Distributed Systems

Processes and Threads in Distributed Systems

WEEK 3: Processes, Threads, Multithreading, Concurrency, Process Management.

Online Lecture Series - 3

Felix Edesa

Addis Ababa Science and Technology University

AASTU

Course Outline

Topics We Will Cover

- **Threads** — Basics & Distributed Use
- **Virtualization** — Principles, Containers, VMs vs. Containers, Applications
- **Clients** — Interfaces, Virtual Desktops, Transparency Tools
- **Servers** — Design Issues, Object Servers, Apache, Clusters
- **Code Migration** — Why, Models, Heterogeneous Systems
- **Summary & Next Steps**

AASTU

Processes in Distributed Systems

Overview: Role of Processes

- **Processes** — Fundamental units in distributed systems
  - Program in execution (from OS perspective)
  - OS focuses on management scheduling
  - Distributed systems add concerns for performance structure
- **Threads** — Enhancing system efficiency
  - Structure clients and servers
  - Improve performance on multicore/multiprocessor systems
  - Sometimes replaced by processes for protection communication

AASTU

Introduction to Threads

What is a Process?

- A **process**:- program in execution
- OS creates virtual processors to run programs
- Process context stores:
  - CPU registers (program counter, stack pointer)
  - Memory maps
  - Open files, privileges, and accounting info
- Provides **concurrency transparency** between processes

AASTU

Introduction to Threads

Thread vs Process

- A **thread** executes a piece of code independently
- Thread context contains minimal info for CPU sharing
- Threads do not enforce strict concurrency protection
- Implications:
  - Often improves performance (multithreading)
  - Requires careful design to avoid data issues

AASTU

Multithreading for Parallelism

Key Points

- Threads can run on separate CPUs/cores
- Shared data stored in main memory
- Transparent performance: works on uniprocessor systems too
- Increasingly important due to cheap multicore systems
- Applications: Servers, client-server apps, smartphones

AASTU

Contexts in Multithreading

Understanding Contexts

- **Processor Context** — Minimal info stored in CPU registers
  - Includes program counter, stack pointer, addressing registers
  - Required to resume execution after an interrupt
- **Thread Context** — Adds thread-specific info
  - Contains processor context plus thread state
  - Enables CPU sharing among multiple threads
- **Process Context** — Full context for a process
  - Includes thread context and additional OS info
  - May include Memory Management Unit (MMU) registers
  - Supports isolation between processes

