Distributed Systems
Operating System Support

Chapter 7:

7.1 Introduction

7.2 The operating system layer

7.4 Processes and threads

7.5 Communication and invocation

7.6 operating system architecture

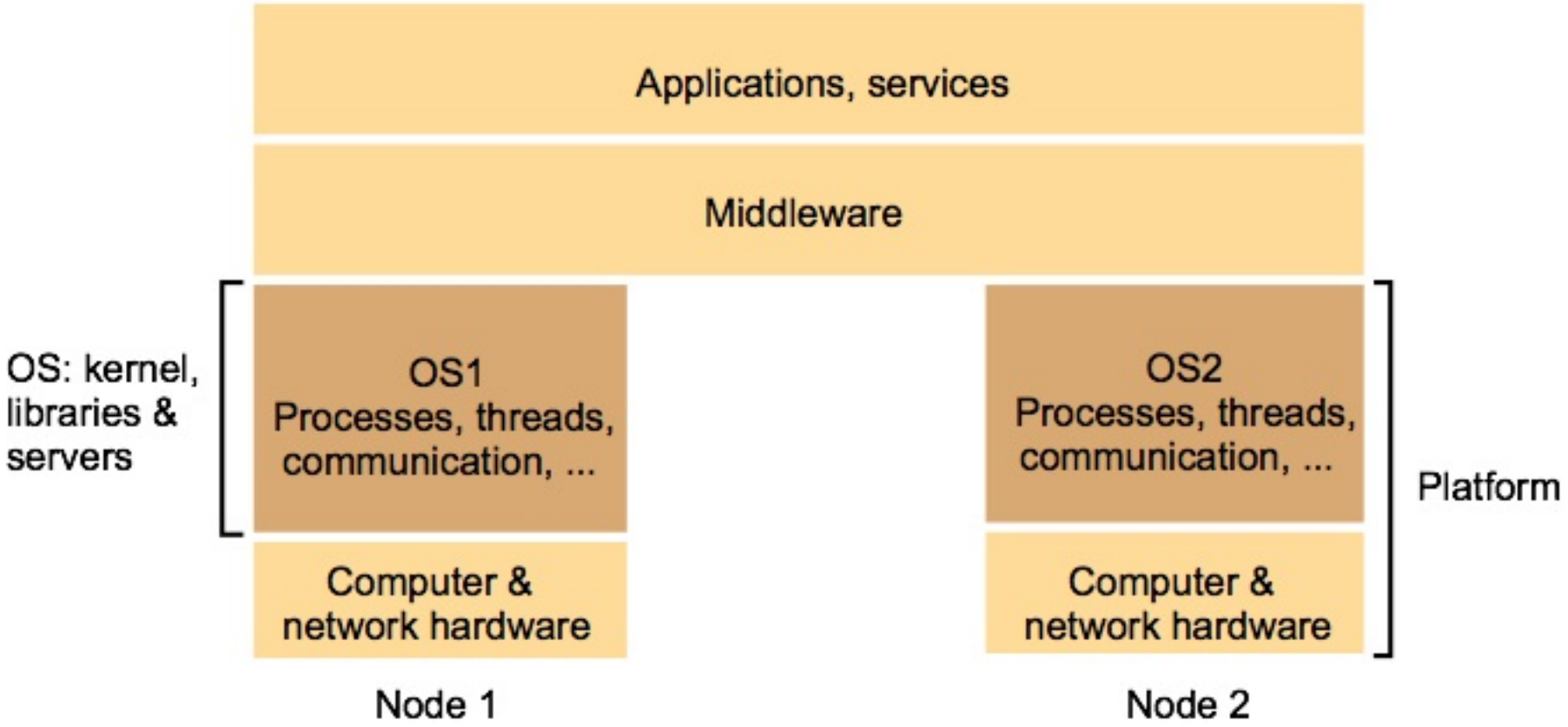
Viewing: These slides must be viewed in slide show mode.

Learning objectives

- Know what a modern operating system does to support distributed applications and middleware
  - Definition of network OS
  - Definition of distributed OS
- Understand the relevant abstractions and techniques, focussing on:
  - processes, threads, ports and support for invocation mechanisms.
- Understand the options for operating system architecture
  - monolithic and micro-kernels
- If time:
  - Lightweight RPC

System layers

Figure 7.1



*

Middleware and the Operating System

- Middleware implements abstractions that support network-wide programming. Examples:
  - *RPC and RMI (Sun RPC, Corba, Java RMI)*
  - *event distribution and filtering (Corba Event Notification, Elvin)*
  - *resource discovery for mobile and ubiquitous computing*
  - *support for multimedia streaming*
- Traditional O
  - simplify, protect
- Network OS's
  - do the same but
    enable remote p

What is a distributed OS?

- Presents users (and applications) with an integrated computing platform that hides the individual computers.
- Has control over all of the nodes (computers) in the network and allocates their resources to tasks without user involvement.
  - In a distributed OS, the user doesn't know (or care) where his programs are running.
- Examples:
  - Cluster computer systems
  - V system, Sprite, Globe OS
  - WebOS, Amoeba

*

The support required by middleware and distributed applications

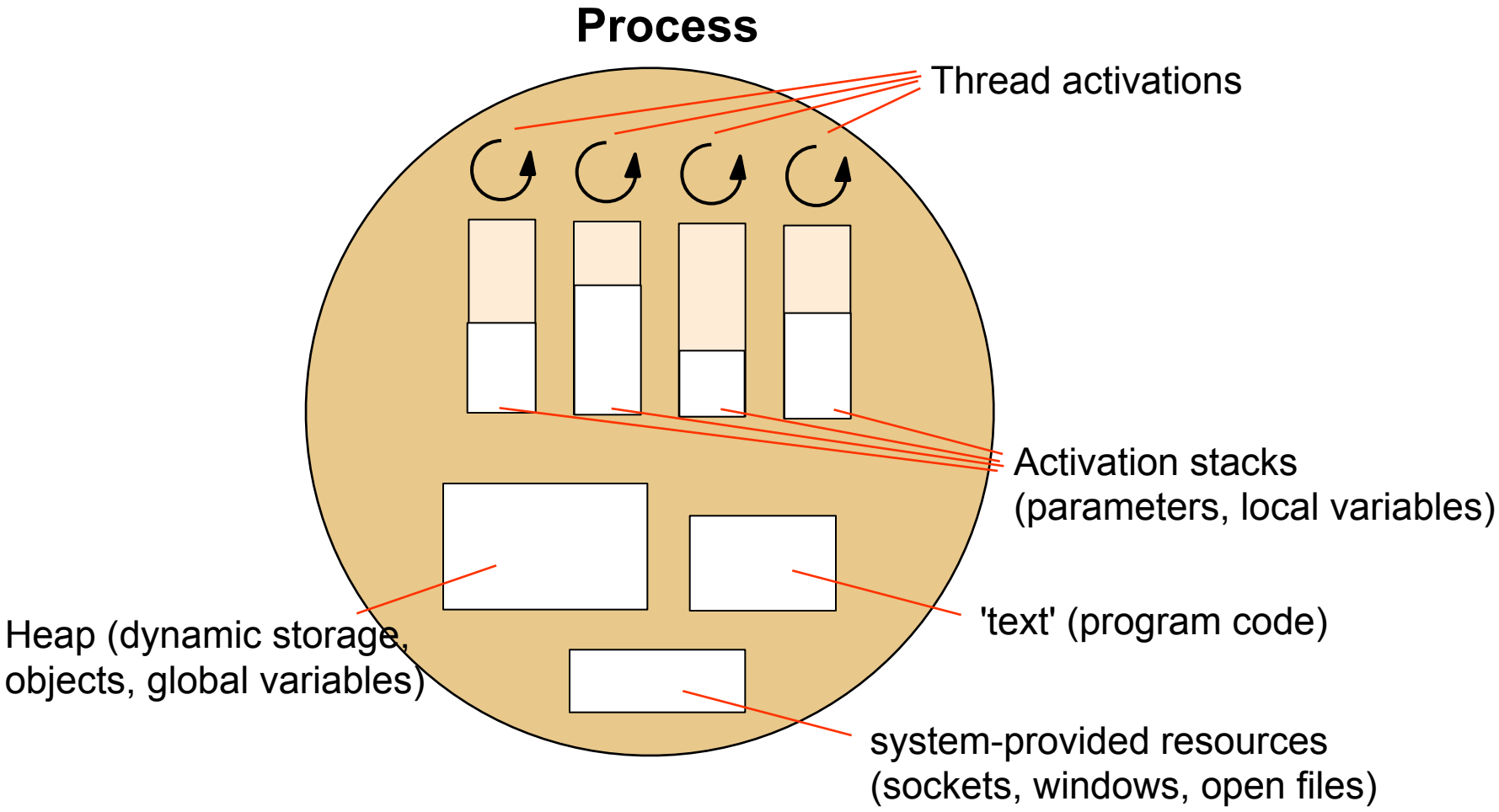
OS manages the basic resources of computer systems:

- processing, memory, persistent storage and communication.

It is the task of an operating system to:

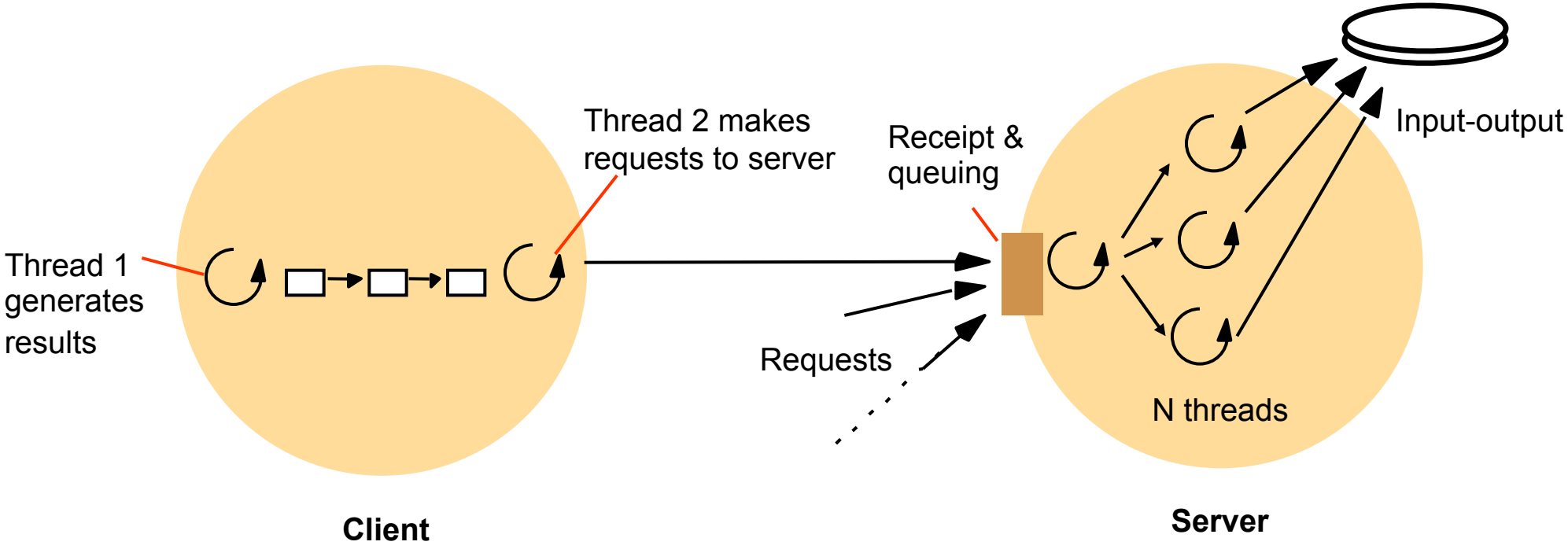
- raise the programming interface for these resources to a more useful level:
  - *By providing abstractions of the basic resources such as: processes, unlimited virtual memory, files, communication channels*
  - *Protection of the resources used by applications*
  - *Concurrent processing to enable applications to complete their work with minimum interference from other applications*
- provide the resources needed for (distributed) services and applications to complete their task:
  - *Communication - network access provided*
  - *Processing - processors scheduled at the relevant computers*

Threads concept and implementation



Client and server with threads

Figure 6.5



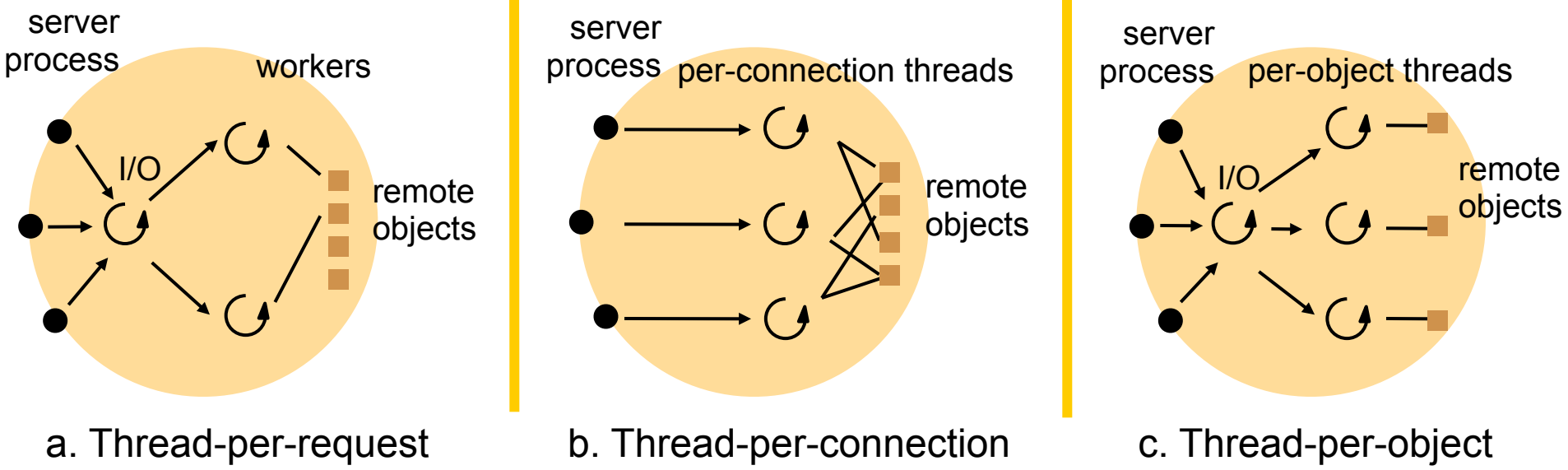
The 'worker pool' architecture

See Figure 4.6 for an example of this architecture programmed in Java.

