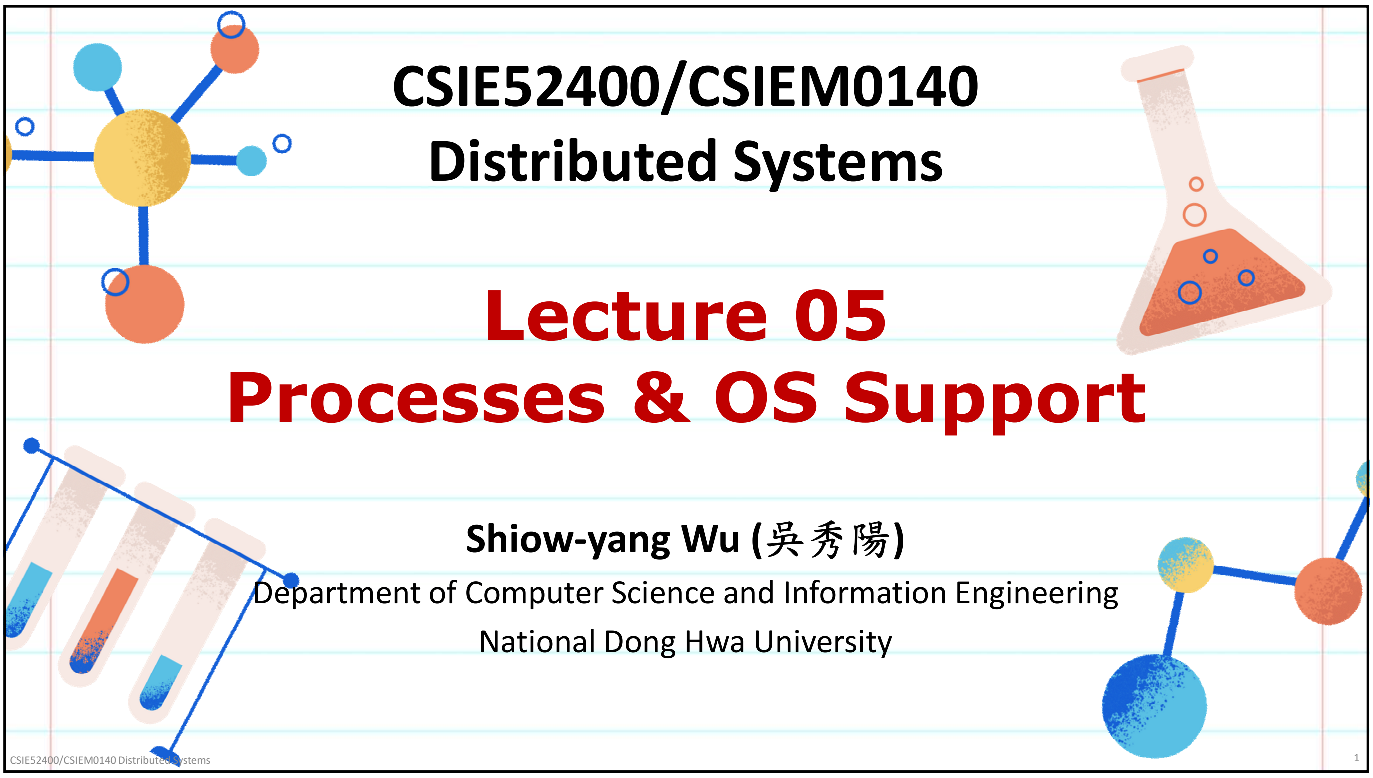


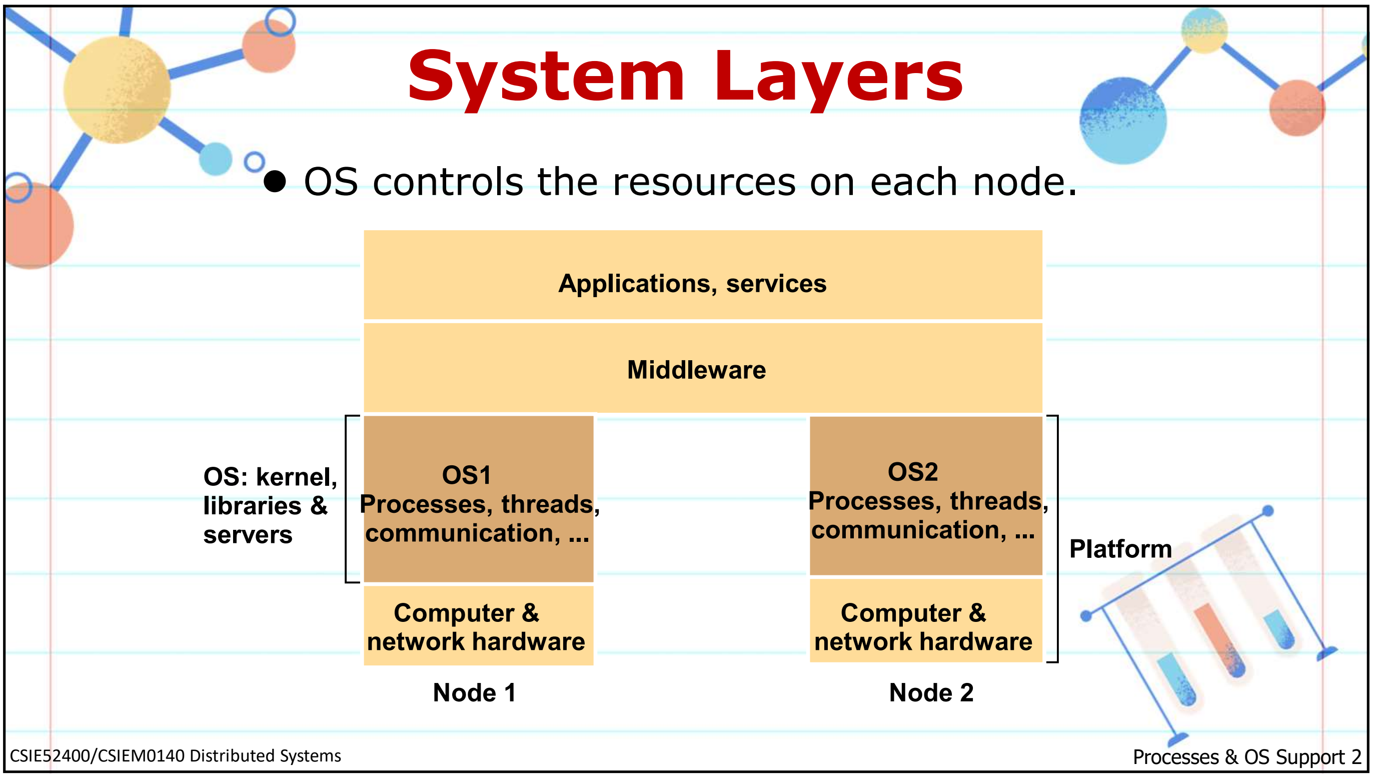
CSIE52400/CSIEM0140 Distributed Systems

Lecture 05 Processes & OS Support

Shiow-yang Wu (吳秀陽)

Department of Computer Science and Information Engineering

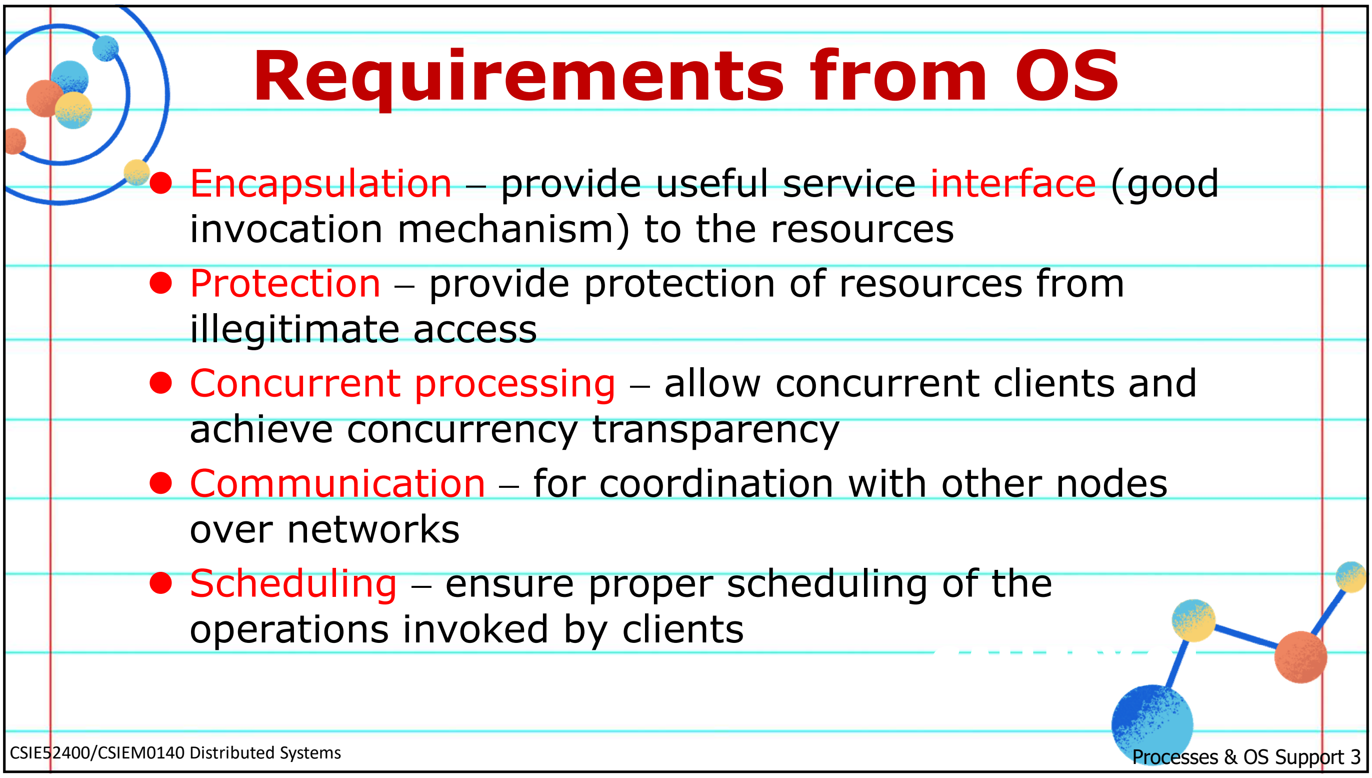
National Dong Hwa University



System Layers

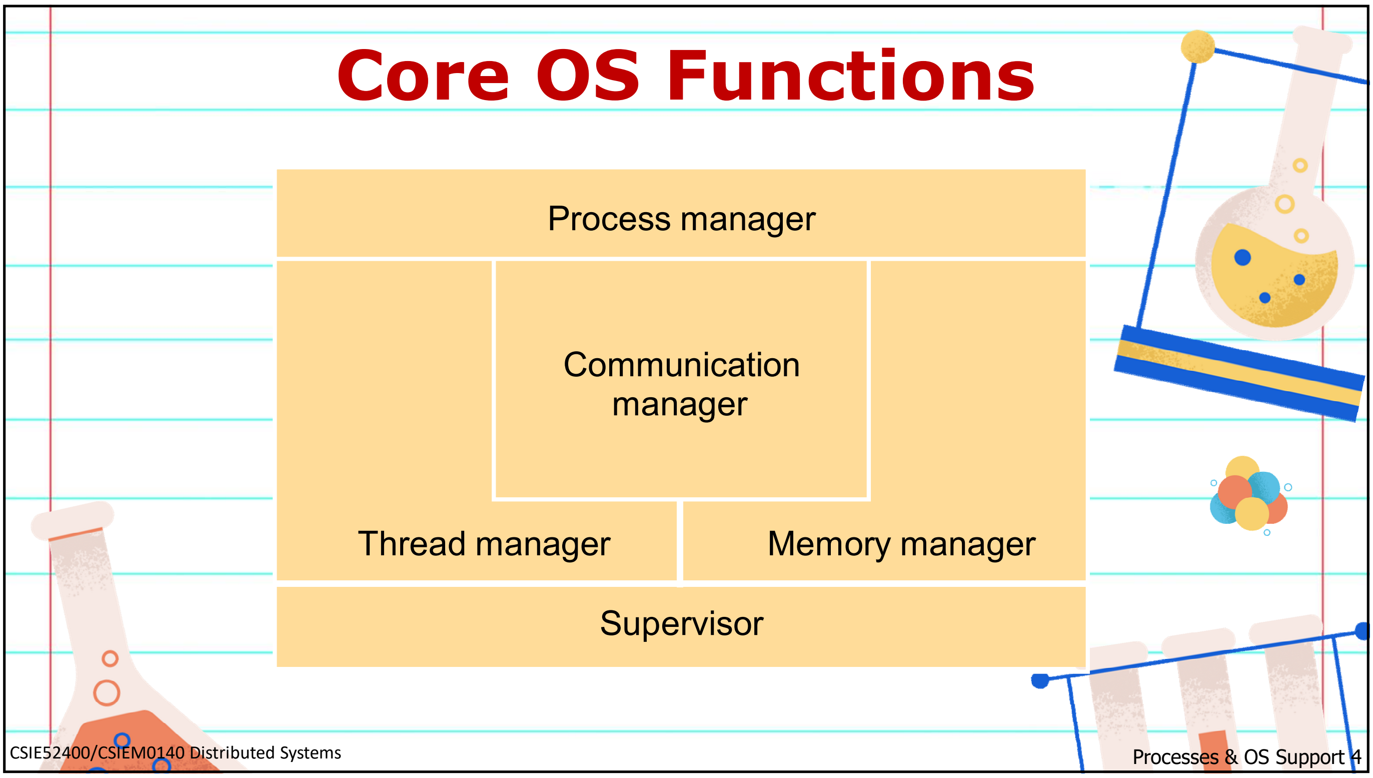
- OS controls the resources on each node.

Applications, services
Middleware
OS: kernel, libraries & servers
OS1
Processes, threads, communication, ...
Computer & network hardware
Node 1
OS2
Processes, threads, communication, ...
Computer & network hardware
Node 2
Platform

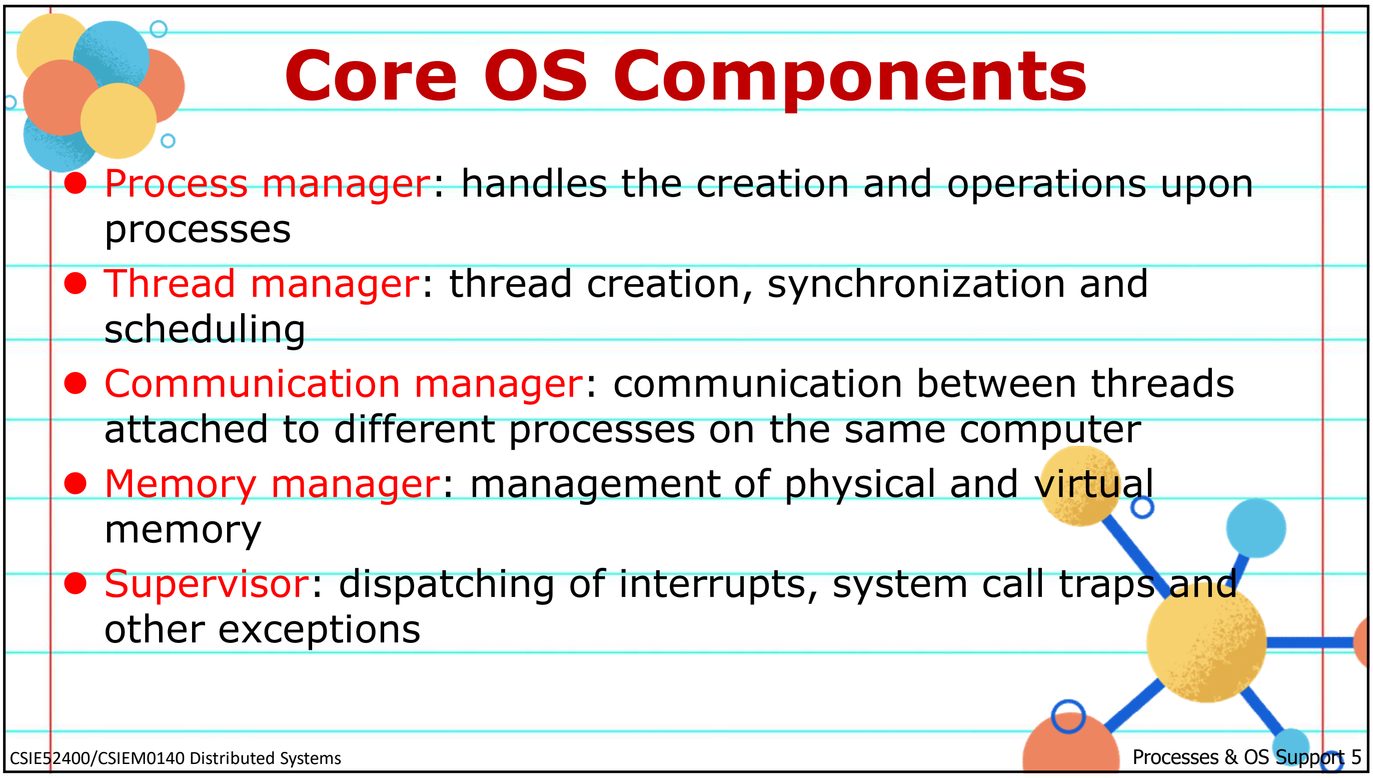


Requirements from OS

- Encapsulation – provide useful service interface (good invocation mechanism) to the resources
- Protection – provide protection of resources from illegitimate access
- Concurrent processing – allow concurrent clients and achieve concurrency transparency
- Communication – for coordination with other nodes over networks
- Scheduling – ensure proper scheduling of the operations invoked by clients

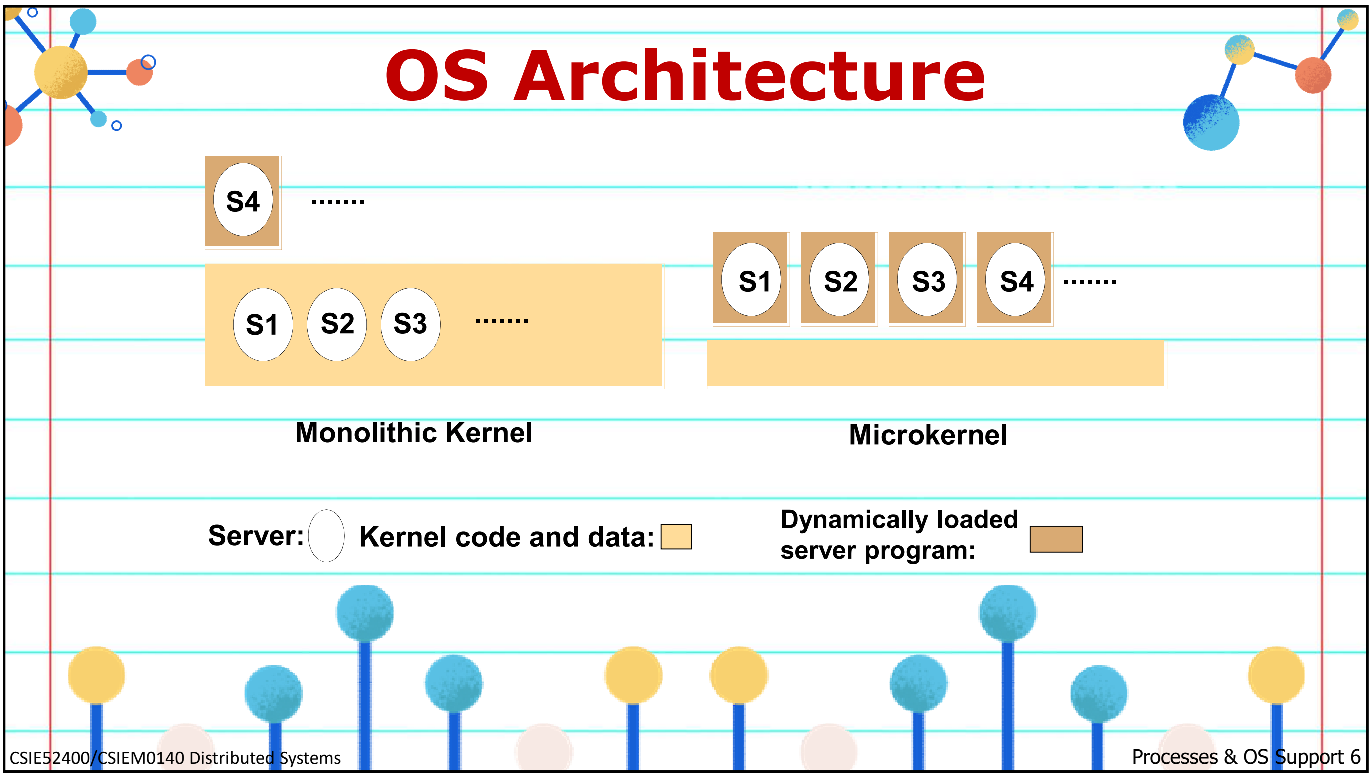


Core OS Functions



Core OS Components

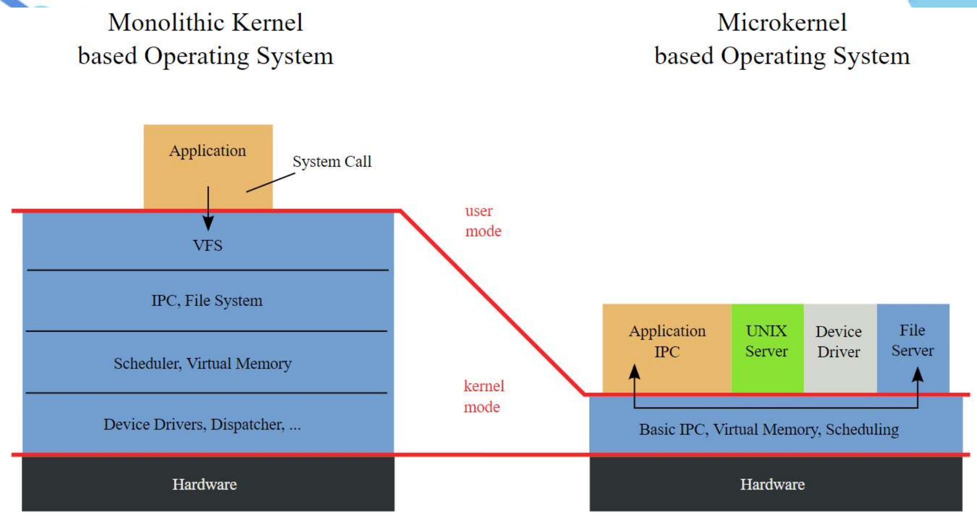
- Process manager: handles the creation and operations upon processes
- Thread manager: thread creation, synchronization and scheduling
- Communication manager: communication between threads attached to different processes on the same computer
- Memory manager: management of physical and virtual memory
- Supervisor: dispatching of interrupts, system call traps and other exceptions



OS Architecture

Server: ○ Kernel code and data: □ Dynamically loaded server program: ■

Monolithic vs Microkernel



Monolithic Kernel
based Operating System

Application

System Call

VFS

IPC, File System

Scheduler, Virtual Memory

Device Drivers, Dispatcher, ...

Hardware

user
mode

kernel
mode

Microkernel
based Operating System

Application
IPC

UNIX
Server

Device
Driver

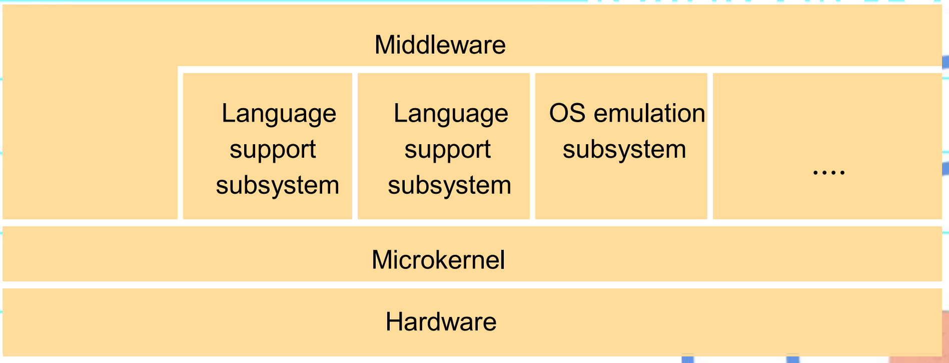
File
Server

Basic IPC, Virtual Memory, Scheduling

Hardware

(https://en.wikipedia.org/wiki/Microkernel)

The Role of Microkernel



Middleware

Language
support
subsystem

Language
support
subsystem

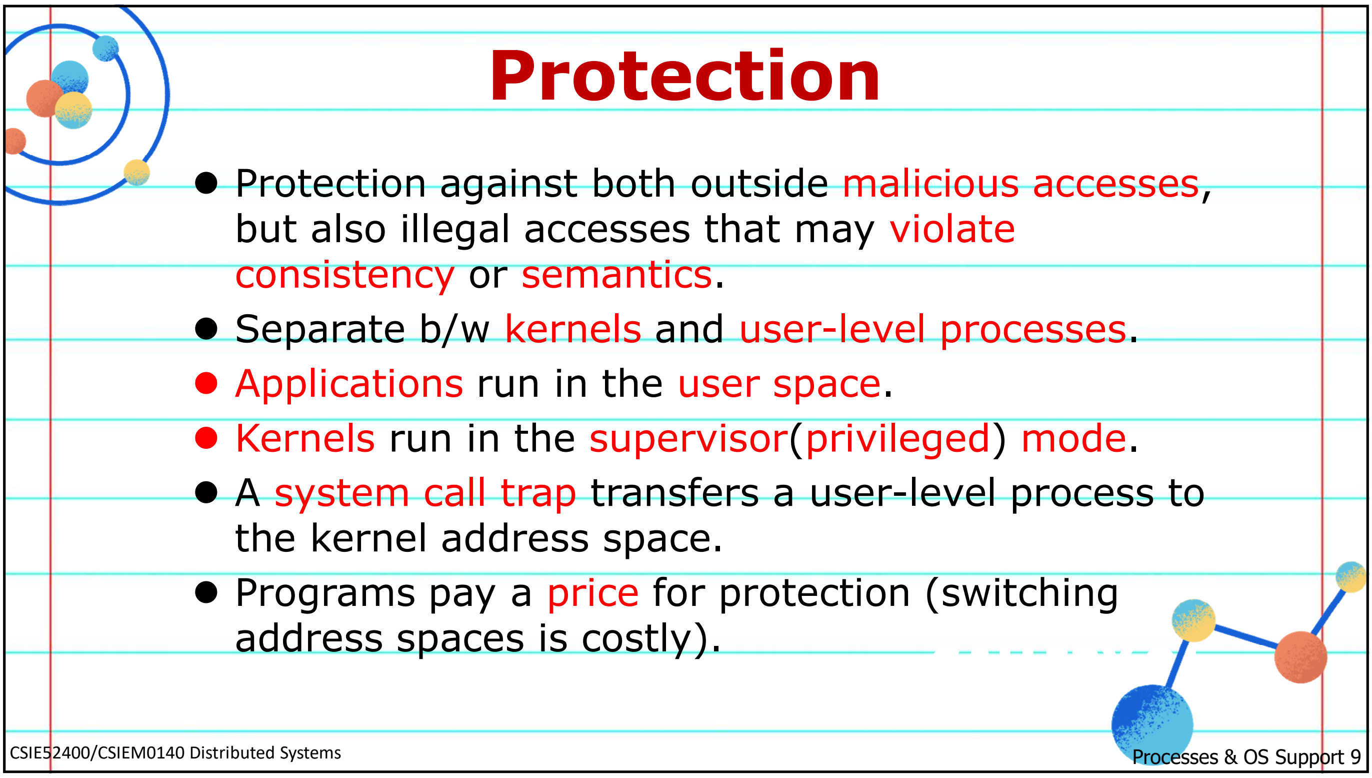
OS emulation
subsystem

....

Microkernel

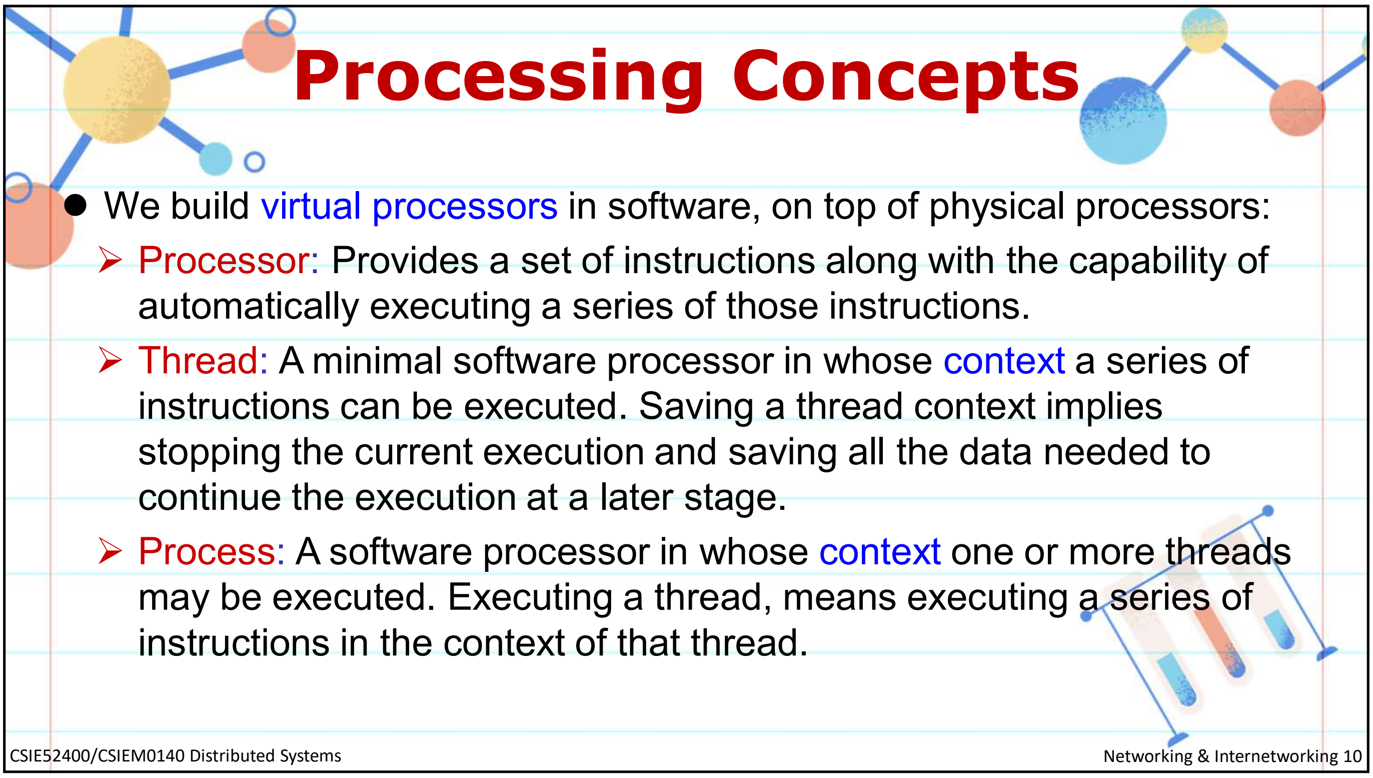
Hardware

The microkernel supports middleware via subsystems



Protection

- Protection against both outside malicious accesses, but also illegal accesses that may violate consistency or semantics.
- Separate b/w kernels and user-level processes.
- Applications run in the user space.
- Kernels run in the supervisor(privileged) mode.
- A system call trap transfers a user-level process to the kernel address space.
- Programs pay a price for protection (switching address spaces is costly).

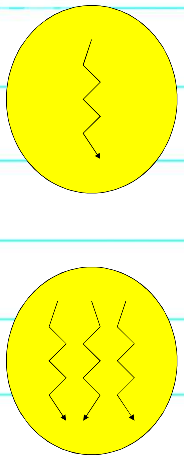


Processing Concepts

- We build virtual processors in software, on top of physical processors:
  - Processor: Provides a set of instructions along with the capability of automatically executing a series of those instructions.
  - Thread: A minimal software processor in whose context a series of instructions can be executed. Saving a thread context implies stopping the current execution and saving all the data needed to continue the execution at a later stage.
  - Process: A software processor in whose context one or more threads may be executed. Executing a thread, means executing a series of instructions in the context of that thread.

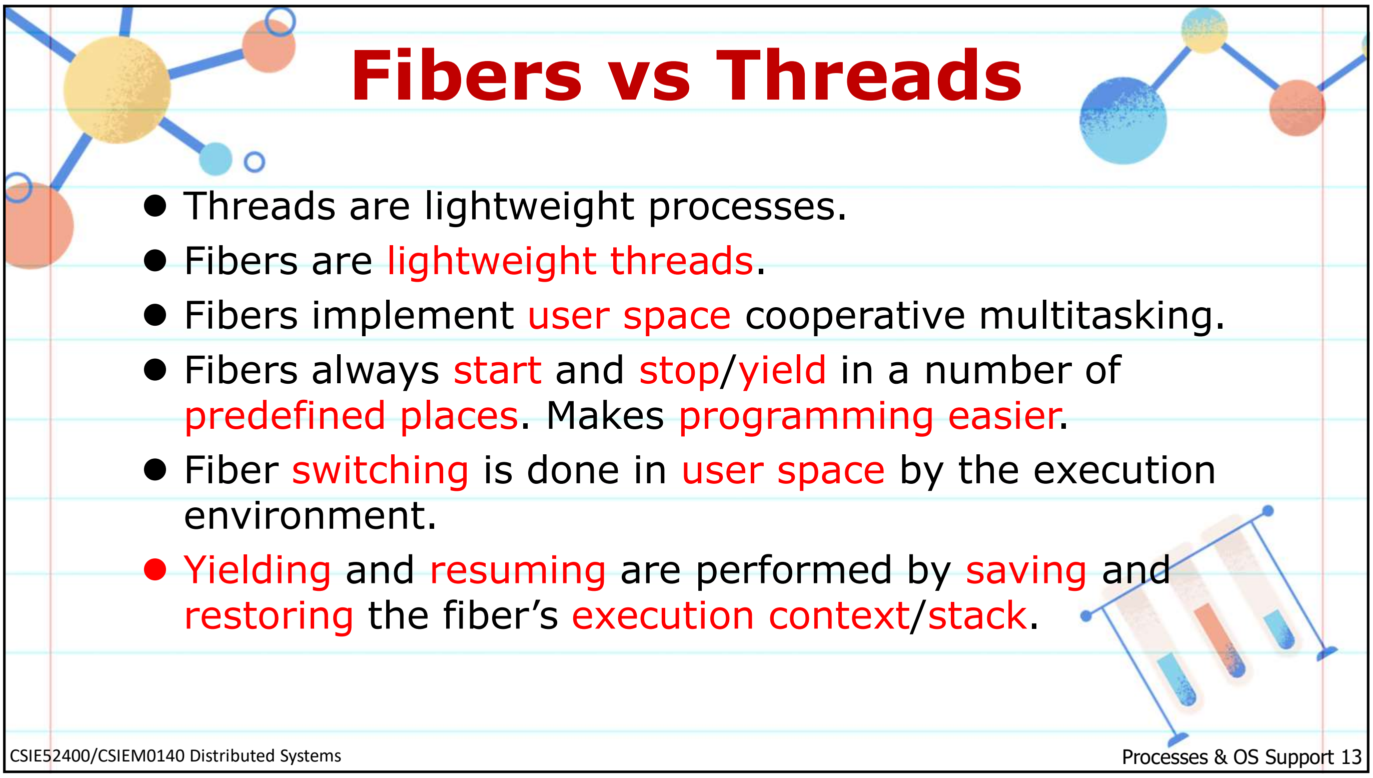
Processes and Threads

- Traditional processes are heavyweight processes
  - allow only a single activity
  - carry all the resources within it
  - make sharing awkward and expensive
- New process concept
  - an execution environment
  - with one or more threads
  - a thread is an OS abstraction of an activity



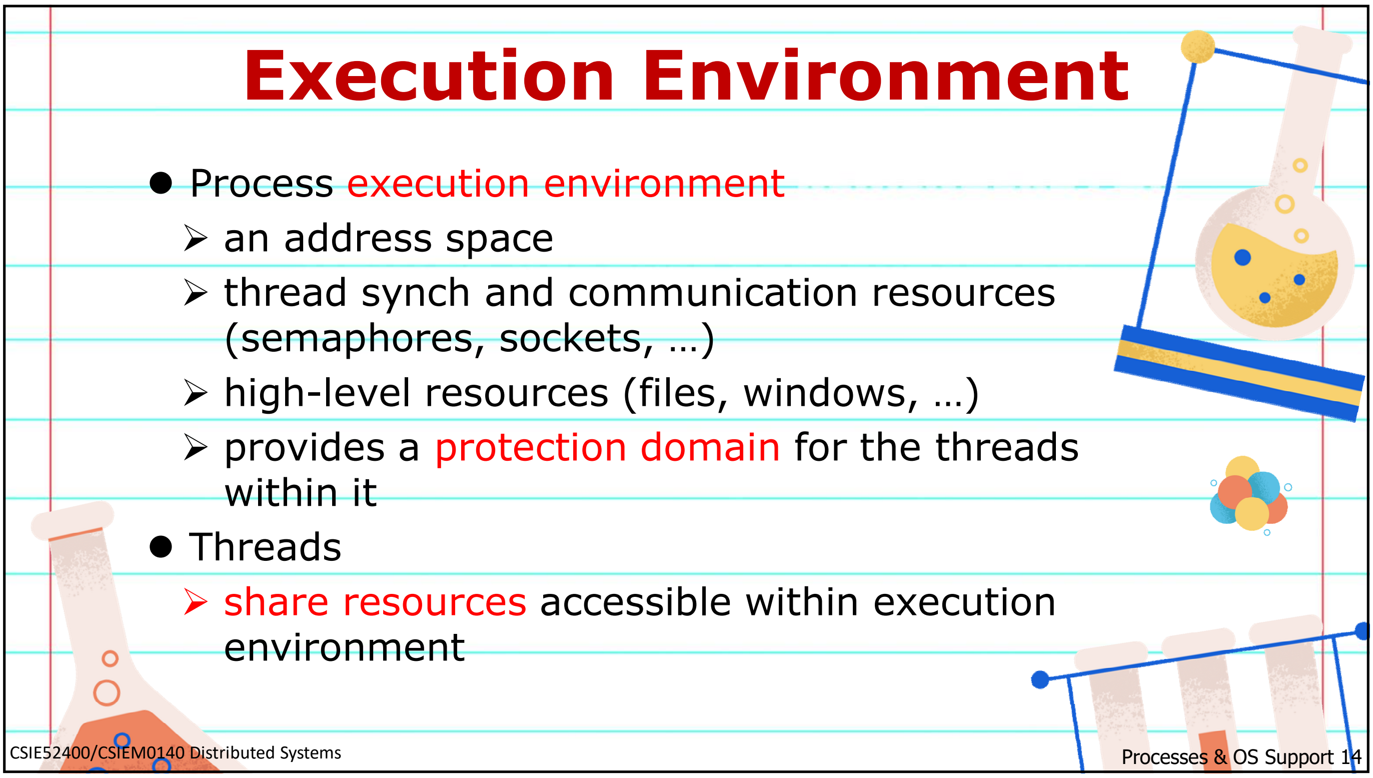
Fibers

- Fibers are even lighter units of execution which are cooperatively scheduled.
- They provide means for running pieces of code that can be paused and resumed.
- Only one fiber will be running at a time.
- A running fiber must explicitly "yield" to allow another fiber to run.
- A fiber can run in any thread in the same process.
- Applications gain performance by managing scheduling themselves.