AASTU

Why Use Threads

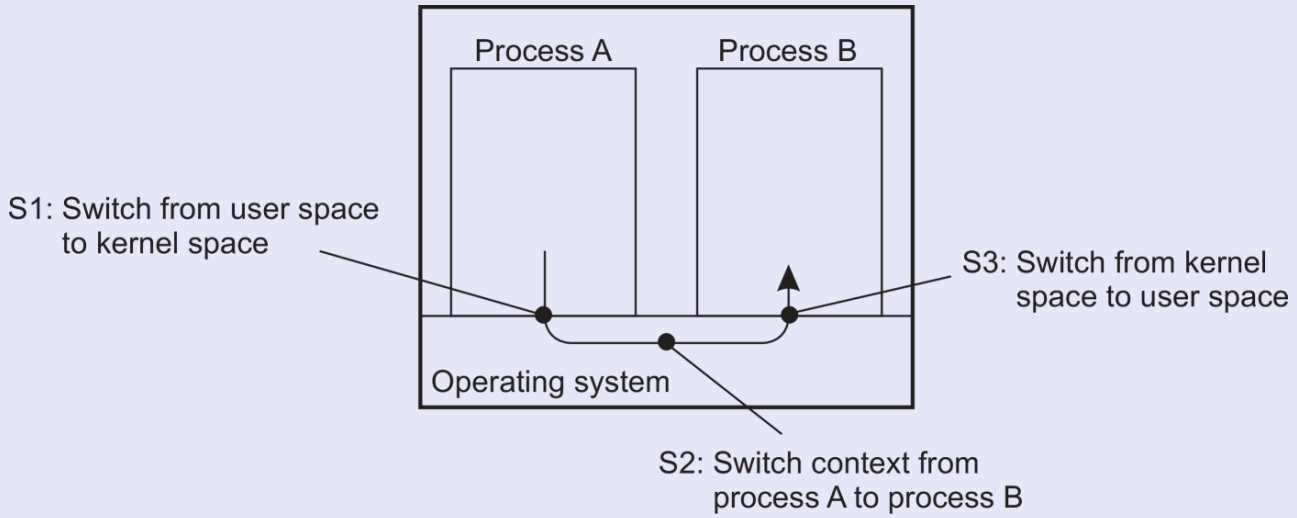
Key Benefits of Threads

- **Avoid needless blocking**
  - In single-threaded processes, I/O operations block the entire process
  - Multithreading allows the OS to switch the CPU to another thread
- **Exploit parallelism**
  - Threads can run simultaneously on multiple cores or processors
  - Improves performance in multicore/multiprocessor systems
- **Avoid costly process switching**
  - Using threads in large applications lowers overhead and boosts responsiveness.

AASTU

Avoid process switching

Avoid expensive context switching



M. van Steen and A. S. Tanenbaum, *Distributed Systems*, 4th ed., 2024.

Advanced Topic: Cost of Context Switch

Measuring Context Switch Overhead

- Context switches incur performance costs, studied extensively.
- Clock handlers (interrupts) are often used to measure overhead.
- Typical clock intervals: 0.5–20 ms (50–2000 Hz).
- Clock handlers assist in:
  - Timing and CPU usage tracking
  - Alarm signals
  - Preempting tasks for fair CPU sharing

AASTU

Advanced Topic: Cost of Context Switch

Direct vs Indirect Overhead

- **Direct overhead**: Time for actual context switch + handler execution
- **Indirect overhead**: Mainly due to cache perturbations
- Intel processors:
  - Context switch: 0.5–1 s
  - Handler execution: 0.5–7 s (implementation dependent)
- Indirect overhead can be much larger (e.g., 80

AASTU

Cache Effects of Context Switches

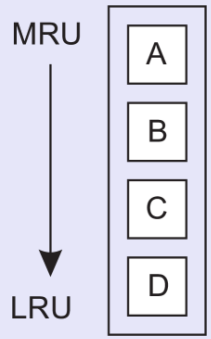
Impact on Performance

- Interrupts can reorganize the cache significantly
- Example: Least Recently Used (LRU) cache block may be evicted
- Accessing evicted blocks reloads them into cache, evicting others
- Even simple interrupts can cause long-lasting cache reorganization
- Overall effect: noticeable performance degradation

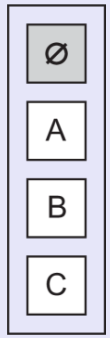
AASTU

Context Switch and Cache Effects

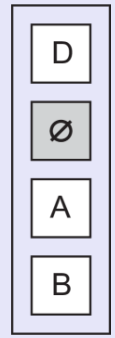
Context Switch



Cache Before Switch



Cache After Switch



M. van Steen and A. S. Tanenbaum, *Distributed Systems*, 4th ed., 2024.

Multiprocessing Example — Highlighted

Python Code

```
1 from multiprocessing import Process
2 from time import gmtime, sleep
3 from random import randint
4 def sleeper(name):
5     t = gmtime()
6     s = randint(1, 20)
7     txt = f"{t.tm_min}:{t.tm_sec} {name} sleeping {s}
8     print(txt)
9     sleep(s)
```

Multiprocessing Example — Part 2

Python Code

```
1       t = gmtime()
2       txt = f"{t.tm_min}:{t.tm_sec} {name} woke up"
3       print(txt)
4
5   if __name__ == "__main__":
6       p = Process(target=sleeper, args=("eve",))
7       q = Process(target=sleeper, args=("bob",))
8       p.start(); q.start()
9       p.join(); q.join()
```

Sample Output

Execution Output

```
40:23 eve is going to sleep for 14 sec
40:23 bob is going to sleep for 4 sec
40:27 bob has woken up
40:37 eve has woken up
```

Multithreading + Multiprocessing Example — Part 1

Python Code — Setup Sleeping Function

```
1 from multiprocessing import Process
2 from threading import Thread
3 from time import sleep
4 from random import randint
5 shared_x = randint(10, 99)
6 def sleeping(name):
7     global shared_x
8     s = randint(1, 20)
```

