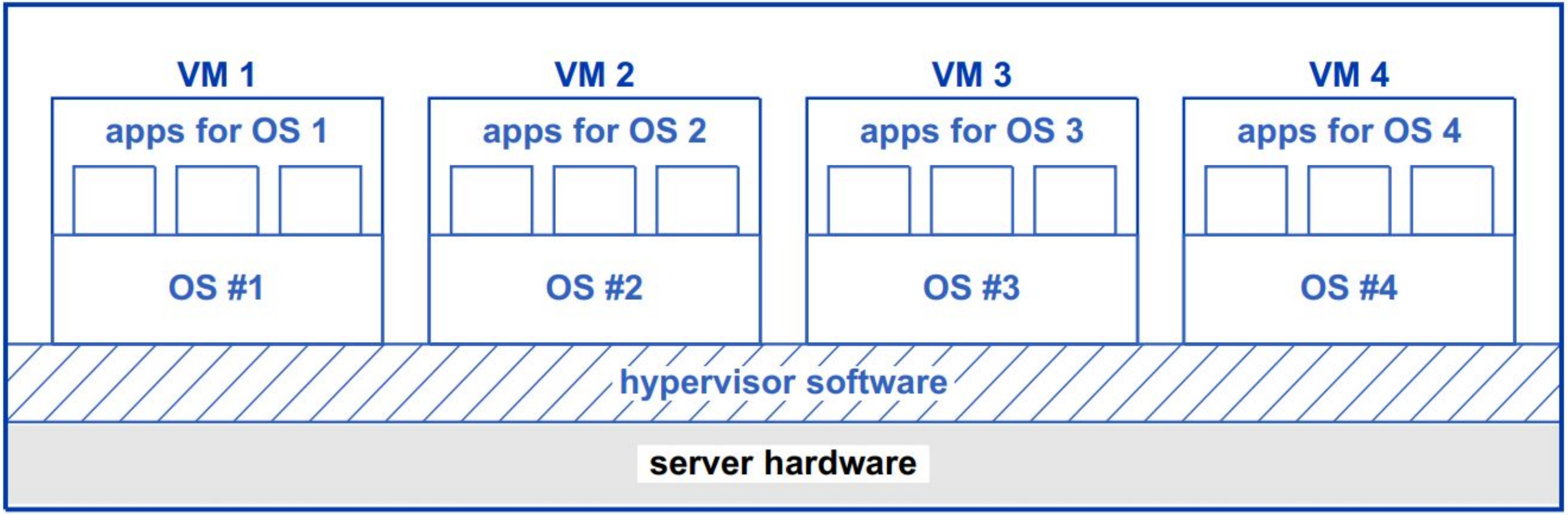
Containers

The Advantages and Disadvantages of VMs

- To understand the motivation for containers and allow both the operating system code and apps running in a VM to execute at hardware speed.
- We must consider the advantages and disadvantages of VMs.
- The chief advantage of the VM approach lies in its support of arbitrary operating systems.
- VM technology virtualizes processor hardware and creates an emulation so close to an actual processor that a conventional operating system built to run directly on hardware can run inside a VM with no change.
- Once a hypervisor has been loaded onto a server, the hypervisor can create multiple VMs and arrange for each to run an operating system that differs from the operating systems running in other VMs.
- Once an operating system has been started in a VM, apps built to use the operating system can run unchanged.
- Thus, a cloud customer who leases a VM has the freedom to choose all the software that runs in the VM, including the operating system.

- VM technology also has some disadvantages. Creating a VM takes time. In particular, VM creation requires booting an operating system.
- Furthermore, the VM approach places computational overhead on the server. Consider, for example, a server that runs four VMs, each with its own operating system.
- Figure 6.1 illustrates the configuration four VMs each running their own operating system.



- In the figure, four operating systems are running on the server, and each injects two forms of overhead.
- First, each operating system runs a scheduler that switches the processor among the apps it runs. Instead of the overhead from one scheduler, a server running VMs experiences scheduling overhead from each of the four.
- Second, and most important, operating systems run additional background processes to handle tasks for apps, such as processing network packets.
- Instead of one set of background processes, a server that runs VMs will run background processes for each VM.
- As additional VMs are added to a server, the overhead increases.

