PART 5

THREADS

Dr. Soha S. Zaghloul

LAYOUT

- Threads
- Multithreaded Processes
- Multithreaded Programming
  - Modes of Execution
  - Amdahl's Law
  - Types of Parallelism

INTRODUCTION

A thread is the most basic unit in CPU utilization.

A single-threaded process is a process involving a single thread. *All our previous study was about single-threaded processes.*

A process may run multiple threads. This is known as a multithreaded process. In such case, the process can perform more than one task at a time in a multiprocessor environment.

A process is also called a heavyweight process.
A thread is known to be a lightweight process.

A thread comprises the following information:

- A thread ID
- A program counter
- A register set
- A stack

All threads belonging to the same process share the same:

- Code section
- Data section
- OS resources

In such systems, the PCB is expanded to include information about each thread.

EXAMPLES (1)

When a user runs a word processor, multiple threads may be needed to perform the following:

- Type-in characters
- Run the spell checker

In certain situations, a single application may be required to perform several similar tasks.

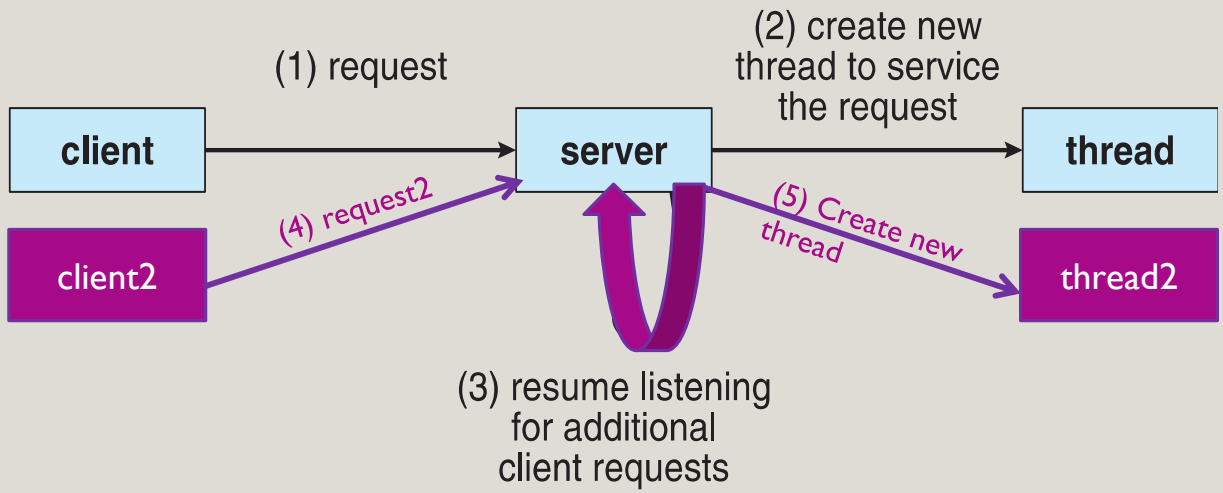
- A web server accepts client requests to access web pages, images, sound, etc… The same task is repeated with each client. Only the URL and client IP differ from one request to another.

We have three alternatives to solve this problem:

- Serve one client at a time:
  - With thousands of clients sending requests, a client will have to wait for a very long time till (s)he is serviced.
- Create a new process with each client's request:
  - Process creation is time consuming and resources intensive.
- Create a new thread with each client's request:
  - This is multithreading. This is considered the most efficient solution because:
    - Threads (lightweight) creation is too much easier than processes (heavyweight) creation.
    - The new thread performs the same tasks as the existing process. Remember that threads share the code with the creating process.
    - Threads creation incur less overhead than creating processes: it is less time consuming and requires less resources.