Multithreading + Multiprocessing Example — Part 2

Python Code — Thread Creation Execution

```
    sleep(s)
    shared_x += 1
def sleeper(name):
    sleeplist = list()
    for i in range(3):
        subsleeper = Thread(target=sleeping, args=(f{nam
        sleeplist.append(subsleeper)
    for s in sleeplist: s.start()
    for s in sleeplist: s.join()
```

Multithreading + Multiprocessing — Sample Output (Part 1)

Execution Output — Part 1

```
eve sees shared x being 71
53:21 eve 0 is going to sleep for 20 seconds
bob sees shared x being 84
53:21 eve 1 is going to sleep for 15 seconds
53:21 eve 2 is going to sleep for 3 seconds
53:21 bob 0 is going to sleep for 8 seconds
53:21 bob 1 is going to sleep for 16 seconds
53:21 bob 2 is going to sleep for 8 seconds
```

AASTU

Multithreading + Multiprocessing — Sample Output (Part 2)

Execution Output — Part 2

```
53:24 eve 2 has woken up, seeing shared x being 72
53:29 bob 0 has woken up, seeing shared x being 85
53:29 bob 2 has woken up, seeing shared x being 86
53:36 eve 1 has woken up, seeing shared x being 73
53:37 bob 1 has woken up, seeing shared x being 87
bob sees shared x being 87
53:41 eve 0 has woken up, seeing shared x being 74
eve sees shared x being 74
```

AASTU

Thread Implementation Approaches

User-Level vs Kernel-Level Threads

- **User-Level Threads**
  - Cheap to create and destroy (memory allocation only)
  - Fast context switching (CPU registers only)
  - Blocking system calls block the entire process
- **Kernel-Level Threads (One-to-One Model)**
  - Each thread is a schedulable entity
  - Operations handled by kernel system calls
  - Thread context switch can be as expensive as process context switch

AASTU

Many-to-Many Threading Model — Concept

Hybrid User-Level Kernel-Level Threads

- **Overview:**
  - Each kernel thread can run multiple user threads
  - User threads managed entirely in user space
  - Scheduler routine selects the next runnable user thread
- **Thread Table:**
  - Shared among kernel threads
  - Mutexes ensure mutually exclusive access

AASTU

Many-to-Many Threading Model — Execution

Execution Flow Context Switching

- **Kernel Threads:**
  - Each executes its own scheduler
  - Finds a runnable user thread
  - Context switch for user threads done entirely in user space
- **Handling Blocking System Calls:**
  - Kernel thread switches from user mode to kernel mode
  - Other kernel threads may continue executing
  - Process-level execution is not blocked

AASTU

Many-to-Many Threading Model — Advantages

Benefits of Using Many-to-Many Model

- Efficient creation, destruction, and synchronization of threads
- Blocking calls in one user thread do not suspend the entire process
- Application sees only user threads, kernel threads hidden
- Multiprocessing support: kernel threads can run on different CPUs/cores

