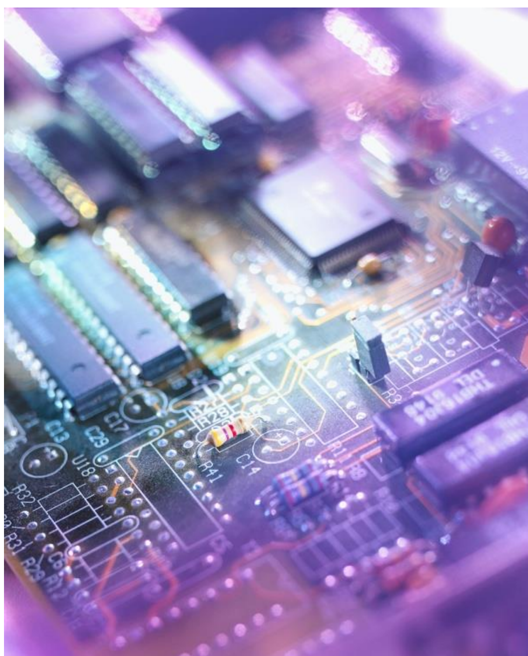
中山大學 SUN YAT-SEN UNIVERSITY

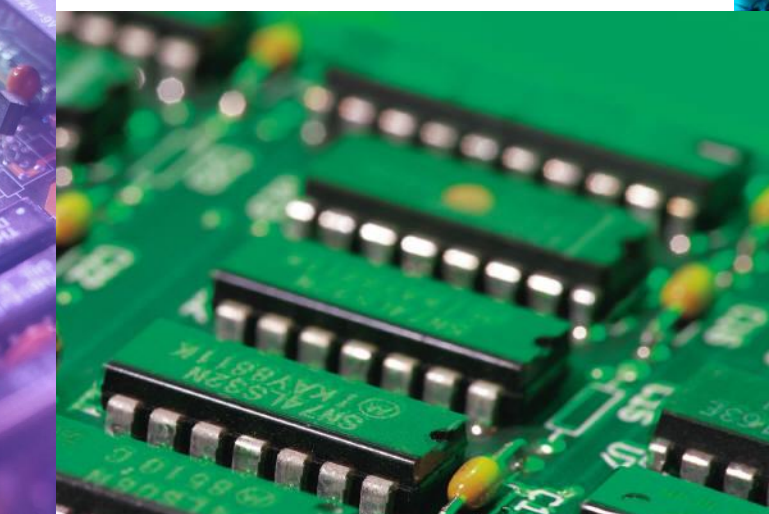
国家超级计算广州中心 NATIONAL SUPERCOMPUTER CENTER IN GUANGZHOU

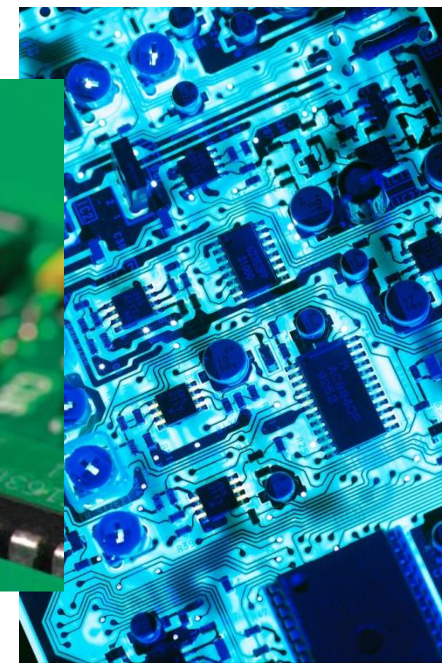
并行硬件和并行软件

任课教师：吴迪

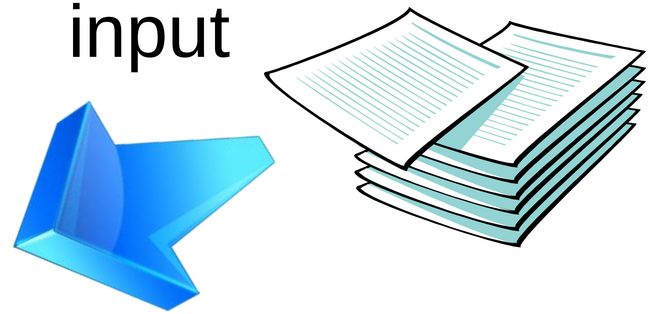
Some background





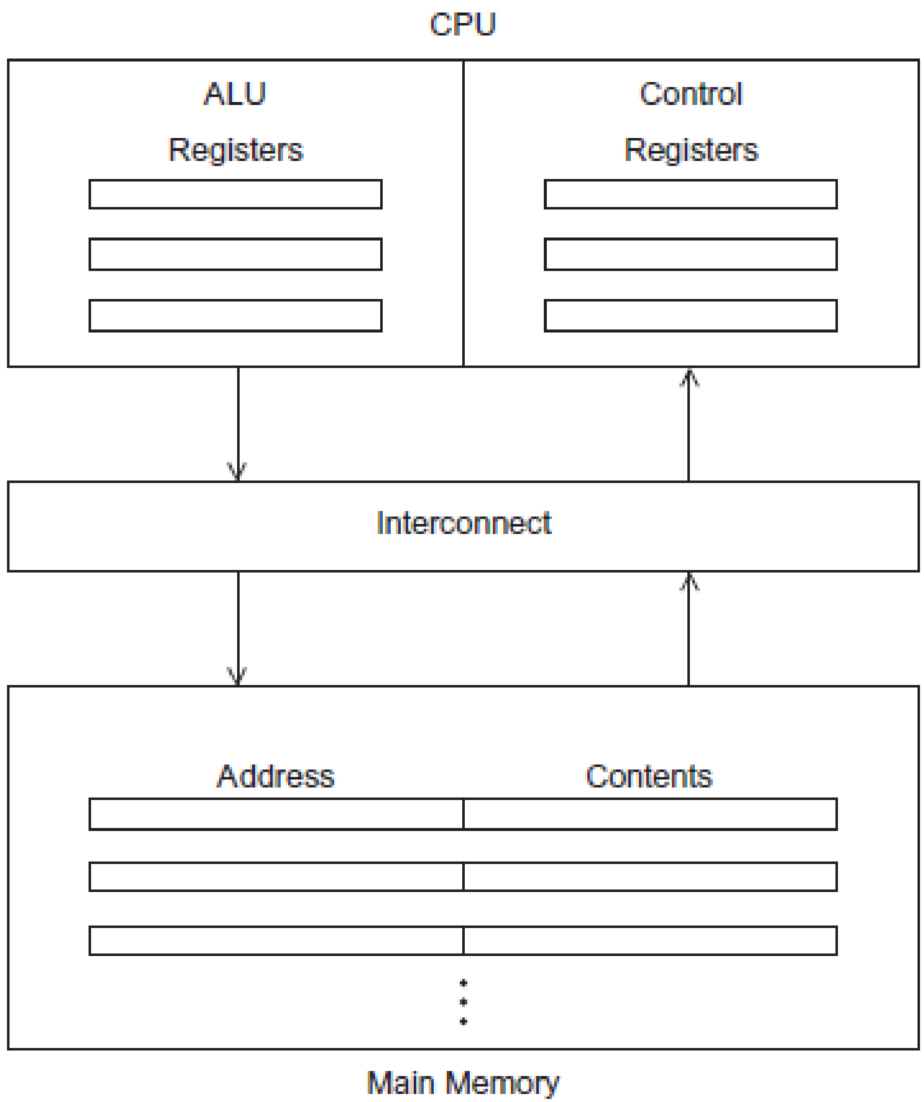


Serial hardware and software



Computer runs one program at a time.

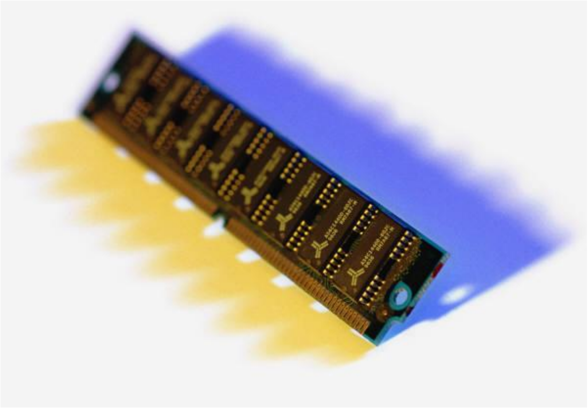
The von Neumann Architecture



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.



Central processing unit (CPU)

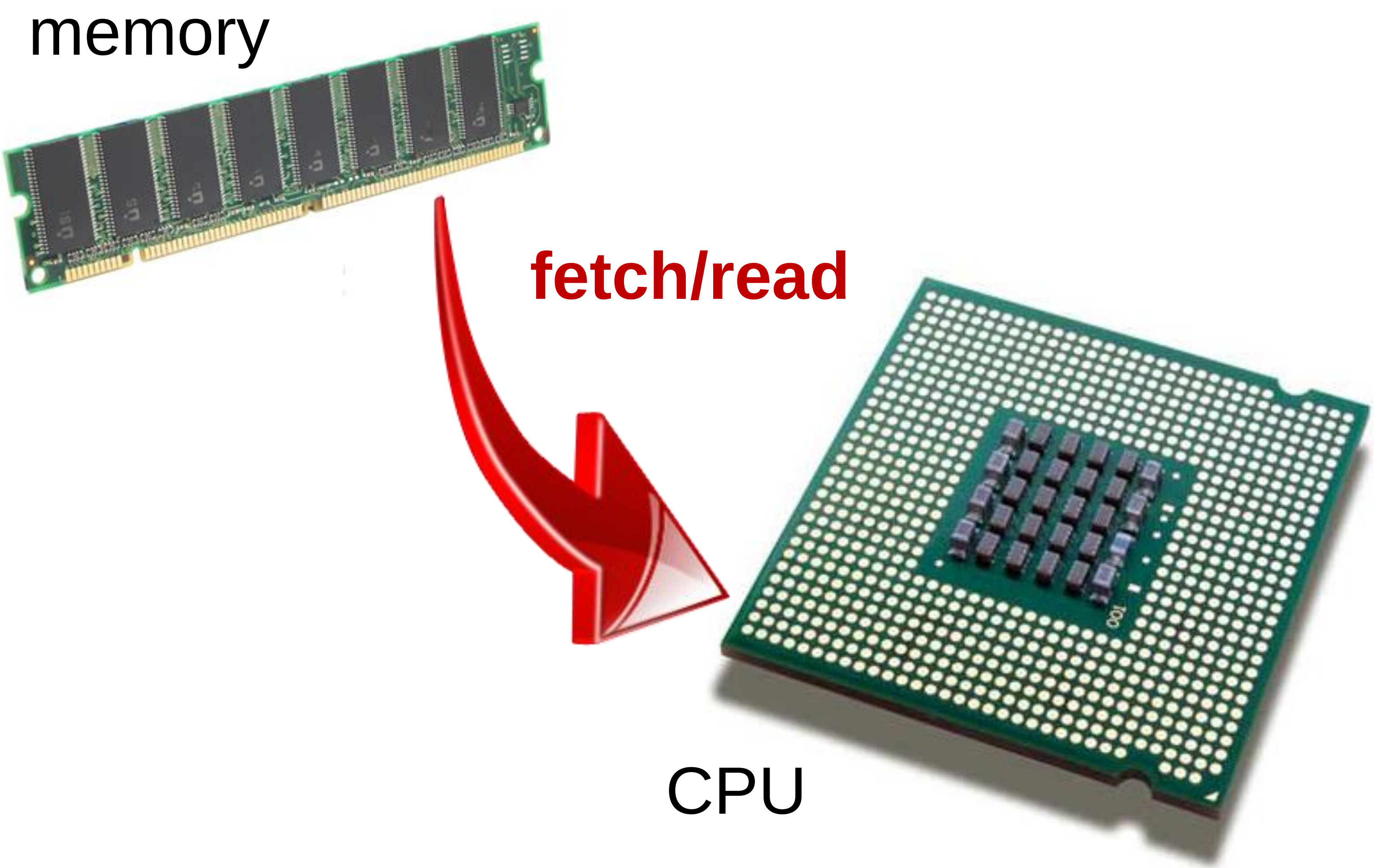
- Divided into two parts.

