4.4 RESOURCE LIMITS AND PROCESS ISOLATION

4.4. 1. Introduction

Modern computing systems run multiple applications on shared physical hardware, especially in cloud environments, data centers, and enterprise servers. To ensure that these applications run efficiently, securely, and without interference, technologies such as virtualization and containerization make use of two fundamental concepts: **resource limits** and **process isolation**. Resource limits define how much CPU power, memory, disk I/O, or network bandwidth a workload is allowed to use. Without these limits, a single application can consume excessive resources, causing performance degradation or system instability.

Process isolation ensures that applications are separated from one another, preventing one application's actions from affecting others. This is extremely important for security, stability, and multi-tenant computing. Virtualization provides isolation at the hardware level using hypervisors and separate operating systems, while container technologies provide lightweight isolation using the Linux kernel. Together, resource limits and process isolation form the foundation of efficient multi-application environments.

4.4.2. Virtualization Technologies

Virtualization creates multiple **Virtual Machines (VMs)** on a single physical server. Each VM behaves like a complete computer with its own operating system, kernel, virtual hardware, and applications. This is achieved through a software layer called the **hypervisor**.

4.4.2.1 Hypervisors

Hypervisors are classified into two types:

- **Type-1 Hypervisors (Bare-metal):** Installed directly on physical hardware. *Examples: VMware ESXi, Microsoft Hyper-V, Xen.*
- **Type-2 Hypervisors (Hosted):** Installed on top of a host operating system. *Examples: VMware Workstation, VirtualBox.*

These hypervisors divide hardware resources such as CPU cores, RAM, storage, and network bandwidth into isolated virtual environments.

4.4.2.2 Process Isolation in Virtualization

Virtualization provides **strong isolation** because each VM runs a separate OS kernel and has independent virtual hardware. A failure or attack inside one VM cannot directly affect another.
Isolation is achieved through the following mechanisms:

Mechanisms of Isolation

- **Virtual CPUs (vCPUs):** Each VM is assigned a specific number of vCPUs that map to physical CPU cores.
- **Memory Virtualization:** Uses hardware support like Extended Page Tables (Intel EPT) or Nested Page Tables (AMD NPT) to isolate RAM usage.
- **Storage and I/O Virtualization:** Each VM is given virtual disks, virtual network interfaces, and virtual storage devices.
- **Hypervisor Scheduling:** The hypervisor decides when and how long each VM can use CPU or I/O resources.

Benefits

- Very strong security boundaries
- Complete fault isolation
- Ability to run different operating systems (Linux & Windows together)

Limitations

- High resource overhead
- Slower startup time
- Requires full OS per VM

4.4.2.3 Resource Limits in Virtualization

Hypervisors enforce limits to ensure fairness and predictable performance.

CPU Limits

- **CPU Shares:** Assigns priority among VMs.
- **CPU Reservation:** Guarantees minimum CPU availability.
- **CPU Cap:** Limits maximum CPU usage.
- **vCPU Pinning:** Maps VM CPUs to particular physical cores.

Memory Limits

- **Static Allocation:** Fixed memory assigned to a VM.
- **Memory Ballooning:** Dynamically adjusts memory under load.
- **Memory Reservation:** Guarantees a minimum amount.
- **Memory Overcommit:** Allows VMs to request more memory than physically available (handled carefully).

Storage I/O Limits

- Restriction on **IOPS** (Input/Output Operations Per Second)
- Bandwidth caps
- Quality of Service (QoS) policies

Network Limits

- Traffic shaping
- Rate limiting
- Assigning priority queues

These features prevent one VM from consuming all resources, ensuring stable multi-tenant performance.

4.4.3. Container Technologies

Containerization offers a lightweight alternative to virtualization. Instead of running a full OS for each workload, containers share the host OS kernel but use isolation techniques to create separate environments.
Examples: **Docker, Kubernetes, LXC, Podman**.

Containers are widely used for microservices, DevOps automation, and cloud-native applications due to their speed and low overhead.

4.4.3.1 Process Isolation in Containers

Containers rely on **Linux Namespaces** to isolate system resources.

Types of Namespaces

- **PID Namespace:** Isolates process IDs; each container sees only its processes.
- **Mount Namespace:** Provides separate filesystem views.
- **Network Namespace:** Each container has its own IP address and routing.
- **IPC Namespace:** Isolates shared memory and message queues.
- **UTS Namespace:** Provides separate hostnames.
- **User Namespace:** Maps container users to host users, improving security.

How Isolation Works

Even though containers share the same kernel, namespaces ensure that each container behaves like an independent system. This makes containers efficient for scaling and fast deployments.

4.4.3.2 Resource Limits in Containers

Containers use **cgroups (Control Groups)** to enforce resource limits.

CPU Limits

- **cpu.shares:** Relative CPU priority
- **cpu.quota & cpu.period:** Strict CPU limit
- **CPU Pinning:** Bind containers to specific CPU cores

Example:

```
docker run --cpus="1.5" ubuntu
```

Memory Limits

- Maximum memory allocation
- Swap usage restrictions
- Memory reservation for guaranteed availability

Example:

```
docker run -m 512m ubuntu
```

I/O Limits

- Restricts disk read/write speed
- Limits on IOPS

Network Limits

- Bandwidth limits
- Traffic shaping (using tc)
- Kubernetes network policies

Containers are extremely fast and efficient, but because they share the same kernel, their isolation strength is lower compared to VMs.

Comparison Between Virtualization and Containers

Feature	**Virtual Machines**	**Containers**
Isolation Strength	Very strong (hardware level)	Moderate (kernel level)
Performance Overhead	High	Very low
Startup Time	Slow (seconds–minutes)	Very fast (milliseconds)
kernel Sharing	No	Yes
Resource Efficiency	Less	Very high
Best Use-case	Strong security, legacy OS	Microservices, cloud-native apps

Containers provide speed and efficiency, while VMs provide strong security and OS-level flexibility.

Importance of Resource Limits & Process Isolation

Resource limits and isolation are critical in all multi-application computing systems. They ensure:

- **Prevention of Resource Starvation**
  No single workload can consume all CPU or memory.
- **Security and Protection**
  One compromised workload cannot access another.
- **Stability and Reliability**
  Failures do not spread across VMs or containers.
- **Multi-Tenant Cloud Support**
  Different customers can safely share the same hardware.
- **Predictable Performance**
  Critical applications receive guaranteed resources.

Real-World Use Cases

- **Cloud Platforms (AWS, Azure, GCP)**
  VM instance types use fixed resource limits; Kubernetes manages container limits.
- **DevOps and CI/CD Systems**
  Containers provide isolated build and testing environments.
- **High-Density Compute Clusters**
  Containers allow packing many applications efficiently.
- **Security-Sensitive Workloads**
  VMs are preferred due to strong isolation.

Resource limits and process isolation play an essential role in virtualization and container technologies. Virtualization provides strong hardware-level isolation using hypervisors, while containers provide lightweight isolation using Linux namespaces and cgroups. Both technologies complement each other and are used extensively in modern cloud computing environments to ensure performance, security, and efficient resource utilization. These concepts enable safe multi-tenant environments, predictable system behavior, and the ability to run diverse applications on shared hardware without interference. They form the backbone of scalable computing in modern data centres, cloud platforms, and enterprise infrastructures.