AASTU

Many-to-Many Threading Model — Implementations

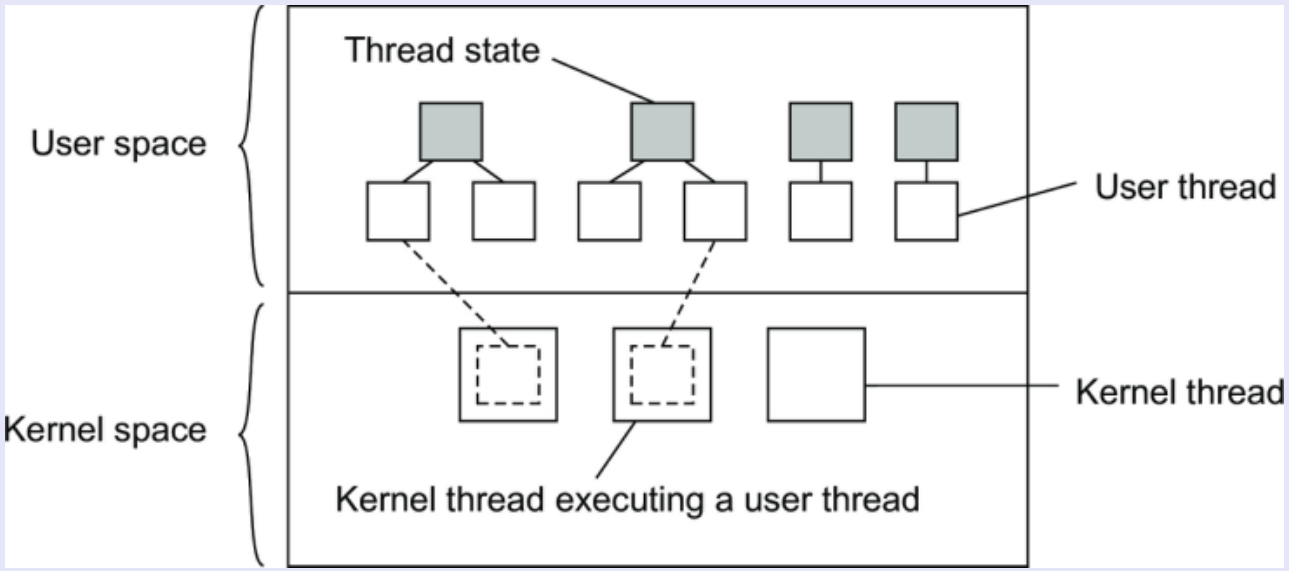
Example Implementations

- Go programming language — combines user kernel threads
- Libfibre runtime system — exemplary many-to-many threading
- Arachne — kernel threads hidden, supports core allocation visibility

AASTU

Combining user-level and kernel-level threads

Two-level threading approach



M. van Steen and
A. S. Tanenbaum, *Distributed Systems*, 4th ed., 2024.

Multithreaded Clients — Parallel Connections

Parallel Connections Example

- After fetching main HTML file, separate threads fetch remaining components
- Each thread sets up its own connection to the server
- Standard blocking calls can be used without suspending the main process
- Result: user can interact with content while additional data is still being fetched

AASTU

Multithreaded Clients — Replicated Servers

Performance Advantages

- Servers can be replicated to handle load or improve speed
- Threads allow simultaneous connections to different replicas
- Entire web document can be displayed faster compared to a non-multithreaded client
- True parallel streams of incoming data are possible only with threads

AASTU

Thread-Level Parallelism (TLP)

Mathematical Definition of TLP

- Let c_i denote the fraction of time exactly i threads are executing
- c_0 is the fraction of idle time (no threads running)
- Maximum number of threads that can execute simultaneously: N

$$\text{TLP} = \frac{\sum_{i=1}^{N} i \cdot c_i}{1 - c_0}$$

- TLP measures effective hardware utilization by multithreading
- Example: A TLP of 2.0 implies two cores are effectively utilized

AASTU

TLP Example in Web Browsers

Web Browser Performance

- Suppose a browser executes up to $N = 4$ threads:

$$c_0 = 0.2, \quad c_1 = 0.4, \quad c_2 = 0.3, \quad c_3 = 0.1$$

- Compute TLP:

$$\text{TLP} = \frac{1 \cdot 0.4 + 2 \cdot 0.3 + 3 \cdot 0.1}{1 - 0.2} = \frac{1.3}{0.8} \approx 1.625$$

- Interpretation: Effectively 1–2 cores are utilized
- Multithreading organizes tasks but may not fully exploit all cores

AASTU

Multithreaded Servers in Distributed Systems

Key Advantages of Multithreaded Servers

- Simplifies server code and improves maintainability
- Exploits parallelism even on uniprocessor systems
- On multicore processors, multithreading enables high performance
- Threads allow handling blocking operations without halting the entire server

AASTU

Multithreaded Servers in Distributed Systems

Key Advantages of Multithreaded Servers

- Simplifies server code and improves maintainability
- Exploits parallelism even on uniprocessor systems
- On multicore processors, multithreading enables high performance
- Threads allow handling blocking operations without halting the entire server

AASTU

Dispatcher/Worker Model in Multithreaded Servers

Server Operation Steps

- Dispatcher thread continuously reads incoming client requests
- Each request is assigned to an idle worker thread
- Worker threads perform blocking operations (e.g., reading from disk)
- If a worker thread is blocked, another thread is scheduled immediately
- Maximizes CPU utilization and increases the number of requests handled per time unit

