Distributed Systems
CS6421
Execution Environments

Prof. Roozbeh Haghnazar

Slides Credit:

Prof. Tim Wood and Prof. Roozbeh Haghnazar

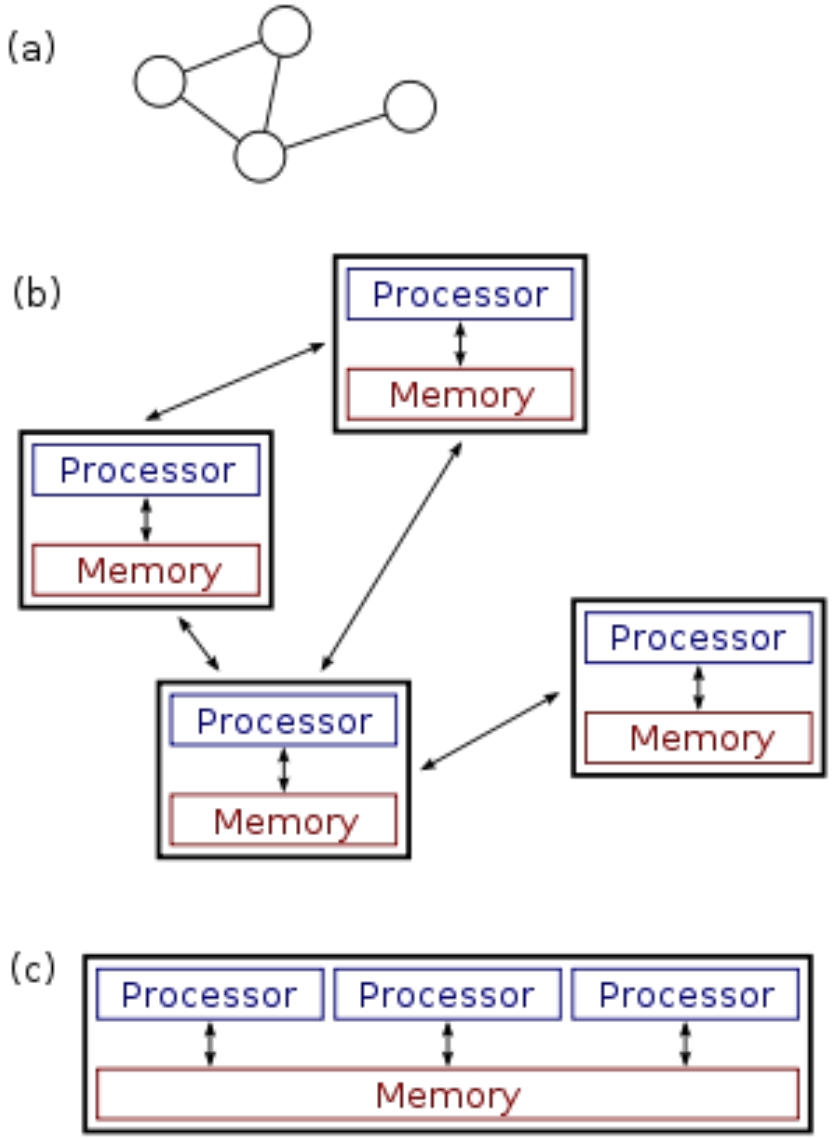
Includes material adapted from Van Steen and Tanenbaum's Distributed Systems book

Cloud Computing vs Distributed Computing

- Distributed computing is the use of distributed systems to solve single large problems by distributing tasks to single computers in the distributing systems. On the other hand, cloud computing is the use of network hosted servers to do several tasks like storage, process and management of data.

Cloud Computing vs Distributed Computing

- Figure (a) is a schematic view of a typical distributed system; the system is represented as a network topology in which each node is a computer and each line connecting the nodes is a communication link.
- Figure (b) shows the same distributed system in more detail: each computer has its own local memory, and information can be exchanged only by passing messages from one node to another by using the available communication links.
- Figure (c) shows a parallel system in which each processor has a direct access to a shared memory.



Cloud Computing vs Distributed Computing

- Cloud computing usually refers to providing a service via the internet. This service can be pretty much anything, from business software that is accessed via the web to off-site storage or computing resources whereas distributed computing means splitting a large problem to have the group of computers work on it at the same time.
  - YouTube is the best example of cloud storage which hosts millions of user uploaded video files.
  - Picasa and Flickr host millions of digital photographs allowing their users to create photo albums online by uploading pictures to their service's servers.
  - Google Docs is another best example of cloud computing that allows users to upload presentations, word documents and spreadsheets to their data servers. Google Docs allows users edit files and publish their documents for other users to read or make edits.