Fibers vs Threads

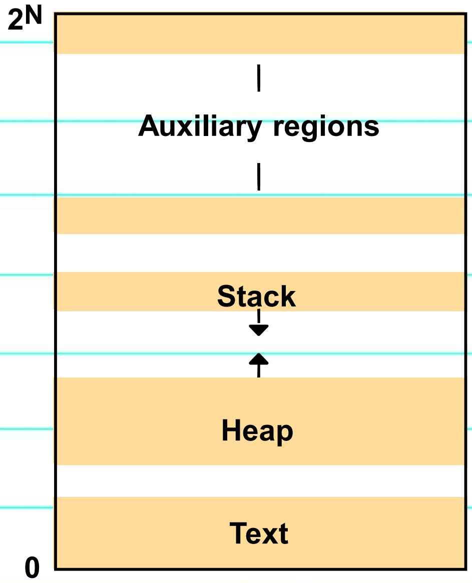
- Threads are lightweight processes.
- Fibers are lightweight threads.
- Fibers implement user space cooperative multitasking.
- Fibers always start and stop/yield in a number of predefined places. Makes programming easier.
- Fiber switching is done in user space by the execution environment.
- Yielding and resuming are performed by saving and restoring the fiber’s execution context/stack.



Execution Environment

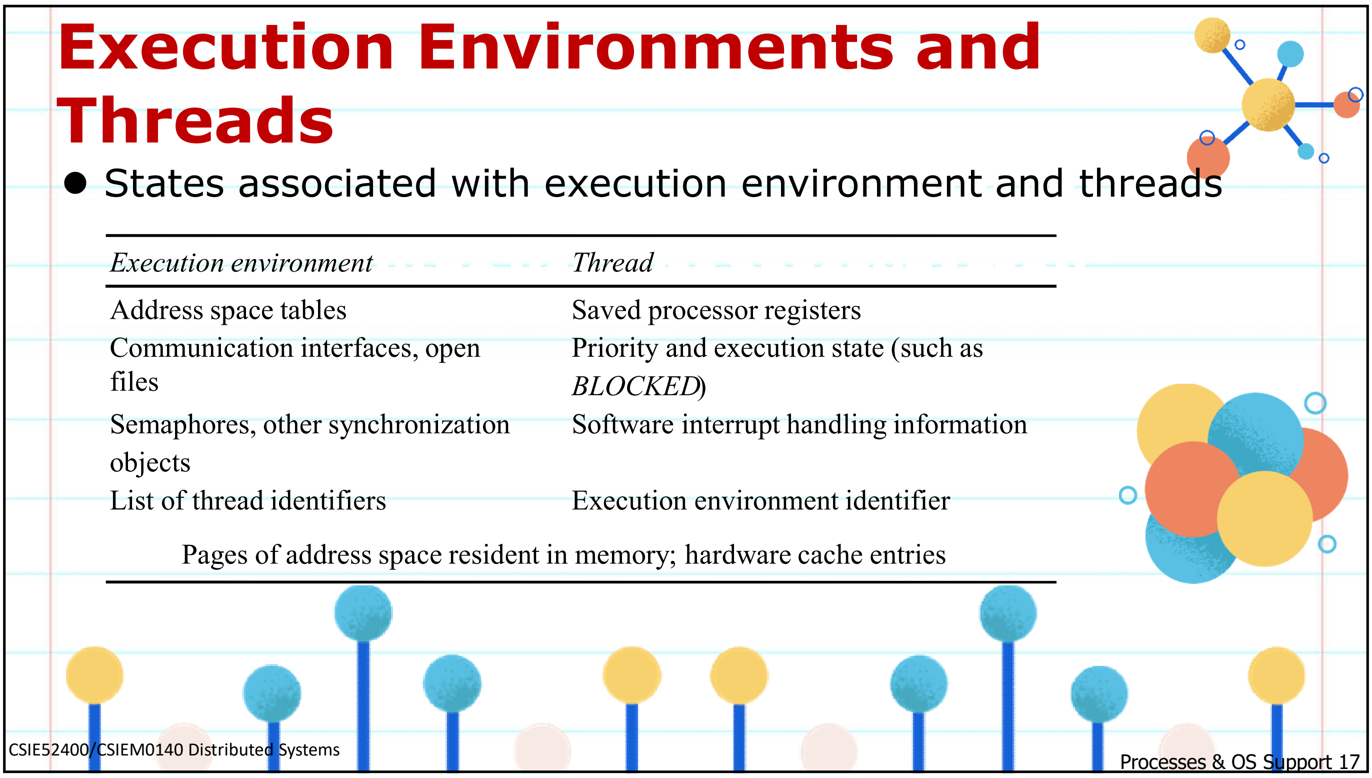
- Process execution environment
  - an address space
  - thread synch and communication resources (semaphores, sockets, …)
  - high-level resources (files, windows, …)
  - provides a protection domain for the threads within it
- Threads
  - share resources accessible within execution environment

Address Space



Regions

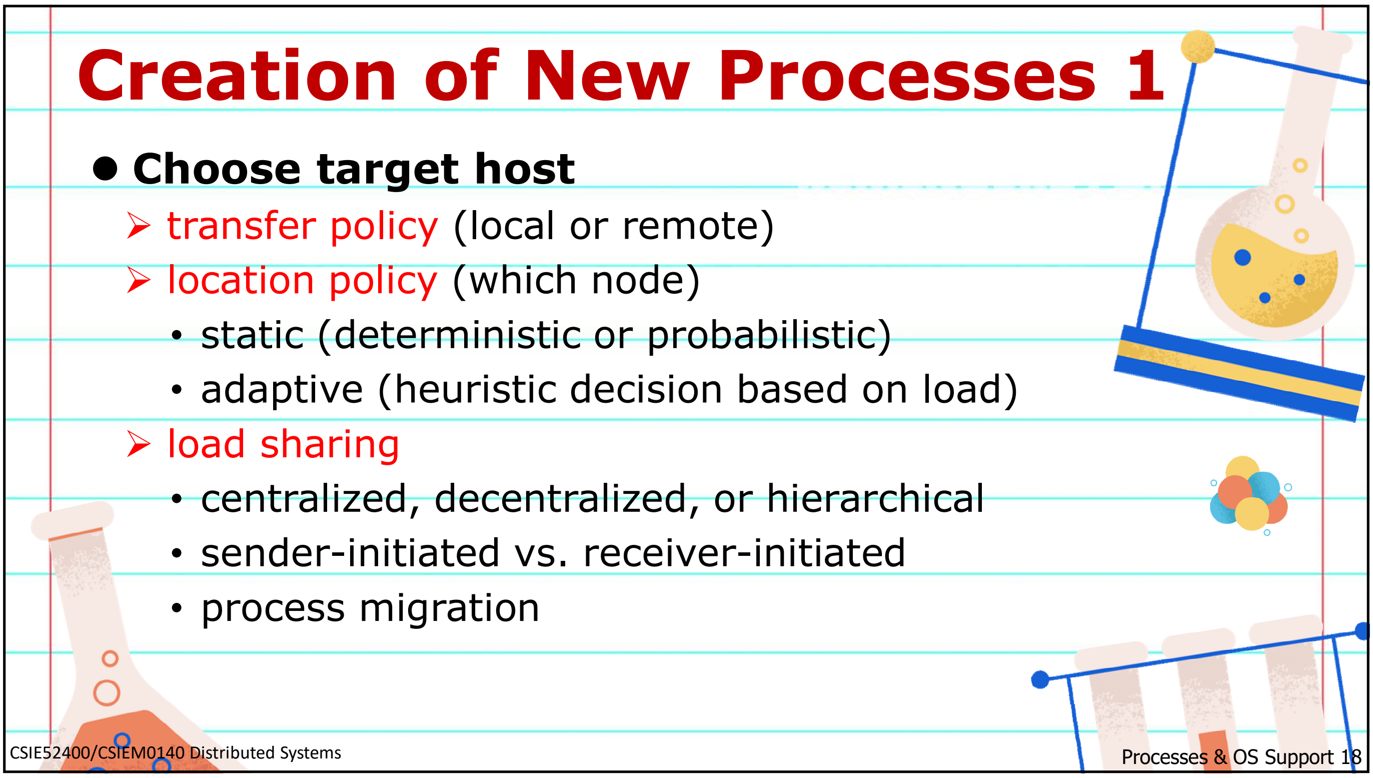
- An area of contiguous virtual memory that is accessible only by the threads of the owning process
- Specified by
  - extent (lowest address and size)
  - RWX permissions
  - growth direction (upwards or downwards)
- Can have shared regions for
  - libraries
  - kernel
  - data sharing and communication



Execution Environments and Threads

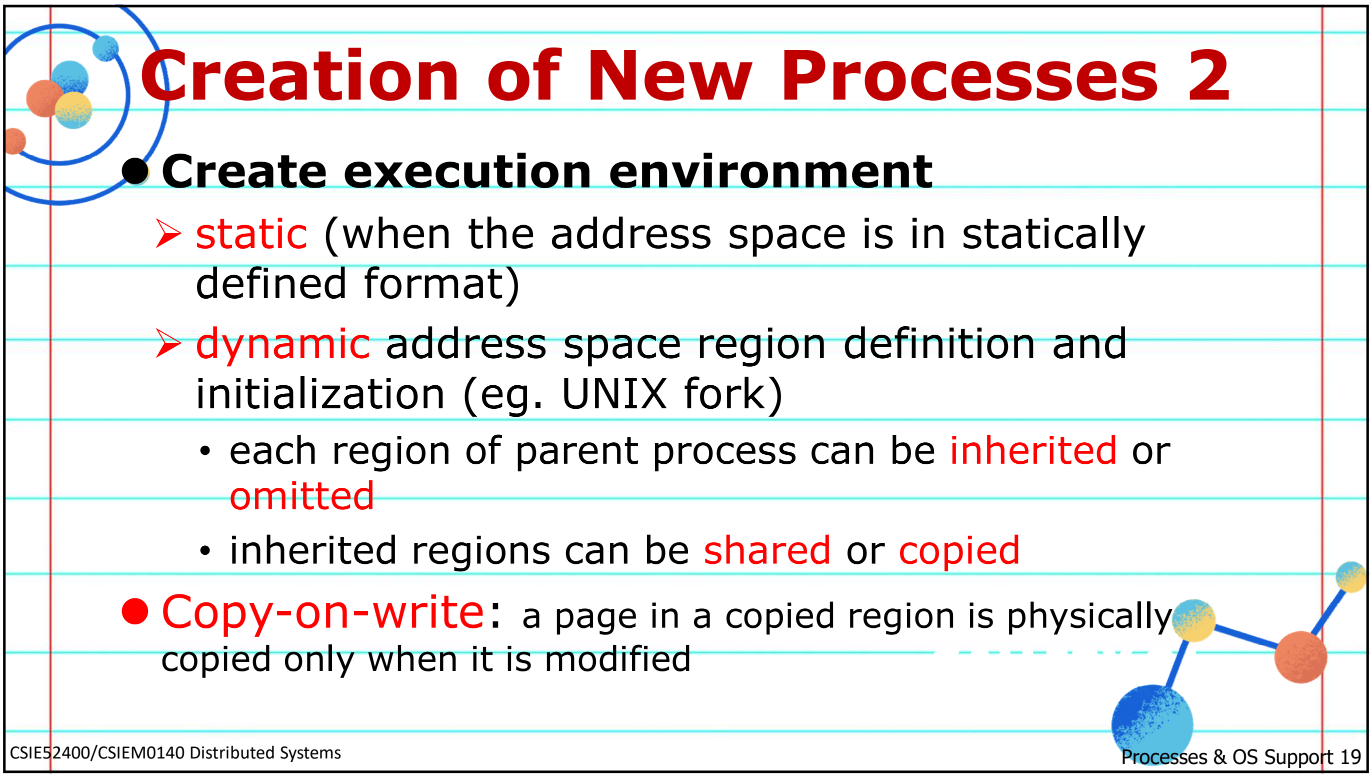
- States associated with execution environment and threads

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	



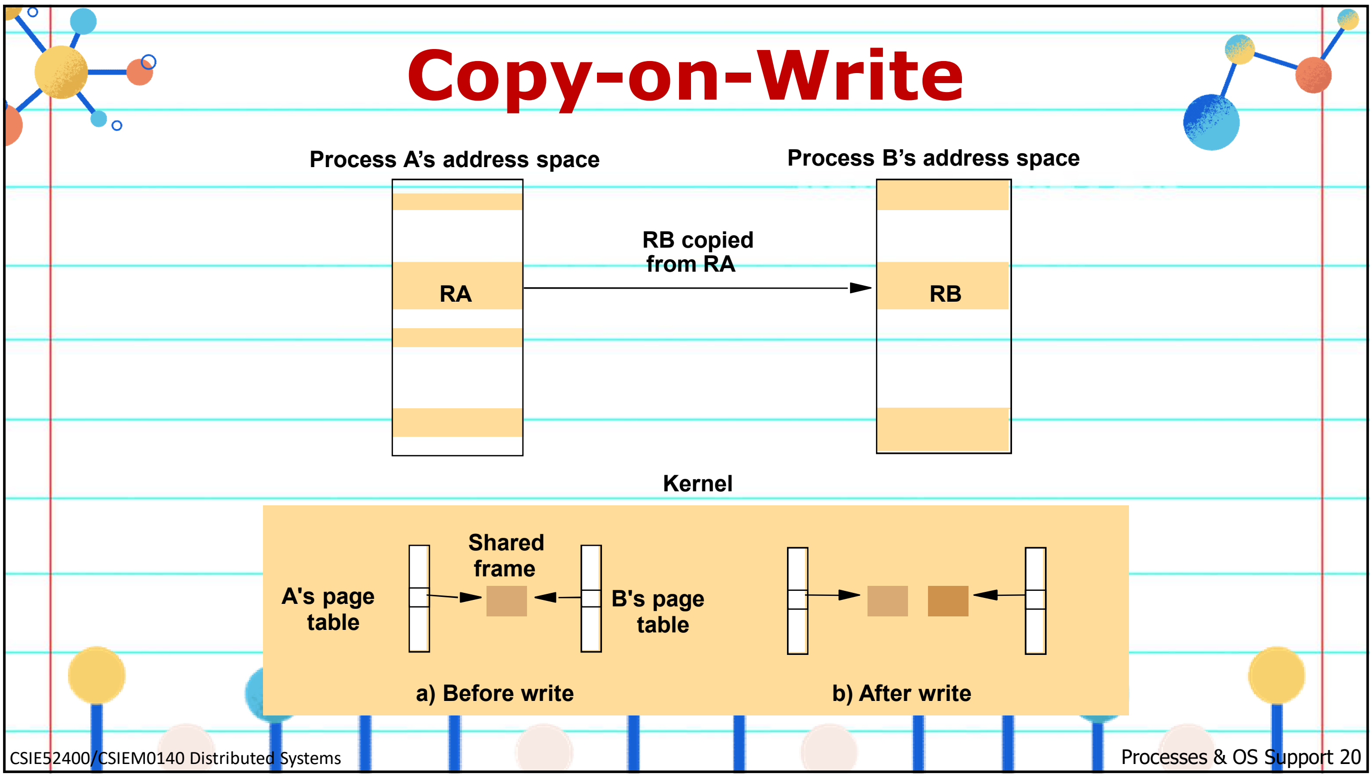
Creation of New Processes 1

- **Choose target host**
  - transfer policy (local or remote)
  - location policy (which node)
    - static (deterministic or probabilistic)
    - adaptive (heuristic decision based on load)
  - load sharing
    - centralized, decentralized, or hierarchical
    - sender-initiated vs. receiver-initiated
    - process migration

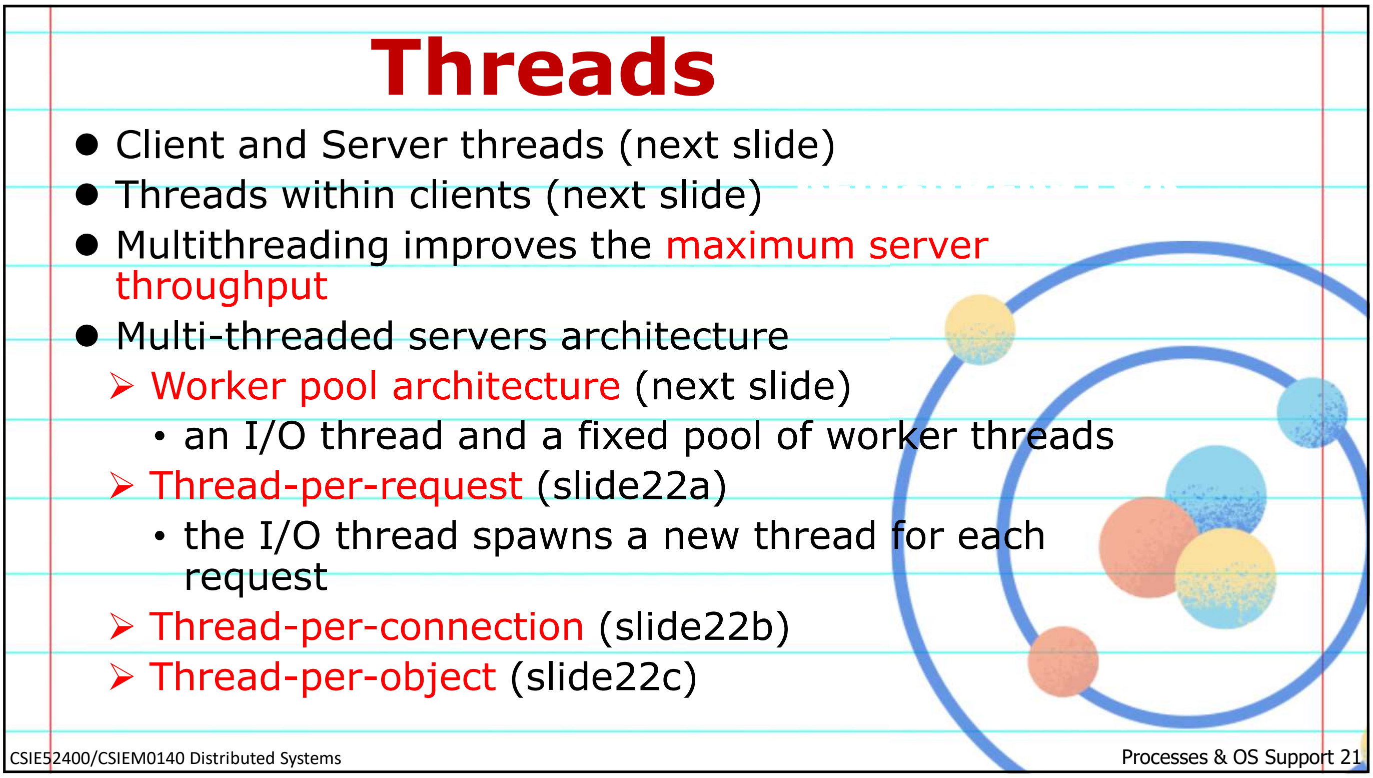


Creation of New Processes 2

- **Create execution environment**
  - static (when the address space is in statically defined format)
  - dynamic address space region definition and initialization (eg. UNIX fork)
    - each region of parent process can be inherited or omitted
    - inherited regions can be shared or copied
- Copy-on-write: a page in a copied region is physically copied only when it is modified

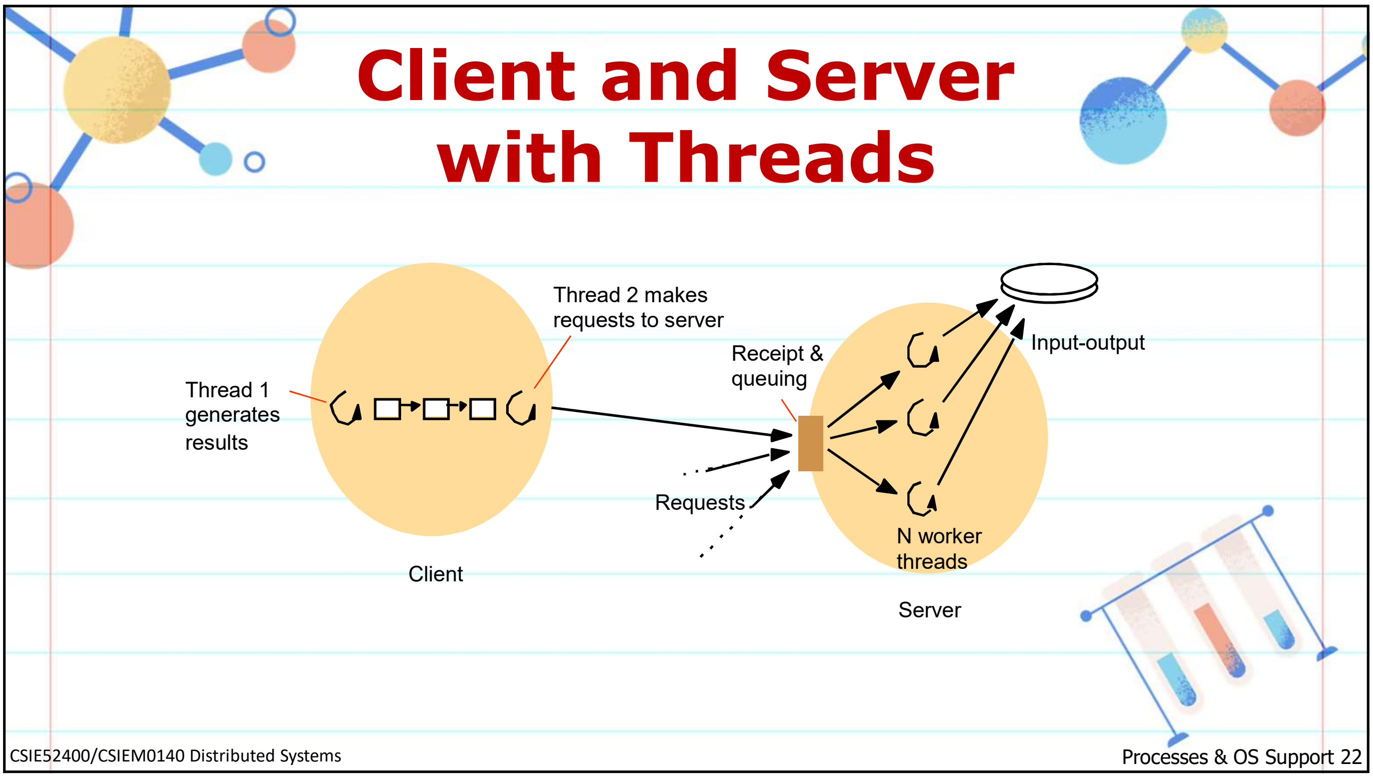


Copy-on-Write



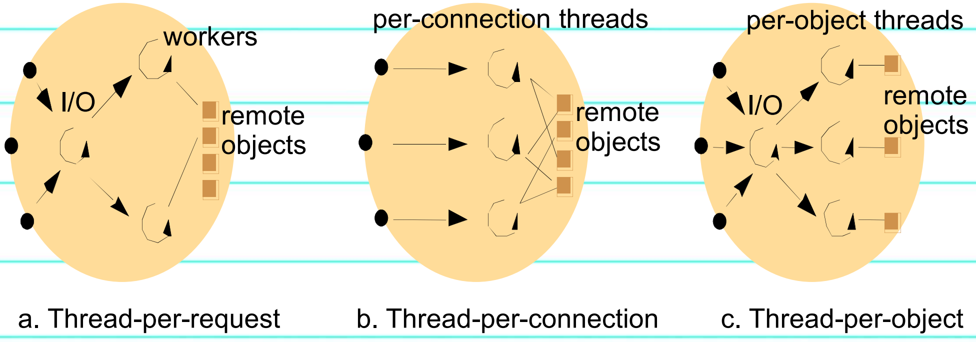
Threads

- Client and Server threads (next slide)
- Threads within clients (next slide)
- Multithreading improves the maximum server throughput
- Multi-threaded servers architecture
  - Worker pool architecture (next slide)
    - an I/O thread and a fixed pool of worker threads
  - Thread-per-request (slide22a)
    - the I/O thread spawns a new thread for each request
  - Thread-per-connection (slide22b)
  - Thread-per-object (slide22c)



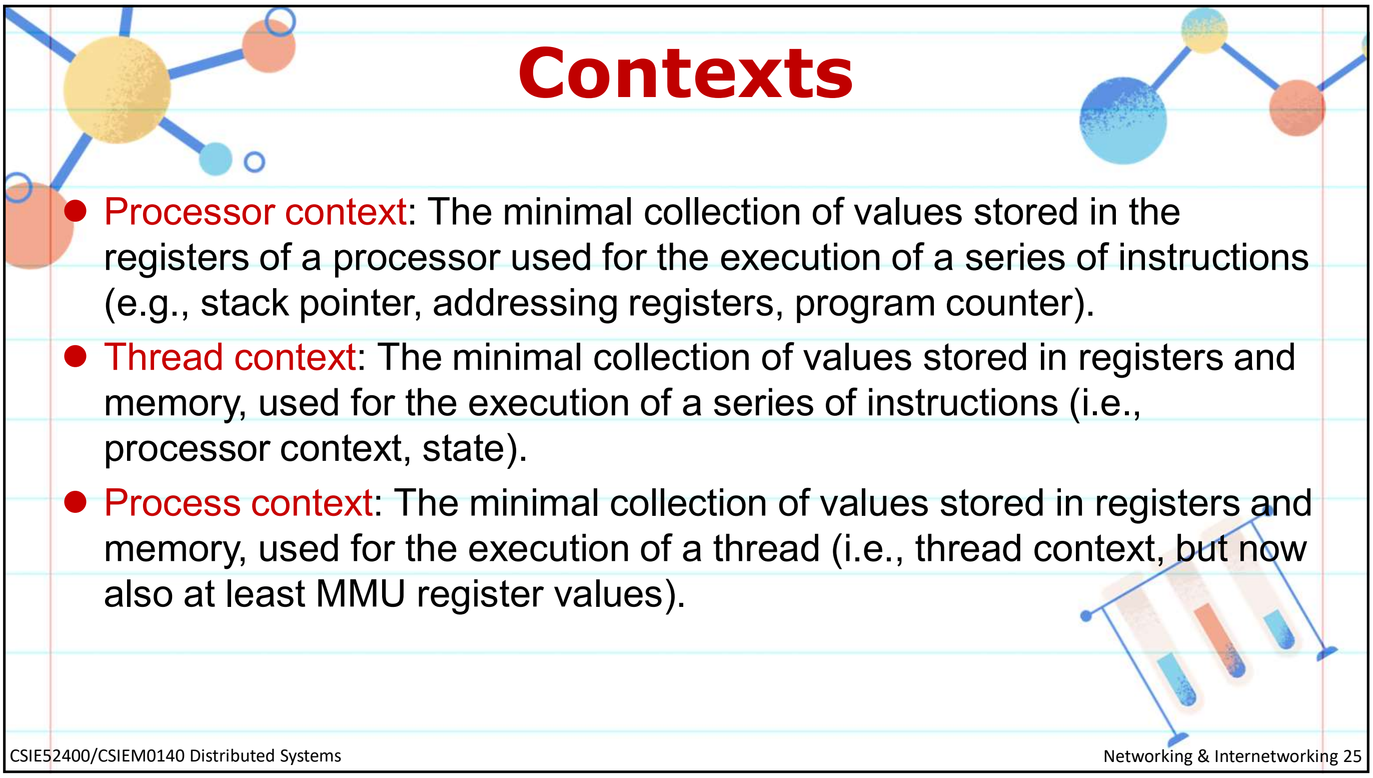
Client and Server with Threads

Alternative Server Threading Architectures



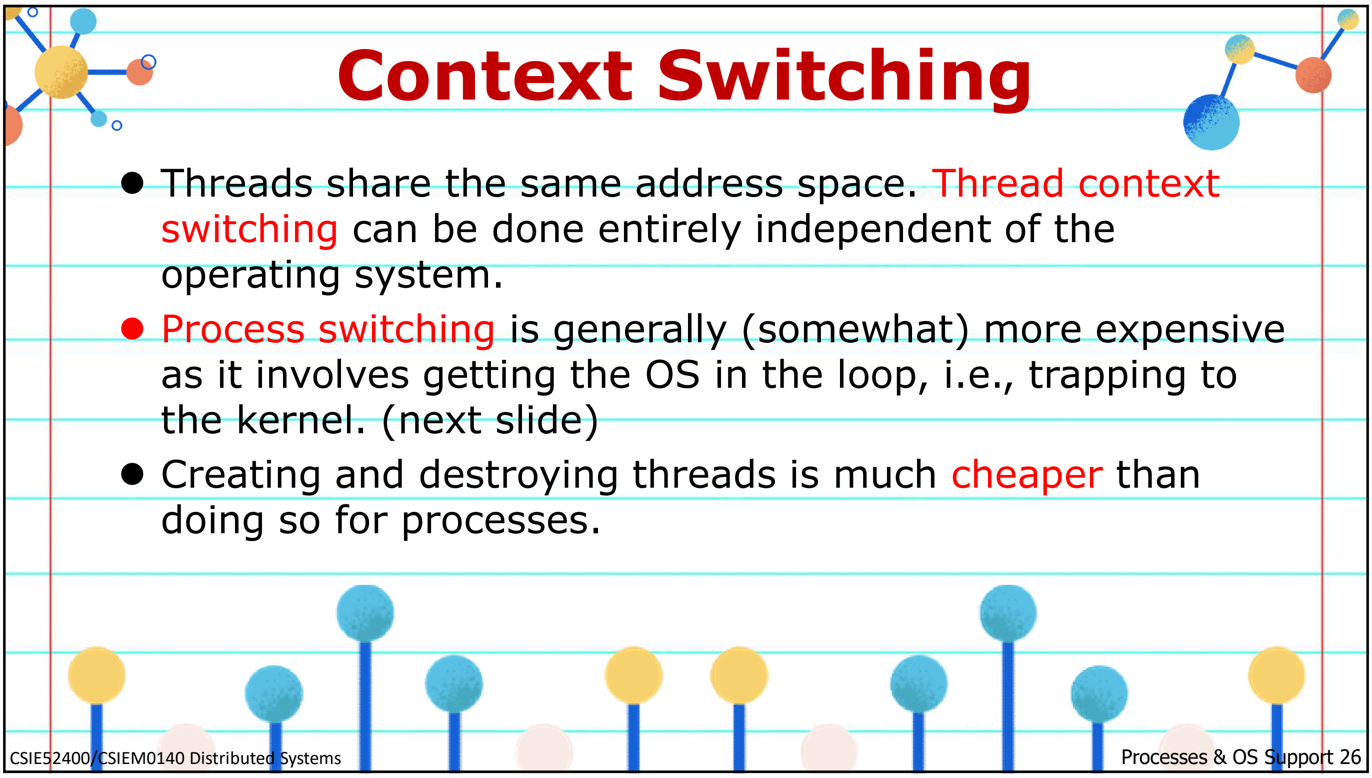
Why Multi-threaded Model

- Allows overlap of memory access of one with computation by another thread/process
- Reduce unnecessary blocking
- Need many computing entities (exploit parallelism)
- Threads are cheaper to create and manage
- Switching b/w threads is much cheaper (next slide)
- Resource sharing is easier and more efficient between threads that share the same exec env
- Useful in the context of large applications
- Many applications are simply easier to structure as a collection of cooperating threads



Contexts

- Processor context: The minimal collection of values stored in the registers of a processor used for the execution of a series of instructions (e.g., stack pointer, addressing registers, program counter).
- Thread context: The minimal collection of values stored in registers and memory, used for the execution of a series of instructions (i.e., processor context, state).
- Process context: The minimal collection of values stored in registers and memory, used for the execution of a thread (i.e., thread context, but now also at least MMU register values).

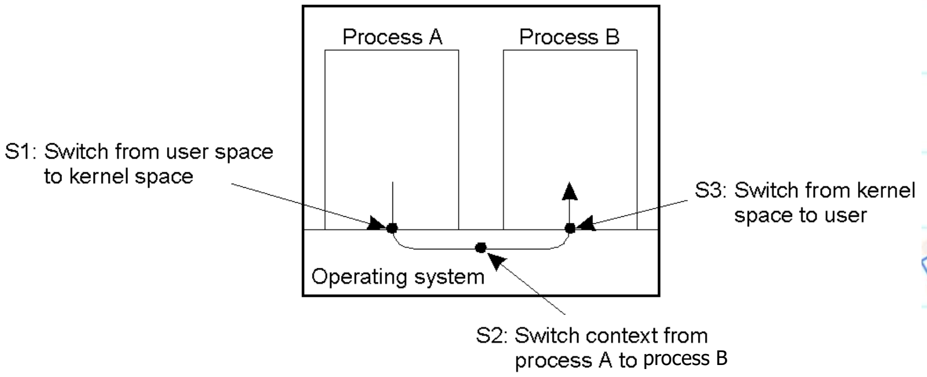


Context Switching

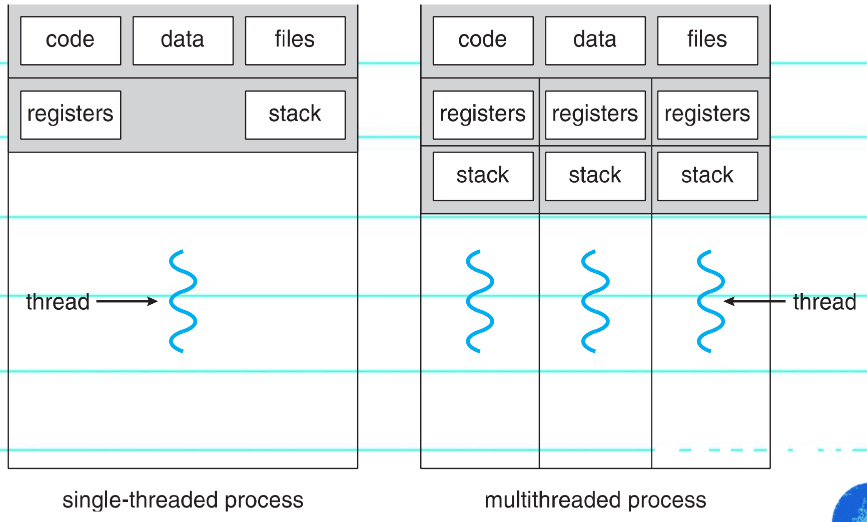
- Threads share the same address space. Thread context switching can be done entirely independent of the operating system.
- Process switching is generally (somewhat) more expensive as it involves getting the OS in the loop, i.e., trapping to the kernel. (next slide)
- Creating and destroying threads is much cheaper than doing so for processes.