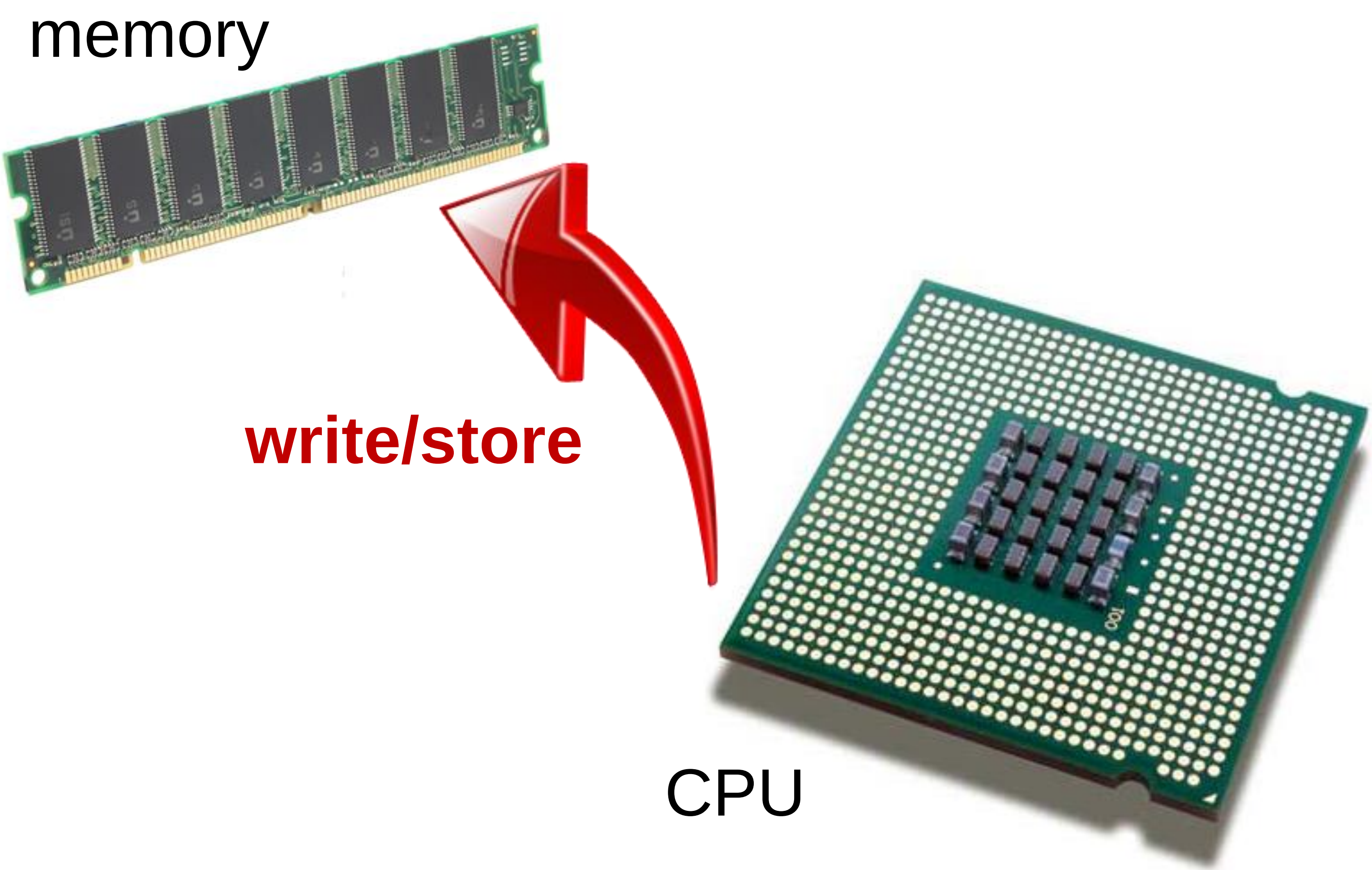
- **Control unit** - responsible for deciding which instruction in a program should be executed. (*the boss*)

- **Arithmetic and logic unit** (ALU) - responsible for executing the actual instructions. (*the worker*)

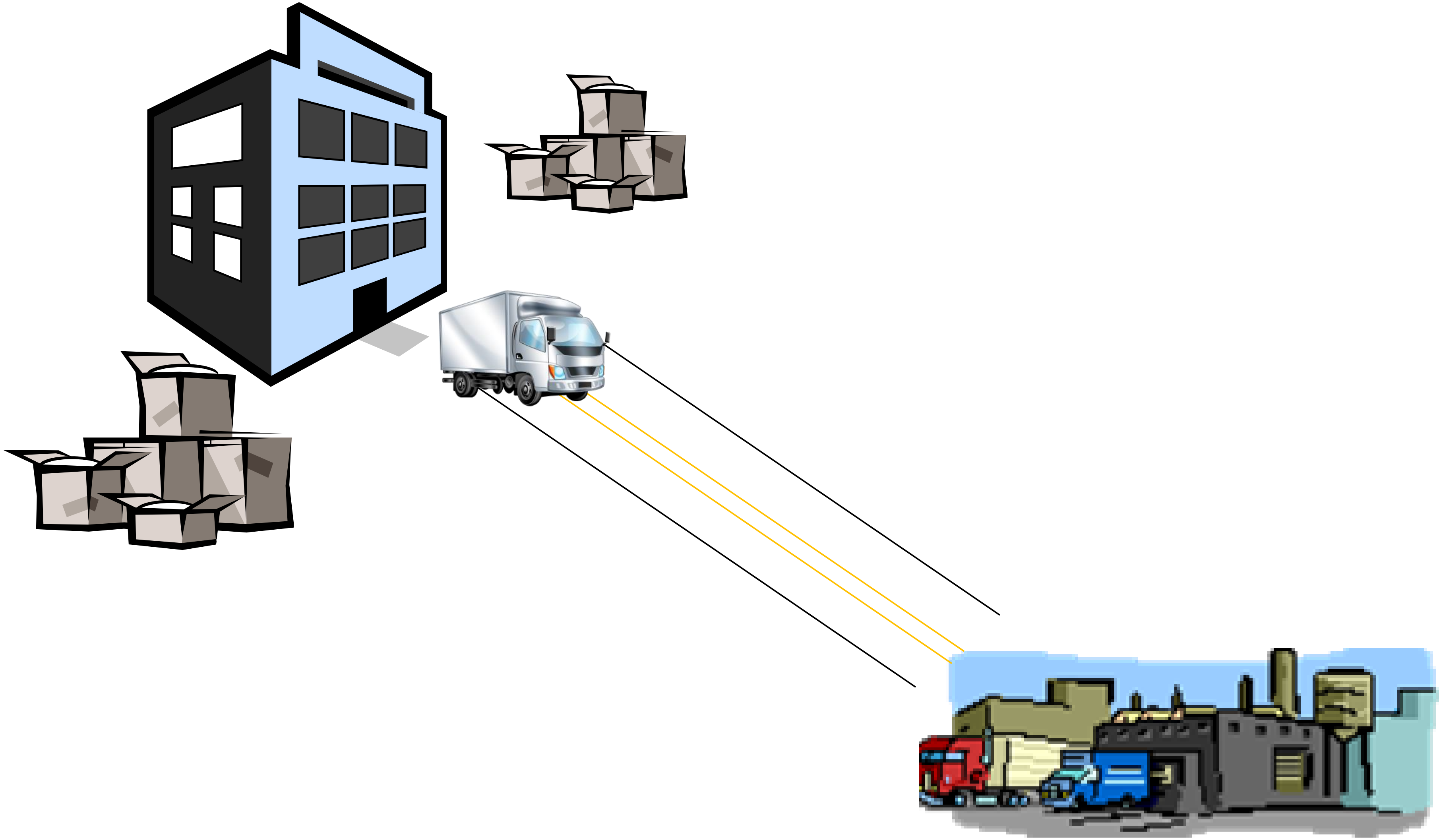
Key terms

- **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.


memory
fetch/read
CPU


memory
write/store
CPU

von Neumann bottleneck



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.
  - Descriptors of resources the OS has allocated to the process.
  - Security information.
  - Information about the state of the process.

Multitasking

- Gives the illusion that a single processor system is running multiple programs simultaneously.

- Each process takes turns running. (**time slice**)

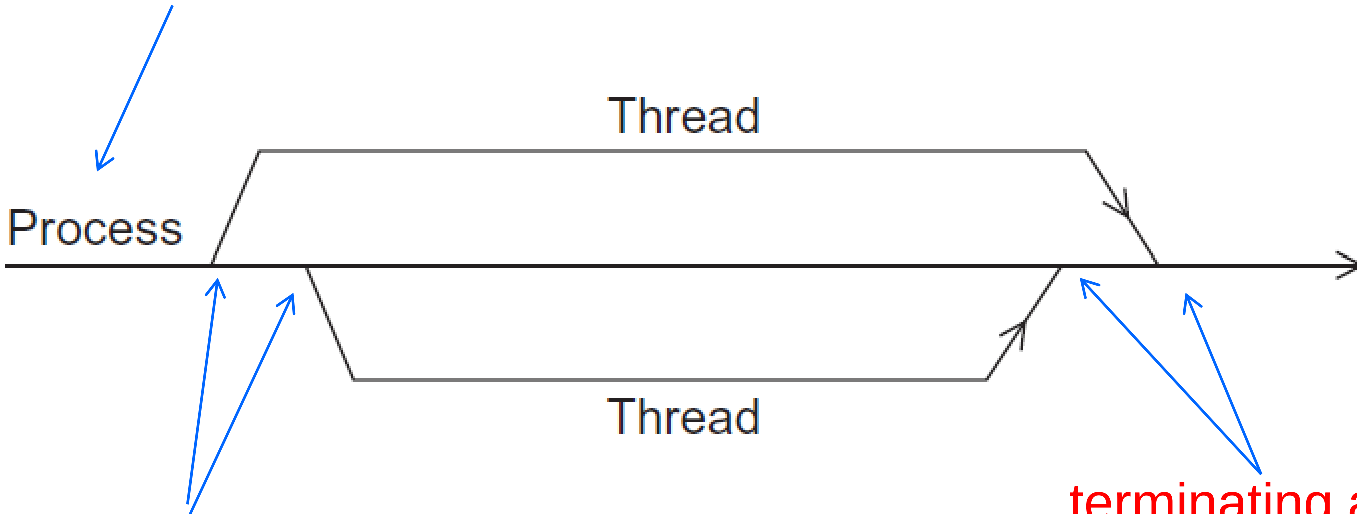
- After its time is up, it waits until it has a turn again.

