

Principles of Distributed Systems

inft-3507

Dr. J.Burns

ADA University

Autumn 2025

Section 3: Processes

Ref: <https://www.distributed-systems.net/index.php/books/ds4/>

Introduction to threads

Basic idea

We build **virtual processors** in software, on top of physical processors:

- Processor:** Provides a set of instructions along with the capability of automatically executing a series of those instructions.
- Thread:** A minimal software processor in whose **context** a series of instructions can be executed. Saving a thread context implies stopping the current execution and saving all the data needed to continue the execution at a later stage.
- Process:** A software processor in whose context one or more threads may be executed. Executing a thread, means executing a series of instructions in the context of that thread.

Context switching

Contexts

- **Processor context:** The minimal collection of values stored in the registers of a processor used for the execution of a series of instructions (e.g., stack pointer, addressing registers, program counter).
- **Thread context:** The minimal collection of values stored in registers and memory, used for the execution of a series of instructions (i.e., processor context, state).
- **Process context:** The minimal collection of values stored in registers and memory, used for the execution of a thread (i.e., thread context, but now also at least MMU register values).

Context switching

Observations

- ① Threads share the same address space. Thread context switching can be done entirely independent of the operating system.
- ② Process switching is generally (somewhat) more expensive as it involves getting the OS in the loop, i.e., trapping to the kernel.
- ③ Creating and destroying threads is much cheaper than doing so for processes.

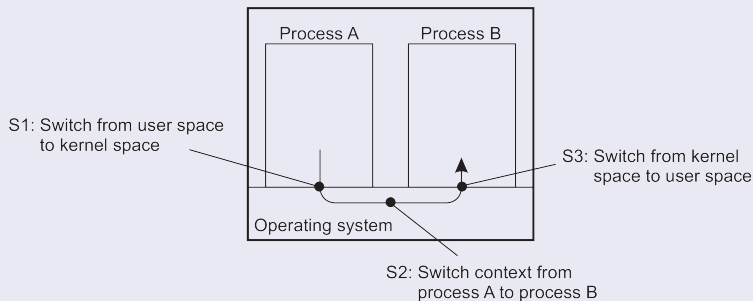
Why use threads

Some simple reasons

- **Avoid needless blocking:** a single-threaded process will **block** when doing I/O; in a multithreaded process, the operating system can switch the CPU to another thread in that process.
- **Exploit parallelism:** the threads in a multithreaded process can be scheduled to run in parallel on a multiprocessor or multicore processor.
- **Avoid process switching:** structure large applications not as a collection of processes, but through multiple threads.

Avoid process switching

Avoid expensive context switching



Trade-offs

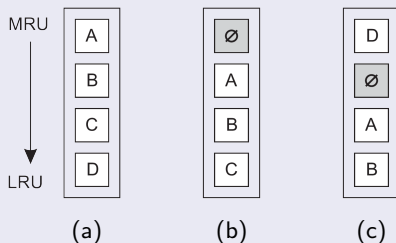
- Threads use the same address space: more prone to errors
- No support from OS/HW to protect threads using each other's memory
- Thread context switching may be faster than process context switching

The cost of a context switch

Consider a simple clock-interrupt handler

- **direct costs:** actual switch and executing code of the handler
- **indirect costs:** other costs, notably caused by messing up the cache

What a context switch may cause: indirect costs



(a) before the context switch

(b) after the context switch

(c) after accessing block *D*.

Threads and operating systems

Main issue

Should an OS kernel provide threads, or should they be implemented as user-level packages?

User-space solution

- All operations can be completely handled **within a single process** \Rightarrow implementations can be extremely efficient.
- **All** services provided by the kernel are done **on behalf of the process in which a thread resides** \Rightarrow if the kernel decides to block a thread, the entire process will be blocked.
- Threads are used when there are many external events: **threads block on a per-event basis** \Rightarrow if the kernel can't distinguish threads, how can it support signaling events to them?

Linux Kernel Threads

- **Task Struct Representation:** In the Linux kernel, threads are implemented as lightweight processes, each represented by a `task_struct` (defined in `include/linux/sched.h`), sharing memory but maintaining separate execution contexts for scheduling.
- **Scheduling with CFS:** The Completely Fair Scheduler (CFS) in `kernel/sched/fair.c` manages kernel threads, treating them as virtual processors and allocating CPU time fairly using a red-black tree.
- **POSIX Threads Integration:** User-space threads (e.g., via `pthread_create`) map to kernel threads, enabling Java's 1:1 threading model to leverage Linux's scheduling for efficient concurrency.

Using threads at the client side

Multithreaded web client

Hiding network latencies:

- Web browser scans an incoming HTML page, and finds that **more files need to be fetched**.
- **Each file is fetched by a separate thread**, each doing a (blocking) HTTP request.
- As files come in, the browser displays them.