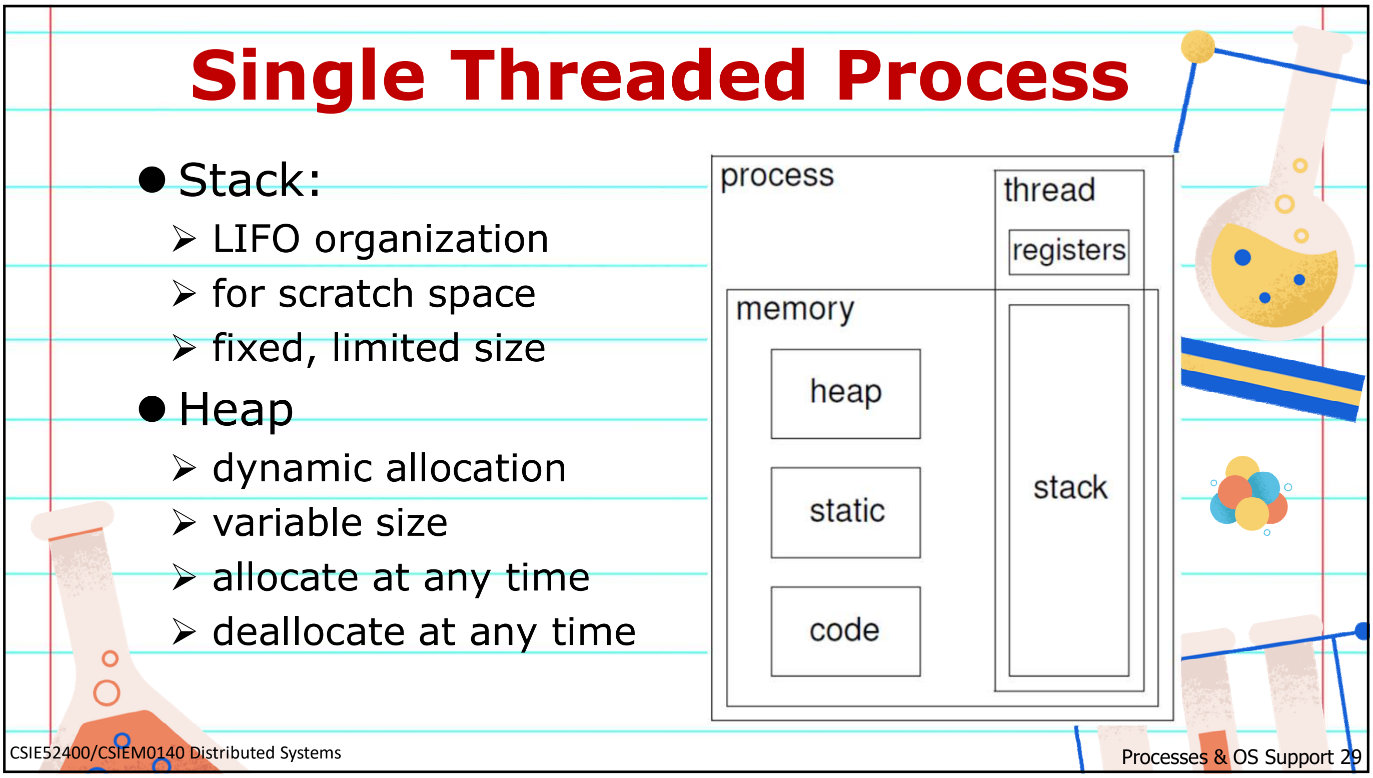
Drawback of IPC

- When structuring large application using multiple processes, the cost of IPC can be very high
- Context switching as the result of IPC


Process A
Process B
Operating system
S1: Switch from user space to kernel space
S2: Switch context from process A to process B
S3: Switch from kernel space to user

Single and Multithreaded Processes





Single Threaded Process

- Stack:
  - LIFO organization
  - for scratch space
  - fixed, limited size
- Heap
  - dynamic allocation
  - variable size
  - allocate at any time
  - deallocate at any time

process

thread

registers

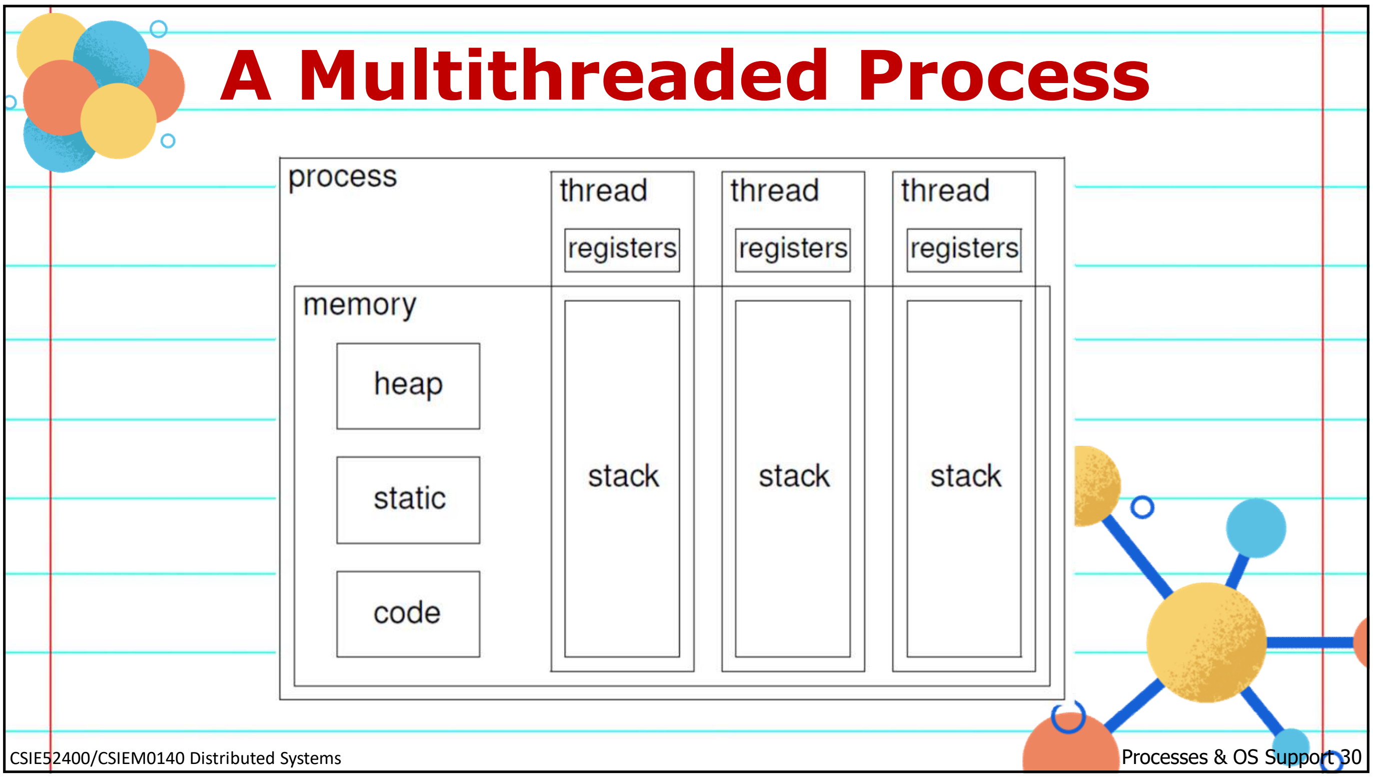
memory

heap

static

code

stack



A Multithreaded Process

process

thread

registers

thread

registers

thread

registers

memory

heap

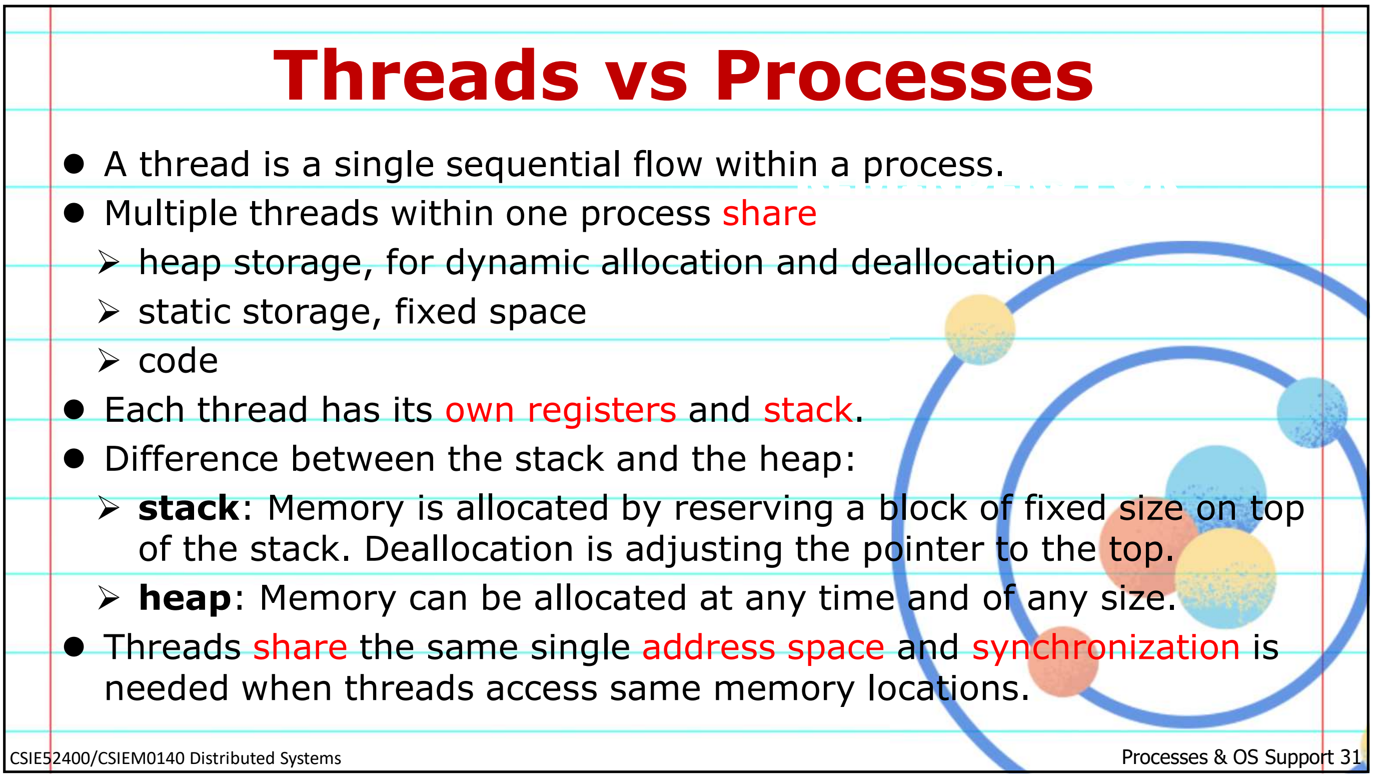
static

code

stack

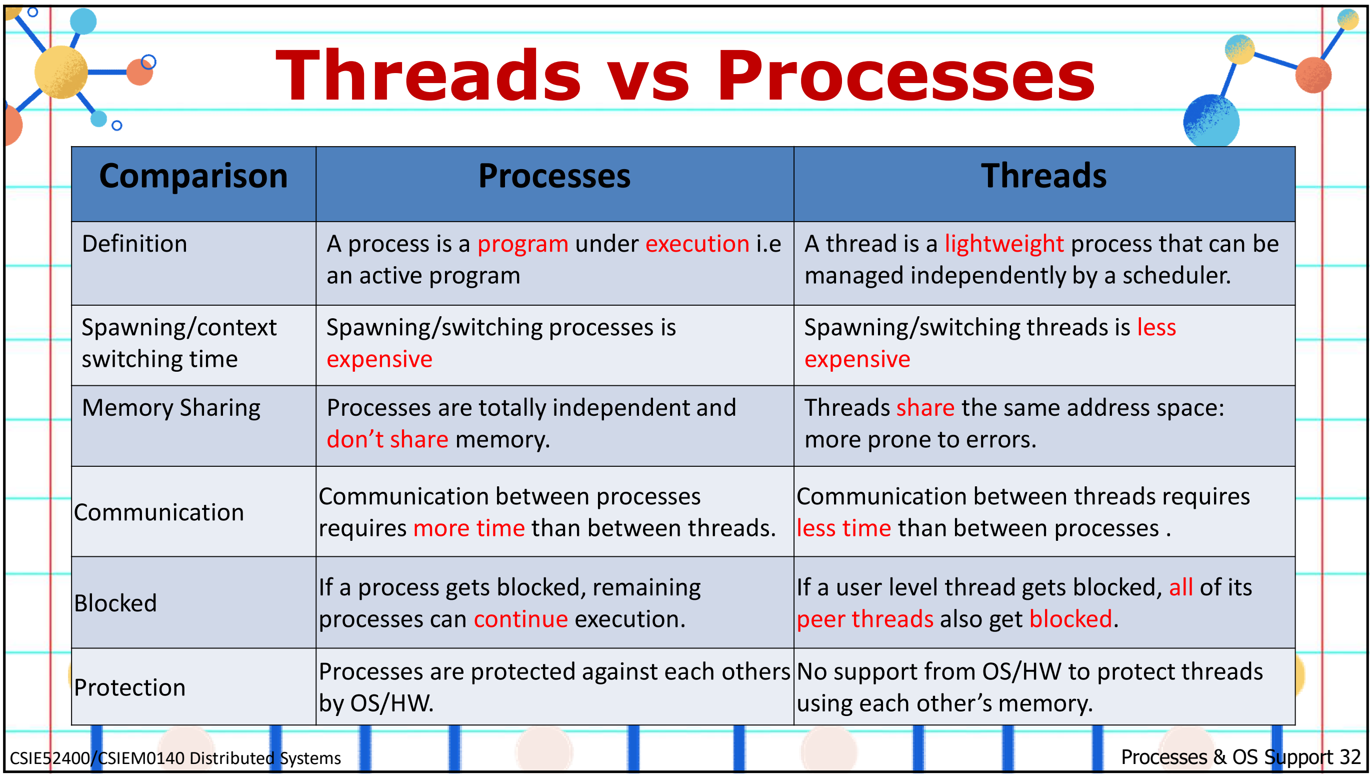
stack

stack



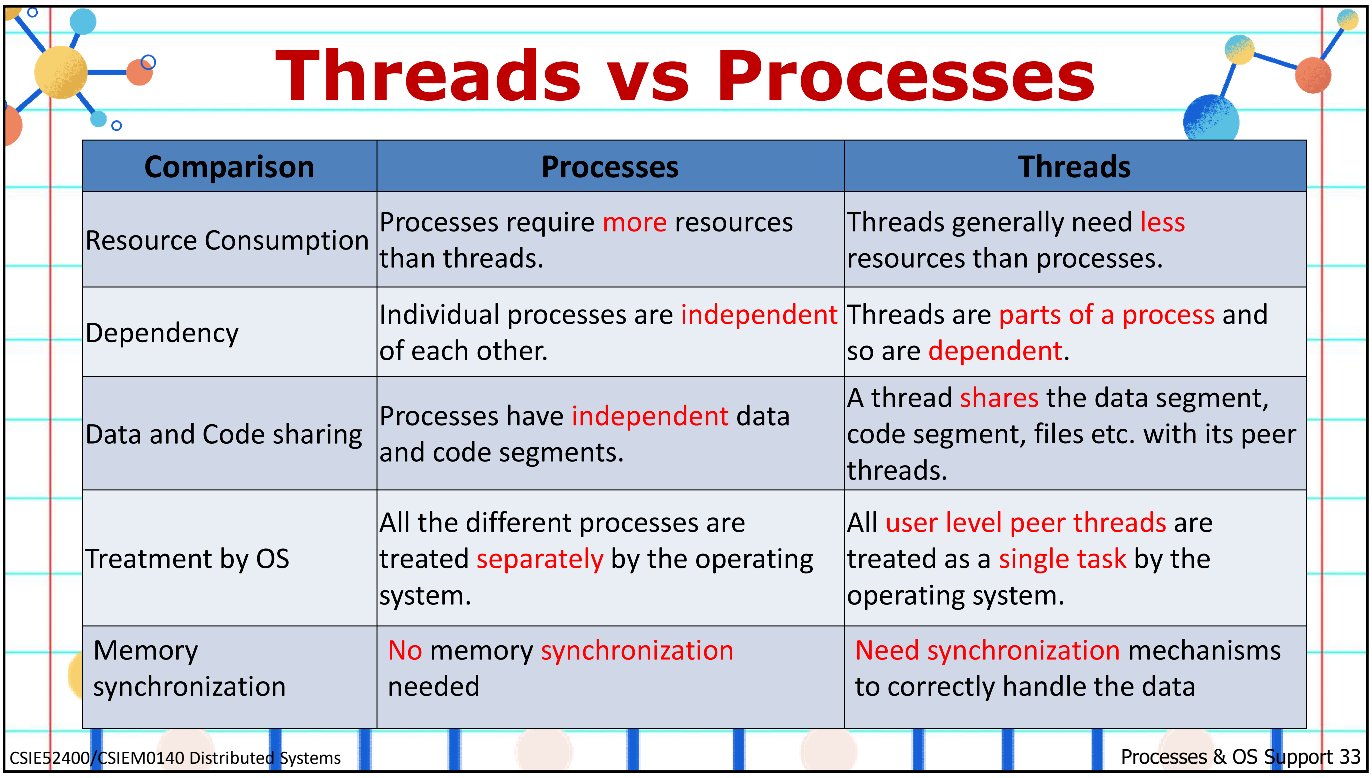
Threads vs Processes

- A thread is a single sequential flow within a process.
- Multiple threads within one process share
  - heap storage, for dynamic allocation and deallocation
  - static storage, fixed space
  - code
- Each thread has its own registers and stack.
- Difference between the stack and the heap:
  - **stack**: Memory is allocated by reserving a block of fixed size on top of the stack. Deallocation is adjusting the pointer to the top.
  - **heap**: Memory can be allocated at any time and of any size.
- Threads share the same single address space and synchronization is needed when threads access same memory locations.



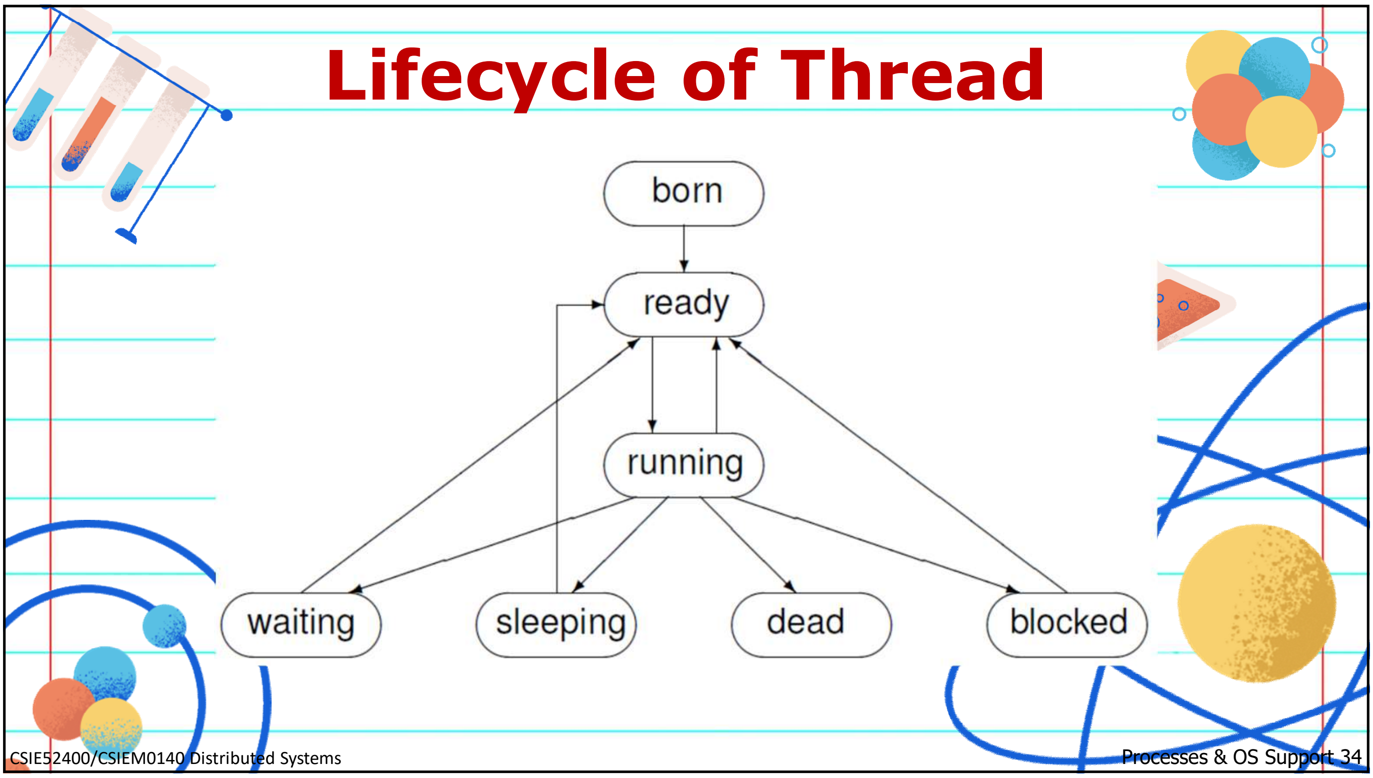
Threads vs Processes

Comparison	Processes	Threads
Definition	A process is a program under execution i.e an active program	A thread is a lightweight process that can be managed independently by a scheduler.
Spawning/context switching time	Spawning/switching processes is expensive	Spawning/switching threads is less expensive
Memory Sharing	Processes are totally independent and don't share memory.	Threads share the same address space: more prone to errors.
Communication	Communication between processes requires more time than between threads.	Communication between threads requires less time than between processes .
Blocked	If a process gets blocked, remaining processes can continue execution.	If a user level thread gets blocked, all of its peer threads also get blocked.
Protection	Processes are protected against each others by OS/HW.	No support from OS/HW to protect threads using each other's memory.

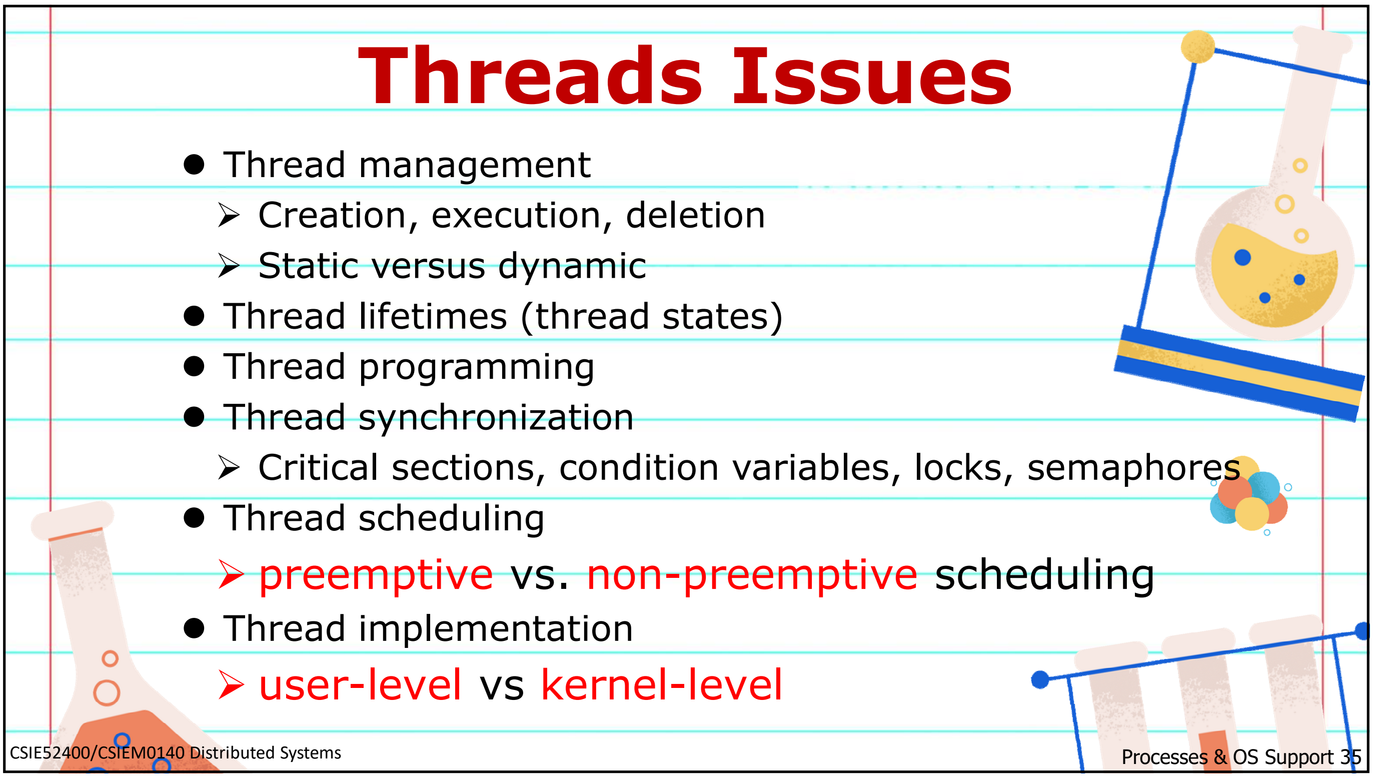


Threads vs Processes

Comparison	Processes	Threads
Resource Consumption	Processes require more resources than threads.	Threads generally need less resources than processes.
Dependency	Individual processes are independent of each other.	Threads are parts of a process and so are dependent.
Data and Code sharing	Processes have independent data and code segments.	A thread shares the data segment, code segment, files etc. with its peer threads.
Treatment by OS	All the different processes are treated separately by the operating system.	All user level peer threads are treated as a single task by the operating system.
Memory synchronization	No memory synchronization needed	Need synchronization mechanisms to correctly handle the data

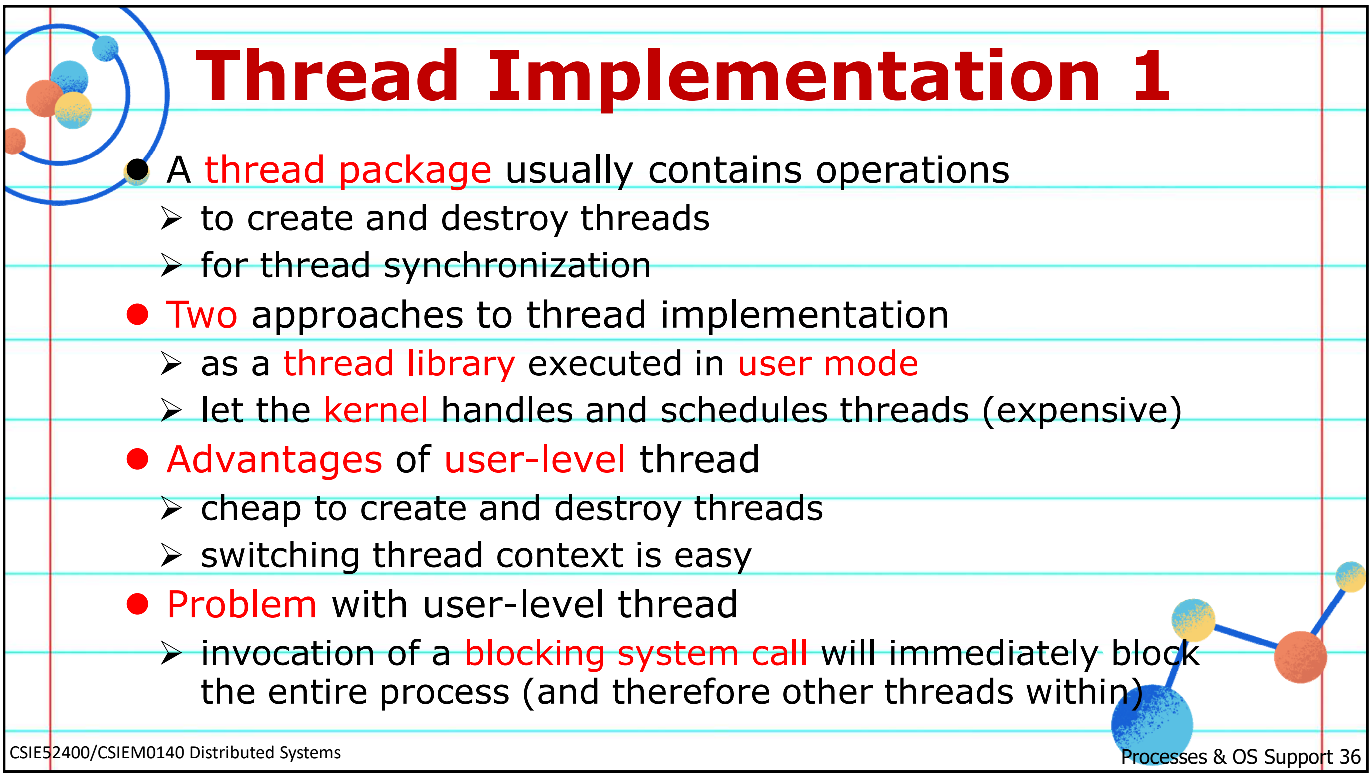


Lifecycle of Thread



Threads Issues

- Thread management
  - Creation, execution, deletion
  - Static versus dynamic
- Thread lifetimes (thread states)
- Thread programming
- Thread synchronization
  - Critical sections, condition variables, locks, semaphores
- Thread scheduling
  - preemptive vs. non-preemptive scheduling
- Thread implementation
  - user-level vs kernel-level



Thread Implementation 1

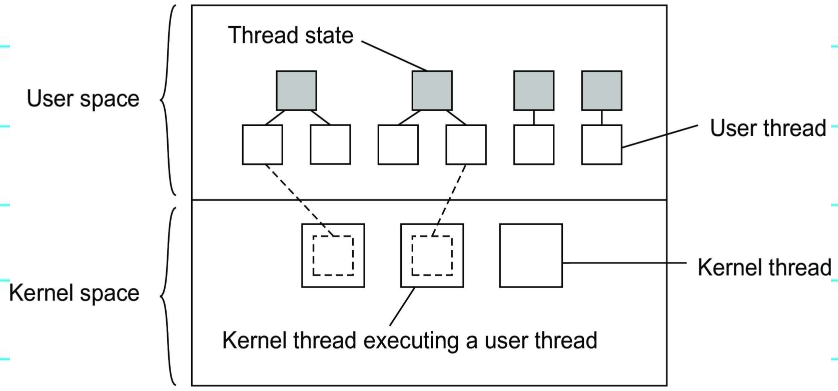
- A thread package usually contains operations
  - to create and destroy threads
  - for thread synchronization
- Two approaches to thread implementation
  - as a thread library executed in user mode
  - let the kernel handles and schedules threads (expensive)
- Advantages of user-level thread
  - cheap to create and destroy threads
  - switching thread context is easy
- Problem with user-level thread
  - invocation of a blocking system call will immediately block the entire process (and therefore other threads within)

Thread Implementation 2

- Kernel solution: the kernel contain the implemenetation of a thread package. All operations return as system calls.
- Operations that block a thread are no longer a problem: the kernel schedules another available thread within the same process.
- Handling external events is simple: the kernel (which catches all events) schedules the thread associated with the event.
- The problem is (or used to be) the loss of efficiency because each thread operation requires a trap to the kernel.
- Try to mix user-level and kernel-level threads into a single concept.
- Performance gain has not turned out to generally outweigh the increased complexity.

Thread Implementation 3

- Introduce a two-level threading approach: kernel threads that can execute user-level threads.



Light-Weight Processes (LWPs)

- Several LWPs per heave-weight process
- User-level threads package
  - ➢ Create/destroy threads and synchronization primitives
- Multithreaded applications – create multiple threads, assign threads to LWPs (one-one, many-one, many-many)
- Each LWP, when scheduled, searches for a runnable thread (two-level scheduling)
  - ➢ Shared thread table: no kernel support needed
- When a LWP thread block on system call, switch to kernel mode and OS context switches to another LWP

Two-Level Scheduling of LWPs

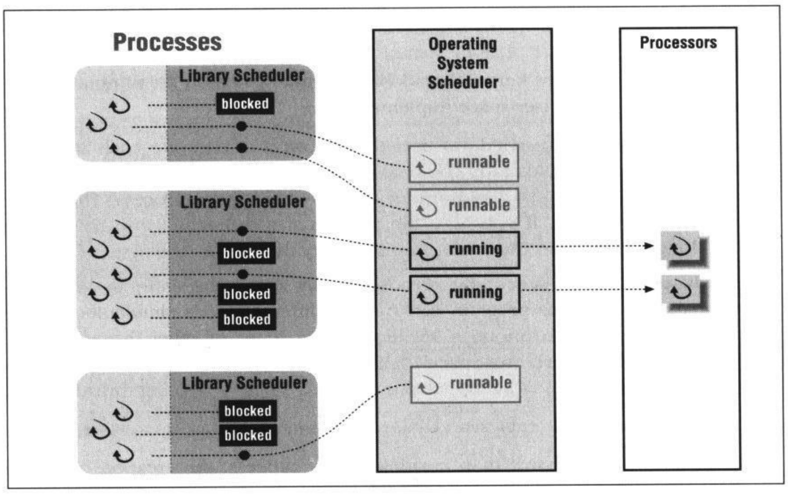
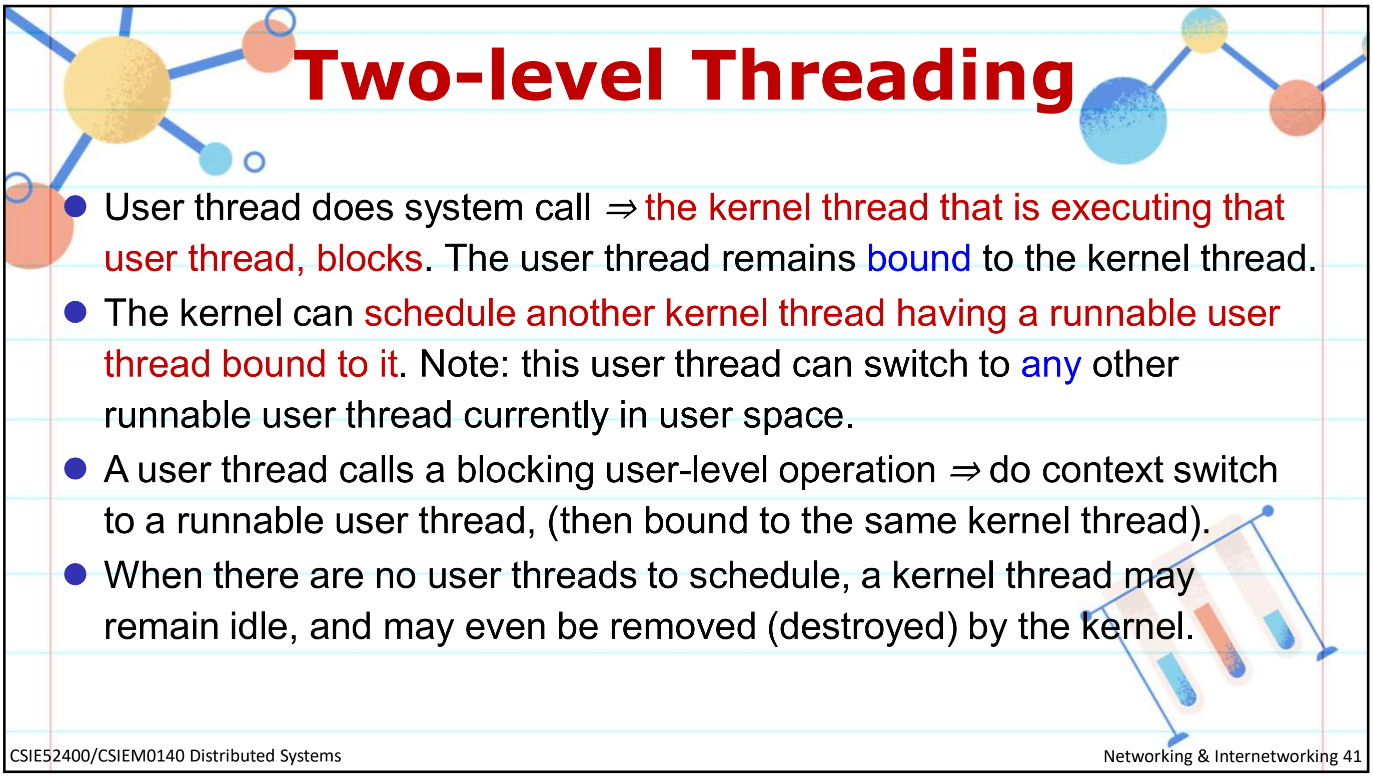
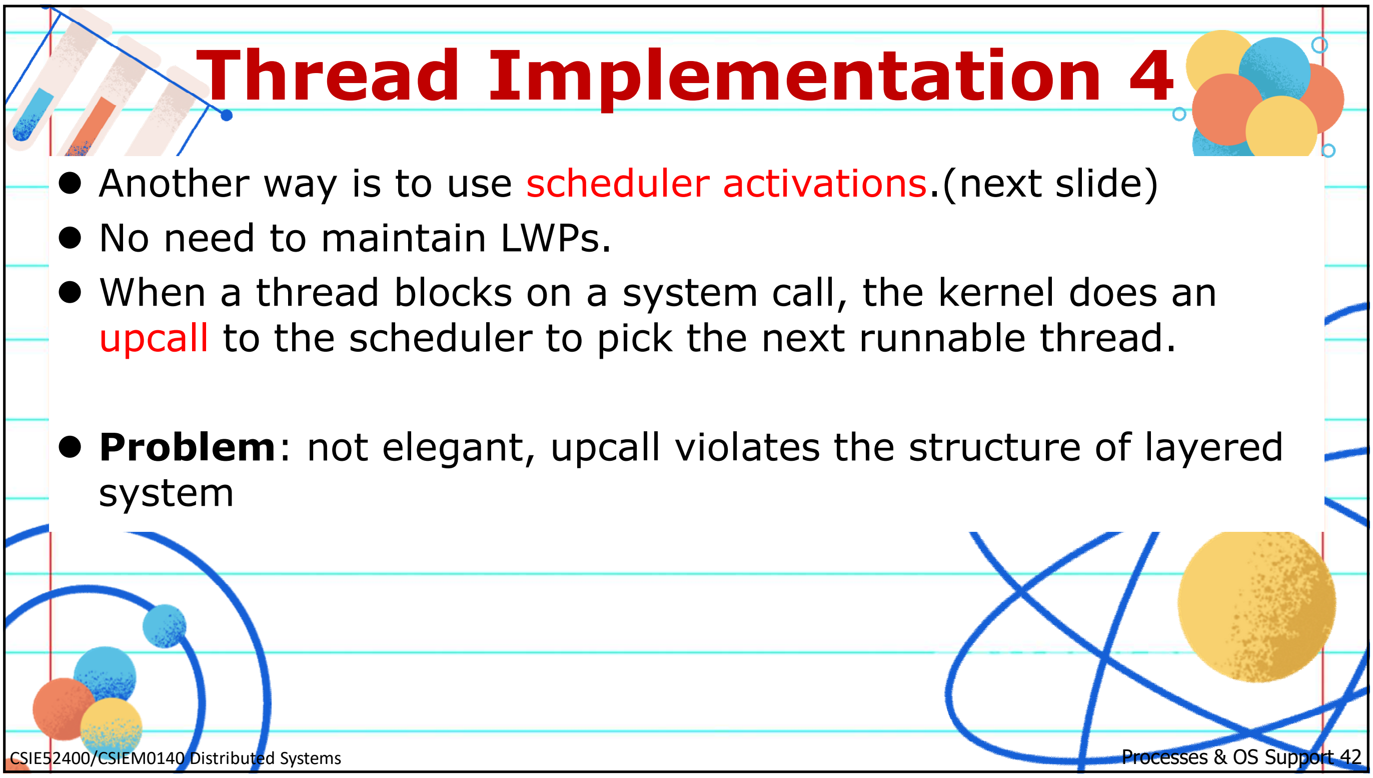


Figure 6–3: Two-level scheduler implementations



Two-level Threading

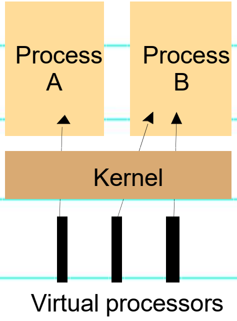
- User thread does system call ⇒ the kernel thread that is executing that user thread, blocks. The user thread remains bound to the kernel thread.
- The kernel can schedule another kernel thread having a runnable user thread bound to it. Note: this user thread can switch to any other runnable user thread currently in user space.
- A user thread calls a blocking user-level operation ⇒ do context switch to a runnable user thread, (then bound to the same kernel thread).
- When there are no user threads to schedule, a kernel thread may remain idle, and may even be removed (destroyed) by the kernel.