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.

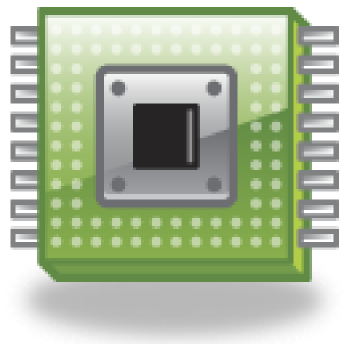
A process and two threads

the “master” thread



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

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



Modifications to the von neumann model

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.

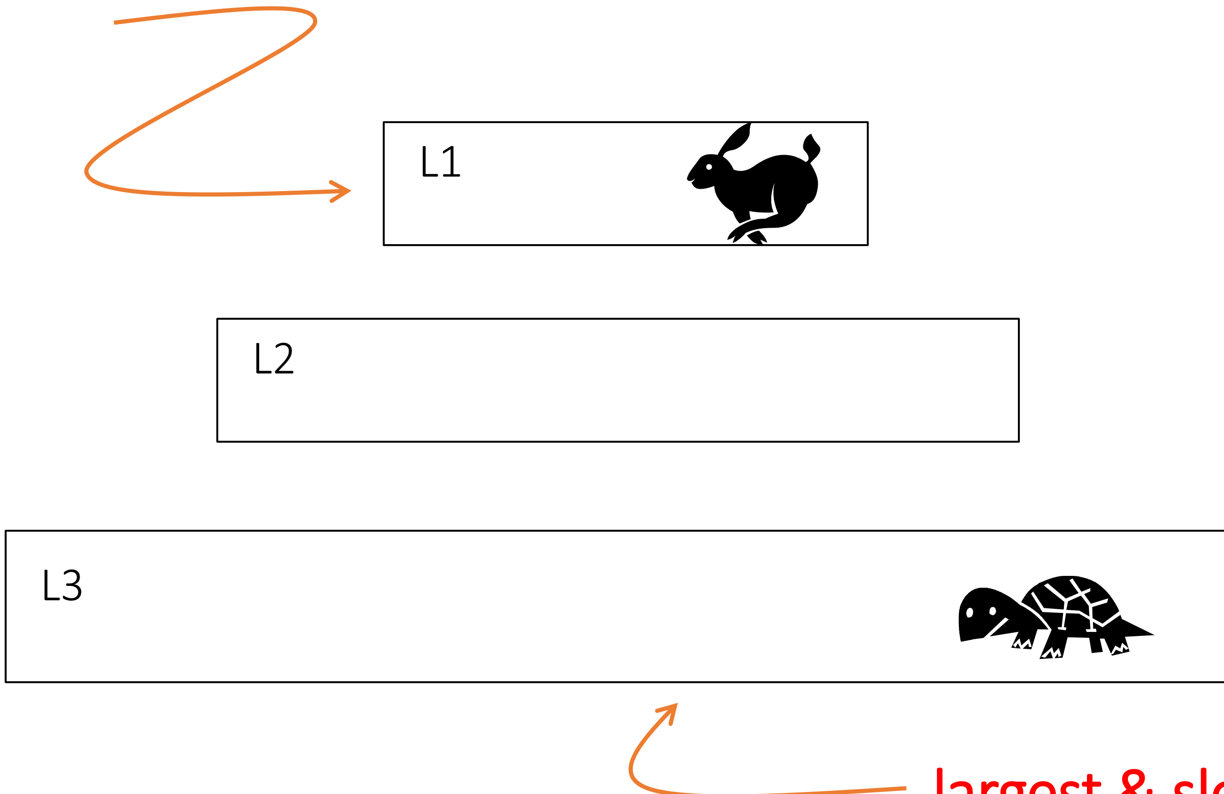
- **Temporal locality** – accessing in the near future.

Principle of locality

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

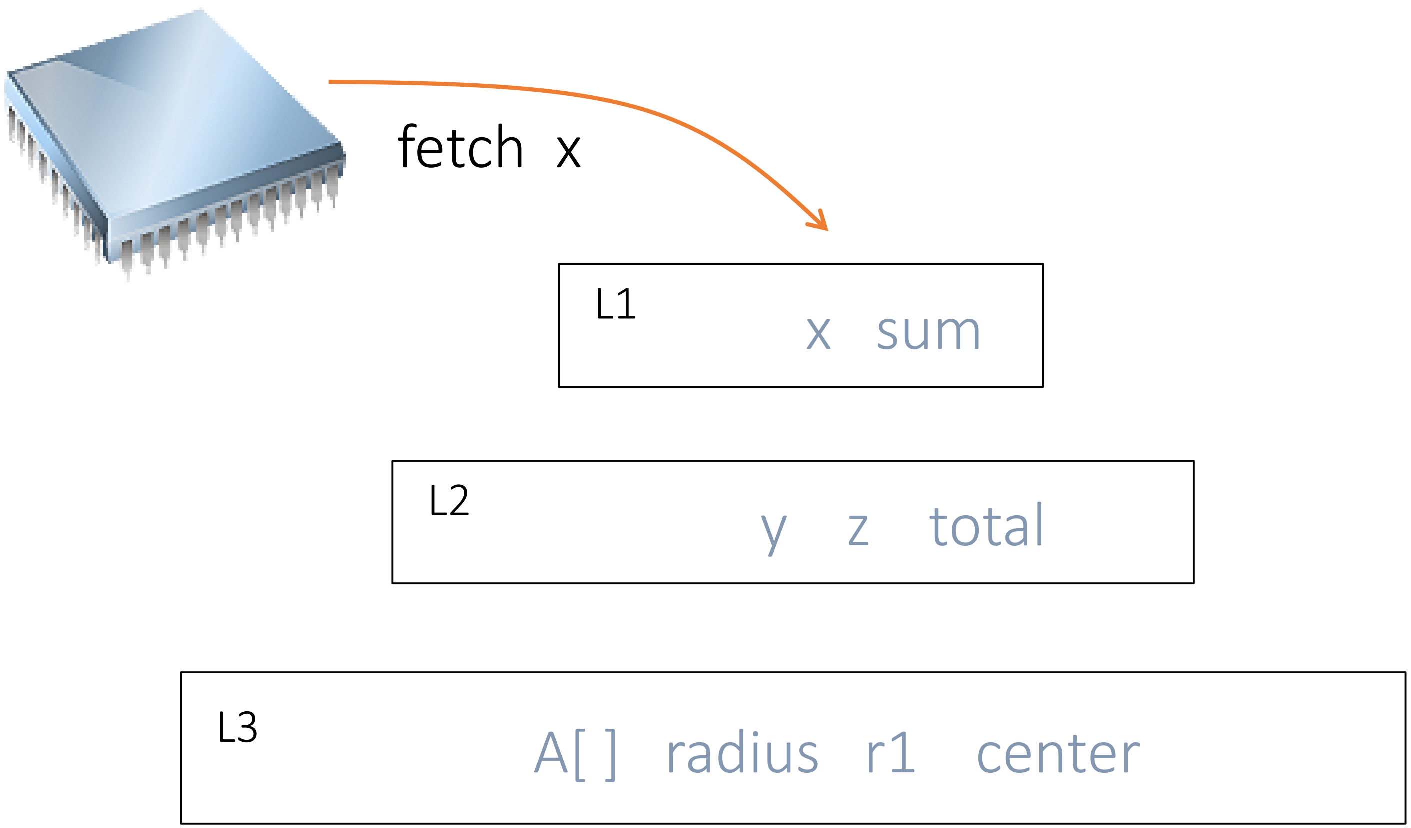
Levels of Cache

smallest & fastest

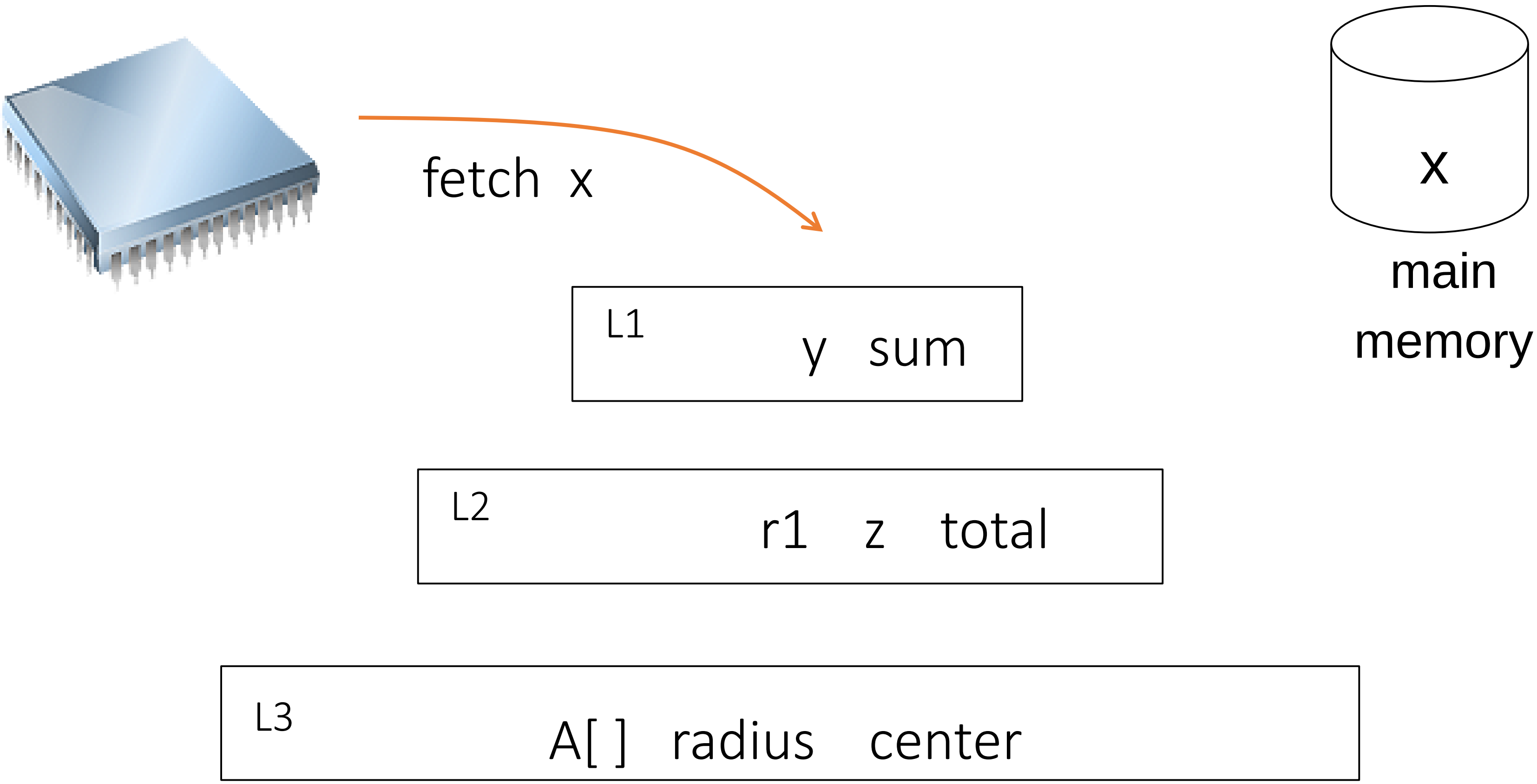


largest & slowest

Cache hit



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 placed 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.

