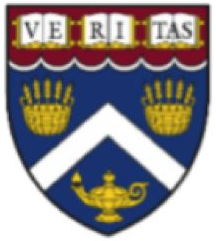
CSCI E-88 Principles Of Big Data Processing

Harvard University Extension, Fall 2019

Marina Popova

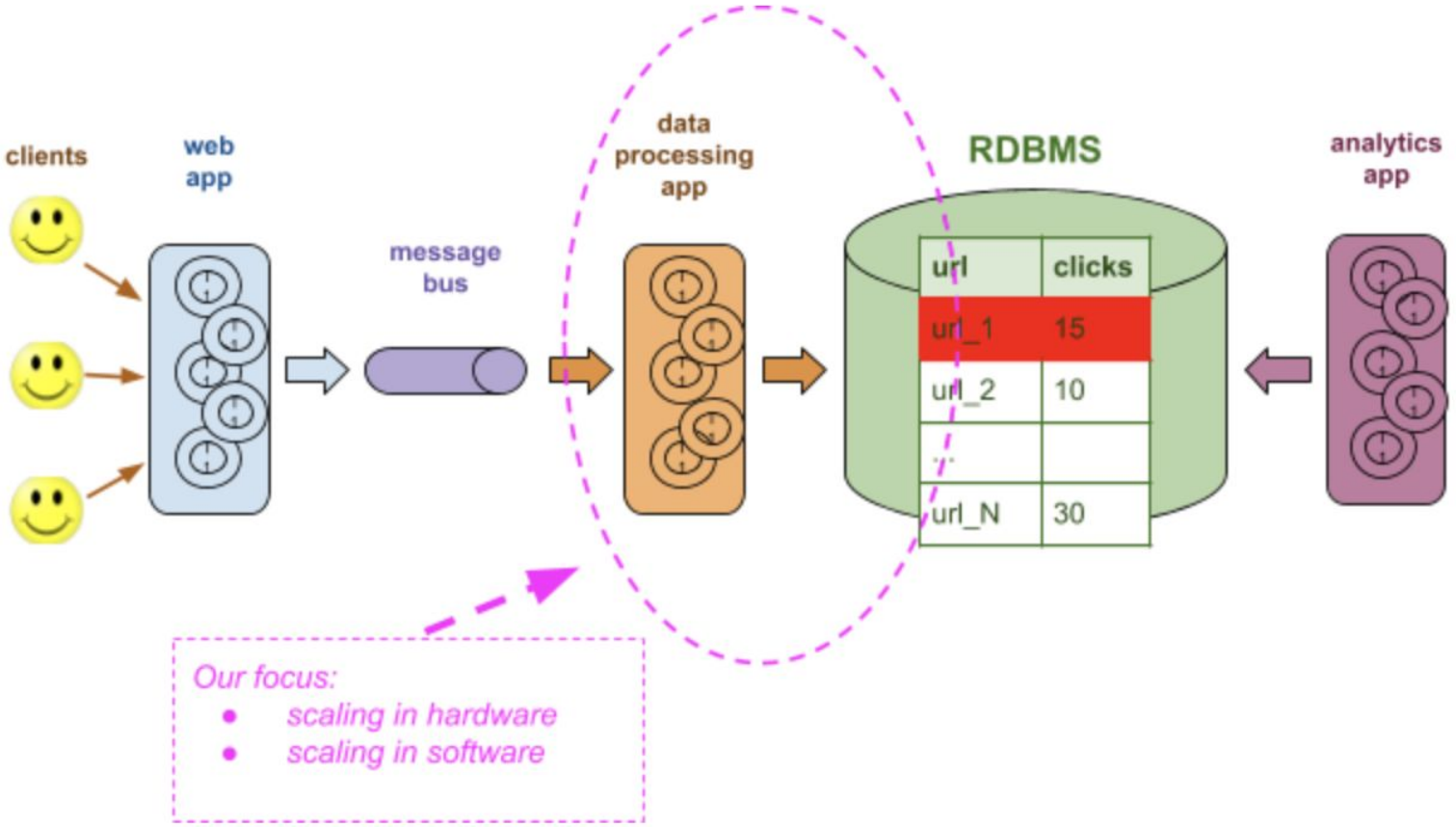


Lecture 2 - Scaling and Shared State Management

Agenda

- Common scaling approaches: on Hardware and Software layers
- Shared state management
- Intro to Redis

Where are we?


clients
web app
message bus
data processing app
RDBMS
analytics app
url
clicks
url_1
15
url_2
10
url_N
30
Our focus:
scaling in hardware
scaling in software

Scaling: horizontal vs. vertical

Traditional approaches to scale applications: vertical and horizontal

Vertical scaling: adding resources to your existing hardware/server: CPU, RAM, disk
Horizontal scaling: adding servers

- Refer to the Lecture 1 where we we scaling our Web and Business apps first vertically, and then horizontally

[JBoss Blog:Ref: http://middlewareblog.redhat.com/2016/04/15/intro-to-scalability/]

A variation of horizontal / vertical scale is to introduce data splitting, so certain types of data or certain operations are located across systems (horizontally) which are optimized for those types of loads (vertically). **That's the XYZ axis scale.**

Scaling: X, Y, Z scaling

Introduced by Martin Abbott and Michael Fisher in their book The Art of Scalability:

Reasons: not all operations or applications need the same resources for high performance. Some may need high CPU and memory. Others may need to store a lot of data and provide fast access to it, so the need large high performance disk storage. Instead of just adding more or better hardware or adding clones to a pool, delivering different types of data to different consumers can be done by handling that data in different ways

X-axis scaling is pretty much traditional horizontal scaling, which distributes the total load across a given number of nodes. **Y-axis and Z-axis scaling**, however, are **two entirely different approaches** by focusing on different things that can be scaled.

Scaling: X, Y, Z scaling

Y-axis scaling refers to breaking out and distributing services and is called ***functional decomposition***. It is a design approach which is reflected in service-oriented and microservices architectures.

Z-axis scaling refers to data partitioning and is also referred to as ***data decomposition***.

It is a way of distributing data among many nodes or blocks as a way of improving performance. The underlying data sets contain the same type of information, but any given partition only contains part of the information. Multiple threads can have the same code (instructions) being executed but they are using different sets of [localised] data

Both functional decomposition and data decomposition are core principles used in the **map-reduce type parallelization** frameworks - as we will see later

Scaling on Hardware Level

Main concepts:

- Processor
- CPU
- physical cores
- logical cores ==> hyper-threading
- logical CPU vs virtual CPU vs cloud virtualization
- Multi-processor (with single core per processor) vs multi-core architectures

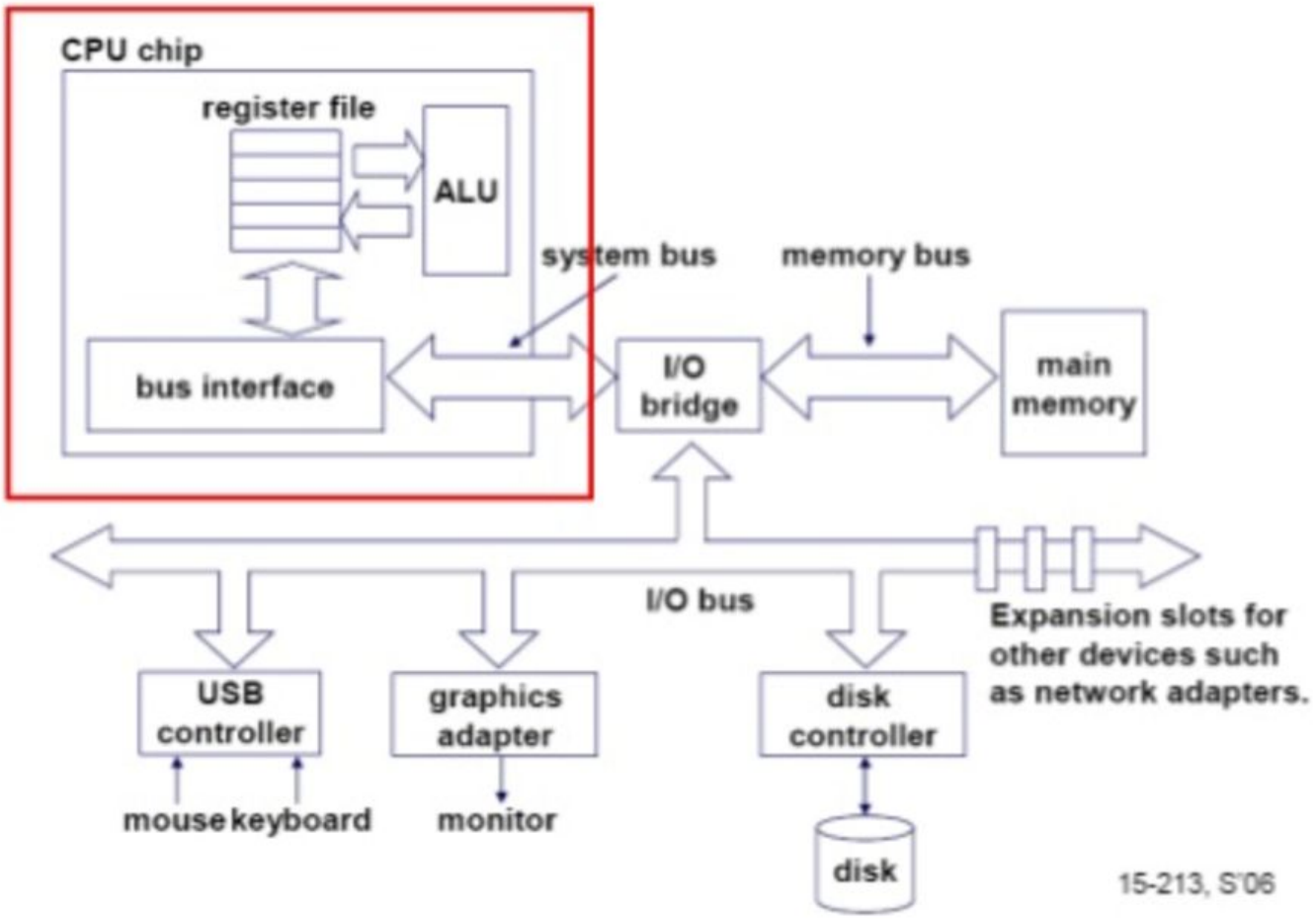
Scaling on Hardware Level

A CPU, or Central Processing Unit, is what is typically referred to as a processor. A processor contains many discrete parts within it, such as one or more memory caches for instructions and data, instruction decoders, and various types of execution units for performing arithmetic or logical operations.

A Core is the physical element that executes the code (like ALUs == Arithmetic Logic Unit).

Usually, each core has all necessary elements to perform computations, register files, interrupt lines etc.