EXAMPLES (2)

The following figure illustrates the example of the web server:



1. The server waits for clients' requests. A client sends a request.
2. The server catches the client's request: it creates a new thread to serve this request.
3. The server resumes listening for addition clients' requests while the thread is executing.
4. The server catches the request of client2, and the same process repeats as much as there are clients' requests.

PROCESS DEFINITION (1)

A process is a program in execution. This is also called a job.

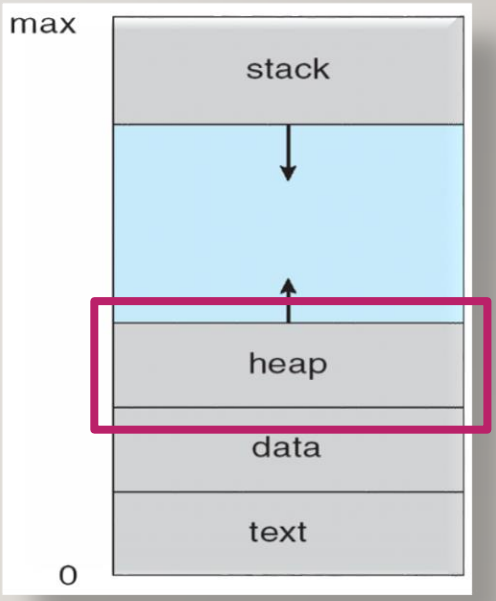
A program is a passive entity; whereas the process is an active one.

In other words, a program becomes a process when it is loaded into memory.

Reminder

A process includes all the information necessary to make it run during its lifetime such as:

- The program code, also known as the text section.
- The program counter that includes the address of the next instruction to be executed.
- The final contents of the processor's registers during the execution of the process.
- The stack: this contains the functions (methods) parameters, return addresses, and global variables.
- The heap: this contains the dynamically allocated memory required during run time.



➔ The heap is shared by all threads of the same process.

MULTITHREADED PROCESSES (1) – SHARED RESOURCES

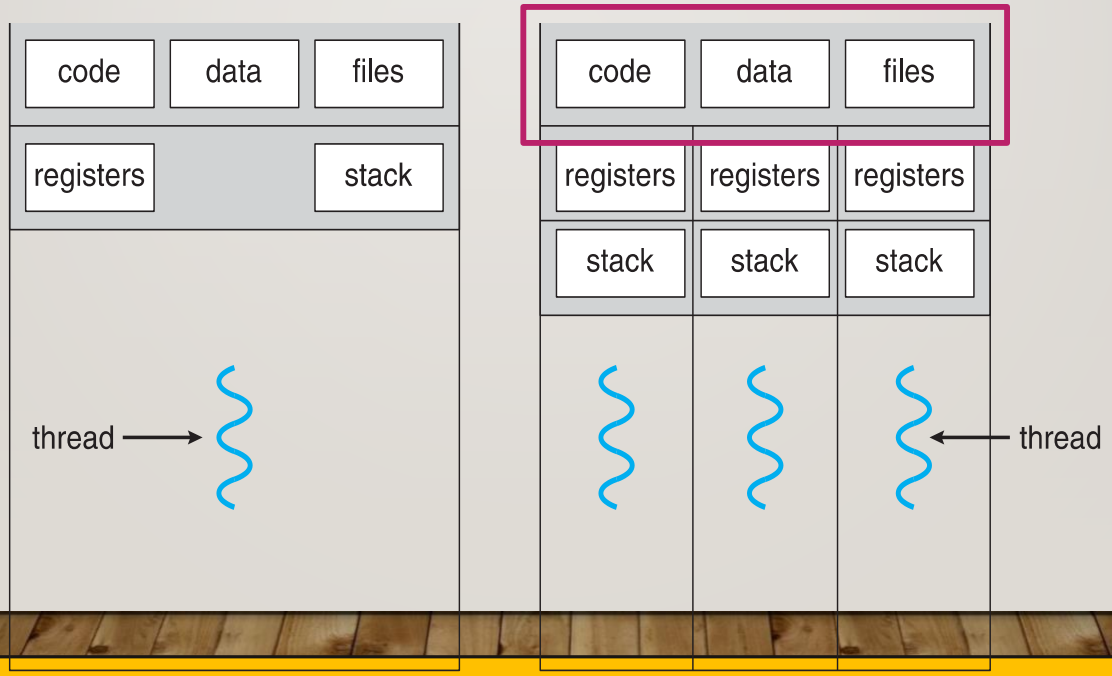
When a process is initially created, it has by default a single thread.