Multiple request-response calls to other machines (RPC)

- A client does several calls at the same time, each one by a different thread.
- It then waits until all results have been returned.
- Note: if calls are to different servers, we may have a **linear speed-up**.

Multithreaded clients: does it help?

Thread-level parallelism: TLP

Let c_i denote the fraction of time that exactly i threads are being executed simultaneously.

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

with N the maximum number of threads that (can) execute at the same time.

Practical measurements

A typical Web browser has a TLP value between 1.5 and 2.5 \Rightarrow threads are primarily used for **logically organizing** browsers.

Using threads at the server side

Improve performance

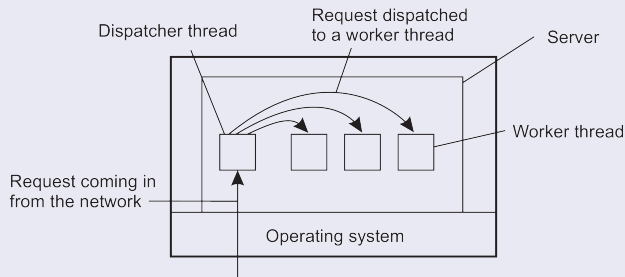
- Starting a thread is cheaper than starting a new process.
- Having a single-threaded server prohibits simple scale-up to a [multiprocessor system](#).
- As with clients: [hide network latency](#) by reacting to next request while previous one is being replied.

Better structure

- Most servers have high I/O demands. Using simple, [well-understood blocking calls](#) simplifies the structure.
- Multithreaded programs tend to be [smaller and easier to understand](#) due to [simplified flow of control](#).

Why multithreading is popular: organization

Dispatcher/worker model



Overview

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

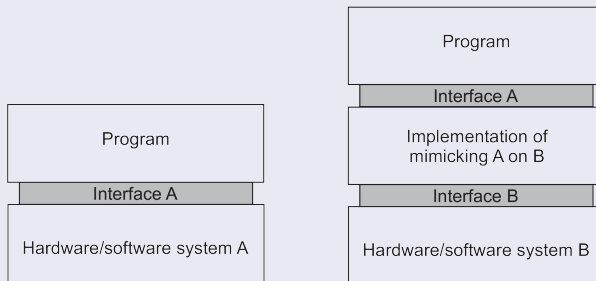
Virtualization

Observation

Virtualization is important:

- Hardware **changes faster** than software
- Ease of **portability** and code migration
- **Isolation** of failing or attacked components

Principle: mimicking interfaces

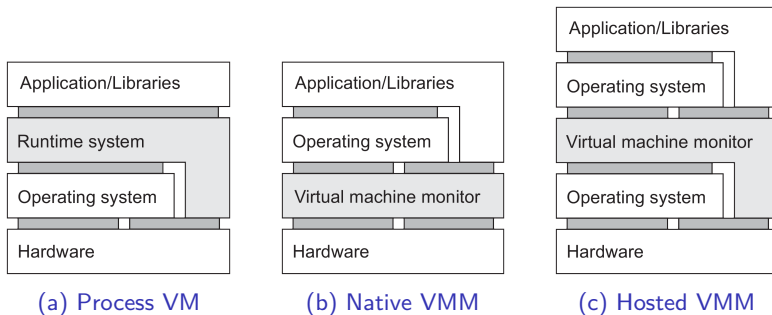


Mimicking interfaces

Four types of interfaces at three different levels

- ① **Instruction set architecture**: the set of machine instructions, with two subsets:
 - Privileged instructions: allowed to be executed only by the operating system.
 - General instructions: can be executed by any program.
- ② **System calls** as offered by an operating system.
- ③ **Library calls**, known as an **application programming interface** (API)

Ways of virtualization

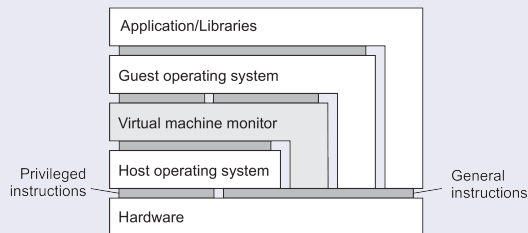


Differences

- (a) Separate set of instructions, an interpreter/emulator, running atop an OS.
- (b) Low-level instructions, along with bare-bones minimal operating system
- (c) Low-level instructions, but delegating most work to a full-fledged OS.

Zooming into VMs: performance

Refining the organization



- **Privileged instruction:** if and only if executed in user mode, it causes a **trap** to the operating system
- **Nonprivileged instruction:** the rest

Special instructions

- **Control-sensitive instruction:** may affect configuration of a machine (e.g., one affecting relocation register or interrupt table).
- **Behavior-sensitive instruction:** effect is partially determined by context (e.g., `POPF` sets an interrupt-enabled flag, but only in system mode).