AASTU

Benefits of Using Threads in Servers

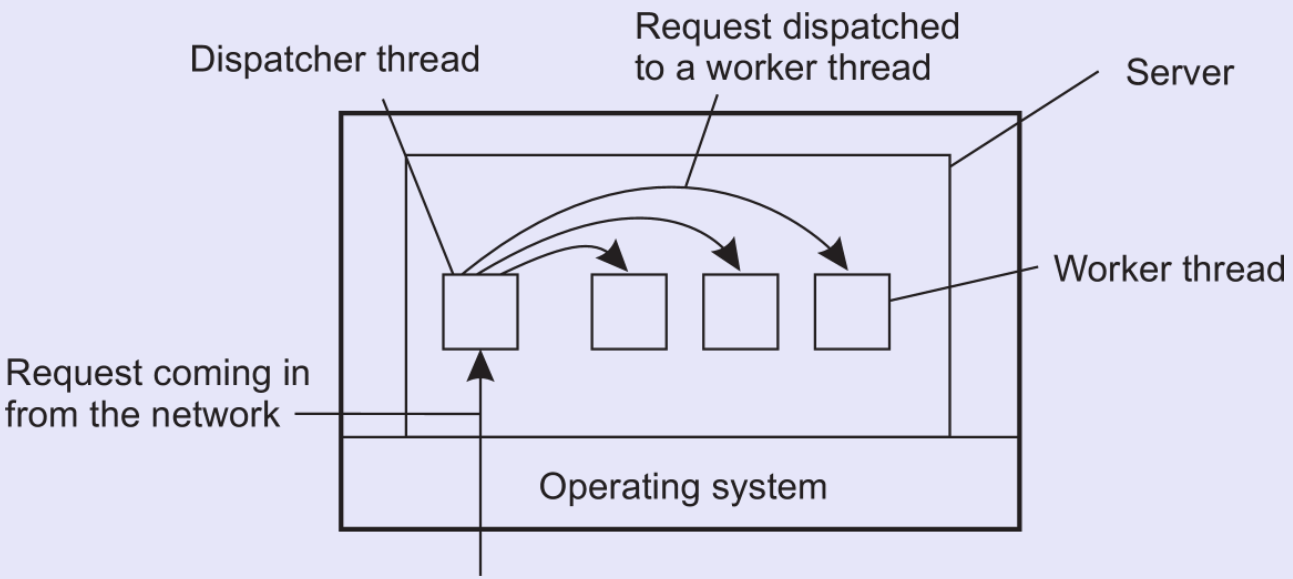
Advantages

- Enables sequential programming with blocking system calls
- Multiple threads can run in parallel, improving throughput
- Threads vs Processes: Processes offer better protection but may reduce performance
- Parallelism through threads is easier to maintain and understand

AASTU

Why multithreading is popular: organization

Dispatcher/worker model



M. van Steen and
A. S. Tanenbaum, *Distributed Systems*, 4th ed., 2024.

AASTU

Constructing a server

Three Ways to construct a server

Model	**Characteristics**
Multithreading	Parallelism, blocking system calls
Single-threaded process	No parallelism, blocking system calls
Finite-state machine	Parallelism, nonblocking system calls

AASTU

3.2 Virtualization

Overview

- Threads/processes allow programs to appear simultaneous.
- Single-core CPUs: illusion of parallelism via rapid switching.
- Resource virtualization extends this idea to hardware/software.
- Enables legacy software to run on new platforms.

AASTU

Virtualization: Importance and Types

Importance

- Reduces platform diversity; apps run on virtual machines.
- Provides portability, flexibility, and isolation (security).
- Supports long-lived software over changing hardware.