This is known as the main thread, or the initial thread.

Multiple threads may then be created within the same process.

Since files are shared between a process and its threads:

- Thread1 points to a record *n* in a file
- Thread2 (or any other thread within the same process) points to the same record
- Therefore, if thread2 performs a *write* operation, record *n* is modified



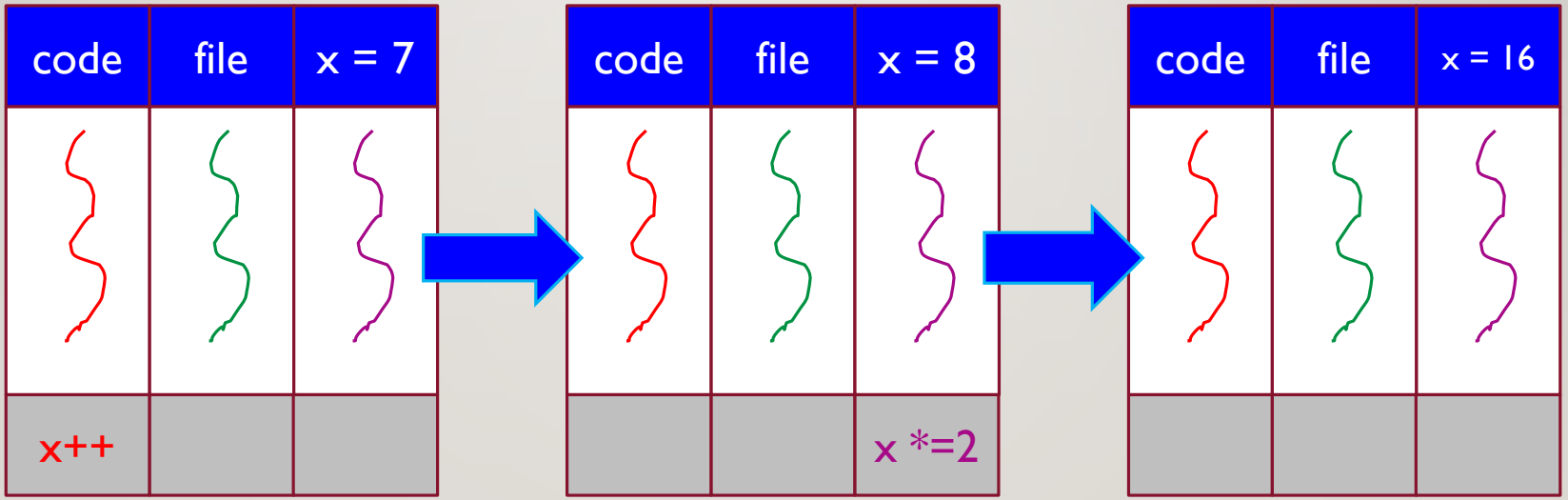
MULTITHREADED PROCESSES (2) – MEMORY ADDRESS SPACE

Assume the variable *x* in the process has value 7.

Thread1 (the red one) added 1 to *x* ➔ all threads see that *x* = 8.

Thread3 (the violet one) doubles *x* ➔ all threads see that *x* = *16*.

In other words, all threads share the same address space with the process.