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

Table 2.1: Assignments of a 16-line main memory to a 4-line cache

Caches and programs

```
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]

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.

Virtual memory (2)

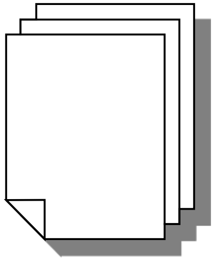
- It exploits the principle of spatial and temporal locality.

- It only keeps the active parts of running programs in main memory.

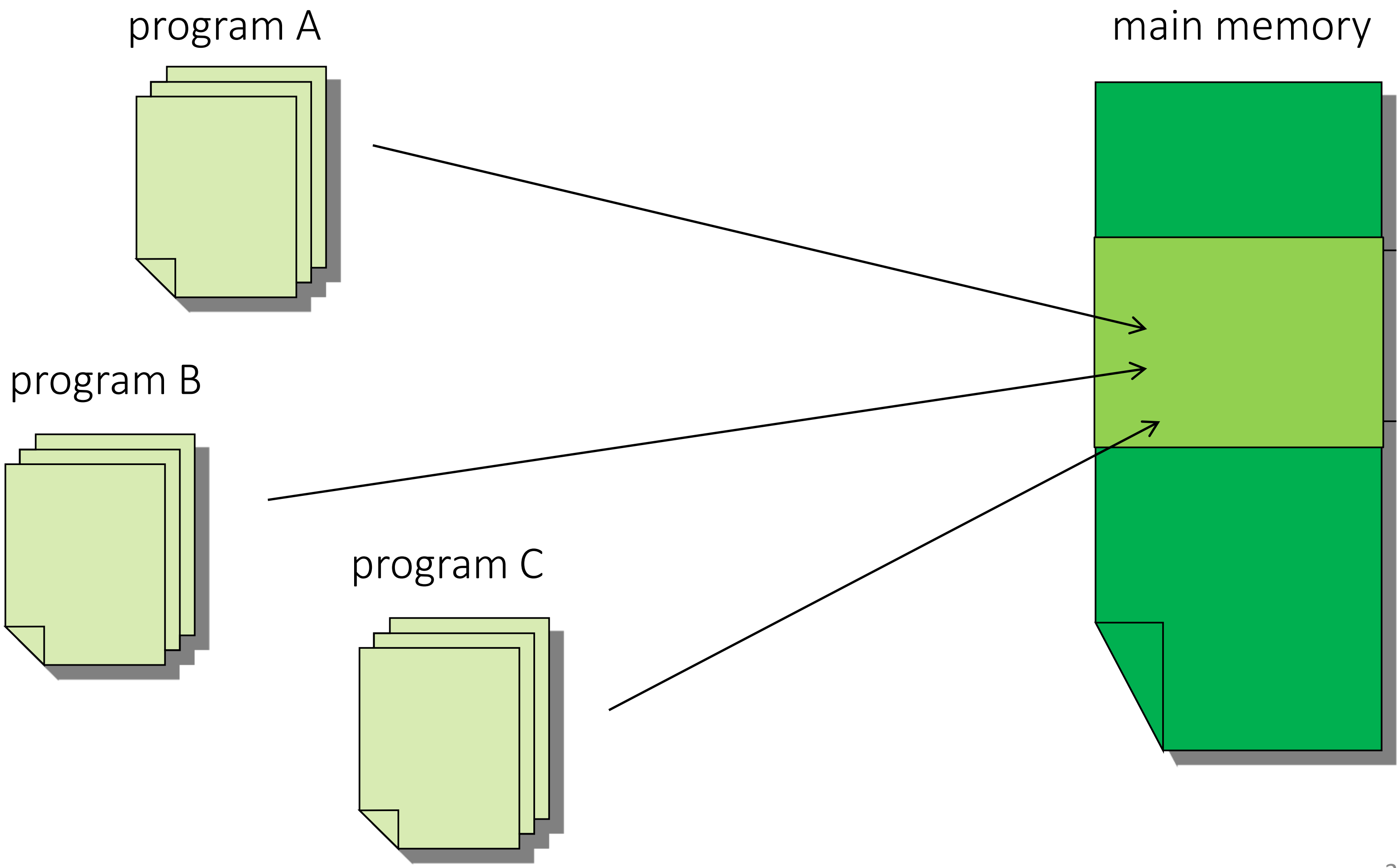
Virtual memory (3)

- **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 (4)



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	$\cdots$	13	12	11	10	$\cdots$	1	0
1	0	$\cdots$	1	1	0	0	$\cdots$	1	1

Table 2.2: Virtual Address Divided into Virtual Page Number and Byte Offset

Translation-lookaside buffer (TLB)

- 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.

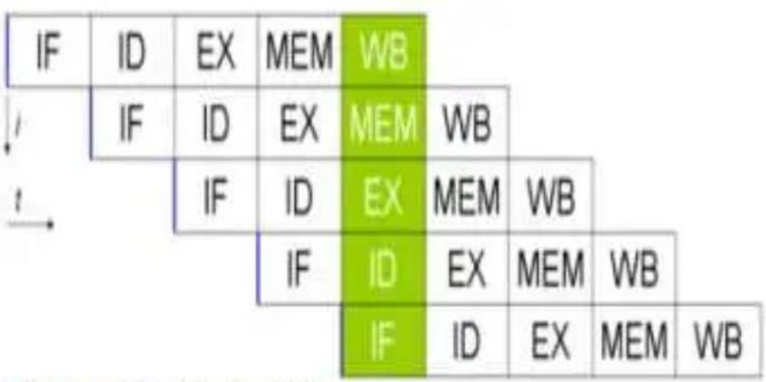
- **Page fault** – attempting to access a valid physical address for a page in the page table but the page is only stored on disk.

Instruction Level Parallelism (ILP)

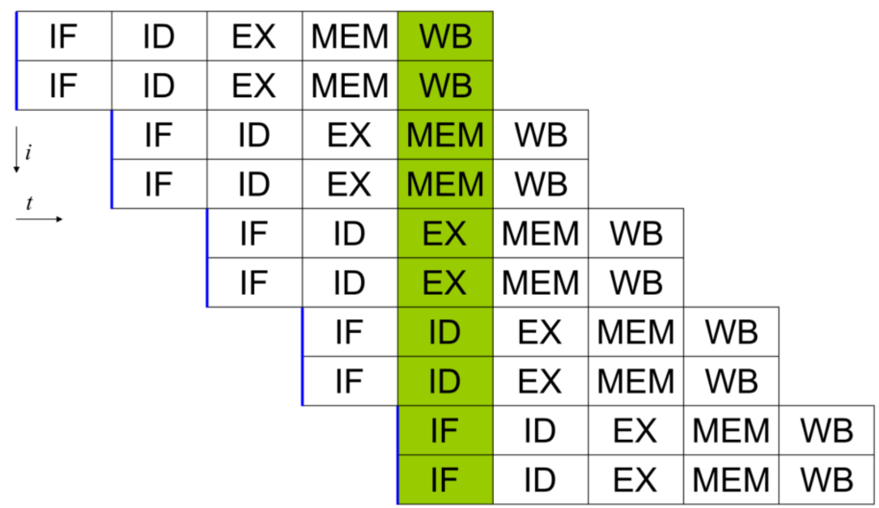
- Attempts to improve processor performance by having multiple processor components or **functional units** simultaneously executing instructions.

Instruction Level Parallelism (2)

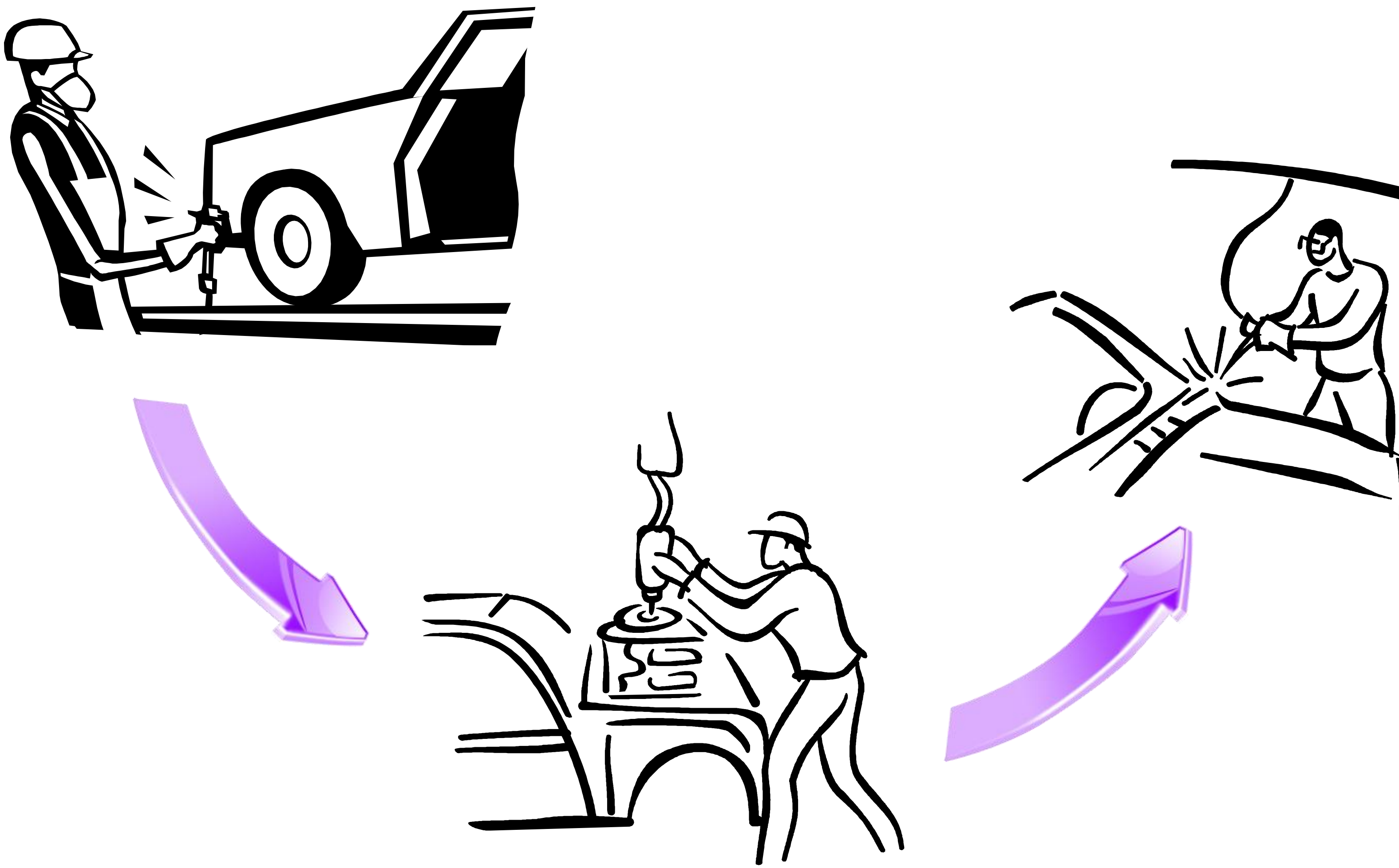
- **Pipelining** - functional units are arranged in stages.
- **Multiple issue** - multiple instructions can be simultaneously initiated.



RISC five-stage pipeline



Pipelining



Pipelining example (1)

Time	Operation	Operand 1	Operand 2	Result
1	Fetch operands	9.87×10^4	6.54×10^3	
2	Compare exponents	9.87×10^4	6.54×10^3	
3	Shift one operand	9.87×10^4	0.654×10^4	
4	Add	9.87×10^4	0.654×10^4	10.524×10^4
5	Normalize result	9.87×10^4	0.654×10^4	1.0524×10^5
6	Round result	9.87×10^4	0.654×10^4	1.05×10^5
7	Store result	9.87×10^4	0.654×10^4	1.05×10^5

Add the floating point numbers
9.87×10^4 and 6.54×10^3

Pipelining example (2)

