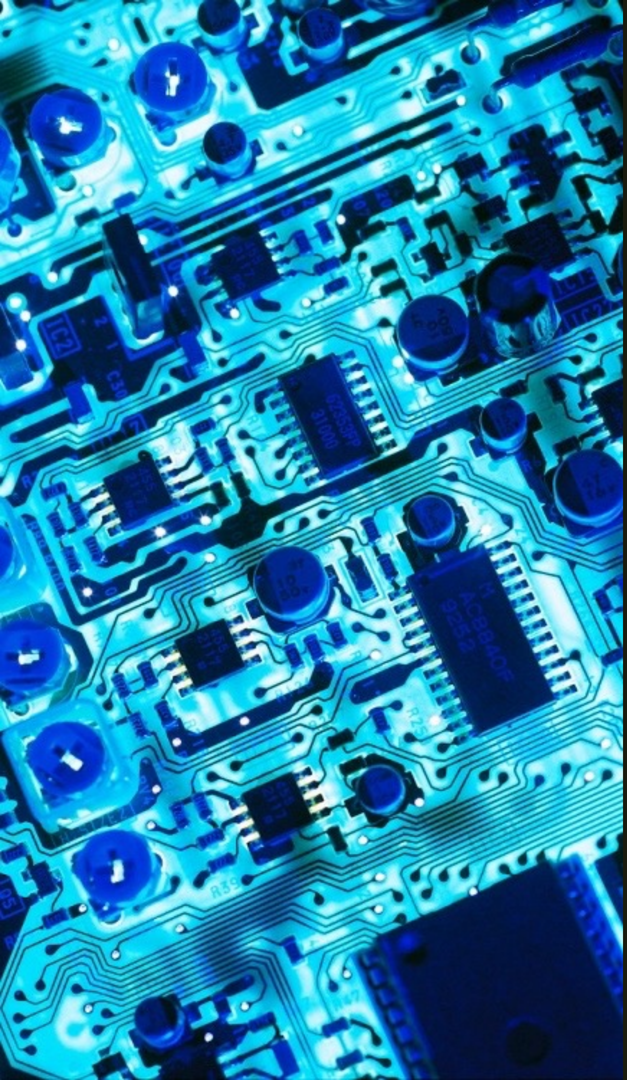
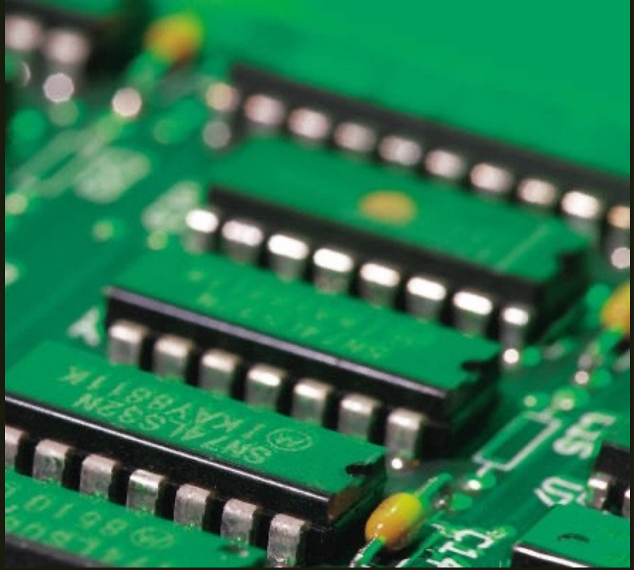
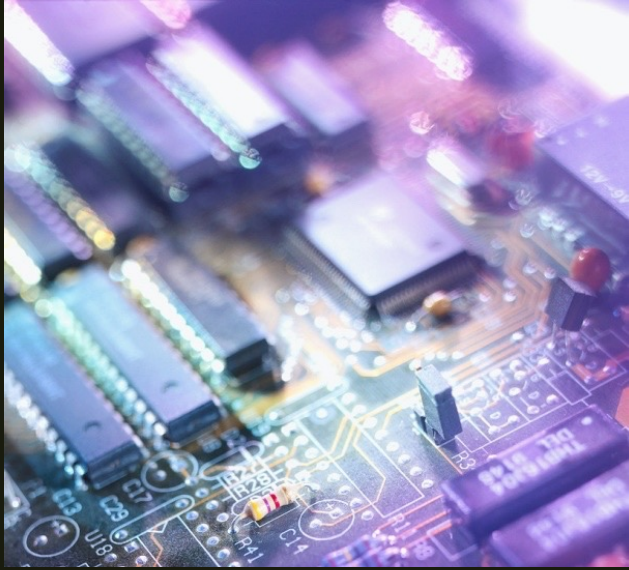






COMPUTING AND MEMORY ARCHITECTURES

Dr. Steve Petruzza

UtahStateUniversity

Reference material for part of this presentation :

An Introduction to Parallel Programming by Peter Pacheco

Some post-quiz review

- Good interested in the various topics
- C/C++ Syntax for pointers
- Memory management and allocation
- Arrays clarifications

C/C++ syntax for pointers

```
#include <iostream>

using namespace std;

int main () {
   int  var = 20;    // actual variable declaration.
   int  *ip;         // pointer variable

   ip = &var;        // store address of var in pointer variable

   cout << "Value of var variable: ";
   cout << var << endl;

   // print the address stored in ip pointer variable
   cout << "Address stored in ip variable: ";
   cout << ip << endl;

   // access the value at the address available in pointer
   cout << "Value of *ip variable: ";
   cout << *ip << endl;

   return 0;
}
```

```
$g++ -o main *.cpp

$main

Value of var variable: 20
Address stored in ip variable: 0x7fff1b872034
Value of *ip variable: 20
```