MULTITHREADED PROCESSES (3) – SHARED ITEMS

The following table illustrates the items shared by all threads in a process (left); and those private to each thread (right):

SHARED BY ALL THREADS IN A PROCESS	PRIVATE TO EACH THREAD
Address space	Program counter
Global variables	Registers
Open files	Stack
Child processes	State
Pending alarms	
Signals and Signal handlers	
Accounting information	

Additional Notes

Threads are easier to create since they don't require separate address spaces.

Threads share the same variables and data structures. Therefore, multithreading requires careful programming so that a single thread modifies a variable/data structure at a time.

Threads use far less resources than processes. This explains why they are called lightweight processes.

Processes are independent of each other. On the other hand, threads are interdependent.

THREADS VS. PROCESSES

The following table compares threads against processes:

PROCESS	THREAD
Heavyweight: resource intensive	Lightweight: it takes less resources as compared to processes.
Context switching: needs to invoke the OS	does not need to invoke the OS
Resources in a multiprocessor environment: Each process executes the same code; but has its own memory and file resources.	All threads share the same code, memory space and file resources.
If one process is blocked, then no other process can execute until the first process is unblocked.	While one thread is blocked and waiting, the other threads in the same task can run.
Resources: Multiple processes without using threads use more resources.	Multiple threaded processes use fewer resources.
Dependence: With multiple processes, each process operates independently of each other.	one thread can read, write or change another thread's data.
Programming: easier to program since processes are independent of each other.	should be made carefully since threads are interdependent.
Address Space: use different address spaces.	threads use the same memory address space.

MULTITHREADED PROGRAMMING (1) – BENEFITS

The advantages of programming with multithreaded processes may be summarized in the following points:

- **Responsiveness**
  - Multithreaded programming allows an application to continue running even if part of it is performing a lengthy operation or is blocked.
  - When the lengthy operation is performed in a separate thread, the application remains interactive with the user.
- **Resource sharing**
  - By default, threads share the memory and resources of the process to which they belong.
- **Economy**
  - Allocating memory and resources for process creation is expensive. In addition context switching between processes is costly.
  - For example, in Solaris, the process creation time is about 30 times a thread creation. Context switching between processes is about five times slower.
- **Scalability**
  - This is the ability of a S/W or H/W system to continue its work efficiently as it grows (scale up) or shrinks (scale down) to meet the user's needs.
  - Multithreading achieves scalability more effectively than multiple processes.

MULTITHREADED PROGRAMMING (2) – MODE OF EXECUTION

Multiple threads run simultaneously (at the same time).

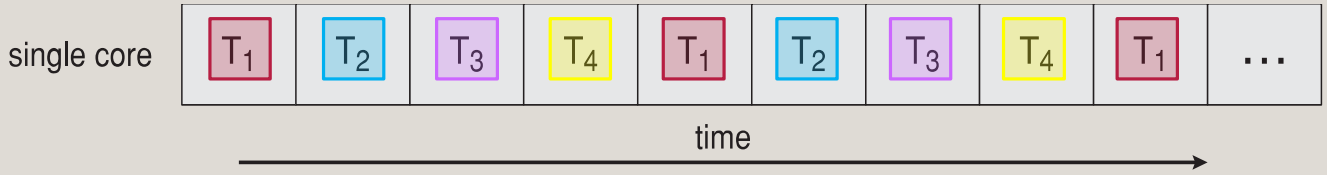
Here we should distinct between two modes of executions with respect to time:

- Concurrent execution
- Parallel execution

MULTITHREADED PROGRAMMING (3) – CONCURRENT EXECUTION

Concurrency means that the execution of threads is "interleaved" on a single-core since it is capable of executing only one thread at a time.

Consider the following figure:



In the above figure, there are four threads running concurrently:

- Thread1 (T1) starts execution;
- Then, T1 stops and T2 executes;
- Then, T2 stops executions and T3 starts execution;
- Finally, T3 stops and T4 executes;
- After that, T4 stops and T1 resumes its execution;
- The same cycle then repeats.

