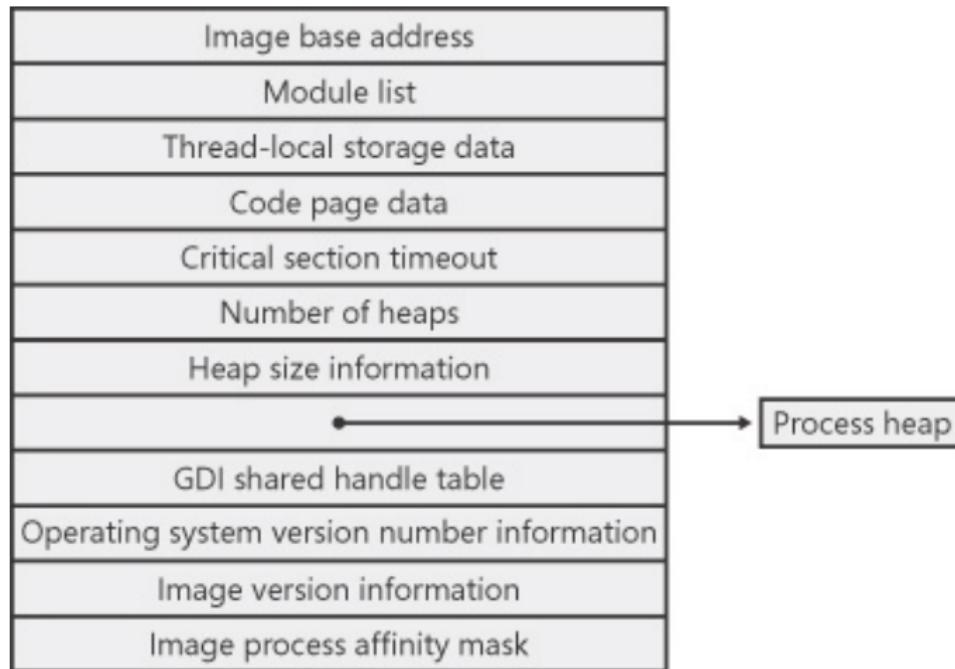
Process Control Blocks (Windows)



Process Block (EPROCESS)



Process Environment Block (PEB)



taskstruct (Linux)

```
struct task_struct {  
    volatile long state;  
    void *stack;  
    pid_t pid;  
    struct list_head tasks;  
    ...  
};
```

Can inspect data in /proc/pid

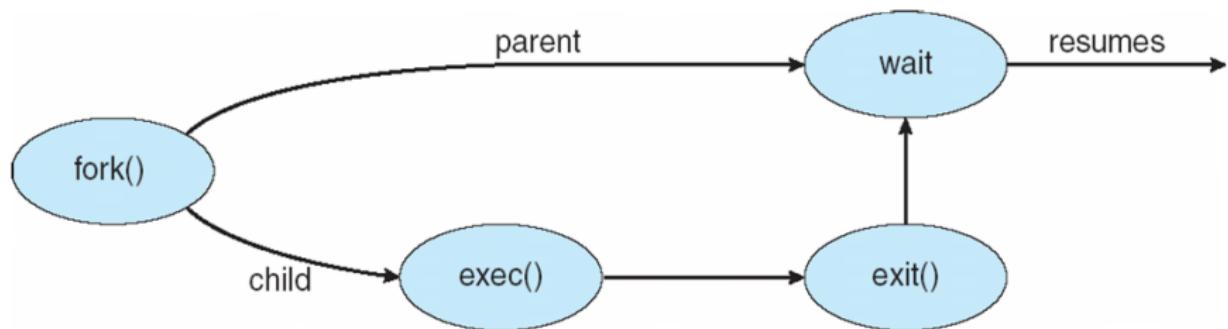
Process Creation

- Parent process create children processes, which, in turn create other processes, forming a tree of processes
- Generally, process identified and managed via a process identifier (pid)
- Resource sharing
 - Parent and children share all resources
 - Children share subset of parent's resources
 - Parent and child share no resources
- Execution
 - Parent and children execute concurrently
 - Parent waits until children terminate