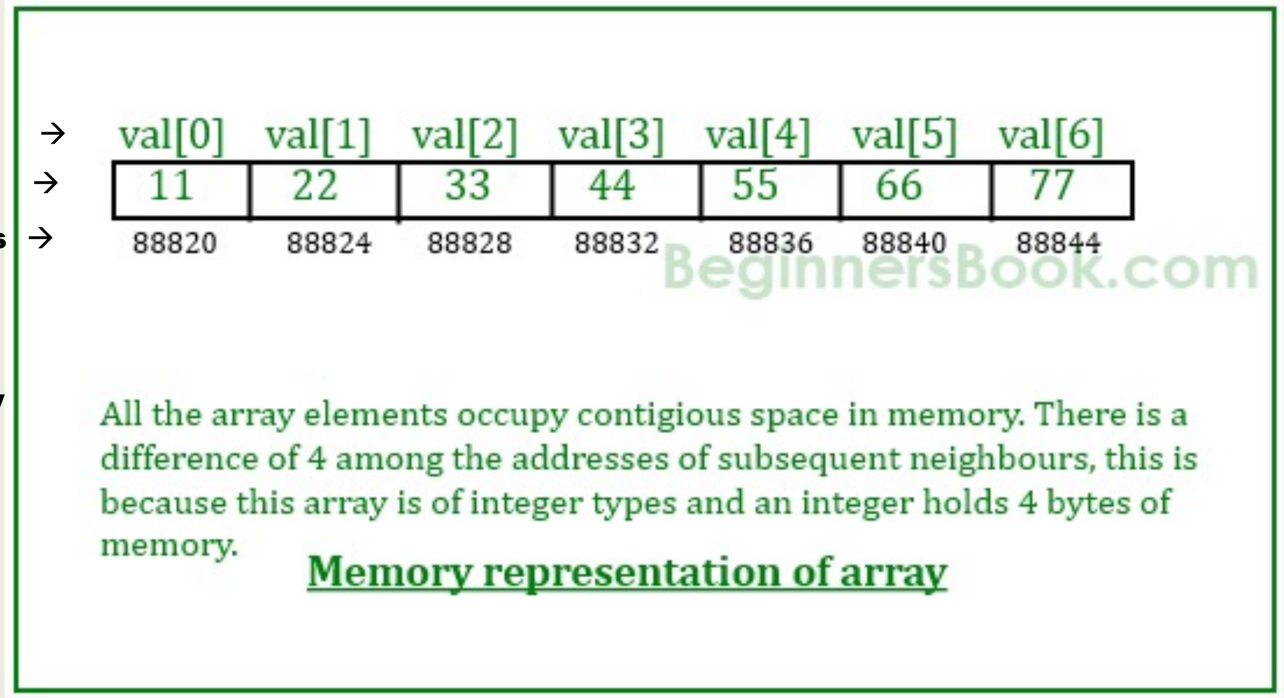
Online compiler
https://rextester.com/l/c_online_compiler_gcc

Arrays example

Indexed access →

value →

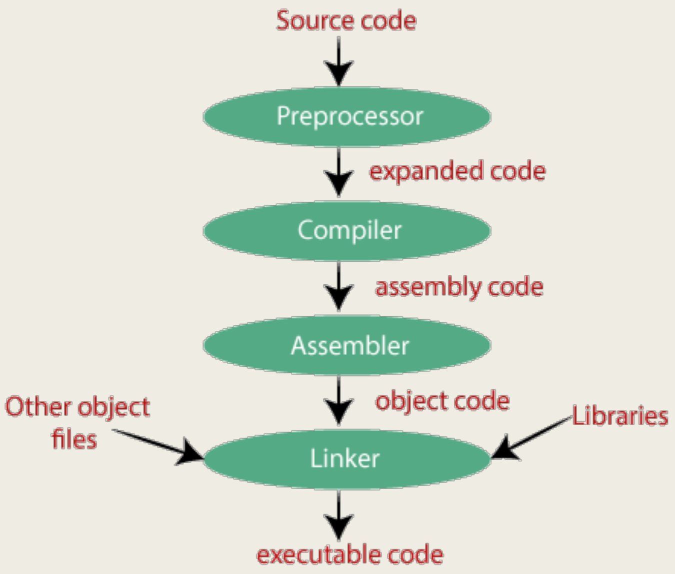
Memory address →



The address of the array corresponds to the address of the first element &val[0] == &val

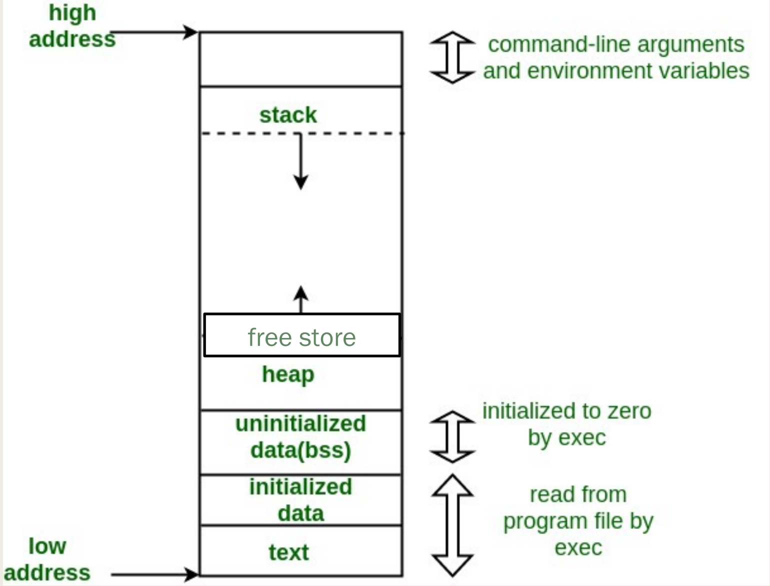
Compilation process

- Preprocessor: expand code in the source files (.c, .cpp))
- Compiler: converts the expanded code into assembly code (.obj)
- Assembler: creates object code (incomplete binary code) using the assembly code (.lib, .o, .a)
- Linker: combine object codes and libraries into the final executable



Memory layout of a C program

- Text: executable instructions
- Initialized data: global and static variables
- Uninitialized data: global and static not initialized variables
- Stack: in a stack frame are stored local variables and function pointers
- Heap and free store: dynamic memory allocation



