Advanced Development Techniques

Parallel programming

Sipos Miklós

Óbudai Egyetem Neumann János Informatikai Kar
Szoftvertervezés és -fejlesztés Intézet
2021

Contents

Fundamentals of parallel programming

Why?

Why do we need parallel programming?

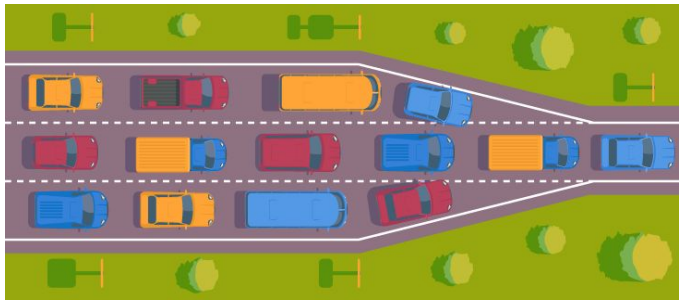
We can have multiple aims:

- decreasing the execution time (or increasing the performance)
- executing only one job within the shortest time
- finishing as much work as possible within one unit of time

➜ **In every case, the final aim is to increase the execution (time)!**

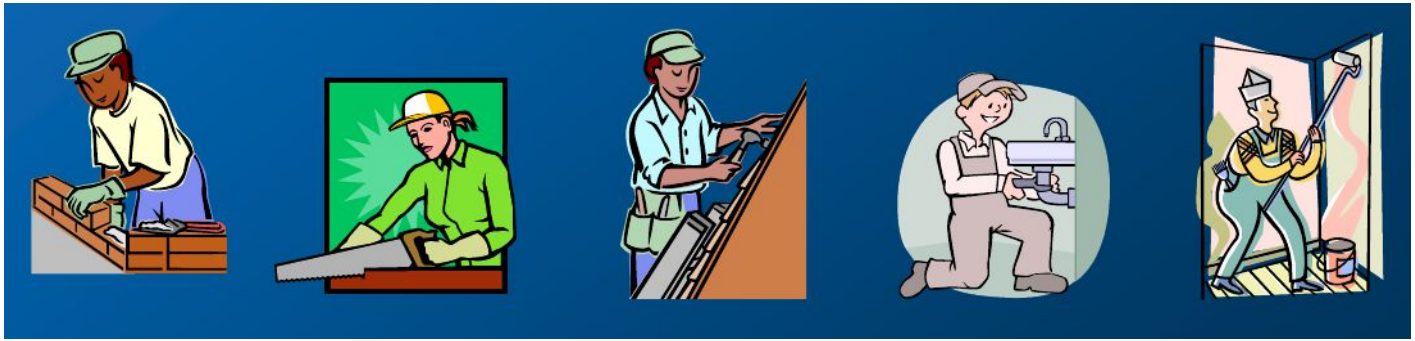
Parallel execution fundamentals

- Computers with The Von Neumann architecture can execute programs and instruction **only** in a sequential way ➔ no parallel option?!
- To execute multiple programs simultaneously
  - More than one CPUs / cores
  - Hyperthreading
  - Time sharing
- If there is more concurrently running/executing instructions, than the execution units then it will be a jam.
- Separation of “concurrently” executing programs
  - Process-level
  - Thread-level



Example 1.

Building a house.

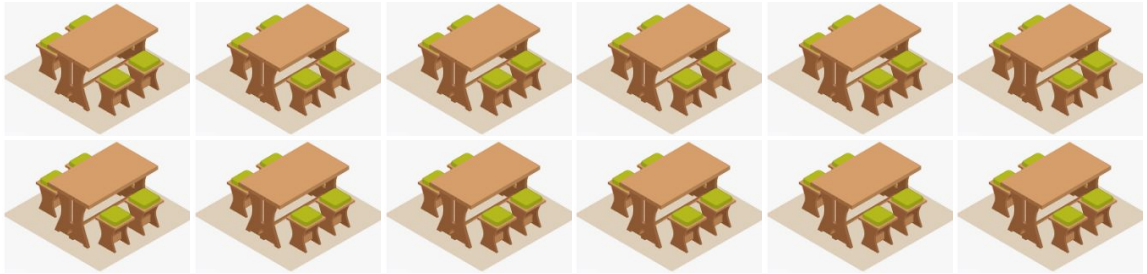


Example 2.

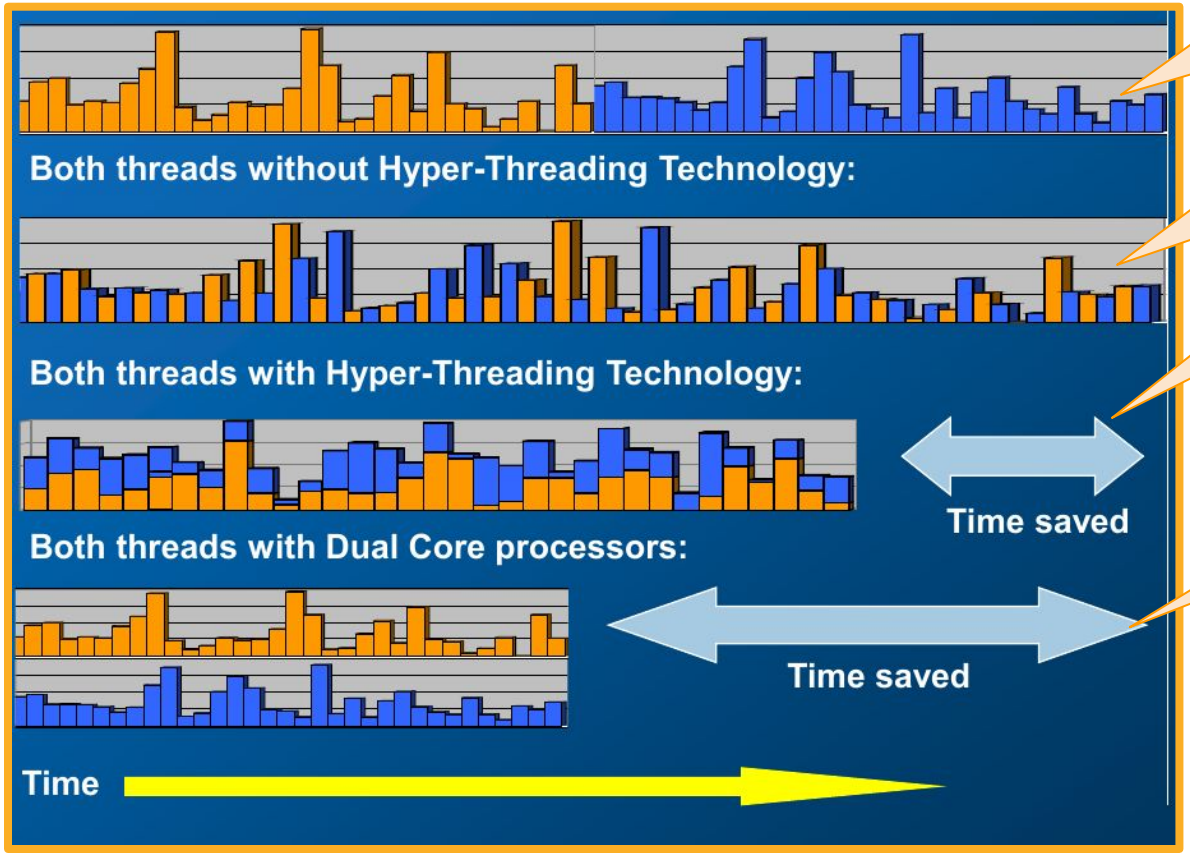
Laying the table for ceremony.

A.) one task in the shortest period of time
laying 1 piece of table ➔ 1 person for everything (dishes, cutlery, glasses, etc.)

B.) as much task as possible within 1 unit of time
laying 100 pieces of table ➔ 1 person per table (sum 100)



Example 3.



"sequential execution"

"time sharing"

"time sharing v2"

"parallel execution"

Hyperthreading (outlook)

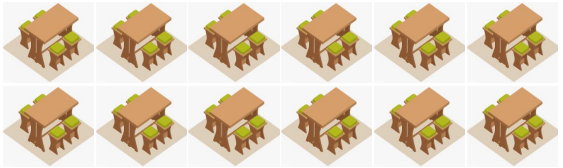
Hyper-threading (officially called Hyper-Threading Technology or HT Technology and abbreviated as HTT or HT) is Intel's proprietary simultaneous multithreading (SMT) implementation used to improve parallelization of computations (doing multiple tasks at once) performed on x86 microprocessors. It was introduced on Xeon server processors in February 2002 and on Pentium 4 desktop processors in November 2002.[4] Since then, Intel has included this technology in Itanium, Atom, and Core 'i' Series CPUs, among others. [5]

For each processor core that is physically present, the operating system addresses two virtual (logical) cores and shares the workload between them when possible. The main function of hyper-threading is to increase the number of independent instructions in the pipeline; it takes advantage of superscalar architecture, in which multiple instructions operate on separate data in parallel. With HTT, one physical core appears as two processors to the operating system, allowing concurrent scheduling of two processes per core. In addition, two or more processes can use the same resources: If resources for one process are not available, then another process can continue if its resources are available.

In addition to requiring simultaneous multithreading support in the operating system, hyper-threading can be properly utilized only with an operating system specifically optimized for it.[6] https://en.wikipedia.org/wiki/Hyper-threading

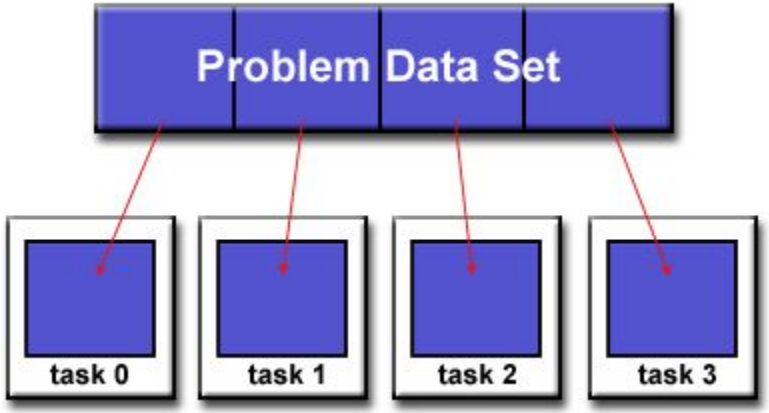
Data decomposition

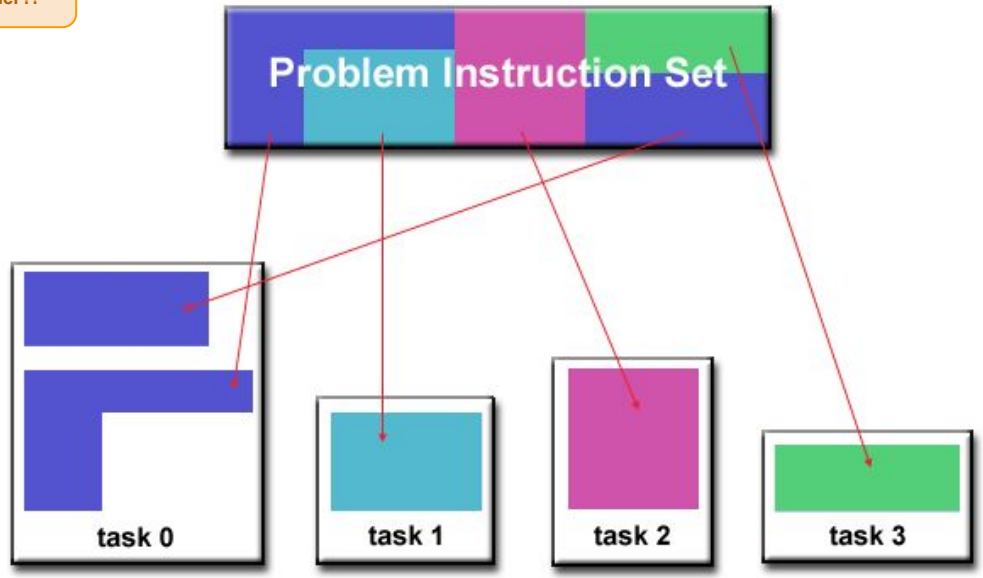
Data or problem decomposition



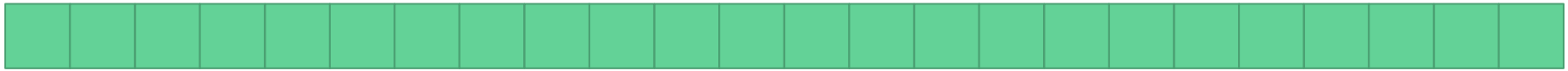
- Decomposition of the original data / problem
  - equal parts?
  - unequal parts?
  - by what logic?
  - to how many pieces?
    - problems eg. granularity

divide and conquer?!