Process Creation

- UNIX examples
 - `fork` system call creates new process
 - will look at fork soon, in one of our practical lectures.
 - `exec` system call used after a fork to replace the process' memory space with a new program

Process Creation



Process Termination

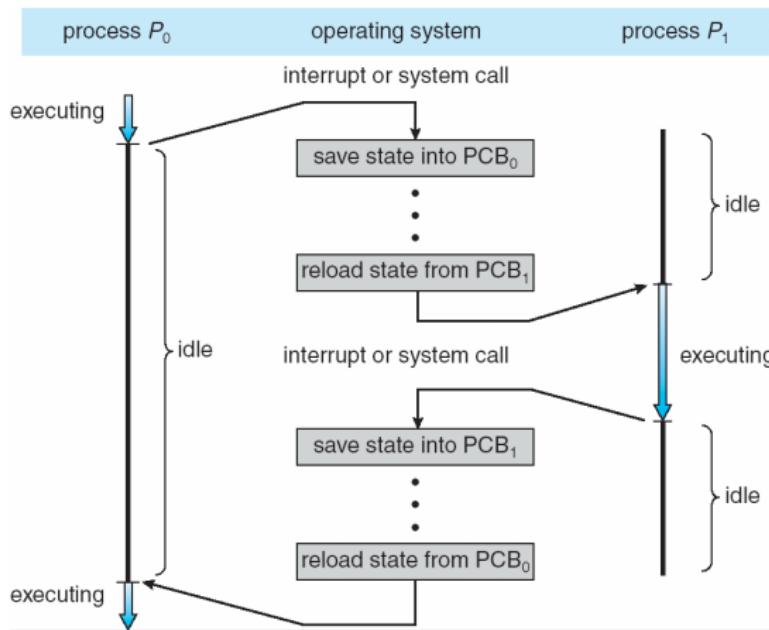
- Process executes last statement and asks the operating system to delete it (exit)
 - Output data from child to parent (via wait)
 - Process' resources are deallocated by operating system
- Parent may terminate execution of children processes (abort)
 - Child has exceeded allocated resources
 - Task assigned to child is no longer required
 - If parent is exiting:
 - Some operating systems do not allow child to continue if its parent terminates — all children terminated (*i.e.* cascading termination)

Concurrency Through Context Switching

Context Switch

- When CPU switches to another process, the system must save the state of the old process and load the saved state for the new process via a context switch
- Context of a process represented in the PCB
- Context-switch time is overhead; the system does no useful work while switching
- Time dependent on hardware support

Context Switch



Summary

- A process is a program in execution
- Processes are managed by the Operating Systems and can be in various stages (executing, waiting etc.)
- Have system calls for creating processes, as child of existing process
- CPU can be switched from one process to another by OS (context switch)
- Context switches are costly