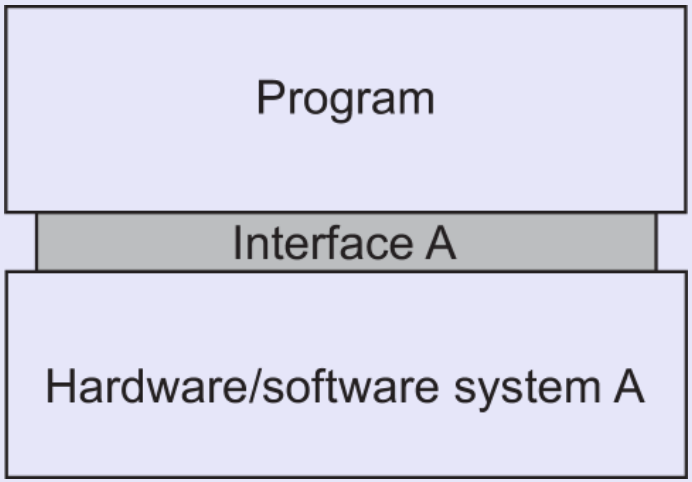
Types of Interfaces

- Instruction Set Architecture (ISA) – hardware-software
- System call interface – operating system
- Library/API interface – hides system calls

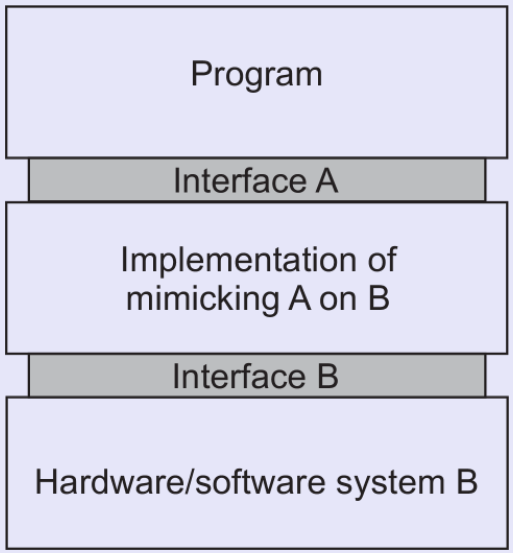
AASTU

Virtualization

Principle: Mimicking Interfaces



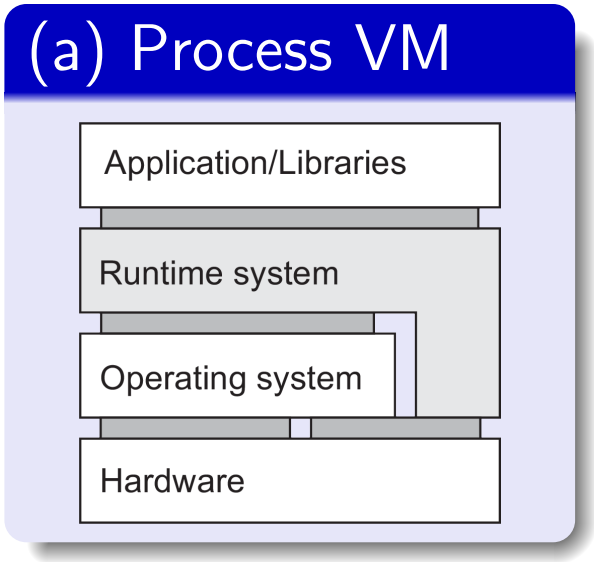
Virtualization Example

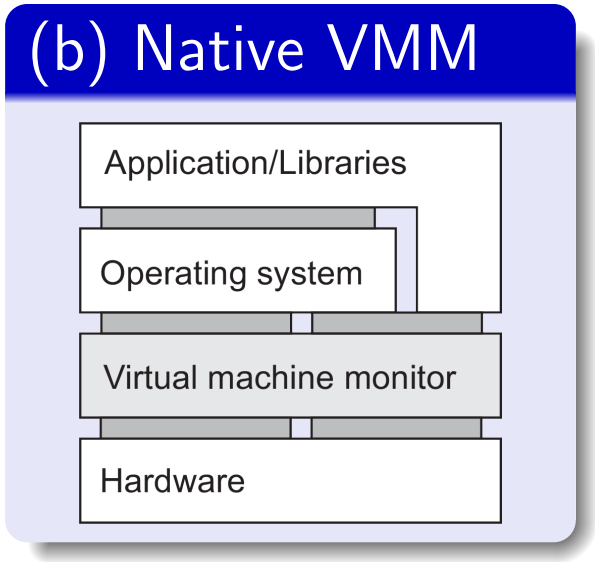


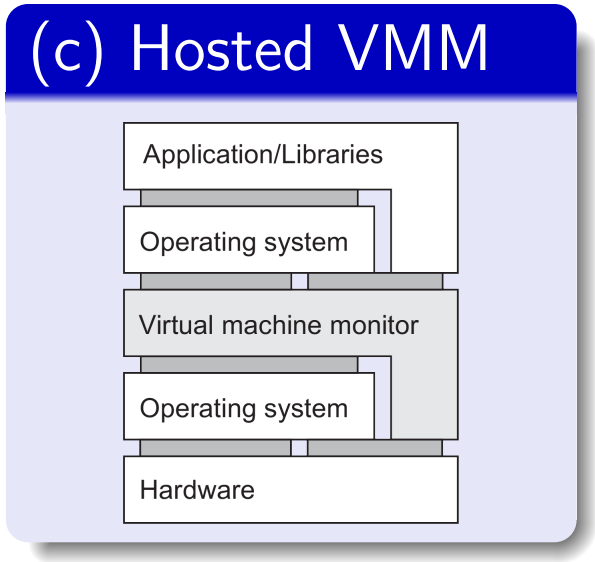
M. van Steen and A. S. Tanenbaum,
Distributed Systems, 4th ed., 2024.

AASTU

Ways of virtualization



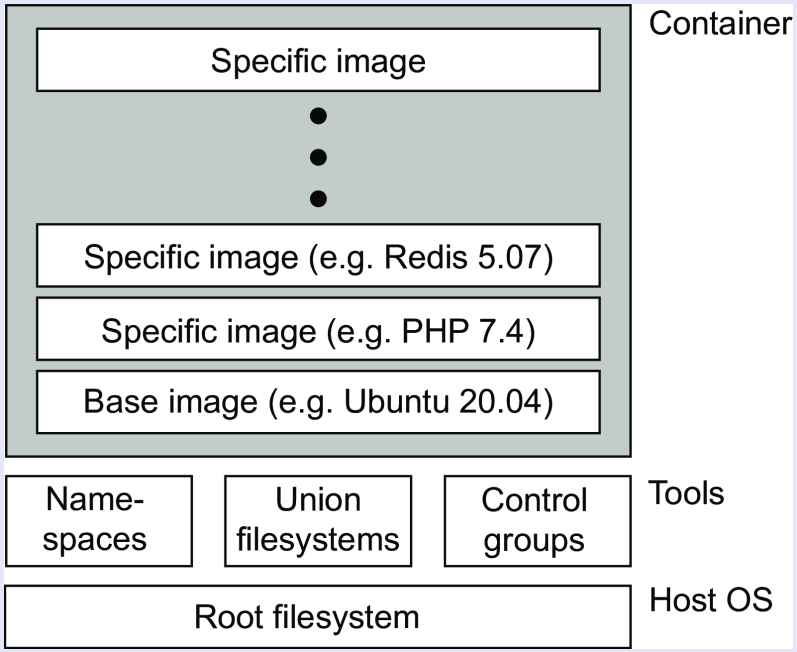




M. van Steen and A. S. Tanenbaum, *Distributed Systems*, 4th ed., 2024.

Containers

Containerization Concept



M. van Steen and A. S. Tanenbaum,
Distributed Systems, 4th ed., 2024.

AASTU

Container Mechanisms

Core Building Blocks

- **Namespaces**: A collection of processes in a container is given their own view of identifiers.
- **Union File System**: Combine several file systems into a layered fashion with only the highest layer allowing for write operations (and the one being part of a container).
- **Control Groups (cgroups)**: Resource restrictions can be imposed upon a collection of processes.

AASTU

Code Migration

What is Code Migration?

- Traditionally, distributed systems mainly focused on passing data.
- In some cases, passing **programs** (even while executing) simplifies system design.
- We take a detailed look at **what code migration is** and why it matters.

AASTU

Reasons for Migrating Code

Why Move Code Between Machines?

- **Performance**: Move processes from heavily loaded to lightly loaded machines.
- **Resource Optimization**: Migrate VMs to minimize active nodes in data centers.
- **Data Proximity**: Execute code close to data sources (e.g., databases, mobile devices).