Traditional Apps and Elasticity on Demand

- VM technology works well in situations where a virtual server persists for a long time (e.g., days) or a user needs the freedom to choose an operating system.
- In many instances, however, a user only runs a single application and does not need all the facilities in an operating system.
- A user who only runs a single app can use a cloud service to handle rapid elasticity — the number of copies of the app can increase or decrease quickly as demand rises and falls.
- In such cases, the overhead of booting an operating system makes VM technology unattractive.
- The question arises: “Can an alternative virtualization technology be devised that avoids the overhead of booting an operating system?”

- A conventional operating system includes a facility that satisfies most of the need: support for concurrent processes.
- When a user launches an app, the operating system creates a process to run the app, and process creation takes much less time than booting an operating system.
- When a user decides to shut down an app, the operating system can terminate a process quickly.
- One can imagine a cloud system that allows customers to deploy and terminate copies of an app.
- Such a facility would consist of physical servers each running a specific operating system.
- When a customer requests an additional copy of a particular app, the provider uses software to choose a lightly-loaded server and launch a copy of the app.
- Similarly, when a customer requests fewer copies, the provider uses software to locate and terminate idle copies.
- Unfortunately, apps running on an operating system do not solve the problem completely because an operating system does not ensure complete isolation among apps run by multiple tenants.
- In particular, most operating systems assume apps will share network access. That is, an operating system obtains an Internet address and allows all apps to use the address.
- An operating system usually provides a file system that all apps share.
- In addition to sharing, most operating systems allow a process to obtain a list of other processes and the resources they have consumed, which means a tenant might be able to deduce what other tenants' apps are doing.

Isolation Facilities in an Operating System

- Starting with the earliest systems, operating systems designers have discovered ways to isolate the computations and data owned by one user from those of another.
- Most operating systems use virtual memory hardware to provide each process with a separate memory address space and ensure that a running app cannot see or alter memory locations owned by other apps.
- User IDs provide additional isolation by assigning an owner to each running process and each file, and forbidding a process owned by one user from accessing or removing an object owned by another user.
- However, the user ID mechanism in most operating systems does not scale to handle a cloud service with arbitrary numbers of customers.
- Therefore, the operating systems community has continued to investigate other ways to isolate running apps.

Linux Namespaces used for Isolation

- Some of the most significant advances in isolation mechanisms have arisen in the open source community; cloud computing has spurred adoption.
- Under various names, such as *jails*, the community has incorporated isolation mechanisms into the Linux operating system.
- Known as *namespaces*, a current set of mechanisms can be used to isolate various aspects of an application. Figure 6.2 lists seven major namespaces used with containers.

Namespace	Controls
Mount	File system mount points
Process ID	Process identifiers
Network	Visible network interfaces
Interprocess communication	Process-to-process communication
User ID	User and group identifiers
UTS	Host and domain names
Control group	A group of processes

Figure 6.2 The seven major namespaces in the Linux operating system that can be used to provide various forms of isolation.

- The mechanisms control various aspects of isolation. For example, the process ID namespace allows each isolated app to use its own set of process IDs 0, 1, 2..., and so on.
- Instead of all processes sharing the single Internet address that the host owns, the network namespace allows a process to be assigned a unique Internet address.
- More important, the application can be on a different virtual network than the host operating system.
- Similarly, instead of all processes sharing the host's file system, the mount namespace allows a process to be assigned a separate, isolated file namespace
- Finally, a control group namespace allows one to control a group of processes.
- One of the most interesting aspects of namespace isolation arises because the mechanisms block access to administrative information about other isolated applications running on the same host.
- For example, the process ID namespace guarantees that if an isolated application X attempts to extract information about a process with ID *p*, the results will only refer to processes in X's isolated environment, even if one or more of the other isolated apps running on the same host happen to have a process with ID *p*.

The Container Approach For Isolated Apps

- When an app uses operating system mechanisms to enforce isolation, the app remains protected from other apps. Conceptually, we think of the app as running in its own environment, surrounded by walls that keep others out. Industry uses the term *container* to capture the idea of an environment that surrounds and protects an app while the app runs.
- At any time, a server computer runs a set of containers, each of which contains an app. Figure 6.3 illustrates the conceptual organization of software on a server that runs containers.

