Principles of Distributed Systems

inft-3507

Dr. J.Burns

ADA University

Autumn 2025

Section 3: Processes and Threads

This content is based on the following public resources: https://www.distributed-systems.net/index.php/books/ds4/

Processes and Threads

Introduction to Processes and Threads

- **Definition**: A process is an independent program in execution with its own memory space, while a thread is a lightweight unit of execution within a process, sharing the process's memory.
- **Memory**: A process has its own isolated memory space (address space), whereas threads within the same process share the same memory space, including code, data, and resources.
- **Overhead**: Processes are heavier, requiring more system resources and time for creation and context switching, while threads are lighter, with lower overhead for creation and switching.
- **Communication**: Inter-process communication (IPC) is complex and slower (e.g., pipes, sockets), while threads communicate faster via shared memory but require synchronization (e.g., locks).

Context switching

Observations

1. Threads share the same address space. Thread context switching is much faster than process context switching:
    1. only registers and program counter need to be saved and restored
2. Process context switching is more expensive (in time and space) as
    1. TLB needs to be flushed
    2. page table is reloaded
    3. address space changes
3. Creating and destroying threads is much cheaper than doing so for processes.

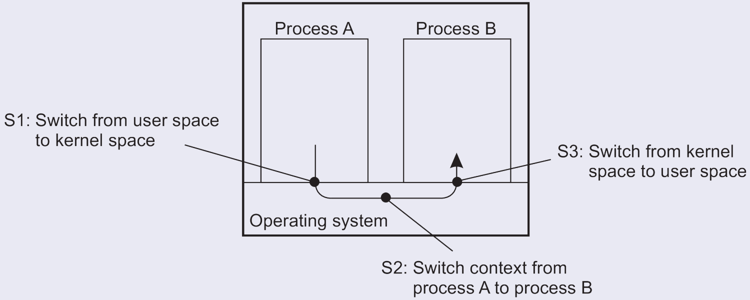
Why use threads

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

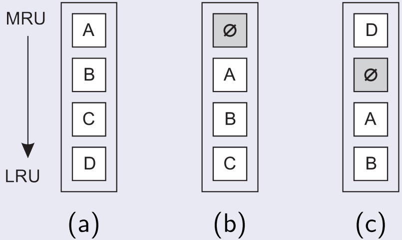
- Threads use the same address space: more prone to errors
- No support from OS/HW to protect threads using each other's memory
- Thread context switching may be faster than process context switching

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.

Thread Switch vs Process Context Switch

Thread Switch

- Switch between threads of the *same process*
- Address space remains unchanged
- Page tables and memory mappings are reused
- Faster than process switches
- Typically used for concurrency within applications

Overhead:

- Save/restore registers
- Update stack pointer and program counter

Process Context Switch

- Switch between threads of *different processes*
- Address space changes
- Page tables must be switched
- TLB often flushed or invalidated
- Slower than thread switches

Overhead:

- Save/restore registers
- Switch address space
- Update memory management state

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

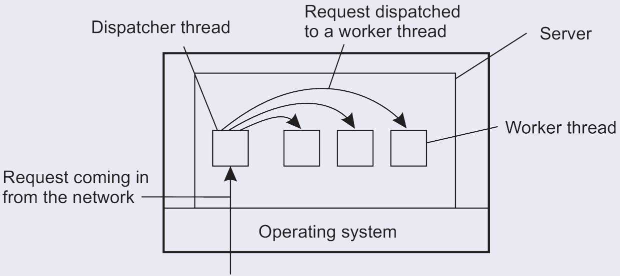
- Starting a thread is cheaper than starting a new process.
- Having a single-threaded server prohibits simple scale-up to a multiprocessor system.
- As with clients: hide network latency by reacting to next request while previous one is being replied.

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

Multithreading	Parallelism, blocking system calls
Single-threaded process	No parallelism, blocking system calls

Virtualization

Scope and Terminology

- **System virtualization:** presents a virtual hardware platform capable of running an OS
- **Process virtualization:** presents a virtual execution environment for a single program (e.g., a managed runtime)
- **Emulation:** reproduces an ISA and/or device behavior in software
- **Hypervisor / VMM:** software layer(s) that create and isolate system VMs

Goal

Provide isolation and controllable sharing of CPU, memory, and I/O while preserving expected software semantics.

Execution Virtualization Spectrum

- **Emulation** (ISA/device reproduction): maximizes compatibility across architectures, typically higher overhead
- **Process VM** (language/bytecode runtime): portability and safety for applications, not a full OS boundary
- **System VM** (hypervisor-based): strong isolation and full OS virtualization

Key distinction

Process VMs virtualize a *program execution model*; hypervisors virtualize a *machine interface*.