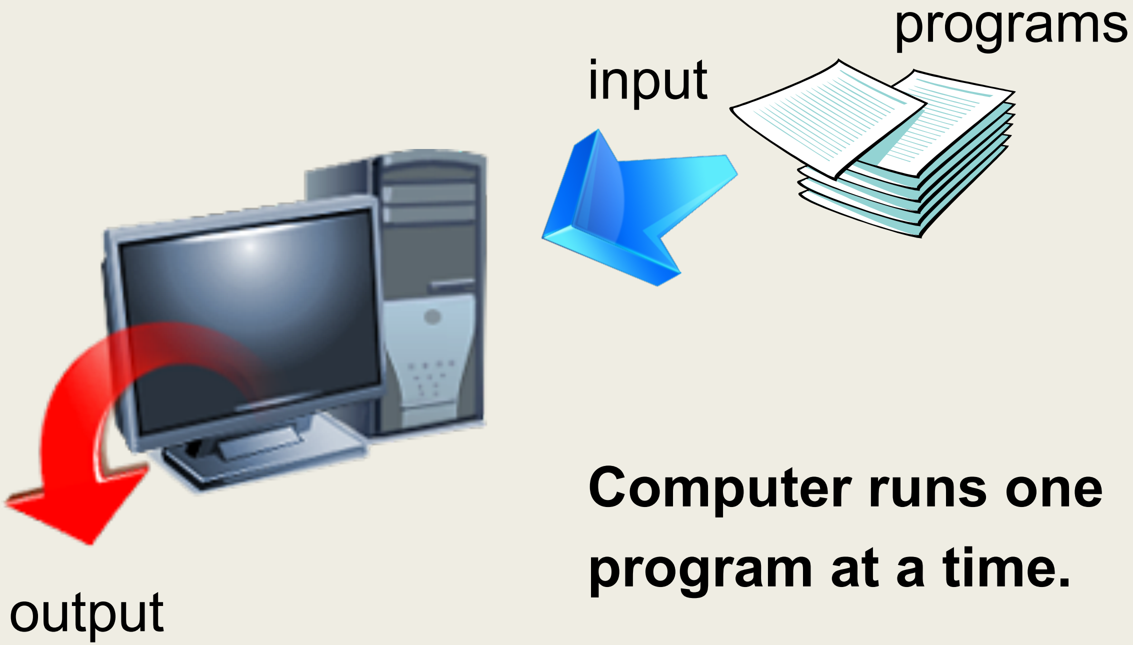
Memory management

- Heap and free store
  - *Free store, dynamic storage allocated/freed by new/delete*
  - *Heap (legacy from C), dynamic storage allocated/freed by malloc/free*

Differences between new and malloc

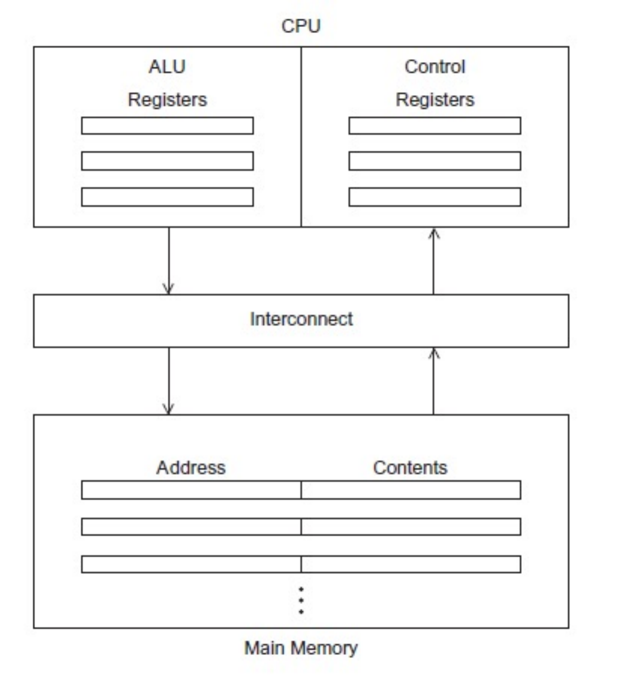
NEW	MALLOC
calls constructor	doesnot calls constructors
It is an operator	It is a function
Returns exact data type	Returns void *
on failure, Throws	On failure, returns NULL
Memory allocated from free store	Memory allocated from heap
can be overridden	cannot be overridden
size is calculated by compiler	size is calculated manually

Serial hardware and software



Computer runs one program at a time.

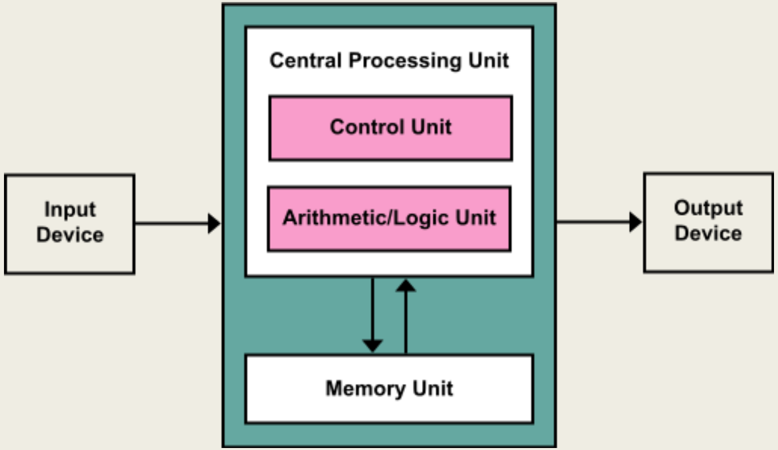
The von Neumann Architecture



- Memory and CPU with an interconnection in between
- Data and execution state are stored in (very fast) registers memory

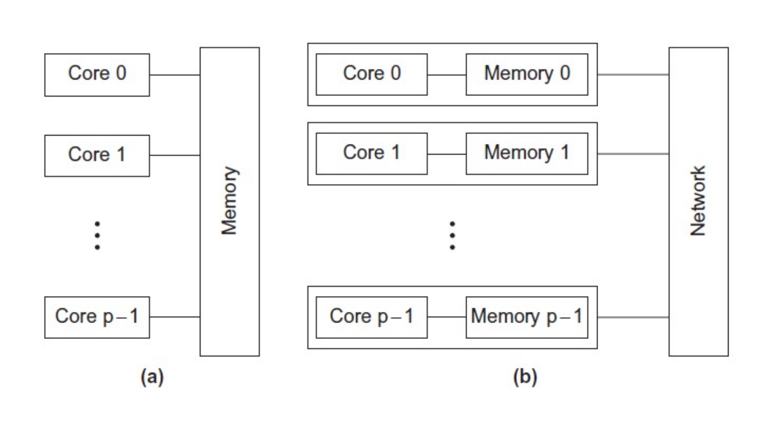
Central processing unit (CPU)