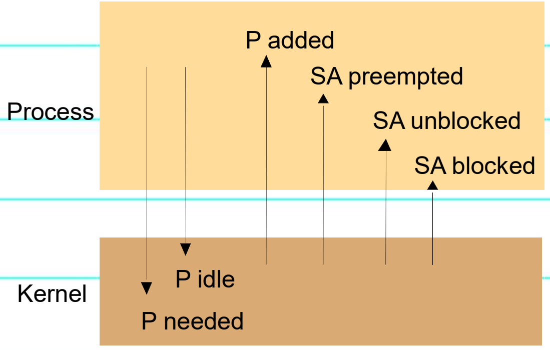
Thread Implementation 4

- Another way is to use scheduler activations.(next slide)
- No need to maintain LWPs.
- When a thread blocks on a system call, the kernel does an upcall to the scheduler to pick the next runnable thread.
- **Problem**: not elegant, upcall violates the structure of layered system

Scheduler Activations (FastThreads)



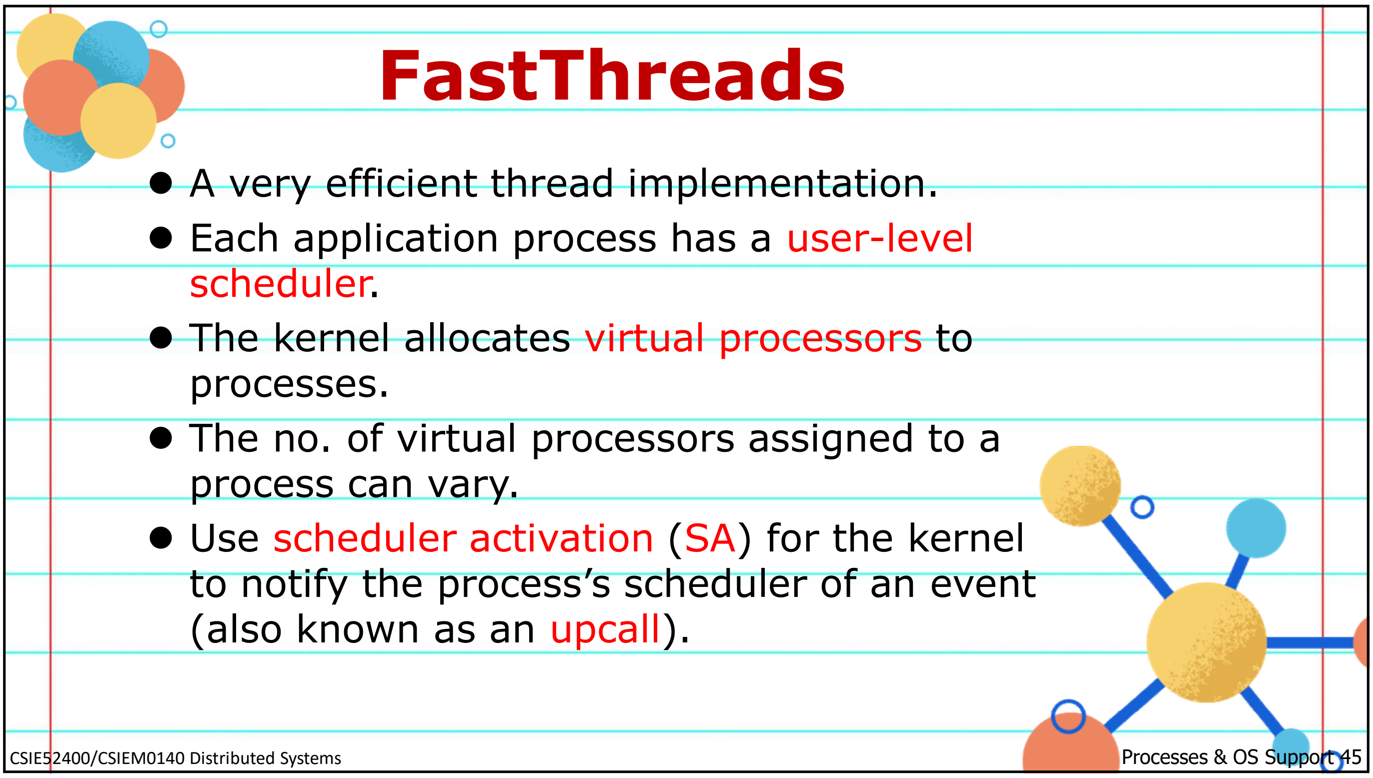
A. Assignment of virtual processors to processes



B. Events between user-level scheduler & kernel
Key: P = processor; SA = scheduler activation

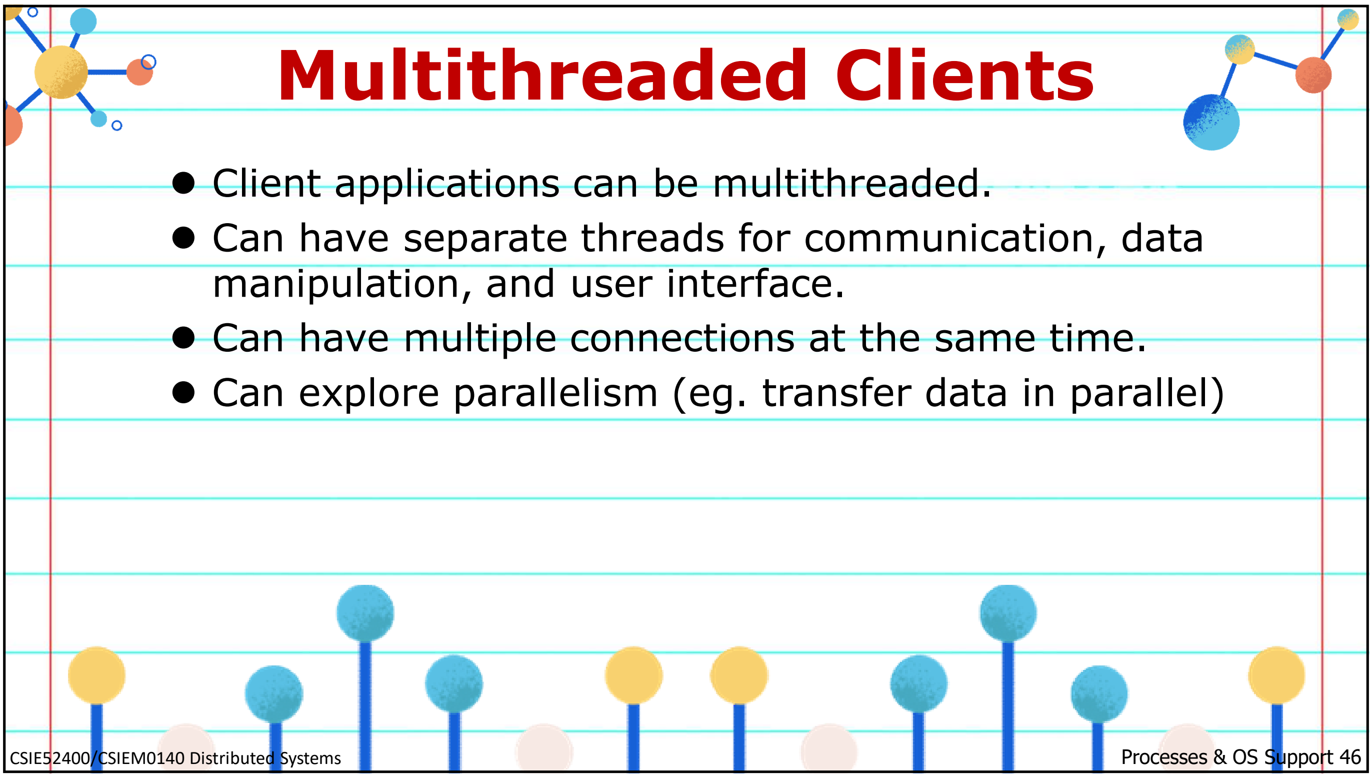
Thread Libraries

- Posix Threads (pThreads)
  - Widely used threads library
  - Conforms to the Posix standard
  - Sample calls: pthread_create,…
  - Typical used in C/C++ applications
  - Can be user-level or kernel-level or via LWPs
- Windows Threads
  - Similar to pThreads for Windows
- Java Threads
  - Native thread support built into the language
  - Threads are scheduled by the JVM
- OpenThreads
  - From the OpenSceneGraph project
  - Intended to provide a minimal and complete object-oriented thread interface for C++



FastThreads

- A very efficient thread implementation.
- Each application process has a user-level scheduler.
- The kernel allocates virtual processors to processes.
- The no. of virtual processors assigned to a process can vary.
- Use scheduler activation (SA) for the kernel to notify the process's scheduler of an event (also known as an upcall).



Multithreaded Clients

- Client applications can be multithreaded.
- Can have separate threads for communication, data manipulation, and user interface.
- Can have multiple connections at the same time.
- Can explore parallelism (eg. transfer data in parallel)

Multi-threaded Clients : Browsers

- Browsers such as **IE** are multi-threaded
- Such browsers can display data before entire document is downloaded: performs multiple simultaneous tasks
  - ➢ Fetch main HTML page, activate separate threads for other parts
  - ➢ Each thread sets up a separate connection with the server
    - Uses blocking HTTP request
  - ➢ Each part (eg. gif image file) fetched separately and in parallel
- Multiple request-response calls to other machines (RPC)
  - ➢ Several calls at the same time, each one by a different thread
  - ➢ Then waits until all results have been returned
  - ➢ Note: if calls are to different servers, we may have a linear speed-up

Thread-level Parallelism(TLP)

- Let c_i denote the fraction of time that exactly *i* threads are being executed simultaneously

$$TLP = \frac{\sum_{i=1}^{N} i \times c_i}{1 - c_0}$$

with *N* the maximum number of threads that (can) execute at the same time.

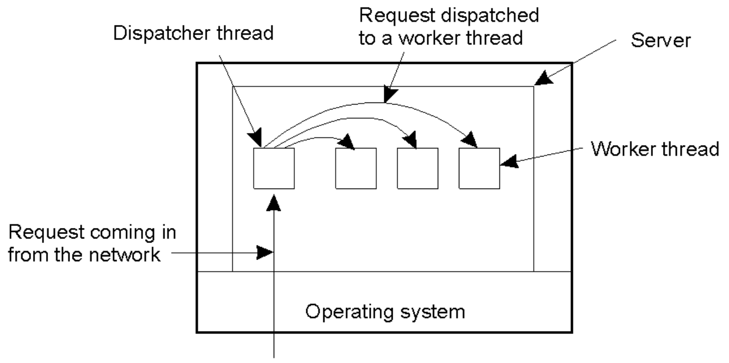
- Practical measurements: A typical Web browser has a TLP value between 1.5 and 2.5 ⇒ threads are primarily used for logically organizing browsers.

Multithreaded Servers

- Improve performance
  - Starting a thread is cheaper than starting a new process.
  - 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.

Multithreaded Servers

- A multithreaded server organized in a dispatcher/worker model.


Dispatcher thread
Request dispatched to a worker thread
Server
Worker thread
Request coming in from the network
Operating system

Multi-threaded Server Example

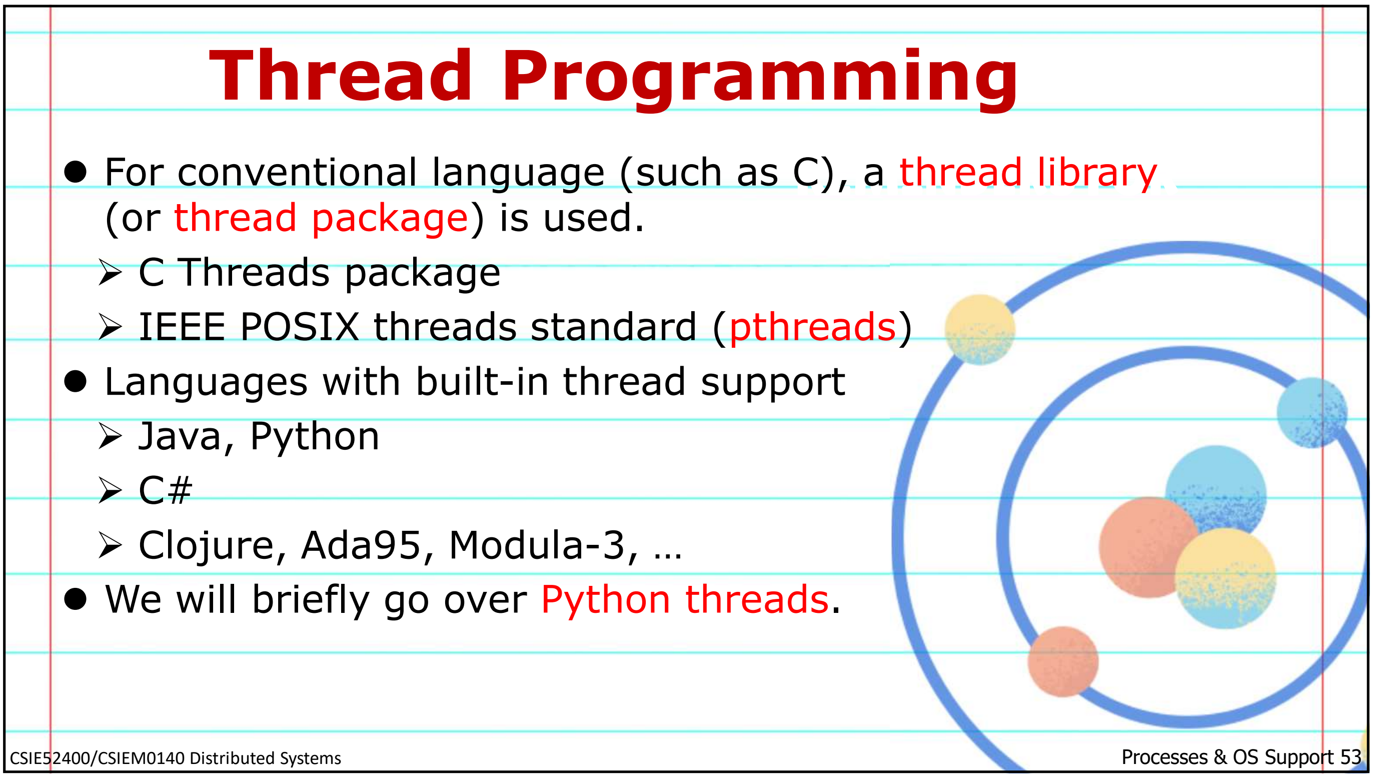
- **Apache** web server: pool of pre-spawned worker threads
- Dispatcher thread waits for requests
- For each request, choose an idle worker thread
- Worker thread uses blocking system calls to service web request

Server Construction

- Three ways to construct a server.

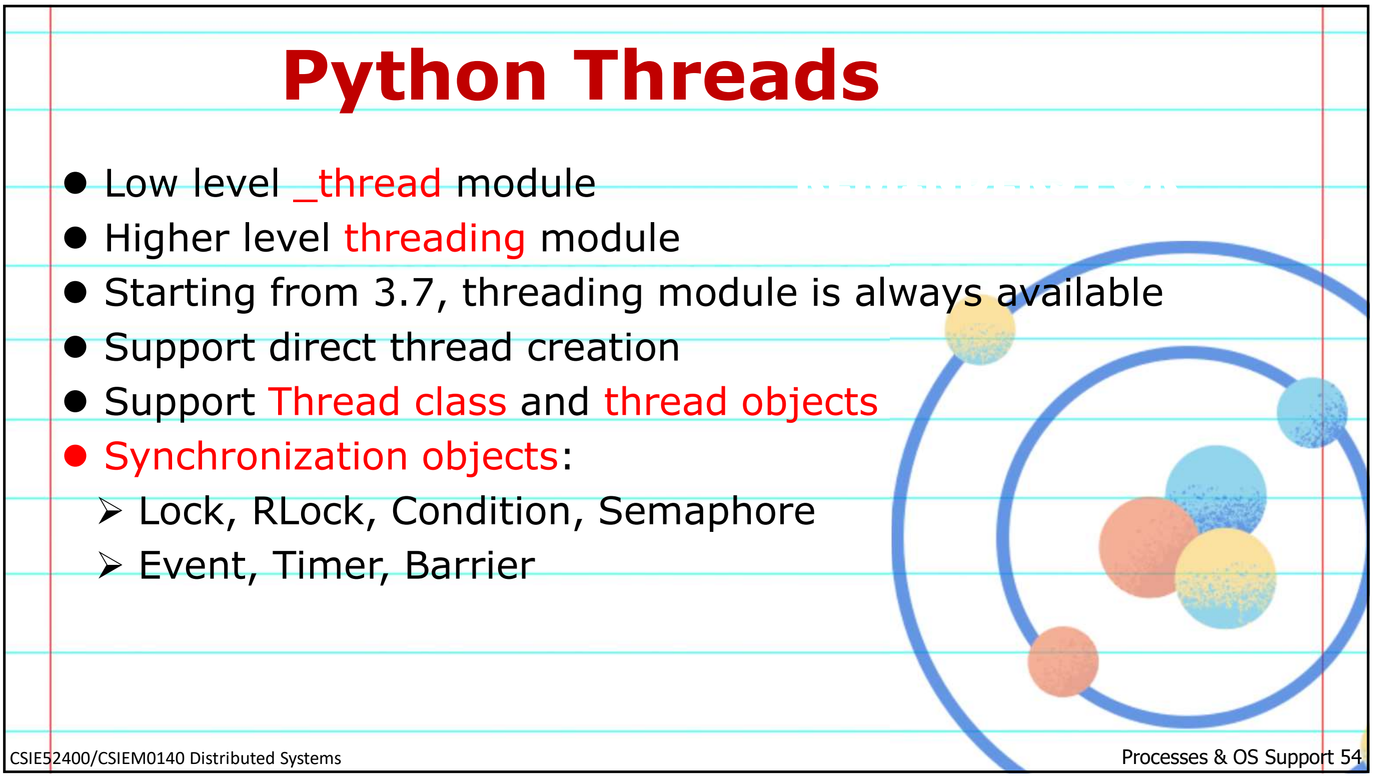
Model	Characteristics
Multithreading	Parallelism, blocking system calls
Single-threaded process	No parallelism, blocking system calls
Finite-state machine	Parallelism, nonblocking system calls

- The multithreaded model retains the "sequential process" model which is much easier to program and still achieve parallelism.



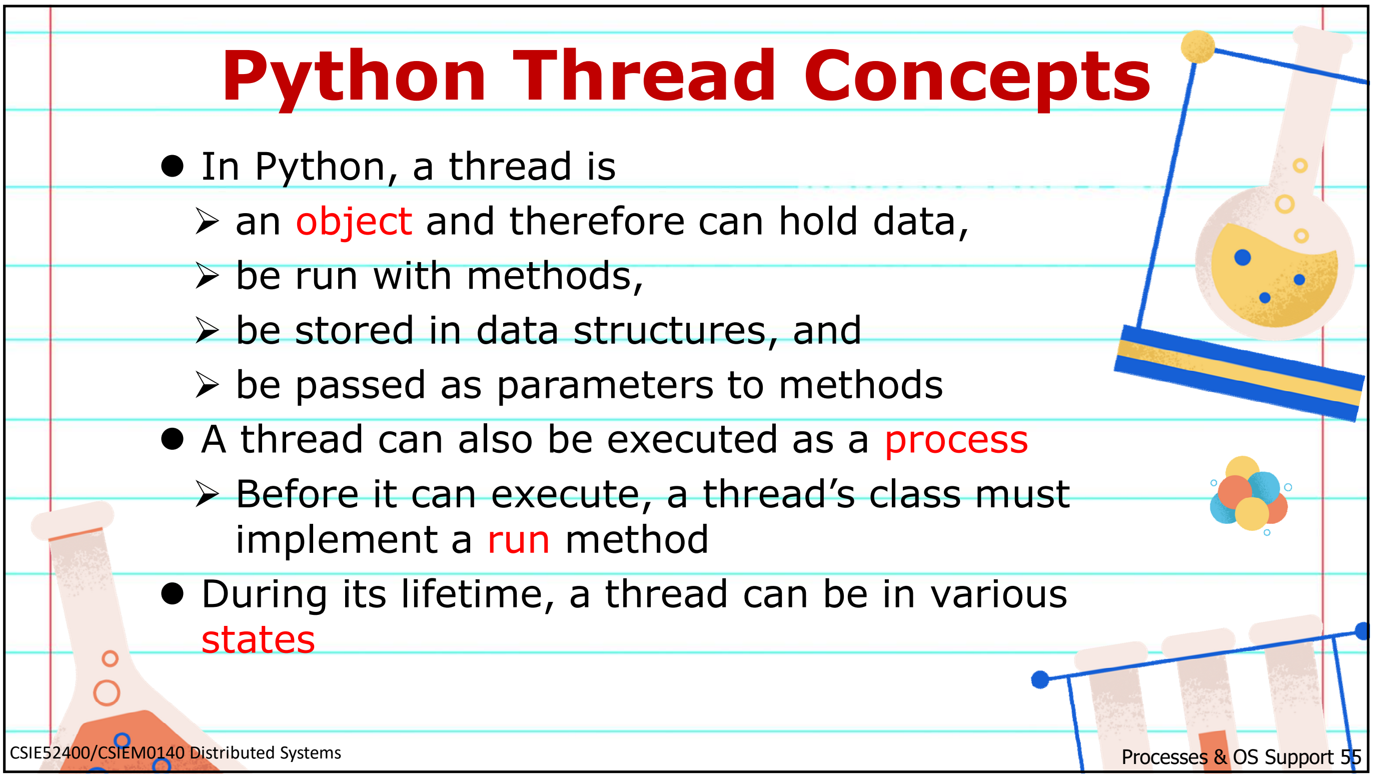
Thread Programming

- For conventional language (such as C), a thread library (or thread package) is used.
  - C Threads package
  - IEEE POSIX threads standard (pthreads)
- Languages with built-in thread support
  - Java, Python
  - C#
  - Clojure, Ada95, Modula-3, …
- We will briefly go over Python threads.



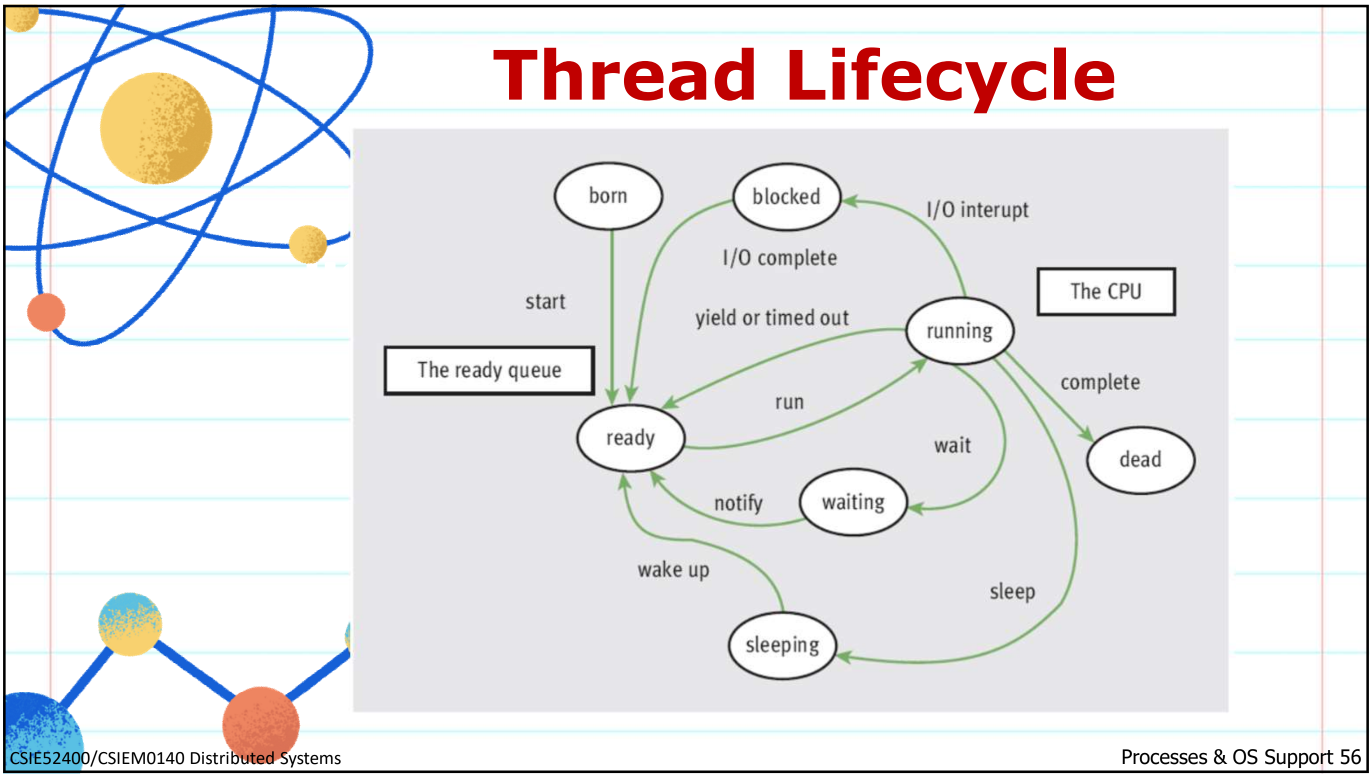
Python Threads

- Low level _thread module
- Higher level threading module
- Starting from 3.7, threading module is always available
- Support direct thread creation
- Support Thread class and thread objects
- Synchronization objects:
  - Lock, RLock, Condition, Semaphore
  - Event, Timer, Barrier

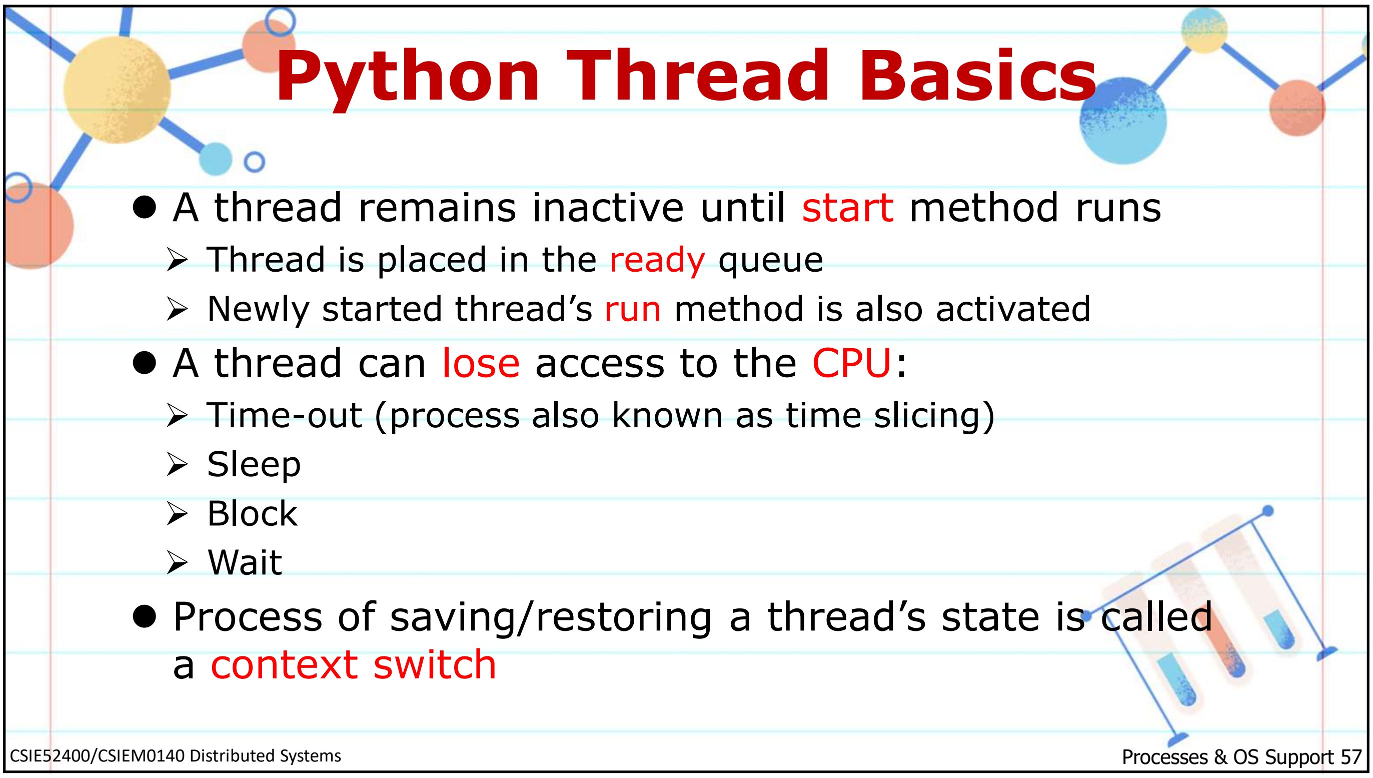


Python Thread Concepts

- In Python, a thread is
  - an object and therefore can hold data,
  - be run with methods,
  - be stored in data structures, and
  - be passed as parameters to methods
- A thread can also be executed as a process
  - Before it can execute, a thread's class must implement a run method
- During its lifetime, a thread can be in various states