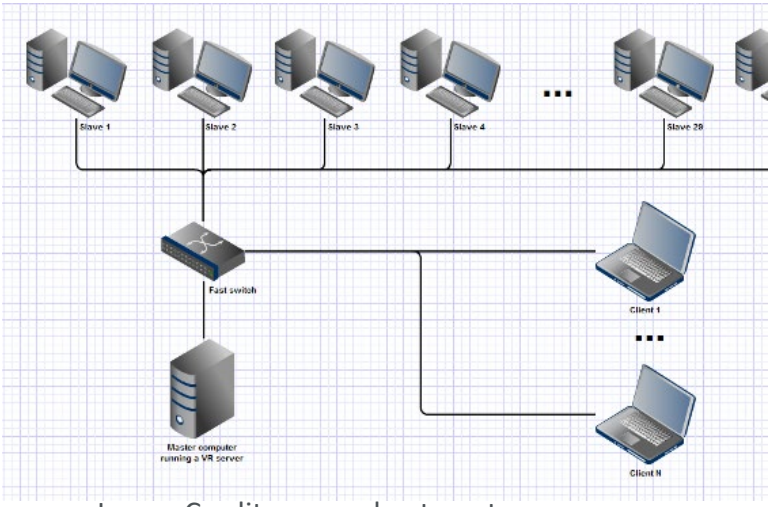


Image Credit :researchgate.net

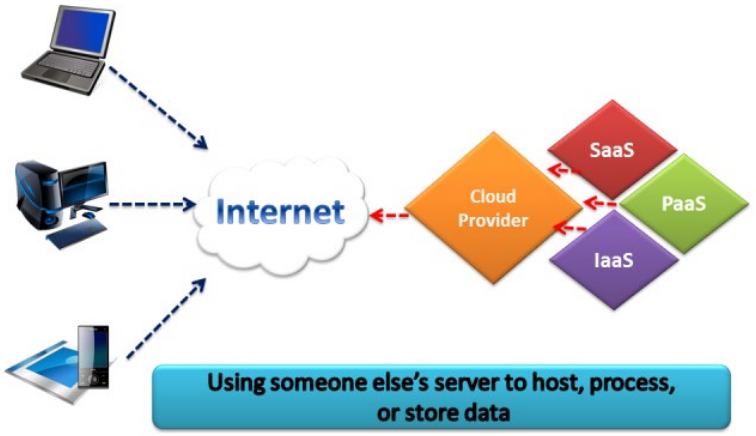


Image Credit: imscindiana.com

Cloud Computing vs Distributed Computing

- The goal of Distributed Computing is to provide collaborative resource sharing by connecting users and resources. Distributed Computing strives to provide administrative scalability (number of domains in administration), size scalability (number of processes and users), and geographical scalability (maximum distance between the nodes in the distributed system).

- Cloud Computing is all about delivering services or applications in on demand environment with targeted goals of achieving increased scalability and transparency, security, monitoring and management. In cloud computing systems, services are delivered with transparency not considering the physical implementation within the Cloud.

Map Reduce Case Study

Prof. Tim Wood & Prof. Roozbeh Haghnazar

A REAL EXAMPLE

- There is a 300MB Text Document and we need to count the frequency of each word.
- Desired Output:

Hello	390
Your	200
Vacation	930
U.S.A	190
Europe	100
Africa	150

A REAL EXAMPLE

- Map Reduce / Hadoop is a great example to solve this problem
- Hadoop is an open source version developed by Yahoo and others
- Goal: Make it easy to use a cluster for large scale data processing tasks

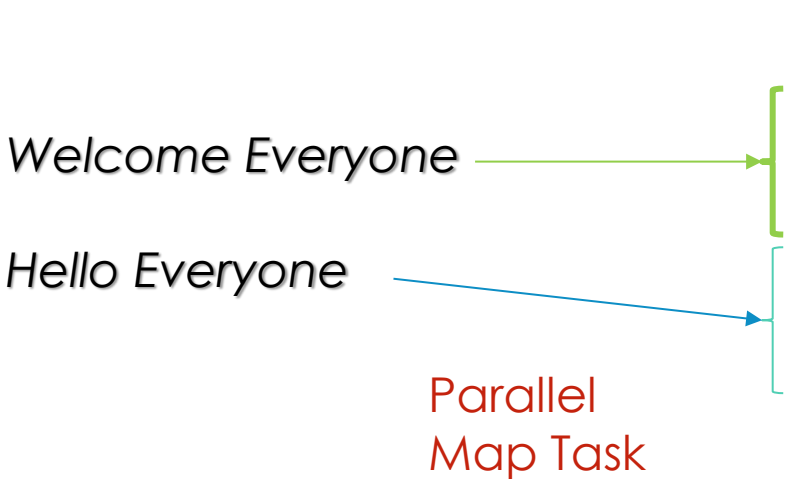
What is map reduce?

- Map square '(1 2 3 4)
  - Output is : (1 4 9 16)
  - [Process each record sequentially and independently]

- Reduce + '(1 4 9 16)
  - Output is : 30
  - [Process set of all records in batches]

Map

- Process individual records to generate intermediate KEY/VALUE pairs
- Parallelly process a large number of individual records

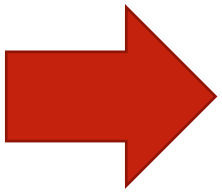


Key	Value
Welcome	1
Everyone	1
Hello	1
Everyone	1

REDUCE

- Processes and merges all intermediate values associated per key

Key	Value
Welcome	1
Everyone	1
Hello	1
Everyone	1

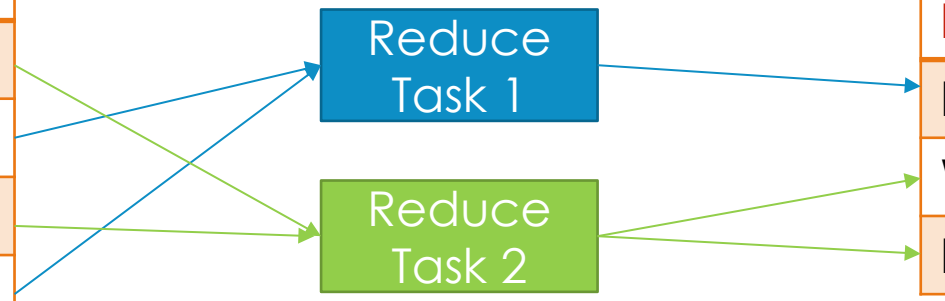


Key	Value
Everyone	2
Welcome	1
Hello	1

REDUCE

- Each key assigned to one Reduce task
- Parallelly Processes and merges all intermediate values by partitioning keys

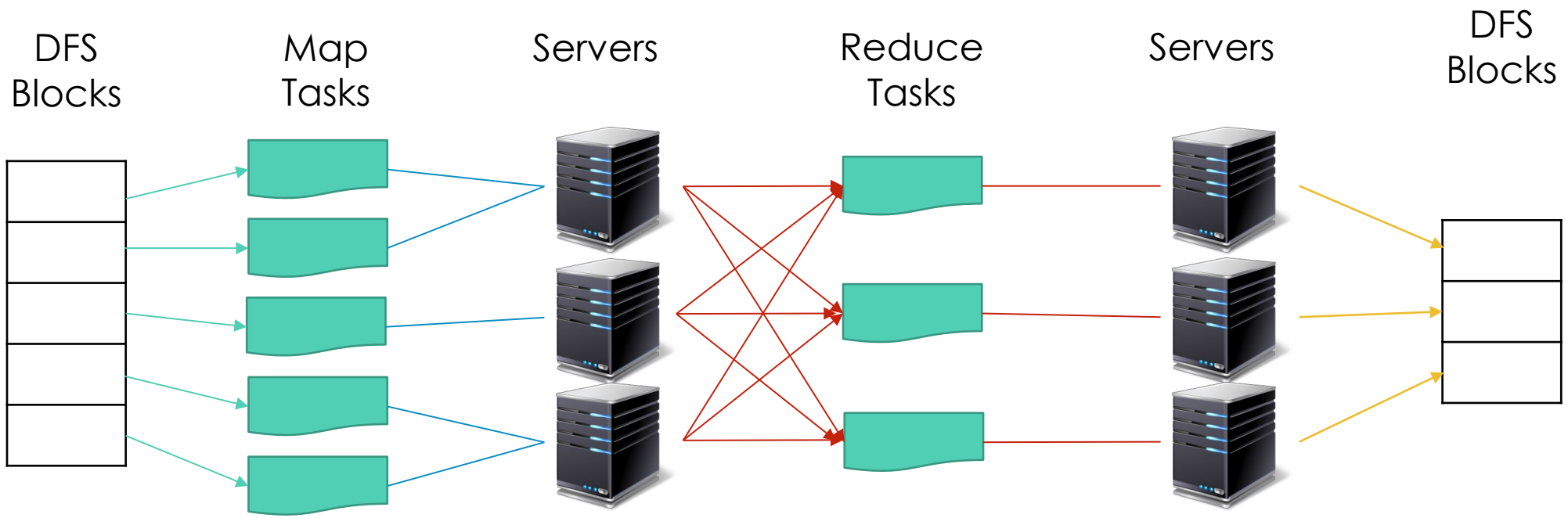
Key	Value
Welcome	1
Everyone	1
Hello	1
Everyone	1