```
float x[1000], y[1000], z[1000];
. . .
for (i = 0; i < 1000; i++)
    z[i] = x[i] + y[i];
```

- Assume each operation takes 1 nanosecond (10^{-9} s).
- This *for* loop takes about 7000 nanoseconds.

Pipelining (3)

- Divide the floating point adder into 7 separate pieces of hardware or functional units.
- First unit fetches two operands, second unit compares exponents, etc.
- Output of one functional unit is input to the next.

Pipelining (4)

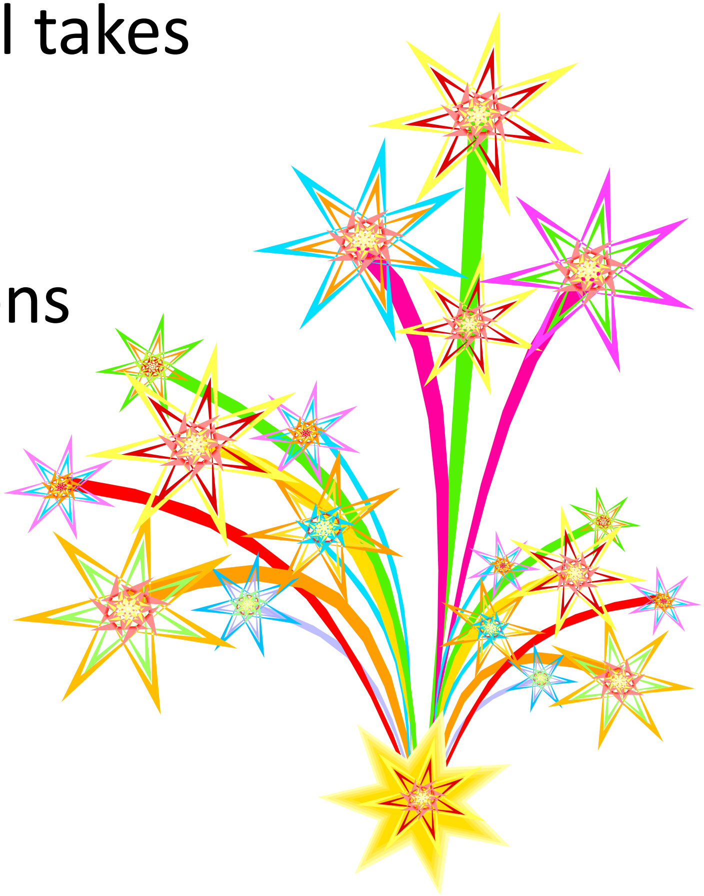
Time	Fetch	Compare	Shift	Add	Normalize	Round	Store
0	0						
1	1	0					
2	2	1	0				
3	3	2	1	0			
4	4	3	2	1	0		
5	5	4	3	2	1	0	
6	6	5	4	3	2	1	0
⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮
999	999	998	997	996	995	994	993
1000		999	998	997	996	995	994
1001			999	998	997	996	995
1002				999	998	997	996
1003					999	998	997
1004						999	998
1005							999

Table 2.3: Pipelined Addition.

Numbers in the table are subscripts of operands/results.

Pipelining (5)

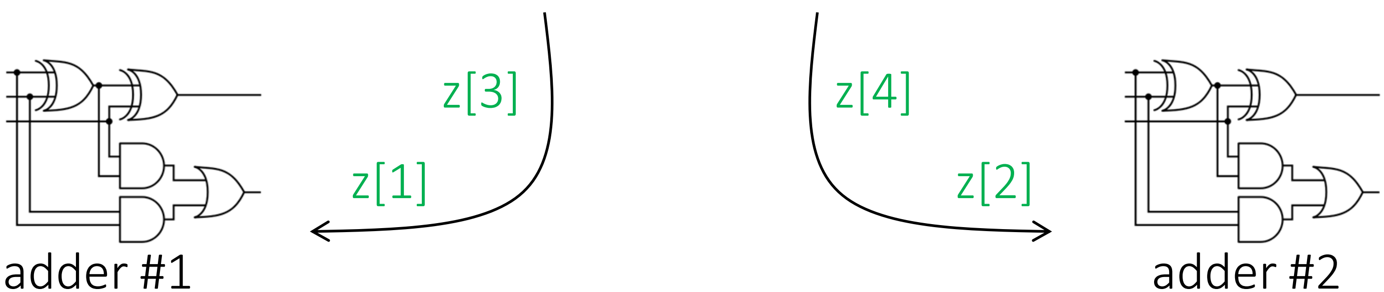
- One floating point addition still takes 7 nanoseconds.
- But 1000 floating point additions now takes 1006 nanoseconds!



Multiple Issue (1)

- Multiple issue processors **replicate** functional units and try to simultaneously execute different instructions in a program.

```
for (i = 0; i < 1000; i++)
  z[i] = x[i] + y[i];
```



Multiple Issue (2)

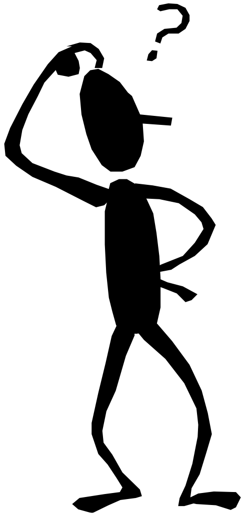
- **static** multiple issue - functional units are scheduled at compile time.

- **dynamic** multiple issue – functional units are scheduled at run-time.

superscalar

Speculation (1)

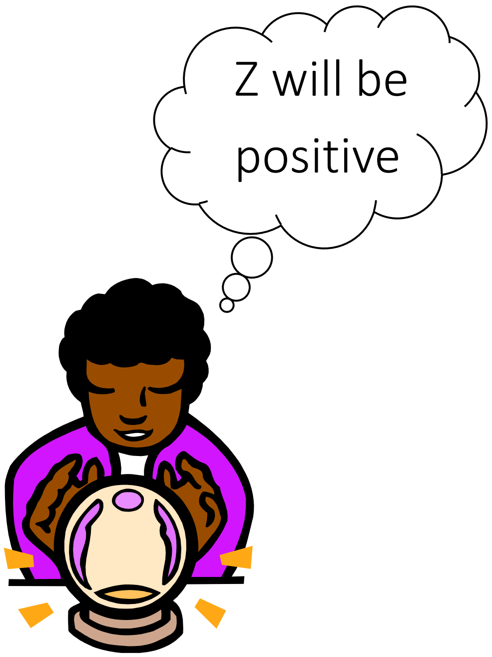
- In order to make use of multiple issue, the system must find instructions that can be executed simultaneously.



- In speculation, the compiler or the processor makes a **guess** about an instruction, and then executes the instruction on the basis of the guess.

Speculation (2)

