

UNIT 3

Processes

Distributed Systems (CACS352) — BCA III/VI

⚠ Please note

*These notes were prepared with AI assistance and **may contain errors or omissions**. Please cross-check definitions, algorithms and numerical examples against your prescribed textbook (e.g., Tanenbaum & van Steen, *Distributed Systems*), your class notes and your instructor before relying on them for exams.*

3.1 Threads

A **process** is a program in execution with its own address space. A **thread** is a lightweight unit of execution inside a process: threads of the same process share the address space, open files and global data, but each has its own program counter, registers and stack. Creating and switching threads is far cheaper than doing the same for processes, since no memory-management context has to be changed.

Aspect	Process	Thread
Address space	Own, protected	Shared with other threads of the process
Creation / context switch	Expensive (MMU state saved)	Cheap
Communication	IPC (pipes, sockets, shared memory)	Direct through shared data (needs synchronisation)
Protection	Strong between processes	None between threads of one process

Threads in distributed systems

- **Multithreaded clients** — hide network latency: a Web browser can render a page while other threads fetch images; the process is not blocked waiting for slow servers.
- **Multithreaded servers** — handle many requests concurrently and exploit multiprocessors. Blocking system calls in one thread do not stop the rest.

Server organisations

Model	Description	Properties
Dispatcher / worker	A dispatcher thread reads incoming requests and hands each to an idle worker thread	Parallelism, blocking calls per thread; most common
Team model	All threads are equal; each fetches and processes requests itself	Simple; needs queue synchronisation
Pipeline	Output of one thread is input of the next	Good for staged processing
Single-threaded (finite-state machine)	One thread with non-blocking calls and events	No parallelism, but no locking; harder to program

Thread implementation

- **User-level threads** — implemented by a library; very cheap switching, but a blocking system call blocks the whole process and threads cannot use several CPUs.
- **Kernel-level threads** — managed by the OS; handle blocking calls and multiprocessors but every operation costs a system call.
- **Hybrid (lightweight processes, LWP)** — user threads are multiplexed onto kernel threads.

Example 3.1 — Benefit of multithreading in a file server

A file server thread needs 15 ms of CPU time per request. If the data is in cache (probability 2/3) no disk access is needed; otherwise a disk read of 75 ms is required (the thread blocks). Compare the throughput of a single-threaded and a multithreaded server.

Solution

Average time per request (single-threaded) = $15 + (1/3) \times 75 = 15 + 25 = 40 \text{ ms}$ → throughput = $1000/40 = 25 \text{ requests/s}$.

With enough threads, the CPU overlaps computation with disk waits, so the CPU is the bottleneck: 15 ms per request → $1000/15 \approx 66.7 \text{ requests/s}$.

Multithreading gives $\approx 2.7\times$ higher throughput here.

3.2 Virtualization

Virtualization extends or replaces an existing interface so that it mimics the behaviour of another system. It allows several isolated **virtual machines** (VMs) or environments to share one physical machine.

Why virtualize?

- Hardware changes faster than software — legacy software can run on new hardware.
- Isolation and better utilisation of servers (server consolidation).
- Portability and easy migration of whole environments; foundation of cloud computing (IaaS).

Architectures of virtual machines

Computer systems offer four interface levels: hardware instruction set (ISA), system calls, library calls (API) and the user-level. Virtualization can happen at each:

- **Process virtual machine** — a runtime system interprets or emulates a program's instructions (e.g., the Java Virtual Machine, .NET CLR).
- **Native (Type 1) virtual machine monitor** — the hypervisor runs directly on hardware and hosts guest operating systems (Xen, VMware ESXi, KVM).
- **Hosted (Type 2) virtual machine monitor** — the hypervisor runs as an application on a host OS (VirtualBox, VMware Workstation).

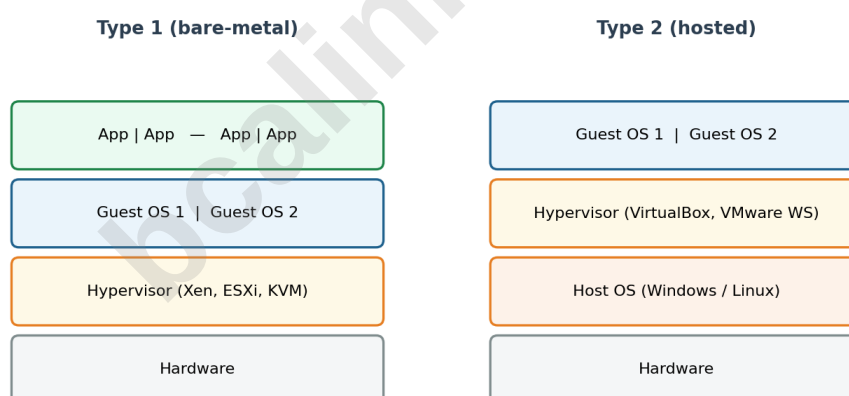


Figure 3.1 — Type 1 (bare-metal) and Type 2 (hosted) hypervisors.