Key	Value
Everyone	2
Welcome	1
Hello	1

- Usually we can use hash partitioning technique:
  - Reduce #=Hash(Key)%Number of Reduce processors

Internal working of Map-reduce



Map/Reduce phases

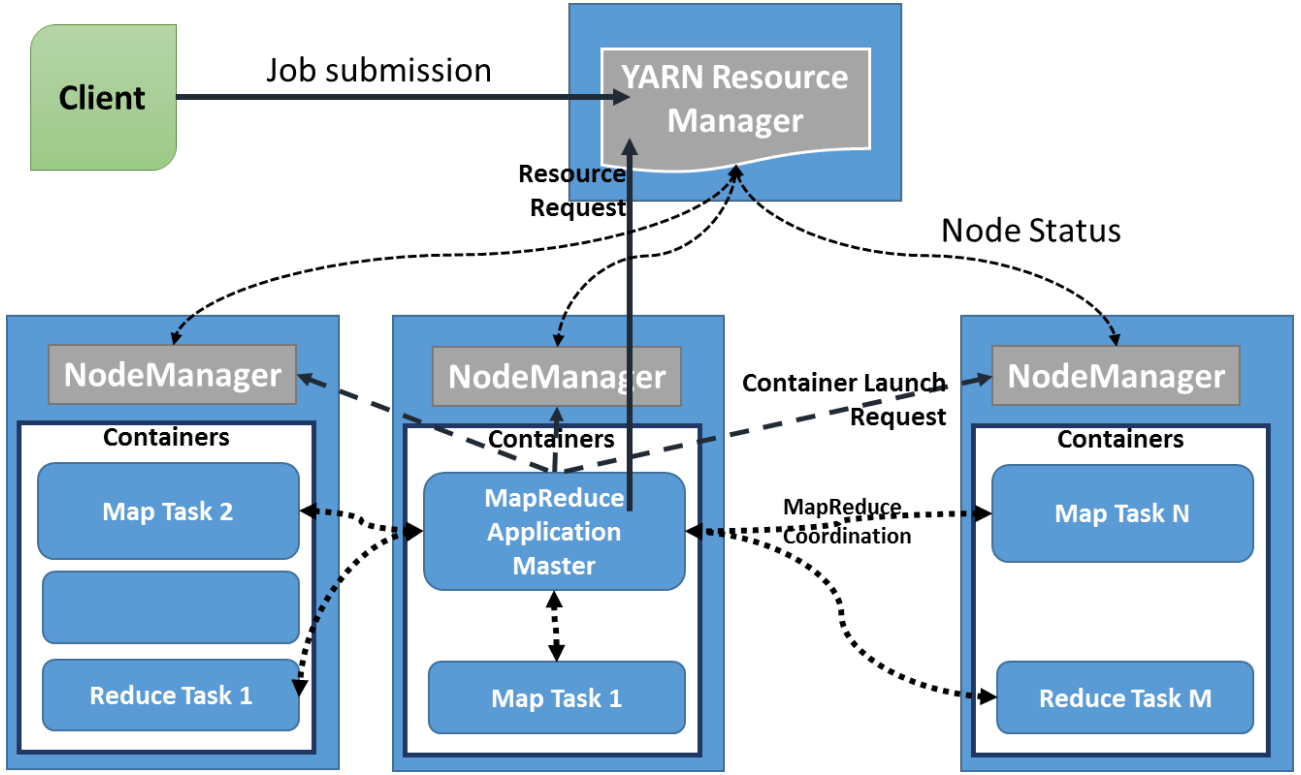
- Map Phase
  - input: data element
  - Convert input data into an intermediate result
  - All map functions can be called independently in parallel
  - output: {list of keys and values}
- Shuffle and partition
  - Sort all outputs by key and combine values into a list
  - Partition keys to create new Reduce tasks
  - output: Key, {list of values}
- Reduce Phase
  - input: Key, {list of values}
  - Combine the list of values for each key to produce an output