*

Alternative server threading architectures

Figure 6.6



a. Thread-per-request b. Thread-per-connection c. Thread-per-object

– Implemented by the server-side ORB in CORBA

(a) would be useful for UDP-based service, e.g. NTP

(b) is the most commonly used - matches the TCP connection model

(c) is used where the service is encapsulated as an object. E.g. could have multiple shared whiteboards with one thread each. Each object has only one thread, avoiding the need for thread synchronization within objects.

*

Threads versus multiple processes

- Creating a thread is (much) cheaper than a process (~10-20 times)
- Switching to a different thread in same process is (much) cheaper (5-50 times)
- Threads within same process can share data and other resources more conveniently and efficiently (without copying or messages)
- Threads within a process are not protected from each other

Figure 6.7 State associated with execution environments and threads

Execution environment	*Thread*
Address space tables	Saved processor registers
Communication interfaces, open files	Priority and execution state (such as *BLOCKED*)
Semaphores, other synchronization objects	Software interrupt handling information
List of thread identifiers	Execution environment identifier
Pages of address space resident in memory; hardware cache entries	

*

Java thread constructor and management methods

Figure 6.8 Methods of objects that inherit from class Thread

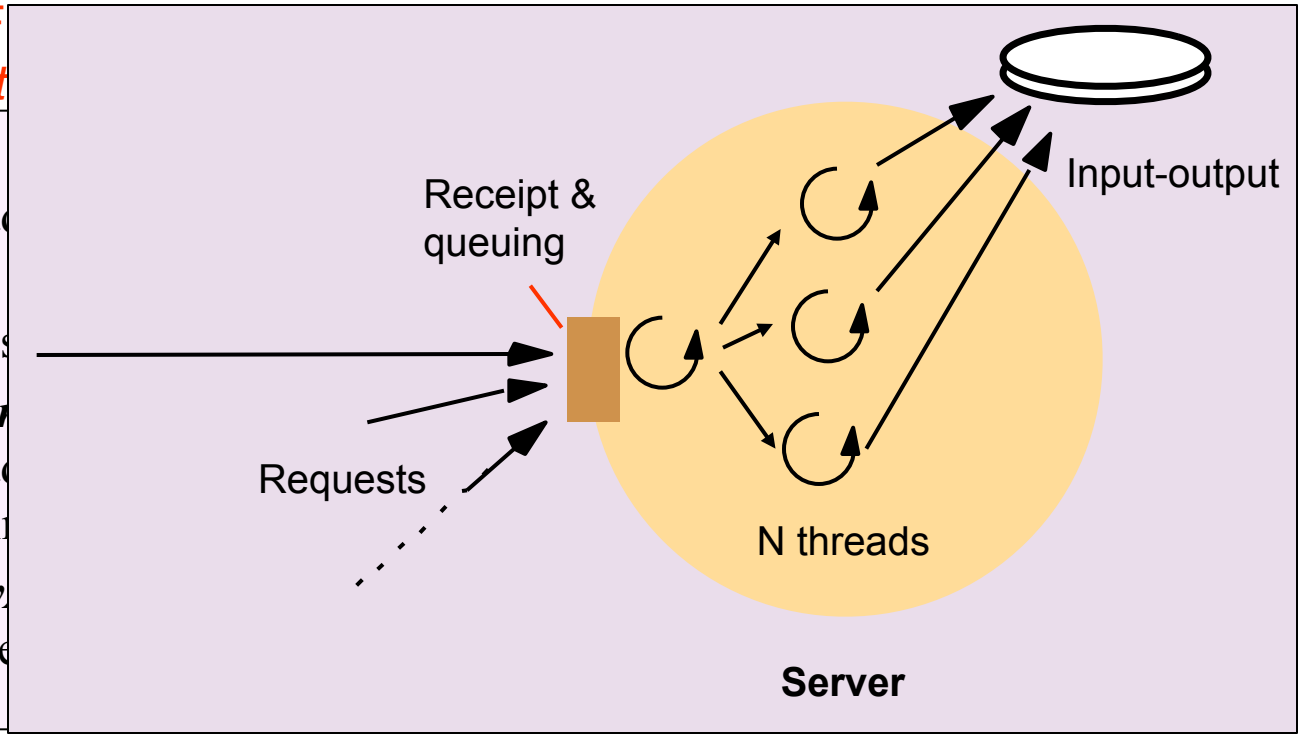
- ***Thread(ThreadGroup group, Runnable target, String name)***
  - Creates a new thread in the *SUSPENDED* state, which will belong to *group* and be identified as *name*; the thread will execute the *run()* method of *target*.
- ***setPriority(int newPriority), getPriority()***
  - Set and return the thread's priority.
- ***run()***
  - A thread executes the *run()* method of its target object, if it has one, and otherwise its own *run()* method (*Thread* implements *Runnable*).
- ***start()***
  - Change the state of the thread from *SUSPENDED* to *RUNNABLE*.
- ***sleep(int millisecs)***
  - Cause the thread to enter the *SUSPENDED* state for the specified time.
- ***yield()***
  - Enter the *READY* state and invoke the scheduler.
- ***destroy()***
  - Destroy the thread.

Java thread synchronization calls

Figure 6.9

See F
object

- ***thread.join(int millisecs)***
  - Blocks the calling threa
- ***thread.interrupt()***
  - Interrupts *thread*: cause
- ***object.wait(long millisecs, i***
  - Blocks the calling threa thread, or the thread is i
- ***object.notify(), object.notify***
  - Wakes, respectively, one



object.wait() and ***object.notify()*** are very similar to the semaphore operations. E.g. a worker thread in Figure 6.5 would use ***queue.wait()*** to wait for incoming requests.

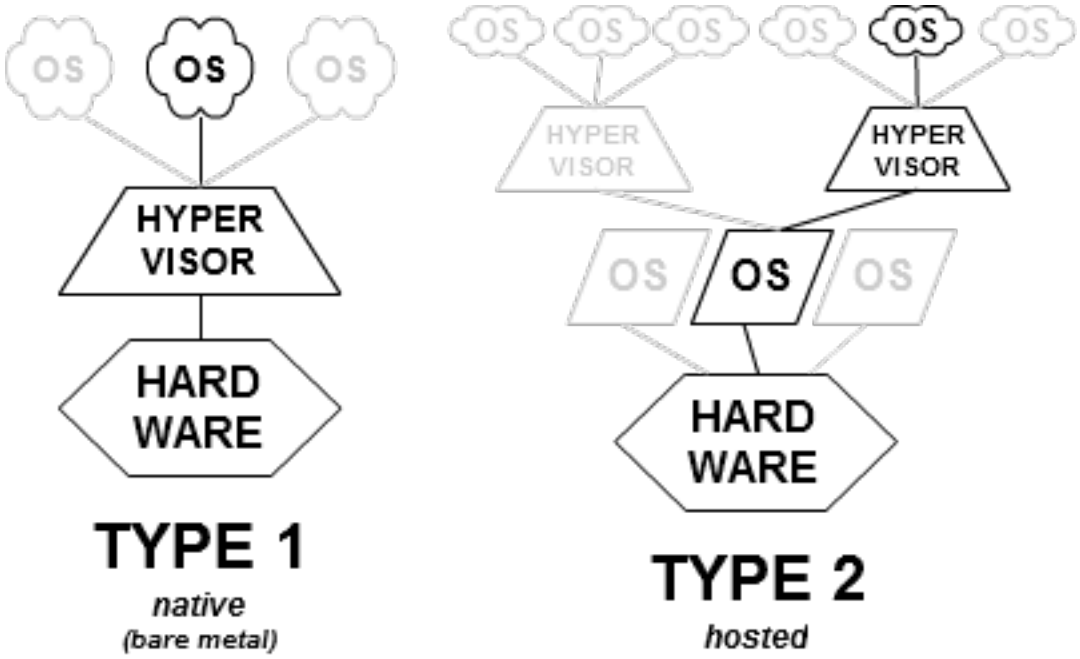
synchronized methods (and code blocks) implement the *monitor* abstraction. The operations within a synchronized method are performed atomically with respect to other synchronized methods of the same object. ***synchronized*** **should be used for any methods that update the state of an object** in a threaded environment.