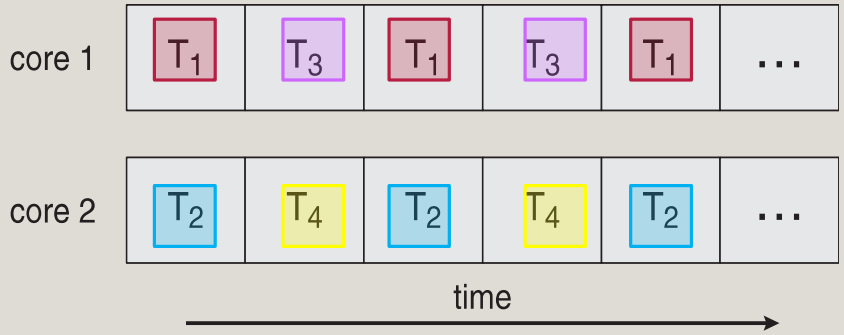
Obviously, no two threads execute at the same time. However, they are interleaving.

The OS switches rapidly between the threads: this gives the illusion that the threads are executing at the same time.

MULTITHREADED PROGRAMMING (4) – PARALLEL EXECUTION

Parallelism means that multiple threads are running "at the same time" on multiple cores or processors.

Consider the following figure:



In the above figure, there are four threads running in parallel:

- Thread1 (T1) starts execution on core1;
  whereas (T2) executes on core2 at the same time.
- Then, (T3) replaces (T1) on core1;
  and (T4) replaces (T2) on core2. Thus, (T3) and (T4) run at the same time.
- After that, (T1) replaces (T3) on core1; and (T2) replaces (T4) on core2.
- The same cycle then repeats.

Obviously:

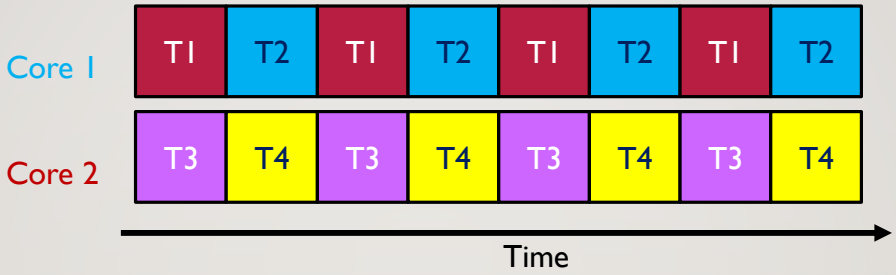
- (T1) and (T2) run in parallel with each other;
- (T3) and (T4) run in parallel with each other;
- (T1) and (T3) run concurrently with each other;
- (T2) and (T4) run concurrently with each other.

MULTITHREADED PROGRAMMING (5) – IMPLEMENTATIONS

Modern Intel CPUs frequently support two threads per core.

This means that the OS may execute up to two threads concurrently per core.

For an Intel dual-core, there are two threads executing concurrently and two threads executing in parallel as follows:



On the other hand, Oracle T4 supports 8 threads per core.

For a dual-core Oracle T4, the execution layout would be as follows:

Multiple threads on the same core allow fast context switching.

As the number of cores increases, as the parallelism increases; and therefore, the execution speed increases.

MULTITHREADED PROGRAMMING (6) – AMDAHL'S LAW

Amdahl's law calculates the performance gained from providing additional cores.

Amdahl's law states that:

$$Speedup \leq \frac{1}{S + \frac{(1 - S)}{N}}$$

Where:

S is the percentage of the application to be executed serially,

N is the number of processing cores

Example:

Assume that we have an application that is designed as 75% parallel and 25% percent (➔ S=0.25). The gained speedup on two cores (N=2) is 1.6 at maximum (Speedup ≤1.6).

Note that if N approaches infinity (ie. we have a very large number of cores), Speedup converges to (1/S).