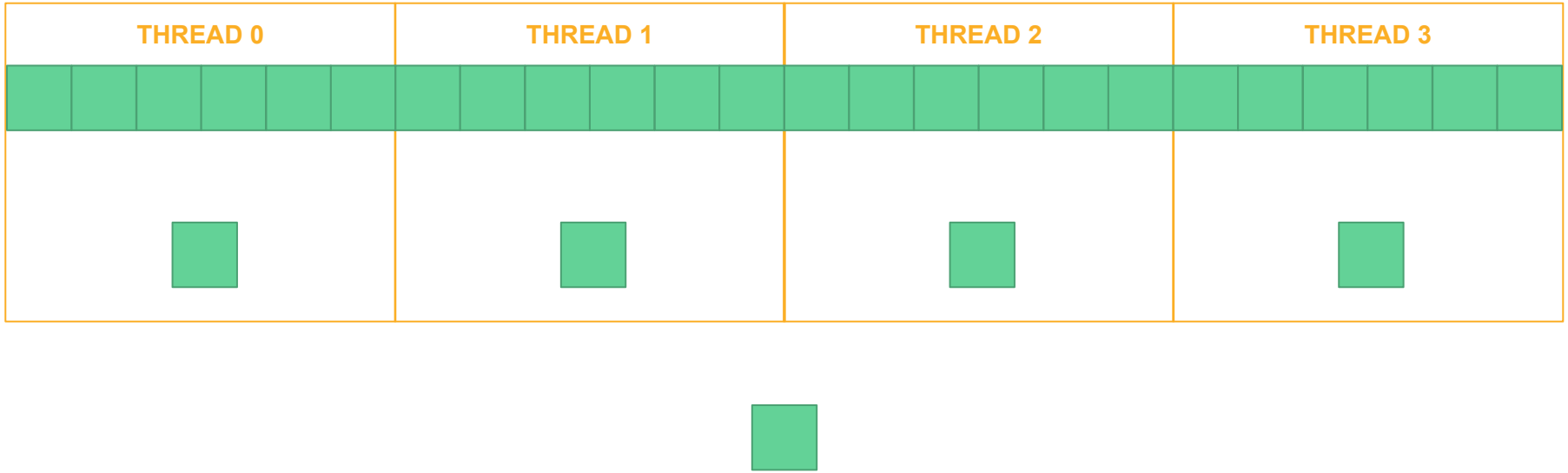




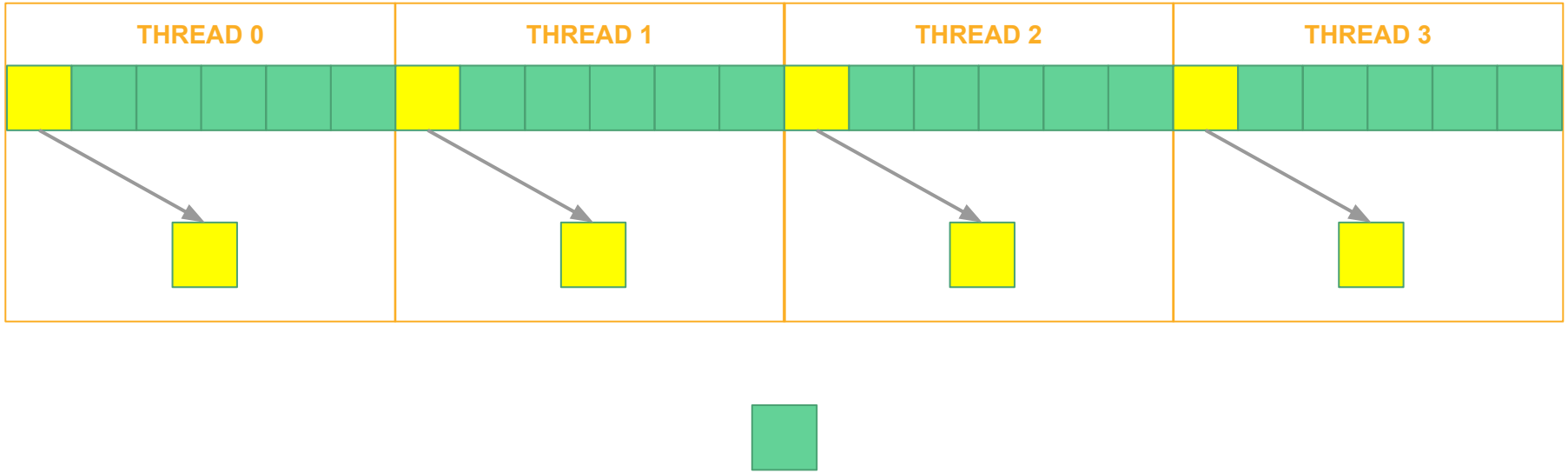
Finding the largest item in an array



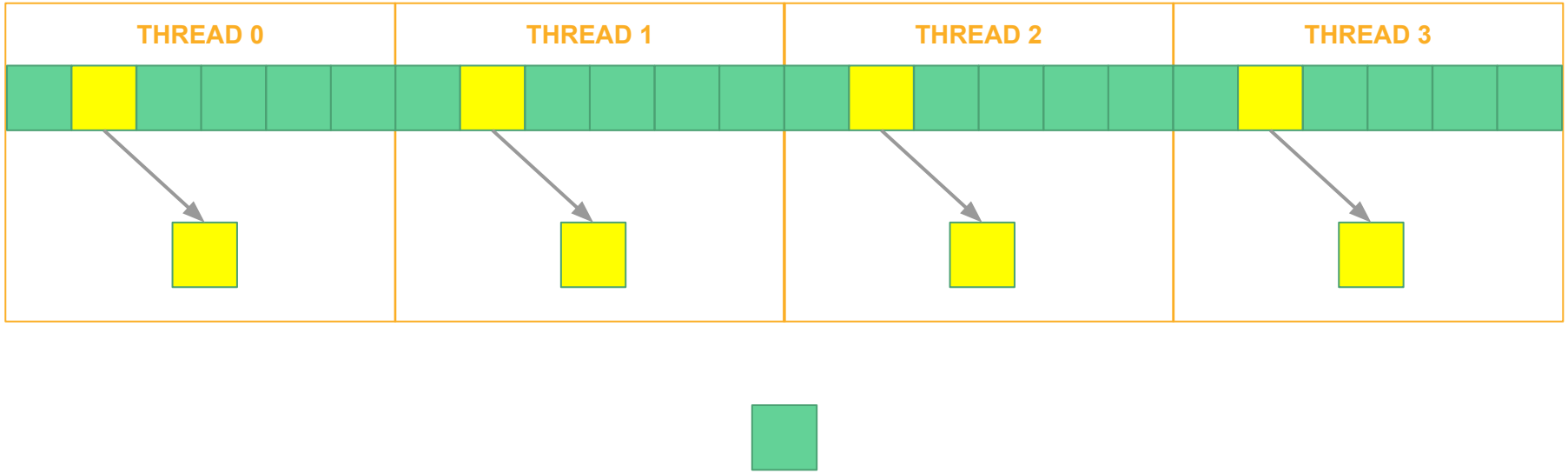
Finding the largest item in an array



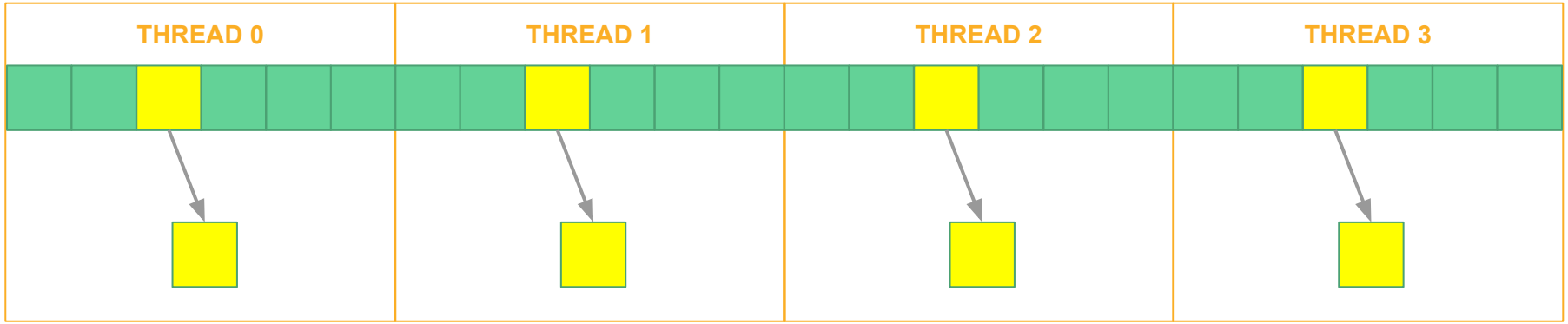
Finding the largest item in an array



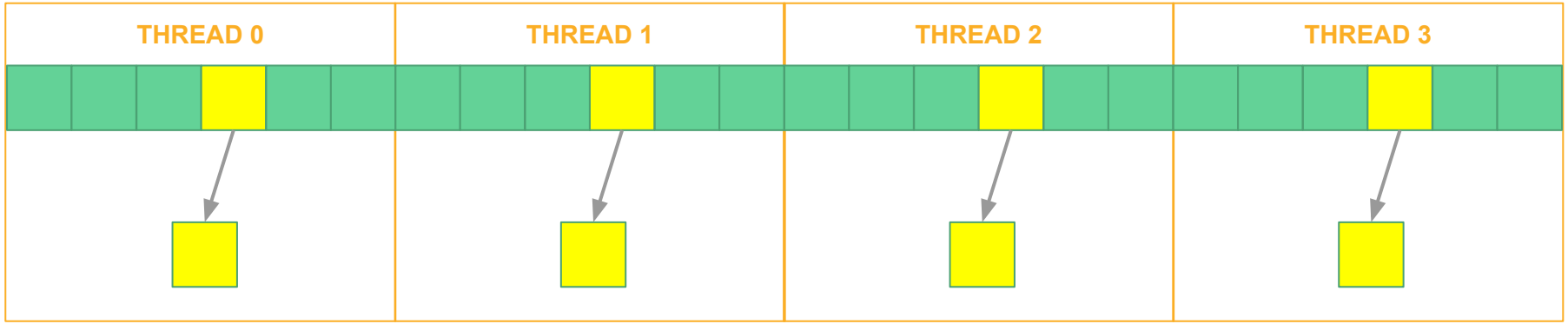
Finding the largest item in an array



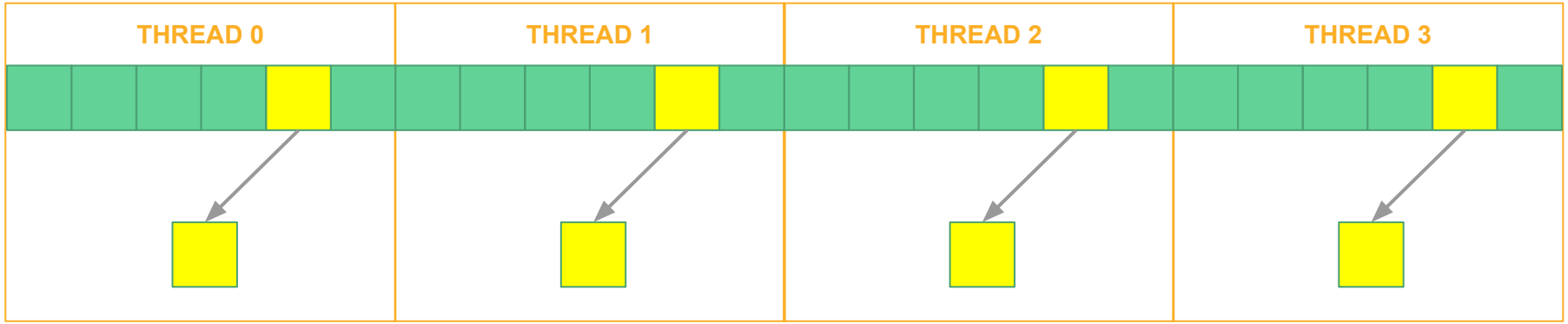
Finding the largest item in an array



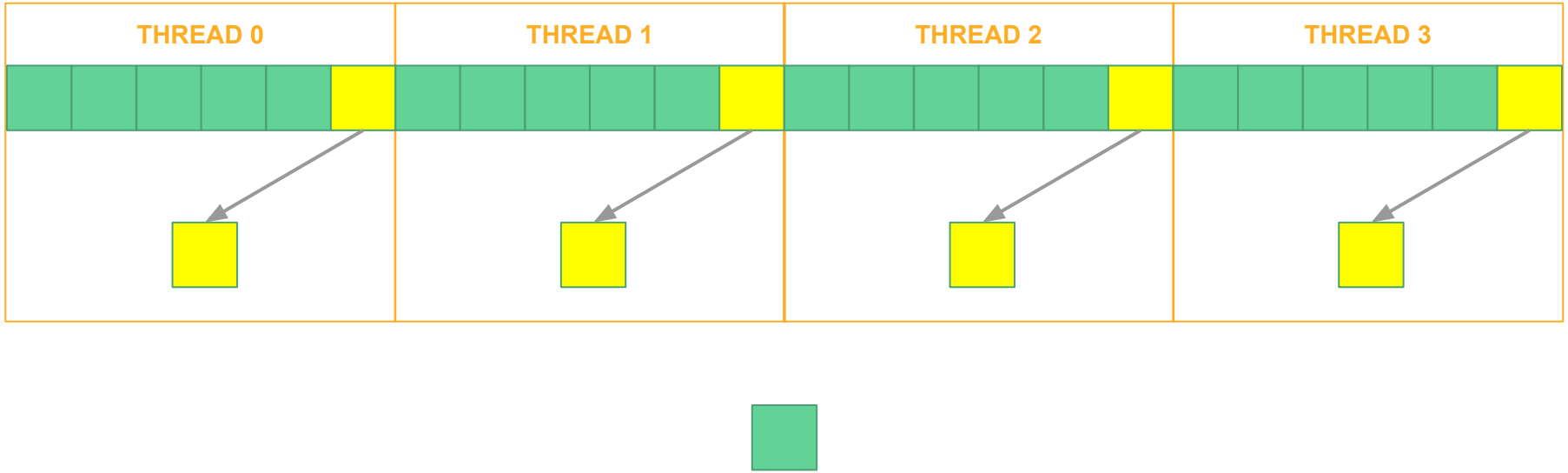
Finding the largest item in an array



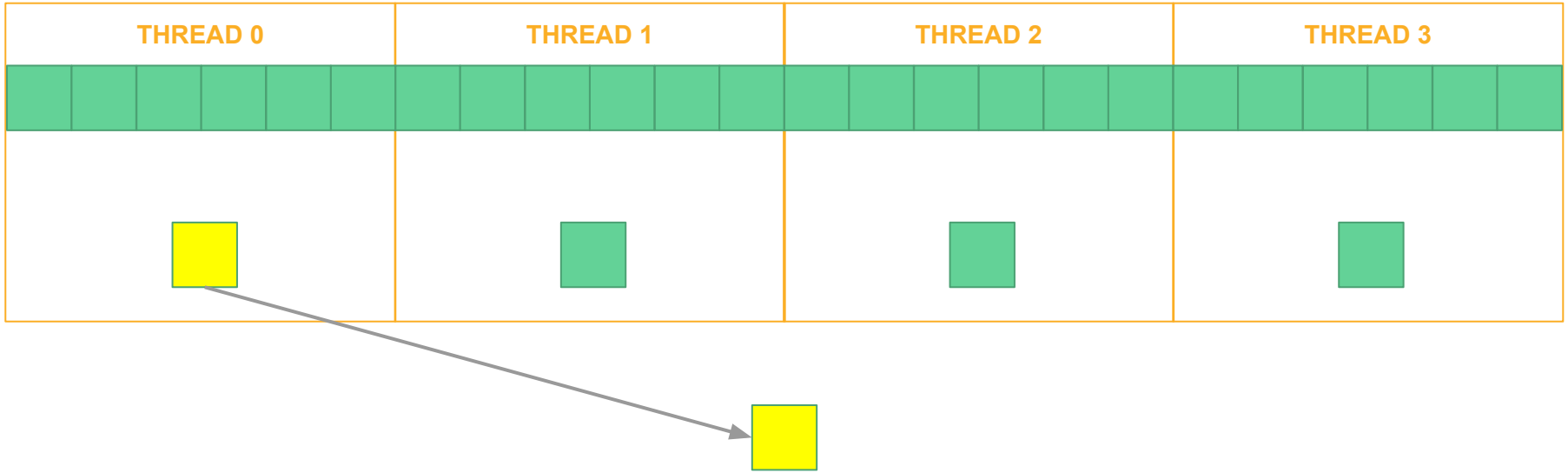
Finding the largest item in an array



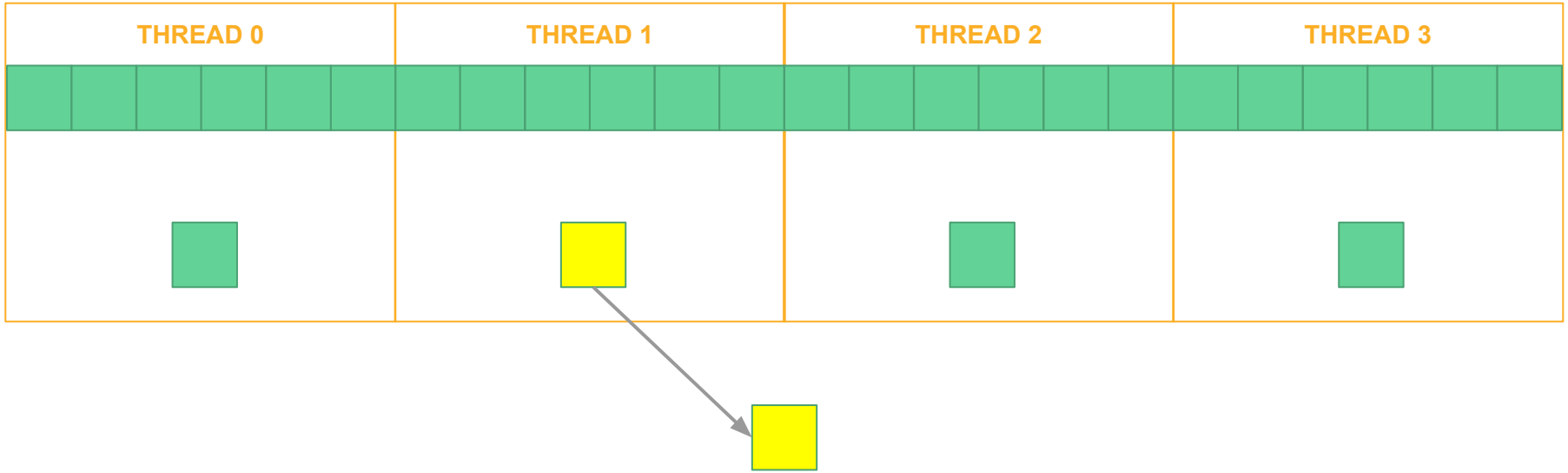
Finding the largest item in an array



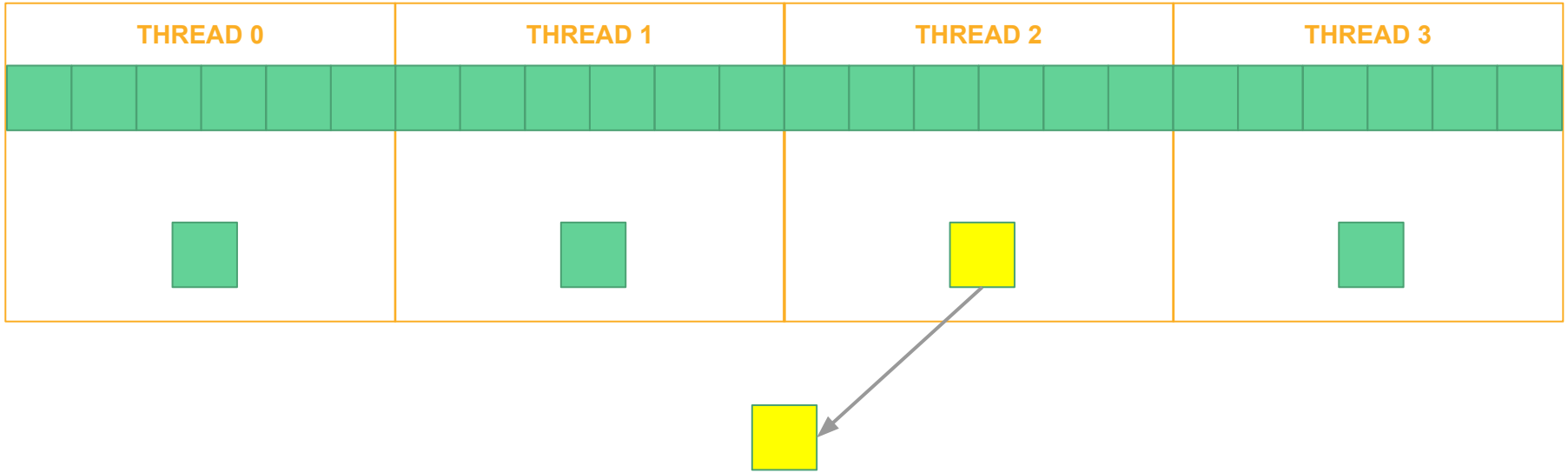