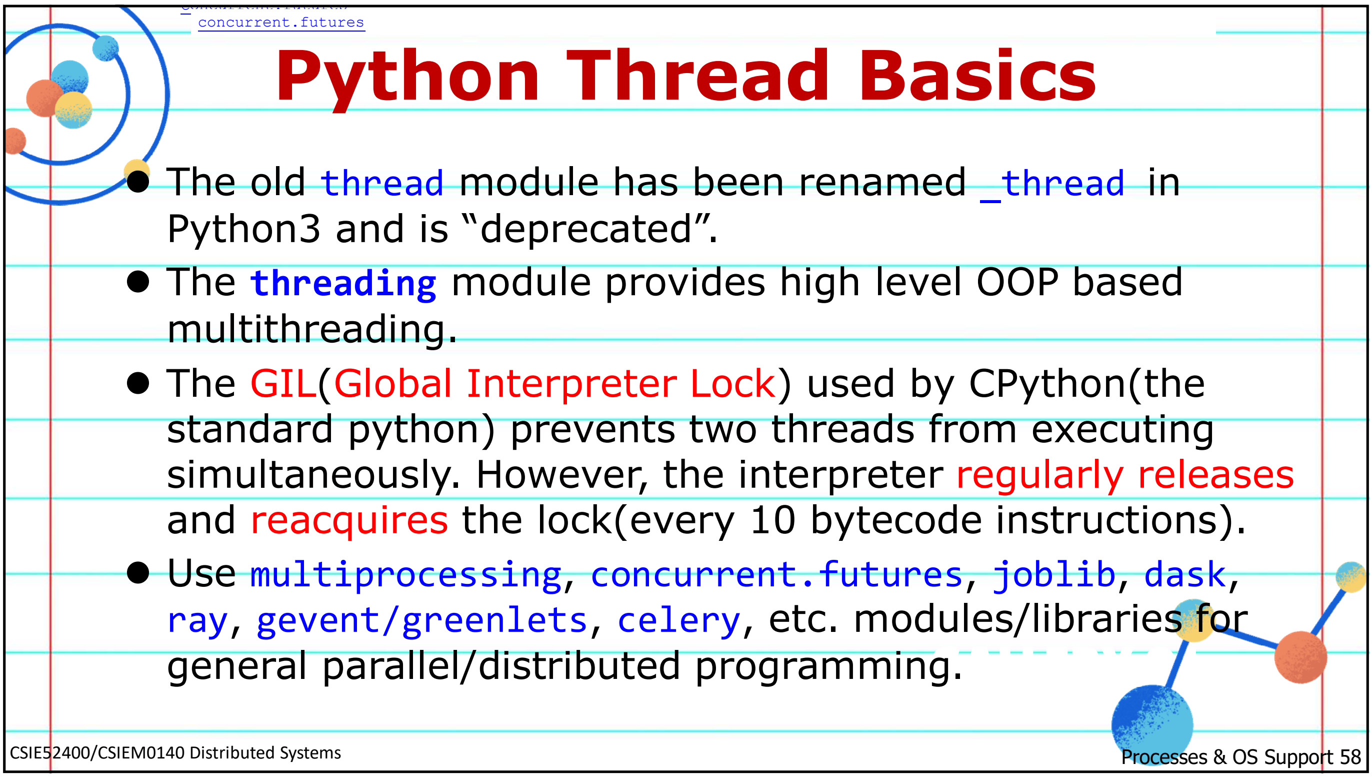


Thread Lifecycle



Python Thread Basics

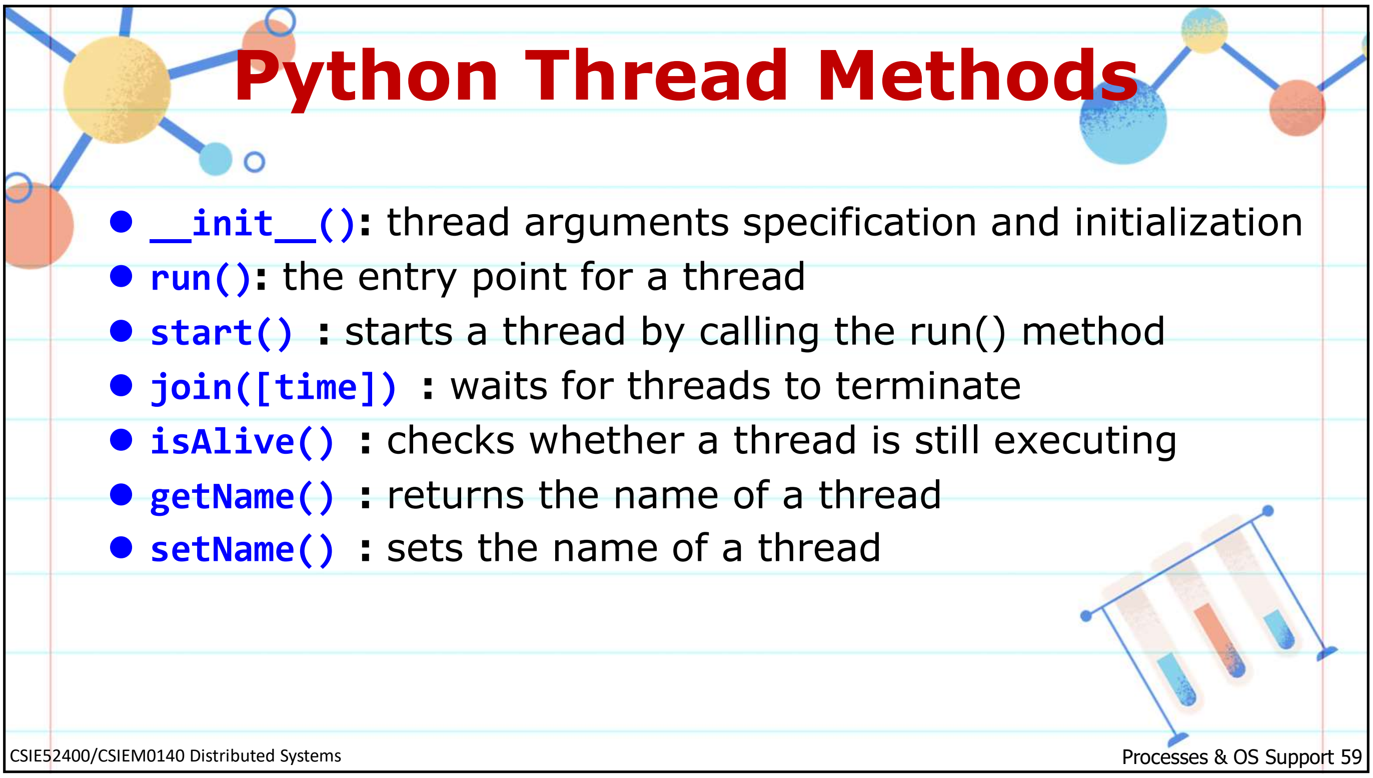
- A thread remains inactive until start method runs
  - Thread is placed in the ready queue
  - Newly started thread's run method is also activated
- A thread can lose access to the CPU:
  - Time-out (process also known as time slicing)
  - Sleep
  - Block
  - Wait
- Process of saving/restoring a thread's state is called a context switch



concurrent.futures

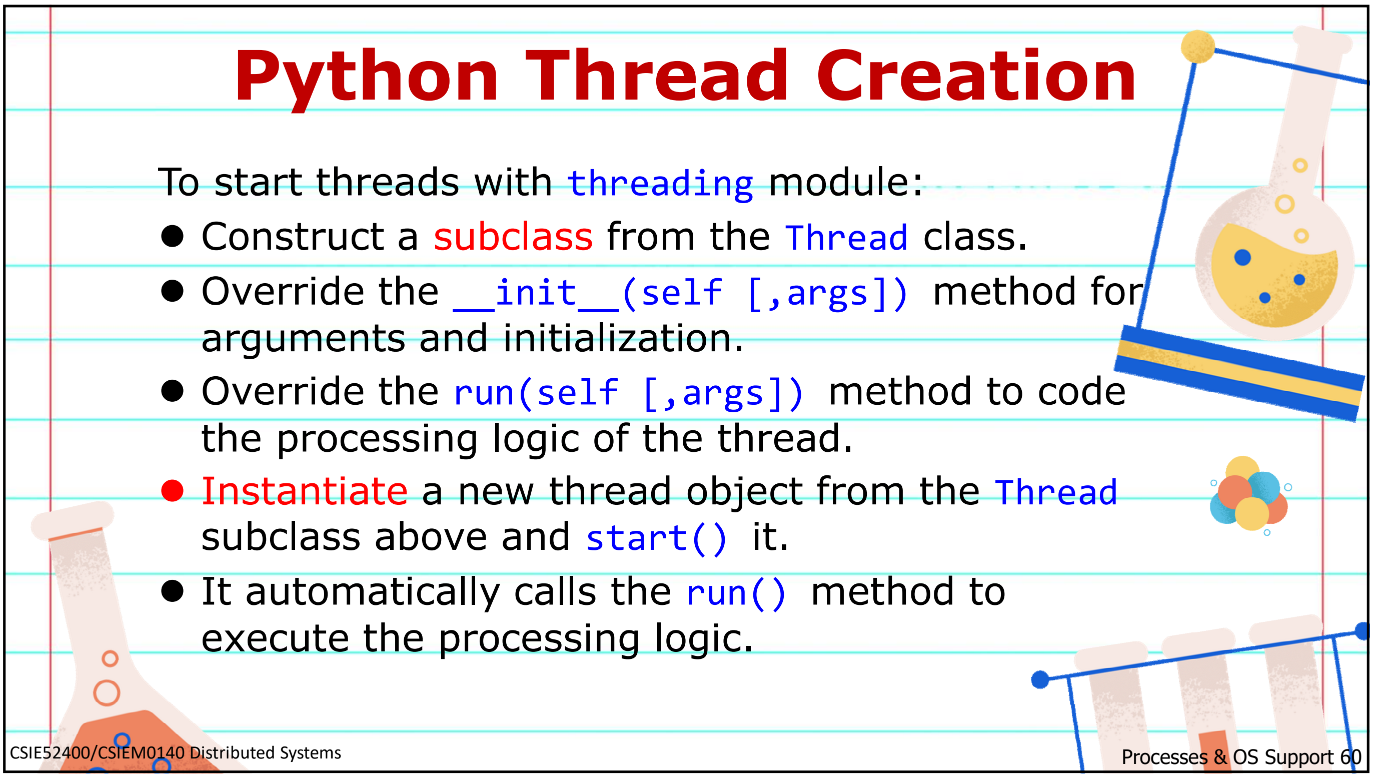
Python Thread Basics

- The old `thread` module has been renamed `_thread` in Python3 and is "deprecated".
- The **`threading`** module provides high level OOP based multithreading.
- The GIL(Global Interpreter Lock) used by CPython(the standard python) prevents two threads from executing simultaneously. However, the interpreter regularly releases and reacquires the lock(every 10 bytecode instructions).
- Use `multiprocessing`, `concurrent.futures`, `joblib`, `dask`, `ray`, `gevent/greenlets`, `celery`, etc. modules/libraries for general parallel/distributed programming.



Python Thread Methods

- `__init__()`: thread arguments specification and initialization
- `run()`: the entry point for a thread
- `start()` : starts a thread by calling the run() method
- `join([time])` : waits for threads to terminate
- `isAlive()` : checks whether a thread is still executing
- `getName()` : returns the name of a thread
- `setName()` : sets the name of a thread



Python Thread Creation

To start threads with `threading` module:

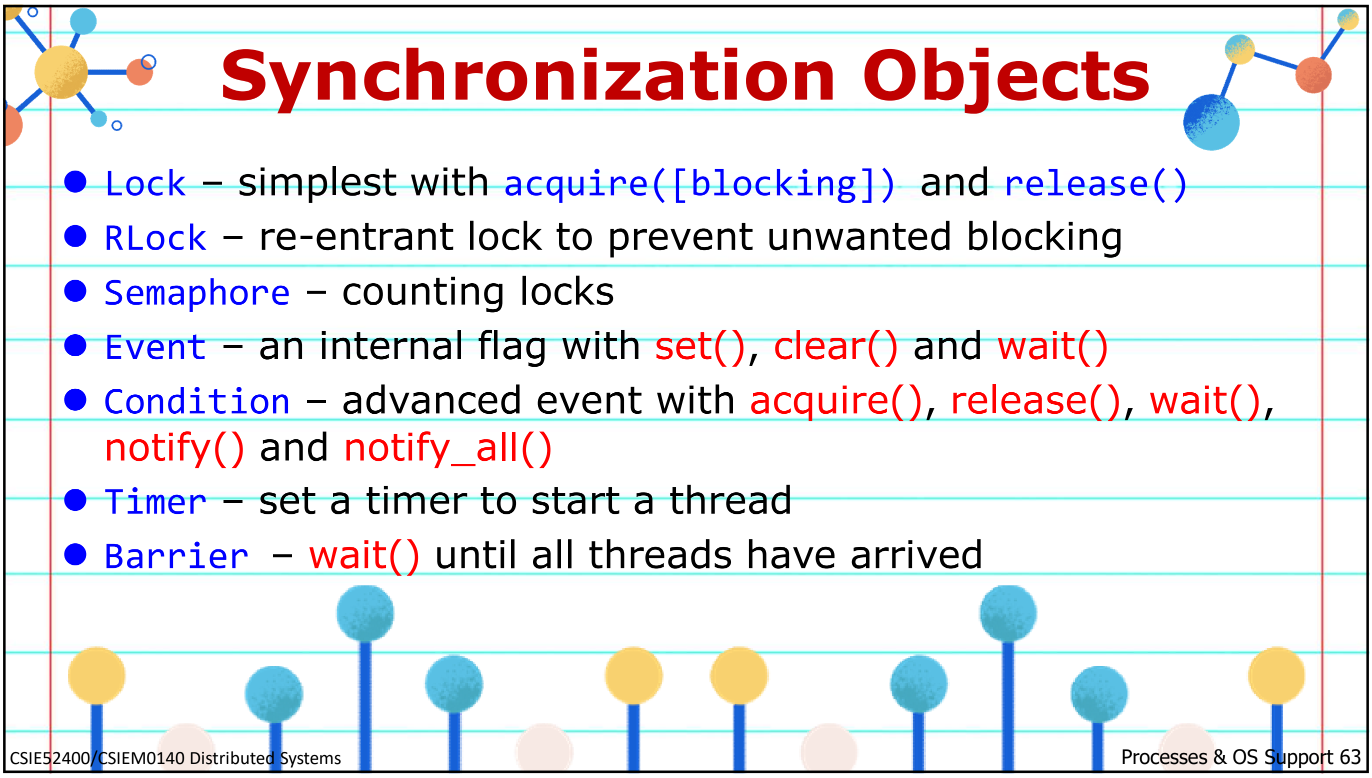
- Construct a subclass from the `Thread` class.
- Override the `__init__(self [,args])` method for arguments and initialization.
- Override the `run(self [,args])` method to code the processing logic of the thread.
- Instantiate a new thread object from the `Thread` subclass above and `start()` it.
- It automatically calls the `run()` method to execute the processing logic.

Python Thread Direct

```
import threading
import time

def loop1_10():
    for i in range(1, 11):
        time.sleep(1)
        print(i)

thrd = threading.Thread(target=loop1_10)
thrd.start()
```

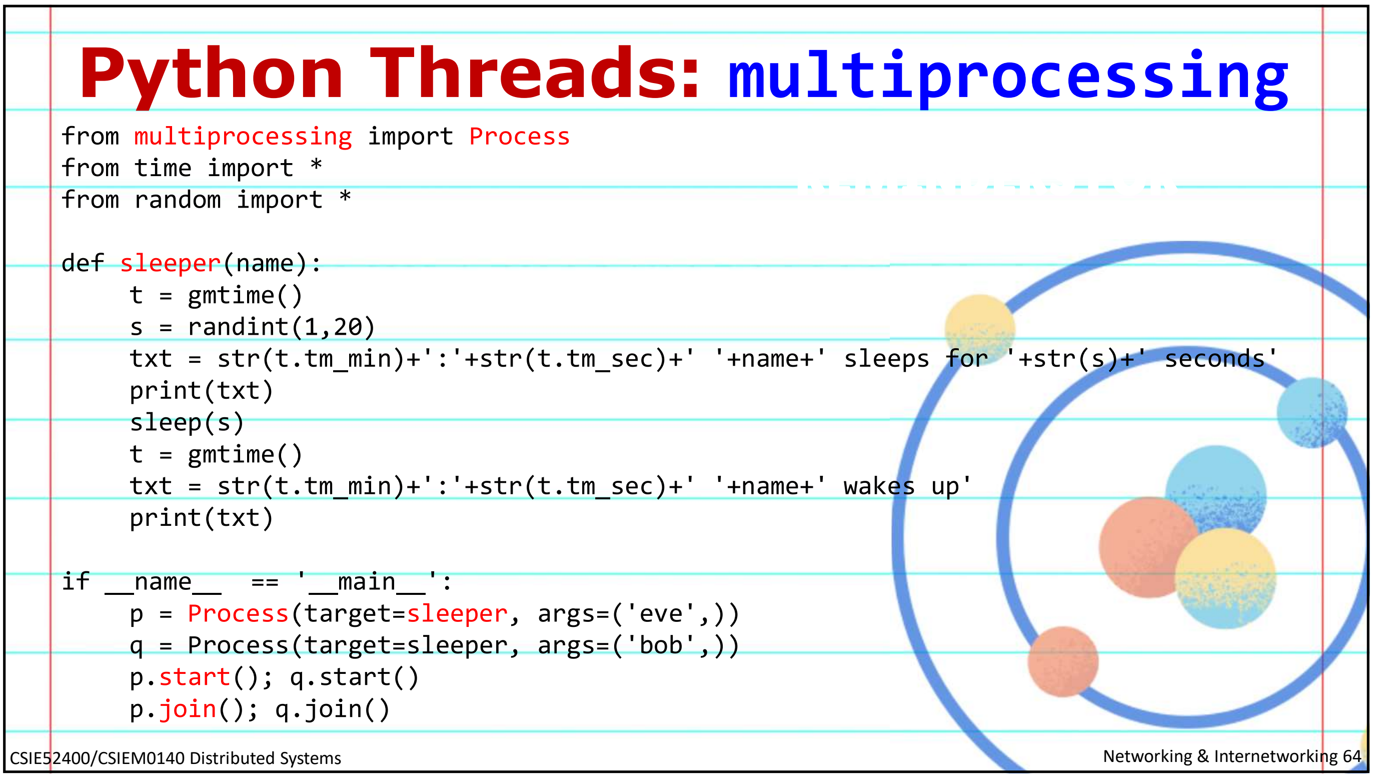
Python Thread Class/Object

```
import threading
import time

class MyThread(threading.Thread):
    def run(self):
        print(self.getName() + " started!")
        time.sleep(1)
        print(self.getName() + " finished!")

if __name__ == '__main__':
    for x in range(4):
        mythread = MyThread(name = "Thread-" + str(x + 1))
        mythread.start()
        time.sleep(.9)
```



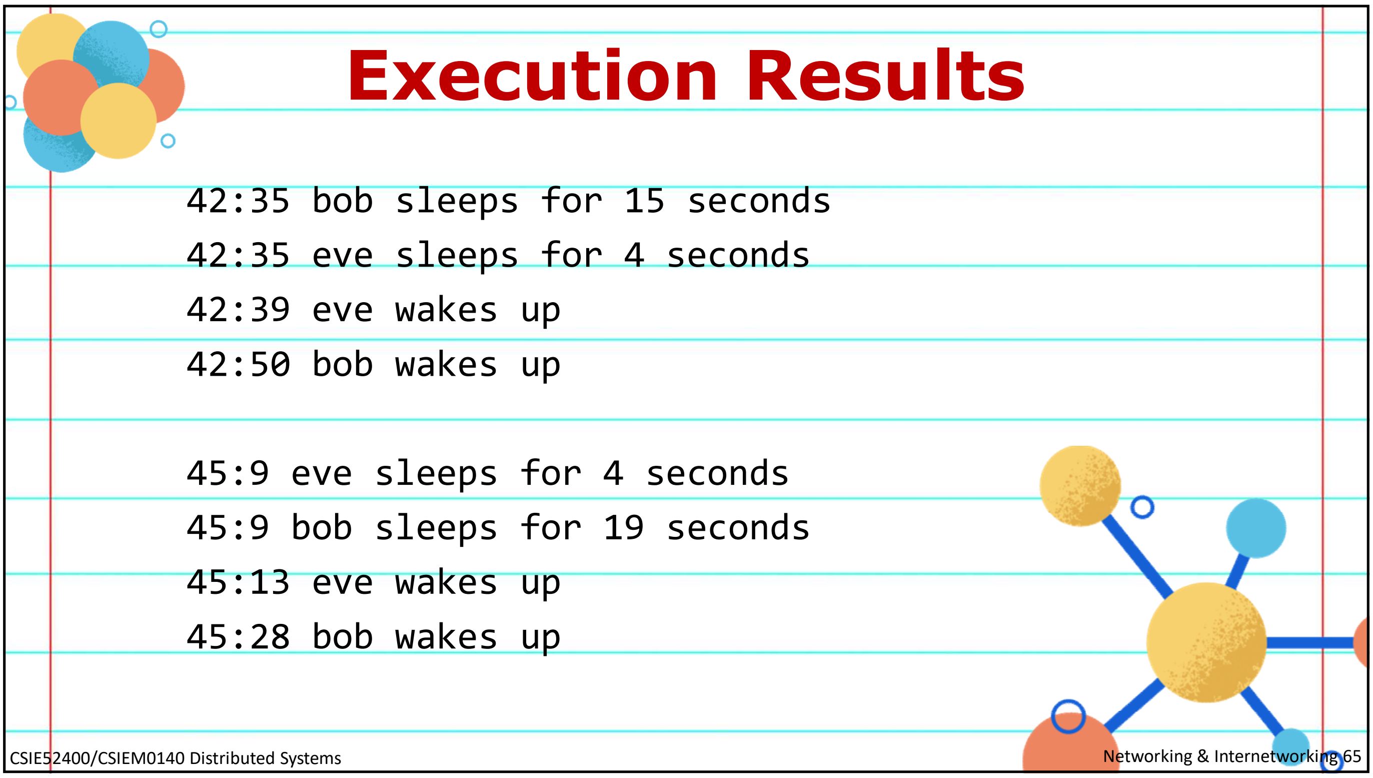
Synchronization Objects

- Lock – simplest with acquire([blocking]) and release()
- RLock – re-entrant lock to prevent unwanted blocking
- Semaphore – counting locks
- Event – an internal flag with set(), clear() and wait()
- Condition – advanced event with acquire(), release(), wait(), notify() and notify_all()
- Timer – set a timer to start a thread
- Barrier – wait() until all threads have arrived



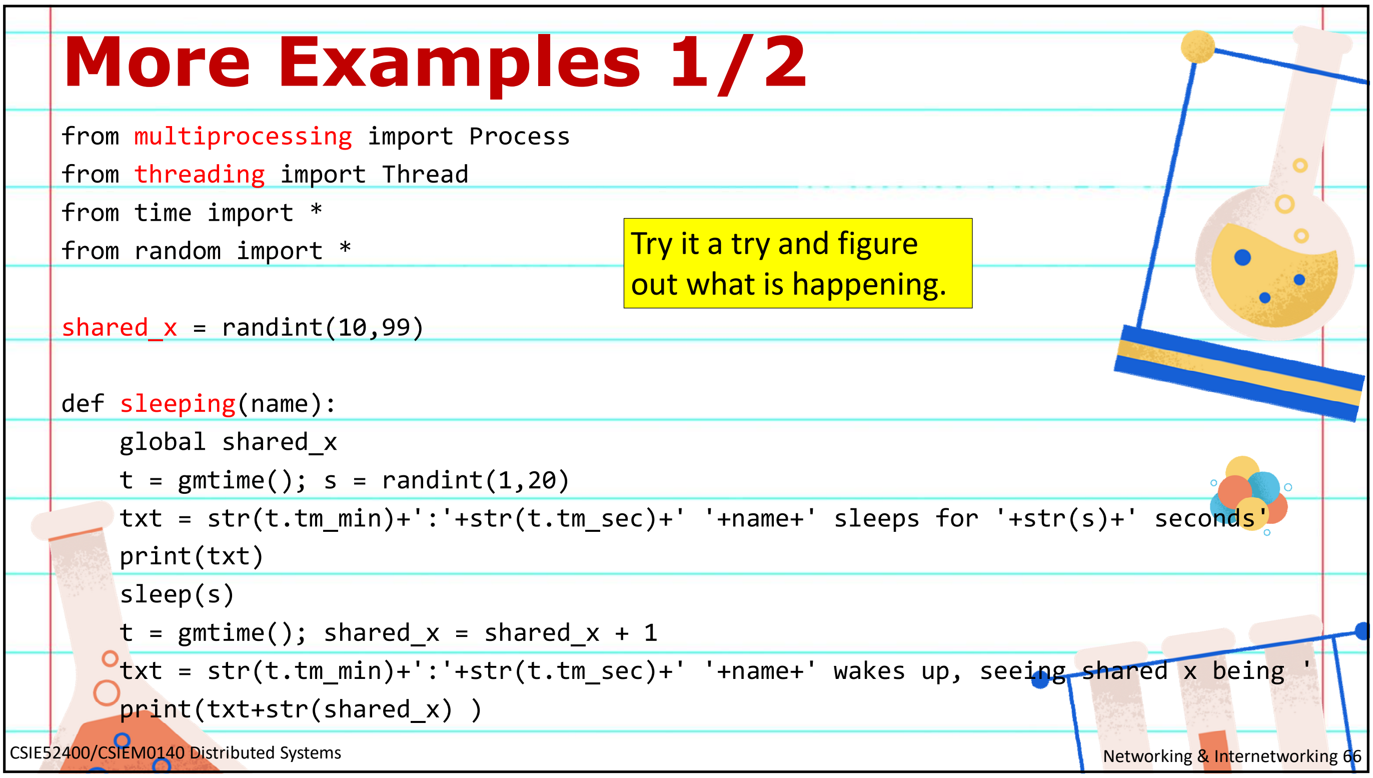
Python Threads: multiprocessing

```
from multiprocessing import Process
from time import *
from random import *

def sleeper(name):
     t = gmtime()
     s = randint(1,20)
     txt = str(t.tm_min)+':'+str(t.tm_sec)+' '+name+' sleeps for '+str(s)+' seconds'
     print(txt)
     sleep(s)
     t = gmtime()
     txt = str(t.tm_min)+':'+str(t.tm_sec)+' '+name+' wakes up'
     print(txt)

if __name__  == '__main__':
     p = Process(target=sleeper, args=('eve',))
     q = Process(target=sleeper, args=('bob',))
     p.start(); q.start()
     p.join(); q.join()
```



Execution Results

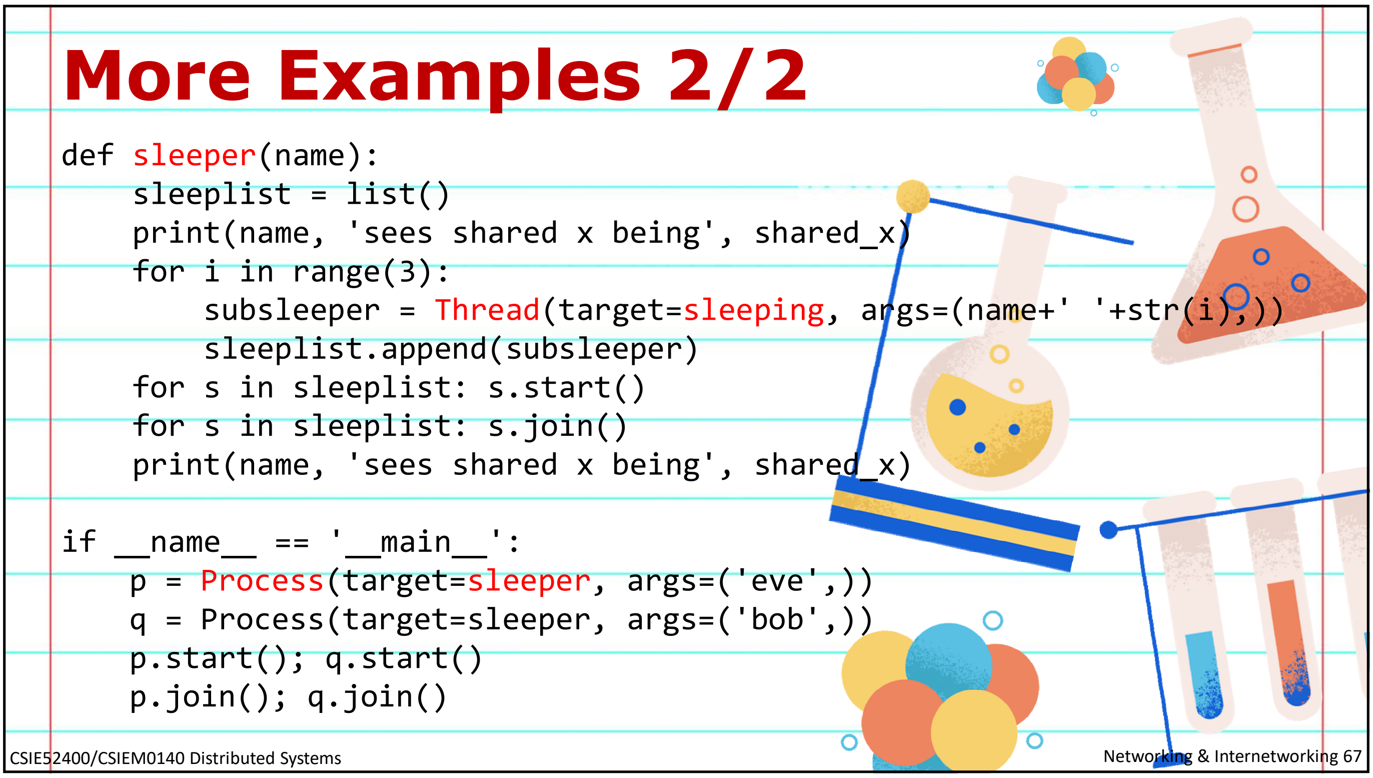
```
42:35 bob sleeps for 15 seconds
42:35 eve sleeps for 4 seconds
42:39 eve wakes up
42:50 bob wakes up

45:9 eve sleeps for 4 seconds
45:9 bob sleeps for 19 seconds
45:13 eve wakes up
45:28 bob wakes up
```



More Examples 1/2

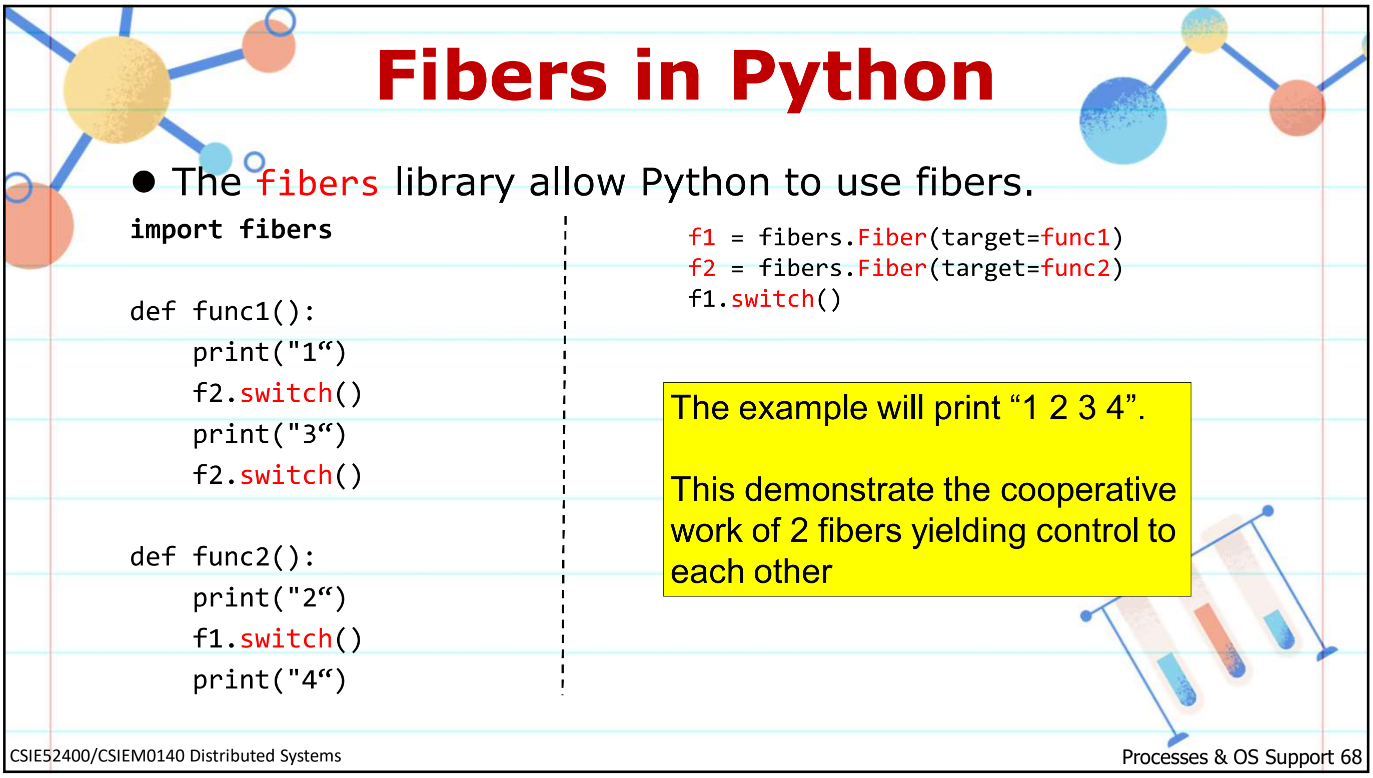
```
from multiprocessing import Process
from threading import Thread
from time import *
from random import *

shared_x = randint(10,99)

def sleeping(name):
    global shared_x
    t = gmtime(); s = randint(1,20)
    txt = str(t.tm_min)+':'+str(t.tm_sec)+' '+name+' sleeps for '+str(s)+' seconds'
    print(txt)
    sleep(s)
    t = gmtime(); shared_x = shared_x + 1
    txt = str(t.tm_min)+':'+str(t.tm_sec)+' '+name+' wakes up, seeing shared x being '
    print(txt+str(shared_x) )
```

Try it a try and figure out what is happening.



More Examples 2/2

```
def sleeper(name):
    sleeplist = list()
    print(name, 'sees shared x being', shared_x)
    for i in range(3):
        subsleeper = Thread(target=sleeping, args=(name+' '+str(i),))
        sleeplist.append(subsleeper)
    for s in sleeplist: s.start()
    for s in sleeplist: s.join()
    print(name, 'sees shared x being', shared_x)

if __name__ == '__main__':
    p = Process(target=sleeper, args=('eve',))
    q = Process(target=sleeper, args=('bob',))
    p.start(); q.start()
    p.join(); q.join()
```



Fibers in Python

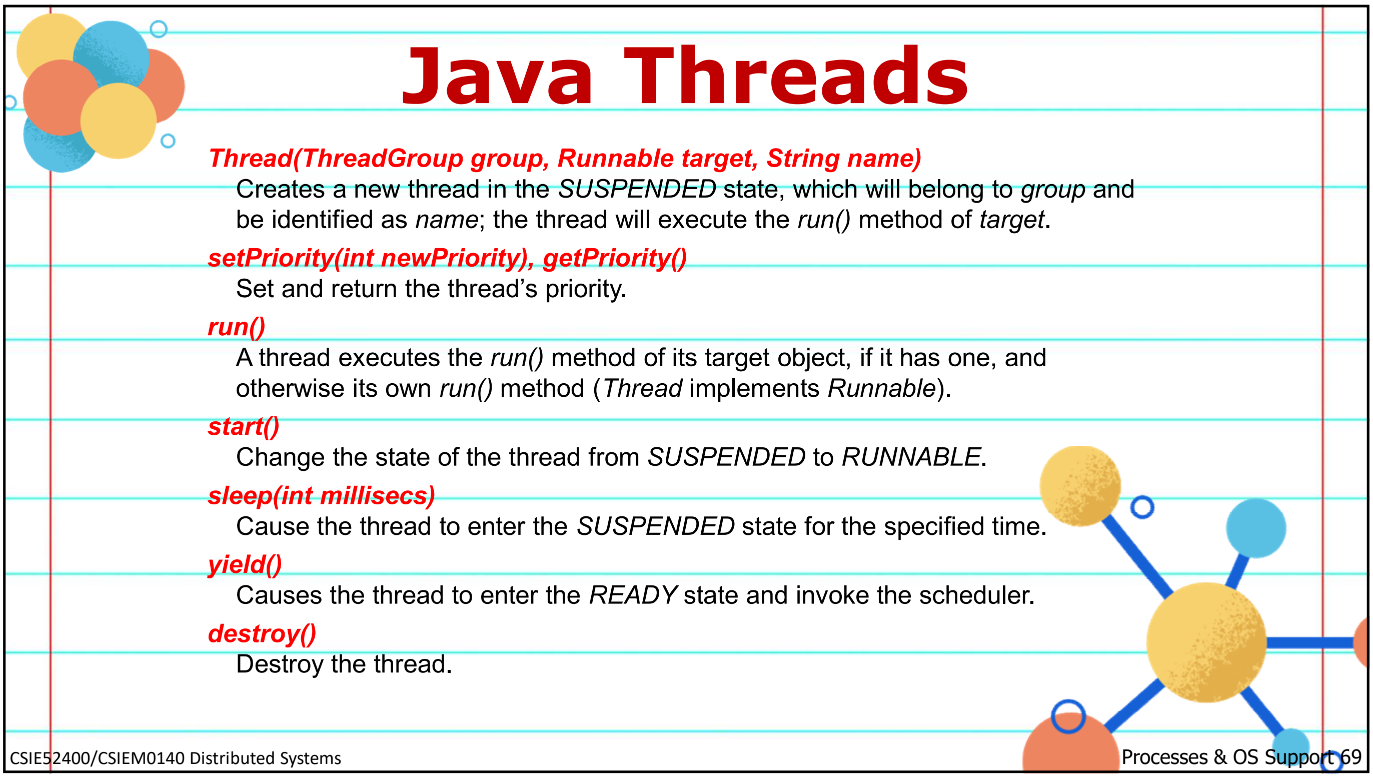
- The `fibers` library allow Python to use fibers.

```
import fibers

def func1():
    print("1“)
    f2.switch()
    print("3“)
    f2.switch()

def func2():
    print("2“)
    f1.switch()
    print("4“)
```

```
f1 = fibers.Fiber(target=func1)
f2 = fibers.Fiber(target=func2)
f1.switch()
```

The example will print “1 2 3 4”.

This demonstrate the cooperative work of 2 fibers yielding control to each other



Java Threads

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.

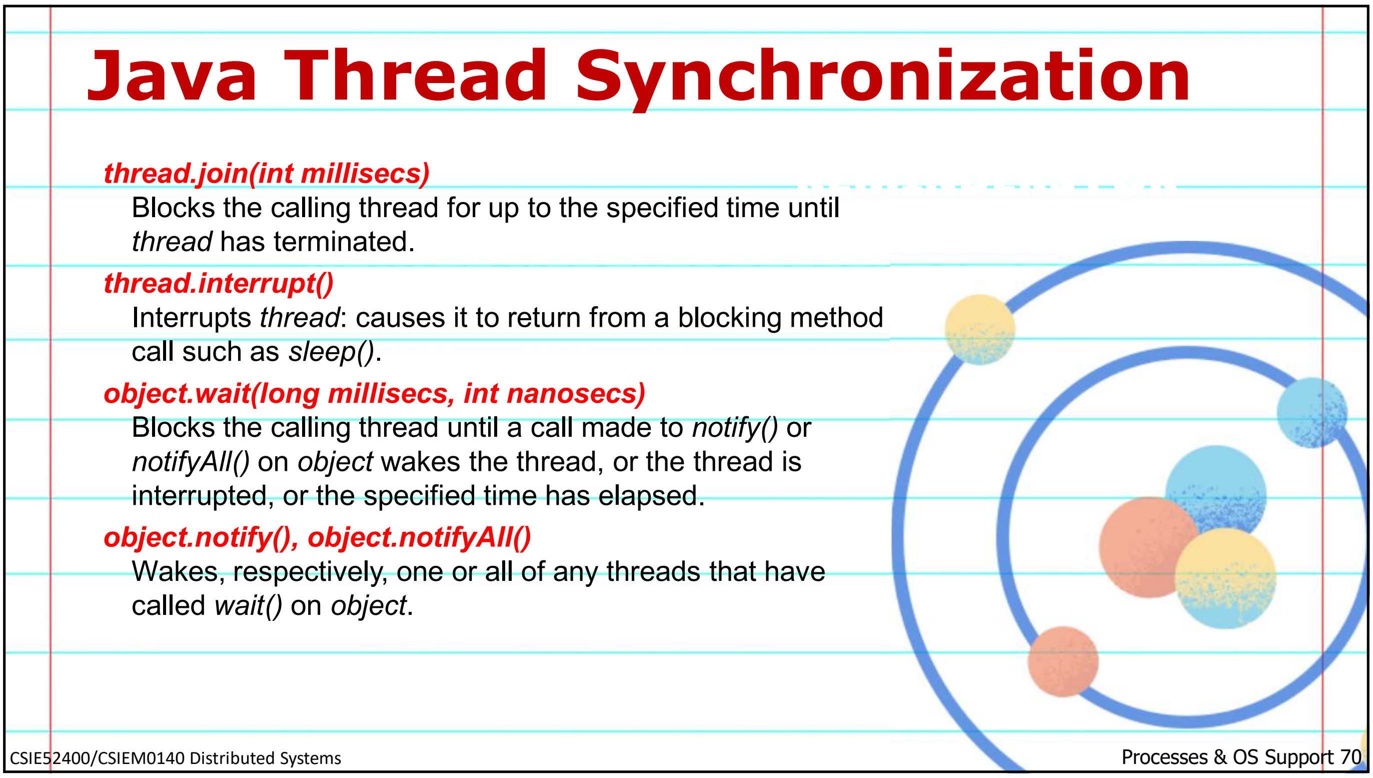
run()
A thread executes the *run()* method of its target object, if it has one, and otherwise its own *run()* method (*Thread* implements *Runnable*).