*

System Virtualization

- Aim: provide multiple virtual machines (virtual hardware images)
- Multiple OS can be run at once
- Separation of resources as for processes
- Motivations
  - resource optimization, server migration
  - cloud computing
  - fast creation/deletion of virtual machines
  - multiple OS on same hardware
- Virtualization goes back to IBM VM 370, and fulfills original µkernel aims

System Virtualization



- *Type-1 hypervisor*: run directly on the host's hardware to control the hardware and to manage guest operating systems. Require specific modifications in guest OS. Examples: XEN, VMware ESX, Oracle VM
- *Type-2 hypervisor:* run on a conventional operating system just as other computer programs do. A guest operating system runs as a process on the host. Examples: VMware workstation, VirtualBox, QEMU

System Virtualization

- *System virtual machines* provide a substitute for a real machine, in order to execute entire operating systems
- A *type-2 hypervisor* uses native execution to share and manage hardware
- Each guest environments is isolated from one another
- Modern hypervisors use hardware-assisted virtualization from the host CPUs.
- Examples: KVM, QEMU, VMware

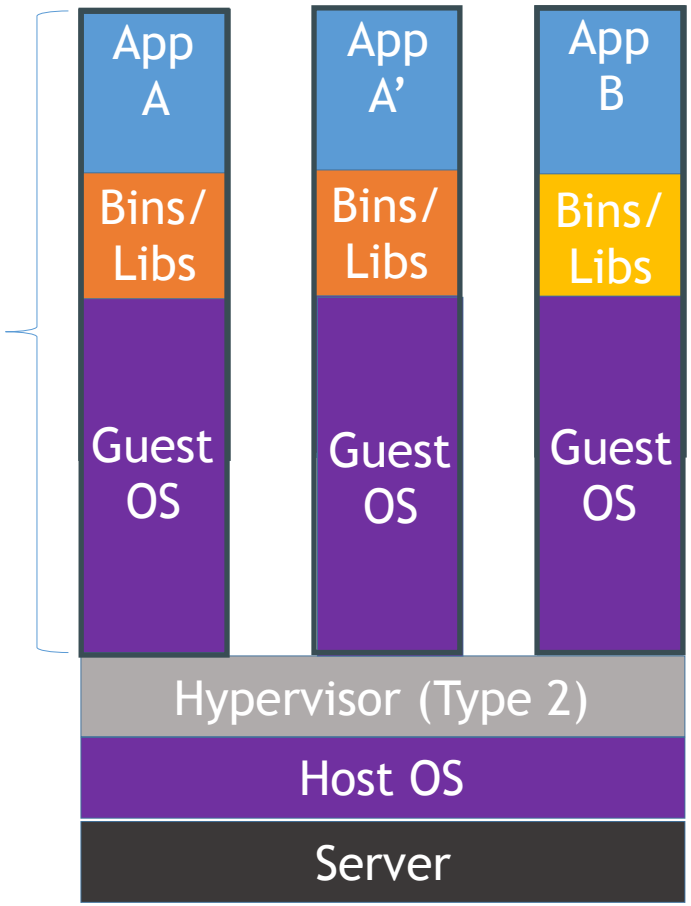


Figure 7.17
The architecture of Xen

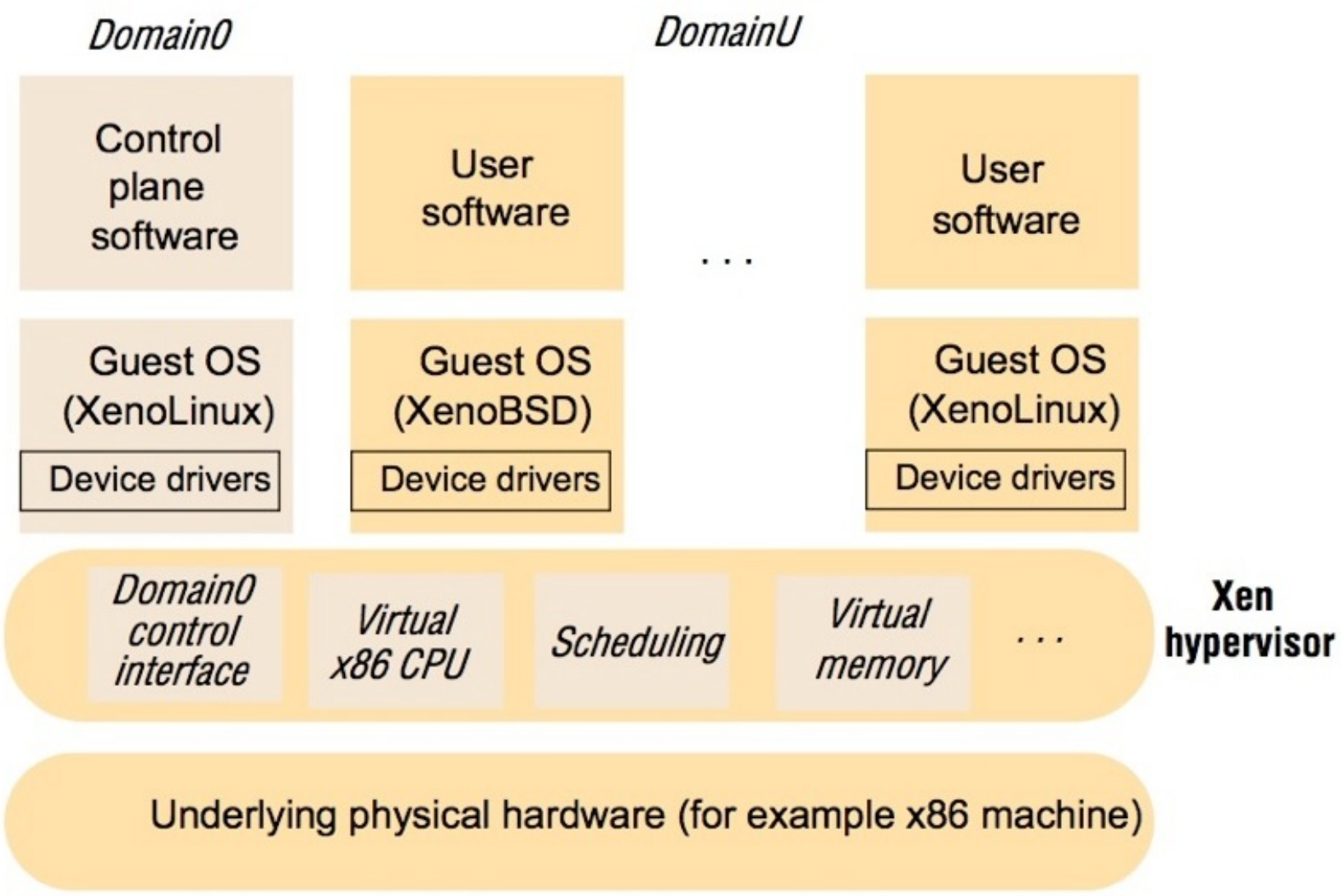


Figure 7.19
Virtualization of memory management

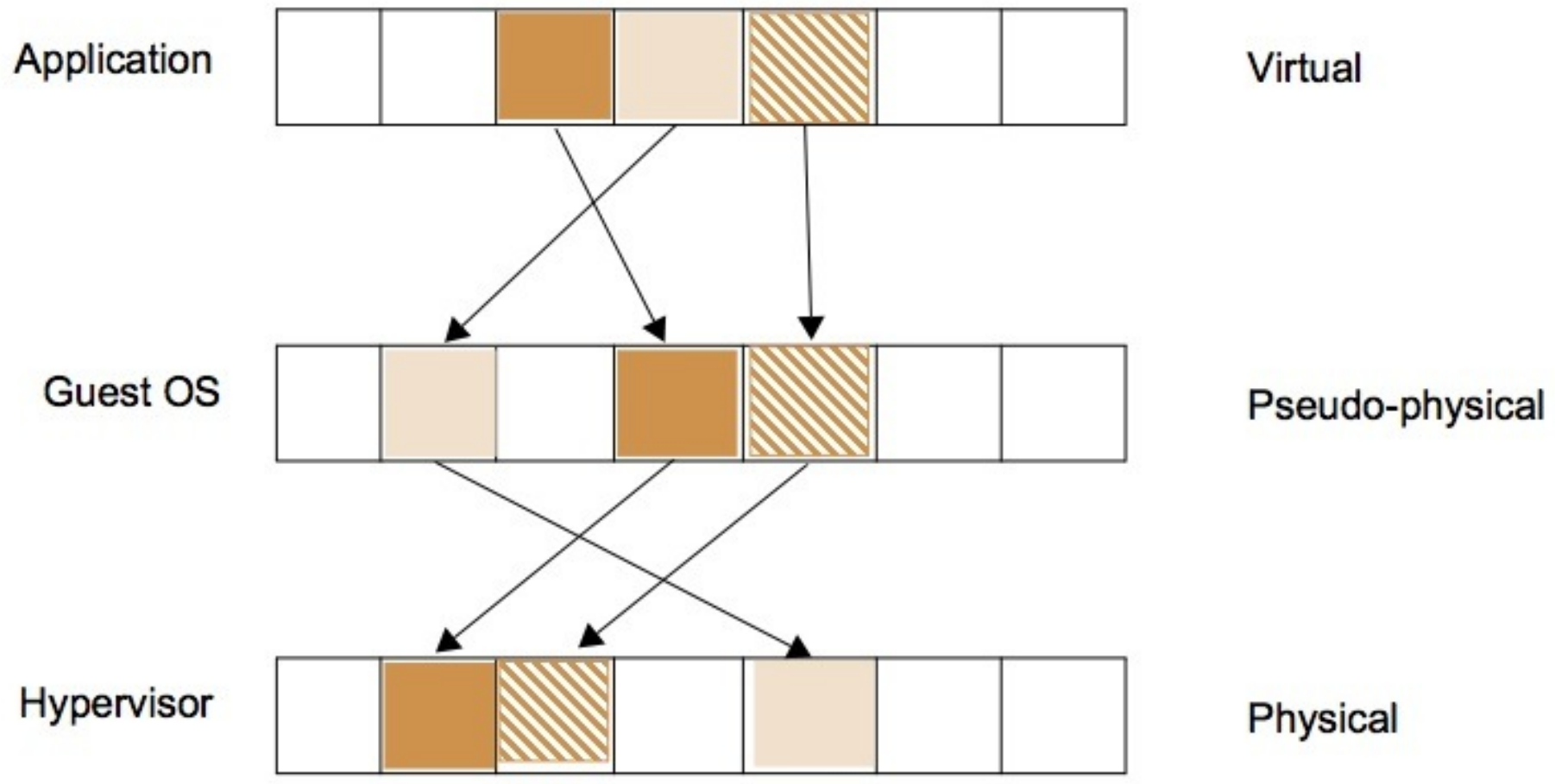
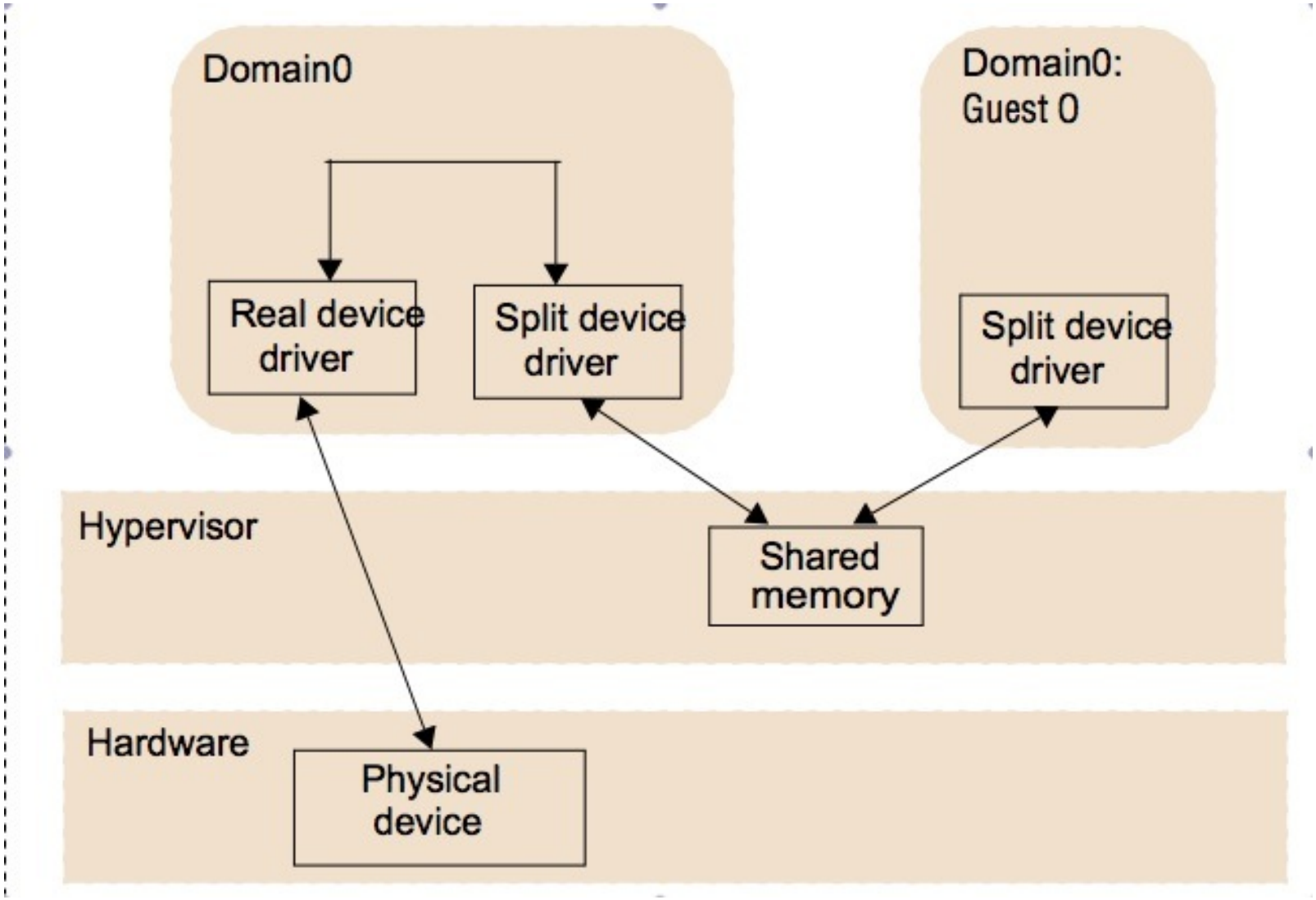