Finding the largest item in an array



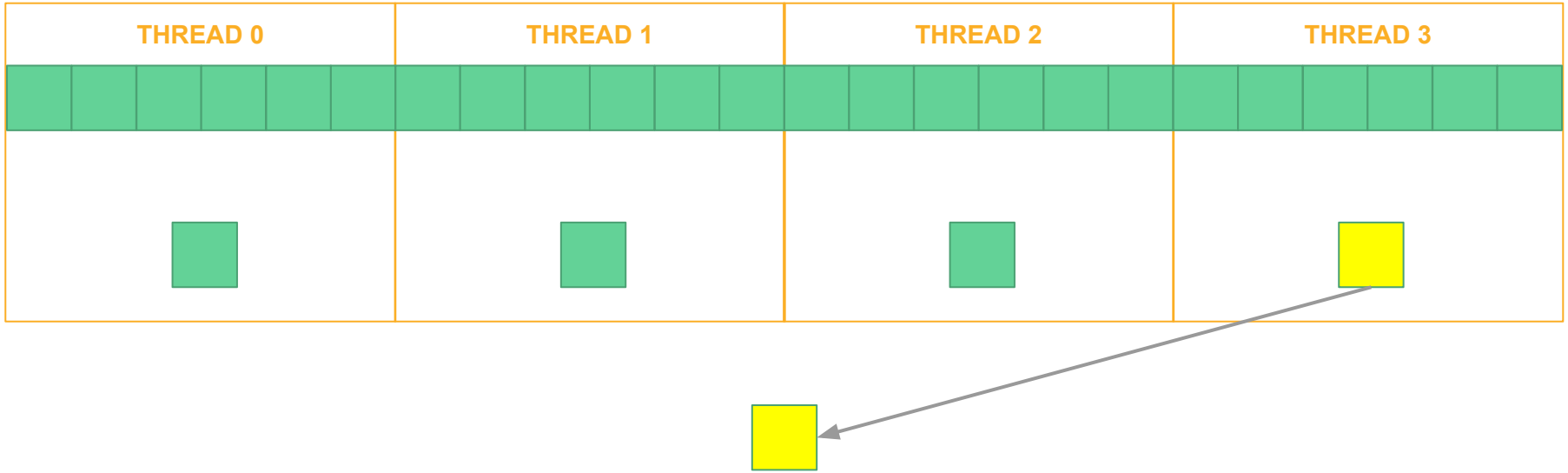
Finding the largest item in an array



Finding the largest item in an array



Finding the largest item in an array



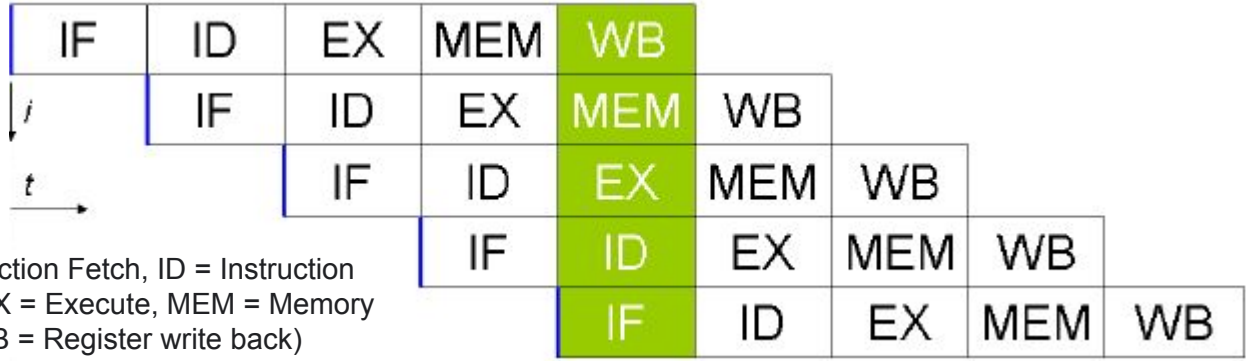
Pipeline architecture

Pipeline based processing / execution

Key points:

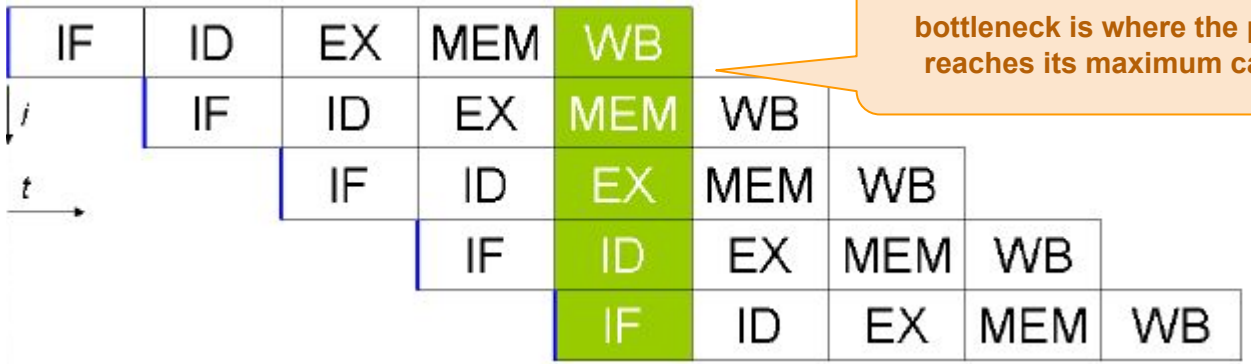
- The execution process is divided up into smaller “steps”, and the instructions are going through step-by-step on these, on every clock-cycle.

- A classical pipeline of a RISC processor:

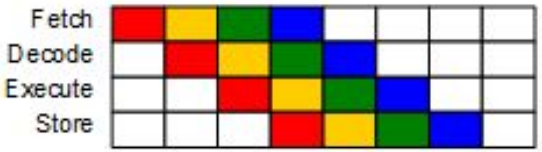


(IF = Instruction Fetch, ID = Instruction Decode, EX = Execute, MEM = Memory access, WB = Register write back)