```
z = x + y ;
if ( z > 0)
    w = x ;
else
    w = y ;
```



If the system speculates incorrectly,

it must go back and recalculate w = y.

Hardware multithreading (1)

- There aren't always good opportunities for simultaneous execution of different threads.

- Hardware multithreading provides a means for systems to continue doing useful work when the task being currently executed has stalled.
  - Ex., the current task has to wait for data to be loaded from memory.

Hardware multithreading (2)

- **Fine-grained** - the processor switches between threads after each instruction, skipping threads that are stalled.

  - **Pros**: potential to avoid wasted machine time due to stalls.

  - **Cons**: a thread that's ready to execute a long sequence of instructions may have to wait to execute every instruction.

Hardware multithreading (3)

- **Coarse-grained** - only switches threads that are stalled waiting for a time-consuming operation to complete.

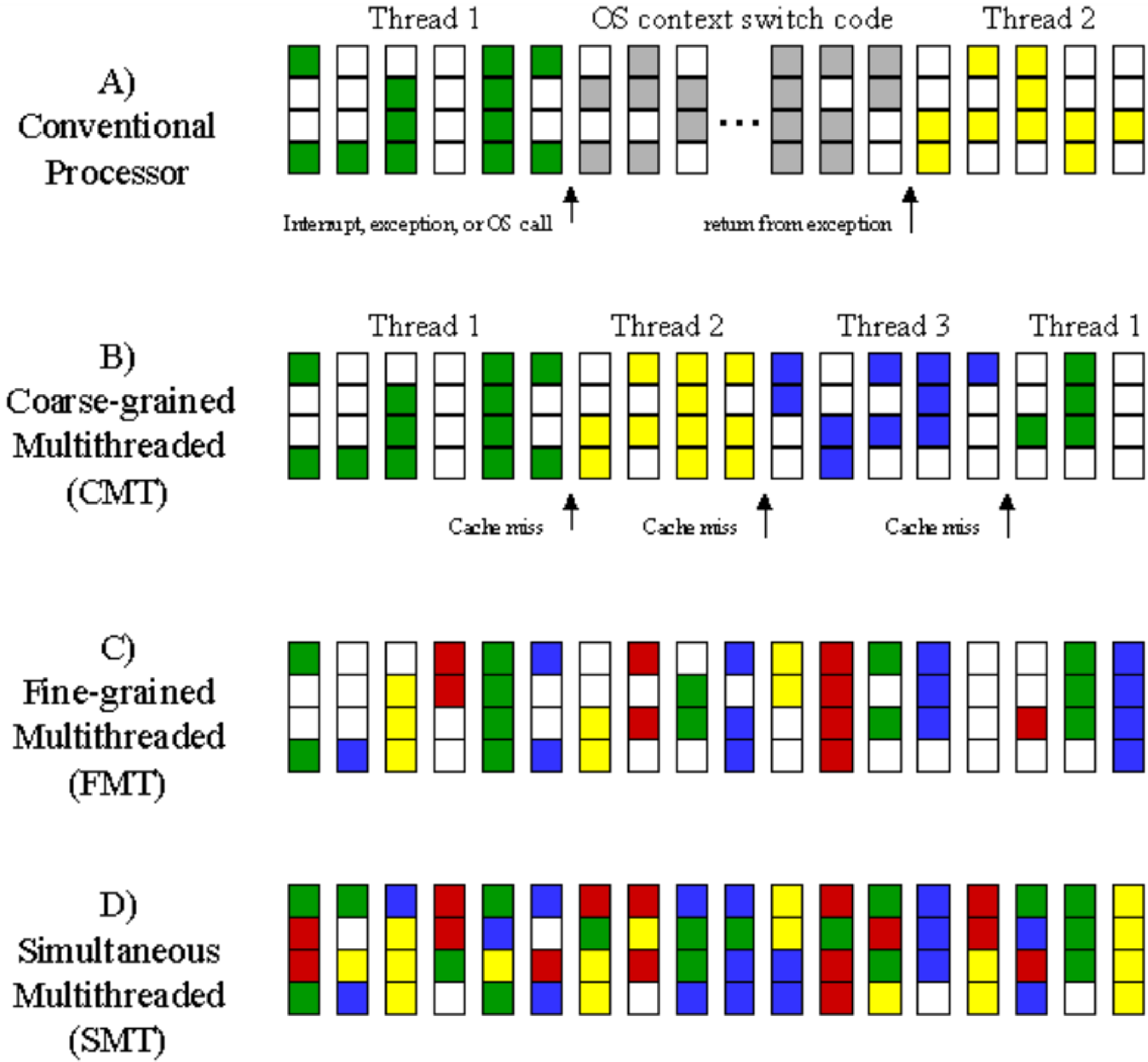
  - **Pros:** switching threads doesn't need to be nearly instantaneous.

  - **Cons:** the processor can be idled on shorter stalls, and thread switching will also cause delays.

Hardware multithreading (3)

- **Simultaneous multithreading (SMT)** - a variation on fine-grained multithreading.

- Allows multiple threads to make use of the multiple functional units.


A) Conventional Processor
Thread 1
OS context switch code
Thread 2
Interrupt, exception, or OS call
return from exception
B) Coarse-grained Multithreaded (CMT)
Thread 1
Thread 2
Thread 3
Thread 1
Cache miss
Cache miss
Cache miss
C) Fine-grained Multithreaded (FMT)
D) Simultaneous Multithreaded (SMT)

Program, Process, Thread

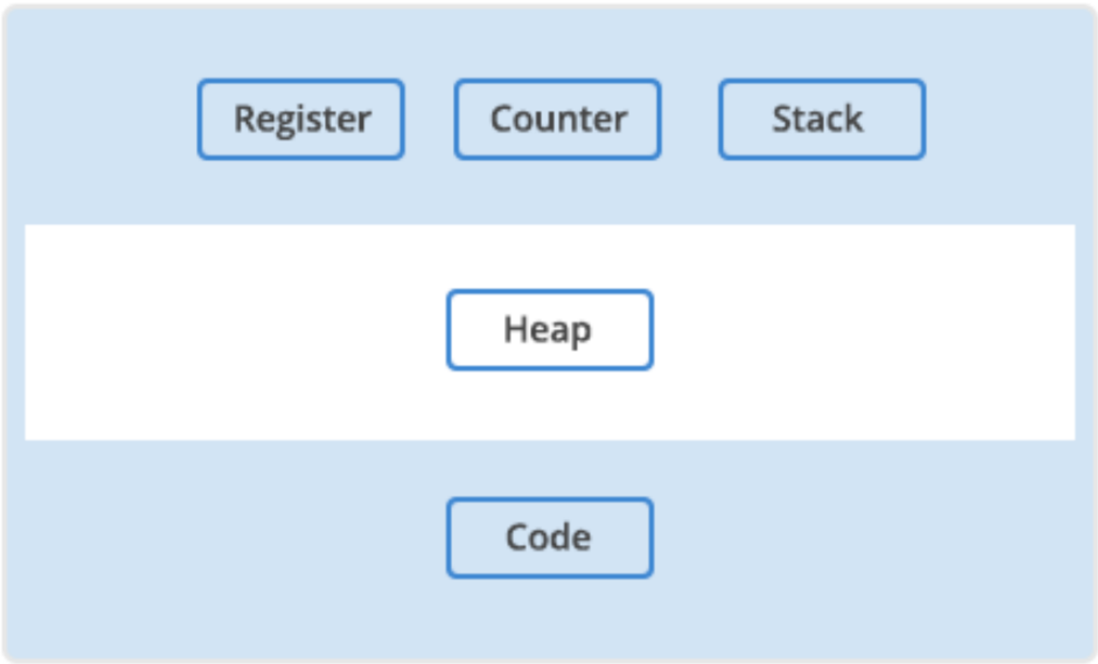
Program vs. Process

- A *program* is the code that is stored on your computer that is intended to fulfill a certain task.

- There can be multiple instances of a single program, and each instance of that running program is a *process*.
  - Each process has a separate memory address space, which means that a process runs independently and is isolated from other processes.

https://www.backblaze.com/blog/whats-the-diff-programs-processes-and-threads/

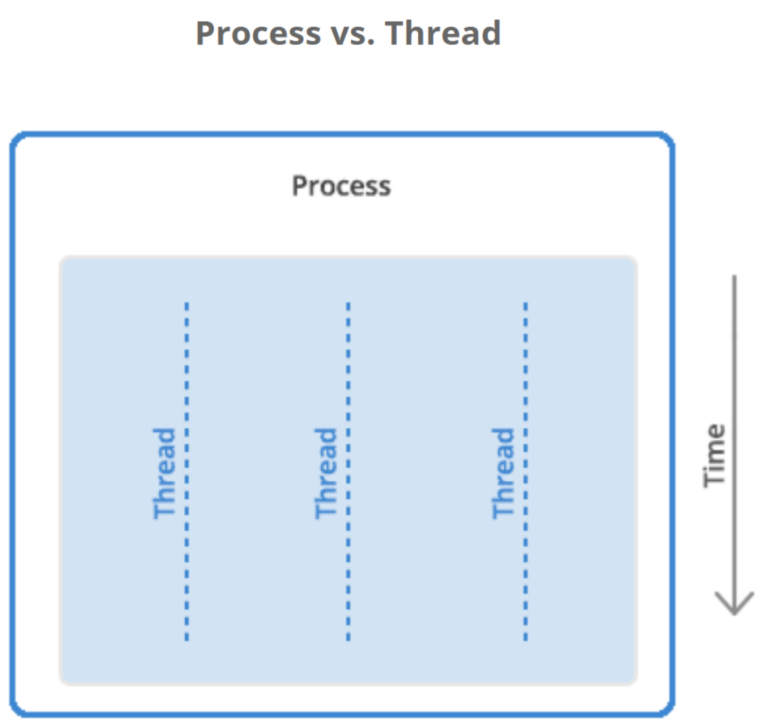
A Computer Process

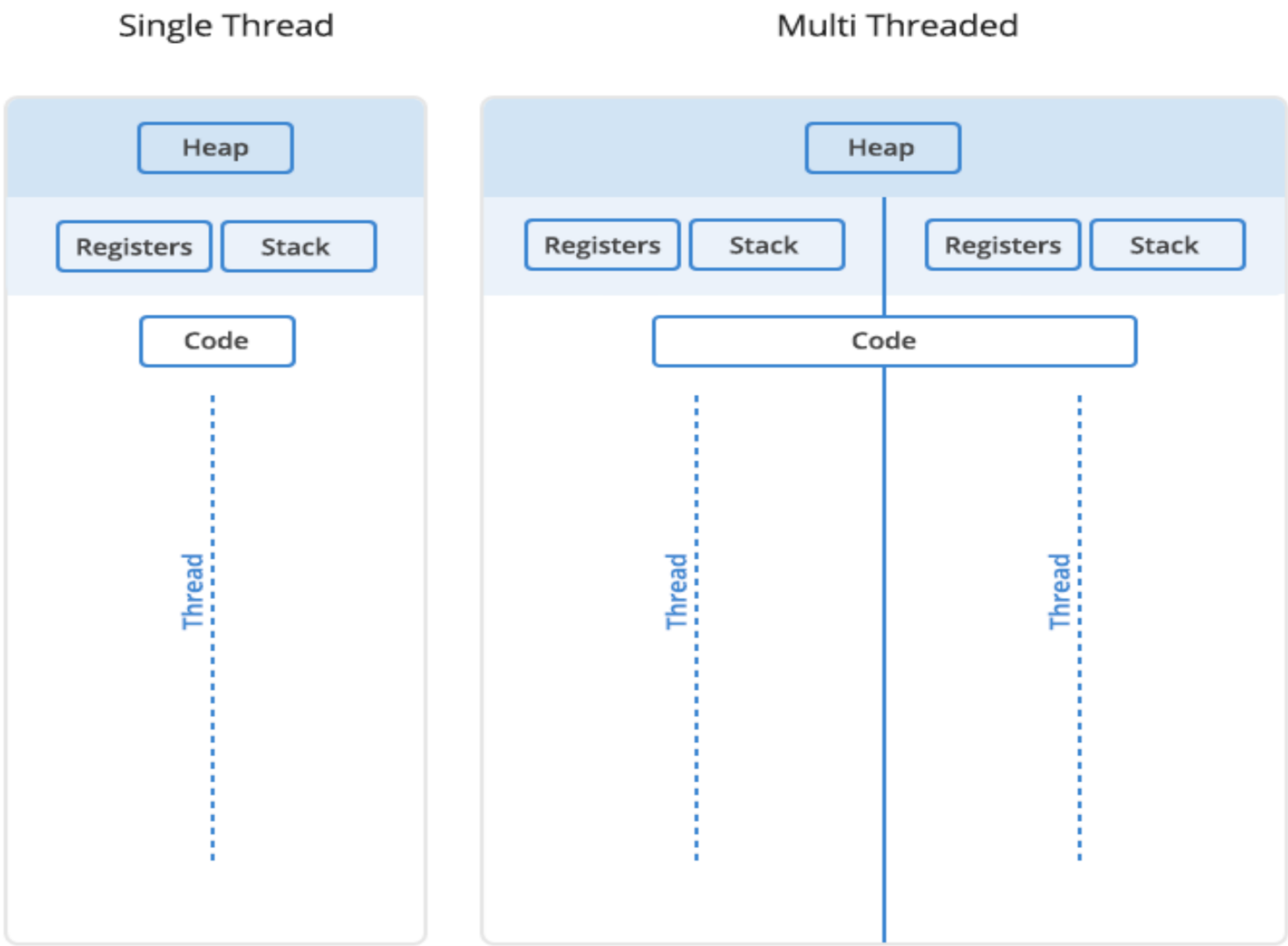