YARN SCHEDULER

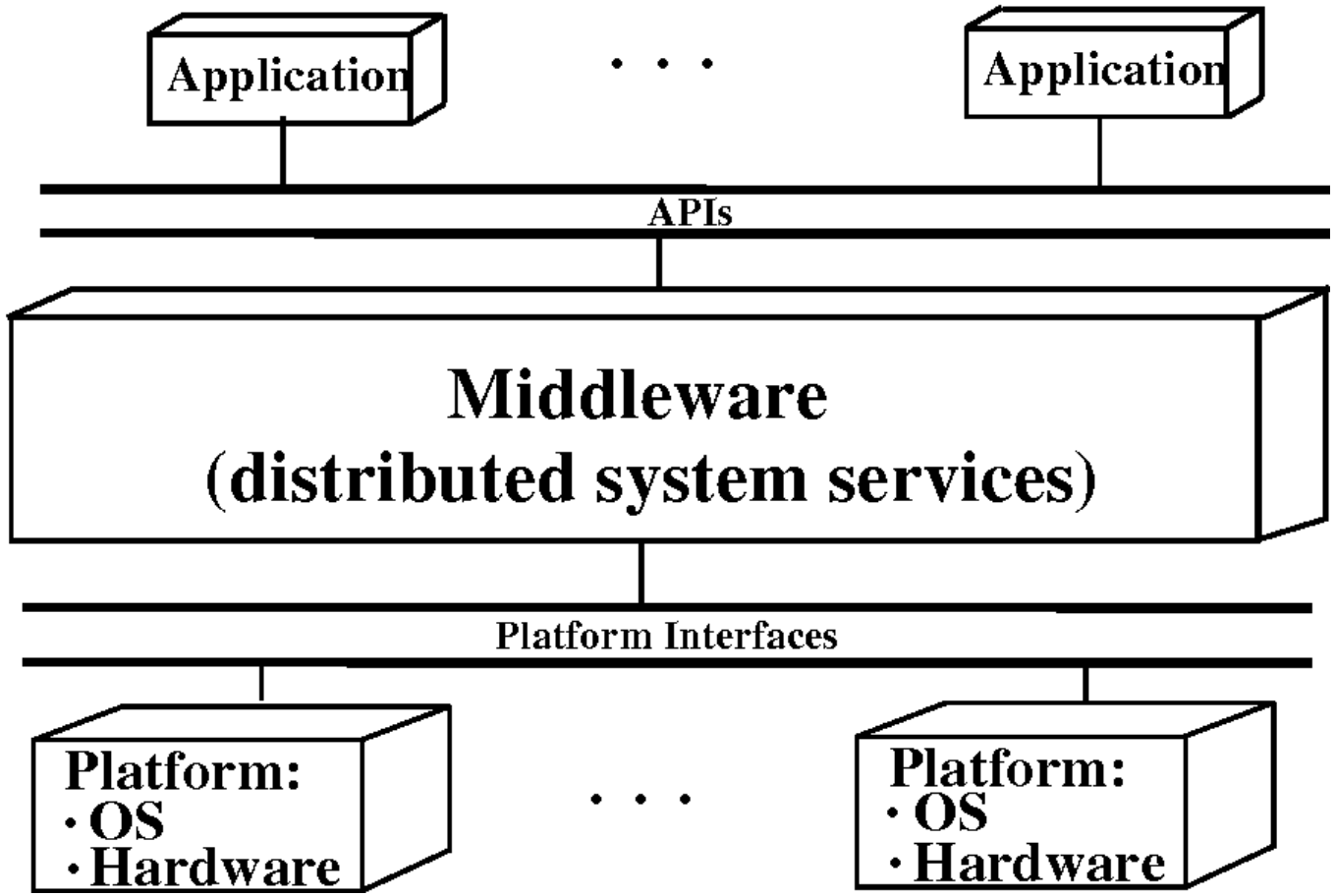
- Yet another Resource Negotiator
- Treat each server as a collection of containers
  - Container is a set of some processors and memory
- 3 main components:
  - Global Resource manager (RM) -> Scheduling
  - Per-Server Node manager (NM)
  - Per-application (job) Application Master (AM)

YARN Scheduler

- There are some servers
- Containers run the jobs
- Application Masters control and execute the tasks
- Node Manager can inform the Resource Manager that hey one task is done.
- Then RM, inform the AM who has a task to get done.
- AM negotiate with NM to get the task to the container to get done.



Distributed System Architecture



Challenges

- Heterogeneity
- Openness
- Security
- Failure Handling
- Concurrency
- Quality of Service
- Scalability
- Transparency

PITFALLS

- Peter Deutsch at Sun Microsystems, formulated these mistakes as the following false assumptions that everyone makes when developing a distributed application for the first time:
  1. The network is reliable
  2. The network is secure
  3. The network is homogeneous
  4. The topology does not change
  5. Latency is zero
  6. Bandwidth is infinite
  7. Transport cost is zero
  8. There is one administrator

Building Blocks

Processes, Threads, VMs....

Prof. Tim Wood & Prof. Roozbeh Haghnazar

What is a process

- *"A program in execution".*
- *In computing, a process is the instance of a computer program that is being executed by one or many threads. It contains the program code and its activity. Depending on the operating system, a process may be made up of multiple threads of execution that execute instructions concurrently.*
- **Traditional operating systems**: concerned with the "local" management and scheduling of processes.
- **Modern distributed systems**: a number of other issues are of equal importance.
- There are three main areas of study:
  1. Threads and virtualization within clients/servers
  2. Process and code migration
  3. Software agents

THREADS

- Modern OSs provide "virtual processors" within which programs execute.
- A programs *execution environment* is documented in the *process table* and assigned
  a *PID*.
- To achieve acceptable performance in distributed systems, relying on the OS's idea
  of a process is often not enough – *finer granularity* is required.
- The solution: Threading.

Problems with Processes

- Creating and managing processes is generally regarded as an *expensive* task (`fork` system call).
- Making sure all the processes peacefully co-exist on the system is not easy (as *concurrency transparency* comes at a price).
- **Threads** can be thought of as an "execution of a part of a program (in user-space)".
- Rather than make the OS responsible for concurrency transparency, it is left to the *individual application* to manage the creation and scheduling of each thread.

Challenges and Trade-offs

- Heterogeneity
- Openness
- **Security**
- **Failure Handling**
- **Concurrency**
- Quality of Service
- **Scalability**
- Transparency

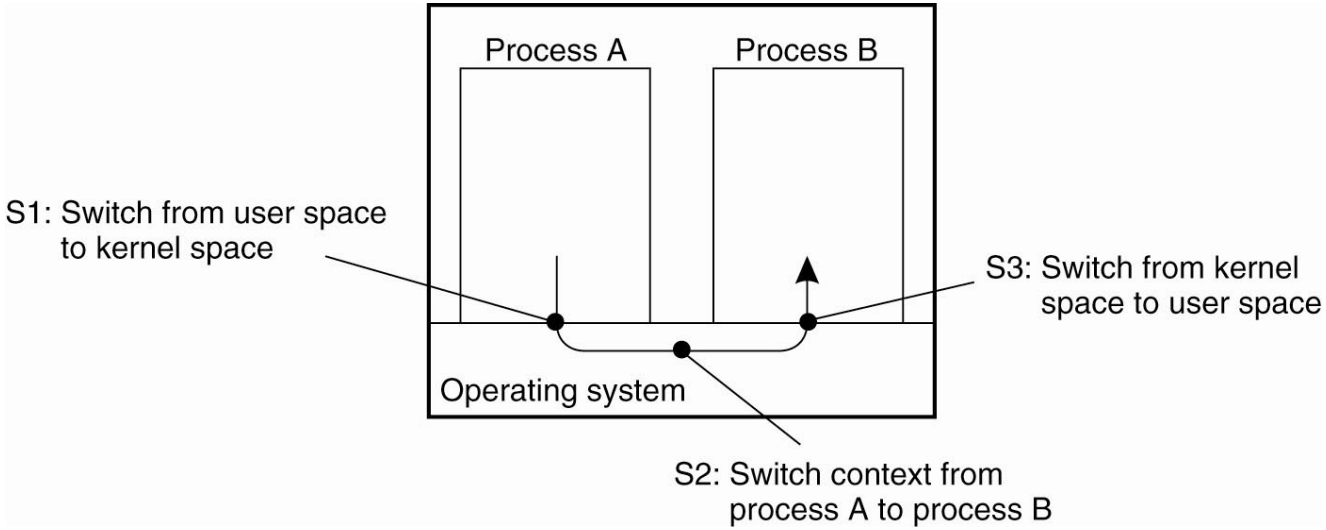
Processes vs Threads?