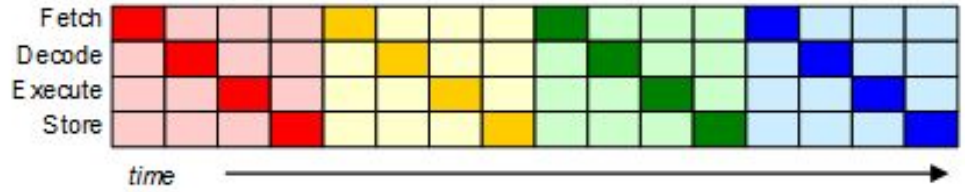
Pipeline based processing / execution



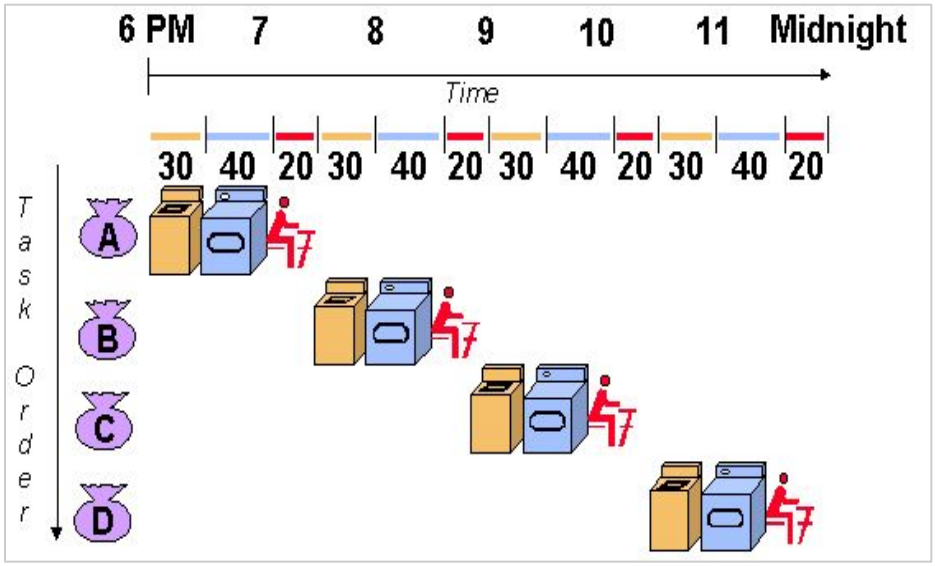
pipelined

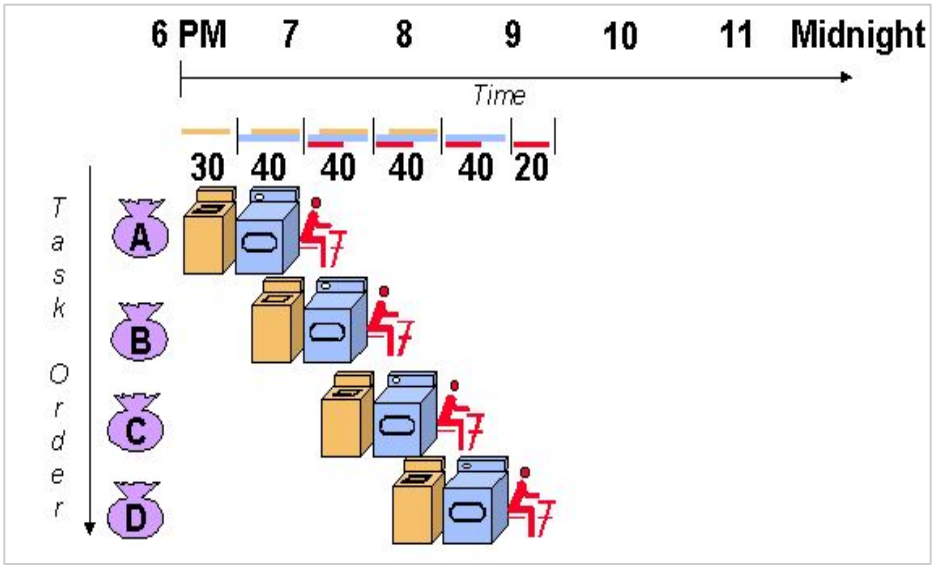


not pipelined



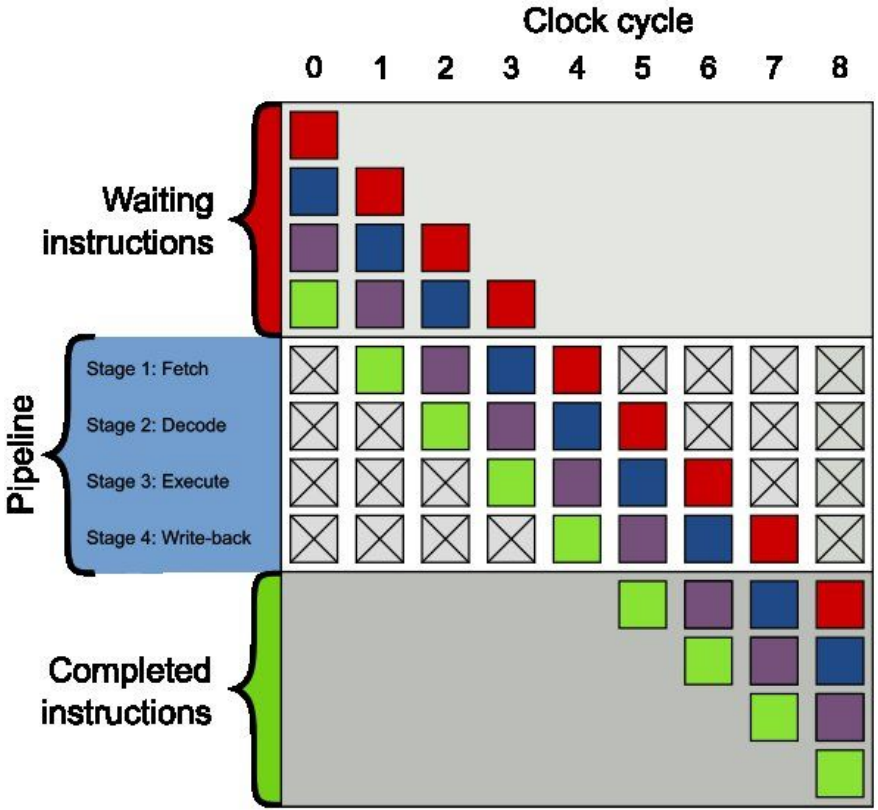
Pipeline based processing / execution



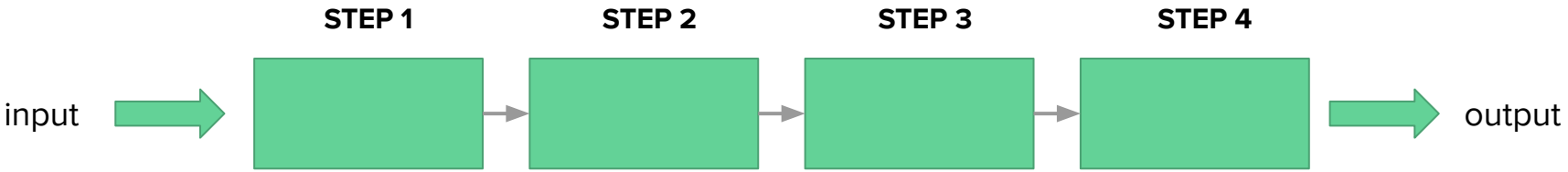


some steps can be made in parallel

Pipeline based processing / execution

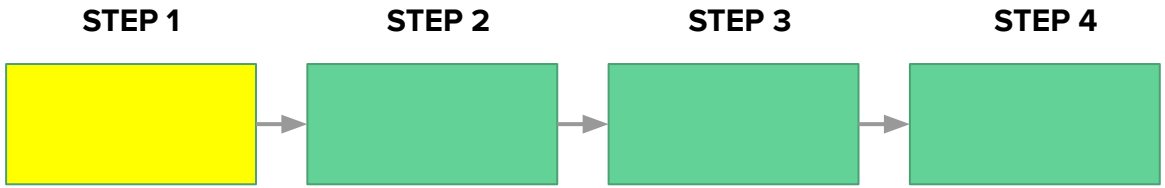


Pipeline based processing / execution

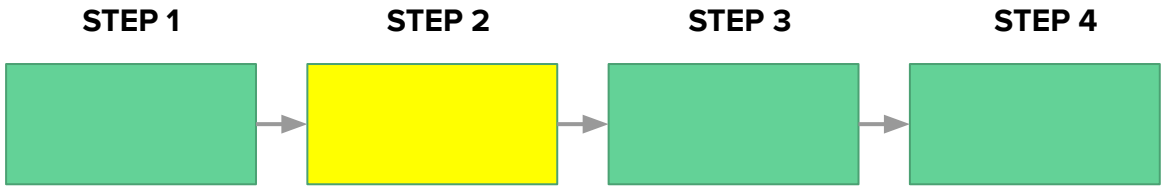


Everyday example: McDonalds drive-through

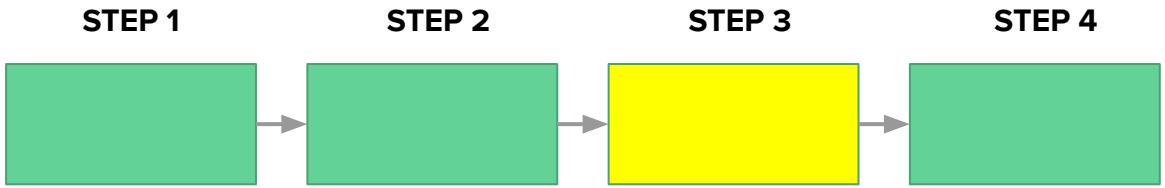
Pipeline based processing / execution (step 1.)



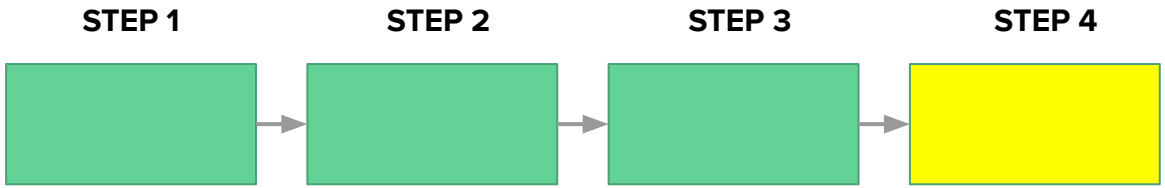
Pipeline based processing / execution (step 2.)



Pipeline based processing / execution (step 3.)

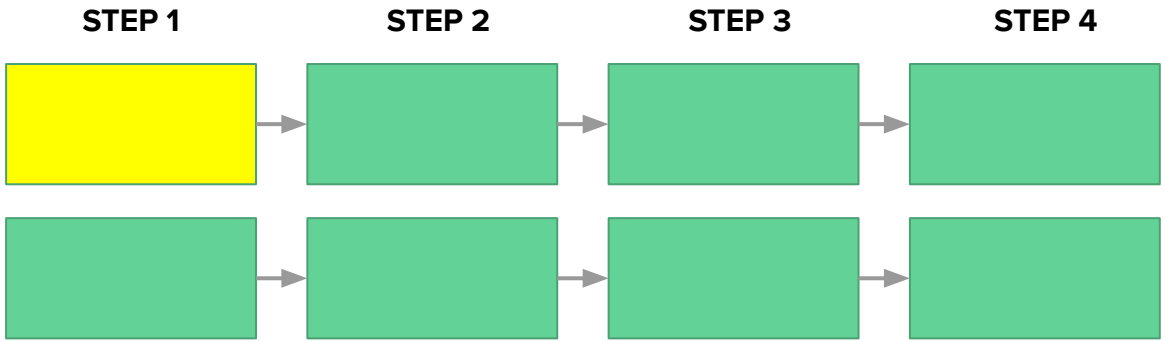


Pipeline based processing / execution (step 4.)

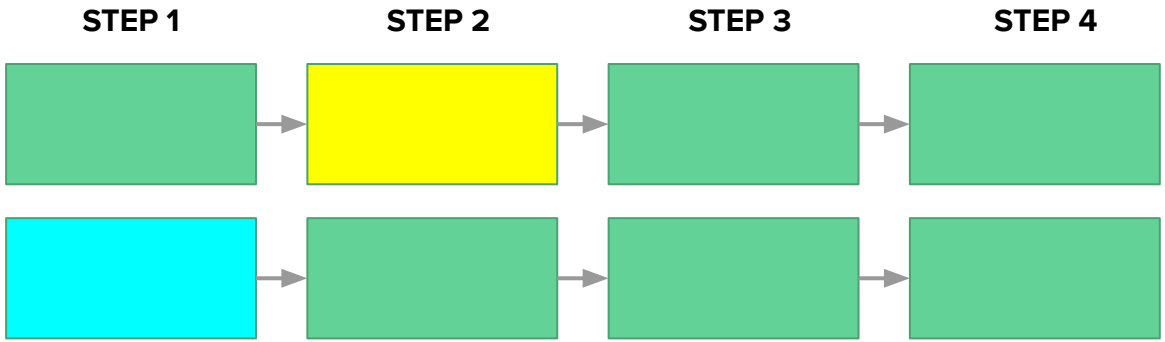


The pipeline finished 1 input in 4 steps.

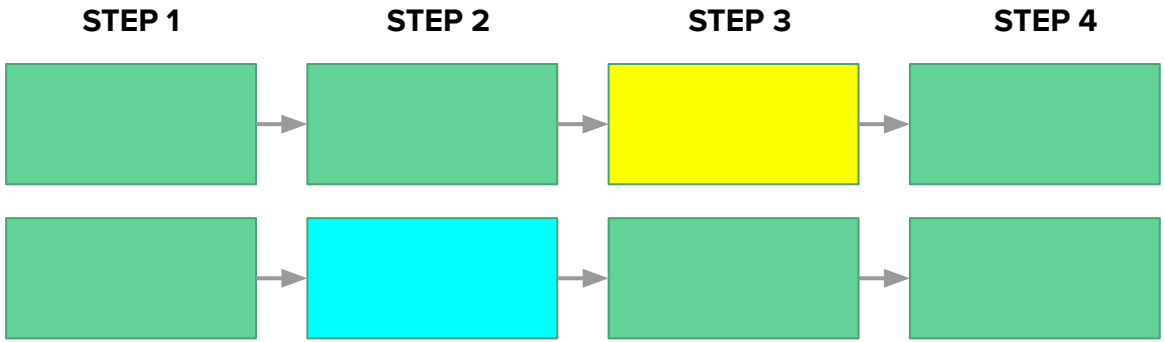
Pipeline based processing / execution (step 1.)



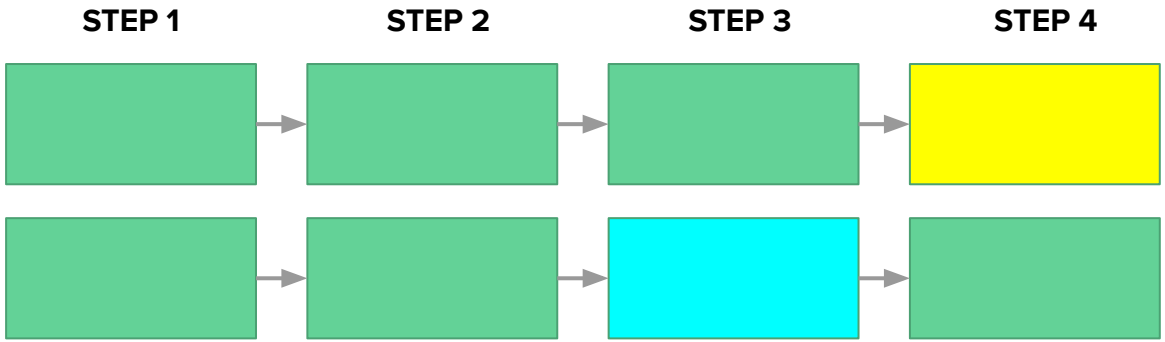
Pipeline based processing / execution (step 2.)