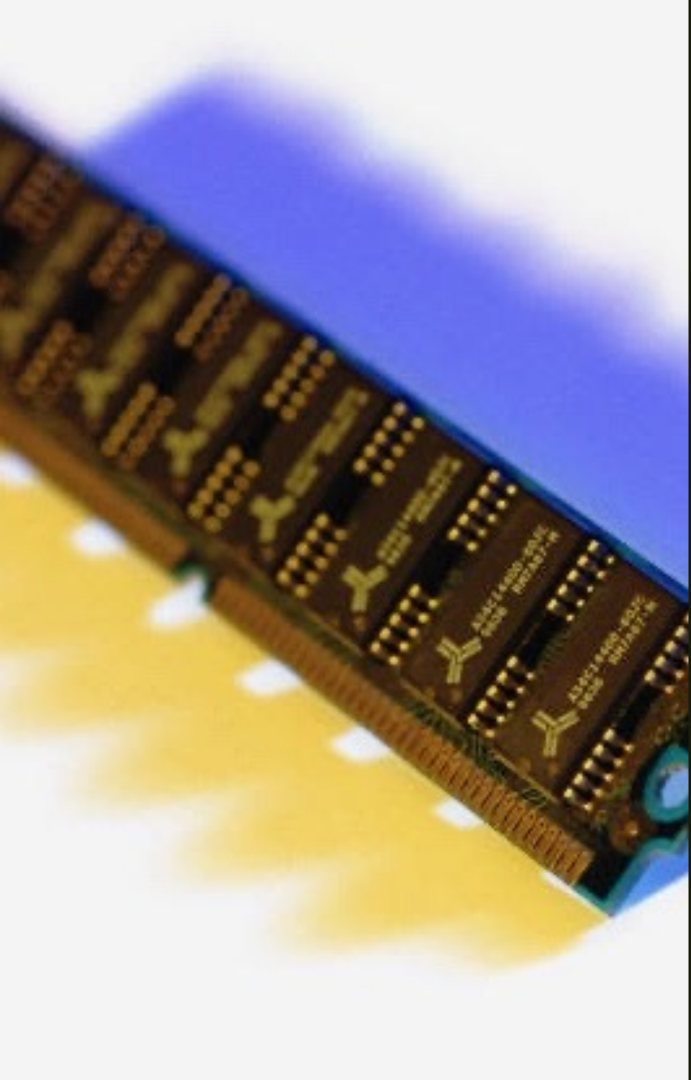
- Divided into two parts.
- **Control unit** - responsible for deciding which instruction in a program should be executed. (*the boss*)
- **Arithmetic and logic unit** (ALU) - responsible for executing the actual instructions. (*the worker*)



Type of parallel systems

- Shared-memory
  - *The cores can share access to the computer's memory.*
  - *Coordinate the cores by having them examine and update shared memory locations.*
- Distributed-memory
  - *Each core has its own, private memory.*
  - *The cores must communicate explicitly by sending messages across a network.*





Main memory

- This is a collection of locations, each of which is capable of storing both instructions and data.
- Every location consists of an address, which is used to access the location, and the contents of the location.

Some definitions

- **Register** – very fast storage, part of the CPU.
- **Program counter** – stores address of the next instruction to be executed.
- **Bus** – wires and hardware that connects the CPU and memory.

An operating system “process”

- An instance of a computer program that is being executed.
- Components of a process:
  - *The executable machine language program*
  - *A block of memory (executable code, call stack, heap)*
  - *Descriptors of resources the OS has allocated to the process*
  - *Security information (which resources the process can access)*
  - *Information about the state of the process (registers, running state)*

Multitasking

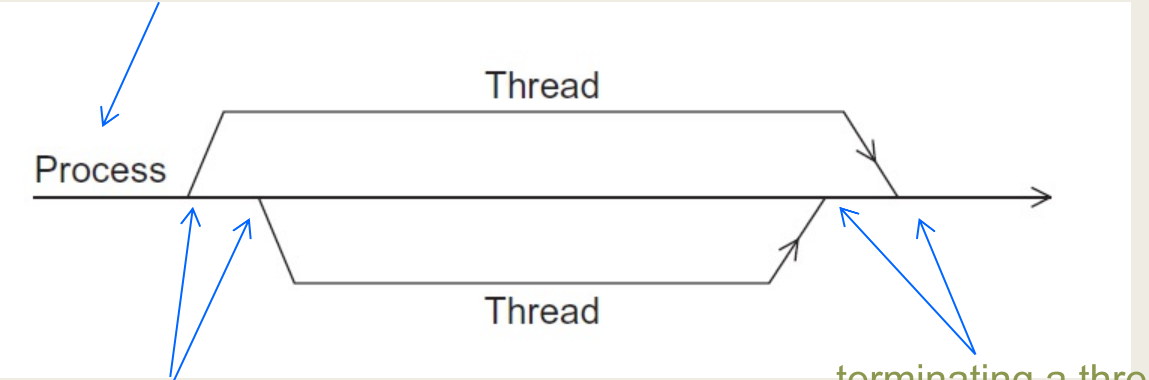
- Gives the **illusion** that a single processor system is running multiple programs simultaneously.
- Each process takes turns running. (**time slice**)
- After its time is up (or has to wait for resources), it waits until it has a turn again. (**blocks**)

Threading

- Threads are contained within processes.
- They allow programmers to divide their programs into (more or less) independent tasks.
- The hope is that when one thread blocks because it is waiting on a resource, another will have work to do and can run.