Figure 7.20
Split device drivers



Docker

- Build once, run anywhere*
  - A clean, safe, hygienic and portable runtime environment for your app.
  - No worries about missing dependencies, packages and other pain points during subsequent deployments.
  - Run each app in its own isolated container, so you can run various versions of libraries and other dependencies for each app without worrying
  - Automate testing, integration, packaging, anything you can script
  - Reduce/eliminate concerns about compatibility on different platforms, either your own or your customers.
  - Cheap, zero-penalty containers to deploy services: a VM without the overhead of a VM, with instant replay and reset of image snapshots

* on any x86 server running a modern Linux kernel (3.2+ generally. 2.6.32+ for RHEL 6.5+, Fedora, & related)

DOCKER - The Challenge

Multiplicity of Stacks

Static website
nginx 1.5 + modsecurity + openssl + bootstrap 2

User DB
postgresql + pgv8 + v8

Queue
Redis + redis-sentinel

Analytics DB
hadoop + hive + thrift + OpenJDK

Background workers
Python 3.0 + celery + pyredis + libcurl + ffmpeg + libopencv + nodejs + phantomjs

Web frontend
Ruby + Rails + sass + Unicorn

API endpoint
Python 2.7 + Flask + pyredis + celery + psycopg + postgresql-client

Do services and apps interact appropriately?

Multiplicity of hardware environments

Development VM

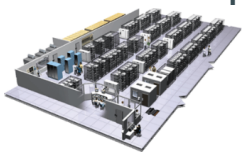
QA server

Customer Data Center

Public Cloud

Disaster recovery

Production Servers

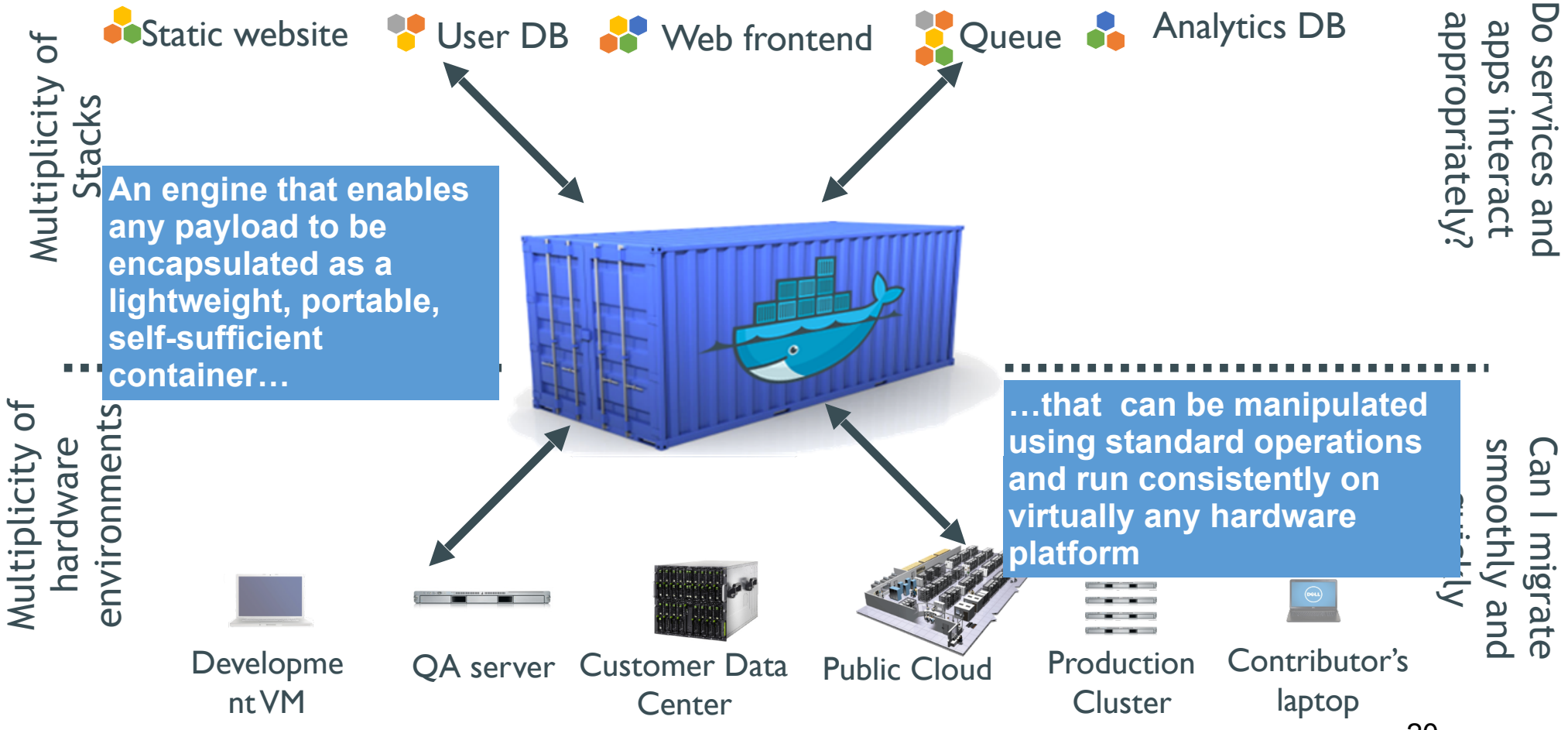


Production Cluster

Contributor's laptop

Can I migrate smoothly and quickly?

Docker is a shipping container system for code

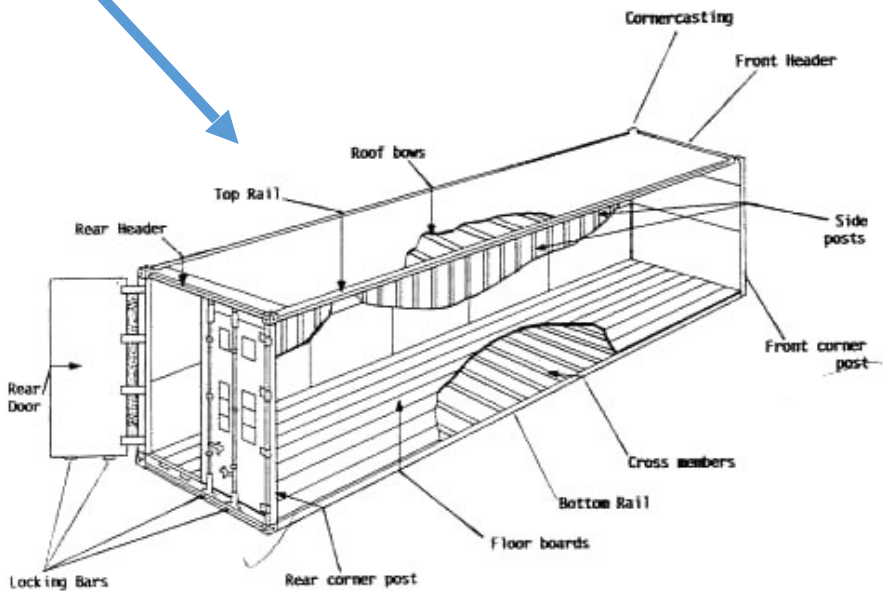


Why Devops Cares?