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()
Causes the thread to enter the *READY* state and invoke the scheduler.

destroy()
Destroy the thread.



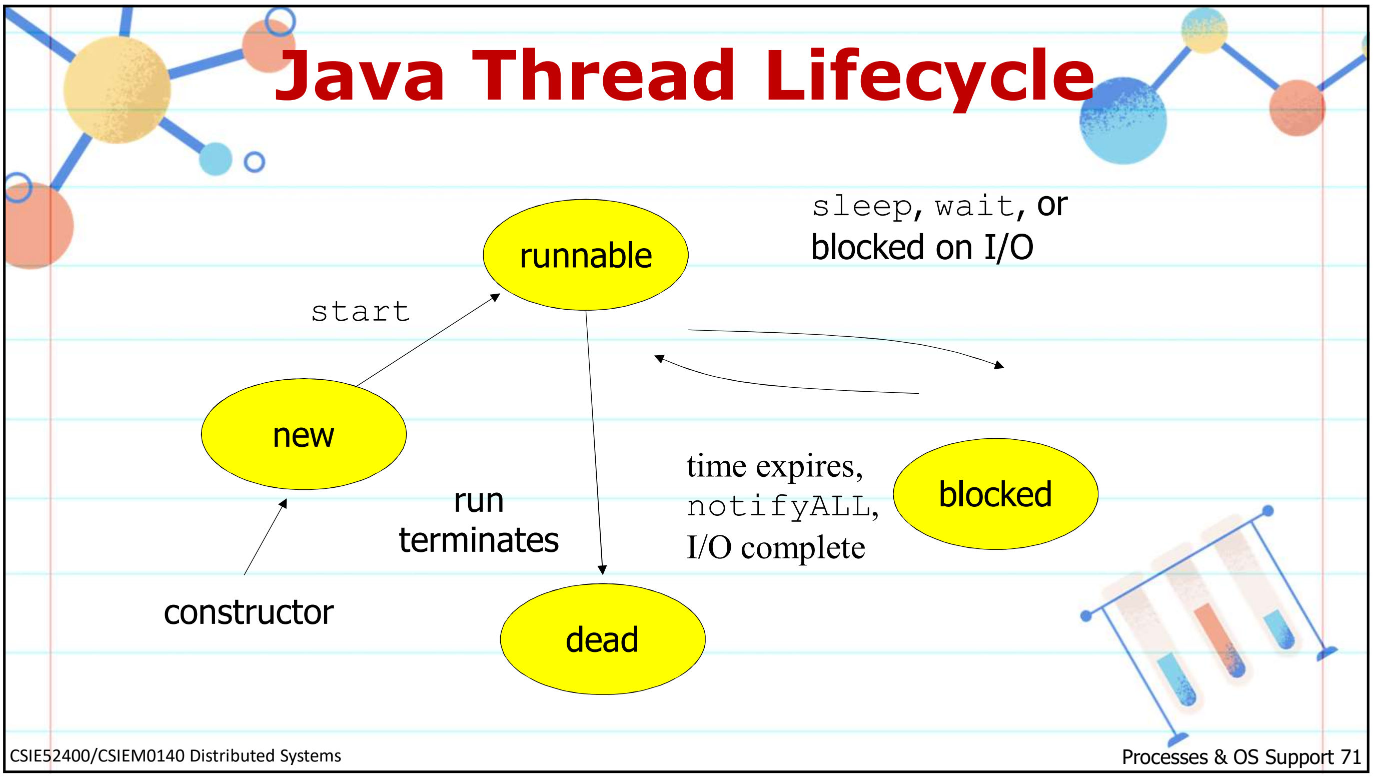
Java Thread Synchronization

thread.join(int millisecs)
Blocks the calling thread for up to the specified time until *thread* has terminated.

thread.interrupt()
Interrupts *thread*: causes it to return from a blocking method call such as *sleep()*.

object.wait(long millisecs, int nanosecs)
Blocks the calling thread until a call made to *notify()* or *notifyAll()* on *object* wakes the thread, or the thread is interrupted, or the specified time has elapsed.

object.notify(), object.notifyAll()
Wakes, respectively, one or all of any threads that have called *wait()* on *object*.



Java Thread Lifecycle

runnable

start

new

constructor

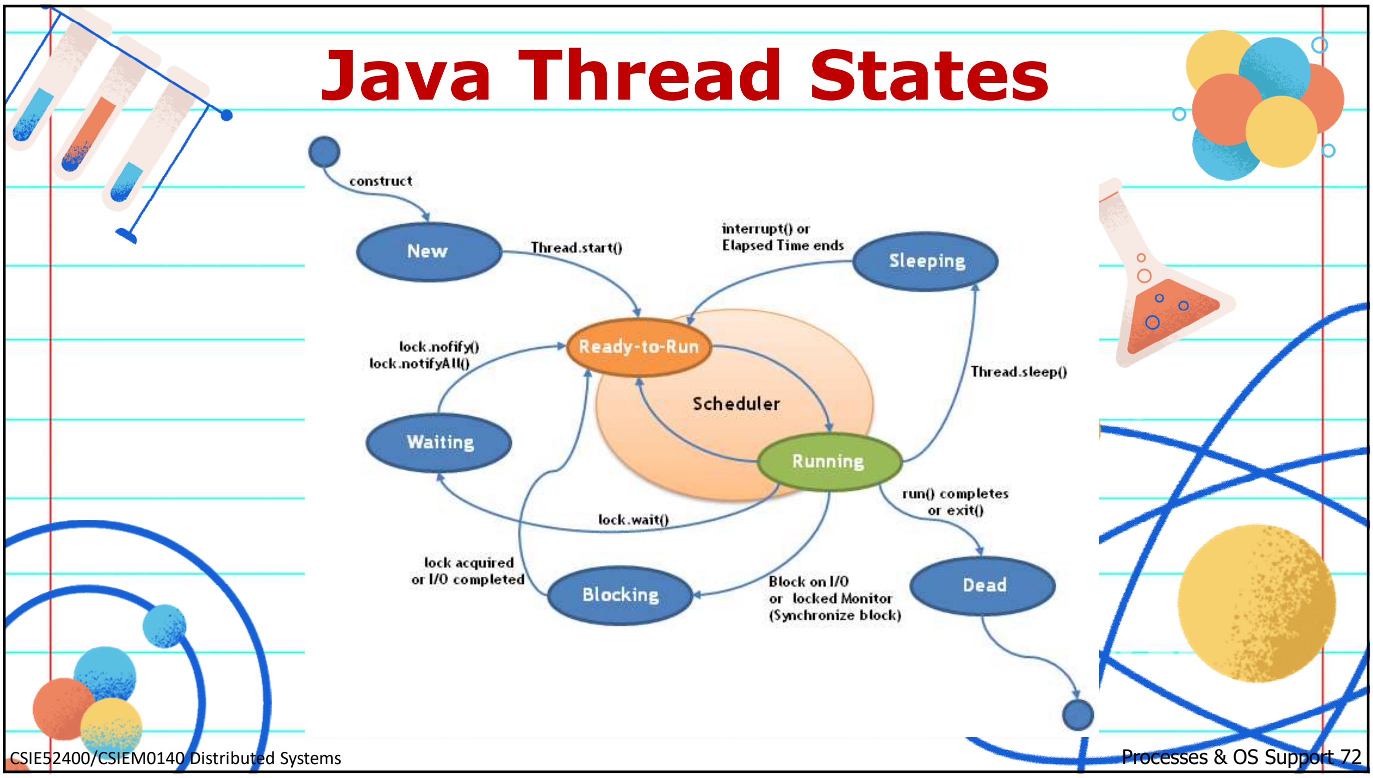
run terminates

dead

`sleep`, `wait`, or blocked on I/O

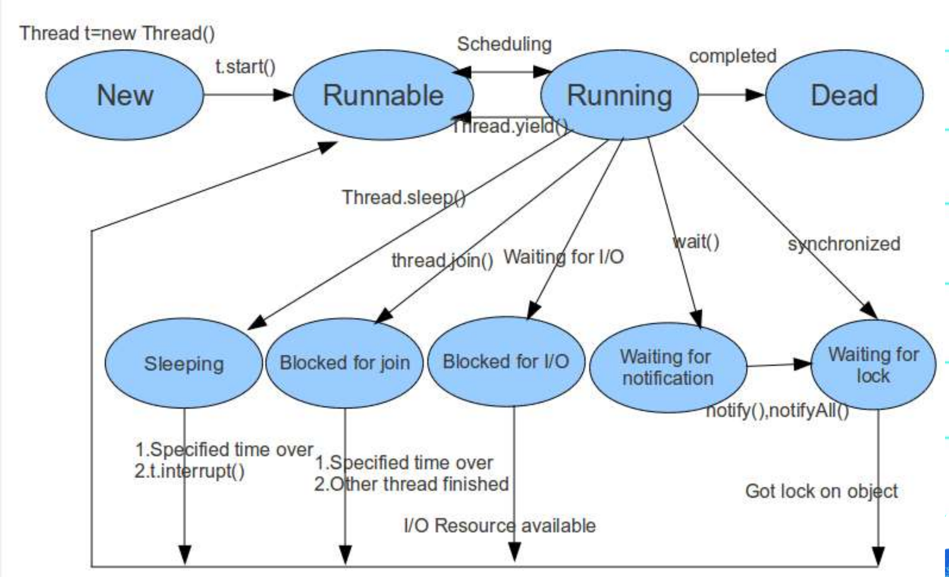
time expires, `notifyALL`, I/O complete

blocked

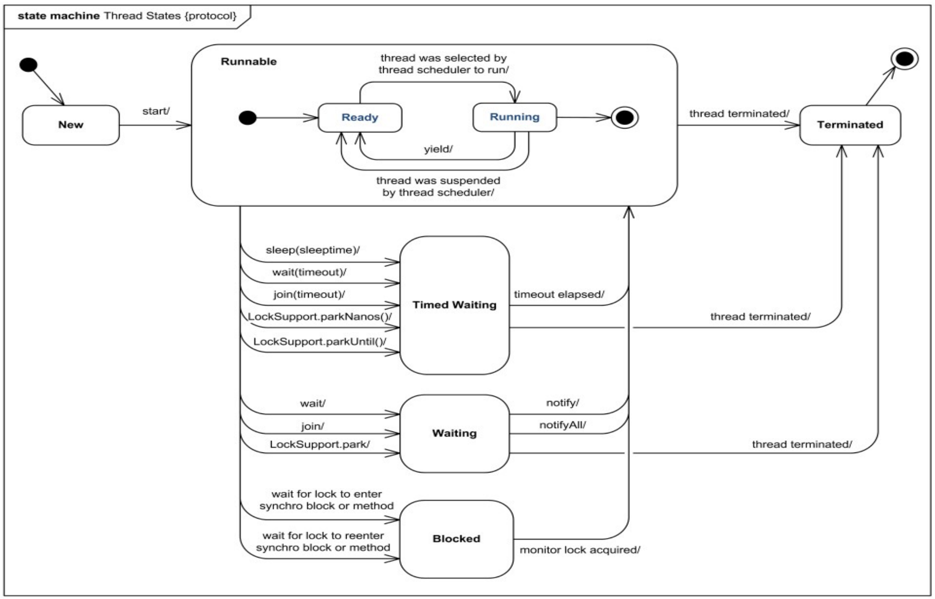


Java Thread States

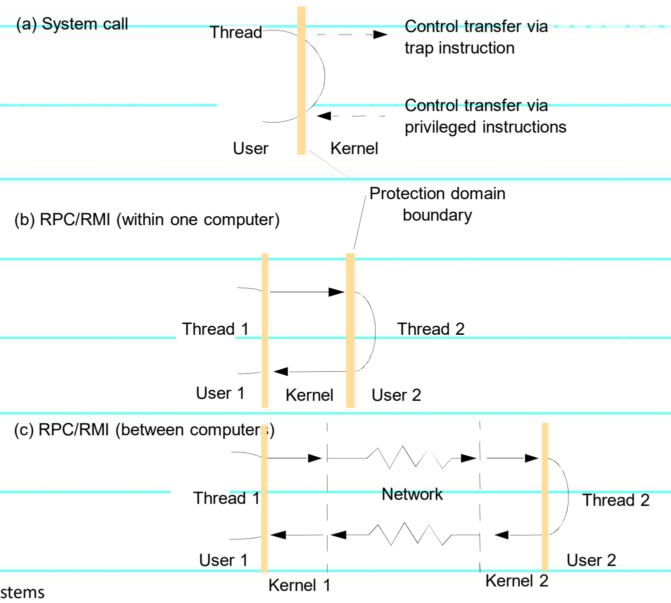
Java Thread Lifecycle



Java Thread Lifecycle



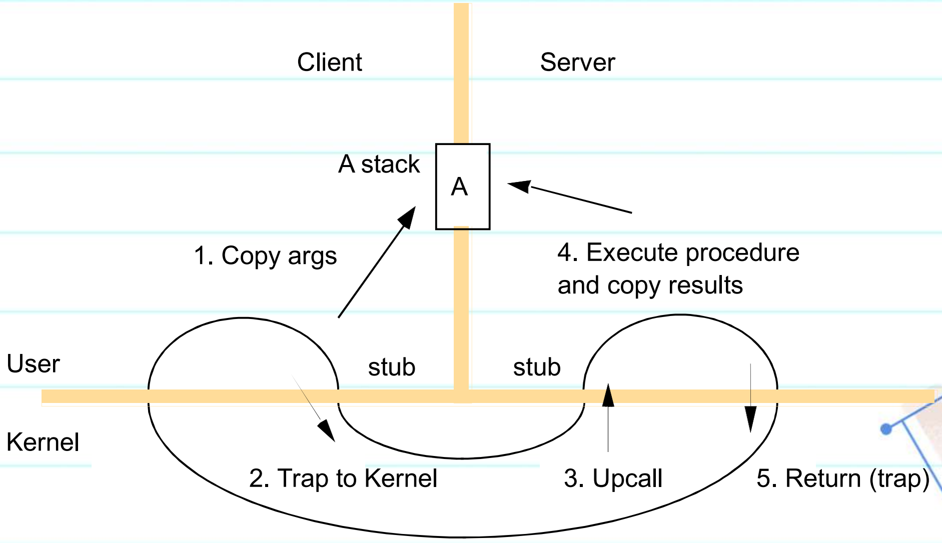
Invocations between Address Spaces


(a) System call
Thread
Control transfer via trap instruction
Control transfer via privileged instructions
User
Kernel
Protection domain boundary
(b) RPC/RMI (within one computer)
Thread 1
Thread 2
User 1
Kernel
User 2
(c) RPC/RMI (between computers)
Thread 1
Network
Thread 2
User 1
User 2
Kernel 1
Kernel 2

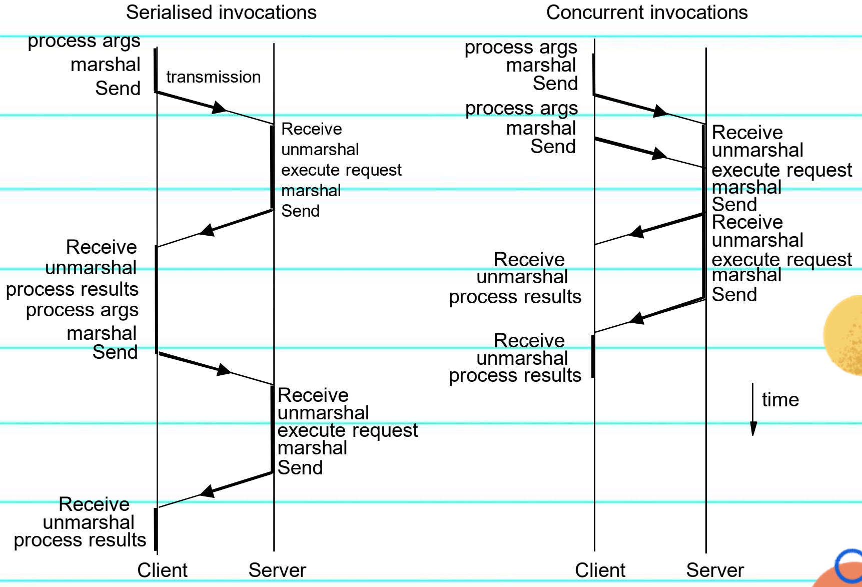
Performance Factors:

1. Synchronous vs asynchronous
2. Domain transition (across address spaces)
3. Network communication
4. Thread scheduling and switching

Lightweight Remote Procedure Call



Serialized vs Concurrent Invocations



Serialised invocations

process args
marshal
Send
transmission
Receive
unmarshal
execute request
marshal
Send
Receive
unmarshal
process results
process args
marshal
Send
Receive
unmarshal
execute request
marshal
Send
Receive
unmarshal
process results
Client
Server

Concurrent invocations

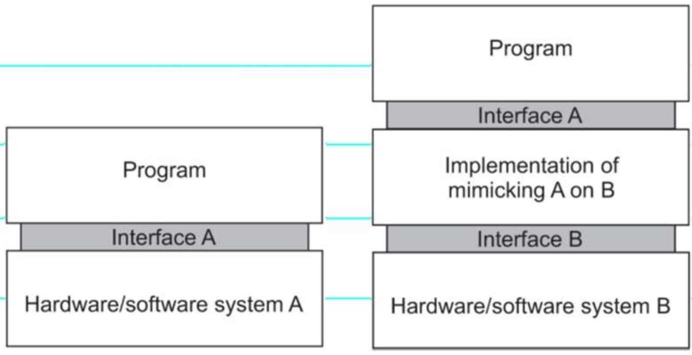
process args
marshal
Send
process args
marshal
Send
Receive
unmarshal
execute request
marshal
Send
Receive
unmarshal
execute request
marshal
Send
Receive
unmarshal
process results
Receive
unmarshal
process results
time
Client
Server

Virtualization

- Virtualization is to provide multiple virtual machines (virtual h/w images with separate OS instances) over underlying physical system.
- **Benefits**:
  - Apps can run on VMs w/o rewritten or recompiled
  - Provide convenient and customized services
  - Dynamic creation/destruction of VMs
  - Easy migration and flexible management
  - Reduce server investment and energy consumption
  - Support cloud computing

Virtualization Principles

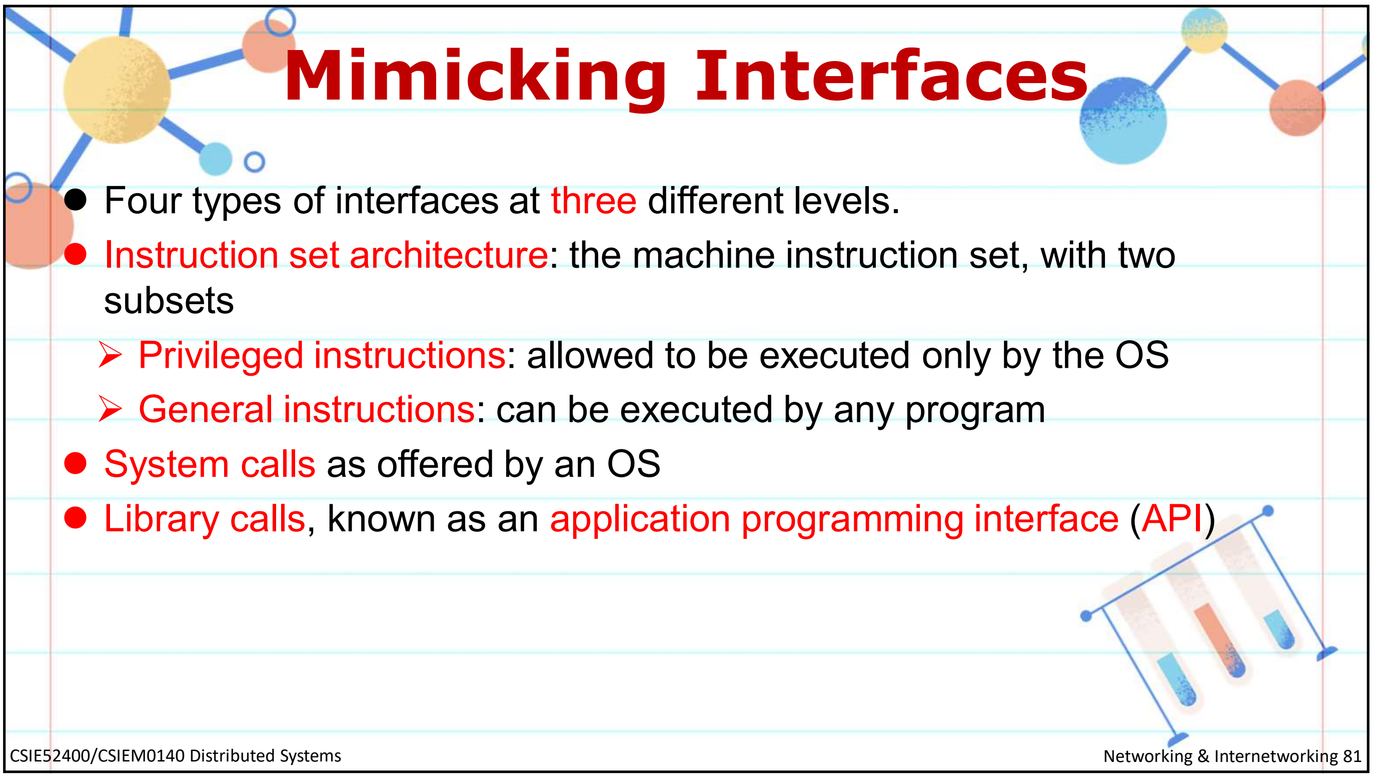
- Basic ideas:
  mimicking interface


Program
Interface A
Hardware/software system A
Program
Interface A
Implementation of mimicking A on B
Interface B
Hardware/software system B

- Virtualization is important:
  - ➢ Hardware changes faster than software
  - ➢ Ease of portability and code migration
  - ➢ Isolation of failing or attacked components

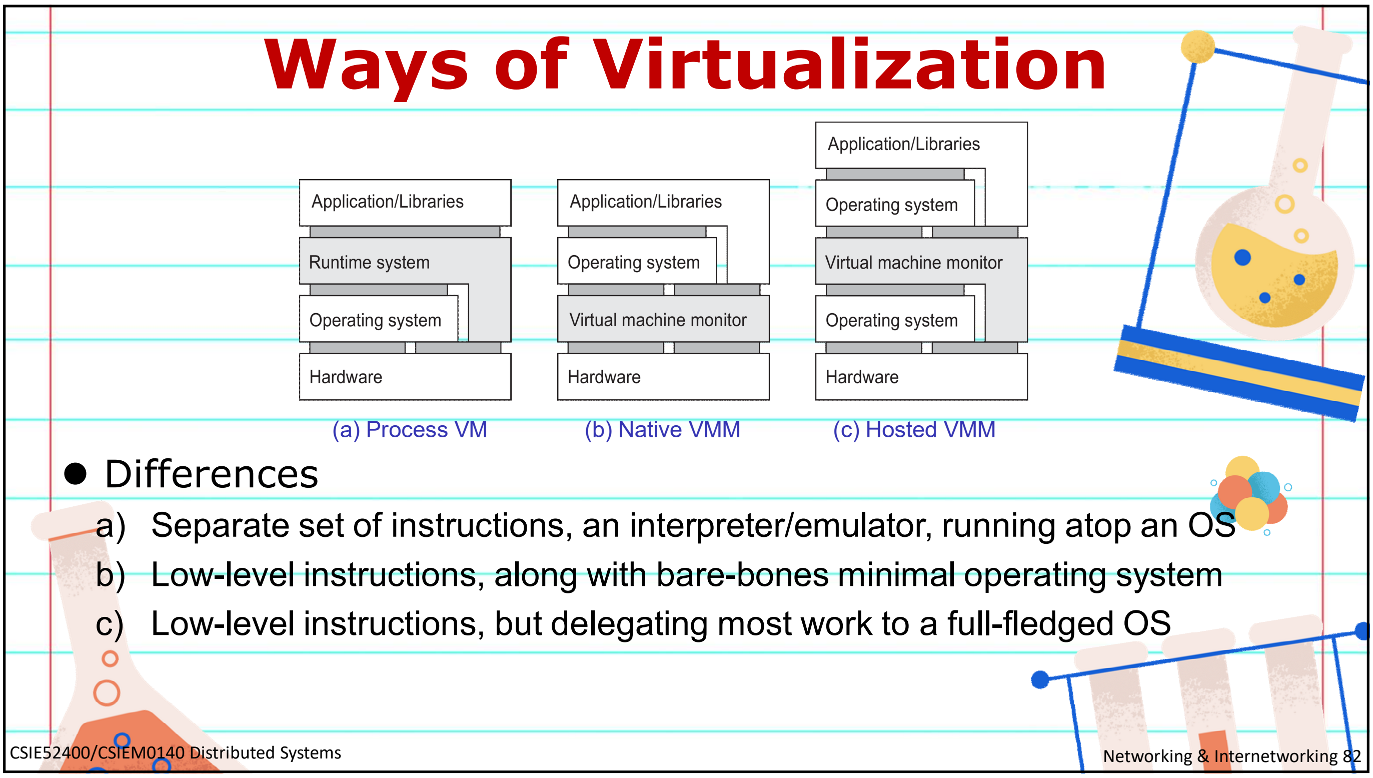
Virtualization & Cloud Computing

- Virtualization is the key to the success of CC.
- Virtualization s/w is used to run multiple Virtual Machines(VMs) on a single physical server to provide functions of multiple physical machines.
- The software is called hypervisor(or virtual machine monitor, VMM) which performs the abstraction of the hardware to the individual VMs.
- It was first invented and popularized by IBM in the 1960s for running multiple software contexts on its mainframe computers.



Mimicking Interfaces

- Four types of interfaces at three different levels.
- Instruction set architecture: the machine instruction set, with two subsets
  - ➢ Privileged instructions: allowed to be executed only by the OS
  - ➢ General instructions: can be executed by any program
- System calls as offered by an OS
- Library calls, known as an application programming interface (API)



Ways of Virtualization

Application/Libraries
Runtime system
Operating system
Hardware

(a) Process VM

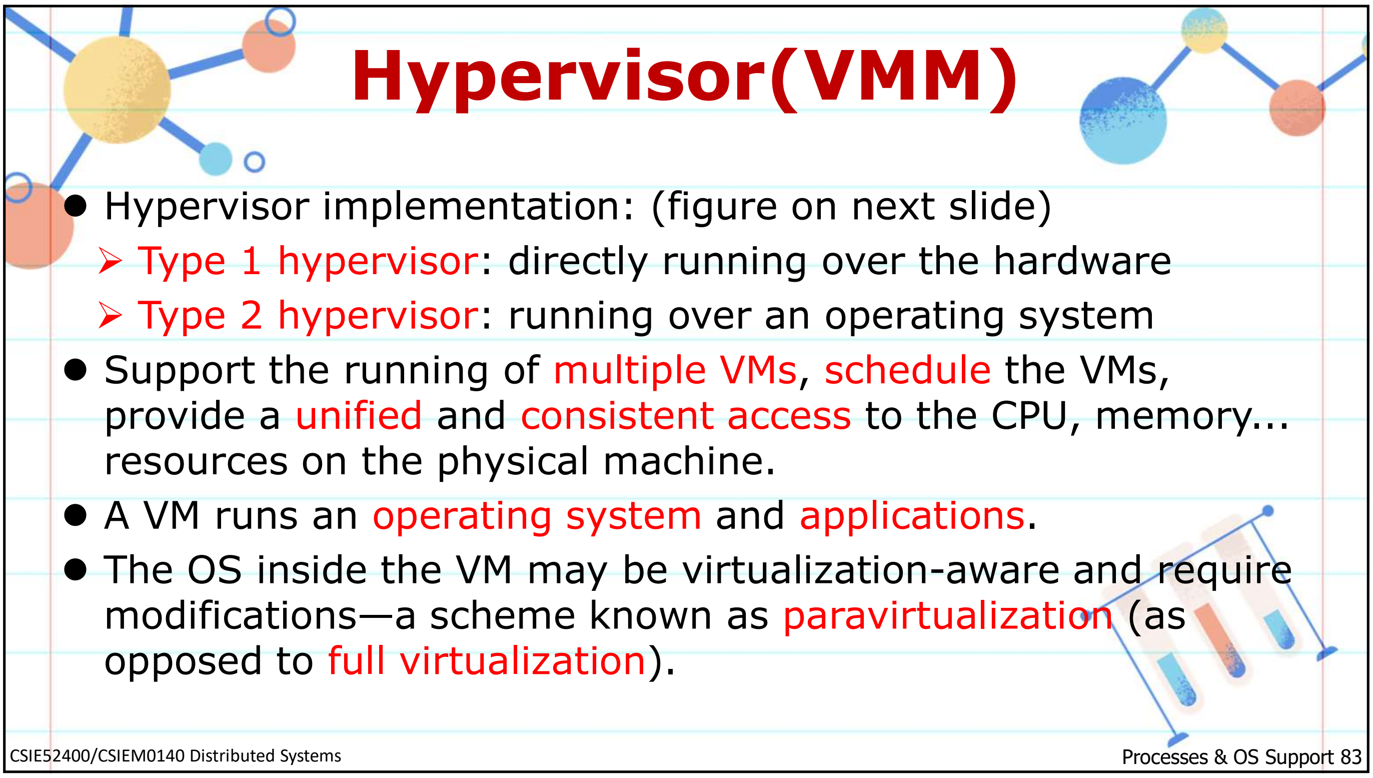
Application/Libraries
Operating system
Virtual machine monitor
Hardware

(b) Native VMM

Application/Libraries
Operating system
Virtual machine monitor
Operating system
Hardware

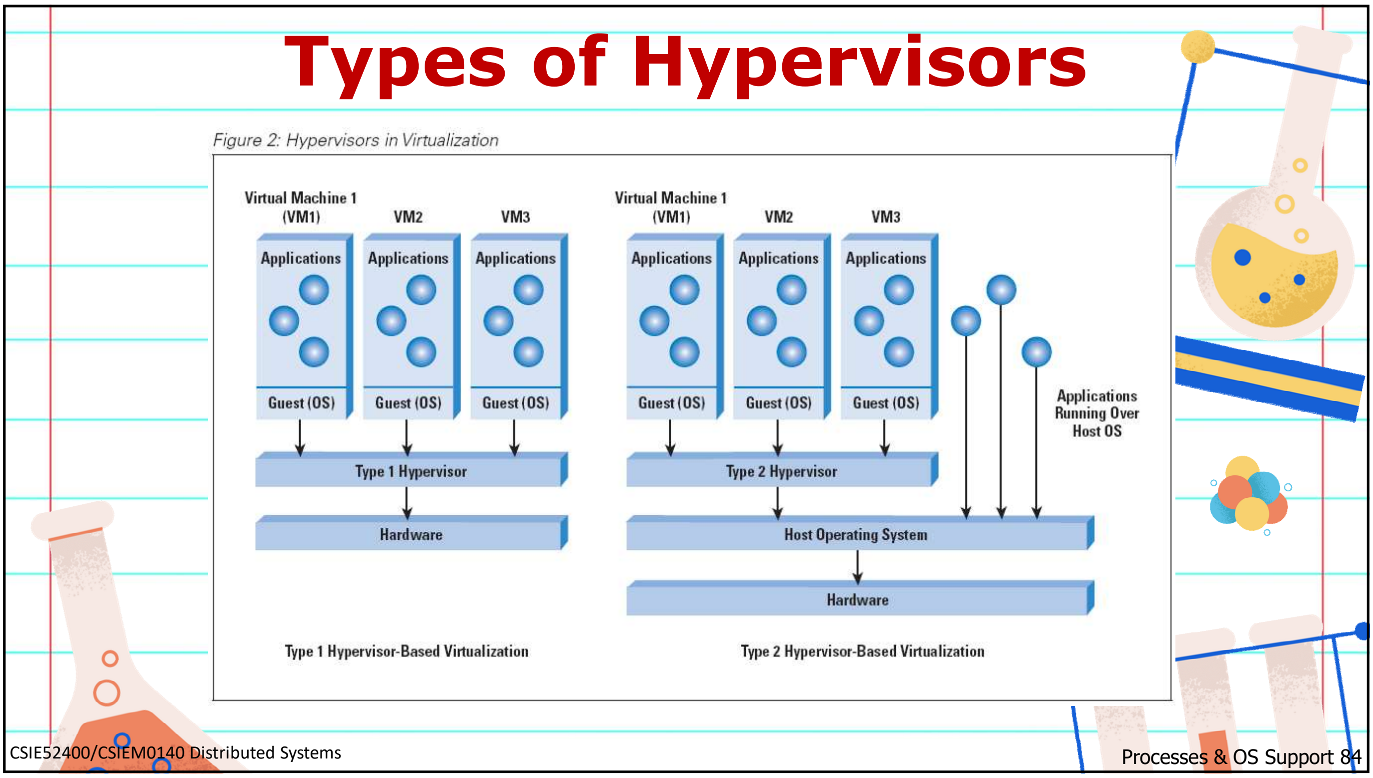
(c) Hosted VMM

- Differences
  a) Separate set of instructions, an interpreter/emulator, running atop an OS
  b) Low-level instructions, along with bare-bones minimal operating system
  c) Low-level instructions, but delegating most work to a full-fledged OS



Hypervisor(VMM)

- Hypervisor implementation: (figure on next slide)
  - Type 1 hypervisor: directly running over the hardware
  - Type 2 hypervisor: running over an operating system
- Support the running of multiple VMs, schedule the VMs, provide a unified and consistent access to the CPU, memory... resources on the physical machine.
- A VM runs an operating system and applications.
- The OS inside the VM may be virtualization-aware and require modifications—a scheme known as paravirtualization (as opposed to full virtualization).