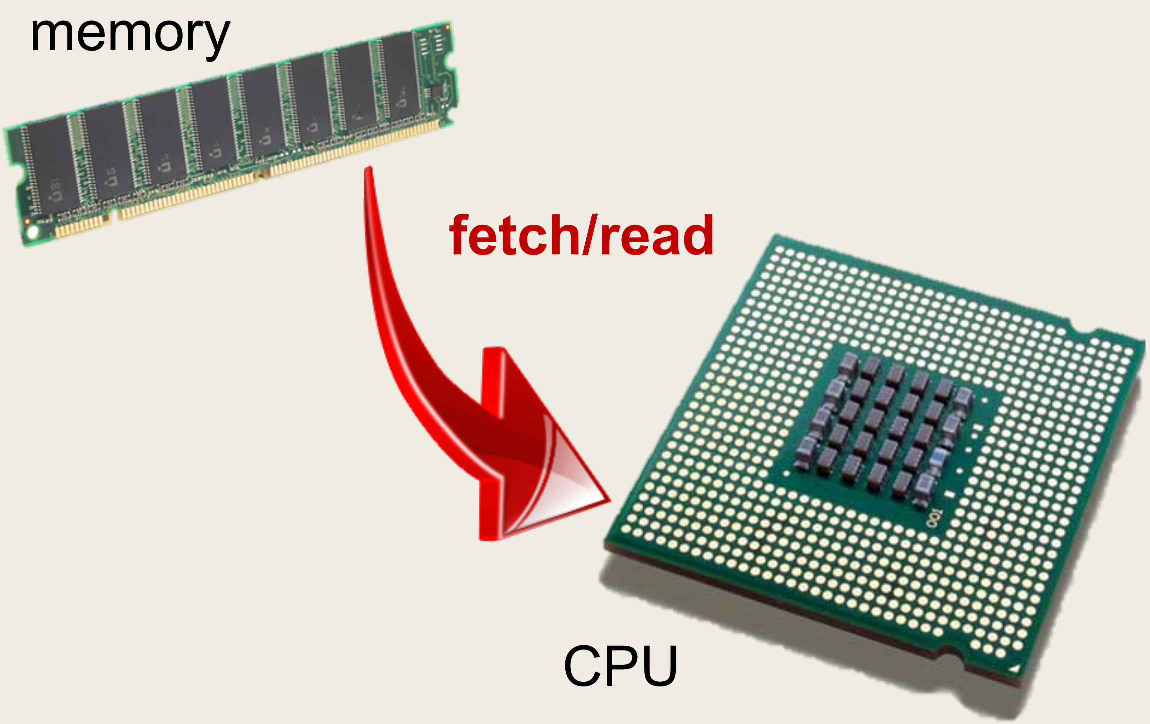
A process and two threads

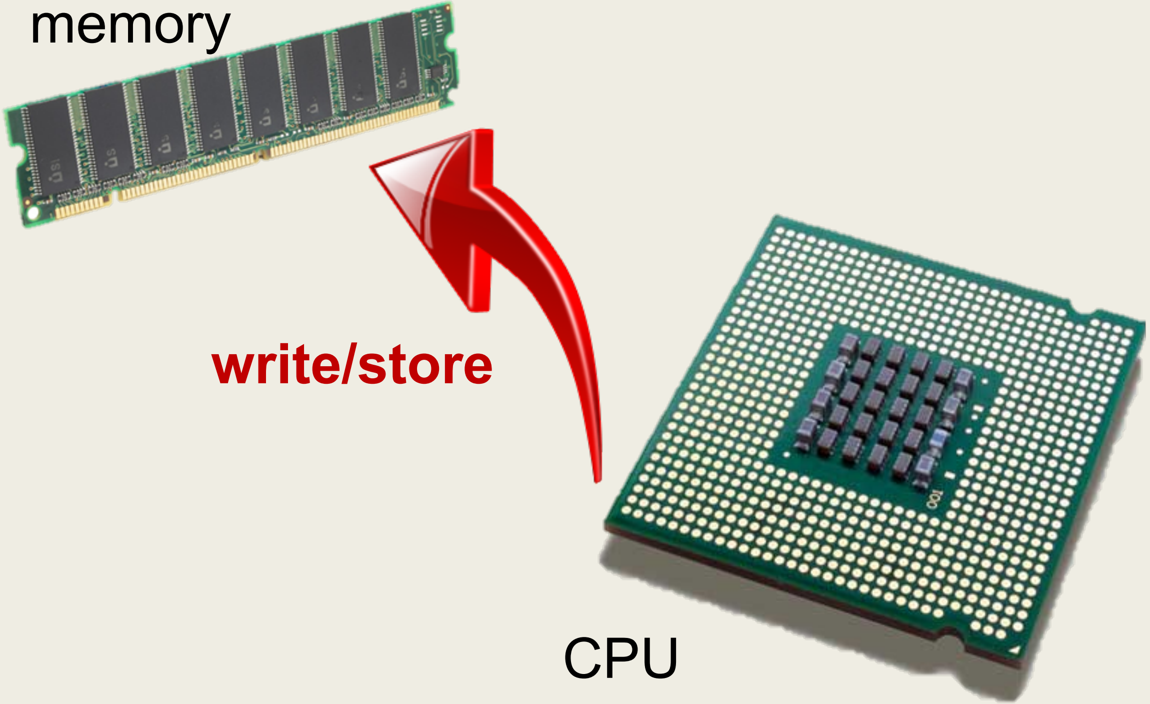
the “master” thread



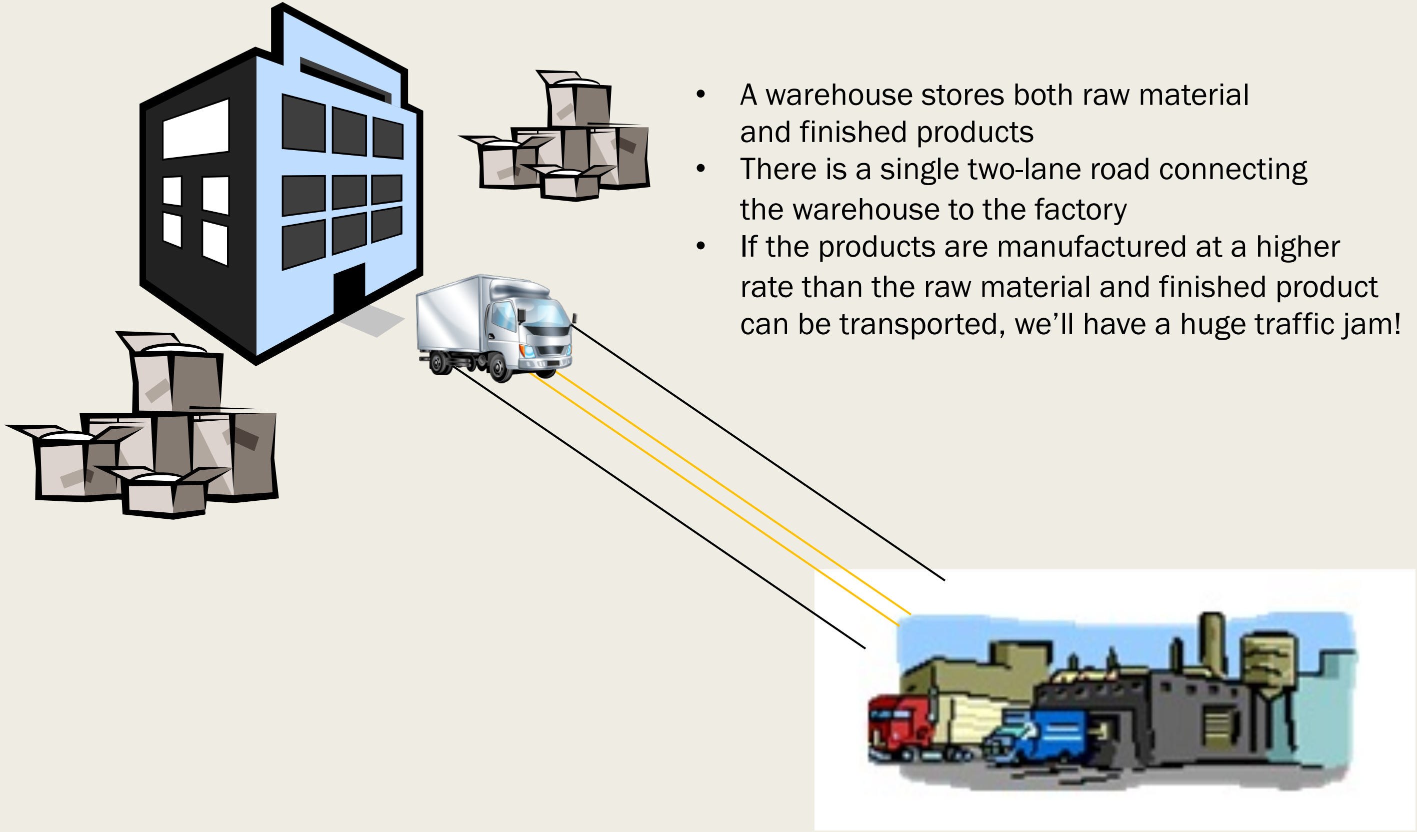
starting a thread
Is called ***forking***