So, assuming that S=0.4, the maximum speedup = 1/0.4 = 2.5

➔ This is the maximum speedup that can be gained for this application.

MULTITHREADED PROGRAMMING (7) – CHALLENGES

Multithreaded programming is not as popular as sequential programming for the following reasons:

- OS designers must write scheduling algorithms that make use of the multiple processing cores to allow simultaneous execution of instructions.
- In addition, the existing programs should be re-written to be multithreaded so that to make use of the hardware advance.
- Moreover, programmers should be skilled to write multithreaded programming.

Developers of multithreaded programs face many challenges, such as:

- Identifying tasks: find areas that can run independently in simultaneous threads.
- Load balancing: ensure that divided tasks perform equal work of equal value. It wouldn't be wise to allocate a core to execute a trivial work.
- Data splitting: data accessed and used by the tasks should be divided to run on separate cores.
- Data dependency: data accesses by tasks should be examined for dependencies between two or more tasks.
- Testing & debugging: when a program is running simultaneously on multiple cores, many different execution paths are possible. This makes testing & debugging such programs more difficult as compared to single-threaded applications.

MULTITHREADED PROGRAMMING (7) – TYPES OF PARALLELISM

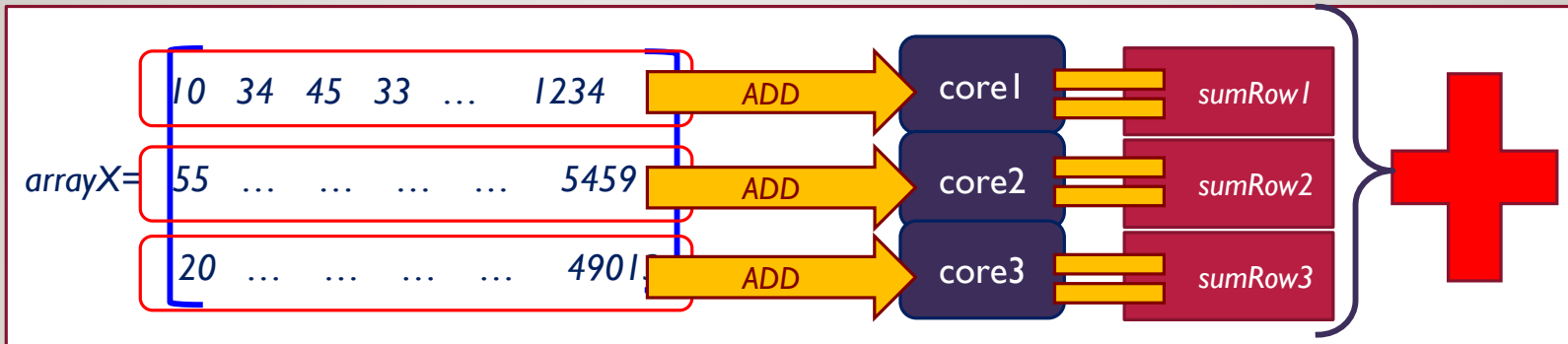
In programs, there are two types of parallelism that can be performed:

- Data Parallelism
- Tasks Parallelism

MULTITHREADED PROGRAMMING (8) – DATA PARALLELISM

Data parallelism focuses on distributing subsets of data across multiple computing cores and performing the same operation on each core on different sets of data.

Consider the example of summing the elements of an array of size *MxN*



In the above example, the elements of each row *i* are summed in parallel to give $sumRow_i$. Then the program adds all *sumRow* to give the sum of all elements in the array.

Compare the above technique with the traditional one:

```
sum = 0;
for (i = 0; i < rows; i++)
  for (j = 0; j < cols; j++)
    sum += arrayX[i][j];
```

How much time is needed in both algorithms?