Scheduling

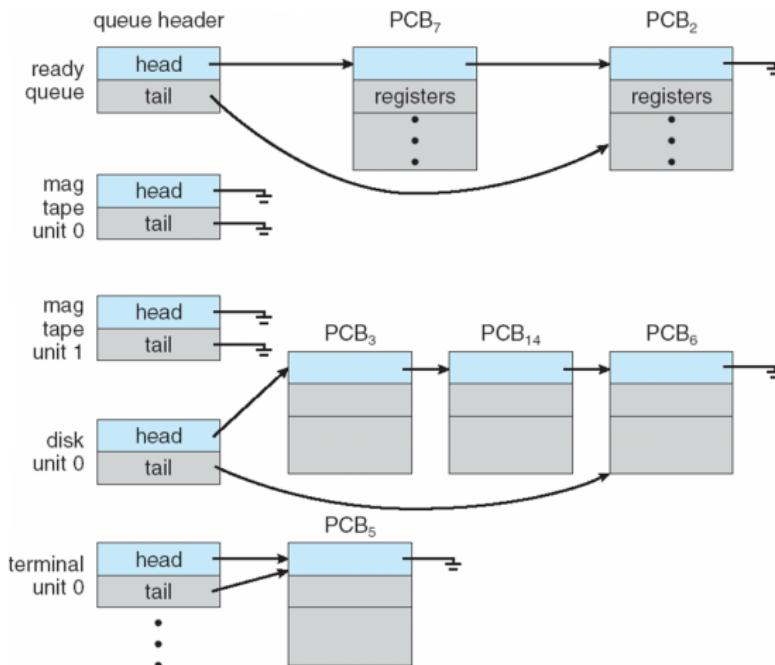
Scheduling Problem

- Have processes competing for resources (eg CPU, disk other devices)
- Important OS function: define a schedule to manage access of processes to these resources
- Typically done by having queues of processes waiting for a specific resource and selecting a process in the queue

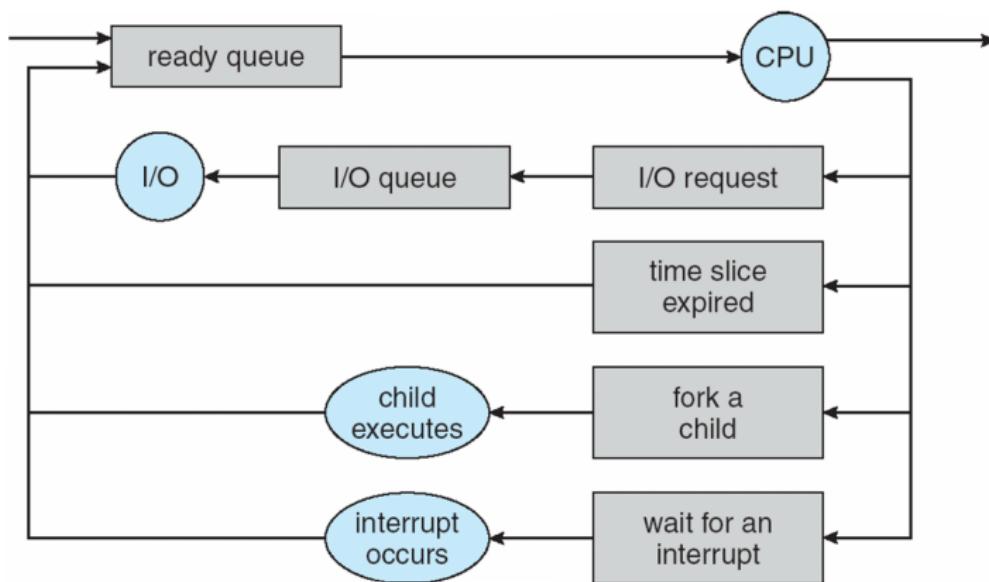
Process Scheduling Queues

- Job queue - set of all processes in the system
- Ready queue - set of all processes residing in main memory, ready and waiting to execute
- Device queues - set of processes waiting for an I/O device
- Processes migrate among the various queues

Process Scheduling Queues



Scheduling Workflow



CPU Scheduling

Problem: Which process ready to execute commands gets the CPU?

key function of the operating system

Prerequisites for successful scheduling:

1.) **CPU-I/O-Burst Cycle**

Experience shows: I/O occurs after **fixed amount of time** in $\geq 90\%$
⇒ appropriate time for re-scheduling

2.) **Preemptive Scheduling:** Processes can be forced to relinquish processor

Scheduling Criteria

Have various, often conflicting criteria to measure success of scheduling:

- CPU utilisation
- Throughput: Number of processes completed within a given time
- Turnaround time: Time it takes for each process to be executed
- Waiting time: Amount of time spent in the ready-queue
- Response time: time between submission of request and production of first response

Have two categories of processes:

- I/O-bound process - spends more time doing I/O than computations, many short CPU bursts
- CPU-bound process - spends more time doing computations; few very long CPU bursts

Scheduling algorithms

1.) First-Come, First-Served (**FCFS**)

Jobs are put in a queue, and served according to arrival time

Easy to implement **but** CPU-intensive processes can cause long waiting time.