terminating a thread
Is called ***joining***

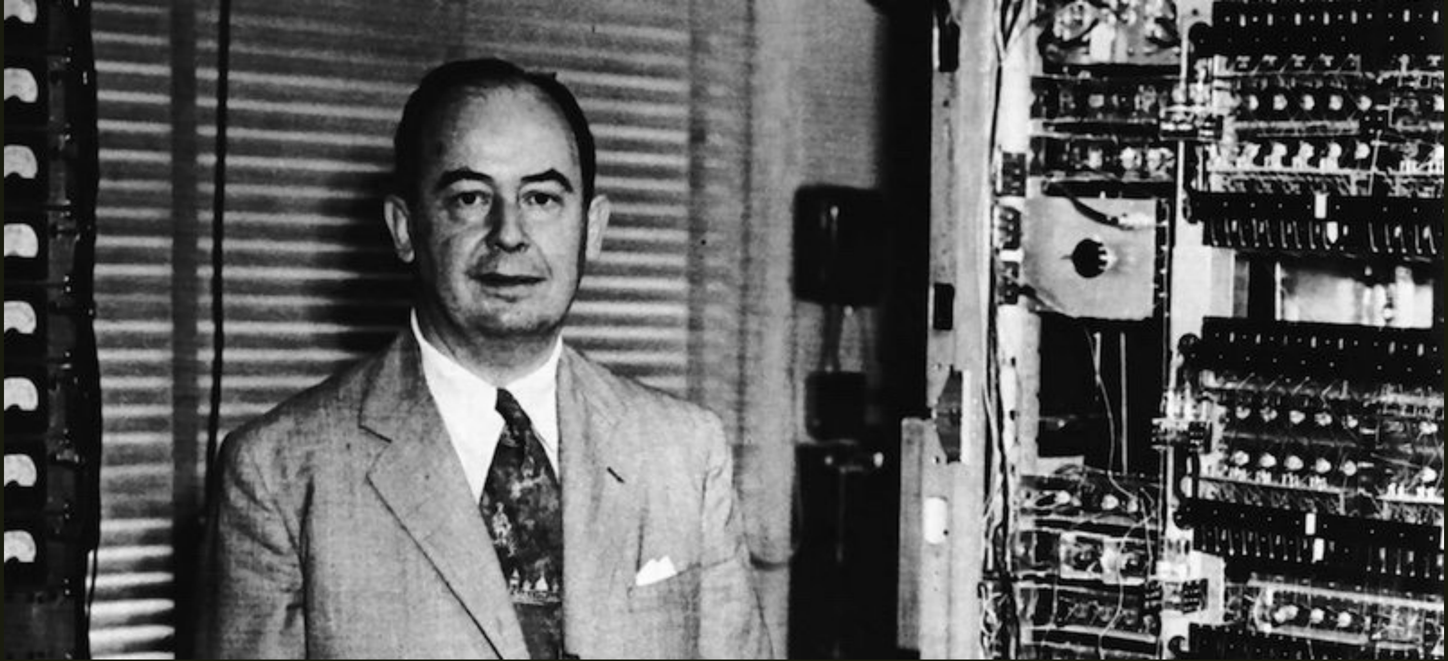

memory
fetch/read
CPU


memory
write/store
CPU

von Neumann bottleneck



- A warehouse stores both raw material and finished products
- There is a single two-lane road connecting the warehouse to the factory
- If the products are manufactured at a higher rate than the raw material and finished product can be transported, we'll have a huge traffic jam!



MODIFICATIONS TO THE VON NEUMANN MODEL

INTRODUCTION TO CACHE (VALLEY?)

No, not that kind of cache!

Basics of caching

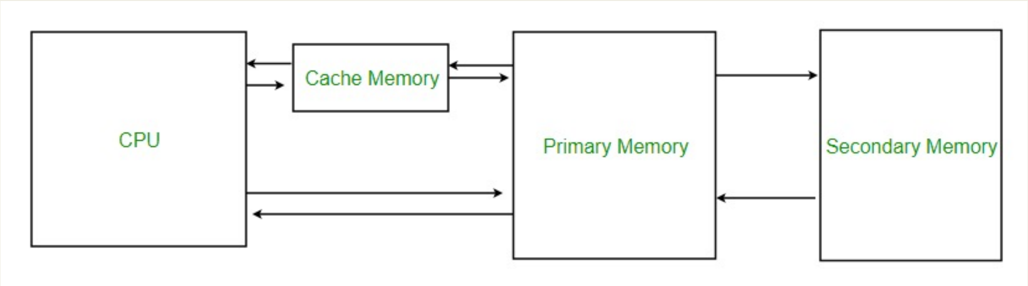
- A collection of memory locations that can be accessed in less time than some other memory locations.
- A CPU cache is typically located on the same chip, or one that can be accessed much faster than ordinary memory.

Principle of locality

- Accessing one location is followed by an access of a nearby location.

- **Spatial locality** – accessing a nearby location.

- **Temporal locality** – accessing in the near future.