Important Implications

- Two Important Implications:
  1. Threaded applications often run faster than non-threaded applications (as context-switches between kernel and user-space are avoided).
  2. Threaded applications are harder to develop (although simple, clean designs can help here).
- Additionally, the assumption is that the development environment provides a
  Threads Library for developers to use
  (most modern environments do).

Thread Usage in Non-distributed Systems

- Blocking can be avoided
- Excellent support for multi-processor systems (each running their own thread).
- Expensive context-switches can be avoided.
- For certain classes of application, the design and implementation is made considerably easier.



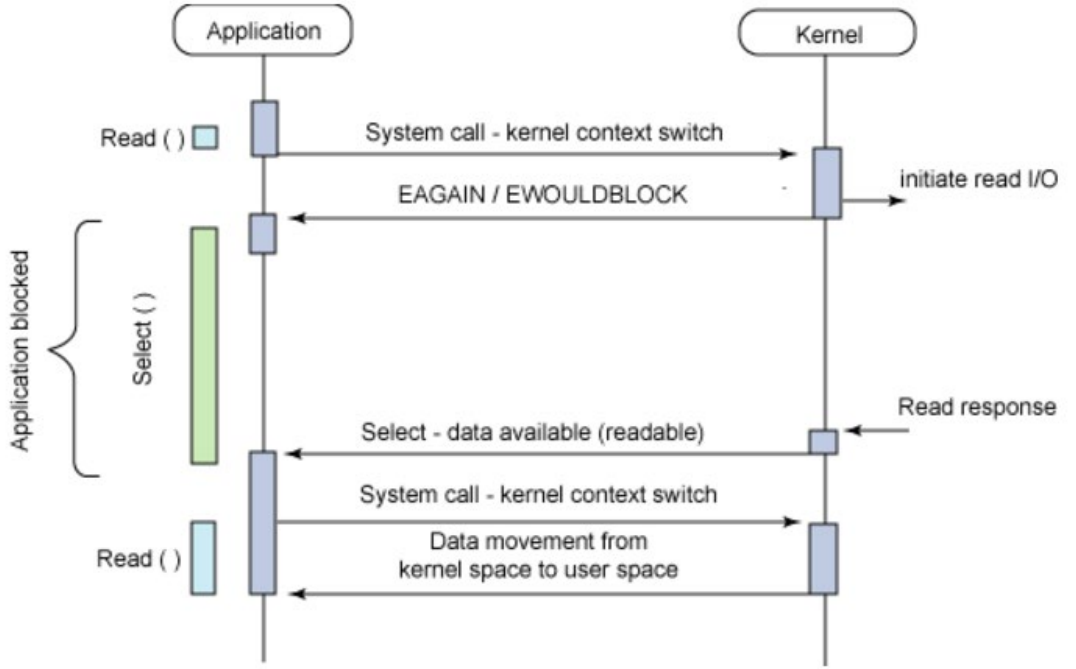
Context switching as the result of IPC

Thread Implementation

- User Level
  - Cheap to create and destroy threads
  - Switching Context can often be done in just few instructions
  - Only drawback is that invocation of a blocking system call will immediately block the entire process
- Kernel Level
  - Every thread operation will have to be carried out by the Kernel, requiring system call
  - Switching thread context become as expensive as switching process context
  - While kernel-level threads are less efficient than user-level threads in some aspects, they still provide significant performance benefits over processes. For example:
    - Threads share the same memory space, reducing the overhead of inter-thread communication compared to inter-process communication.
    - They allow true parallelism on multi-core processors, unlike user-level threads limited to user-space scheduling.

BLOCKING SYSTEM CALL

- A blocking system call is a type of system call that causes the process (or thread) that invoked it to pause execution until a specific event or condition is fulfilled. During this time, the process/thread cannot continue executing, as it is waiting for the system call to complete.
- Key Characteristics of Blocking System Calls:
  - Wait for Completion:The calling thread or process cannot proceed until the system call finishes its task. For example:
    - Reading from a file: The process waits until the data is read.
    - Network requests: The process waits for the network response.
    - Sleep: The process/thread waits for the timer to expire.
  - Involves Kernel Interaction:These calls typically involve waiting for resources managed by the operating system kernel, such as I/O devices, timers, or synchronization primitives.
  - Common Examples:
    - read(): Waits for input from a file or device.
    - accept(): Waits for an incoming connection on a socket.
    - recv(): Waits for data from a network socket.
    - wait(): Waits for a child process to terminate.



Why Does a Blocking System Call Affect User-Level Threads?

- **User-Level Threads Context**:
  - In user-level threading, all threads in a process are managed in user space by a thread library. The kernel views the entire process as a single entity and is unaware of the individual threads within it.
- **What Happens During a Blocking System Call**:
  - If one user-level thread makes a blocking system call, the kernel sees the **entire process** as waiting (because the kernel doesn't distinguish between threads in the process).
  - As a result, the kernel blocks the entire process, including all other user-level threads, even if they are ready to execute.

EXAMPLE SCENARIO:

:

1. **Thread A** performs computation (e.g., calculating Fibonacci numbers).
2. **Thread B** waits for user input using the read() system call.