FCFS with preemption is called **Round-Robin**
standard method in time sharing systems

Problem: get the **time quantum (time before preemption)** right.

- **too short**: too many context switches
- **too long**: Process can monopolise CPU

Shortest Job First

Next job is one with **shortest CPU-burst time** (shortest CPU-time before next I/O-operation)

Not implementable, but this is algorithm with the **smallest average waiting time**

⇒ Strategy against which to **measure other ones**

Approximation: Can we **predict the burst-time?**

Only hope is extrapolation from previous behaviour done by weighting recent times more than older ones.

$$\tau_{n+1} = \alpha t_n + (1 - \alpha)\tau_n$$

Priority Scheduling

Assumption: A priority is associated with each process
CPU is allocated to **process with highest priority**
Equal-priority processes scheduled according to FCFS

Two variations:

- **With preemption**: newly-arrived process with higher priority may gain processor immediately if process with lower priority is running
- **Without preemption**: newly arrived process always waits

Preemption good for ensuring quick response time for high-priority processes

Disadvantage: **Starvation** of low-priority processes **possible**

Solution: Increase priority of processes after a while (**Ageing**)

Multilevel Queue Scheduling

Applicable when processes can be **partitioned into groups** (eg interactive and batch processes):

Split ready-queue into several separate queues, with separate scheduling algorithm

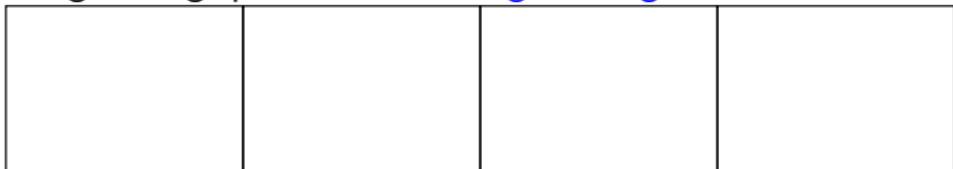
Scheduling between queues usually implemented as pre-emptive priority scheduling

Possible setup of queues:

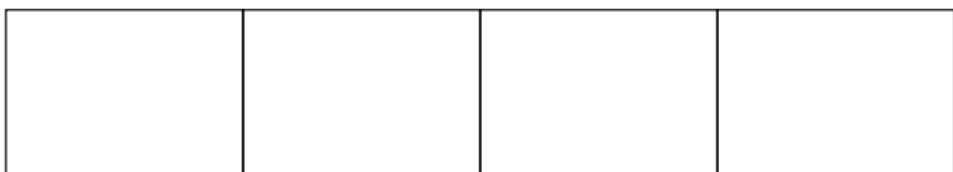
- System processes
- Interactive processes
- Interactive editing processes
- Batch processes

Other way of organising queues: according to length of CPU-burst

Burst time
1ms



Burst time
2ms



Burst time
4ms



Scheduling for Multiprocessor Systems

CPU scheduling more complex when multiple CPU's are available

Most common case: **Symmetric multiprocessing (SMP)**:

- all processors are identical, can be scheduled independently
- have separate ready-queue for each processor (Linux), or shared ready-queue

Processor Affinity

Process affinity for CPU on which it is currently running

- **Soft Affinity** current CPU only preferred when re-scheduled
- **Hard Affinity** Process may be bound to specific CPU
Advantage: caches remain valid, avoiding time-consuming cache invalidation and recovery

Load Balancing

Idea: use all CPU's equally (goes against processor affinity)

- **Push migration:** periodically check load, and push processes to less loaded CPU's
- **Pull migration:** idle CPU's pull processes from busy CPU's

Linux Implementation

Several schedulers may co-exist, assign fixed percentage of CPU-time to each scheduler

Important schedulers:

- Round-robin scheduler with priorities (the default scheduler)
- Real-time scheduler (process needs to be assigned explicitly to this one) (typically FIFO)

Round-Robin Scheduler with priorities

implemented in an interesting way:

maintain tree of processes ordered by runtime allocated so far

pick next process as one with least runtime allocated so far

insert new process in ready queue at appropriate place in tree

Priorities handled by giving weights to run-times.