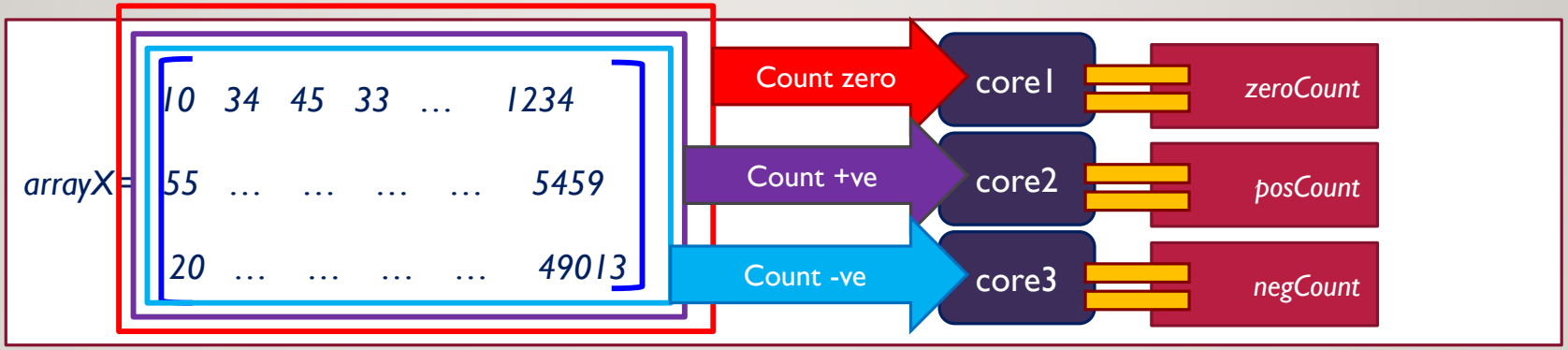
MULTITHREADED PROGRAMMING (9) – TASKS PARALLELISM

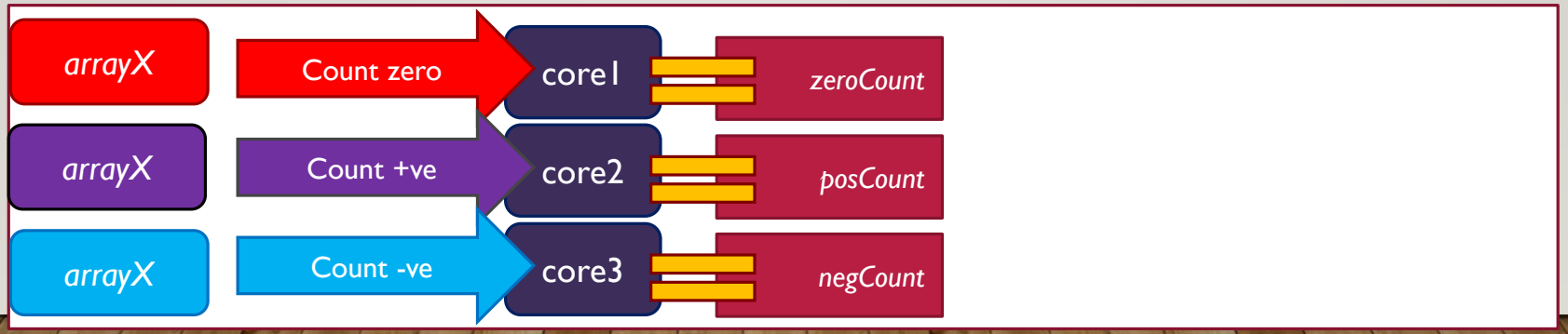
Tasks parallelism involves distributing tasks – rather than data – across multiple cores.

Each core will be performing a separate task.

Each task may be working on the same set of data or on different data sets.

Consider the example of counting the number of zeroes, of negative numbers and that of positive numbers in the same array.





In practice, few applications strictly follow either data or tasks parallelism. In most instances, applications use a hybrid of these two strategies.