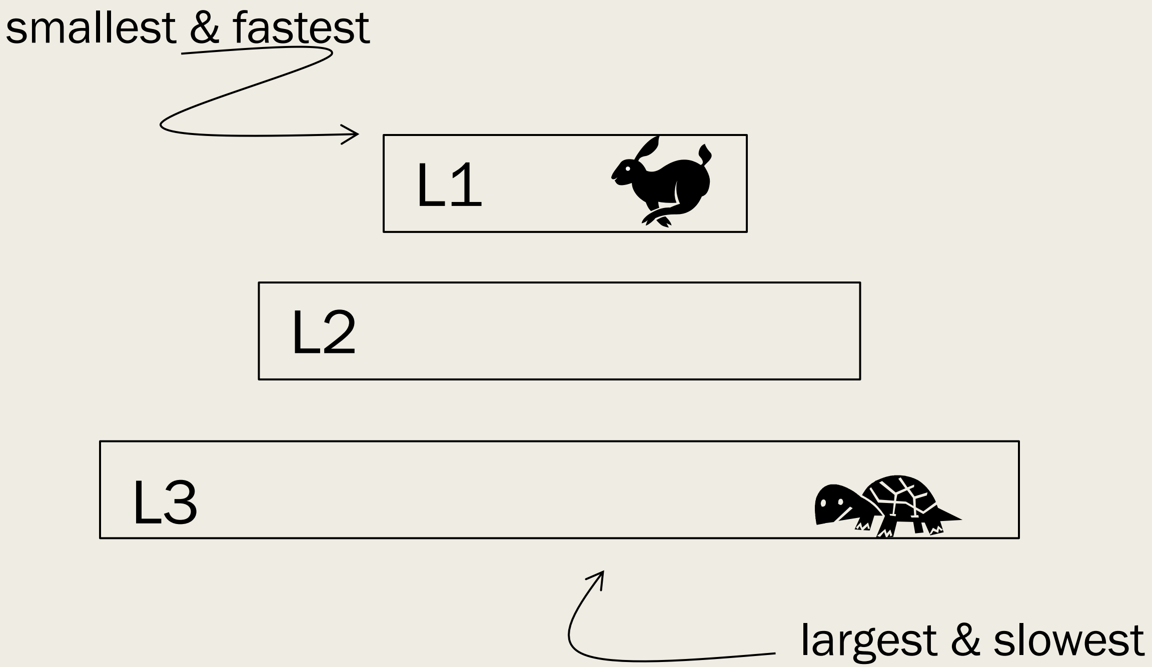


Principle of locality

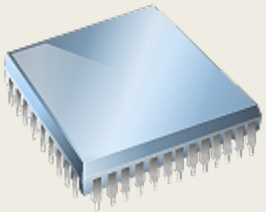
```
float z[1000];
…
sum = 0.0;
for (i = 0; i < 1000; i++)
   sum += z[i];
```

- Cache is accessed in blocks (or cache lines)
- Cache blocks are 8-16 times bigger than memory locations
- When reading one entry of an array
  we are actually fetching 8-16 values in the cache

Levels of Cache



Cache hit



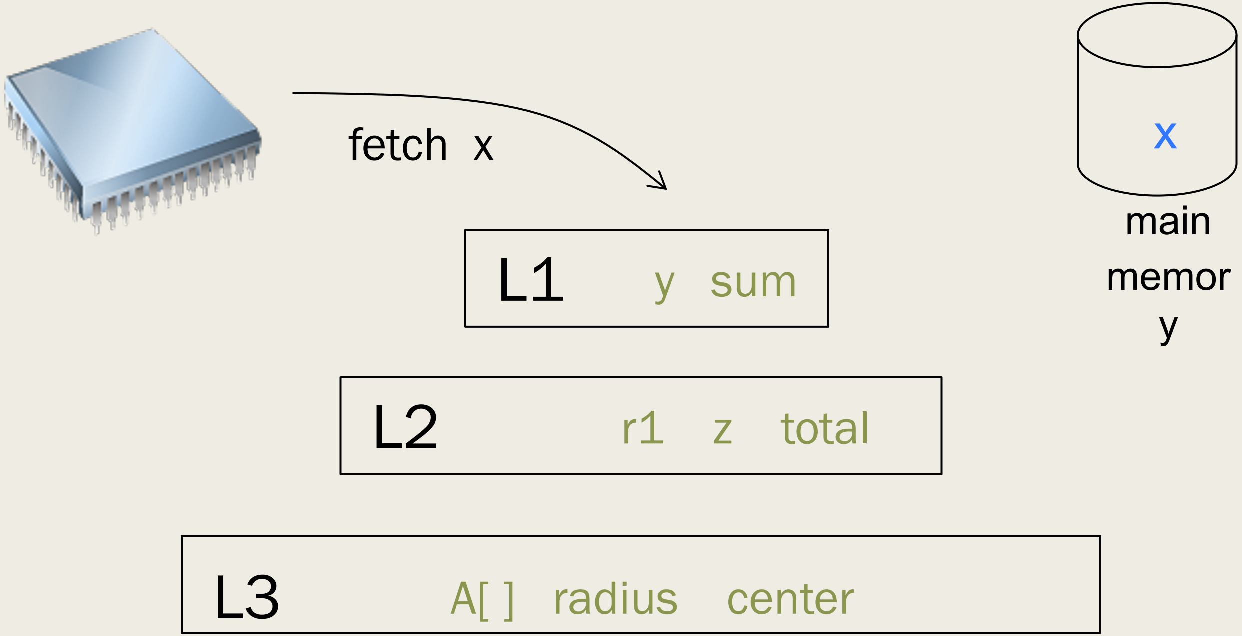
fetch x

L1 x sum

L2 y z total

L3 A[ ] radius r1 center

Cache miss



Issues with cache

- When a CPU writes data to cache, the value in cache may be inconsistent with the value in main memory.
- **Write-through** caches handle this by updating the data in main memory at the time it is written to cache.
- **Write-back** caches mark data in the cache as dirty. When the cache line is replaced by a new cache line from memory, the dirty line is written to memory.

Cache mappings

- **Full associative** – a new line can be placed at any location in the cache.
- **Direct mapped** – each cache line has a unique location in the cache to which it will be assigned.
- ***n*-way set associative** – each cache line can be place in one of *n* different locations in the cache.

n-way set associative

- When more than one line in memory can be mapped to several different locations in cache we also need to be able to decide which line should be replaced or evicted.
- Common solution: evict the Least Recently Used (LRU)

Example

Memory Index	Cache Location		
	Fully Assoc	Direct Mapped	2-way
0	0, 1, 2, or 3	0	0 or 1
1	0, 1, 2, or 3	1	2 or 3
2	0, 1, 2, or 3	2	0 or 1
3	0, 1, 2, or 3	3	2 or 3
4	0, 1, 2, or 3	0	0 or 1
5	0, 1, 2, or 3	1	2 or 3
6	0, 1, 2, or 3	2	0 or 1
7	0, 1, 2, or 3	3	2 or 3
8	0, 1, 2, or 3	0	0 or 1
9	0, 1, 2, or 3	1	2 or 3
10	0, 1, 2, or 3	2	0 or 1
11	0, 1, 2, or 3	3	2 or 3
12	0, 1, 2, or 3	0	0 or 1
13	0, 1, 2, or 3	1	2 or 3
14	0, 1, 2, or 3	2	0 or 1
15	0, 1, 2, or 3	3	2 or 3

Assignments of a 16-line main memory to a 4-line cache

Caches and programs

How is a matrix stored in memory?

```
double A[MAX][MAX], x[MAX], y[MAX];
. . .
/* Initialize A and x, assign y = 0 */
. . .
/* First pair of loops */
for (i = 0; i < MAX; i++)
   for (j = 0; j < MAX; j++)
      y[i] += A[i][j]*x[j];
. . .
/* Assign y = 0 */
. . .
/* Second pair of loops */
for (j = 0; j < MAX; j++)
   for (i = 0; i < MAX; i++)
      y[i] += A[i][j]*x[j];
```

Cache Line	Elements of A			
0	A[0][0]	A[0][1]	A[0][2]	A[0][3]
1	A[1][0]	A[1][1]	A[1][2]	A[1][3]
2	A[2][0]	A[2][1]	A[2][2]	A[2][3]
3	A[3][0]	A[3][1]	A[3][2]	A[3][3]

Which pair of loops is faster?