Single-core computer



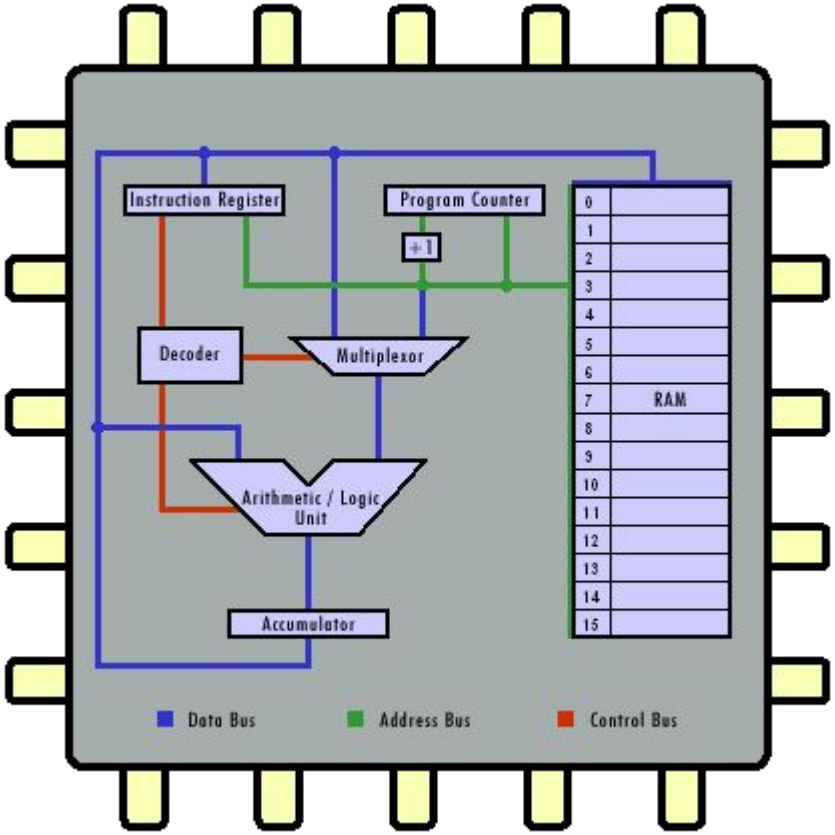
https://www.slideshare.net/piyushmittalin/multi-corearchitecture

Scaling on Hardware Level

CPU in details:

- ALU: Arithmetic Logic Unit
- Registers
- RAM
- Control Unit (Decoder, Multiplexor):

responsible for directing the flow of instructions and data within the CPU



Ref:
https://courses.cs.vt.edu/csonline/MachineArchitecture/Lessons/CPU/index.html

Scaling on Hardware Level

A multi*core* CPU has multiple execution cores on one CPU. It basically means that a certain subset of the CPU's components is duplicated, so that multiple "cores" can work in parallel on separate operations. This is called CMP, Chip-level Multiprocessing.

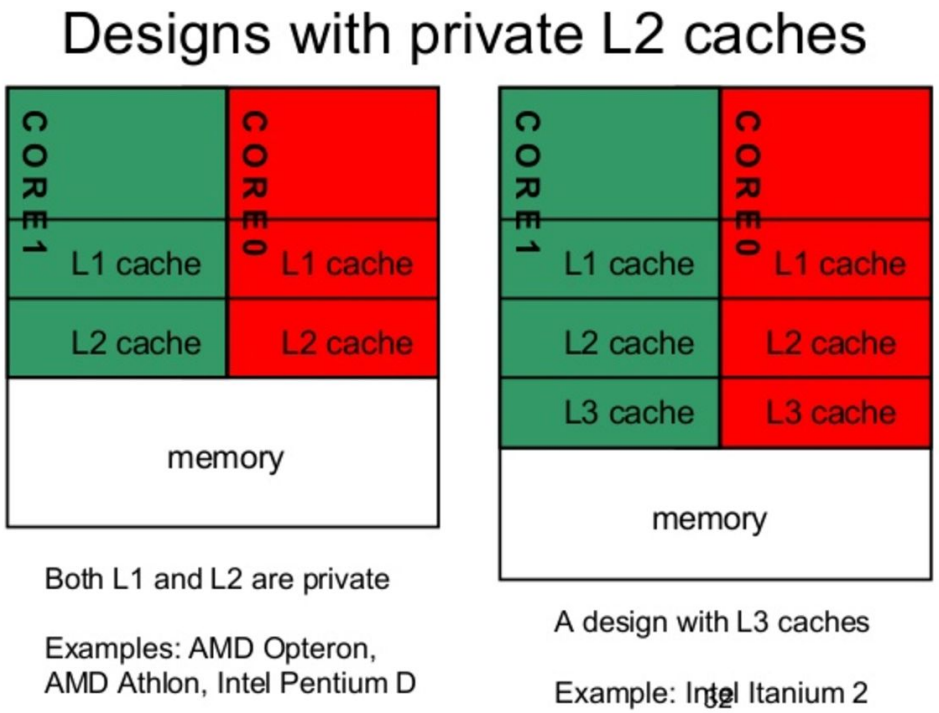
For example, a multicore processor may have a separate L1 cache and execution unit for each core, while it has a shared L2 cache for the entire processor.

That means that while the processor has one big pool of slower cache, it has separate fast memory and arithmetic/logic units for each of several cores. This would allow each core to perform operations at the same time as the others

Scaling on Hardware Level

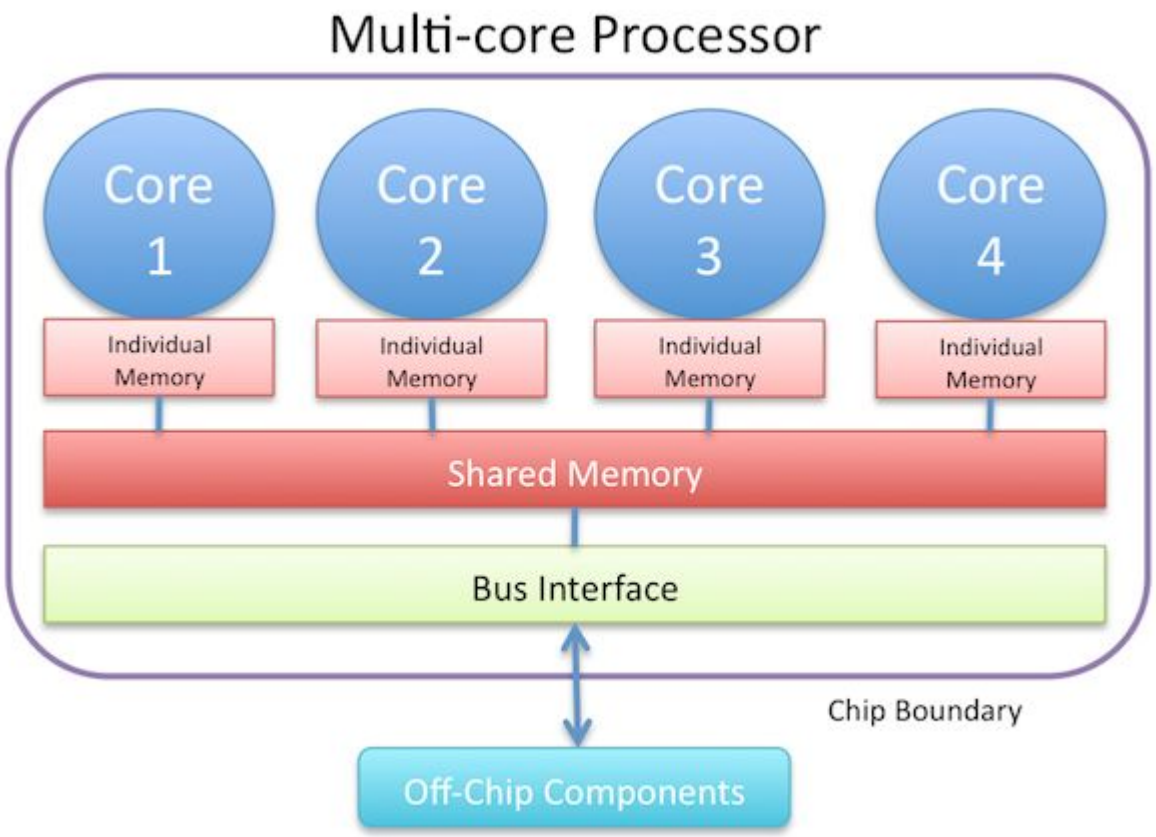
Memory model for Multi-core CPUs:

- L1 cache is always private
- L2 and L3 caches may be shared in some architectures
- Memory is always shared



Ref:
https://www.slideshare.net/piyushmittalin/multi-corearchitecture/2

Scaling on Hardware Level



Scaling on Hardware Level

Hyper-Threading

Hyper-Threading Technology is a form of simultaneous multithreading technology introduced by Intel

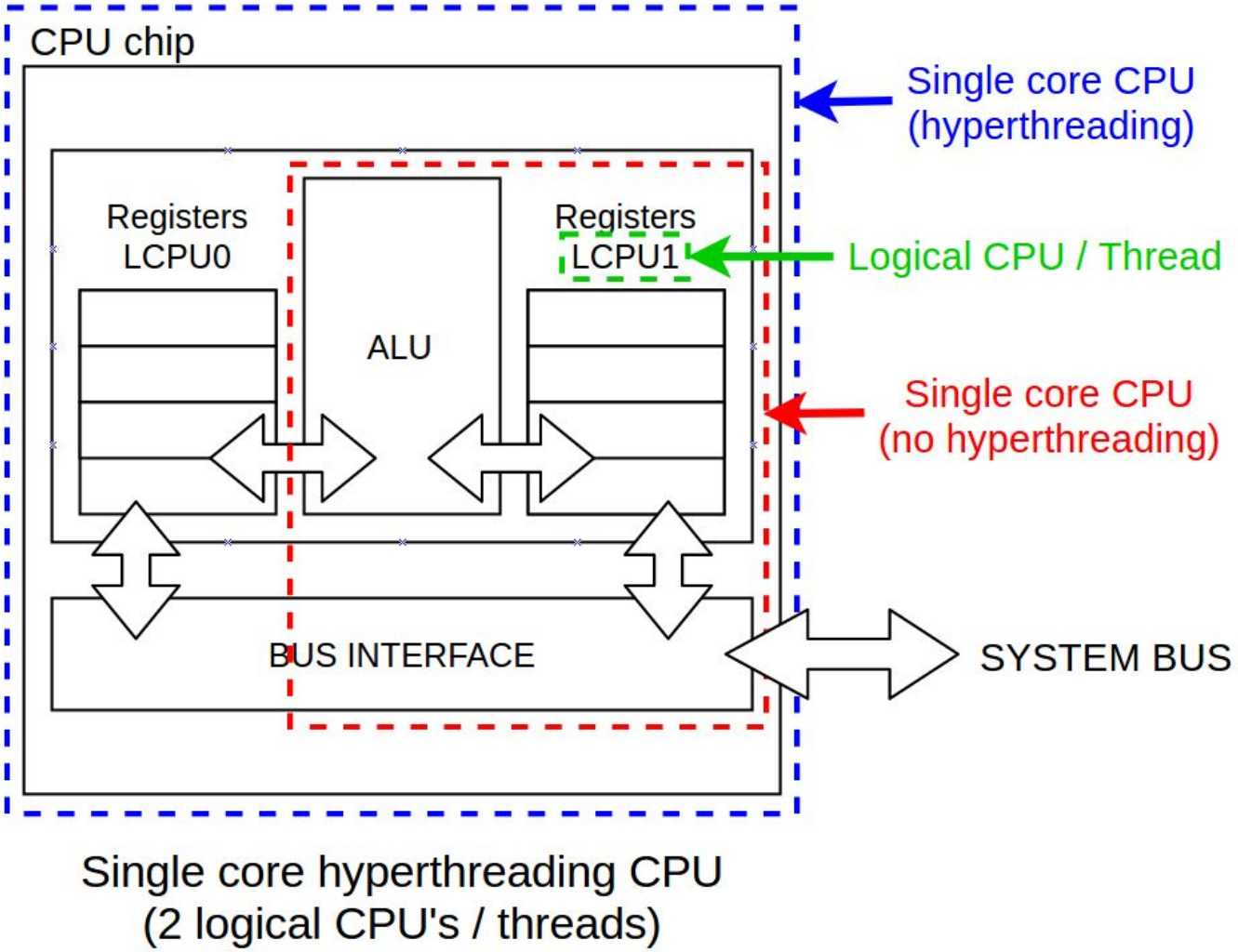
A processor with HT consists of two logical processors[cores] per physical CPU

each CORE has its own processor architectural state: Registers, local thread caches

But both cores share the execution resources: execution engine [ALU], caches, and system bus interface

Hyperthreading is far less efficient then adding a full core, since if two threads need access to the same execution resource at the same time, one of them must wait for the other to finish.

logical vs physical cores:

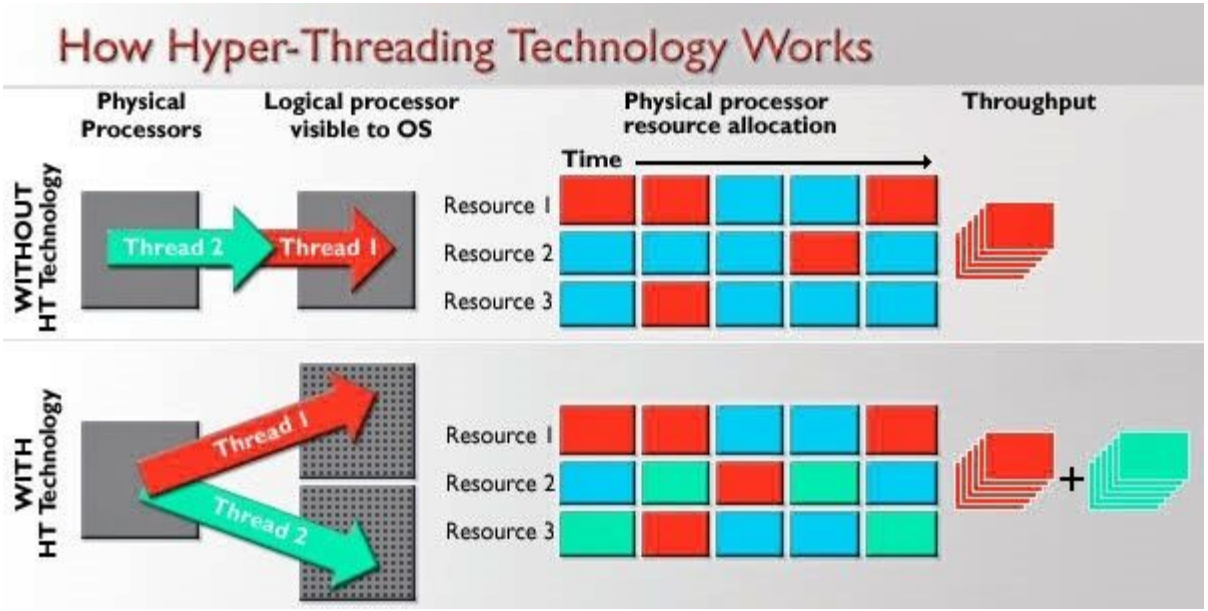


Single core hyperthreading CPU
(2 logical CPU's / threads)

Scaling on Hardware Level

Illustration of HT work:

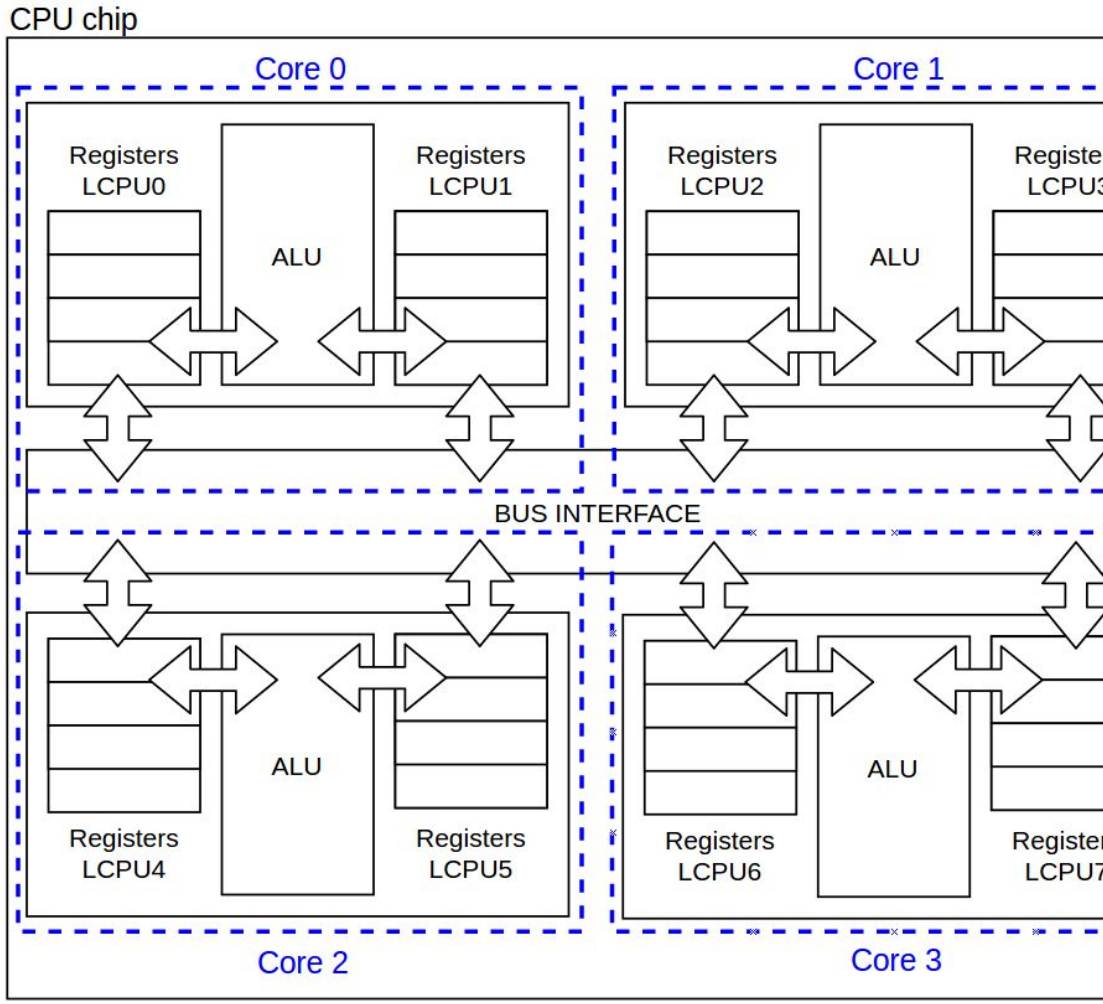
the blue rectangle represent unused capability of 1 core, and is easy to determine that with Hyper Threading the execution cores are more used in a unit of time and also finishes the tasks quickly



Scaling on Hardware

Multi-core system with HT:

Cores in such system have
faster (than physical CPUs)
communications between
them by means of a shared internal bus



Quad-core hyperthreading CPU

Scaling on Hardware Level

logical CPU vs virtual CPU vs cloud virtualization:

The virtual CPU term is comparable to logical CPU but it usually refers to those CPUs mapped to virtual machines from the underlying host hardware, which can be physical or logical cpus, HT or not.

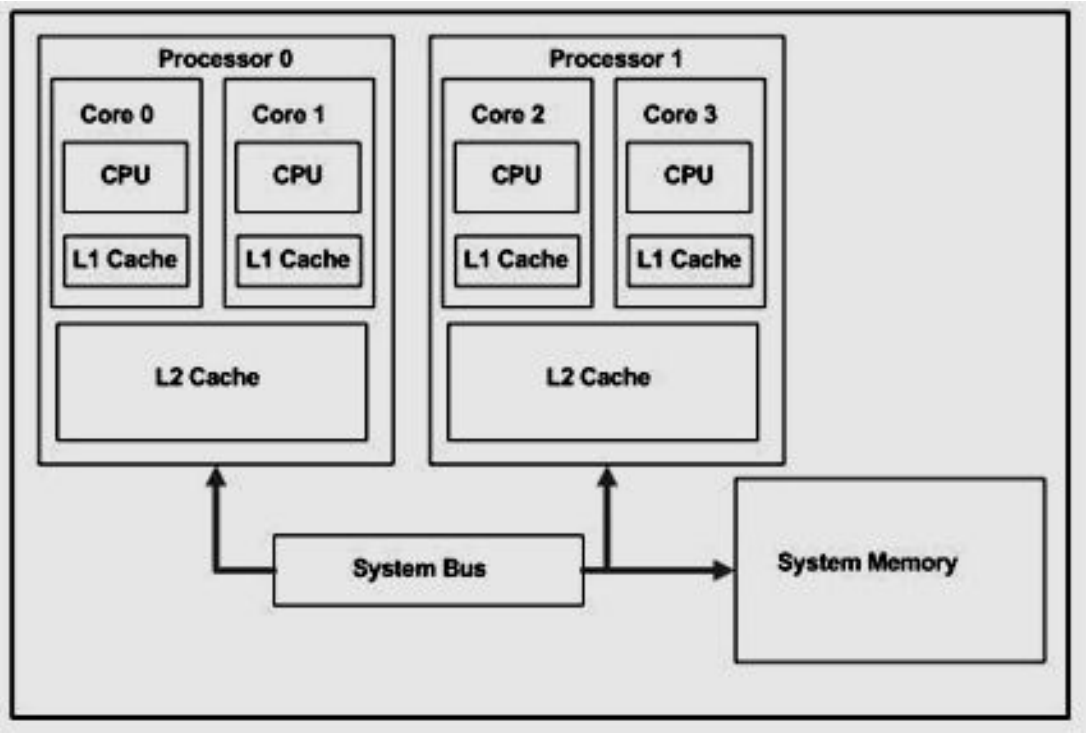
Normally 1 logical cpu from host server is mapped to 1 virtual cpu inside virtual machine, so they are almost equivalent terms.

Also, VMs can be either created on your physical hardware (like VMWare or VirtualBox VMs) or they can be cloud-based (like AWS EC2s), in which case you will have a cloud-based virtualization and horizontal scaling

Scaling on Hardware Level

Multi-processor vs multi-core architectures

Multi-processor system is a system with multiple physical processors, that communicate via System bus
However, the system must have support for a multiprocessor to work
A processor can have multiple cores - in which case It is a multi-core processor



Scaling on Hardware Level

CPU vs GPU

Ref: https://www.cse.wustl.edu/~jain/cse567-11/ftp/multcore/

- There are many **specific types of multi-core processors** serving varying functions in computing, for example Graphics Processing Units (GPUs).
- Modern GPUs have hundreds of cores, though these cores are significantly different from a CPU core.
- GPUs were primarily designed to accelerate 3D graphics, but evolved to be used to accelerate other computational applications. The term General Purpose computing/computation on GPUs (GPGPU) now refers to the use of GPUs for other applications besides traditional 3D graphics
- CPUs and GPUs are designed to be efficient at significantly different types of tasks. The architectural differences between CPUs and GPUs:
  - CPUs perform better on latency-sensitive, partially sequential, single sets of tasks
  - GPUs perform better with latency-tolerant, highly parallel and independent tasks
- Lately the most popular use of GPUs is for ML and AI areas - especially ML model trainings

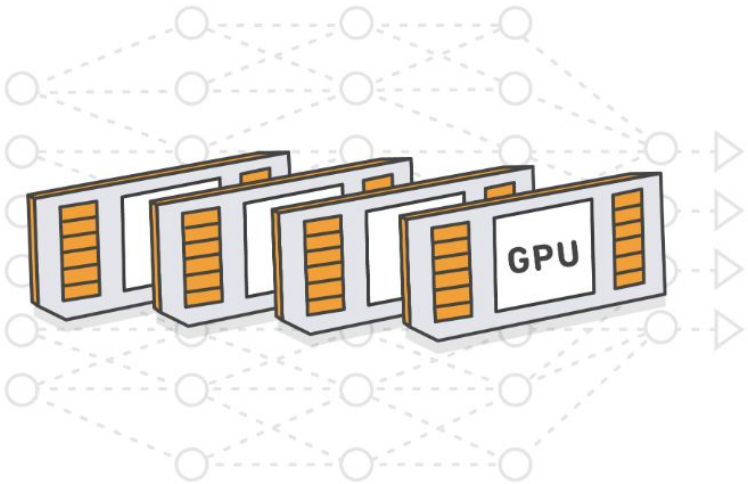
Scaling on Hardware Level

Amazon EC2 P2 Instances

Powerful, Scalable GPU instances for high-performance computing

Amazon EC2 P2 Instances are powerful, scalable instances that provide GPU-based parallel compute capabilities. For customers with graphics requirements, see G2 instances for more information.

P2 instances, designed for general-purpose GPU compute applications using CUDA and OpenCL, are ideally suited for machine learning, high performance databases, computational fluid dynamics, computational finance, seismic analysis, molecular modeling, genomics, rendering, and other server-side workloads requiring massive parallel floating point processing power.