Summary

- Have several algorithms for scheduling the CPU: FCFS, Round Robin, Priority Scheduling
- Also need to distribute processes amongs several CPUs or cores
- Linux: implements round-robin scheduler with priorities

File systems

File System

- Function: main permanent data storage
Speed bottleneck!
- Capacity not a problem nowadays: 2 TB disks even for PC.
But *backup becoming a problem.*
- Logical view (view of programmer): tree structure of files together with read/write operation and creation of directories
- Physical view: sequence of blocks, which can be read and written. OS has to map logical view to physical view, must impose tree structure and assign blocks for each file

Two main possibilities to realize filesystem:

- **Linked list**: Each block contains pointer to next
⇒ Problem: random access (`seek()`) costly: have to go through whole file until desired position.
- **Indexed allocation**: Store pointers in one location: so-called index block (similar to page table). To cope with vastly differing file sizes, may introduce **indirect index blocks**.

Index blocks are called `inodes` in Unix.

Inodes store additional information about the file (eg size, permissions)

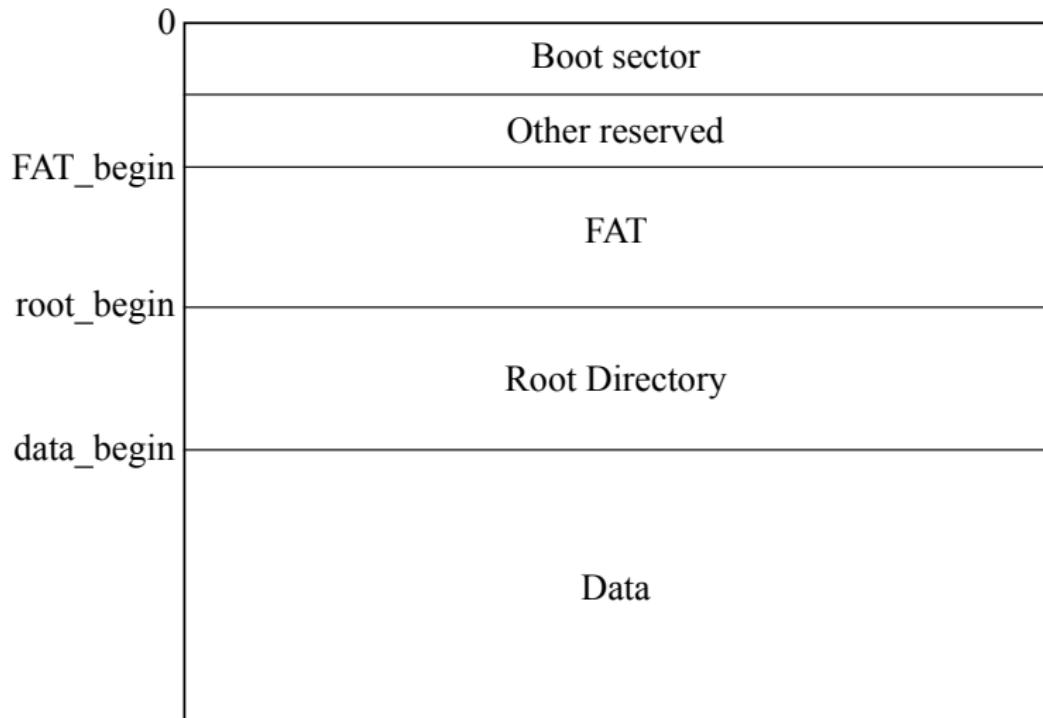
Worked Example

Worked example – based on
<http://www.tavi.co.uk/phobos/fat.html>

Example: FAT

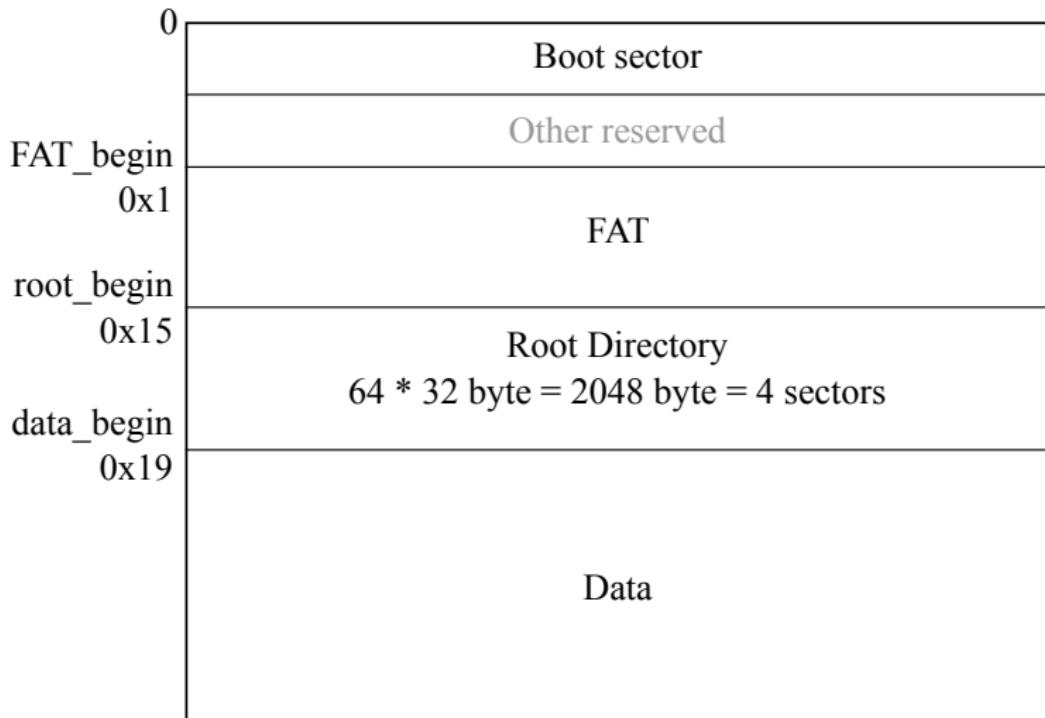
- F(ile) A(llocation) T(able) – dates back to 70s.
- Useful for explaining filesystem concepts, modern filesystems are more complicated
- Variants FAT12, FAT16, FAT32 define number of bits per FAT entry – we focus on FAT16
- Sector = disk unit (e.g. 512 byte), aka block
- Cluster = multiple sectors (factor 1, 2, 4, ..., 128)
(here: assume cluster = 1 sector)
- Uses linked list (“cluster chain”) to group clusters

Example: FAT16 Structure



Example: FAT16 Bootsector

Example: FAT16 with Offsets



Example: File Allocation Table

FAT_begin

FFF0	FFFF	0003	2	0004	3
0006	4	0000	5	FFFF	6
0009	8	FFFF	9	0000	A
000F	C	0010	D	0000	E
FFFF	10	0000	12	0000	13
				0000	14

2 → 3 → 4 → 6

7 → 8 → 9

C → F → D → 10

Example: File in Root Directory

Block 21 (0x0015)