Diagram: Taxonomy of Virtualization Approaches

Emulation
- Virtualizes an ISA and/or devices in software
- Primary benefit: cross-architecture compatibility

Process Virtual Machine (e.g., JVM)
- Virtualizes an application execution model (bytecode + runtime services)
- Primary benefit: portability, safety properties, managed services

program boundar

System Virtual Machine (Hypervisor / VMM)
- Virtualizes CPU, memory, and devices to run an unmodified OS
- Primary benefit: strong isolation and resource control at machine granularity

OS boundary

Figure: Conceptual taxonomy. Emulation targets compatibility, process VMs target portable execution for programs, and system VMs target full OS virtualization and isolation.

System VMs: Type 1 and Type 2 Hypervisors

- **Type 1 (bare-metal):** hypervisor runs directly on hardware, commonly with a privileged management domain
- **Type 2 (hosted):** hypervisor/VMM runs atop a general-purpose host OS and leverages host drivers/services

Engineering focus

The practical security boundary is determined by the **trusted computing base (TCB)**: what must be trusted for isolation.

Diagram: Type 1 vs Type 2 Stack Organization

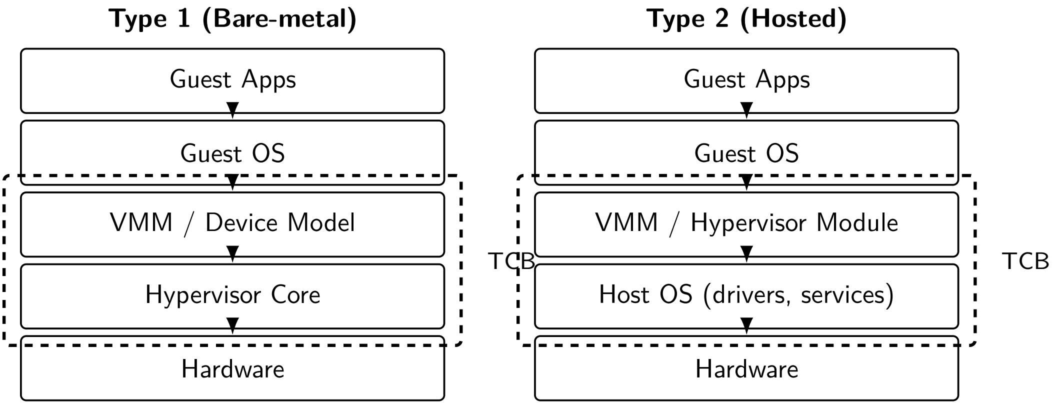


Figure: Representative stack organizations. In Type 2 systems, the host OS becomes part of the trusted base for isolation.

Emulation vs Hypervisor-Based Virtualization

- **Emulation:** can execute software for a different ISA; overhead arises from instruction translation and device modeling
- **System VM with hardware assist:** guest code runs largely natively; overhead concentrates in traps/exits and I/O paths
- **Process VM (e.g., JVM):** does not virtualize devices or privileged CPU state; relies on the host OS for isolation

Implication

ISA portability and OS-level isolation are different objectives; they are frequently conflated but have different mechanisms and costs.

Resource Control: CPU and Memory

CPU control concepts

- **Capacity:** number of vCPUs and their mapping to physical CPUs
- **Shares/weights:** proportional allocation under contention
- **Limits/caps:** enforce maximum CPU usage (often expressed as a percentage of a core set)
- **Reservations:** minimum guaranteed CPU capacity (where supported)

Memory control concepts

- **Static limit:** maximum assigned memory for a VM
- **Ballooning:** cooperative reclamation from guests to enable overcommit
- **Swapping/reclamation:** provider-side memory pressure mechanisms (risk of latency inflation)