Moving on .... to the Software Level Scaling

@Marina Popova

Scaling on Software Level

Processes and Threads

- Most operating systems represent applications as **processes**. This means that the application has its own address space (== view of memory), where the OS makes sure that this view and its content are isolated from other applications.
- A process consists of one or more **threads**, which carry out the real work of an application by executing machine code on a CPU. The operating system determines, which thread executes on which CPU
- threads and processes can run concurrently on multi-core CPUs. **A single process or thread only runs on a single core at a time.** If there are more threads requesting CPU time than available cores (generally the case), the operating system scheduler will move threads on and off cores as needed.

- OS doesn't much care which process the threads are from. It will usually schedule threads to processors / cores regardless of which process the thread is from. This could lead to four threads from one process running at the same time, as easily as one thread from four processes running at the same time.

Lets look at both Processes and Threads in details

Scaling on Software Level

What is a Process?
A process is an executing instance of a computer program. There may be multiple copies of the same program running simultaneously.
Process layout when loaded into memory for execution:

- **Text section** contains compiled code of the program logic.
- **Data** section stores global and static variables.
- **Heap** section contains dynamically allocated memory (ex. when you use malloc or new in C or C++).
- **Stack** section stores local variables and function return values
- **PCB:** the OS maintains a special table called **process control block (PCB)** to keep track of process state. PCB table contains various information about the process, the main two items are: **program counter (PC) and CPU registers.**
- PC points to the next instruction to execute
- CPU registers hold temporary execution information such as instruction arguments.

Scaling on Software Level

What is a Thread?

- A thread is a flow of execution through the process code
- From an OS perspective, just like a process, a thread has a private stack, program counter and a set of CPU registers.
- A thread is also called a **lightweight process** because it shares code, data, heap and open files with the parent process
- Each thread belongs to exactly one process and no thread can exist outside a process. Each thread represents a separate flow of control.

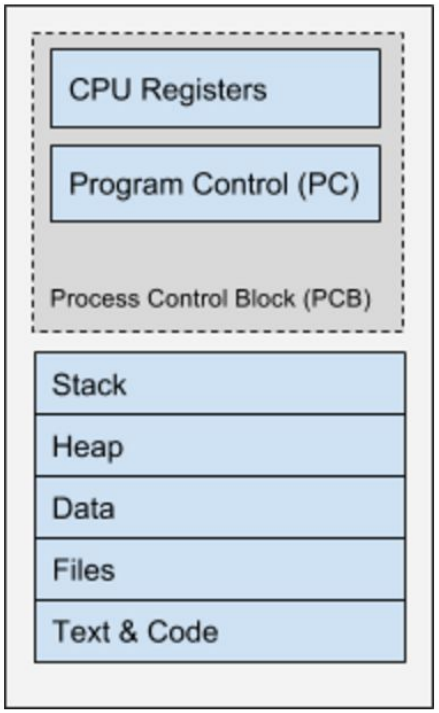
Stack vs Heap:

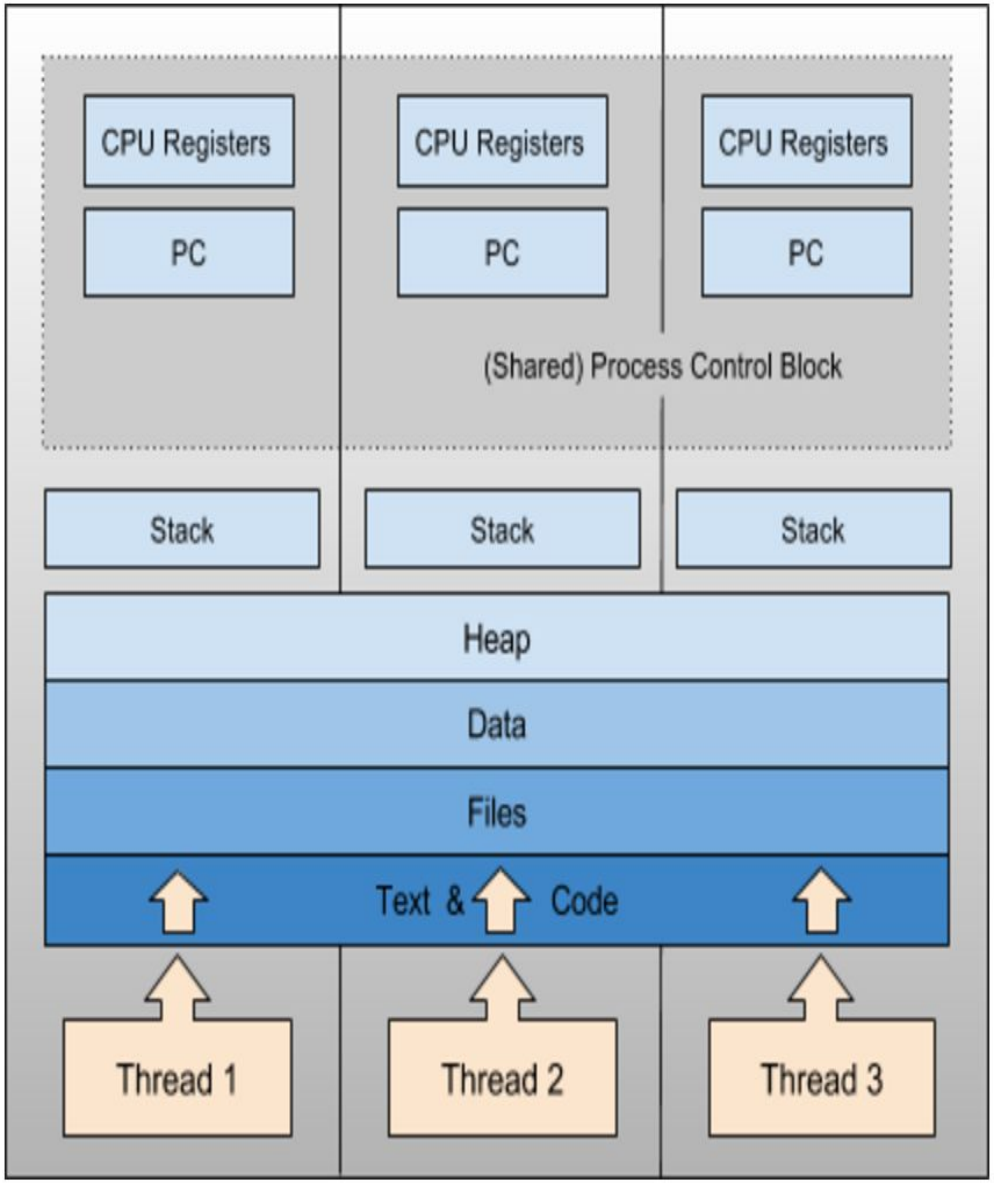
Stack frames: only exist during the execution time of a function, and contains only data needed for program execution and function calls (like parameters, return address, local vars)

Heap: an extra storage that is totally independent of the run-time stack of the program and can be access at any point, whether the actual thread is executing on the core or not

https://medium.com/fhinkel/confused-about-stack-and-heap-2cf3e6adb771

One process, one thread memory layout vs. one process, three threads





Scaling on Software Level

Context Switching

In a single processor system, there is no real multitasking. CPU time is shared among running processes. When the time slice for a running process expires, a new process has to be loaded for execution. **Switching from one process or thread to another is called context switch**.

Process context switch involves saving and restoring process state information including program counter, CPU registers and process control block which is a relatively expensive (in terms of CPU time) operation.

Similarly, **thread context switch** involves pushing all thread CPU registers and program counter to the thread private stack and saving the stack pointer. Thread context switch compared to process context switch is relatively cheap and fast as it only involves saving and restoring CPU registers.

Scaling on Software Level

There are different types of threads, which is defined by the thread management type:

User level threads are managed by a user level library and
Kernel (CPU core) level threads are managed by the operating system kernel code.
To better understand the difference between user level code and kernel level code, refer to the following article:
http://www.8bitavenue.com/2015/07/difference-between-system-call-procedure-call-and-function-call/

User Level Threads
Even though user level threads are managed by a user level library, they still require a kernel system call to operate.
The kernel does not have to and usually does not know anything about the user thread management. It only takes care of the execution part.

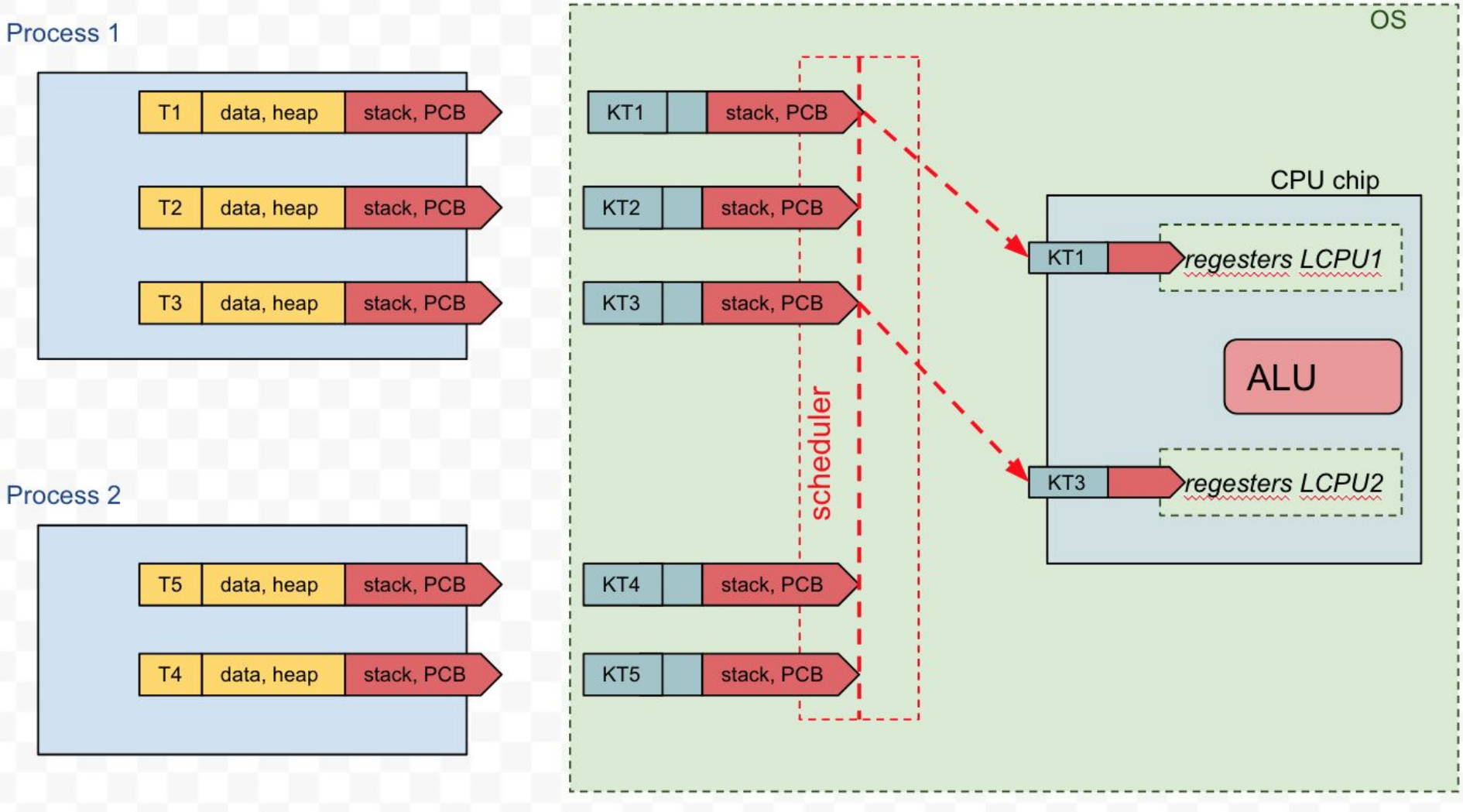
User level threads are typically fast. Creating threads, switching between threads and synchronizing threads only needs a procedure call. They are a good choice for non blocking tasks otherwise the entire process will block if any of the threads blocks.