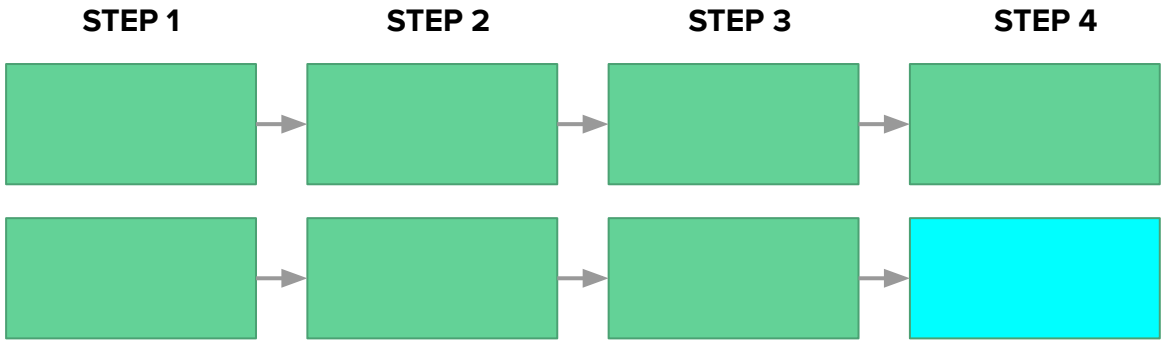
Pipeline based processing / execution (step 3.)



Pipeline based processing / execution (step 4.)

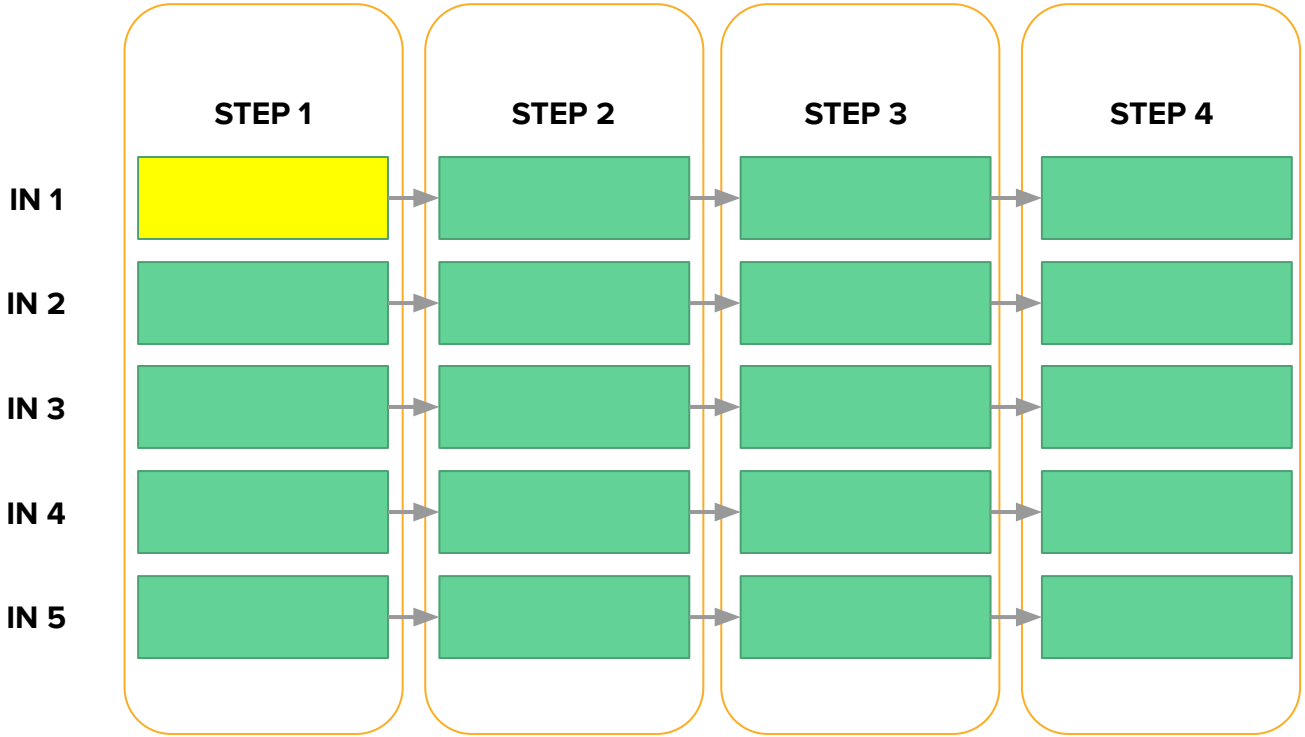


Pipeline based processing / execution (step 5.)

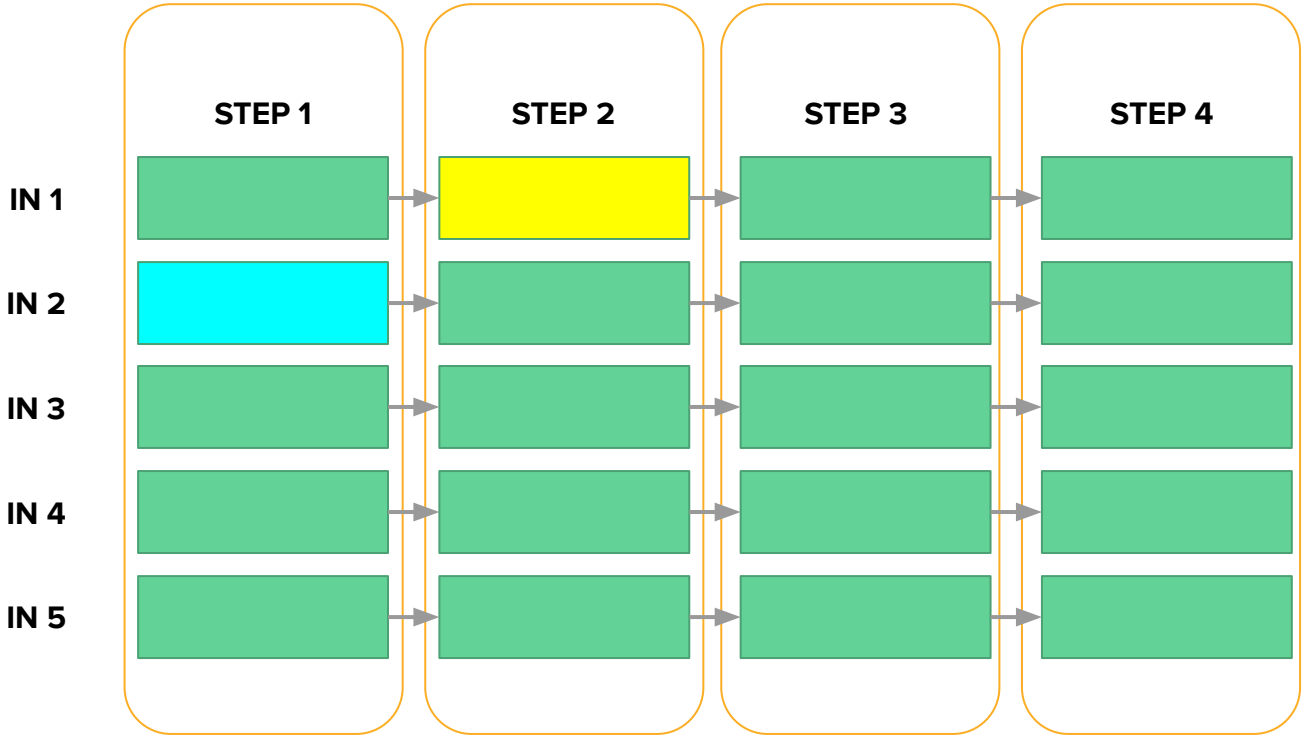


The pipeline finished 2 inputs in 5 steps.

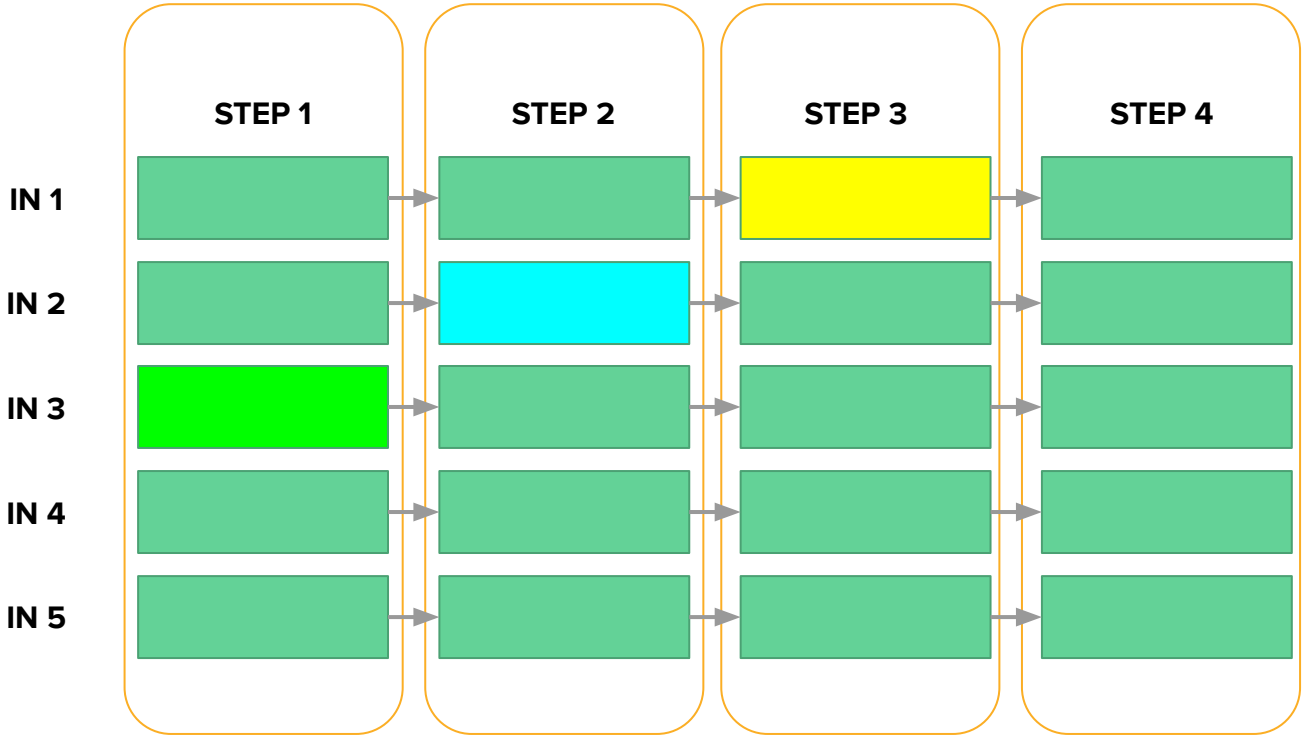
Pipeline based processing / execution (step 1.)



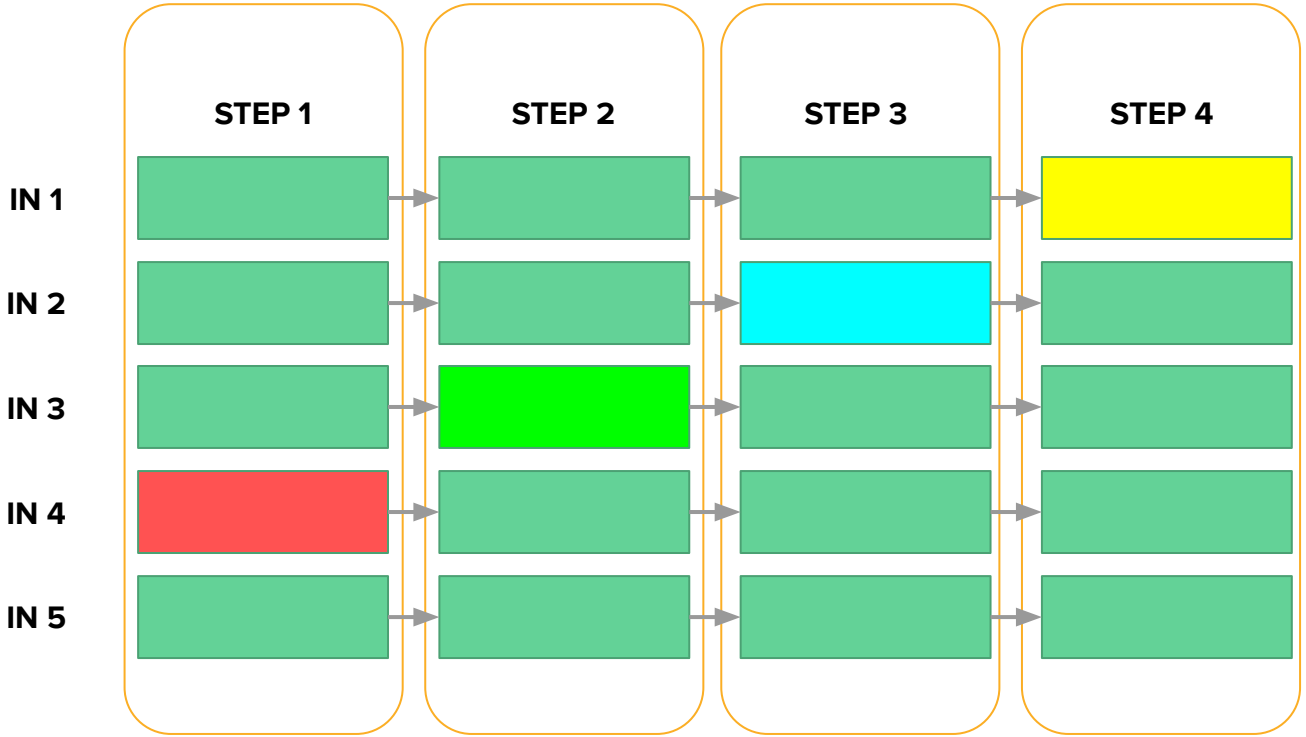
Pipeline based processing / execution (step 2.)