Hint: count the number of cache miss

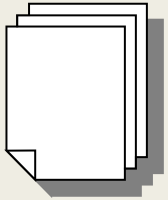
Virtual memory (1)

- If we run a very large program or a program that accesses very large data sets, all of the instructions and data may not fit into main memory.
- Virtual memory functions as a cache for secondary storage.
- It exploits the principle of spatial and temporal locality.
- It only keeps the active parts of running programs in main memory.

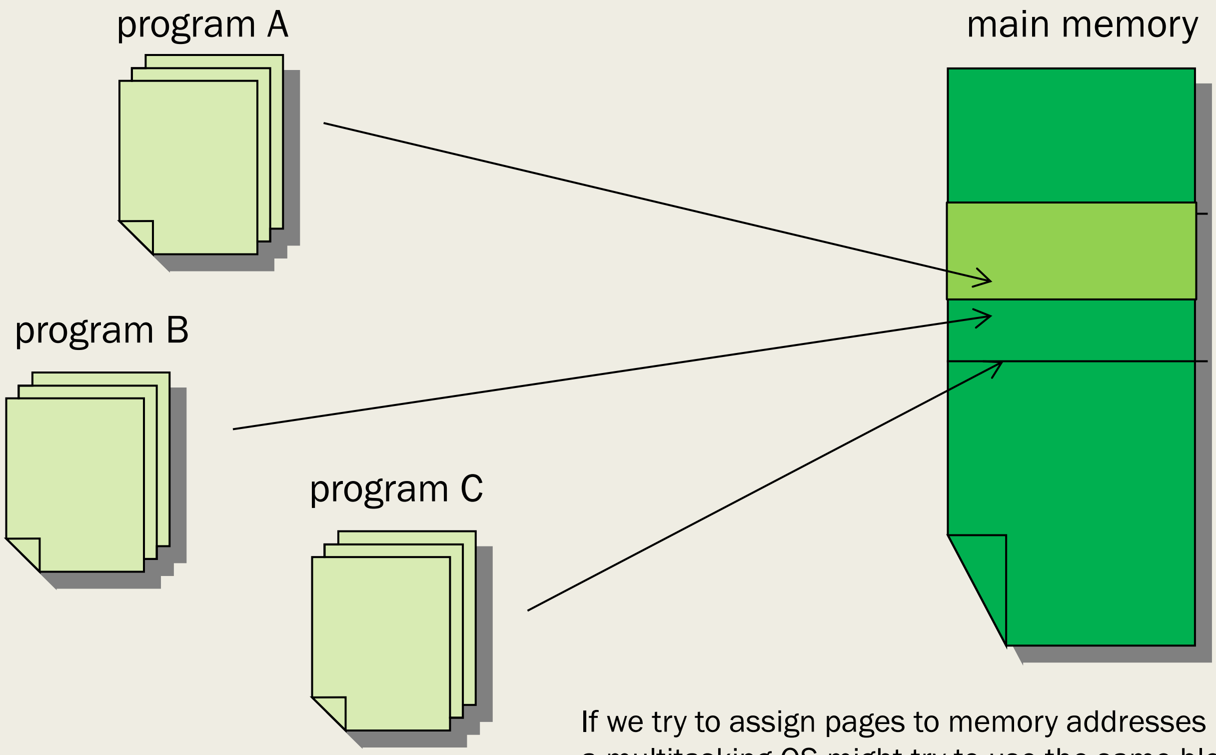
Virtual memory (2)

- **Swap space** - those parts that are idle are kept in a block of secondary storage.

- **Pages** – blocks of data and instructions.
  - *Usually these are relatively large.*
  - *Most systems have a fixed page size that currently ranges from 4 to 16 kilobytes.*



Virtual memory (3)



If we try to assign pages to memory addresses
a multitasking OS might try to use the same blocks
for different programs!

Virtual page numbers

- When a program is compiled its pages are assigned *virtual* page numbers.
- When the program is run, a table is created that maps the virtual page numbers to physical addresses.
- A **page table** is used to translate the virtual address into a physical address.

Page table

Virtual Address									
Virtual Page Number					Byte Offset				
31	30	⋯	13	12	11	10	⋯	1	0
1	0	⋯	1	1	0	0	⋯	1	1

Virtual Address Divided into Virtual Page Number and Byte Offset

- Using a page table has the potential to significantly increase each program's overall run-time.
- A special address translation cache in the processor.

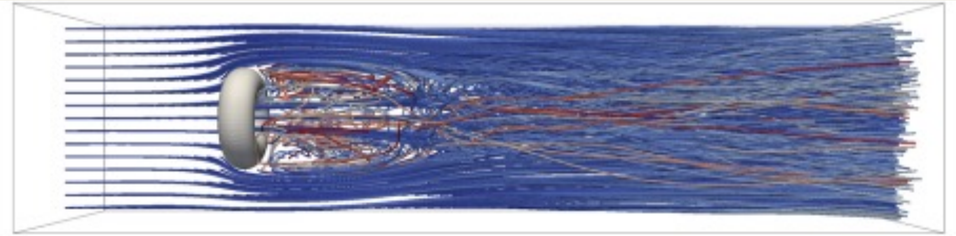
Translation-lookaside buffer (2)

- It caches a small number of entries (typically 16–512) from the page table in very fast memory.

- **Page fault** – attempting to access a valid physical address for a page in the page table but the page is only stored on disk.

When data is too large, think about your main memory as a “cache”

- When you deal with very large datasets you can’t store them all in memory
- You need to develop some strategy to fetch the portions of data that you need. This is also called out-of-core computation
- Example: computing pathlines/streamlines on a large vector field
  - *Which blocks of data do I **fetch** first?*
  - *Which blocks of data do I **evict** first?*



Project Idea!

Similar Example car aerodynamics: https://youtu.be/2RBOtd-Z8O8?t=46