If Thread B makes the blocking read() call:

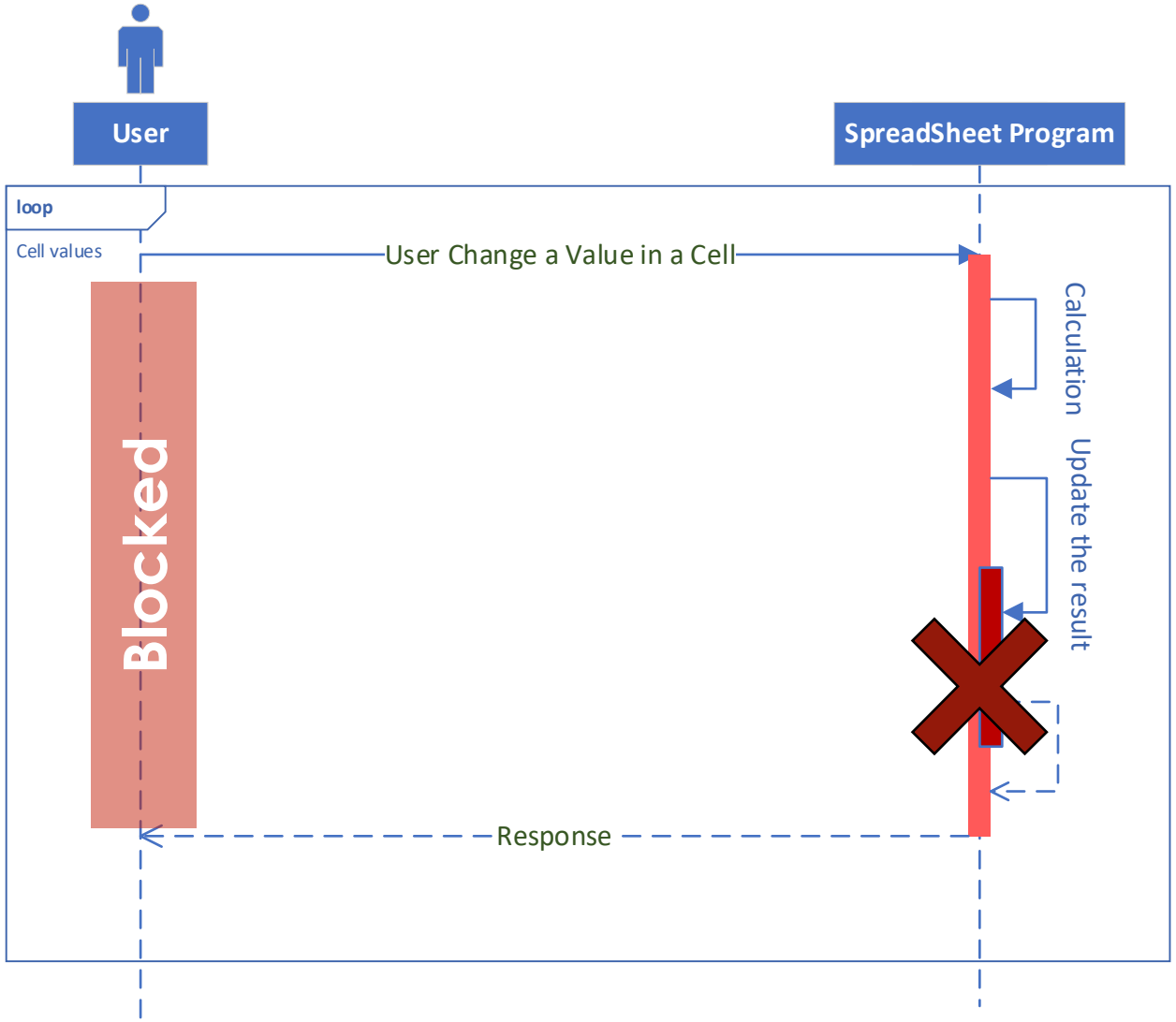
- The kernel blocks the entire process, which includes Thread A.

- Thread A cannot continue executing its computations, even though it is not dependent on Thread B's input.

This is the **major drawback** of user-level threads: one thread's blocking operation halts the execution of all threads in the process.