Scaling on Software Level

Kernel Level Threads

- Kernel level threads are managed by the OS, therefore, thread operations (ex. Scheduling) are implemented in the kernel code. This means kernel level threads may favor thread heavy processes.
- **kernel threads can utilize multiprocessor systems by splitting threads on different processors or cores**
- They are a good choice for processes that block frequently. If one thread blocks it does not cause the entire process to block.
- Kernel level threads are **slower** than user level threads due to the management overhead. Kernel level context switch involves more steps than just saving some registers.
- Finally, they are **not portable** because the implementation is operating system dependent.

Threads-to-Core Journey



Scaling on Software Level

How do User Threads map to the Kernel Threads??

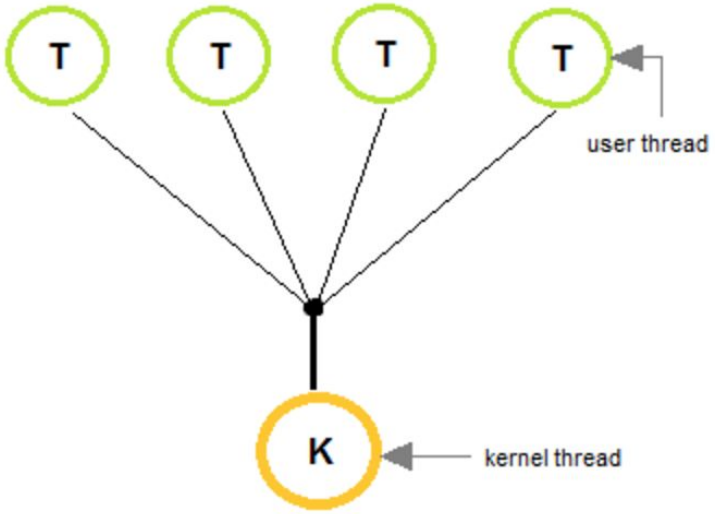
Ref: http://www.studytonight.com/operating-system/multithreading

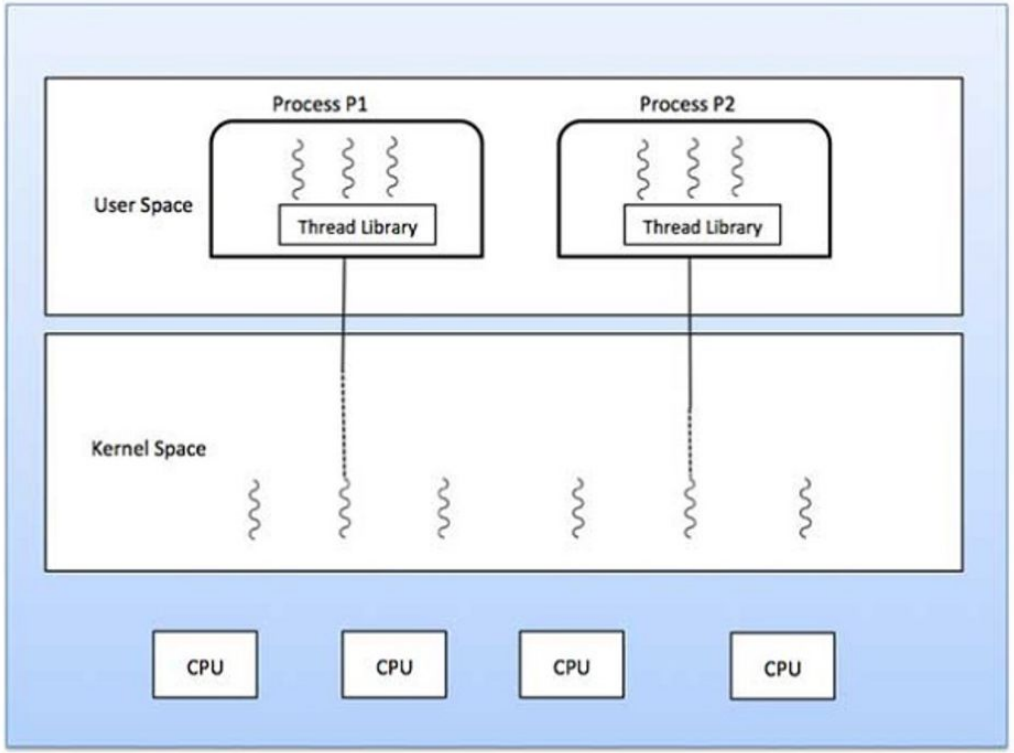
Multithreading Models
The user threads must be mapped to kernel threads, by one of the following strategies:

- Many-To-One Model:
- One-To-One Model
- Many-To-Many Model

Many-To-One Model

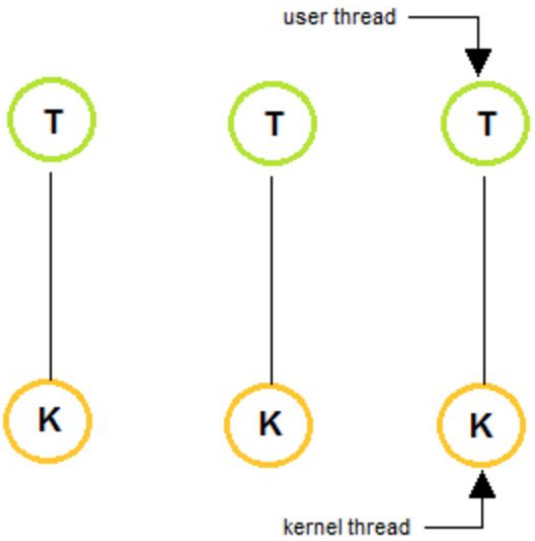
- many user-level threads are all mapped onto a single kernel thread
- Thread management is handled by the thread library in user space, which is efficient in nature.

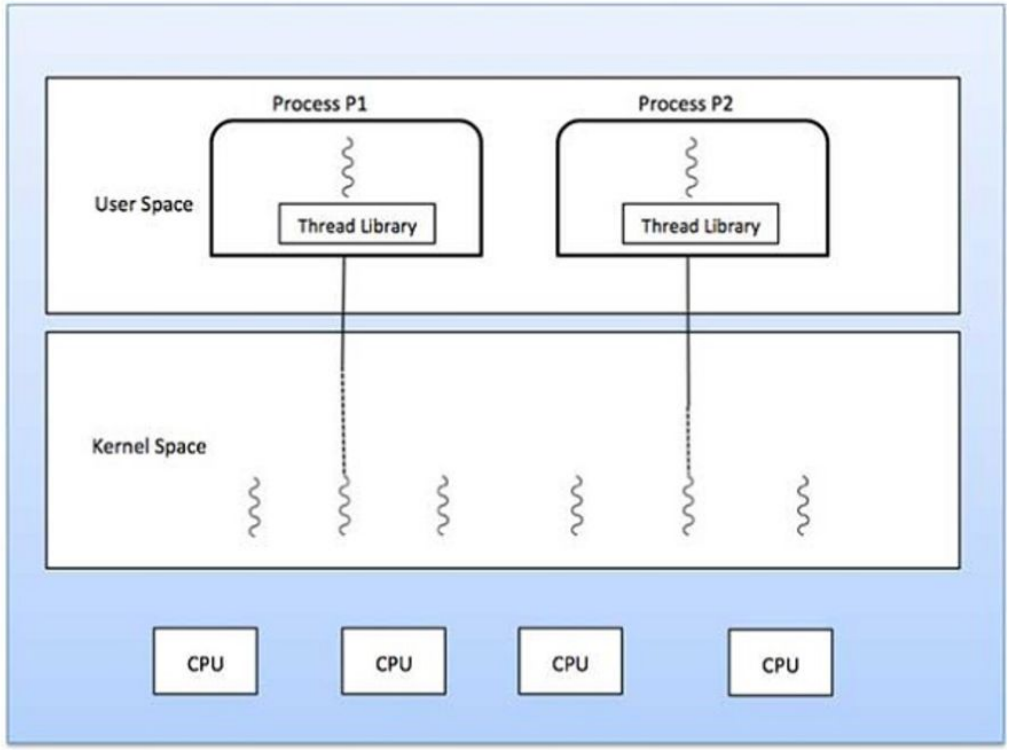




One-To-One Model

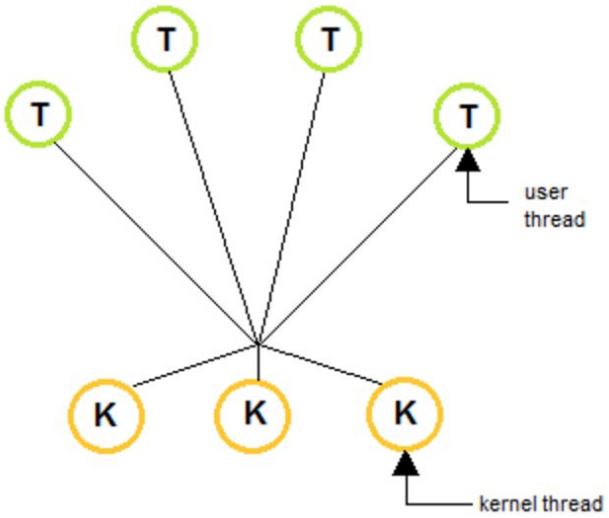
- The one-to-one model creates a separate kernel thread to handle each and every user thread.
- Most implementations of this model place a **limit on how many threads can be created**.
- Linux and Windows from 95 to XP implement the one-to-one model for threads

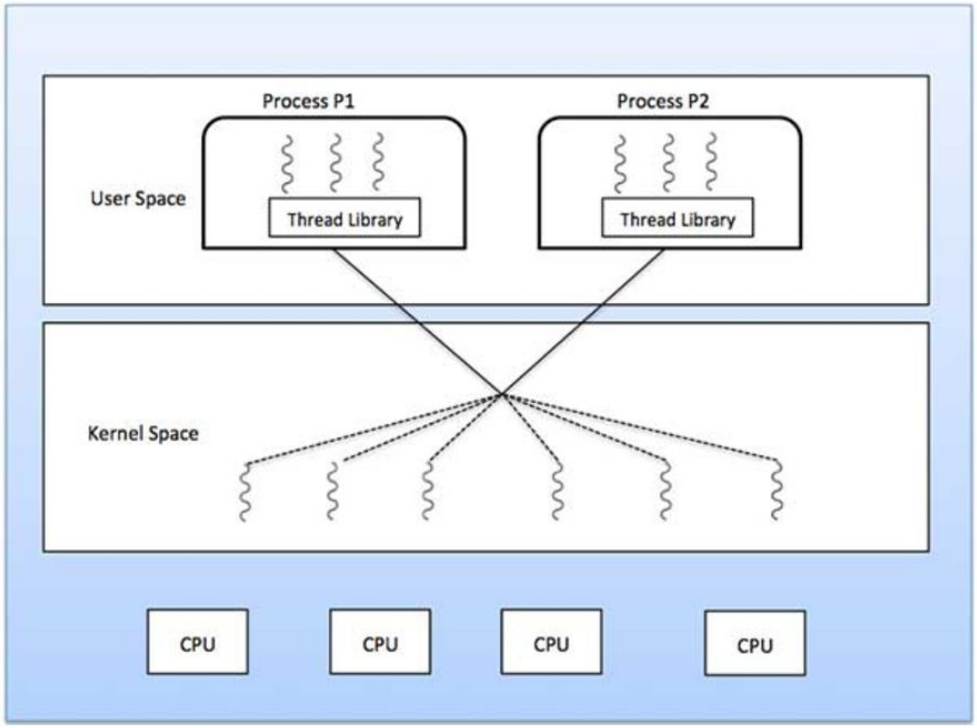




Many-To-Many Model

- The many-to-many model multiplexes any number of user threads onto an equal or smaller number of kernel threads, combining the best features of the one-to-one and many-to-one models.
- Users can create any number of the threads.
- Blocking the kernel system calls does not block the entire process.
- Processes can be split across multiple processors





Scaling on Software Layer

Most implementations have converged to 1:1 model since it utilizes the server's processing power and relieves the task of scheduling from the languages and libraries.