- Configure once…run anything
  - Make the entire lifecycle more efficient, consistent, and repeatable
  - Increase the quality of code produced by developers.
  - Eliminate inconsistencies between development, test, production, and customer environments
  - Support segregation of duties
  - Significantly improves the speed and reliability of continuous deployment and continuous integration systems
  - Because the containers are so lightweight, address significant performance, costs, deployment, and portability issues normally associated with VMs

Why it works—separation of concerns

- Developer
  - Worries about what's "inside" the container
    - *His code*
    - *His Libraries*
    - *His Package Manager*
    - *His Apps*
    - *His Data*
  - All Linux servers look the same
- Ops Guy
  - Worries about what's "outside" the container
    - Logging
    - Remote access
    - Monitoring
    - Network config
  - All containers start, stop, copy, attach, migrate, etc. the same way



Major components of the container:

More technical explanation

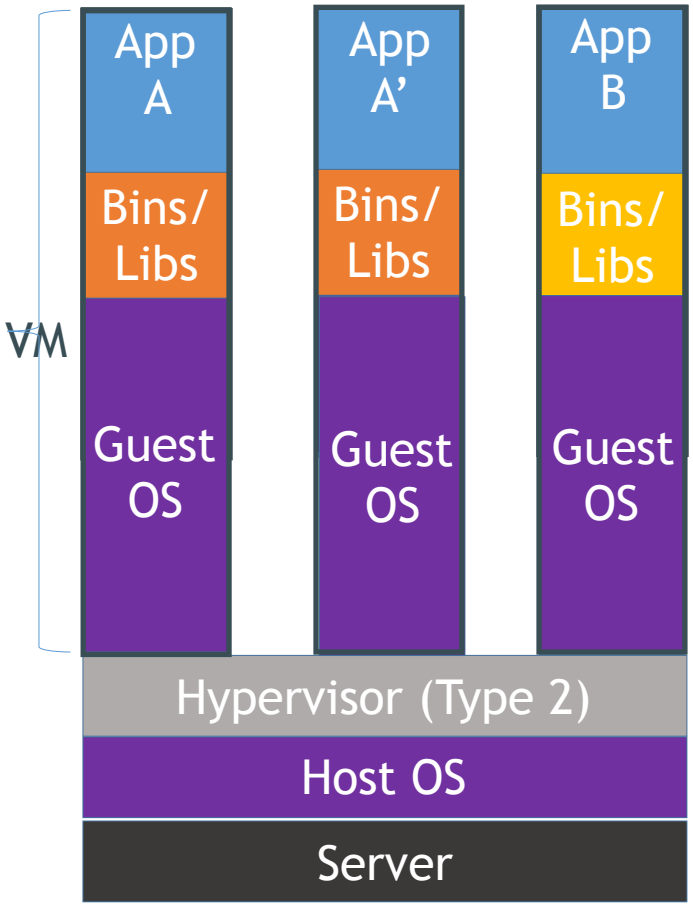
WHY

- Run everywhere
  - Regardless of kernel version (2.6.32+)
  - Regardless of host distro
  - Physical or virtual, cloud or not
  - Container and host architecture must match*
- Run anything
  - If it can run on the host, it can run in the container
  - i.e. if it can run on a Linux kernel, it can run

WHAT

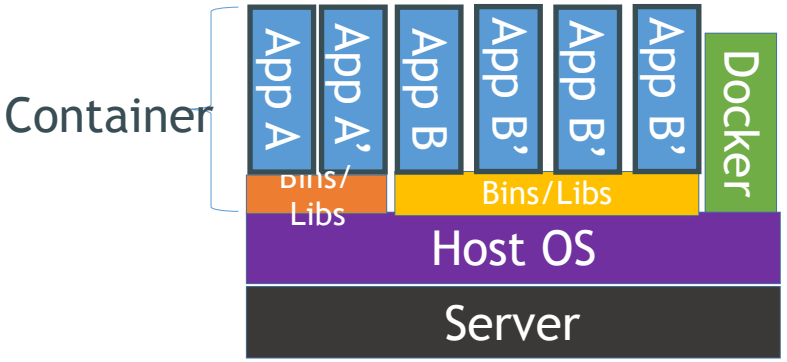
- High Level—It's a lightweight VM
  - Own process space
  - Own network interface
  - Can run stuff as root
  - Can have its own /sbin/init (different from host)
  - <<machine container>>
- Low Level—It's *chroot* on steroids
  - Can also *not* have its own /sbin/init
  - Container=isolated processes
  - Share kernel with host
  - No device emulation (neither HVM nor PV) from host)
  - <<application container>>

VMs vs. Containers



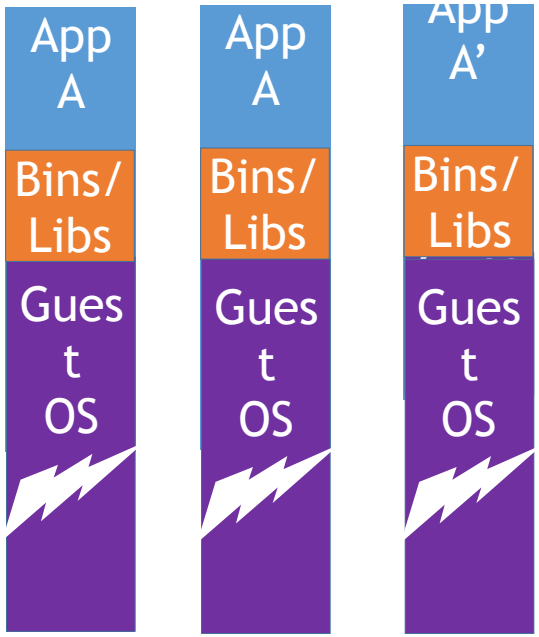
Containers are isolated, but share OS and, where appropriate, bins/ libraries

…result is significantly faster deployment, much less overhead, easier migration, faster restart



Why are Docker containers lightweight?

VMs



VMs

Every app, every copy of an app, and every slight modification of the app requires a new virtual server

Containers

Original App

(No OS to take up space, resources, or require restart)