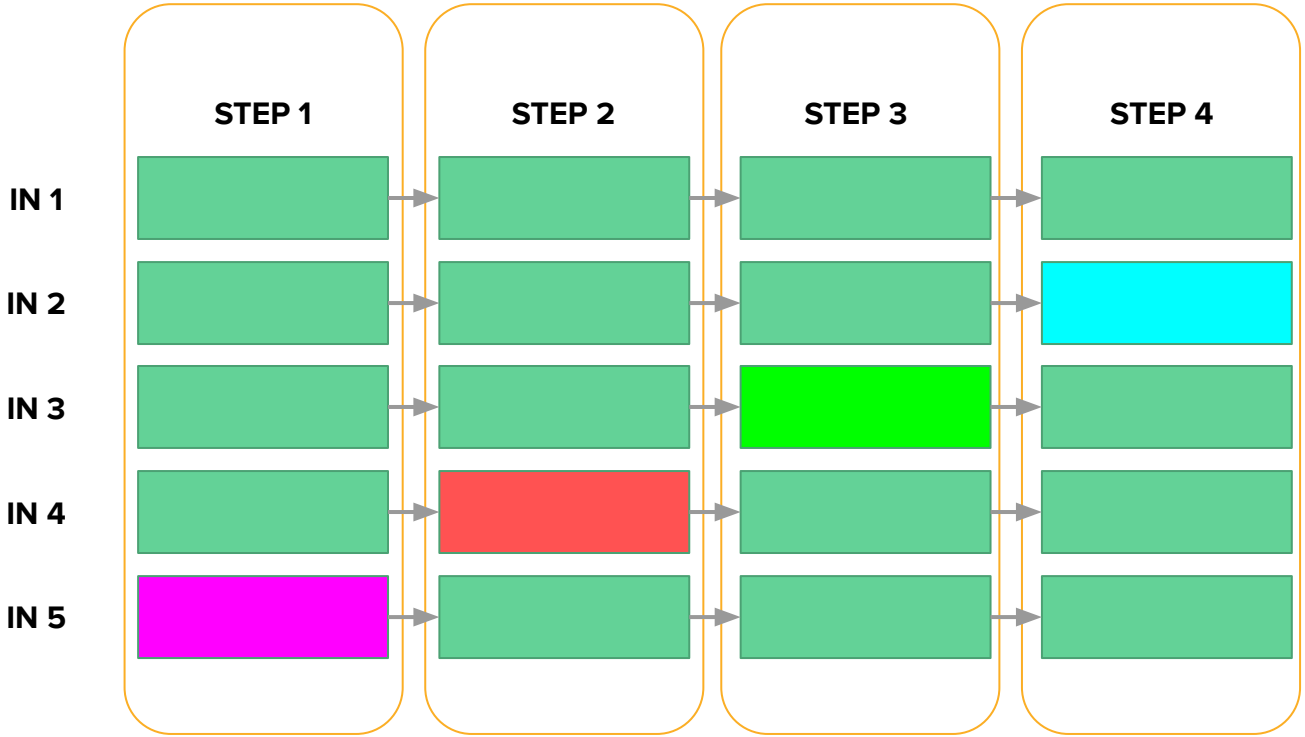
Pipeline based processing / execution (step 3.)



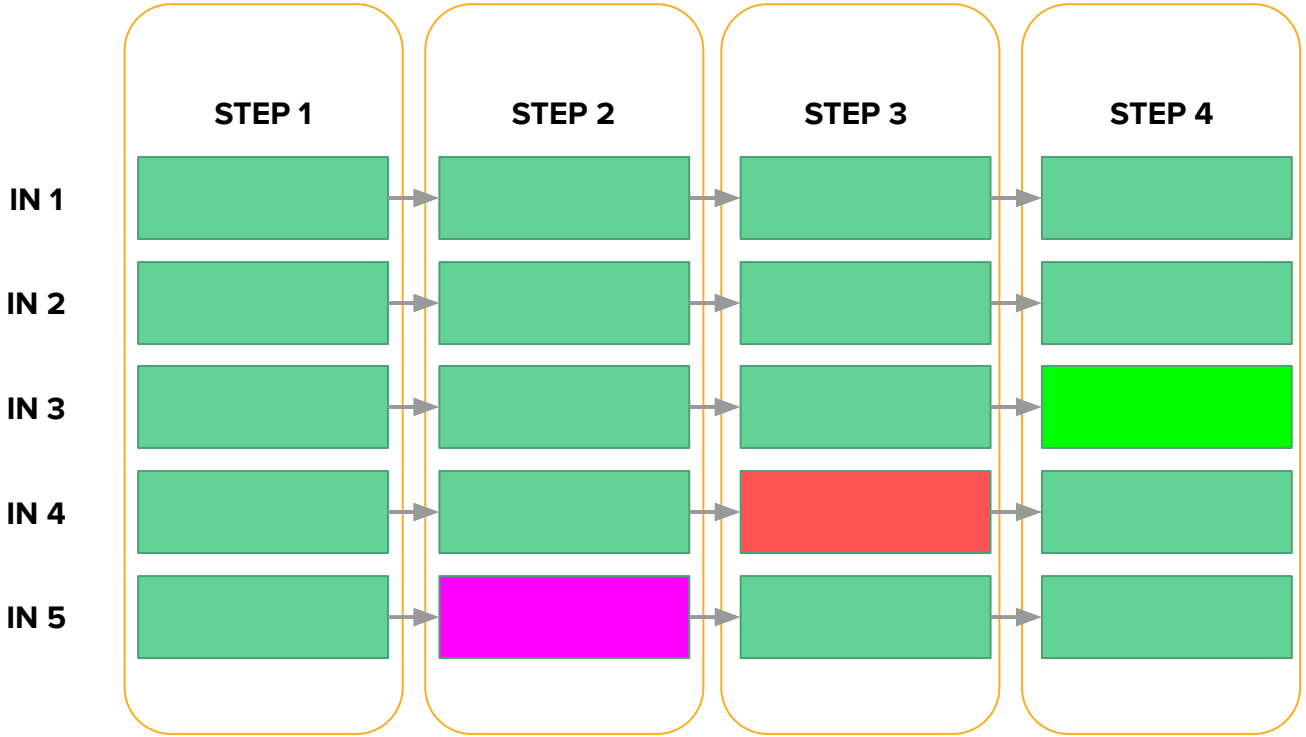
Pipeline based processing / execution (step 4.)



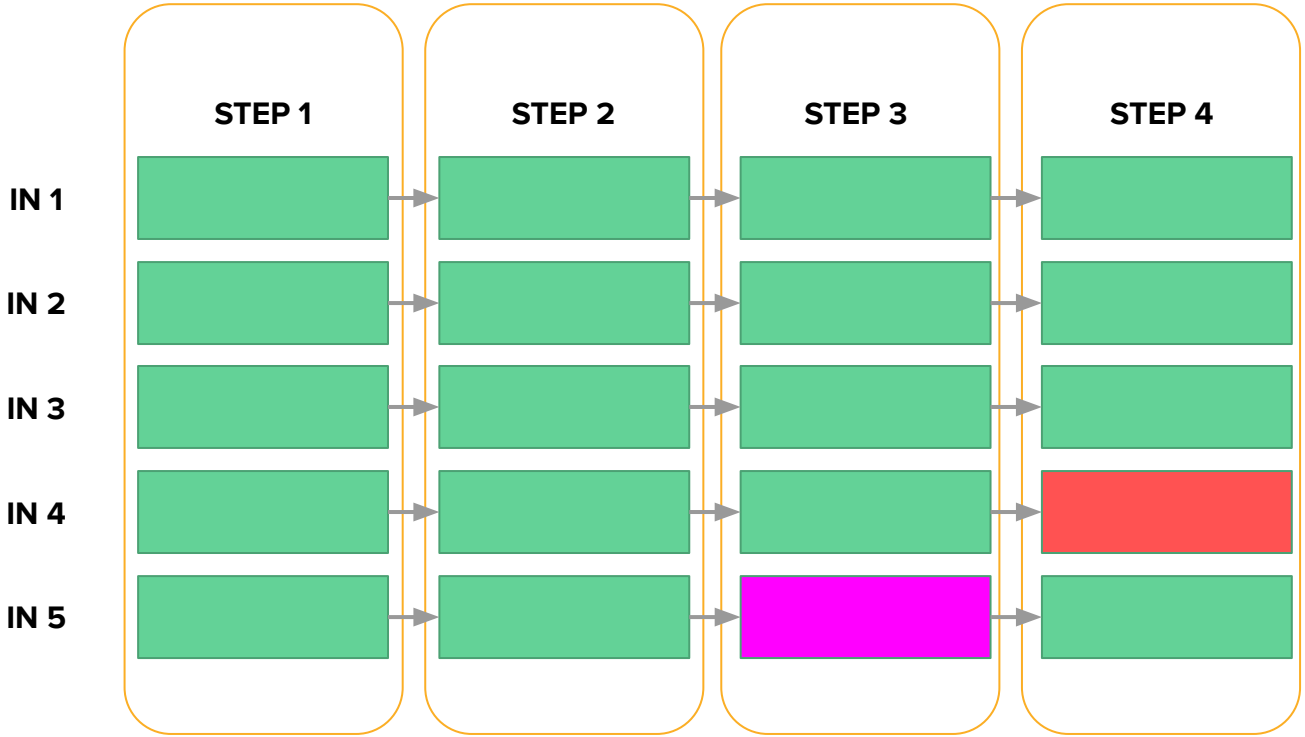
Pipeline based processing / execution (step 5.)



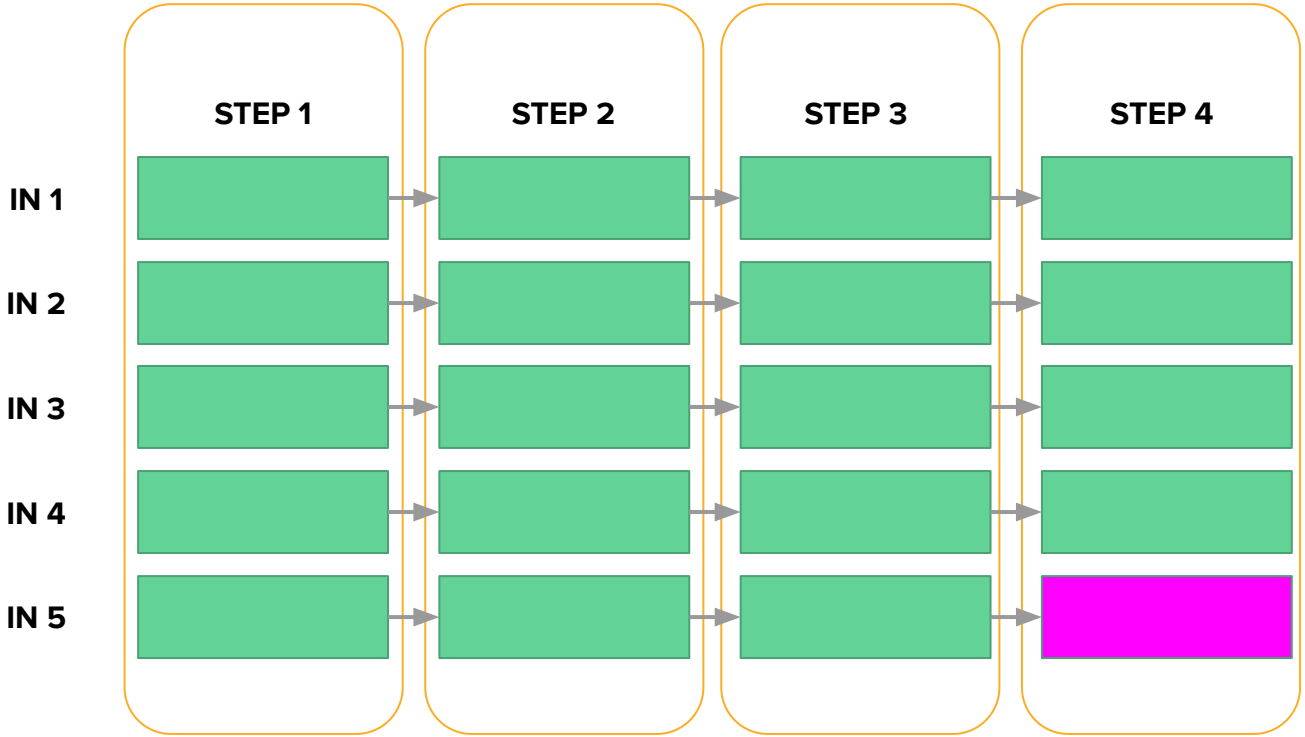
Pipeline based processing / execution (step 6.)



Pipeline based processing / execution (step 7.)

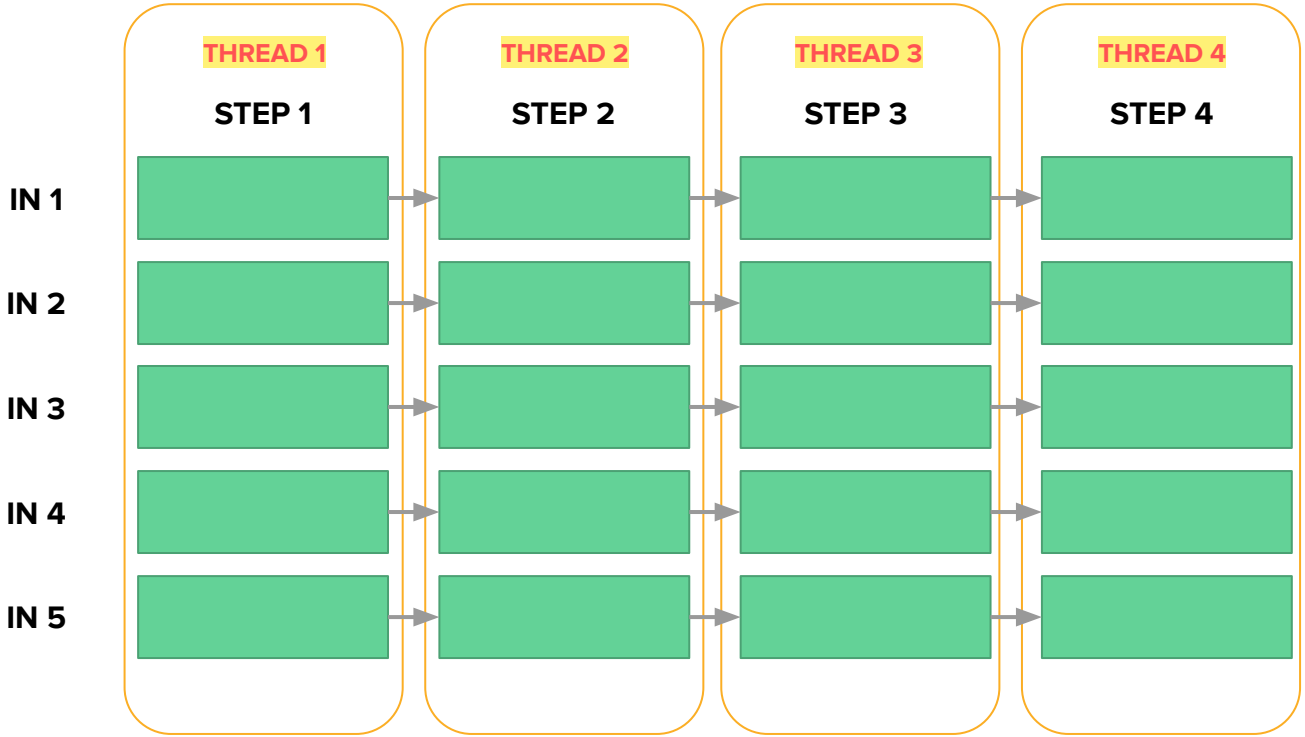


Pipeline based processing / execution (step 8.)