Diagram: Resource Allocation and Enforcement Points

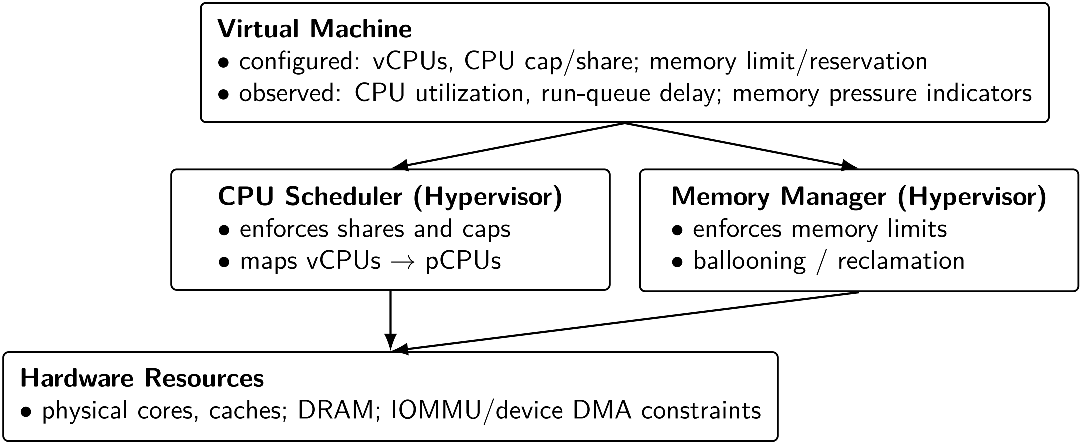


Figure: Enforcement points for CPU and memory control. Allocation policies are implemented by the hypervisor scheduler and memory manager, which multiplex physical resources across VMs.

Concrete Control Examples (CPU % and Memory %)

CPU

- **CPU cap (percentage):** limit a VM to (e.g.) 60% of one core or 240% of a 4-core entitlement
- **Shares/weights:** if two VMs have weights 2:1, they receive that proportion under contention
- **Pinning/affinity:** restrict vCPUs to selected cores to control interference and improve predictability

Memory

- **Memory limit:** hard upper bound on guest-usable memory
- **Reservation:** minimum memory intended to remain available (platform-dependent)
- **Overcommit with ballooning:** reclaim unused guest pages before swapping at the host

Summary

- Emulation, process VMs, and hypervisors address different requirements and expose different boundaries
- Type 1 and Type 2 hypervisors differ mainly in where the trusted base resides and how drivers are obtained
- Modern platforms treat CPU and memory as first-class, policy-controlled resources: shares, caps, reservations, and limits

Virtualization and Containers

Virtualization is about running workloads as if they had their own machine.

In Linux container-style virtualization, two kernel features provide most of the illusion:

- **Namespaces** $\rightarrow$ *isolation*: "what you can see"
- **Control groups (cgroups)** $\rightarrow$ *control*: "what you can use"

A **container** is typically: *namespaces* + *cgroups* (plus filesystem + tooling).
If multiple applications share one OS kernel, we want:

- Strong **separation of views** (processes, network, mounts, hostnames, users)
- Predictable **resource sharing** (CPU, memory, I/O) without one app starving others

Linux containers achieve this without a hypervisor by virtualizing *interfaces to the kernel*.

Namespaces: isolation of kernel resources

A **namespace** gives a process a private view of a specific kernel resource.

Common namespaces:

- **PID** (process IDs): processes see a different PID tree
- **Mount** (MNT): separate mount table / filesystem view
- **Network** (NET): interfaces, routes, ports isolated
- **UTS**: hostname/domainname isolation
- **IPC**: SysV IPC, POSIX message queues, etc.
- **User** (USER): user/group ID mapping (enables unprivileged containers)
- **Cgroup** namespace: hides cgroup paths

Control groups (cgroups): accounting and limits

cgroups group processes and apply resource *accounting, limits, and prioritization*.

Typical controls:

- **CPU**: shares/quotas; prevent CPU hogging
- **Memory**: limits; OOM behavior per group
- **I/O**: throttle block I/O
- **PIDs**: limit number of processes

Key idea: **namespaces isolate**; **cgroups constrain**.

How they fit together (container model)

A container runtime typically:

1. Creates new **namespaces** for the process (isolation)
2. Places the process into **cgroups** (resource governance)
3. Sets up a root filesystem and mounts (often with overlayfs)
4. Configures networking (veth pairs, bridges, NAT, etc.)

The result: a process that *looks* like it runs on its own system and can be *limited*.

Diagram: cgroups as a resource governance tree

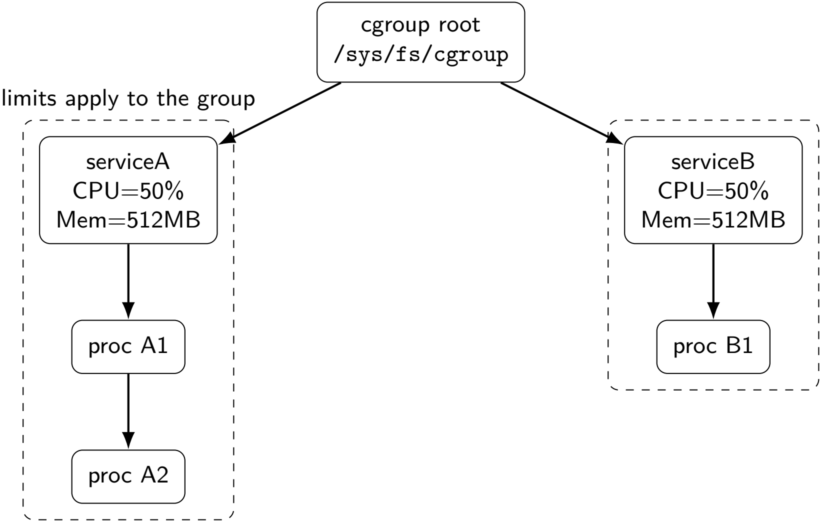


Figure: Cgroups organize processes into a hierarchy and apply resource limits/accounting per group.