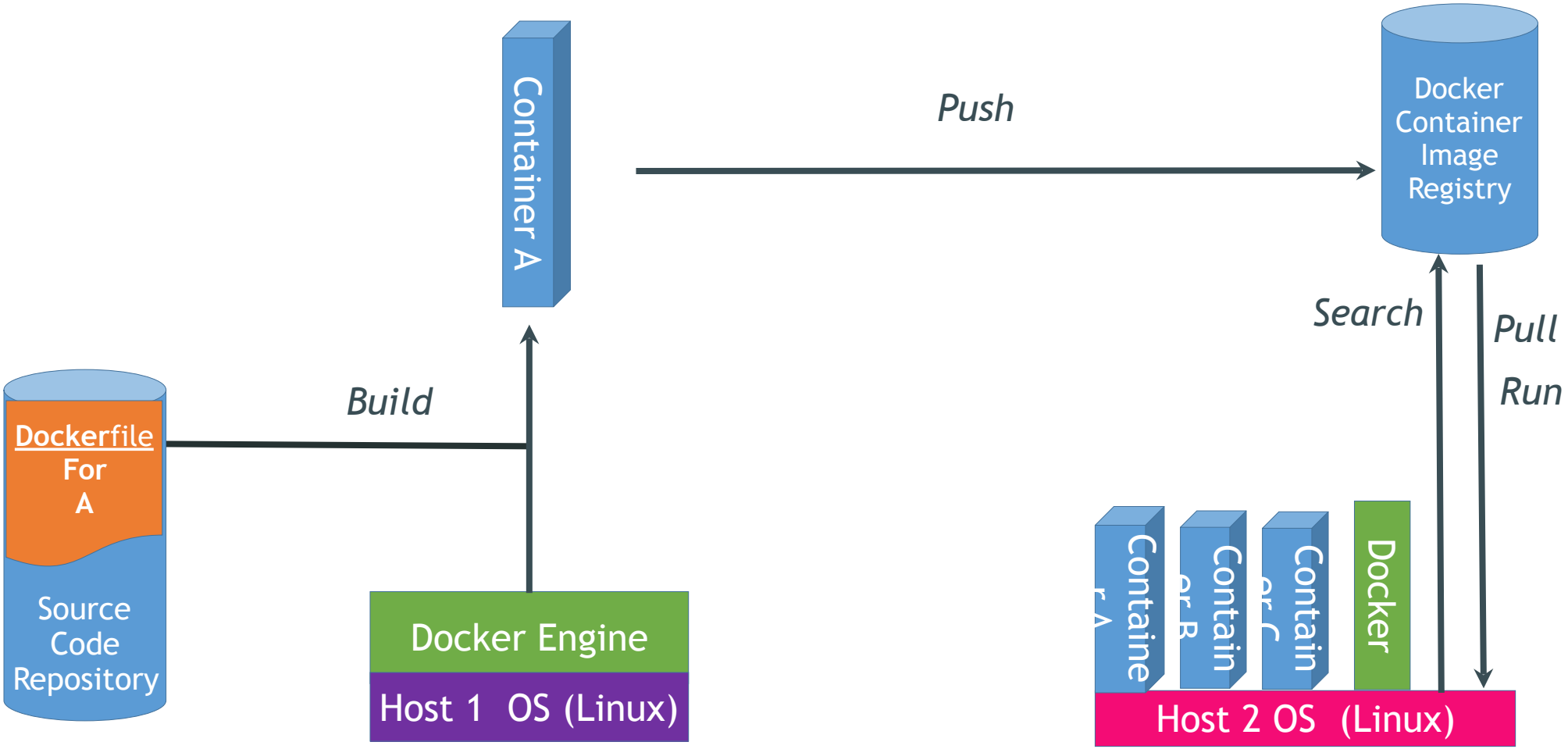
Copy of App

No OS. Can Share bins/libs

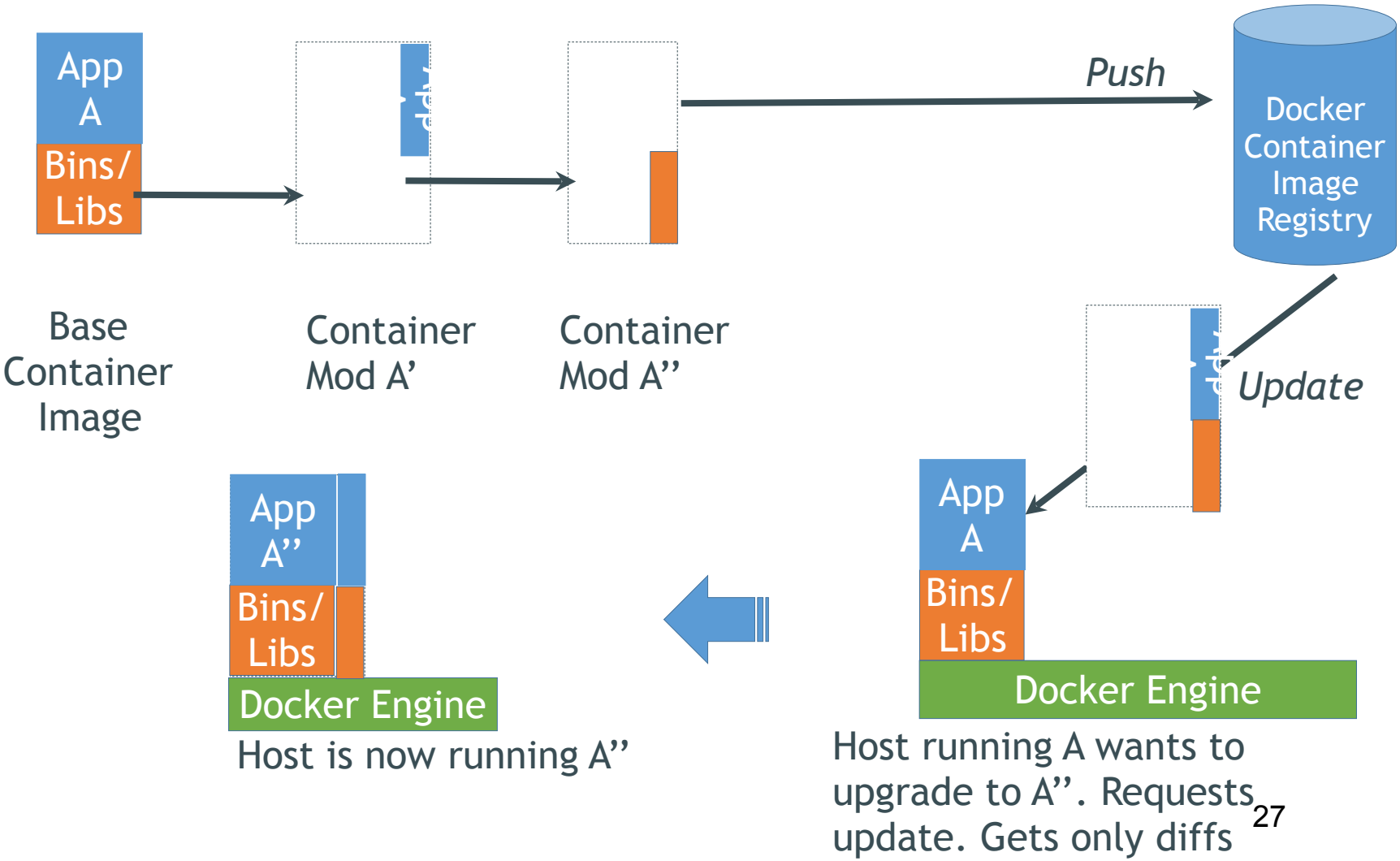
Modified App

Copy on write capabilities allow us to only save the diffs between container A and container A'

What are the basics of the Docker system?



Changes and Updates



Summary

- The OS provides *local* support for the implementation of distributed applications and middleware:
  - Manages and protects system resources (memory, processing, communication)
  - Provides relevant local abstractions
    - *files, processes, threads, communication ports*
- Middleware provides general-purpose *distributed* abstractions
  - RPC, DSM, event notification, streaming
- Invocation performance is important
  - it can be optimized, E.g. Firefly RPC, LRPC
- Microkernel architecture for flexibility
  - The KISS principle ('Keep it simple – stupid!')
    - *has resulted in migration of many functions out of the OS*
- Virtual Machines
- Containers

Relevant topics not covered

- Protection of OS resources (Section 6.3)
  - Control of access to distributed resources is covered in Chapter 7 - Security
- Mach OS case study (Chapter 18)
  - Halfway to a distributed OS
  - The communication model is of particular interest. It includes:
    - *ports that can migrate betwen computers and processes*
    - *support for RPC*
    - *typed messages*
    - *access control to ports*
    - *open implementation of network communication (drivers outside the kernel)*
- Thread programming (Section 6.4, and many other sources)
  - e.g. Doug Lea's book *Concurrent Programming in Java*, (Addison Wesley 1996)