Containers

Containers (Docker, LXC) virtualise at the *operating-system* level: many isolated user-space instances share one kernel, using namespaces for isolation and cgroups for resource limits. They start in seconds and use less memory than full VMs but provide weaker isolation.

Feature	Virtual machine	Container
Isolation level	Hardware (own guest OS)	OS-level (shared kernel)

Feature	Virtual machine	Container
Start-up time	Minutes	Seconds or less
Size / overhead	GBs, higher	MBs, lower
Security isolation	Stronger	Weaker

3.3 Clients

Networked user interfaces

Two ways to build a client: (1) an application-specific **client program** that knows the remote protocol (fat client), or (2) a general **thin client** that only renders a user interface, e.g., the **X Window System** — the *X server* runs on the user's machine (controls screen/keyboard/mouse) and *X clients* (applications) may run remotely, talking through the X protocol. Thin clients reduce management costs but can suffer latency.

Client-side software for distribution transparency

- **Access transparency** — a **client stub** hides differences in machine architecture and marshals parameters.
- **Location / migration / relocation transparency** — the client middleware keeps track of the server's current location and rebinds after moves.
- **Replication transparency** — a client-side proxy forwards a request to all replicas and returns one reply.
- **Failure transparency** — the client middleware can retry, or switch to another server.

3.4 Servers

General design issues

- **Iterative server** — handles one request at a time; **concurrent server** — passes each request to a separate thread or process (or a pooled worker).
- **Endpoints (ports)** — clients contact a server at an address plus a port. Well-known ports (e.g., HTTP 80, HTTPS 443, SSH 22, DNS 53) are fixed; otherwise a **daemon** (e.g., DCE daemon) keeps an endpoint table.
- **Superservers** — a single listening process (e.g., *inetd*) waits on many ports and starts the right service on demand.

- **Interrupting a server** — out-of-band data or a separate control connection allows a user to abort a long-running request.
- **Stateless vs stateful** — a stateless server keeps no client state (crash recovery is easy, e.g., HTTP, NFSv3); a stateful server records client information (better performance, but state may be lost on crash and needs recovery — e.g., file servers that track open files).

Server clusters

Large services run on **clusters** organised in three tiers: a logical **switch/load balancer** (first tier) that forwards requests, an **application-processing** tier and a **distributed file/database system** tier. Load balancers may operate at the transport layer (TCP handoff) or the application layer (content-aware). Server clusters can be locally or geographically distributed; in the latter, DNS or anycast redirects clients to a nearby cluster.

Managing server clusters

Administration tools monitor nodes and services, remove failed nodes and add new ones. Examples: cluster managers such as Kubernetes (schedules and heals containerised services) and Google's Borg.

3.5 Code Migration

Code migration moves programs — and sometimes their execution state — between machines. Reasons: (1) **performance** — move computation to where the data is or to a lightly loaded machine (load sharing); (2) **flexibility** — clients download code for new protocols (e.g., JavaScript, applets) and configure dynamically.

Models for code migration

A process consists of three segments: the **code segment** (instructions), the **resource segment** (references to external resources such as files, devices) and the **execution segment** (state: stack, program counter, private data).

- **Weak mobility** — only the code segment (plus initialisation data) is moved; execution restarts from the beginning (e.g., Java applets, JavaScript).
- **Strong mobility** — the execution segment is moved as well; the process resumes exactly where it stopped.
- **Sender-initiated** — the machine holding the code starts the migration (e.g., uploading a program to a server); **receiver-initiated** — the target starts it (e.g., a browser downloading an applet).

Resource binding	Meaning	Example
By identifier	Needs exactly the same resource	A URL of a specific Web site
By value	Needs a resource with the same value	A standard library
By type	Needs a resource of the same type	A local printer or monitor

Note: Combined with resource types (unattached, fastened, fixed) these bindings decide whether a resource is moved, copied, re-bound, or referenced across the network during migration.

Migration in heterogeneous systems

Code compiled for one architecture will not run on another, and execution state is machine-specific. Solutions: use **interpreted languages** or **process virtual machines** (JVM), or migrate whole **virtual machines** — *live migration* copies memory pages while the VM keeps running and pauses it only for a final short switch (used by Xen/VMware/KVM for load balancing and maintenance).

Key Takeaways

- ✓ Threads share an address space; they are cheaper than processes and let clients and servers overlap computation and I/O.
- ✓ Server models: dispatcher/worker, team, pipeline, single-threaded FSM; user-level, kernel-level and hybrid threads.
- ✓ Virtualization: process VMs, Type 1 and Type 2 hypervisors, and OS-level containers.
- ✓ Clients hide distribution (stubs, proxies) and thin clients such as X move the UI to the user's machine.
- ✓ Servers may be iterative or concurrent, stateless or stateful; large services use load-balanced clusters.
- ✓ Code migration: weak vs strong mobility, sender- vs receiver-initiated; live VM migration solves heterogeneity.

— End of Unit 3 —