Diagram: namespaces provide separate “views”

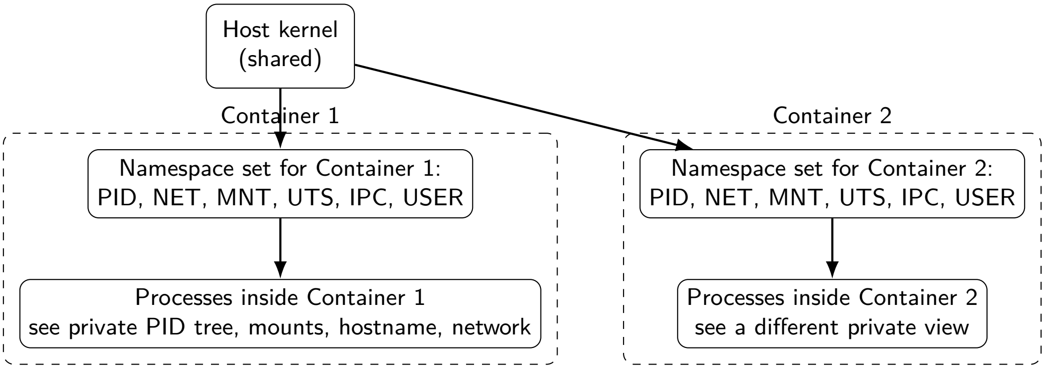


Figure: Namespaces isolate what processes can see; multiple containers can share one kernel while keeping distinct views.

A simple isolation example with `unshare`

`unshare` creates new namespaces for a process (and its children).

We'll demonstrate isolation by creating:

- a new **UTS namespace** (private hostname)
- a new **PID namespace** (PID 1 inside)
- a new **mount namespace** (private mount table)
- optionally a **user namespace** (run as "root" inside without host root)

Example: unshare for a private hostname + PID tree

```
# Create new user, mount, UTS, and PID namespaces.
# --map-root-user maps your user to root inside the user namespace.
unshare --user --map-root-user --mount --uts --pid --fork /bin/bash
```

Inside the new shell:

```
# UTS namespace: hostname is private
hostname container-demo

# PID namespace: this shell's child processes start a new PID tree
echo "my pid:" $$

# Compare process listing (you should see a tiny PID universe)
ps -ef
```

Example: mount namespace isolation (private mounts)

In the same `unshare` shell:

```
# Make mount operations "private" to this namespace
mount --make-rprivate /

# Create a temporary mount only visible here
mkdir -p /tmp/ns-mount
mount -t tmpfs tmpfs /tmp/ns-mount

# Observe it exists here:
mount | grep ns-mount
```

On the host (outside the namespace), that tmpfs mount will not appear in `mount` output.

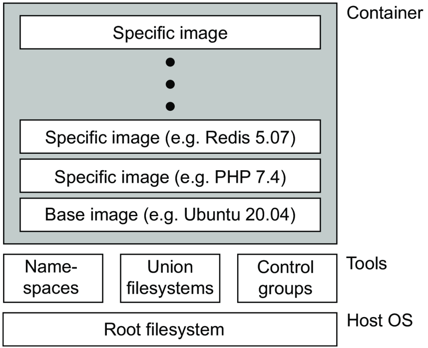
What isolation did we get?

From the example:

- **UTS namespace**: hostname change does not affect the host
- **PID namespace**: processes see a new PID hierarchy (often with PID 1 inside)
- **Mount namespace**: mounts are private to the namespace
- **User namespace**: you can have root-like IDs inside without being host root

This is the core of container isolation, and it ties directly back to virtualization:
virtualize the process's view of the system without virtualizing the hardware.

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.

Summary

- Virtualization can be done at different layers: hardware (VMs) vs. OS interfaces (containers).
- **Namespaces** isolate what processes can see (views of kernel resources).
- **cgroups** govern what processes can use (resource limits/accounting).
- `unshare` is a minimal, direct way to see namespace isolation in action.

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).

Summary

Summary and Conclusions

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

- Processes and Threads
- Context Switching
- Multithreading
- Virtualization
- Containerization