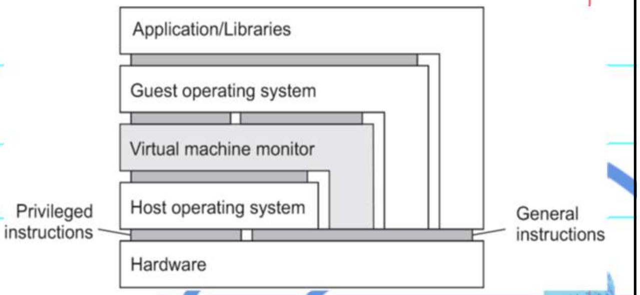


Types of Hypervisors

Figure 2: Hypervisors in Virtualization

Zooming into VMs

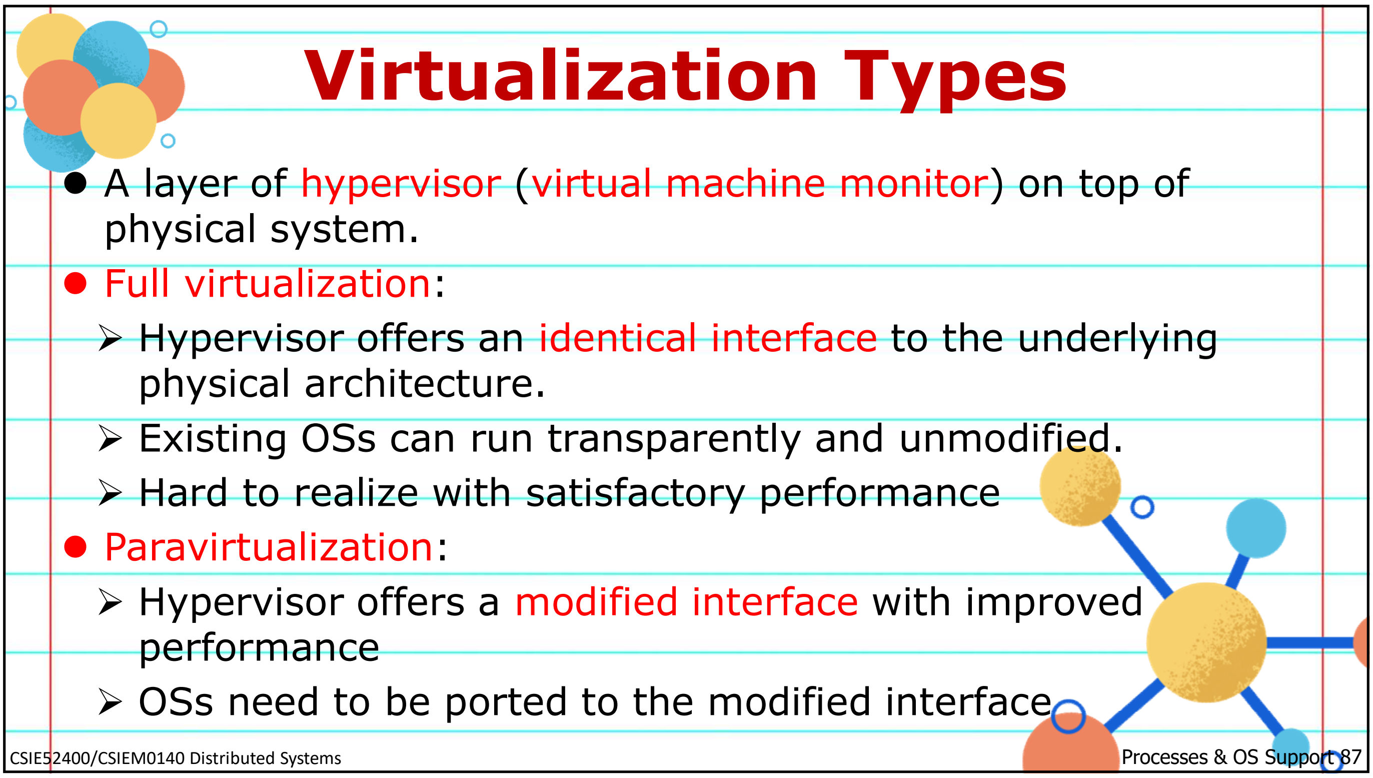
- Privileged instruction: if and only if executed in user mode, it causes a trap to the operating system
- General(Nonpriviliged) instruction: the rest



- **Special instructions**:
  - Control-sensitive instruction: may affect configuration of a machine (e.g., one affecting relocation register or interrupt table)
  - Behavior-sensitive instruction: effect is partially determined by context (e.g., POPF sets an interrupt-enabled flag, but only in system mode)

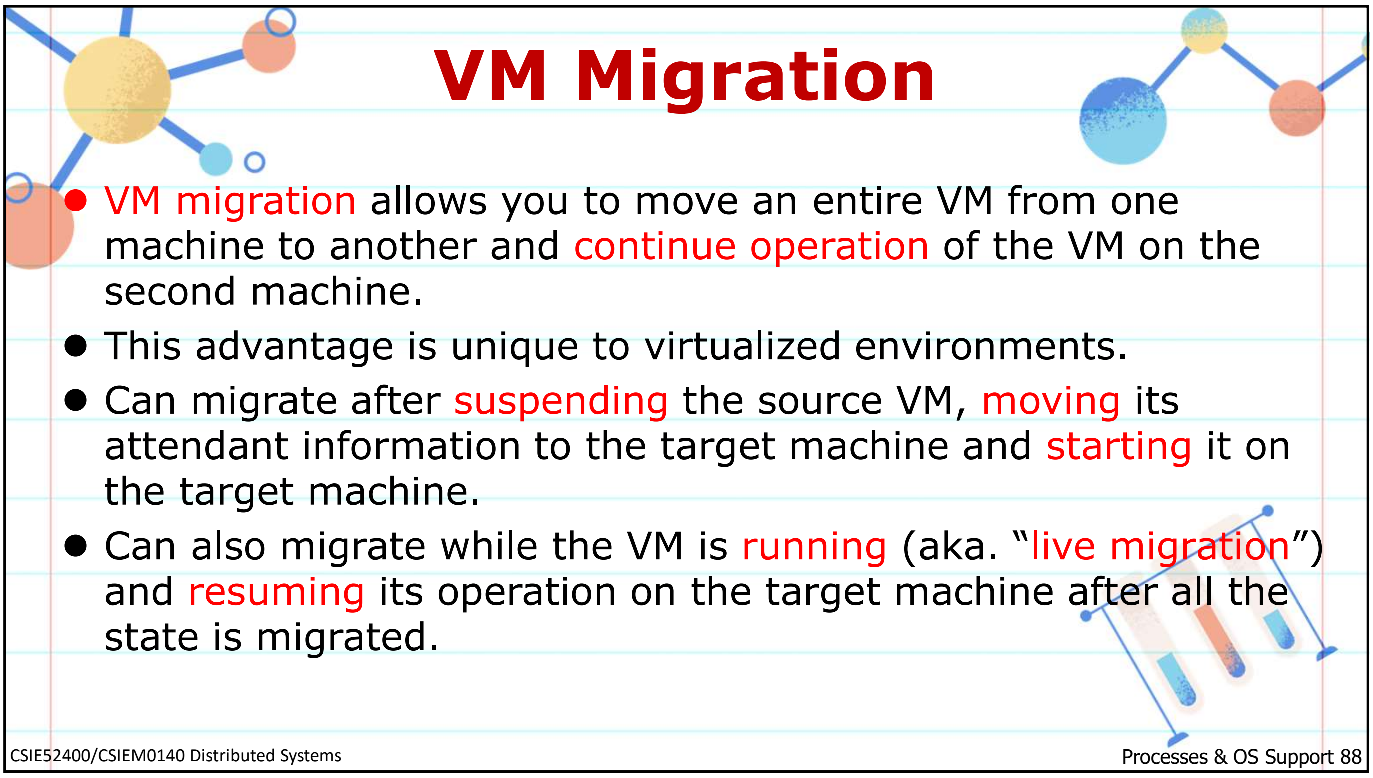
Condition for Virtualization

- Necessary condition: For any conventional computer, a VMM may be constructed if the set of sensitive instructions for that computer is a subset of privileged instructions.
- **Problem**: condition is not always satisfied. There may be sensitive instructions that are executed in user mode without causing a trap to the OS.
- **Solutions**:
  - Emulate all instructions
  - Wrap nonprivileged sensitive instructions to divert control to VMM
  - Paravirtualization: modify guest OS, either by preventing nonprivileged sensitive instructions, or making them nonsensitive (i.e., changing the context).



Virtualization Types

- A layer of hypervisor (virtual machine monitor) on top of physical system.
- Full virtualization:
  - Hypervisor offers an identical interface to the underlying physical architecture.
  - Existing OSs can run transparently and unmodified.
  - Hard to realize with satisfactory performance
- Paravirtualization:
  - Hypervisor offers a modified interface with improved performance
  - OSs need to be ported to the modified interface



VM Migration

- VM migration allows you to move an entire VM from one machine to another and continue operation of the VM on the second machine.
- This advantage is unique to virtualized environments.
- Can migrate after suspending the source VM, moving its attendant information to the target machine and starting it on the target machine.
- Can also migrate while the VM is running (aka. “live migration”) and resuming its operation on the target machine after all the state is migrated.

Benefits of Virtualization

- Elasticity and scalability: Firing up and shutting down VMs involves less effort as opposed to bringing servers up or down.
- Workload migration: Can carry out workload migration with much less effort as compared to migration across physical servers at different locations.
- Resiliency: Can isolate physical-server failure from user services through migration of VMs.

Virtualization and Cloud

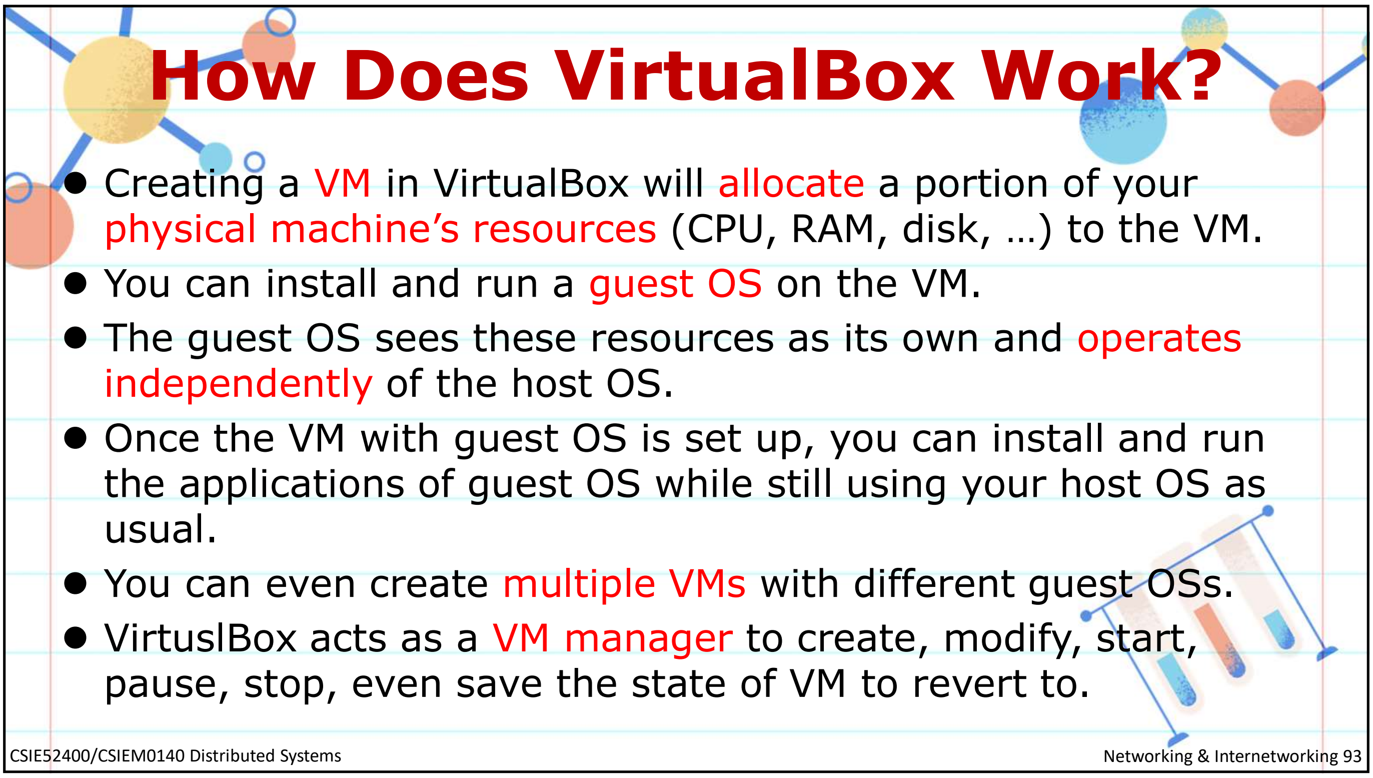
- Virtualization is not a prerequisite for cloud computing.
- However, virtualization provides a valuable toolkit and enables significant flexibility in cloud-computing deployments.
- Therefore, it is almost adopted by all cloud platforms.

Virtualization Software

- Popular virtualization software:
  - VMware (VMware Inc.)
  - VirtualBox (Oracle)
  - Hyper-V (Microsoft)
  - QEMU (open source machine emulator & virtualizer)
  - Xen (open source project)

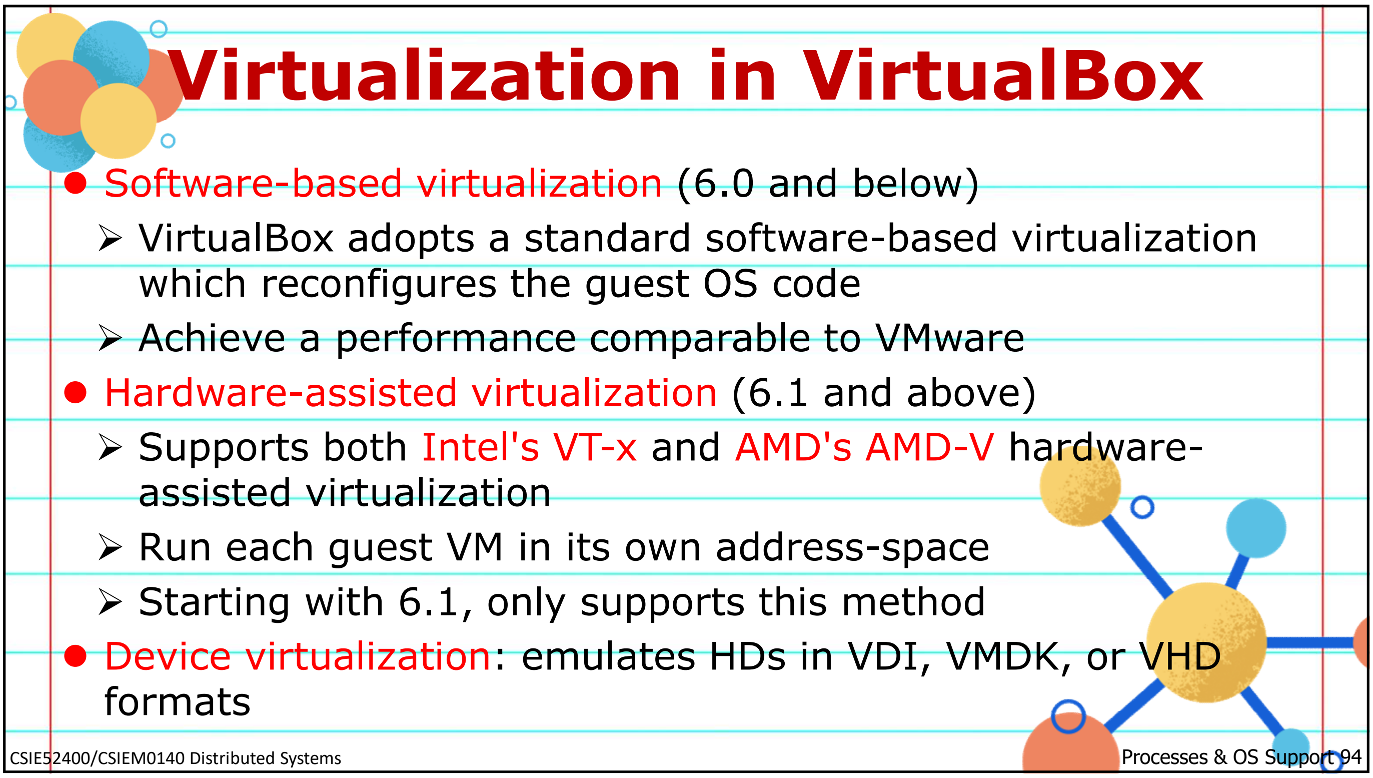
VirtualBox – A Free VM S/W

- VMware workstation/player needs license fee
- VirtualBox is open-source and well maintained by Oracle
- A type-2 hypervisor(i.e. runs on the host operating), very easy to install and use
- Users can load multiple guest OSes under a single host OS
- VirtualBox supports many Host OSes: Windows, Linux, macOS, Solaris, …
- VirtualBox can be accessed from https://www.virtualbox.org/
- Give it a try.



How Does VirtualBox Work?

- Creating a VM in VirtualBox will allocate a portion of your physical machine's resources (CPU, RAM, disk, …) to the VM.
- You can install and run a guest OS on the VM.
- The guest OS sees these resources as its own and operates independently of the host OS.
- Once the VM with guest OS is set up, you can install and run the applications of guest OS while still using your host OS as usual.
- You can even create multiple VMs with different guest OSs.
- VirtuslBox acts as a VM manager to create, modify, start, pause, stop, even save the state of VM to revert to.



Virtualization in VirtualBox

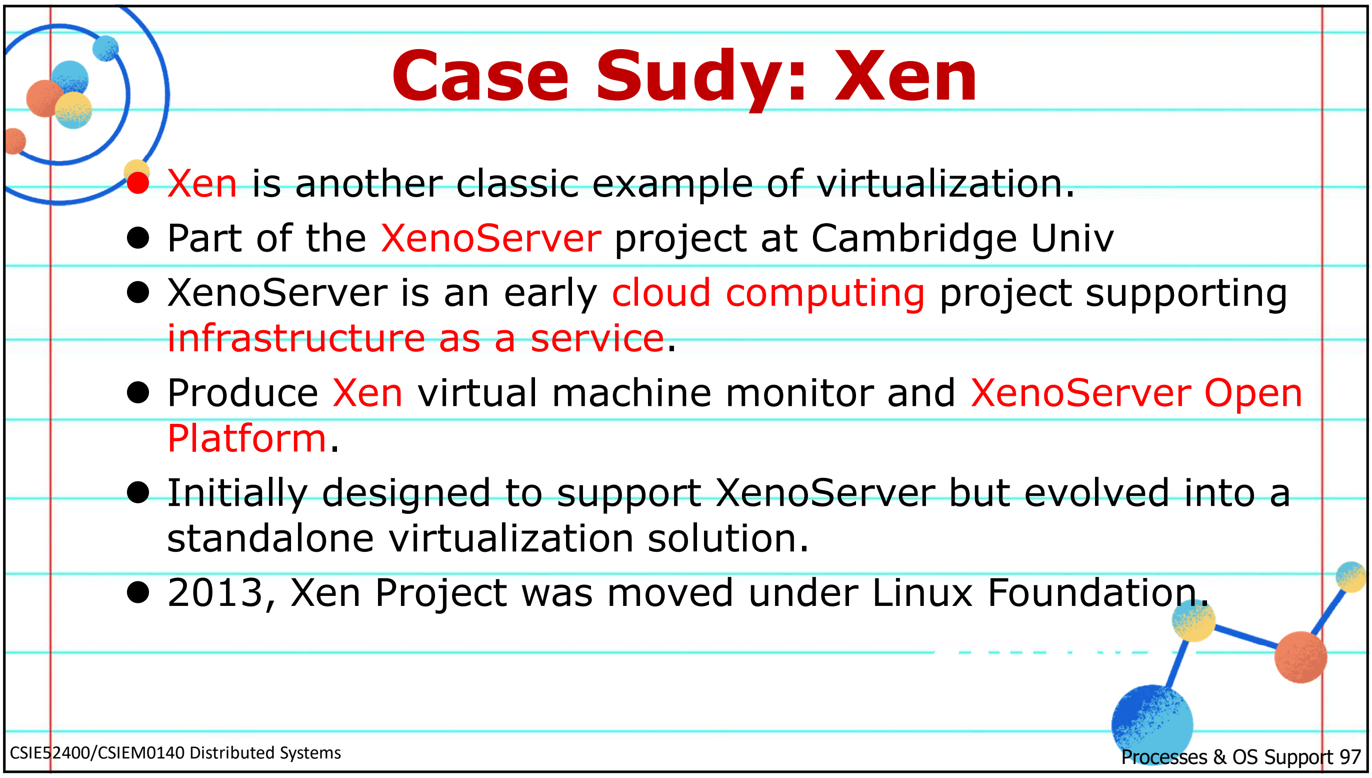
- Software-based virtualization (6.0 and below)
  - VirtualBox adopts a standard software-based virtualization which reconfigures the guest OS code
  - Achieve a performance comparable to VMware
- Hardware-assisted virtualization (6.1 and above)
  - Supports both Intel's VT-x and AMD's AMD-V hardware-assisted virtualization
  - Run each guest VM in its own address-space
  - Starting with 6.1, only supports this method
- Device virtualization: emulates HDs in VDI, VMDK, or VHD formats

Virtualization of CPU

- The primary role of hypervisor
- Popek & Goldberg(1974) divided CPU instructions that can change the machine state into
  - Control-sensitive instructions: change the configuration of resources
  - Behaviour-sensitive instructions: read privileged state and reveal physical resources
- Condition for virtualization: An architecture lends itself to virtualization if all sensitive instructions are privileged instructions. (Not always true in practice but providing a good direction)

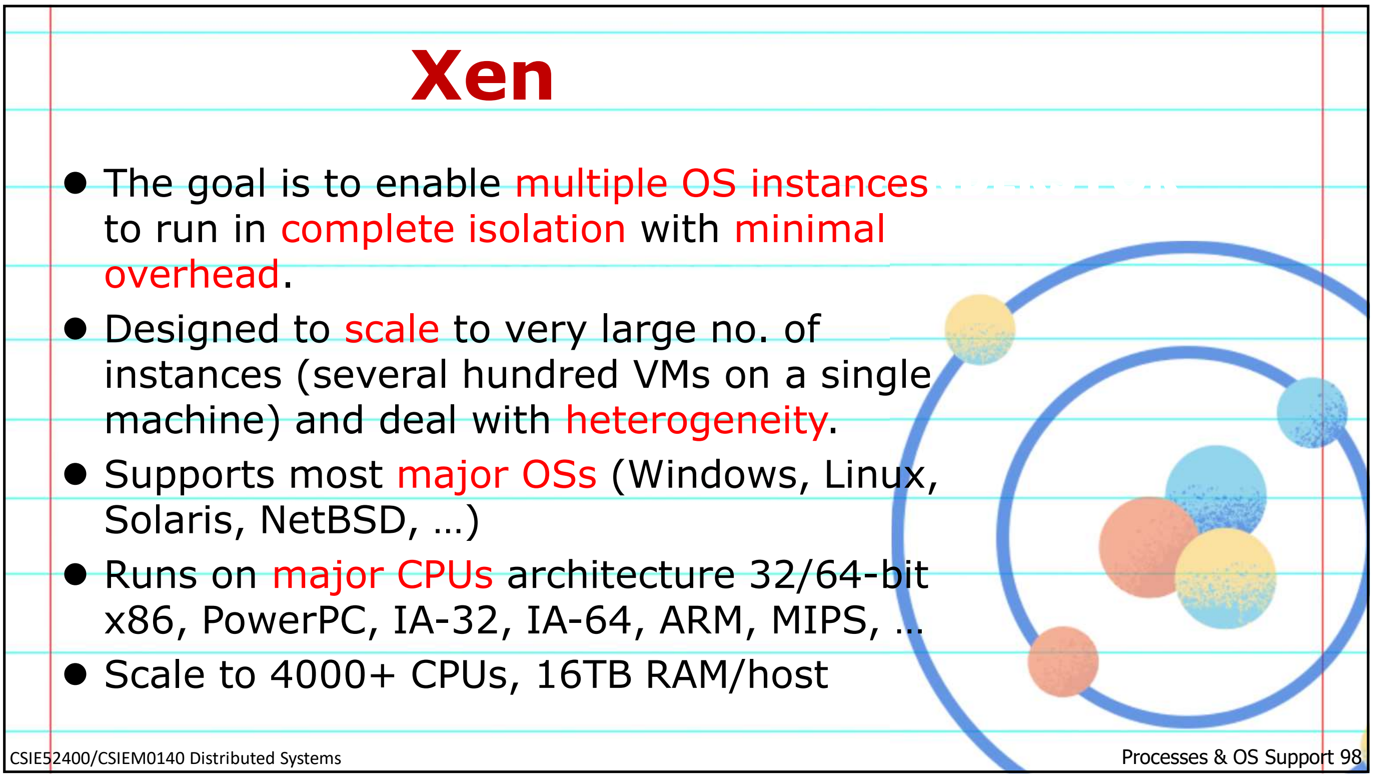
Paravirtualization

- Full virtualization provides a layer of emulation for all instructions and handles sensitive ones within the layer.
  - Guest OSs can run unchanged
  - Expensive
- Paravirtualization
  - Many instructions can run on the bare hardware
  - Privileged instructions are rewritten as hypercalls that trap into the hypervisor
  - Sensitive but nonprivileged instructions should be dealt with by the guest OSs. (need porting)



Case Sudy: Xen

- Xen is another classic example of virtualization.
- Part of the XenoServer project at Cambridge Univ
- XenoServer is an early cloud computing project supporting infrastructure as a service.
- Produce Xen virtual machine monitor and XenoServer Open Platform.
- Initially designed to support XenoServer but evolved into a standalone virtualization solution.
- 2013, Xen Project was moved under Linux Foundation.



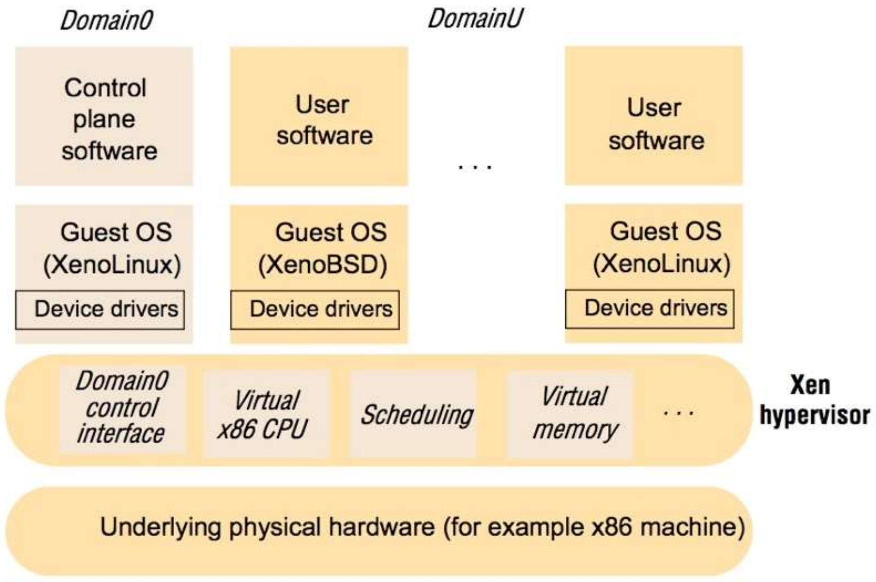
Xen

- The goal is to enable multiple OS instances to run in complete isolation with minimal overhead.
- Designed to scale to very large no. of instances (several hundred VMs on a single machine) and deal with heterogeneity.
- Supports most major OSs (Windows, Linux, Solaris, NetBSD, …)
- Runs on major CPUs architecture 32/64-bit x86, PowerPC, IA-32, IA-64, ARM, MIPS, …
- Scale to 4000+ CPUs, 16TB RAM/host

Xen Architecture

- Xen virtual machine monitor (hypervisor)
  - Virtualize underlying physical resources (CPUs, …)
  - Schedule the physical resources
  - Provide the appearance that each VM has its own (virtualized) physical machine
  - Multiplex the virtual resources onto the physical resources
  - Ensure strong protection between VMs
  - Figure (next slide)

Architecture of Xen



Domain0

DomainU

Control plane software

User software

. . .

User software

Guest OS (XenoLinux)

Device drivers

Guest OS (XenoBSD)

Device drivers

Guest OS (XenoLinux)

Device drivers

Domain0 control interface

Virtual x86 CPU

Scheduling

Virtual memory

. . .

Xen hypervisor

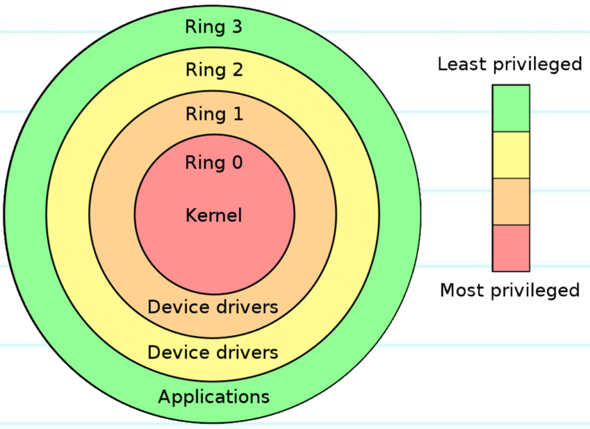
Underlying physical hardware (for example x86 machine)

Xen Design

- Implement only a minimal set of mechanisms for resource management and isolation.
- Primary concern is isolation (domains, faults, …)
- Must be as lightweight as possible to minimize the overhead of two-level (virtual-physical) execution
- Support large no. of VM instances (domains) running guest OSs.
- Guest OSs run in domainU (the unprivileged domain)
- A special domain0 act as control plane with privileged access.

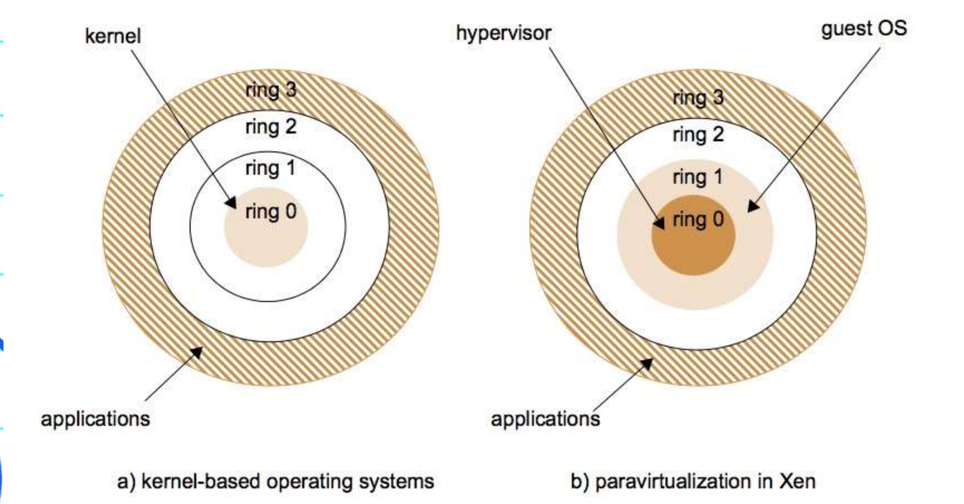
Rings of Privilege

- The hierarchical protection domains (or protection rings) are mechanisms to protect data and functionality from faults and malicious behavior.



Rings of Privilege

- Different OSs may adopt different arch.

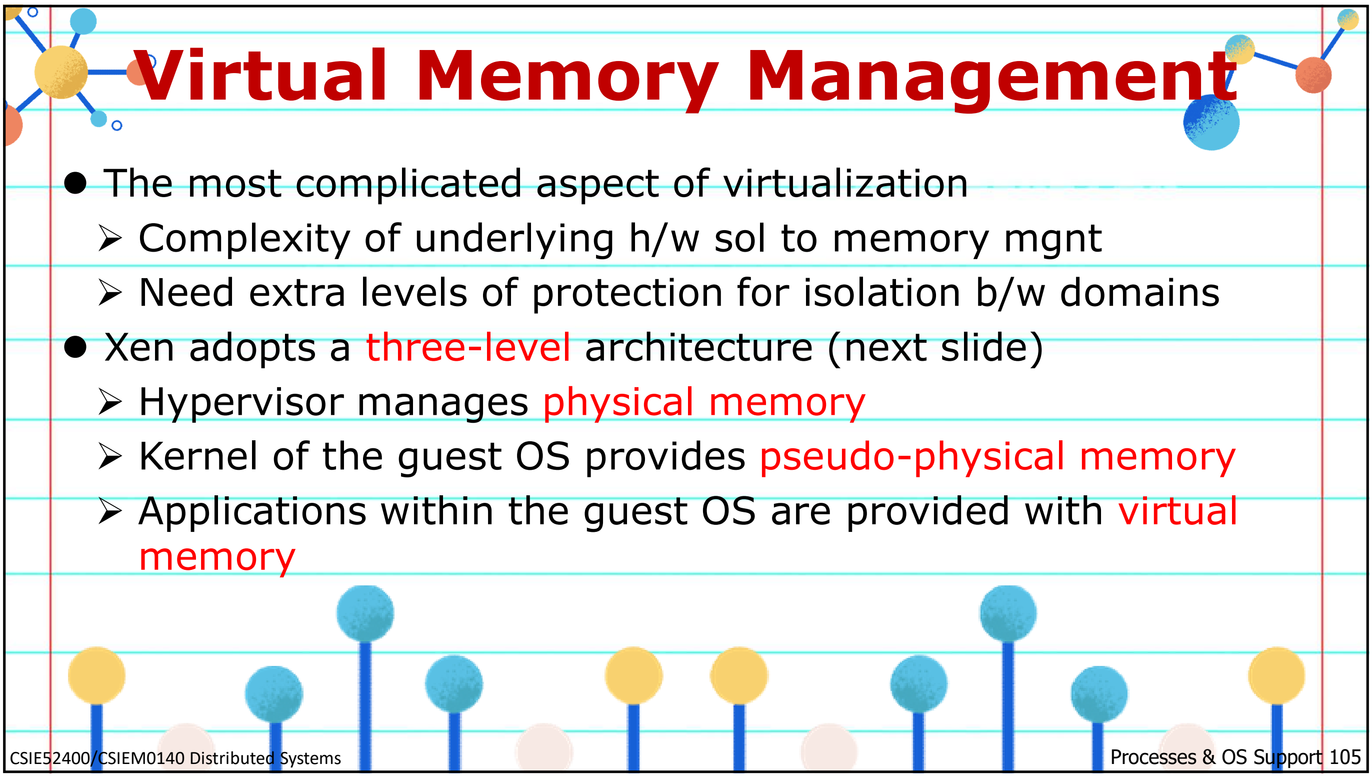


a) kernel-based operating systems

b) paravirtualization in Xen

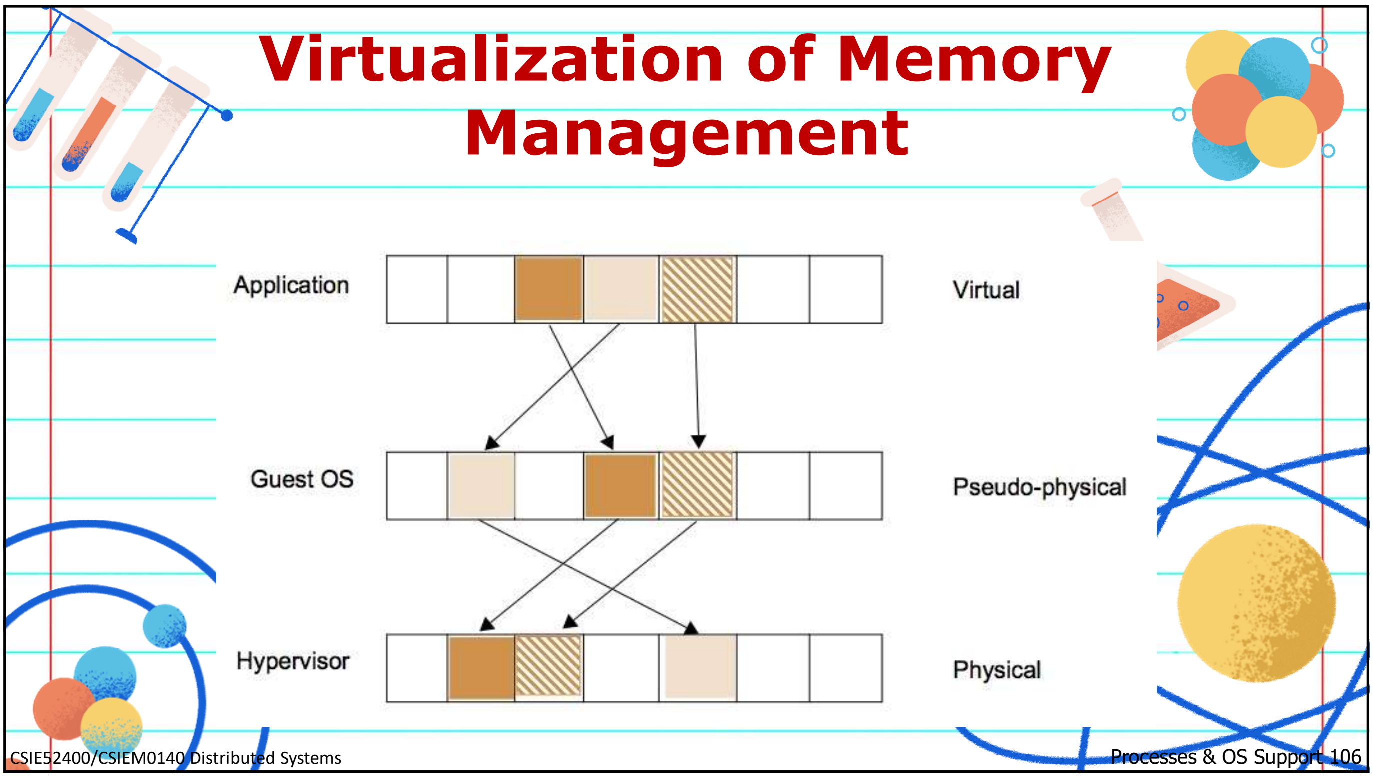
Scheduling

- Many OSs support two-level scheduling
  - ➢ Scheduling of processes
  - ➢ Scheduling of user-level threads within processes
- Xen introduces an extra level of scheduling
  - ➢ Supports virtual CPU (VCPU), each supporting a guest OS
  - ➢ Hypervisor schedules VCPUs onto physical CPUs
  - ➢ Guest OS schedules kernel-level threads onto their allocated VCPUs
  - ➢ Thread libraries schedule user-level threads onto kernel-level threads