- Each process has a separate memory address space, which means that a process runs independently and is isolated from other processes. It cannot directly access shared data in other processes.
- Switching from one process to another requires some time (relatively) for saving and loading registers, memory maps, and other resources.

Process vs. Thread

- A ***thread*** is the unit of execution within a process.
  - A process can have 1~n threads.





- Each thread will have its own stack, but all the threads in a process will share the heap.
- A problem with one thread in a process will certainly affect other threads and the viability of the process itself.

Things to review

1. The program starts out as a text file of programming code,
2. The program is compiled or interpreted into binary form,
3. The program is loaded into memory,
4. The program becomes one or more running processes.
5. Processes are typically independent of each other,
6. While threads exist as the subset of a process.
7. Threads can communicate with each other more easily than processes can
8. But threads are more vulnerable to problems caused by other threads in the same process.

Processes vs. Threads — Advantages and Disadvantages

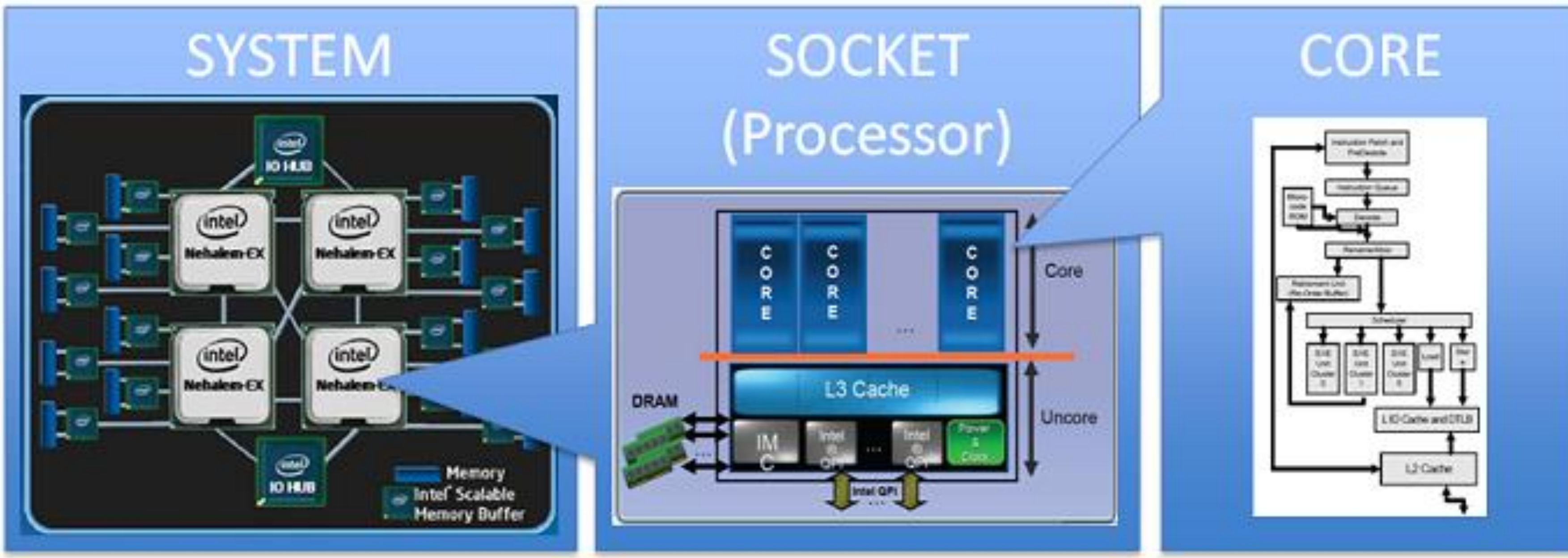
PROCESS	THREAD
Processes are heavyweight operations	Threads are lighter weight operations
Each process has its own memory space	Threads use the memory of the process they belong to
Inter-process communication is slow as processes have different memory addresses	Inter-thread communication can be faster than inter-process communication because threads of the same process share memory with the process they belong to
Context switching between processes is more expensive	Context switching between threads of the same process is less expensive
Processes don’t share memory with other processes	Threads share memory with other threads of the same process

CPU, Core, Processor

CPU vs. Core

- A ***core*** is usually the basic computation unit of the CPU
  - it can run a single program context
  - or multiple ones if it supports hardware threads such as hyperthreading on Intel CPUs
- A ***CPU*** may have one or more cores to perform tasks at a given time.
  - These tasks are usually software processes and threads that the OS schedules.