Types of Thread Libraries
There are three main thread libraries in use today:

1. POSIX Pthreads - may be provided as either a user or kernel library, as an extension to the POSIX standard.
2. Win32 threads - provided as a kernel-level library on Windows systems.
3. Java threads - Since Java generally runs on a Java Virtual Machine, the implementation of threads is based upon whatever OS and hardware the JVM is running on, i.e. either Pthreads or Win32 threads depending on the system.

The **Native POSIX Thread Library** (**NPTL**) is an implementation of the POSIX Threads specification for the Linux operating system.

Illustration of Multithreading in Java

- All Java applications use threads. If no explicit threads are created/used in your application - there will be at least one Main thread of execution which will be scheduled by OS to use one of the cores. There maybe more threads - if some frameworks that use background threads are used
- On Linux, **Java threads are implemented with native threads**, so a Java program using threads is no different from a native program using threads. A "Java thread" is just a thread belonging to a JVM process
- On a modern Linux system (one using NPTL), **all threads belonging to a process have the same process ID and parent process ID, but different thread IDs.** You can see these IDs by running ps -eLf. The PID column is the process ID, the PPID column is the parent process ID, and the LWP column is the thread (LightWeight Process) ID. The "main" thread has a thread ID that's the same as the process ID, and additional threads will have different thread ID values.

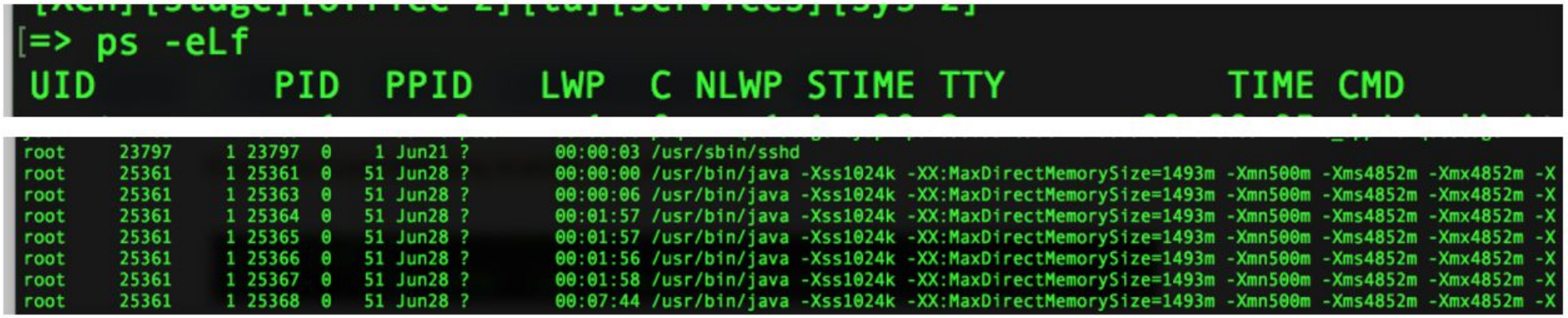


Illustration of Multithreading in Java

- Older Linux systems may use the "linuxthreads" threading implementation, which is not fully POSIX-compliant, instead of NPTL. On a linuxthreads system, threads have different process IDs

- the JVM runs on top of a native OS, and the JVM specification does not specify what model to use for mapping Java threads to kernel threads. This decision is JVM implementation dependant, and may be one-to-one, many-to-many, or many to one.. (On a UNIX system the JVM normally uses PThreads and on a Windows system it normally uses windows threads.)

- Linux kernel uses a 1:1 mapping model. Since JVM only maps a Java Thread to a native OS thread, on Linux JVM is also following 1:1 model.

Illustration of Multithreading in Java

how to determine the right number of threads for your app:

Runtime.availableProcessors() - returns the number of available processors on the server

- The number of processors is basically the number of execution engines [ALU] capable of running your code. These may be physically distinct processors (even though they exist inside the same chip) or they may be logical processors when you're using hyper-threading.
- If you have a quad-processor (4 cores) with no hyper-threading - the result of the function call will be 4
- If you have hyper-threading enabled - the result will be 8

Illustration of Multithreading in Python and Java

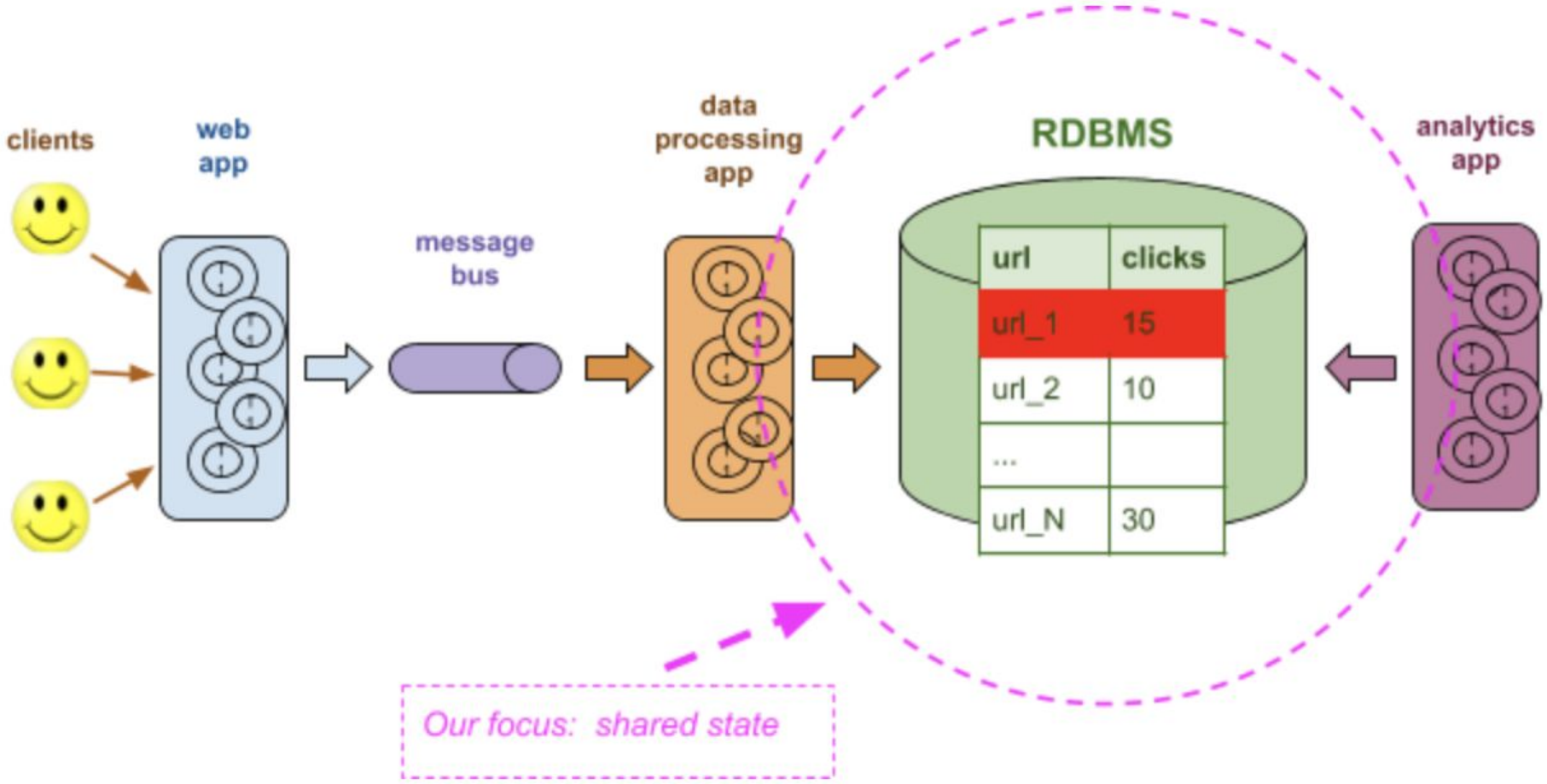
Python and Java: Thread Basics - will be covered in the Lab
Refs:
http://javabeginnerstutorial.com/core-java-tutorial/java-thread-tutorial/
https://www.geeksforgeeks.org/multiprocessing-python-set-2/
https://eli.thegreenplace.net/2012/01/04/shared-counter-with-pythons-multiprocessing

- Python: baic threads and multi-processing
- Low-level thread programming
  - Thread, Runnable
- Java Concurrency lib
  - ExecutorService, Callable, Futures
- Java 8 Streams Parallelization - will NOT be covered - this will be one of topics covered in the CSCIE-88A class in Spring (Functional and Stream Programming)

Moving on .... to Shared State Management

@Marina Popova

Shared State Management



Shared State Management

Why is this important?

- To speed up data processing - you parallelize your applications
- This means that input data has to be distributed between parallel threads/processes
- And the results from all parallel processes have to be collected

Shared data vs shared state

- shared state: results of processing or computation - often used to calculate the next set of values
- shared data: input for your application either static (like meta-data from some SQL DB, data files) or streaming input from a messaging system or something like Twitter

We will focus on the Shard Date part later in the class

Shared State Management

We will now consider how an application that processes web logs and answers some pre-defined queries can be designed and developed

Our original "Online Store Page Analytics" application was processing streaming data - incoming logs from web servers. For simplicity, we will assume that our app will be **reading data from a finite set of static data (files) for now**

We will discuss how the design and implementation of such application evolves as you are scaling it out - and this time we will be **focusing on the shared state handling**

Where does it fall in the Big Picture Map of the data processing pipelines?

- applicable to any batch processing - either in the large-scale or micro-batch context
- applicable to all types of applications (Type 1 - 4)

Shared State Management

As the first step we will be calculating and storing data is SHARED COUNTERS

Why not storing raw data?

- much faster query execution time
- results are not STALE
- suitable for fast stream processing use cases as well as batch pre-processing
- much smaller volume of data to store

This will change as we go into much larger volumes of data and AD-HOC queries (Type 4 systems)

Shared State Management

Example 1:
first attempt at getting click and visitor counts for our Page Analytics application, **using shared state**

Input: files with events - one event per user click on/ open of a URL
In the format: <URL>, <userID>,<timestamp>

Query 1: get count of unique URLs
Query 2: get count of unique visitors (userIDs) per URL
Query 3: get count of unique (by userId and timestamp) clicks per URL

As a side note: How click "uniqeness" is determined: https://help.tune.com/marketing-console/clicks-versus-unique-clicks/

Example data and results

Query 1: count of unique URLs: **3**

Query 2: count of unique visitors (userIDs) per URL:

www.google.com : 3 [u1, u2, u3]

www.me1.com/p1 : 1 [u1]

www.me2.com/p1 : 1 [u4]

Query 3: count of unique clicks per URL:

www.google.com :

u1 : 2

u2 : 3

u3 : 2

www.me1.com/p1 :

u1 : 4

www.me2.com/p1 :

u4 : 2

Shared State Management

single instance multi-threaded application

- What is a shared state in this case?
  It is intermediate results of processing our input data
- Where is it stored?
  shared memory (in-process cache)
- Data structure - **option 1: single Map**:
  Key: url + userId
  Value: count of clicks by this userId on this url
  { "url, userId" --->>> clickCount}

  Query 3: (count of unique clicks) - good!
  Query 1 and 2: very inefficient -- why ??