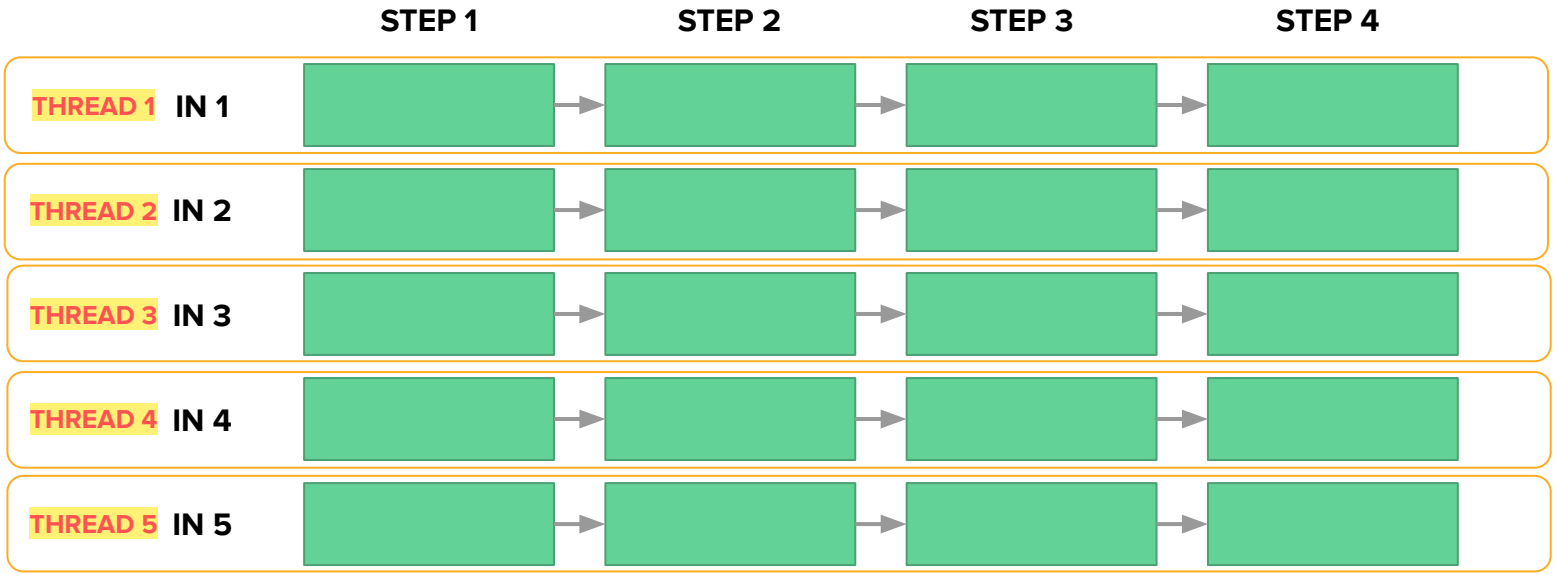


The pipeline finished 5 inputs in steps.

Thread-based representation 1.



Thread-based representation 2.



Processes and threads I.

- Process

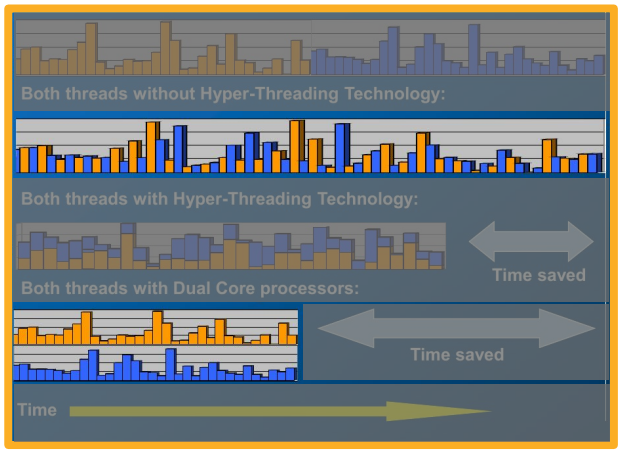
Processes (process)

Nowadays it is trivial, but think about old systems where there were only 1 command line interface and that's all.

- The OS can execute multiple programs simultaneously
  - The OS creates resources to this program, allocated memory for it and create priority for it
  - **The program which is under execution is called a process**
- Modern OSs load a program as a process
  - which owns resources such as:
    - memory
    - instruction counter register (instruction pointer, IP)
    - file pointer etc.
  - and whose run unit is a thread
- Every process owns at least 1 thread ➔ main thread
  - This main thread initializes the process and starts to run the instructions.
  - Within the process every thread shares their code- and data segment.

Processes (process)

- In order to run multiple processes **simultaneously** it requires multiple processor(core)s ➔ this is called **parallel processing**
- But we have the possibility to have different processes get access to the resources on a **time-sharing** base ➔ this is **concurrent processing**



Process (System.Diagnostics.Process)

Methods	
Start()	
CloseMainWindow()	Only with GUI processes
Kill()	
GetCurrentProcess()	
GetProcesses()	List of all processes
WaitForExit()	
Properties	
StartInfo	The assigned ProcessStartInfo instance
PriorityClass	
EnableRaisingEvents	
HasExited	
ExitCode, ExitTime	
StandardInput, StandardOutput	Streams
Events	
Exited	

Process (System.Diagnostics.ProcessStartInfo)

Properties	
FileName	Executable (on its own or using its associated program)
Arguments, WorkingDirectory	
Domain, UserName, Password	
RedirectStandardInput, RedirectStandardOutput	Redirect the specified stream?
ErrorDialog	True/false, if the execution fails
UseShellExecute	Use the OS shell execute method (Advantage: more clever execution, independent execution. Disadvantage: cannot read output)
Verb	Examples of verbs are "Edit", "Open", "OpenAsReadOnly", "Print", and "Printto"
WindowStyle	Initial window size (minimized, maximized)

Process / example 0.

Link: https://gitlab.com/siposm/oktatas-hft-20211/-/tree/master/_ARCHIVED/LA-09-process/parhuzamositas_process

- get and list all the processes which are running at the moment in the system

```
foreach (var p in Process.GetProcesses().OrderBy(x => x.Id))
    Console.WriteLine(string.Format("#{0}\t {1}", p.Id, p.ProcessName));
```

Process / example 1.

Link: https://gitlab.com/siposm/oktatas-hft-20211/-/tree/master/_ARCHIVED/LA-09-process/parhuzamositas_process

- open websites in browser programmatically

```
string[] urls = new string[]
{
    "http://users.nik.uni-obuda.hu/sztf2/",
    "http://users.nik.uni-obuda.hu/siposm/",
    "http://users.nik.uni-obuda.hu/gitstats/",
    "http://users.nik.uni-obuda.hu/prog3/",
    "http://users.nik.uni-obuda.hu/to/",
    "http://users.nik.uni-obuda.hu/prog4/"
};

for (int i = 0; i < urls.Length; i++) // i = 1000 ?
{
    Process p = new Process();
    p.StartInfo = new ProcessStartInfo();
    p.StartInfo.FileName = @"C:\Program Files (x86)\Google\Chrome\Application\chrome.exe";
    // filename = url esetén is az alapértelmezett program nyílik meg

    p.StartInfo.Arguments = urls[i];
    p.Start();

    //System.Threading.Thread.Sleep(1000);

}
```

Process / example 2.

Link: https://gitlab.com/siposm/oktatas-hft-20211/-/tree/master/_ARCHIVED/LA-09-process/parhuzamositas_process

- create a program which sums numbers from X to Y
- run it in 5 separate processes, in parallel
- X and Y should be given as argument

```
class Program
{
    static void Main(string[] args)
    {
        int min = int.Parse(args[0]);
        int max = int.Parse(args[1]);

        int sum = 0;
        for (int i = min; i <= max; i++)
        {
            Console.WriteLine($"[{i}]");
            System.Threading.Thread.Sleep(250); // negyed msp
            sum += i;
        }
        Console.WriteLine(" > SUM: " + sum);
    }
}
```

counter.exe

```
static void Main(string[] args)
{
    Random r = new Random();
    Process[] procs = new Process[5];

    for (int i = 0; i < procs.Length; i++)
    {
        procs[i] = new Process()
        {
            StartInfo = new ProcessStartInfo()
            {
                FileName = @"D:\CODES\oktatas-whp-19201\LA-09-process\parhuzamositas_process\02-counter\bin\Debug\02-counter.exe",
                Arguments = r.Next(10) + " " + r.Next(11,20),
                RedirectStandardOutput = true, // próba: = false
                UseShellExecute = false
            }
        };

        procs[i].Start();
    }

    SyncPoint(procs);
}
```

my program

```
static void SyncPoint(Process[] procs)
{
    for (int i = 0; i < procs.Length; i++)
    {
        procs[i].WaitForExit();
        Console.WriteLine(procs[i].StandardOutput.ReadToEnd());
    }
    // --- at this point, all of the processes will be done ~ synced
}
```

blocking

get return value

Using events the blocking part (WaitForExit) can be avoided.

```
p.EnableRaisingEvents = true;
p.Start();
p.Exited += P_Exited; // (A)
```

getting the return value from the event

```
private static void P_Exited(object sender, EventArgs e)
{
    Console.WriteLine((sender as Process).StandardOutput.ReadToEnd());
}
```

Process / example 3.

Link: https://gitlab.com/siposm/oktatas-hft-20211/-/tree/master/_ARCHIVED/LA-09-process/parhuzamositas_process

- open only the selected file types in notepad++

```
DirectoryInfo di = new DirectoryInfo("./files-to-check/");

var files = di.GetFiles("*", SearchOption.AllDirectories)
    .OrderBy( x => x.Name )
    .Where( x =>
            x.Name.Contains(".xml") ||
            x.Name.Contains(".json") ||
            x.Name.Contains(".cs") ||
            x.Name.Contains(".sql")).ToArray();


Process[] procs = new Process[files.Length];

for (int i = 0; i < procs.Length; i++)
{
    procs[i] = new Process()
    {
        StartInfo = new ProcessStartInfo()
        {
            //FileName = @"C:\Program Files\Notepad++\notepad++.exe",
            FileName = "Notepad++.exe",
            Arguments = files[i].DirectoryName + @"\" +files[i].Name
        }
    };
    procs[i].Start();
}
```

Process / example 4.

Link: https://gitlab.com/siposm/oktatas-hft-20211/-/tree/master/_ARCHIVED/LA-09-process/parhuzamositas_process

- pass input arguments to the process

```
static void Main(string[] args)
{
    string open = args[0];
    string save = args[1];
    int timeDelay = int.Parse(args[2]);
```

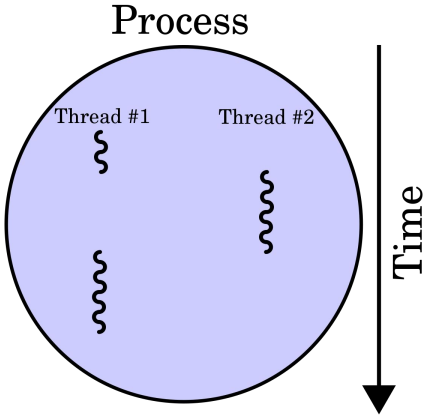
```
StartInfo = new ProcessStartInfo()
{
    FileName = @"D:\CODES\oktatas-whp-19201\LA-09-process\parhuza
    Arguments = inputs[i] + " " + outputs[i] + " " + delays[i]
}
```

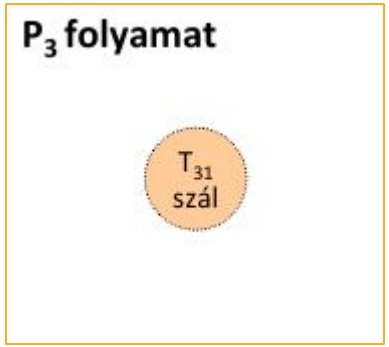
Processes and threads II.
- Thread

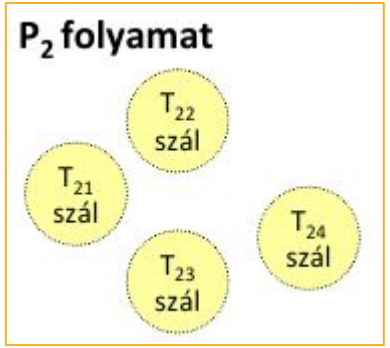
Threads (thread)

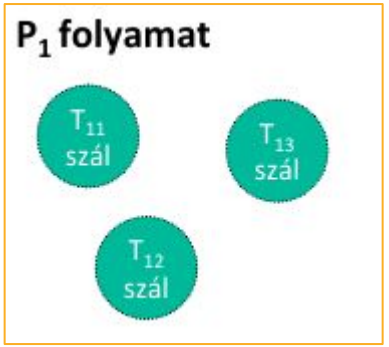
- Thread = **a discrete series of interrelated instructions.**
- Every process has at least 1 thread ➜ main thread
- There could be multiple threads inside a process
- Concurrent threads can be run on the same core; but to reach parallelism multiple cores are needed.
- Threads are using their host processes' memory segment.