  - X = number cores * number of hardware threads per core.

CPU vs. Core



- A CPU contains (1-many) cores.
- Each core can execute (1-many) threads depending on hyper-threading technology

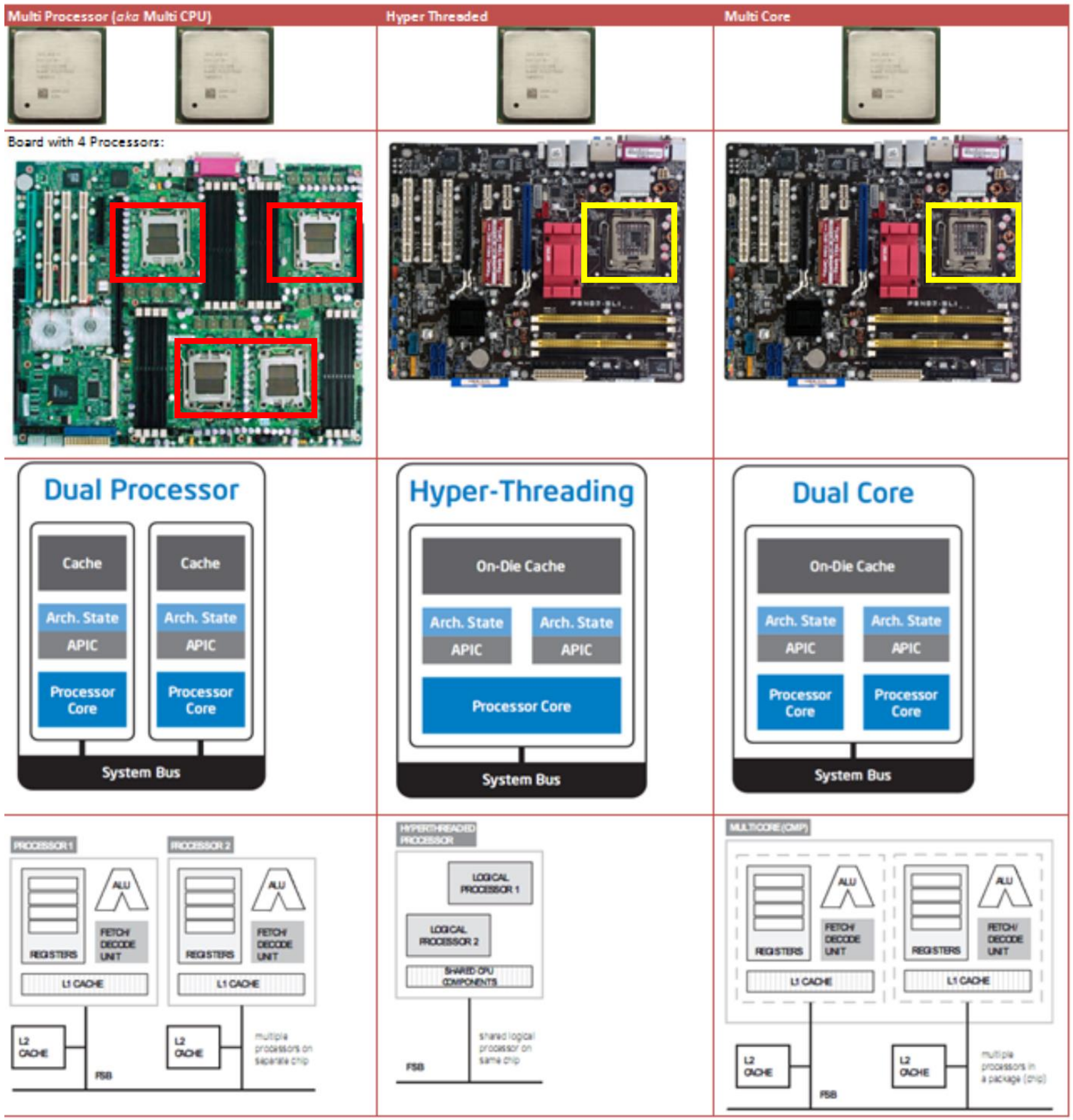
Processor vs. CPU

- A *processor* is a generic term used to describe any sort of CPU, regardless of cores.

- A multiprocessor system contains more than one CPU, allowing them to work in parallel.
  - This is called SMP, or Symmetric Multi-Processing.

- A multi*core* CPU has multiple cores on one CPU.

Multi-processor Single-core CPU Multi-Core CPU



Can a single **process** run in multiple cores?

- Yes, a single process can run multiple threads on different cores.

Can a single thread run in multiple cores?

- There is no such thing as a single thread running on multiple cores simultaneously.
  - Thread migration is possible

- It doesn't mean, however, that instructions from one thread cannot be executed in parallel.

- There are mechanisms called instruction pipelining and out-of-order execution that allow it.

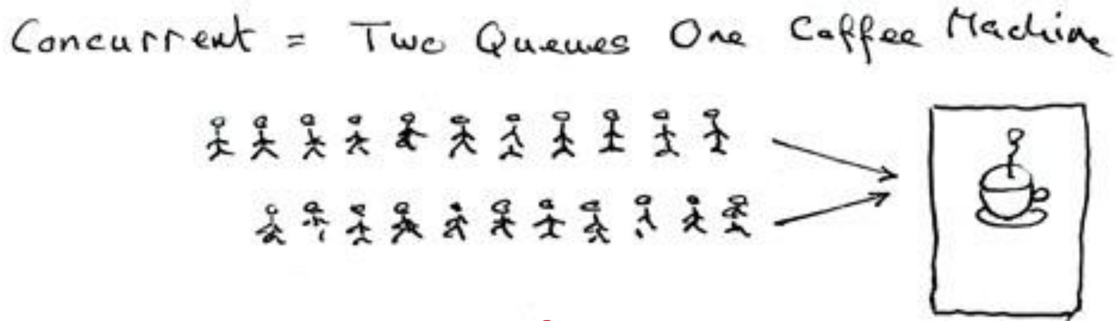
Concurrent vs. Parallel

Concurrent vs. Parallel

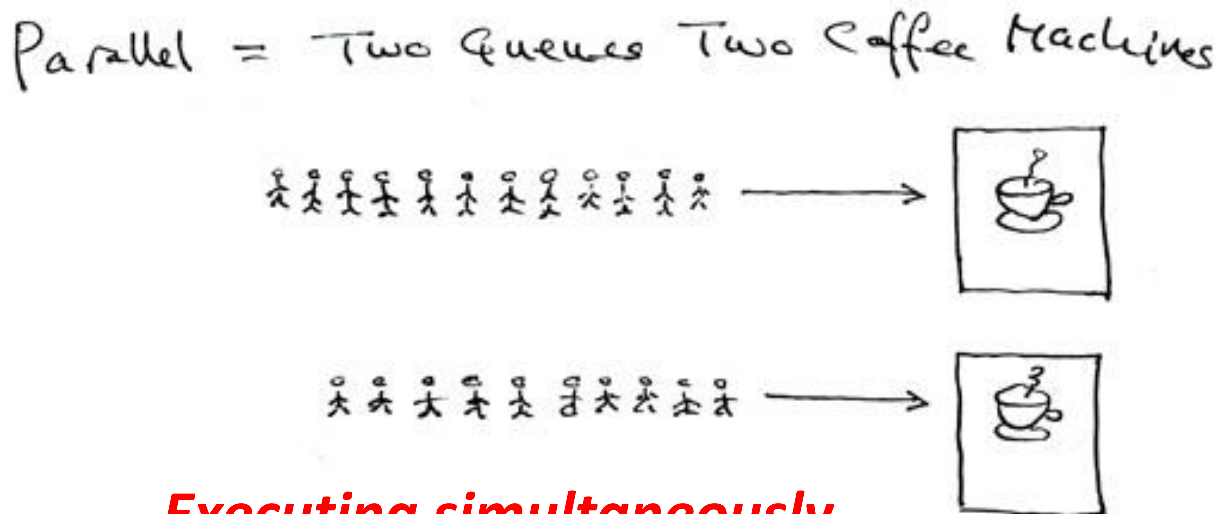
- A system is said to be **concurrent** if it can support two or more actions ***in progress at the same time***.

- A system is said to be **parallel** if it can support two or more actions executing ***simultaneously***.

- **Concurrent** = Two queues and one coffee machine.
- **Parallel** = Two queues and two coffee machines.



in progress at the same time.



Executing simultaneously

© Joe Armstrong 2013

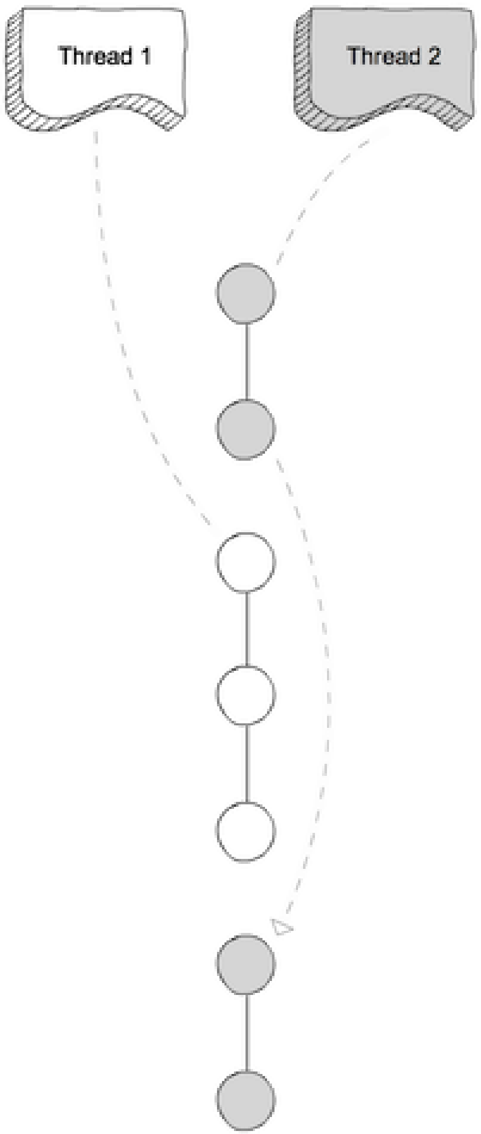
Parallel Computing vs. Concurrent Computing

- In parallel computing, execution occurs at the *same physical instant*
  - parallel computing is impossible on a (one-core) single processor

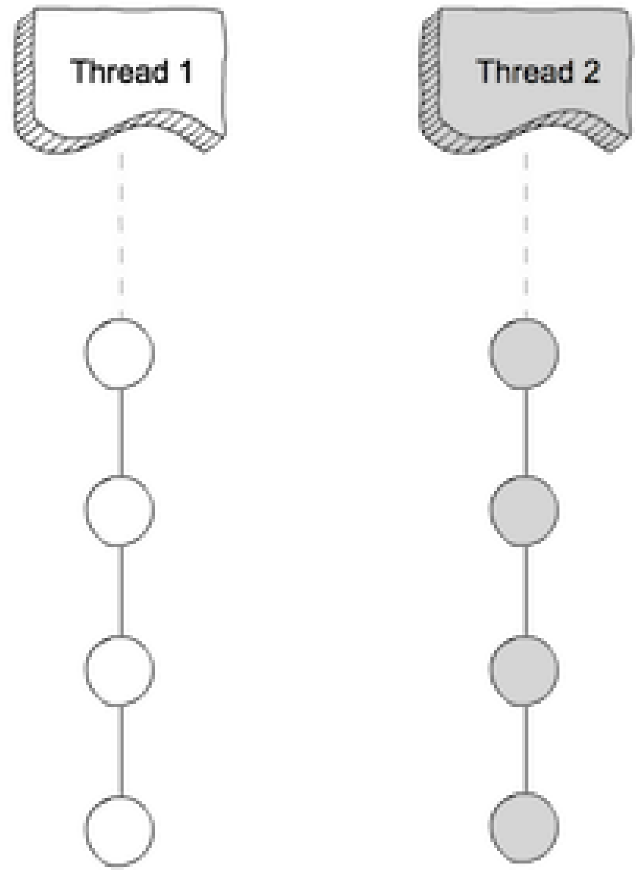
- concurrent computing consists of process lifetimes overlapping, but *execution need not happen at the same instant*
  - concurrent processes can be executed on one core by interleaving the execution steps of each process via time-sharing slices

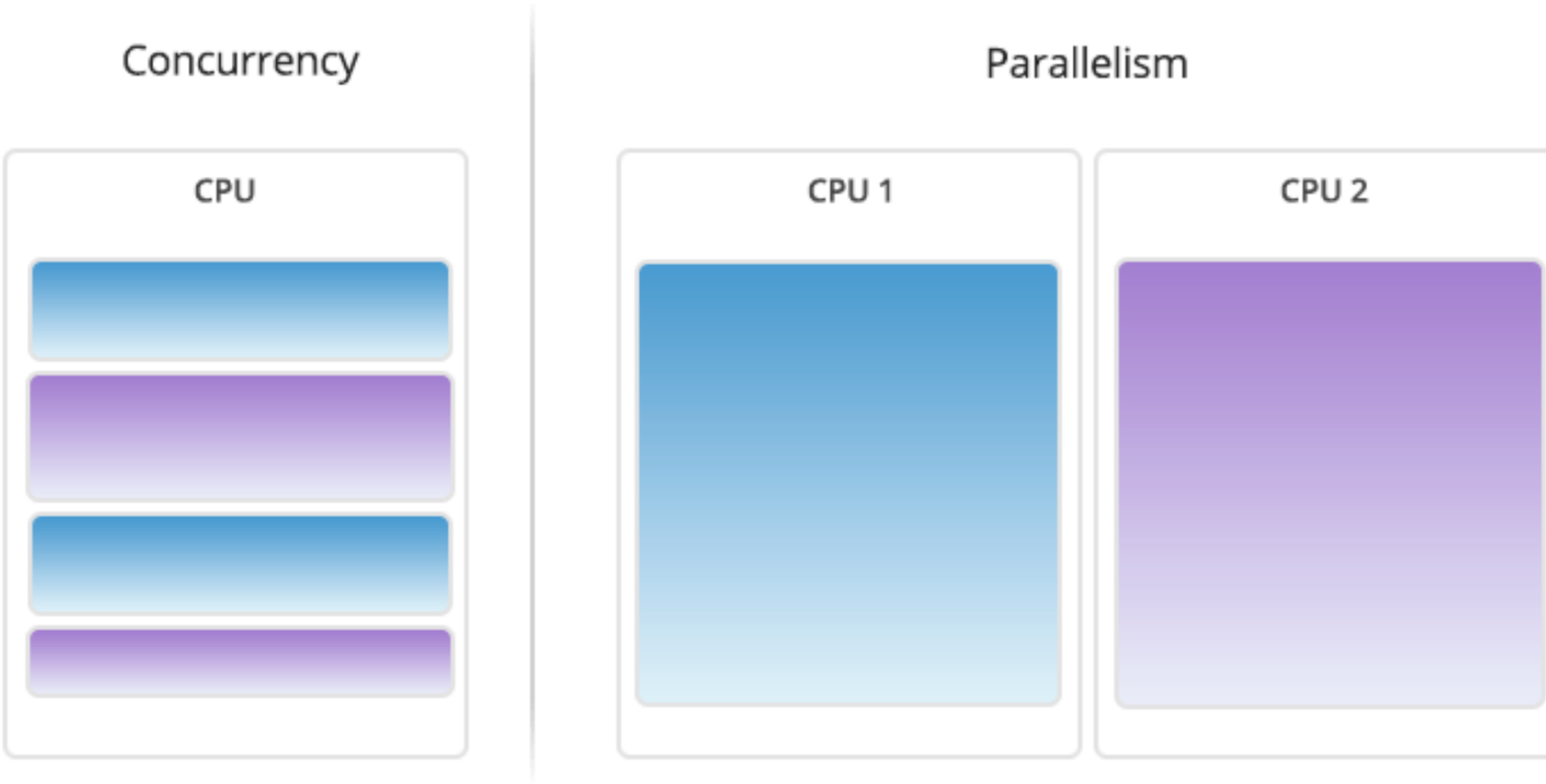
https://en.wikipedia.org/wiki/Parallel_computing

Concurrent Computing



Parallel Computing




Concurrency
Parallelism
CPU
CPU 1
CPU 2