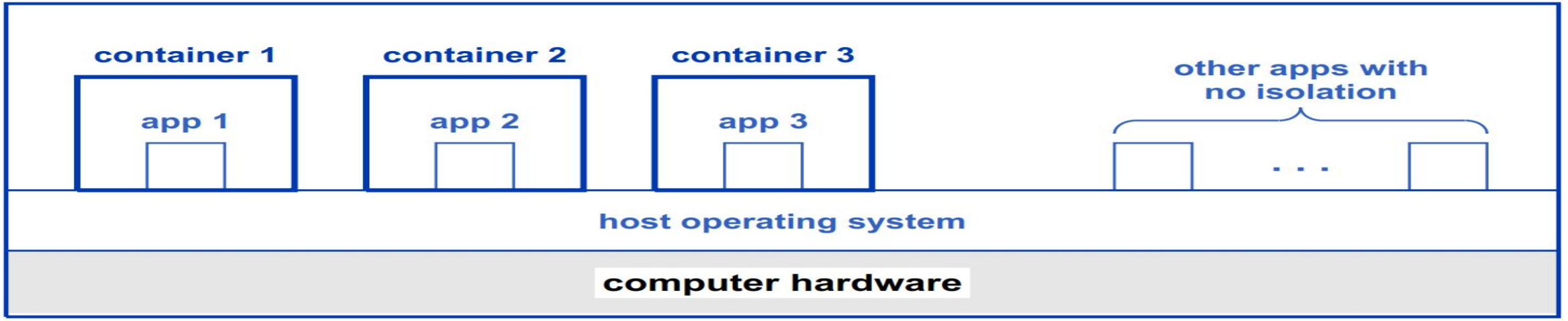


Figure 6.3 The conceptual organization of software when a server computer runs a set of containers.

- As the figure illustrates, it is possible to run conventional apps outside of containers at the same time as containers.
- Although conventional, unprivileged apps cannot interfere with apps in containers, a conventional app may be able to obtain some information about the processes running in containers.
- Therefore, systems that run containers usually restrict conventional apps to the control software used to create and manage containers.

Docker Containers

- One particular container technology has become popular for use with cloud systems. The technology resulted from an open source project known as *Docker*†. The
  Docker approach has become prominent for four main reasons:
  d Tools that enable rapid and easy development of containers
  d An extensive registry of software for use with containers
  d Techniques that allow rapid instantiation of an isolated app
  d Reproducible execution across hosts

Development tools

The Docker technology provides an easy way to develop apps that can be deployed in an isolated environment.

Unlike a conventional programming system in which a programmer must write significant pieces of code, Docker uses a high-level approach that allows a programmer to combine large pre-built code modules into a single image that runs when a container is deployed.

The Docker model does not separate a container from its contents.

That is, one does not first start a Docker container running and then choose an app to run inside the container.

Instead, a programmer creates all the software needed for a container, including an app to run, and places the software in an image file.

A separate image file must be created for each app.

When an image file runs, a container is created to run the app.
We say the app has been *containerized*.

Extensive registry of software

In addition to the basic tools programmers can use to create software for containers, the Docker project has produced *Docker Hub*, an extensive registry of open source software that is ready to use.

The registry enables a programmer to share deployable apps, such as a web server, without writing code.

More important, a user (or an operator) can combine pieces from the registry in the same way that a conventional program uses modules from a library.

Rapid instantiation

Because a container does not require a full operating system, a container is much smaller than a VM.

Consequently, the time required to download a container can be an order of magnitude less than the time required to download a VM.

In addition, Docker uses an early binding approach that combines all the libraries and other run-time software that will be needed to run the container into an image file.

Unlike a VM that must wait for an operating system to boot, a container can start instantly.

Furthermore, unlike a conventional app that may require an operating system to load one or more libraries dynamically, the early binding approach means a Docker container does not need the operating system to perform extra work when the container starts.

As a result, the time required to create a container is impressively small.

Reproducible execution

Once a Docker container has been built, the container image becomes *immutable* —the image remains unchanged, independent of the number of times the image runs in a container.

Furthermore, because all the necessary software components have been built in, a container image performs the same on any system.

As a result, container execution always gives reproducible results.