Condition for virtualization

Necessary condition

For any conventional computer, a virtual machine monitor may be constructed if the set of sensitive instructions for that computer is a subset of the set of privileged instructions.

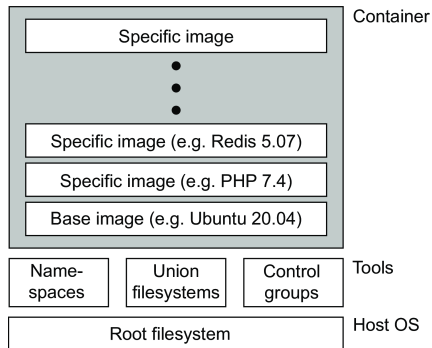
Problem: condition is not always satisfied

There may be sensitive instructions that are executed in user mode without causing a trap to the operating system.

Solutions

- Emulate all instructions
- Wrap nonprivileged sensitive instructions to divert control to VMM
- **Paravirtualization**: modify guest OS, either by preventing nonprivileged sensitive instructions, or making them nonsensitive (i.e., changing the context).

Containers



- **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**: resource restrictions can be imposed upon a collection of processes.

VMs and cloud computing

Three types of cloud services

- **Infrastructure-as-a-Service** covering the basic infrastructure
- **Platform-as-a-Service** covering system-level services
- **Software-as-a-Service** containing actual applications

IaaS

Instead of renting out a physical machine, a cloud provider will rent out a VM (or VMM) that may be sharing a physical machine with other customers \Rightarrow almost complete isolation between customers (although performance isolation may not be reached).

Virtual desktop environment

Logical development

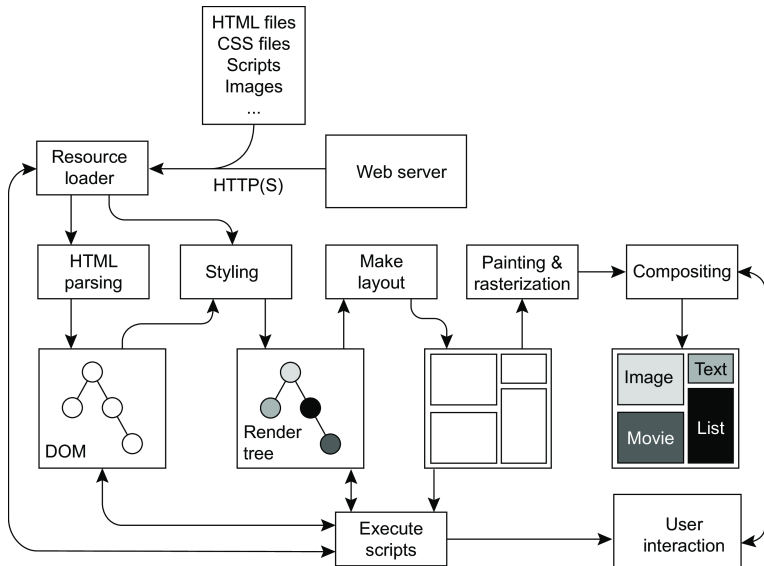
With an increasing number of cloud-based applications, the question is how to use those applications from a user's premise?

- **Issue:** develop the ultimate networked user interface
- **Answer:** use a Web browser to establish a seamless experience



The Google Chromebook

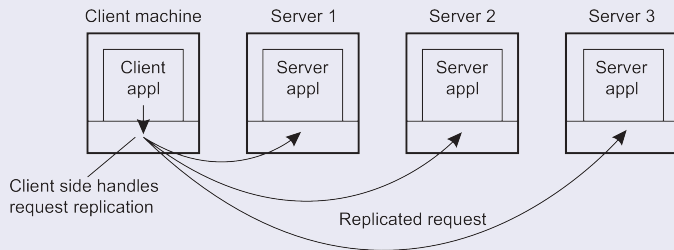
The anatomy of a Web browser



Client-side software

Generally tailored for distribution transparency

- **Access transparency:** client-side stubs for RPCs
- **Location/migration transparency:** let client-side software keep track of actual location
- **Replication transparency:** multiple invocations handled by client stub:



- **Failure transparency:** can often be placed only at client (we're trying to mask server and communication failures).

Servers: General organization

Basic model

A process implementing a specific service on behalf of a collection of clients. It waits for an incoming request from a client and subsequently ensures that the request is taken care of, after which it waits for the next incoming request.

Two basic types

- **Iterative server**: Server handles the request before attending a next request.
- **Concurrent server**: Uses a **dispatcher**, which picks up an incoming request that is then passed on to a separate thread/process.