AASTU

Code Migration and Performance

Improving Performance

- Offload client tasks to servers to reduce network traffic.
- Run form-processing logic on clients to minimize server load.
- Mobile agents: small programs that move across sites to perform tasks in parallel.

AASTU

Code Migration for Privacy and Security

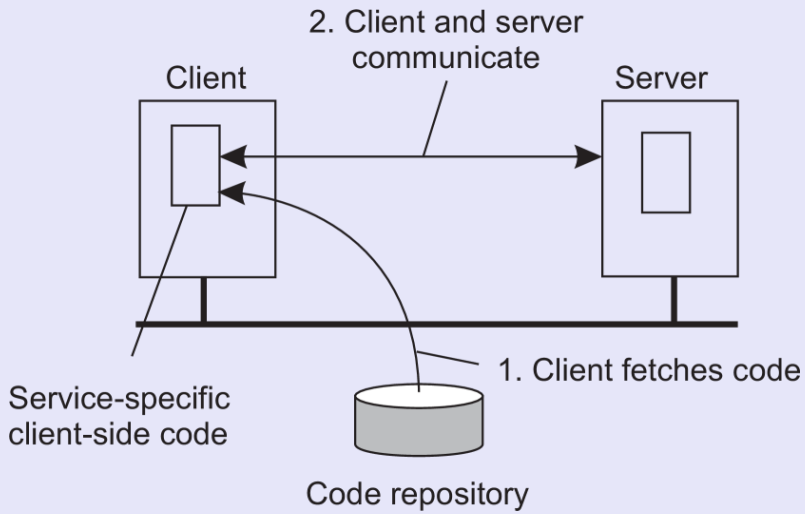
Federated Learning Example

- Sensitive data (e.g., medical records, financial transactions) stays local.
- Instead of moving data, models are trained by sending code to devices.
- Improves privacy while still leveraging distributed machine learning.

AASTU

Reasons to migrate code

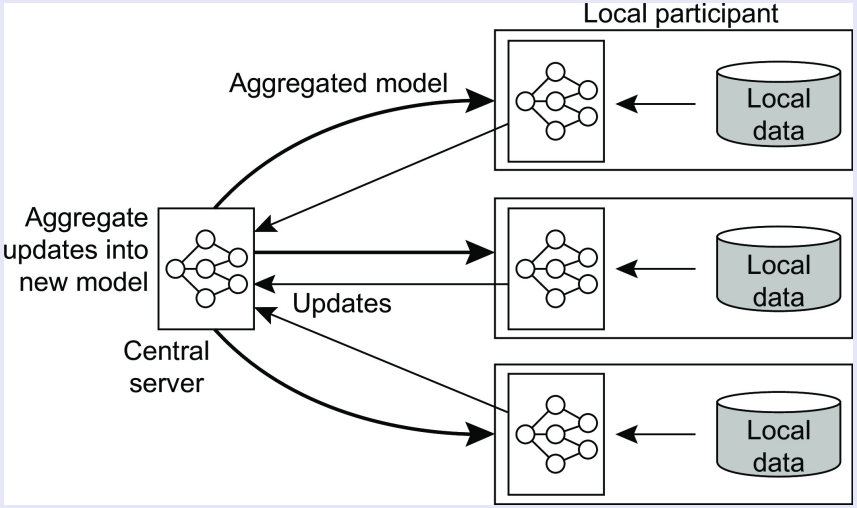
Load distribution



M. van Steen and A. S. Tanenbaum,
Distributed Systems, 4th ed., 2024.

Reasons to migrate code

Privacy and security



M. van Steen and A. S. Tanenbaum,
Distributed Systems, 4th ed., 2024.

Questions?

Thank you for your attention!

AASTU

References

Suggested References

1. M. van Steen and A. S. Tanenbaum, *Distributed Systems*, 4th ed., Version 4.02, February 2024.
2. G. Coulouris, J. Dollimore, T. Kindberg, and G. Blair, *Distributed Systems: Concepts and Design*, 5th ed., Addison-Wesley, 2012.
3. N. Lynch, *Distributed Algorithms*, 2nd ed., Morgan Kaufmann, 1996.
4. R. E. Bryant and D. O'Hallaron, *Computer Systems: A Programmer's Perspective*, 4th ed., Pearson, 2024.

AASTU