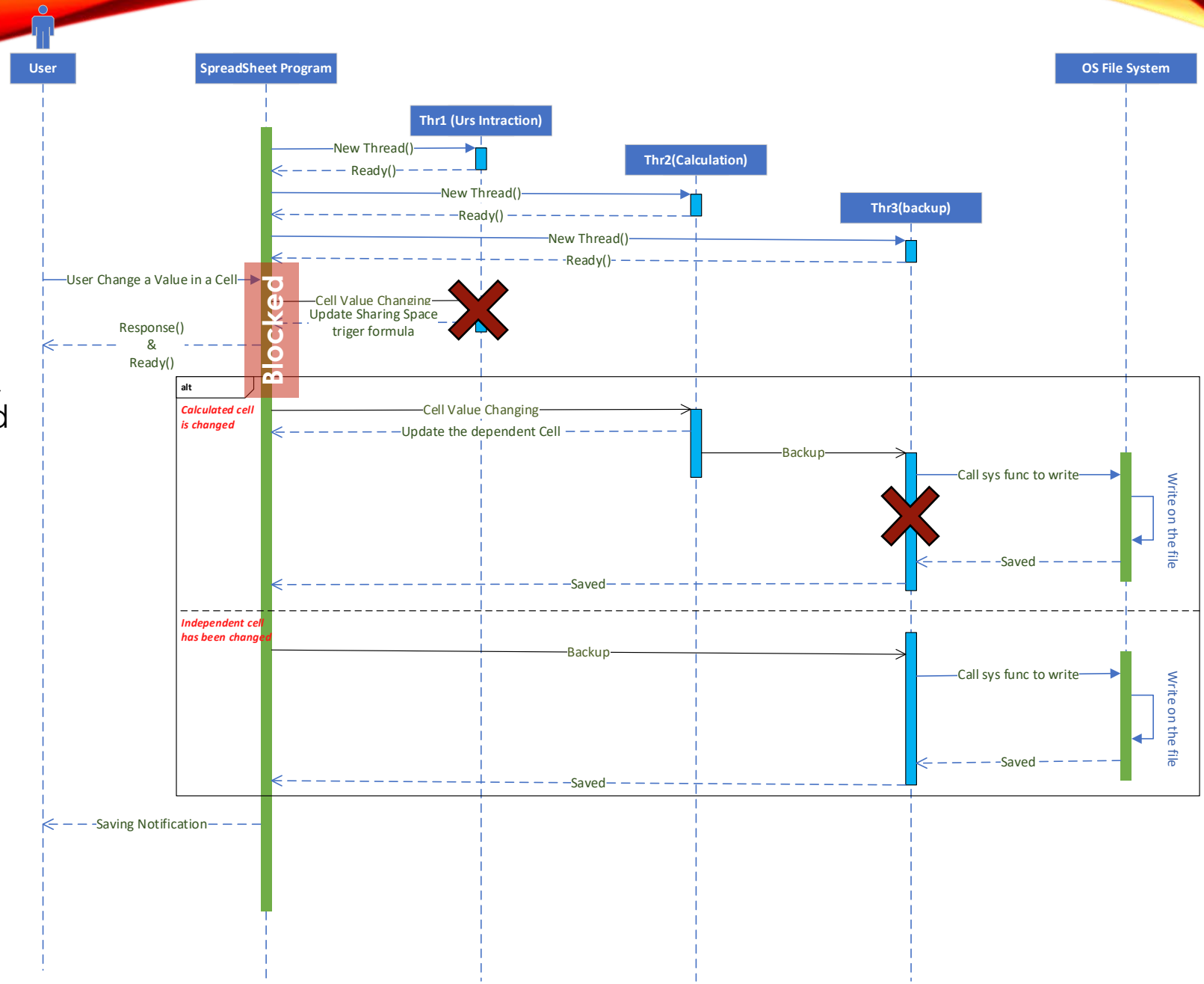
SINGLE THREAD

- Ex: when a user wants to change a number in a cell, the process of application block the process to calculate the chain of formulas (dependent cells) in the sheet and then update the result and then the user can change another value



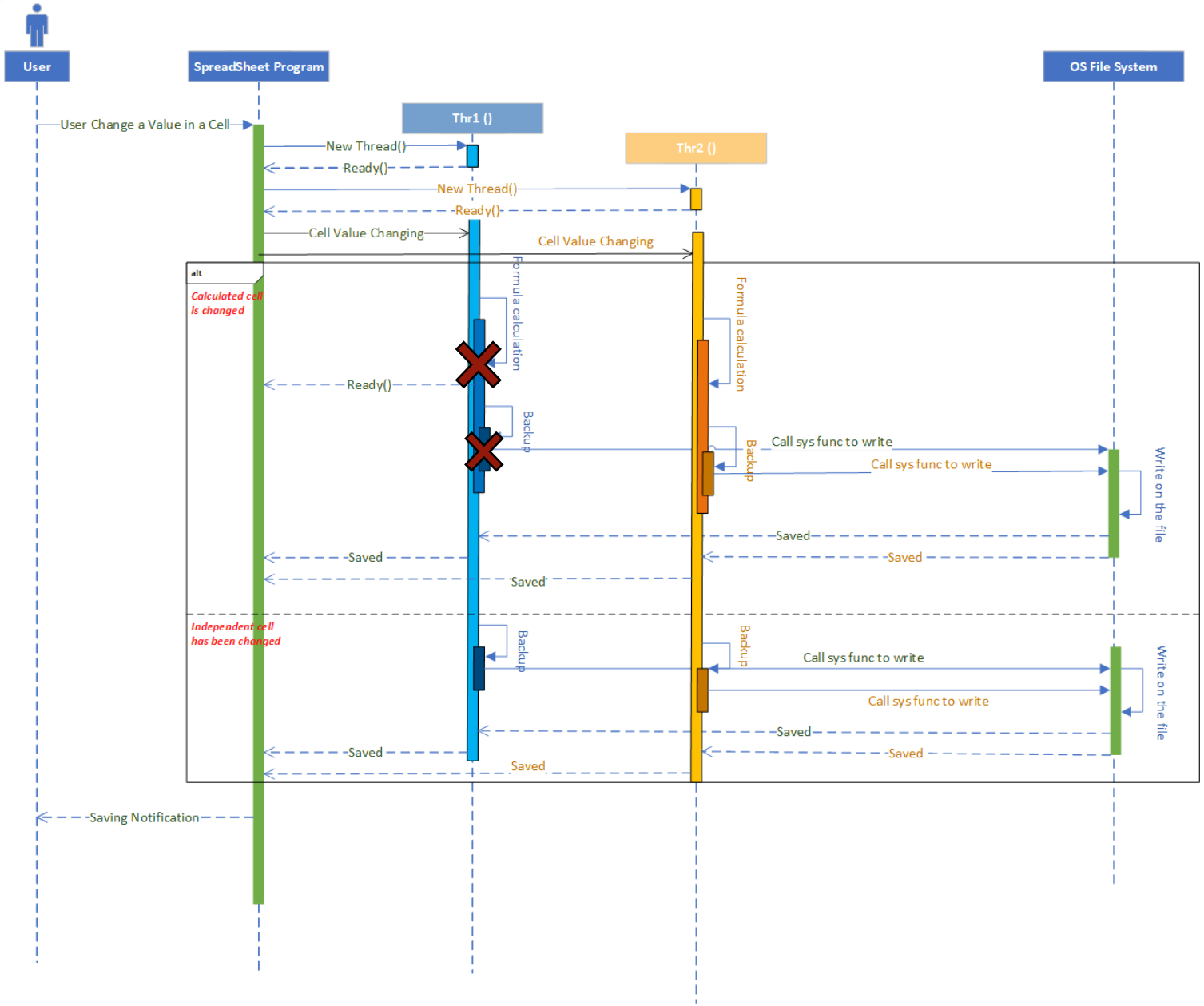
Multi Thread

- Ex: Based on the last example, in this multi thread design approach, we assigned each task to a thread to make asynchronous activities