For example:

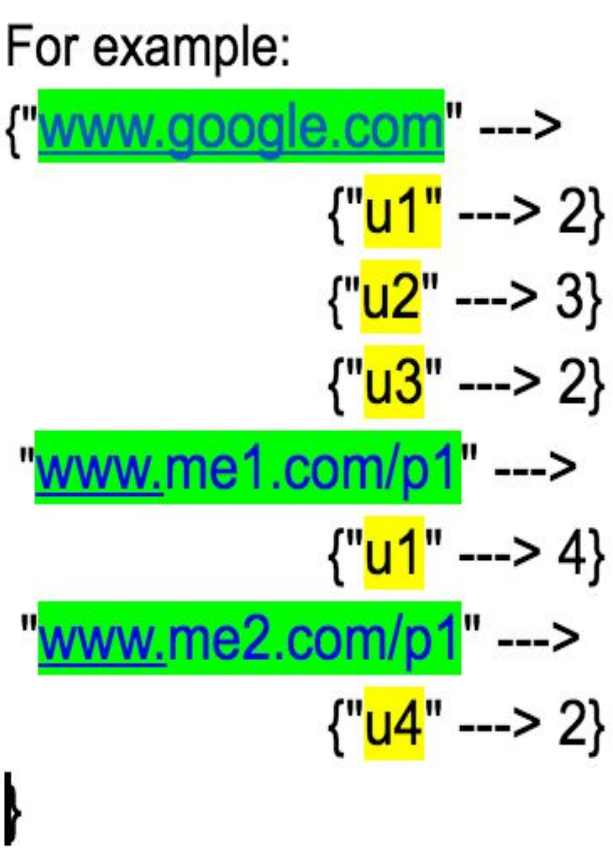
{"www.google.com, u1" ---> 2}
{"www.google.com, u2" ---> 3}
{"www.google.com, u3" ---> 2}
{"www.me1.com/p1, u1" ---> 4}
{"www.me2.com/p1, u4" ---> 2}

Shared State Management

single instance multi-threaded application

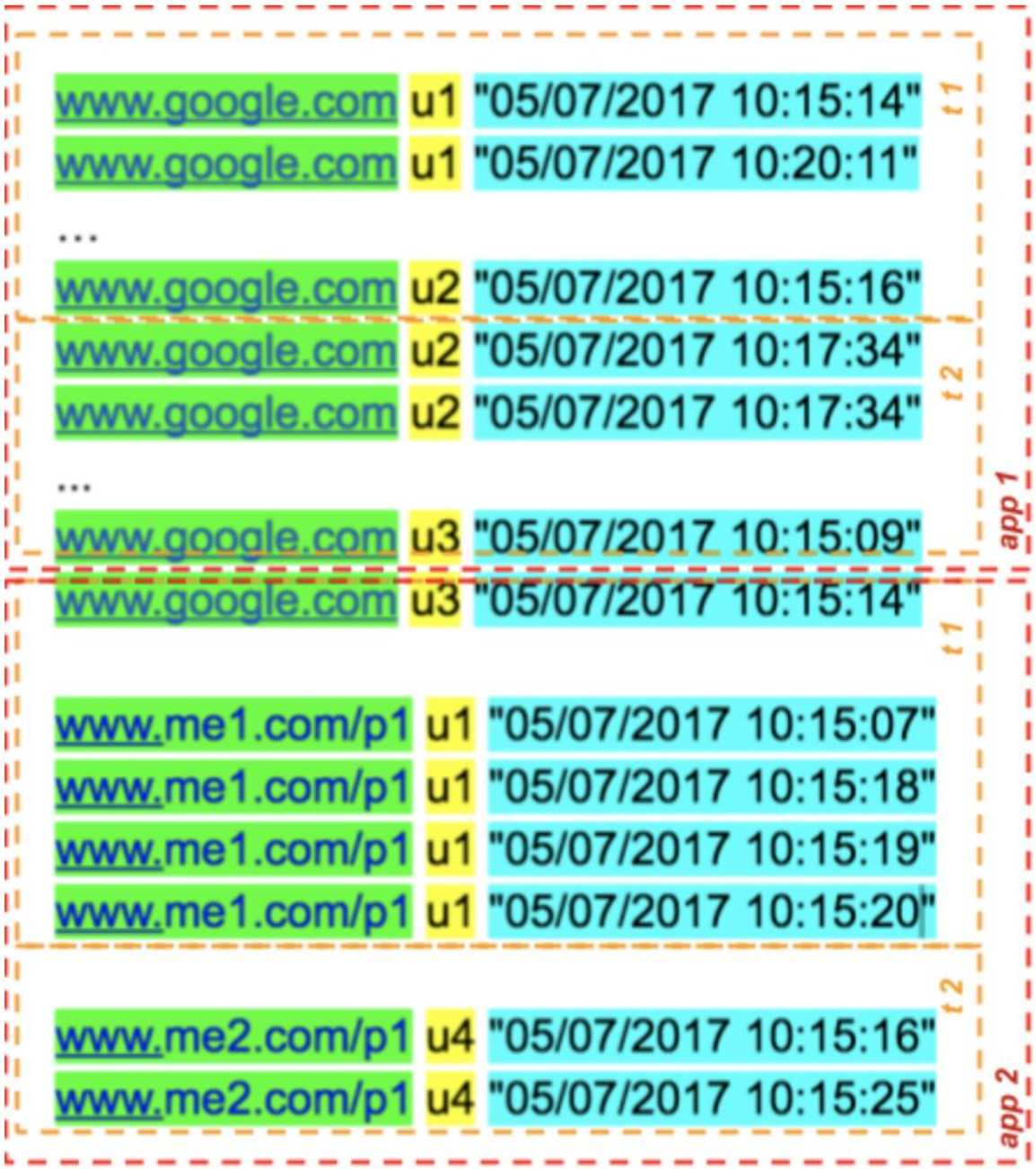
Data structure - **option 2: Map of Maps**:
{ "url" --->>> {"userId" --->>> clickCount} }

Getting results:
Query 1: (count of unique URLs)
size of the main map
Query 2: (count of unique visitors per URL)
size of the secondary map for specific URL
Query 3: (count of unique clicks per URL)
value from the secondary map



Shared State Management

multiple app instances - one machine:



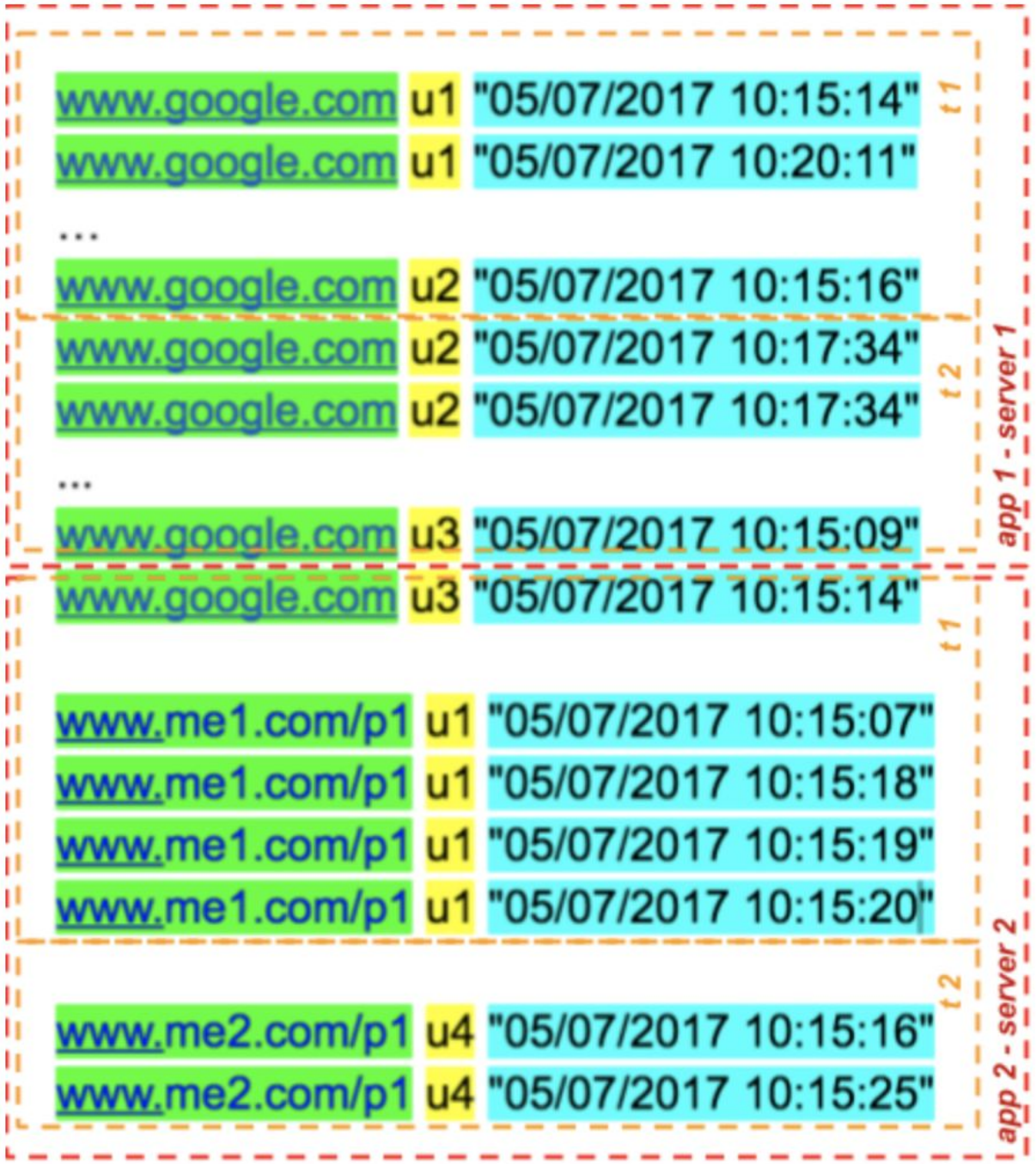
Shared State Management

multiple app instances - one machine
Shared State Options?

- Option 1: in-process cache (severe limitations)
  - cached objects scoped to the memory of each individual app ; could work only if there is no need to share counters but this is not what our model does in this approach (this is exactly what MR does…)
- Option 2: local FS
  - doable, but very bad option - have to lock for access from multiple apps/threads
- Option 3: shared DB (MySQL, Oracle, etc.)
  - supports full SQL and ACID
- Option 4: local or remote/distributed cache service - data is totally in memory (Redis, Memcached, EHCache, Confluence/Oracle)
  - faster than DB, usually
  - can configure disk-based persistence at the performance cost….
- Option 5: NoSQL key/value storage

Shared State Management

multiple app instances - multiple machines:



Shared State Management

multiple apps on multiple machines
Shared State Options?

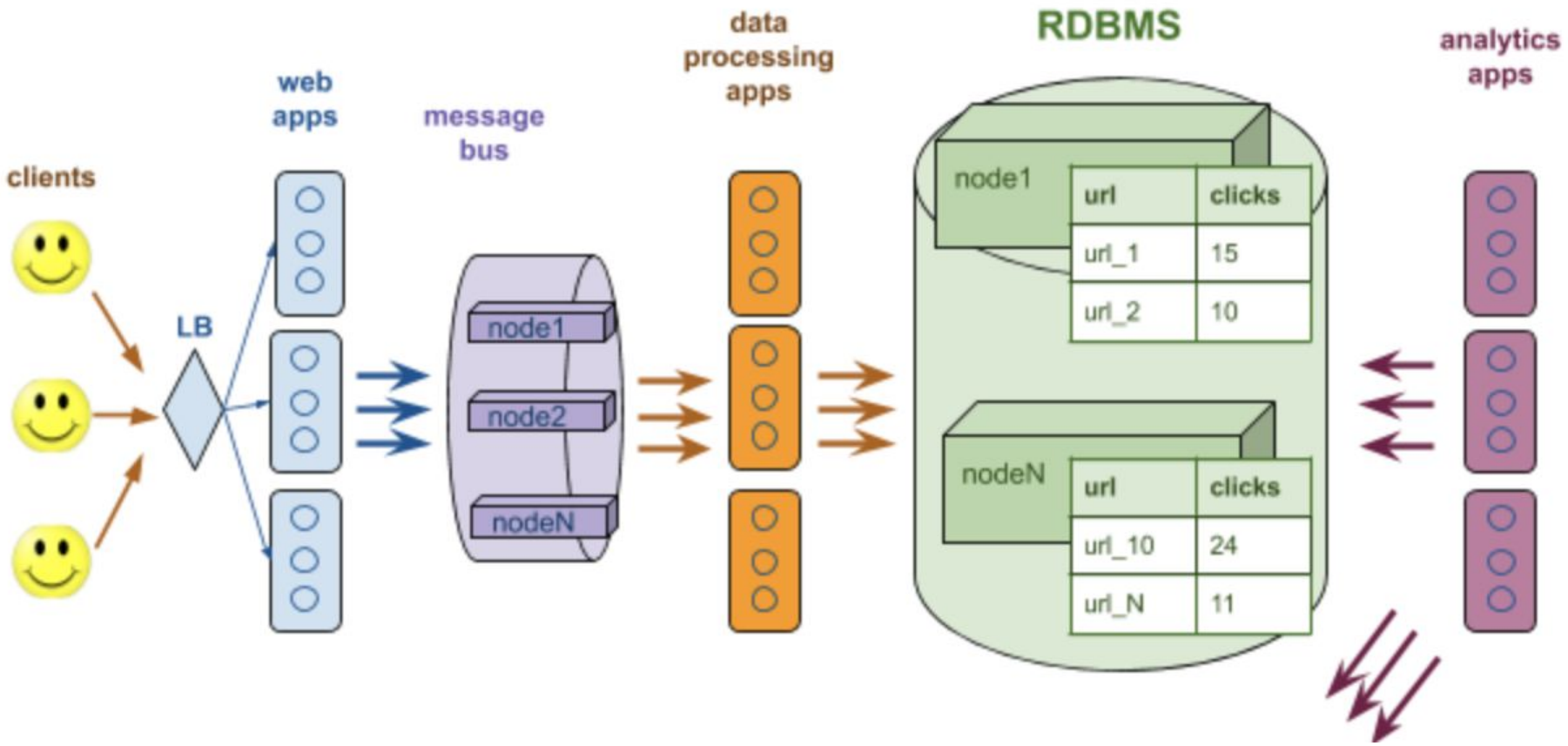
- Option 1: in-process cache - same limitations is in the use case above
- Option 2: distributed FS (local FS is not an option anymore)
  - Example - HDFS, S3
  - Issues with mutable objects - so it will not work for this Approach (using shared mutable data)
  - One of the best options for immutable data - will cover HDFS in more details in the next Lecture
- Option 3: shared DB
- Option 4: Distributed Cache service
- Option 5: NoSQL key/value storage