	0	1	2	3	4	5	6	7	8	9	a	b	c	d	e	f
000	43	4f	38	38	33	2d	41	32	20	20	20	28	00	00	00	C0883-A2 (....
010	00	00	00	00	00	00	91	9e	65	39	00	00	00	00	00e9.....
020	46	4f	4f	42	41	52	20	20	54	58	54	21	00	a3	91	9e F00BAR TXT!....
030	65	39	65	39	00	00	91	9e	65	39	c6	10	1a	00	00	00 e9e9....e9.....
040	4e	45	54	57	4f	52	4b	20	56	52	53	20	00	b6	91	9e NETWORK VRS
050	65	39	65	39	00	00	91	9e	65	39	4e	0f	92	06	00	00 e9e9....e9N.....
060	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00

sector 0xF4E

length 0x692 = 1682 byte

filename & extension

Example: File in FAT

Block 16 (0x0010)

	0	1	2	3	4	5	6	7	8	9	a	b	c	d	e	f
000	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00
010	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00
020	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00
030	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00
040	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00
050	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00
060	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00
070	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00
080	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00
090	00	00	00	00	00	00	00	00	00	00	00	00	4f	0f	50	0f
0a0	51	0f	ff	ff	00	00	00	00	00	00	00	00	00	00	00	00
0b0	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00

4 block cluster
chain (2048 byte)

Example: FAT Limits

- Max. volume size: 2 GB ($2^{16} \cdot 32\text{ kB}$)
- Max. file size: 2 GB
- Max. number of files: 65,460 (32 kB clusters)
- Max. filename length: 8 + 3
- FAT32 / exFAT have higher limits
- Newer filesystems (NTFS, ext4) also overcome these limits, using other data structures (e.g. B-tree for dir structure, bitmap for allocation)

Further Aspects

Further aspects of filesystems

Caching

Disk blocks used for storing directories or recently used files cached in main memory

Blocks periodically written to disk

⇒ Big efficiency gain

Inconsistency arises when system crashes

Reason why computers must be shutdown properly

Journaling File Systems

To minimise data loss at system crashes, ideas from databases are used:

- Define **Transaction points**: Points where cache is written to disk
 - ⇒ Have consistent state
- Keep log-file for each write-operation
 - Log enough information to unravel any changes done after latest transaction point

Disk Access

Disk access contains three parts:

- **Seek**: head moves to appropriate track
- **Latency**: correct block is under head
- **Transfer**: data transfer

HDDs: Time necessary for seek and latency dwarfs transfer time
⇒ Distribution of data and scheduling algorithms have vital impact on performance for HDDs, less so for SSDs

Disk Scheduling Algorithms

Standard algorithms apply, adapted to the special situation:

1.) FCFS: easiest to implement, but: may require lots of head movements

2.) Shortest Seek Time First: Select job with minimal head movement

Problems:

- may cause starvation
- Tracks in the middle of disk preferred

Algorithm does not minimise number of head movements

3.) **SCAN-scheduling**: Head continuously scans the disk from en to end (lift strategy)

⇒ solves the fairness and starvation problem of SSTF

Improvement: **LOOK-scheduling**:

head only moved as far as last request (lift strategy).

Particular tasks may require different disk access algorithms

Example : Swap space management

Speed absolutely crucial ⇒ different treatment:

- Swap space stored on separate partition
- Indirect access methods not used

- Special algorithms used for access of blocks
Optimised for speed at the cost of space (eg increased internal fragmentation)

Linux Implementation

Interoperability with Windows and Mac requires support of different file systems (eg vfat)

⇒ Linux implements common interface for all filesystems

Common interface called **virtual file system**

virtual file system maintains

- inodes for files and directories
- caches, in particular for directories
- superblocks for file systems

All system calls (eg open, read, write and close) first go to virtual file system

If necessary, virtual file system selects appropriate operation from real file system

Disk Scheduler

Kernel makes it possible to have different schedulers for different file systems

Default scheduler (Completely Fair Queuing) based on lift strategy
have in addition separate queue for disk requests for each process
queues served in Round-Robin fashion

Have in addition No-op scheduler: implements FIFO

Suitable for SSD's where access time for all sectors is equal