Process ↔ Thread









Why do we use threads? (= parallel programming in general)

- Advantages
  - **increased performance**
  - **better resource management / usage**
  - efficient datasharing

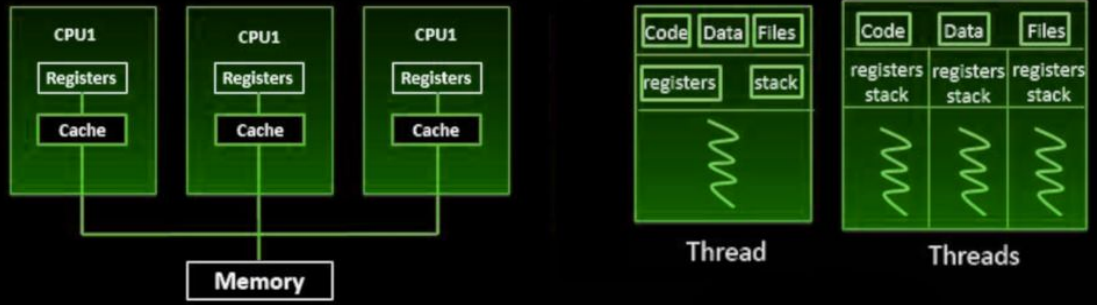
- Disadvantages
  - **race condition**
  - **deadlock**
  - **code complexity**
  - portability problems
  - hard testing and debugging

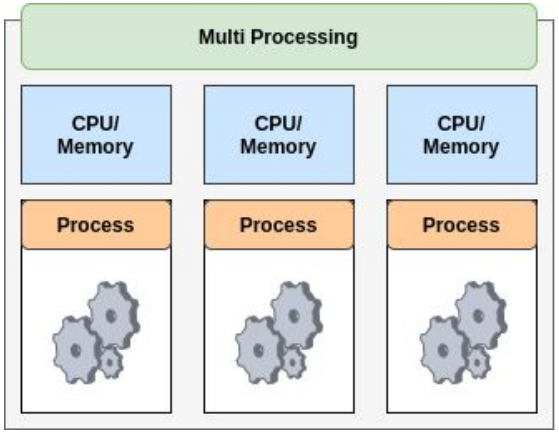
Parallelism

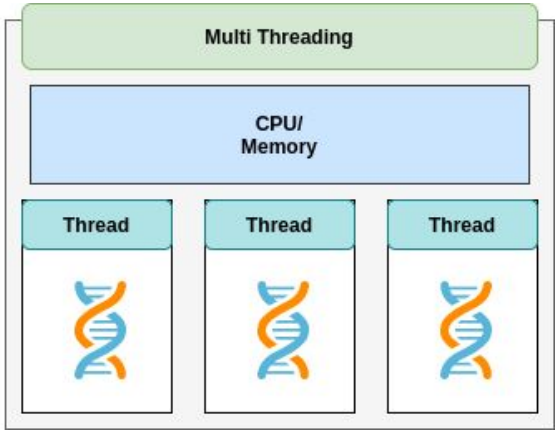
- Multiprocessing
  - **Multiple processes are working on the same task**
  - Each process uses its own CPU for their calculations
  - Each process works on different memory segment
    - for data exchange they have to communicate

- Multithreading
  - **Multiple threads (inside one process) are working on the task**
  - Each thread uses separate CPU cores for their calculations
  - Threads use the same shared memory segment
    - for data usage the accessing order is important

Multiprocessing vs Multithreading







Thread management

By this term we mean the followings:

- thread
  - creation
  - deletion
  - execution management
- registration and management of the concurrently running threads
- context switching between the threads

Possibilities for multi-threaded processing

- **Asynchronous** (...async) method call
  - We can call delegates and special methods supporting this pattern
- **Thread** class
  - Low-level management of threads
  - Simple (simply “execute this method in another thread”), but requires lots of accuracy and planning
    - Concurrency? Return values?
- **ThreadPool** class
  - There is a set/pool of threads waiting for usage
  - No need for individually creating/destroying threads
  - Simple and very efficient, but lacks functionality (e.g. thread identification)
- **BackgroundWorker** class
  - Typically used to separate the UI thread from long operations
  - Very limited functionality
- **TPL / Task** class
  - High-level thread management, unified API with high functionality
- **TPL / Parallel** class
  - Data-parallel execution, Parallel loops
- **Async-Await**

Thread (System.Threading.Thread)

Methods	
Start()	
Suspend(), Resume()	
Abort()	Rarely used
GetHashCode()	Usable for identification
Sleep()	Wait a given timespan
Join()	Wait for a thread to finish
Properties	
CurrentCulture, CurrentUICulture	
IsBackground	The execution of a program is finished when the last foreground thread is finished (the background threads are aborted)
IsThreadPoolThread	
ManagedThreadID	
Name	
Priority	
ThreadState	

```
1   using System;
2   using System.Threading;
3
4   class Program
5   {
6           static void Main(string[] args)
7           {
8                   Console.WriteLine("Now we create a thread");
9                   Console.WriteLine("Main thread ({0})",
10  Thread.CurrentThread.GetHashCode());
11                  Thread newThread = new Thread(ThreadMethod);
12          newThread.Name = "NewThread";
13                  newThread.Start();
14                  newThread.Join();
15          }
16
17          static void ThreadMethod()
18          {
19                  Console.WriteLine("{0} (ID: {1})",
20                          Thread.CurrentThread.Name,
21                          Thread.CurrentThread.GetHashCode());
22          }
23  }
```

Basically in bird-view it is the same as working with Processes (except there are a few method with different names eg. join vs waitForexit), but in the background we should know what is the difference.

Main thread?
The Console is a 1 threaded environment.

Join()?
Synchronization, see later.

The job to be done
We pass to the thread what is the work/job to be done. Note that is it a much better way compared to processes!

Thread example code

Thread

- Low-level tool to define threads
- Hard to handle how threads are build on top of each other
- Expensive to create and/or delete and switching between them
  ➜ threadpool ➜ Task
- No in-built option for safe-abortion
- No option to define return type
- Method parameterization not so developer friendly

➜ general solution: Task (see later)

Thread

```
Thread[] threads = new Thread[10];

for (int i = 0; i < threads.Length; i++)
{
    threads[i] = new Thread(Count);
    threads[i].Start(i);
}
```

ThreadStart or ParameterizedthreadStart delegate is accepted but we often add lambda instead.

Count ⇒ what is the job which has to be done.

```
for (int i = 0; i < threads.Length; i++)
    threads[i].Join();
```

Text file processing in parallel

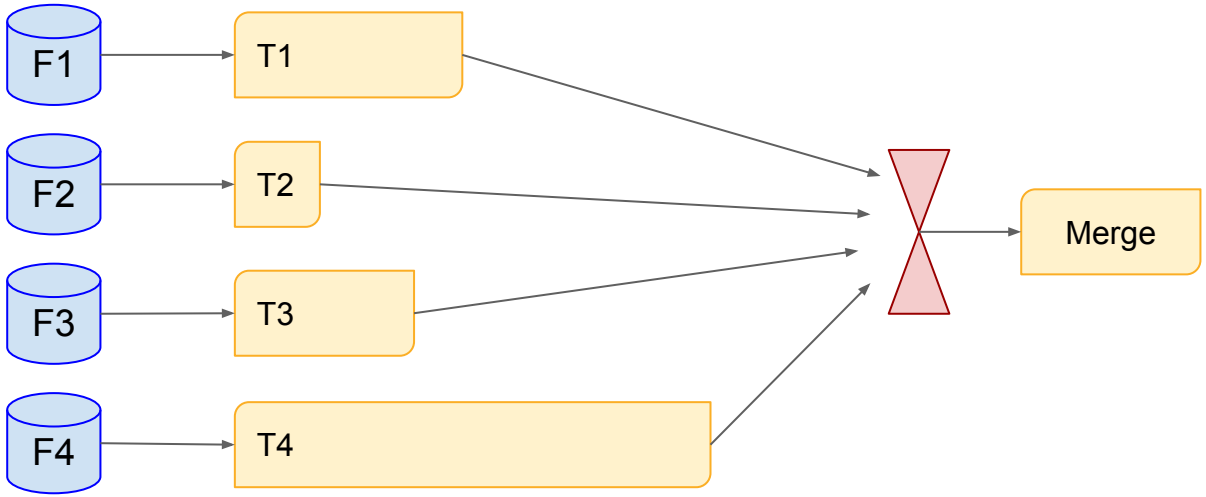
https://github.com/siposm/oktatas-hft-20211/tree/master/LA-09-thread/02-dataprocess

https://github.com/siposm/oktatas-hft-20211/blob/master/LA-09-thread/02-dataprocess/Program.cs

Text file processing in parallel

Task:

- extract important sections of a text file (eg. in log analysis)
- the extracted sections should be merged together in 1 output file



```
class DataProcessor
{
    public static void Process(object o)
    {
        DataTransfer dt = o as DataTransfer;

        StreamReader sr = new StreamReader(dt.OpenFile);
        string full = sr.ReadToEnd();

        string saveStr = "";
        foreach (var item in full.Split('\n'))
            if (item.Contains("DATE:"))
                saveStr += item + "\n";

        System.Threading.Thread.Sleep(dt.TimeDelay * 1000);

        StreamWriter sw = new StreamWriter(dt.SaveAs);
        sw.Write(saveStr);
        sw.Close();
    }
```

```
public static void CollectData(string[] outputs)
{
    string full = "";
    for (int i = 0; i < outputs.Length; i++)
    {
        StreamReader sr = new StreamReader(outputs[i]);
        full += sr.ReadToEnd() + "\n";
        sr.Close();
    }
    StreamWriter sw = new StreamWriter("./_output/final.txt");
    sw.Write(full);
    sw.Close();
}
```

```
Thread[] threads = new Thread[inputs.Length];

for (int i = 0; i < threads.Length; i++)
{
    threads[i] = new Thread(DataProcessor.Process);
    threads[i].Start(new DataTransfer()
    {
        OpenFile = inputs[i],
        SaveAs = outputs[i],
        TimeDelay = delays[i]
    });
}

// szinkronizációs pont
for (int i = 0; i < threads.Length; i++)
    threads[i].Join();

// ezen a ponton biztosan minden kimeneti fájl előállt már
DataProcessor.CollectData(outputs);
```

1. Start

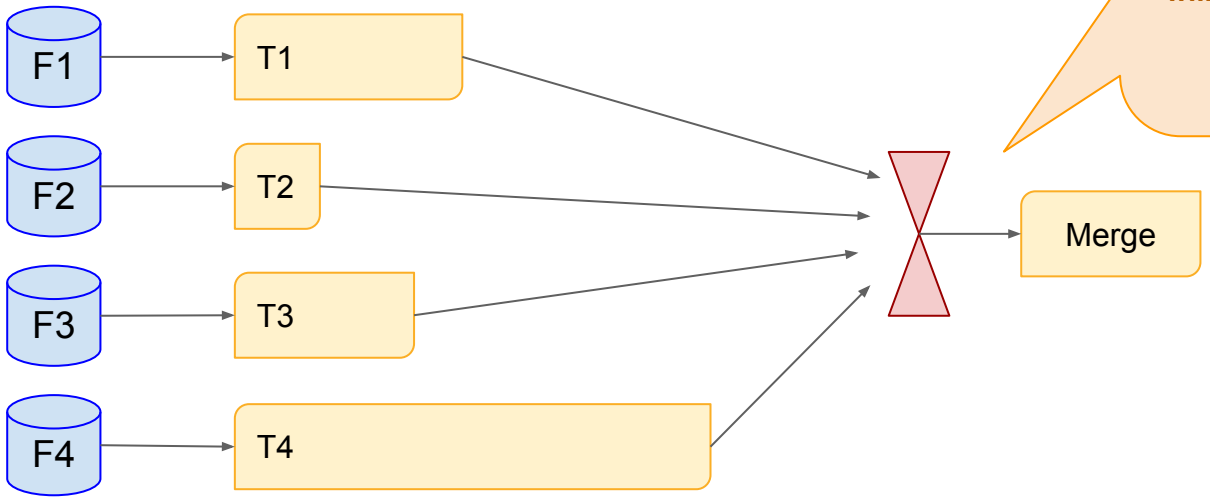
2. Sync

3. Merge

Text file processing in parallel

Task:

- extract important sections of a text file (eg. in log ana
- the extracted sections should be merged together in



If the merge part is not needed then we would not need the sync point. But since we need it, the bottleneck will be the longest-running thread, which must be waited.

Text file processing in parallel

Note 1.: of course using Process or Task the same can be achieved, but the key is that how easy (developer-friendly) to work with these in code. (And we have not talked about exception or cancellation yet.)

Example with Process:

- https://github.com/siposm/oktatas-hft-20211/tree/master/_archived/LA-09-process/parhuzamositas_process
- https://github.com/siposm/oktatas-hft-20211/tree/master/_archived/LA-09-process/parhuzamositas_process/08-fileProcessor
- https://github.com/siposm/oktatas-hft-20211/tree/master/_archived/LA-09-process/parhuzamositas_process/09-fileProcessor-main

Note 2.: if the threads (or the files) are in some kind of correlation then maybe the parallel processing part must be structured differently. ➔ The solution must be profoundly thoughtful. This is true for every programming case, but in parallel programming it's more important, since it is easily possible that we will create a **slower** program using (incorrectly used) parallelism compared to sequential.

End of 1st section.

Sources

Miklós Árpád, Dr. Vámossy Zoltán Design possibilities and methods of parallel algorithms lecture slides

Szabó-Resch Miklós Zsolt és Cseri Orsolya Eszter Advanced Programming lecture slides

Dr. Kertész Gábor Programming of Parallel and Distributed Systems lecture slides

MSDN

Sipos Miklós BPROF SzT specialization's Advanced Development Techniques lecture slides

Sipos Miklós GitHub codes

Thanks for your attention!

Sipos Miklós

sipos.miklos@nik.uni-obuda.hu

https://users.nik.uni-obuda.hu/siposm/