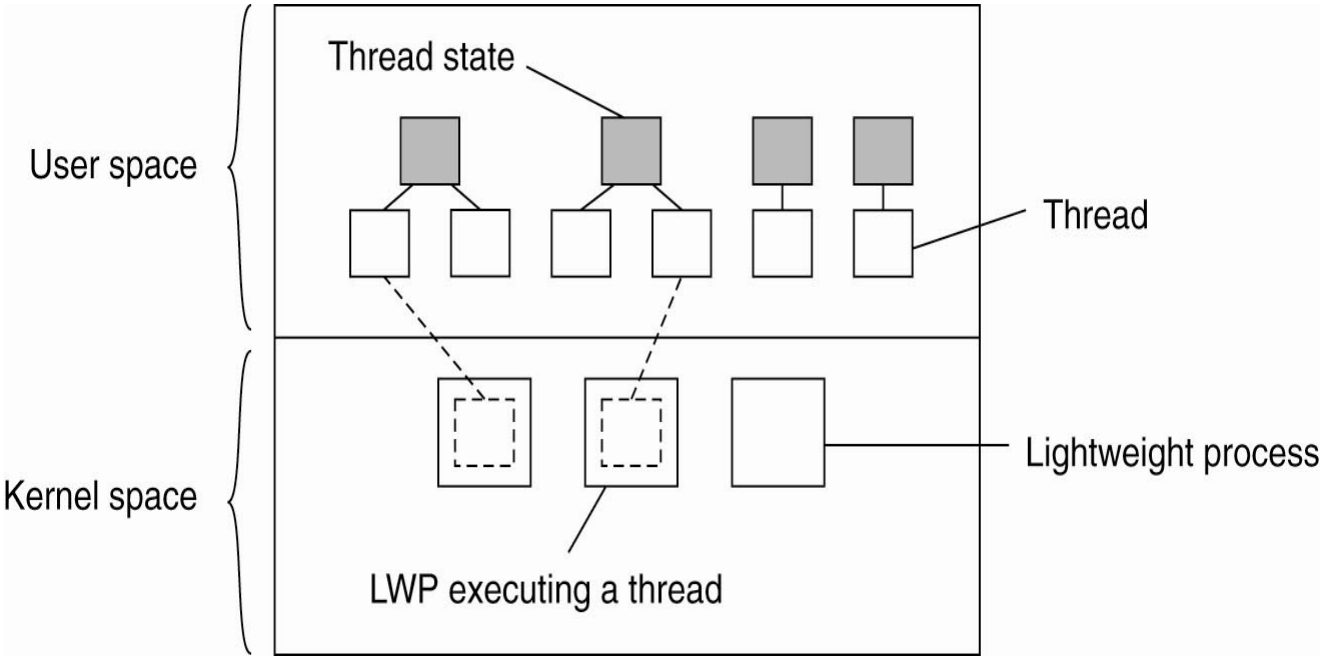
Multi Thread

- Ex: This is another approach to assign each user action and its following required procedures to one thread



Thread Implementation

- Hybrid form
  - Lightweight processes (LWP)
  - LWP runs in a context of single process and there can be several LWP per single process



Threads in Distributed Systems

- Important characteristic: a *blocking call* in a (kernel-based) thread does not result in the entire process being blocked.
- This leads to the key characteristic of threads within distributed systems:
  - "We can now express communications in the form of maintaining multiple logical connections at the same time (as opposed to a single, sequential, blocking process)."

Example: MT Clients and Servers

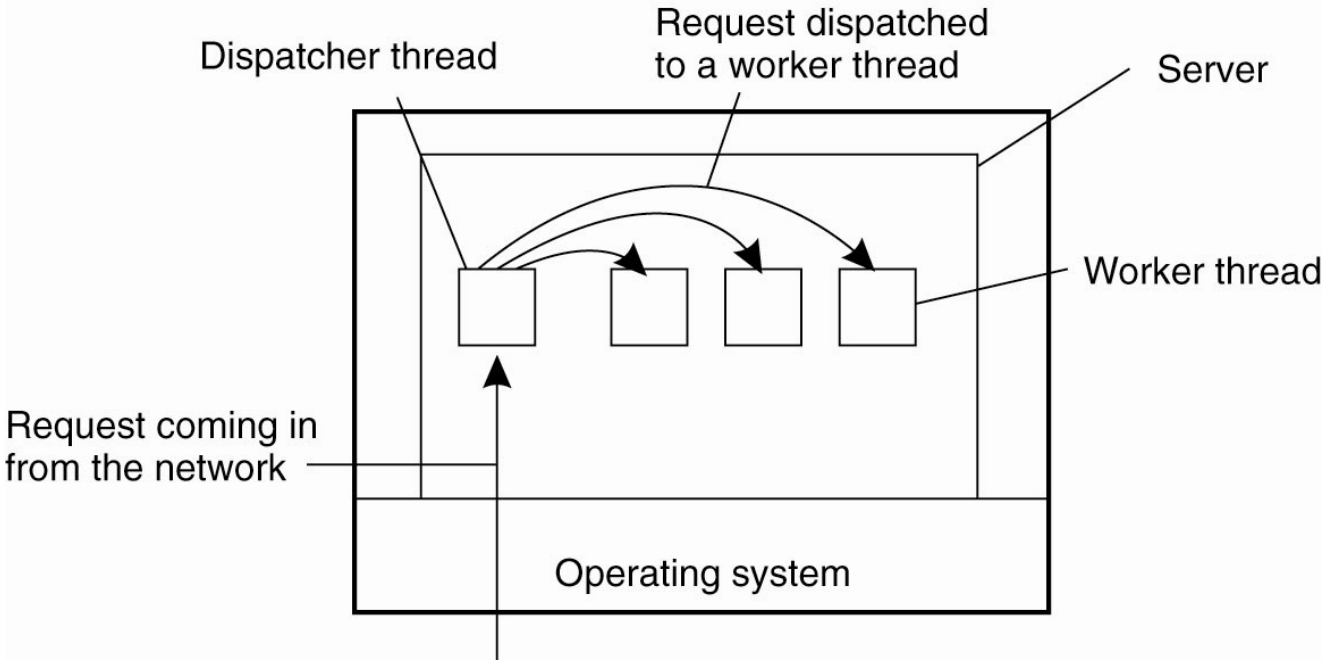
- Mutli-Threaded Client: to achieve acceptable levels of *perceived performance*, it is often necessary to hide communications latencies.
- Consequently, a requirement exists to start communications while doing something else.
- Example: modern Web browsers.
- This leads to the notion of "truly parallel streams of data" arriving at a multi-threaded client application.

Example: MT-Servers

- Although threading is useful on clients, it is much more useful in distributed systems servers.
- The main idea is to exploit parallelism to attain high performance.
- A typical design is to organize the server as a single "dispatcher" with multiple threaded "workers", as diagrammed overleaf.

Multithreaded Server

- Dispatcher can find a suitable thread to serve client.
- Dispatcher transfer requests to proper worker thread



A multithreaded server organized in a dispatcher/worker model

(End of Lecture 2)

Video on Azure HaaS