Cache Service is often a better option than a shared RDBMS !

Why? coming soon ....

Shared State Management

Lets consider Option 3 (Shared DB) to hold our shared state first:



Shared State Management

Option 3 (Shared DB) - Example schema:

url (PK)	user_id (PK)	click_count
www.google.com	u1	2
www.google.com	u2	3
www.google.com	u3	2
www.me1.com/p1	u1	4
www.me2.com/p1	u4	2

Shared State Management

To manage counters, we could use SELECT … FOR UPDATE like functionality of SQL DBs - to do atomic increments of counts;
example for MySQL DB (from MySQL documentation):

To implement reading and incrementing the counter, first perform a locking read of the counter using FOR UPDATE, and then increment the counter. For example:

```
SELECT click_count FROM url_user_clicks FOR UPDATE
    WHERE url = $url and user_id = $userId;
UPDATE click_count SET click_count = click_count + 1;
```

A SELECT ... FOR UPDATE reads the latest available data, setting exclusive locks on each row it reads. Thus, it sets the same locks a searched SQL UPDATE would set on the rows.

Shared State Management

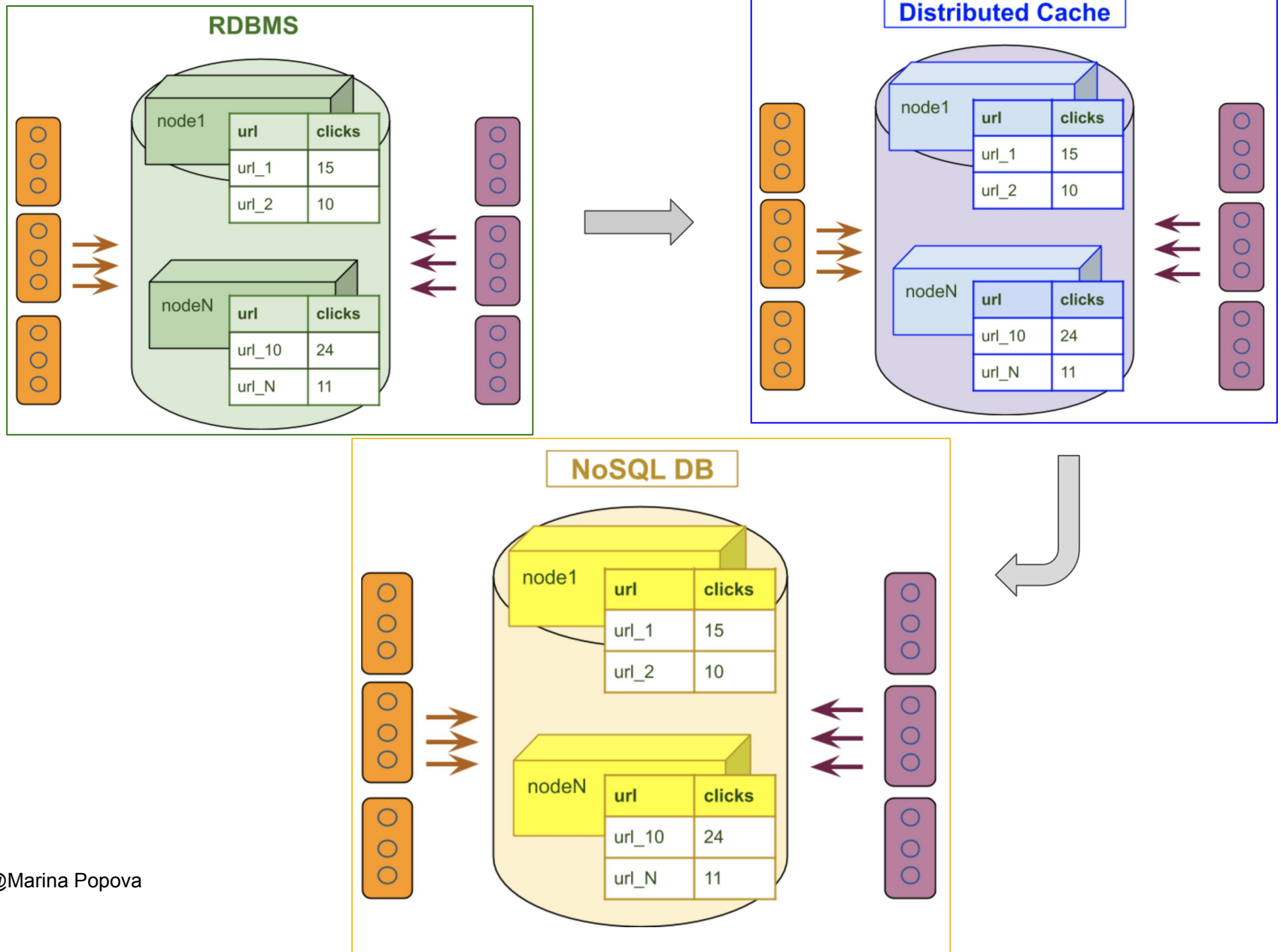
Another SQL approach is to use "ON DUPLICATE KEY UPDATE" functionality, which requires only one command:

```
INSERT INTO URL_USER_CLICKS (url,user_id,click_count) VALUES ($url,$user_id,$click_count)
ON DUPLICATE KEY UPDATE click_count = click_count + $click_count
```

Both approaches suffer from the same major issue:

Single row lock contention

Shared State Management - next options:



Distributed Cache

RDBMS vs Distributed Cache

- fully indexed vs key-value storage
- on-disk only vs in-memory persistence
- ACID support vs. key/value only semantics

Distributed Cache vs full-blown NoSQL:

- performance - read/writes - while supporting:
  - data replication
  - data consistency

Distributed Cache

Use cases when DC can be very beneficial for handling mutable shared state:

- data loss is tolerable (can be treated as volatile cache) - can disable replication → very fast access to shared mutable data
- Low key contention - single row contention is not degrading cache performance (many DCs are optimized for that)
- Speed is extremely important
- persistence on disk is not required

When is this viable?

- Streaming layer: 100% accuracy is not required, but fast access to immediate results is!
  - Type 1: Real-time [Limited range] + Ad-hoc queries systems
- Batch layer: when required input data can be replayed to re-compute in-memory data (if data loss occurred and latency is Ok)
  - Type 3 and Type 4 systems

Distributed Cache

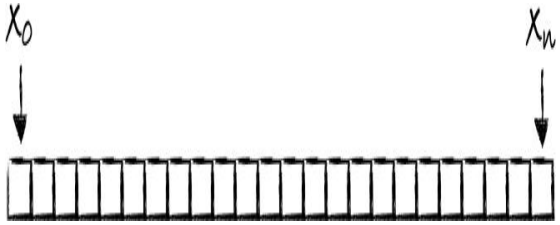
Example Distrbuted Cache frameworks: Redis, memcached, Terracotta/EHCache, Elasticache(AWS)

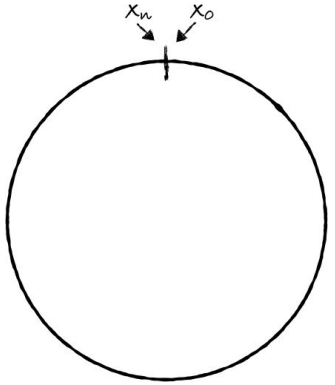
How do they work?

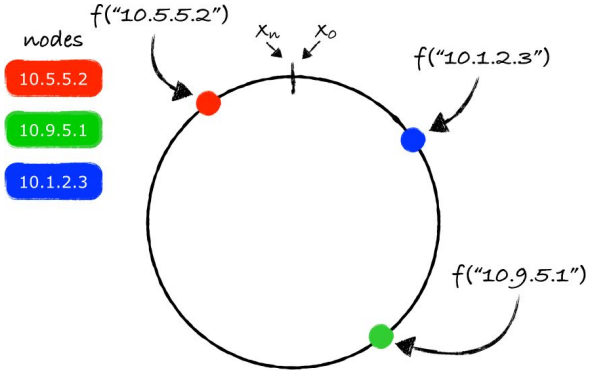
Consistent Hashing!
http://blog.carlosgaldino.com/consistent-hashing.html

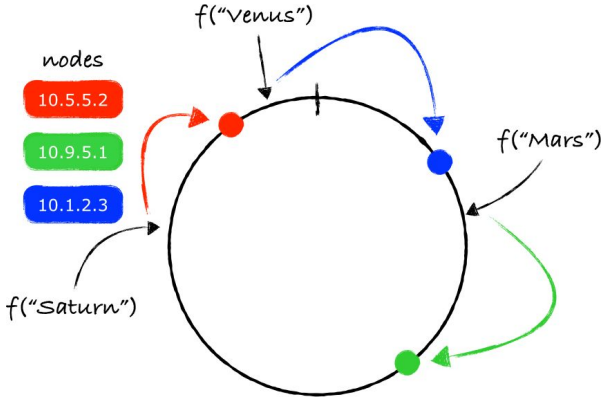
Consistent Caching

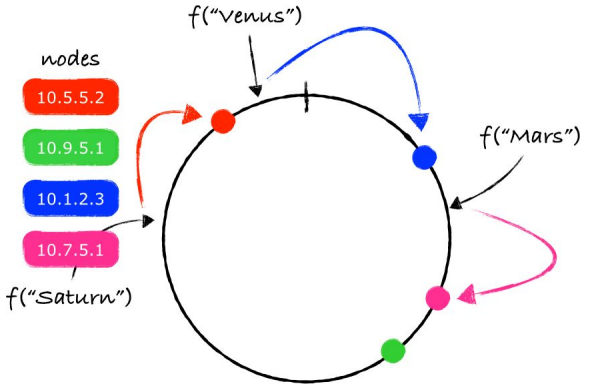
Range of values of hash function f:











Redis as Distributed Cache

Redis:

- used as a cache: **scaling up and down** using consistent hashing
- used as a store:
  - **a fixed keys-to-nodes map is used, so the number of nodes must be fixed and cannot vary**
  - OR: have to use Redis Cluster - to be able to rebalance keys between nodes when nodes are added or removed

Why Redis?

- Very fast atomic operations on different shared data structures
- Not just key-value store - reacher types are supported
- Blazing fast for volatile data storage and caching