Observation

Concurrent servers are the norm: they can easily handle multiple requests, notably in the presence of blocking operations (to disks or other servers).

Servers and state

Stateless servers

Never keep **accurate** information about the status of a client after having handled a request:

- Don't record whether a file has been opened (simply close it again after access)
- Don't promise to invalidate a client's cache
- Don't keep track of your clients

Consequences

- Clients and servers are **completely independent**
- **State inconsistencies** due to client or server crashes **are reduced**
- Possible **loss of performance** because, e.g., a server cannot anticipate client behavior (think of prefetching file blocks)

Question

Does connection-oriented communication fit into a stateless design?

Servers and state

Stateful servers

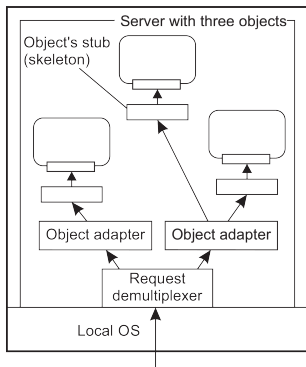
Keeps track of the status of its clients:

- Record that a file has been opened, so that prefetching can be done
- Knows which data a client has cached, and allows clients to keep local copies of shared data

Observation

The **performance of stateful servers can be extremely high**, provided clients are allowed to keep local copies. As it turns out, **reliability is often not a major problem**.

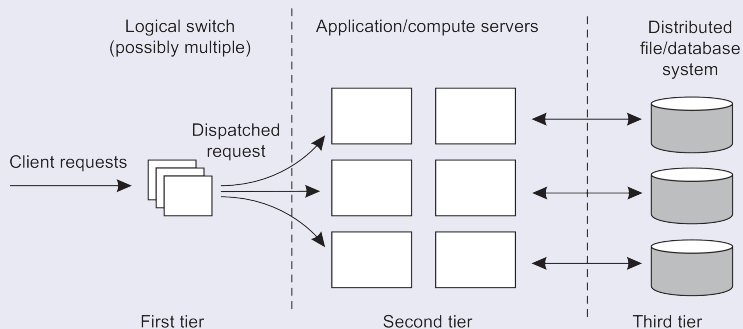
Object servers



- **Activation policy:** which actions to take when an invocation request comes in:
 - Where are code and data of the object?
 - Which threading model to use?
 - Keep modified state of object, if any?
- **Object adapter:** implements a specific activation policy

Three different tiers

Common organization



Crucial element

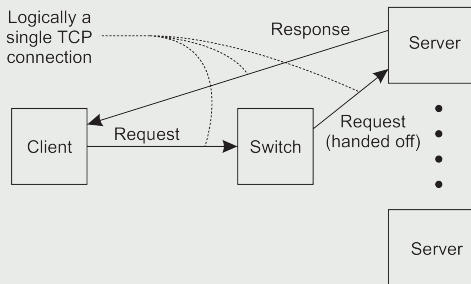
The first tier is generally responsible for passing requests to an appropriate server: [request dispatching](#)

Request Handling

Observation

Having the first tier handle all communication from/to the cluster may lead to a **bottleneck**.

A solution: TCP handoff



When servers are spread across the Internet

Observation

Spreading servers across the Internet may introduce administrative problems. These can be largely circumvented by using data centers from a single cloud provider.

Request dispatching: if locality is important

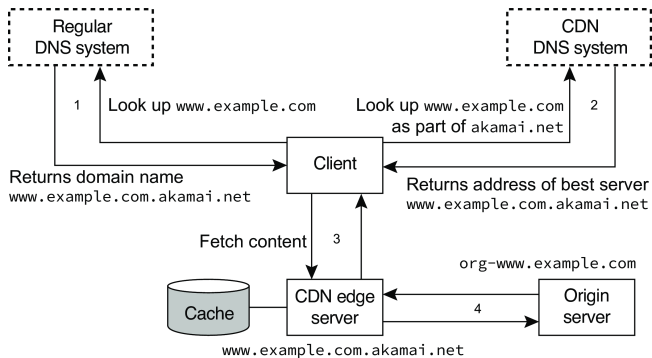
Common approach: use DNS:

- 1 Client looks up specific service through DNS - client's IP address is part of request
- 2 DNS server keeps track of replica servers for the requested service, and returns address of most local server.

Client transparency

To keep client unaware of distribution, let DNS resolver act on behalf of client. Problem is that the resolver may actually be **far from local** to the actual client.

A simplified version of the Akamai CDN



Important note

The cache is often sophisticated enough to hold more than just passive data. Much of the application code of the origin server can be moved to the cache as well.

Summary and Conclusions

We have discussed processes and threads in Distributed Systems, namely:

- Processes and Threads
- Context Switching
- Multithreading
- Virtualization
- Containerization
- VDE
- Stateful and Stateless Servers