Virtual Memory Management

- The most complicated aspect of virtualization
  - Complexity of underlying h/w sol to memory mgnt
  - Need extra levels of protection for isolation b/w domains
- Xen adopts a three-level architecture (next slide)
  - Hypervisor manages physical memory
  - Kernel of the guest OS provides pseudo-physical memory
  - Applications within the guest OS are provided with virtual memory



Virtualization of Memory Management

Application Virtual

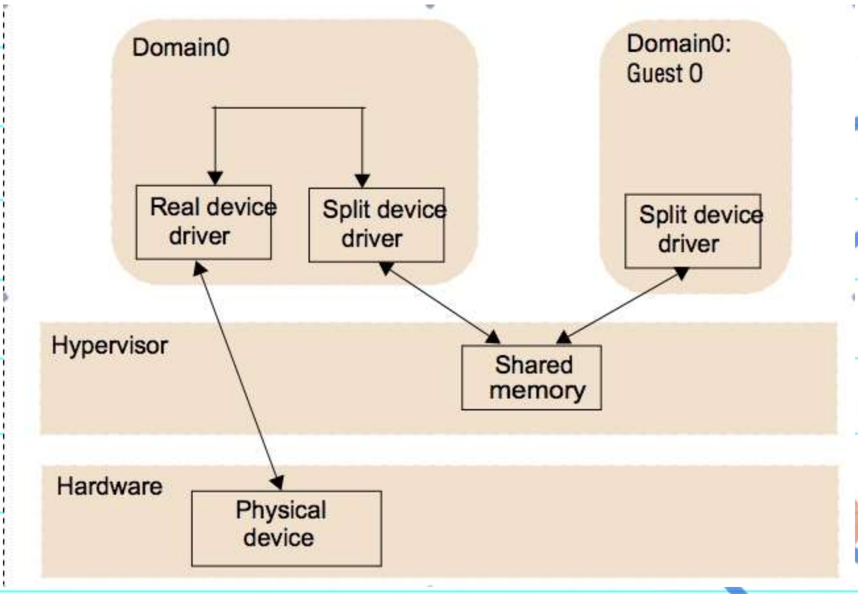
Guest OS Pseudo-physical

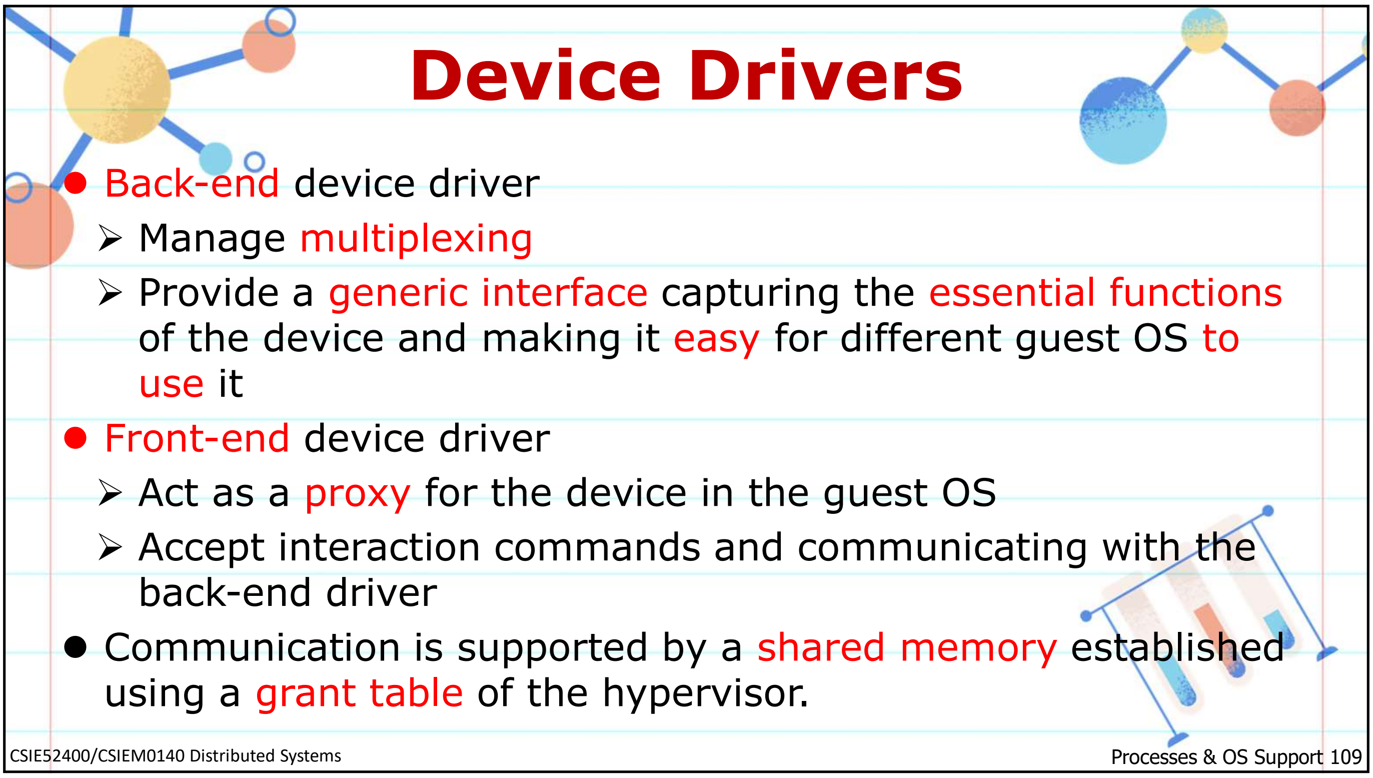
Hypervisor Physical

Device Management

- Rely on split device drivers (next slide)
- Access to a physical device is controlled exclusively by **domain0** with a real device driver.
- Xen need to provide an abstraction with device multiplexing s.t. each guest OS can have its own virtual device.
  - Back-end device driver runs in domain0
  - Front-end device driver runs in the guest OS
  - Two drivers communicate to provide device access for the guest OS

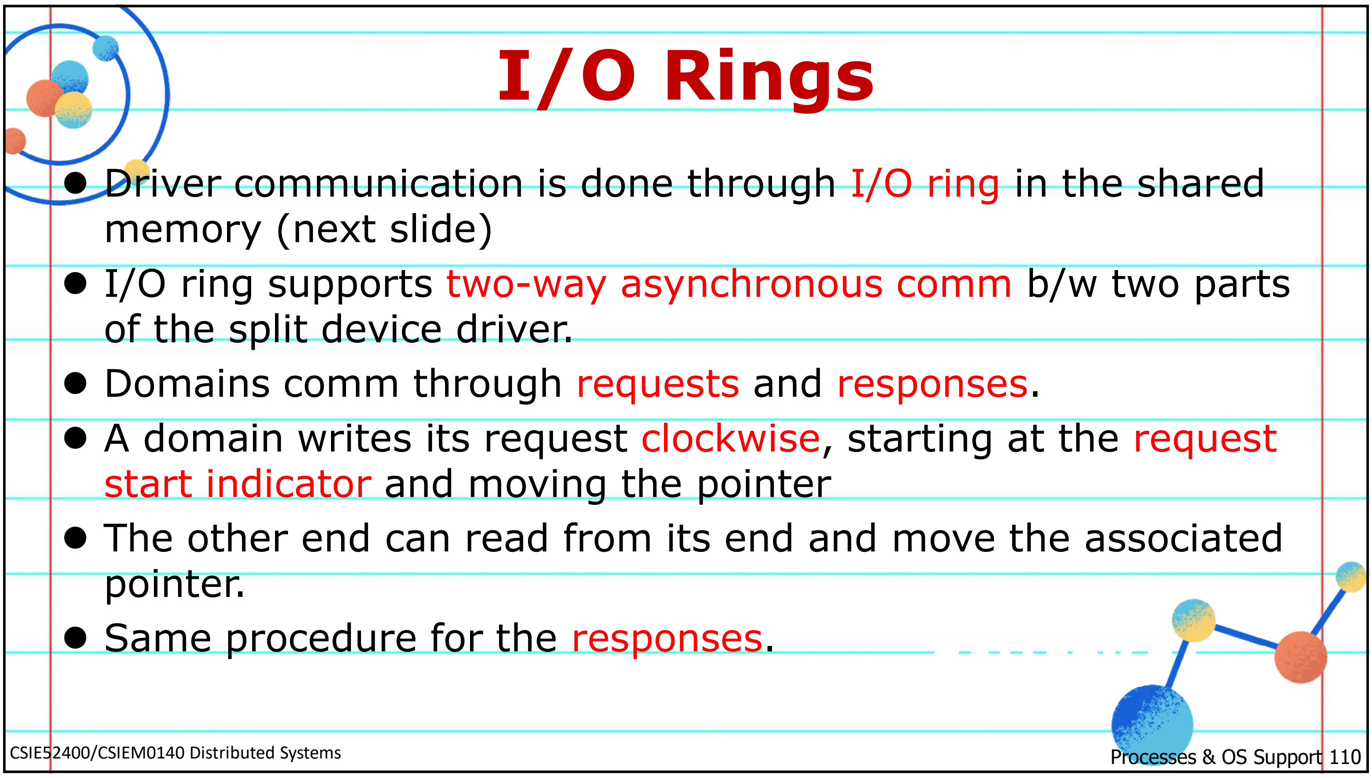
Split Device Drivers





Device Drivers

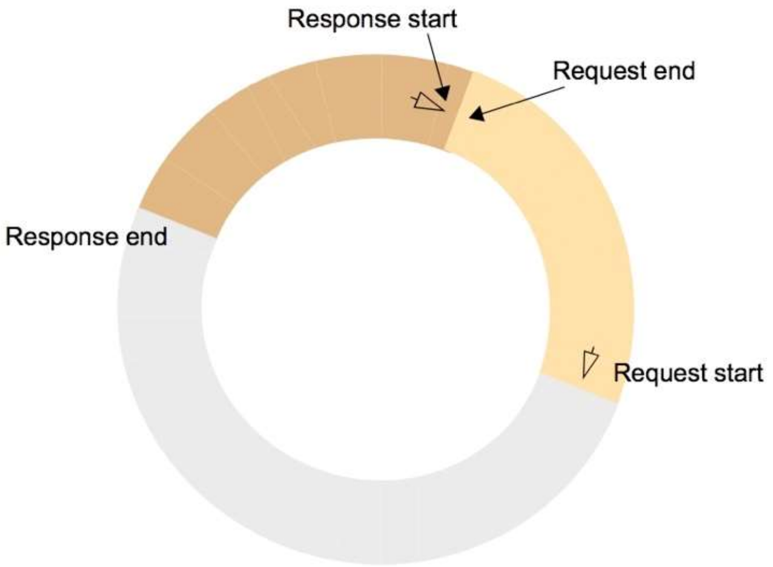
- Back-end device driver
  - Manage multiplexing
  - Provide a generic interface capturing the essential functions of the device and making it easy for different guest OS to use it
- Front-end device driver
  - Act as a proxy for the device in the guest OS
  - Accept interaction commands and communicating with the back-end driver
- Communication is supported by a shared memory established using a grant table of the hypervisor.



I/O Rings

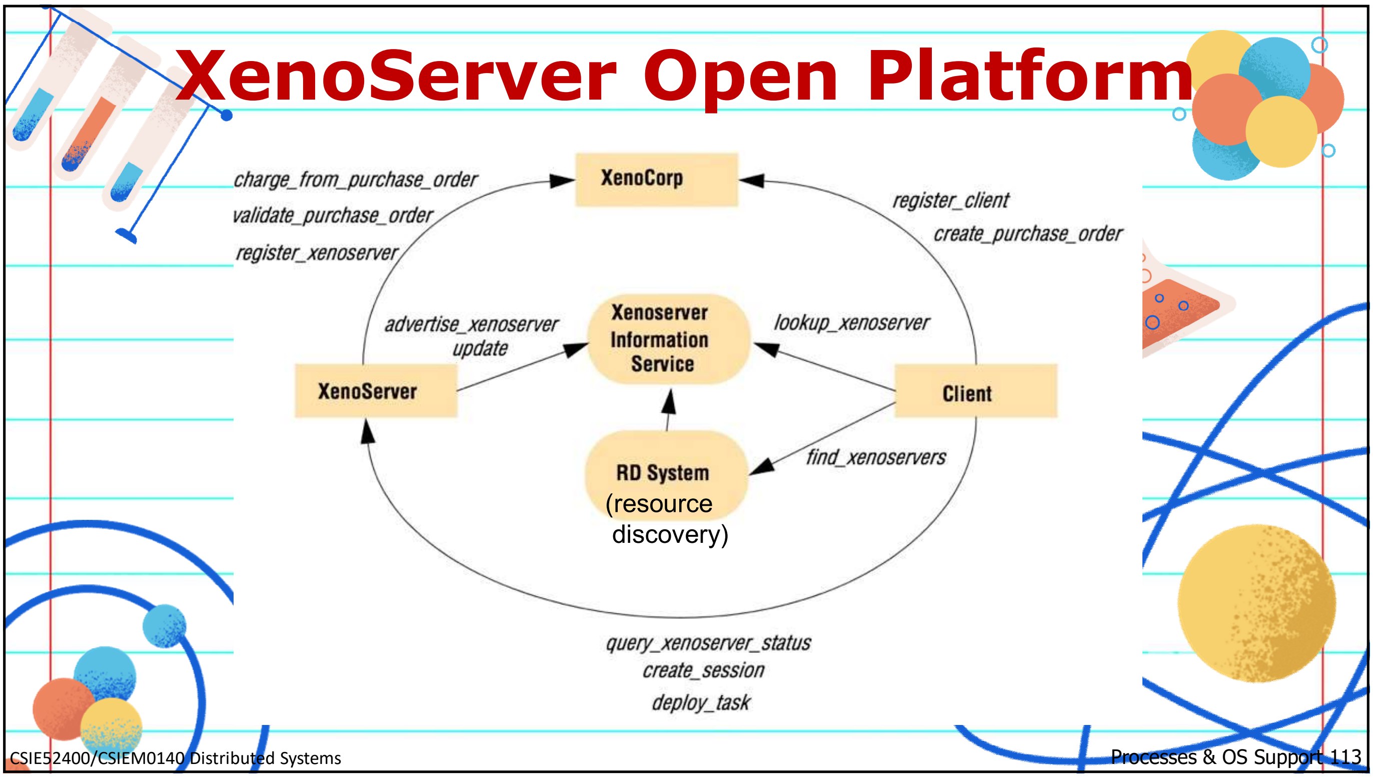
- Driver communication is done through I/O ring in the shared memory (next slide)
- I/O ring supports two-way asynchronous comm b/w two parts of the split device driver.
- Domains comm through requests and responses.
- A domain writes its request clockwise, starting at the request start indicator and moving the pointer
- The other end can read from its end and move the associated pointer.
- Same procedure for the responses.

I/O Rings

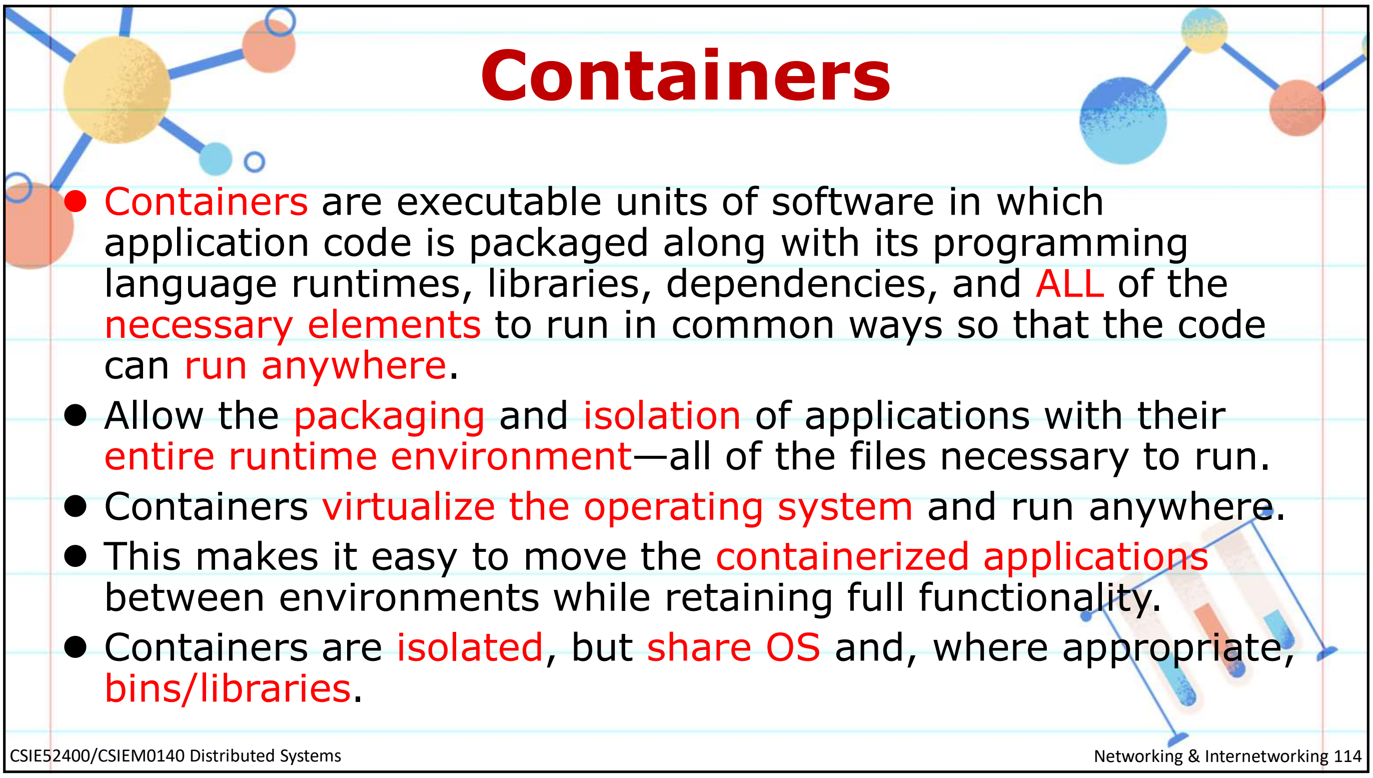


Porting a Guest OS

- Replace all privileged instructions used by the OS with the relevant hypercalls.
- Reimplement all other sensitive instructions in a way that preserves the semantics of the assoc ops
- Port the virtual memory subsystem
- Develop split device drivers for the required set of devices
- Some other more specific tasks need to be carried out: time, clock, …



XenoServer Open Platform

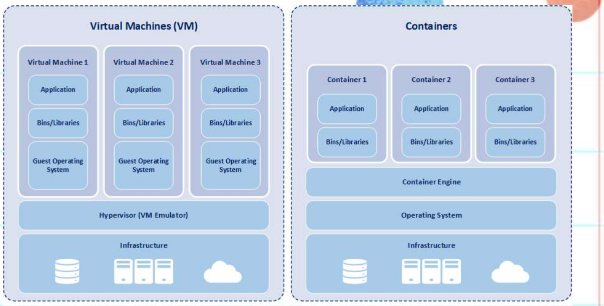


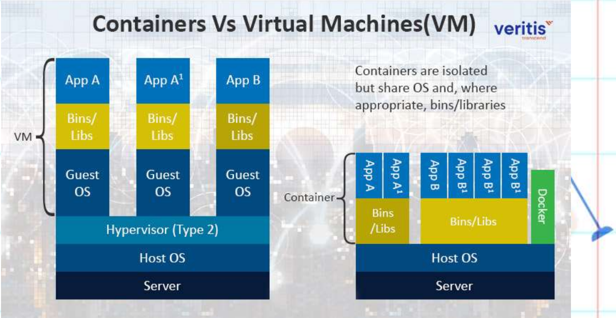
Containers

- Containers are executable units of software in which application code is packaged along with its programming language runtimes, libraries, dependencies, and ALL of the necessary elements to run in common ways so that the code can run anywhere.
- Allow the packaging and isolation of applications with their entire runtime environment—all of the files necessary to run.
- Containers virtualize the operating system and run anywhere.
- This makes it easy to move the containerized applications between environments while retaining full functionality.
- Containers are isolated, but share OS and, where appropriate, bins/libraries.

Containers vs VM

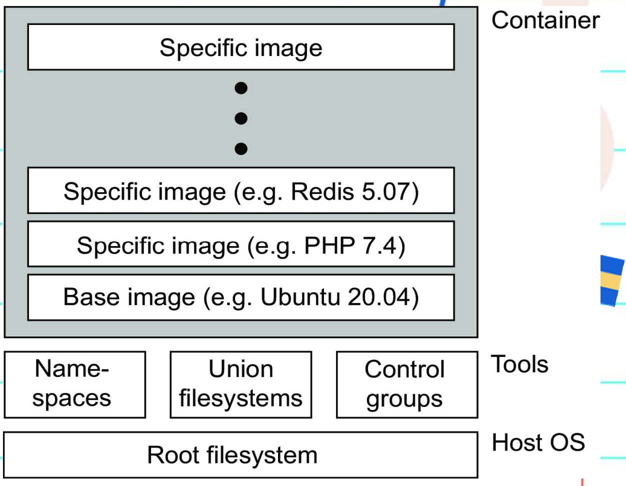
- Both are mechanisms to abstract physical hardware and run applications within independent spaces.
- They are both ways of deploying applications while isolating the application from the underlying hardware.
- But they function differently: containers share an OS while VM contain a complete and independent OS.





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



Container

Specific image

●
●
●

Specific image (e.g. Redis 5.07)

Specific image (e.g. PHP 7.4)

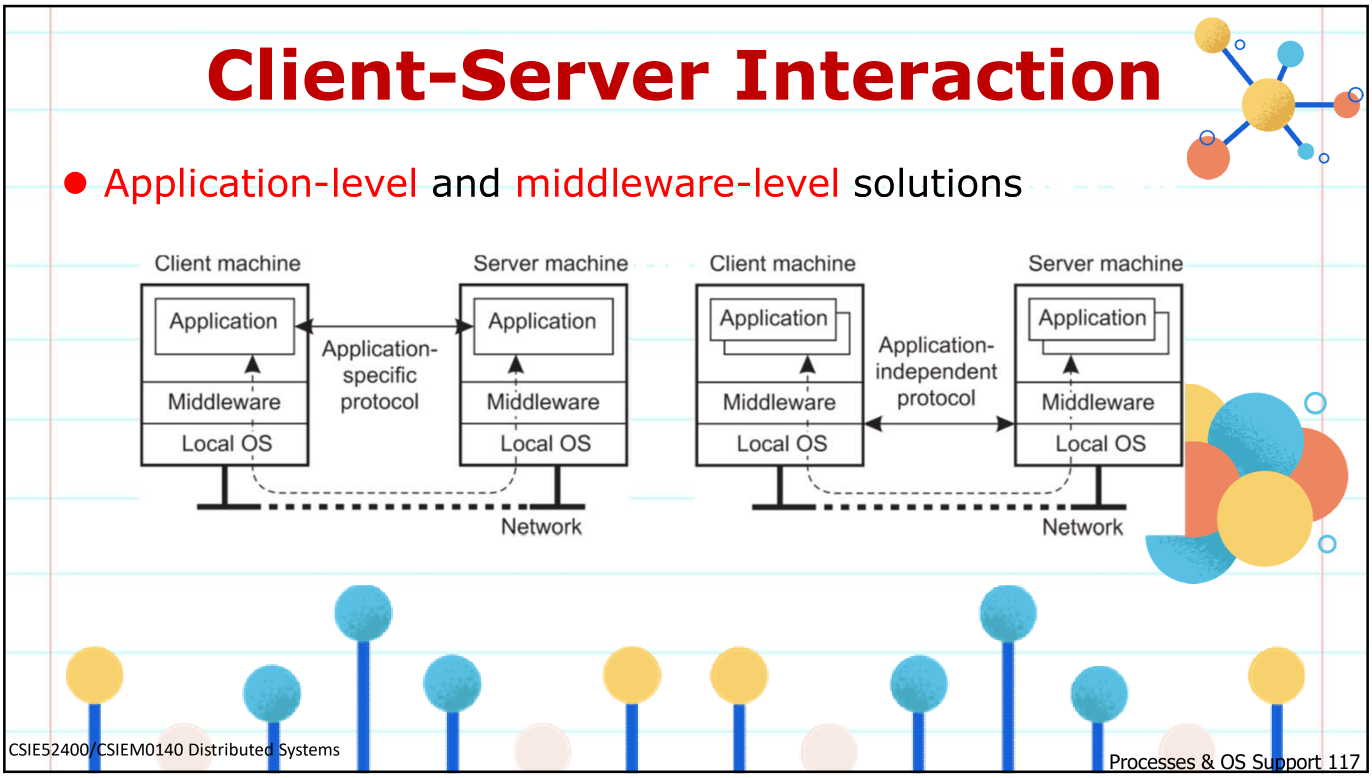
Base image (e.g. Ubuntu 20.04)

Tools

Name-spaces | Union filesystems | Control groups

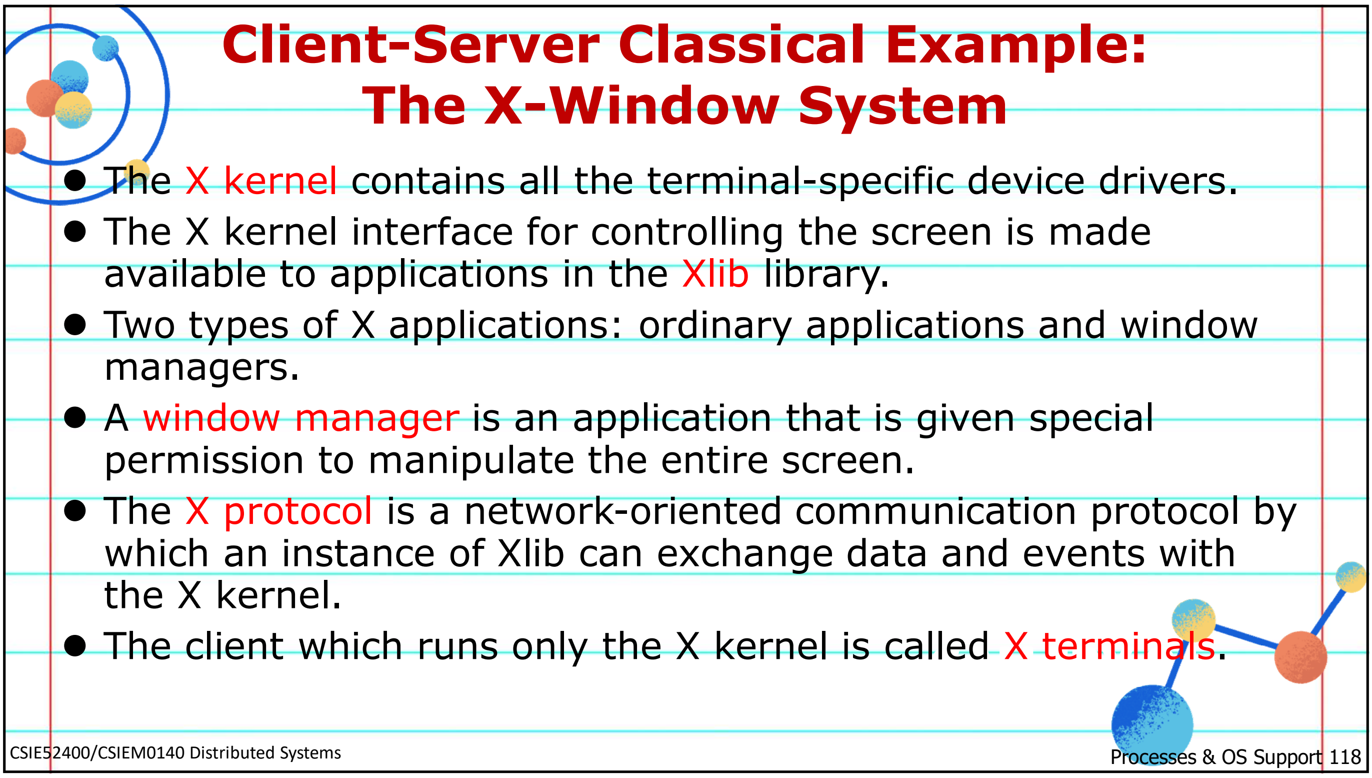
Host OS

Root filesystem



Client-Server Interaction

- Application-level and middleware-level solutions

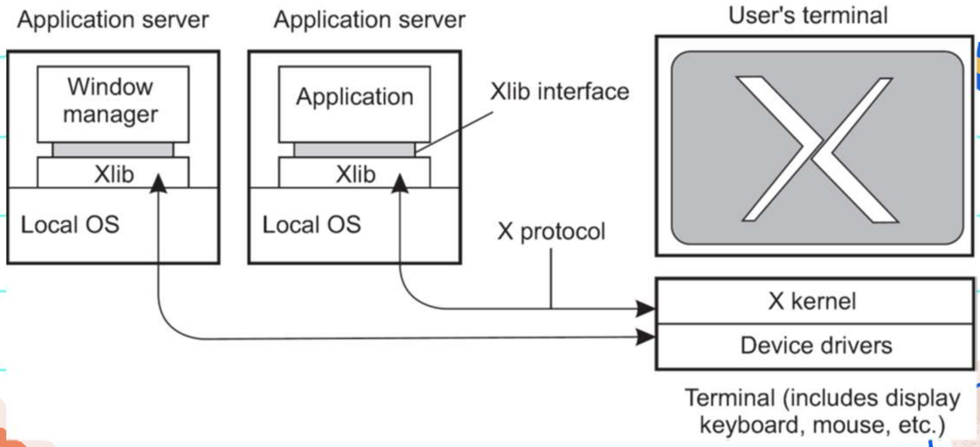


Client-Server Classical Example: The X-Window System

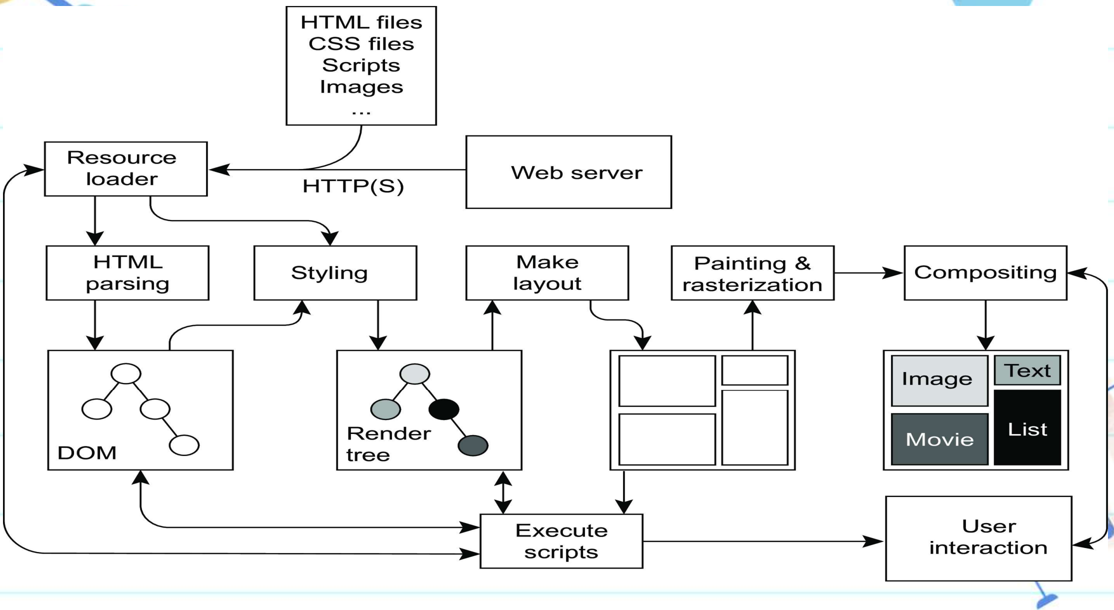
- The X kernel contains all the terminal-specific device drivers.
- The X kernel interface for controlling the screen is made available to applications in the Xlib library.
- Two types of X applications: ordinary applications and window managers.
- A window manager is an application that is given special permission to manipulate the entire screen.
- The X protocol is a network-oriented communication protocol by which an instance of Xlib can exchange data and events with the X kernel.
- The client which runs only the X kernel is called X terminals.

Client-Server Classical Example: The X-Window System

- The basic organization of the X Window System (Windows Remote Desktop offers similar func).

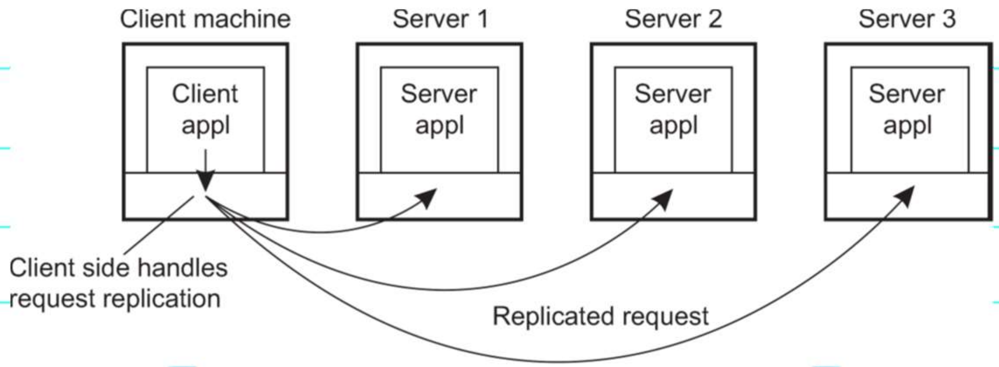


Anatomy of a Web Browser



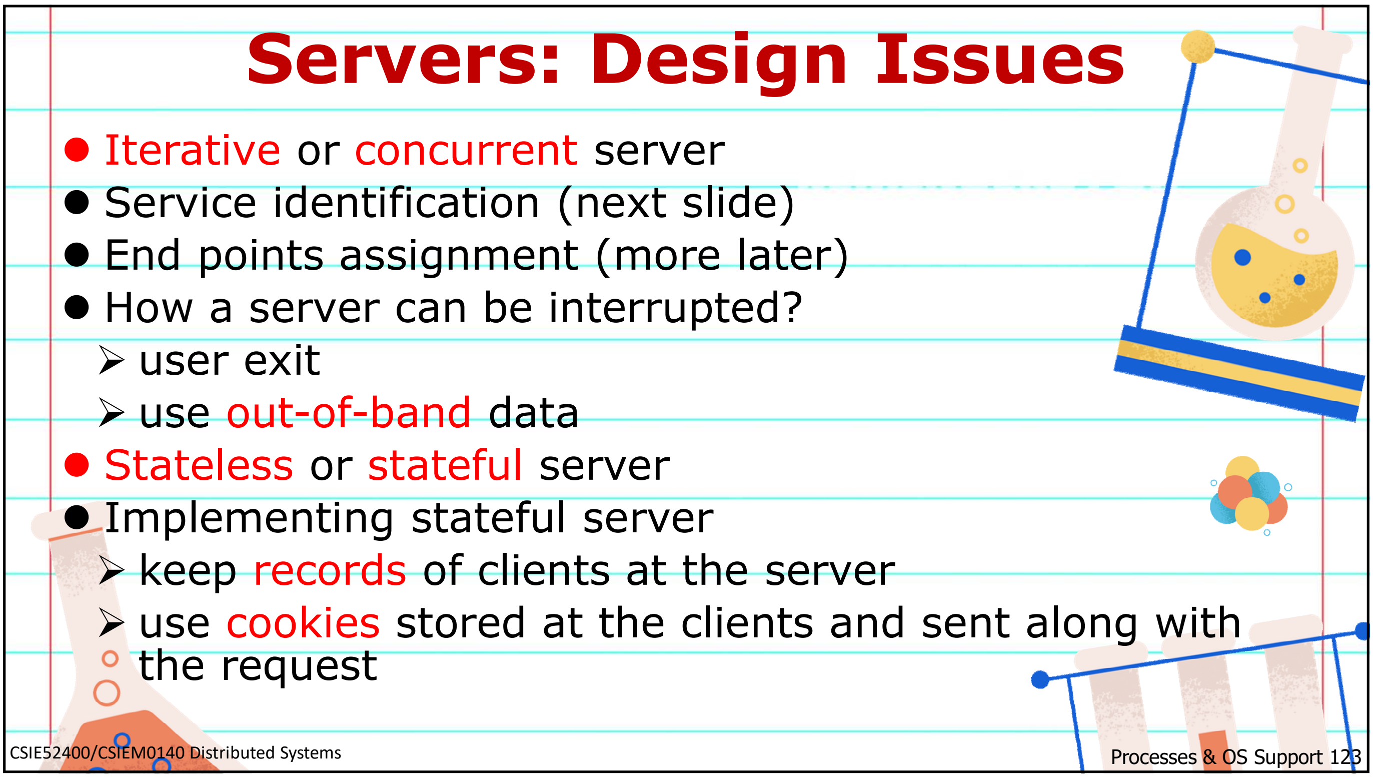
Client-Side Software for Distribution Transparency

- A possible approach to transparent replication of a remote object using a client-side solution.



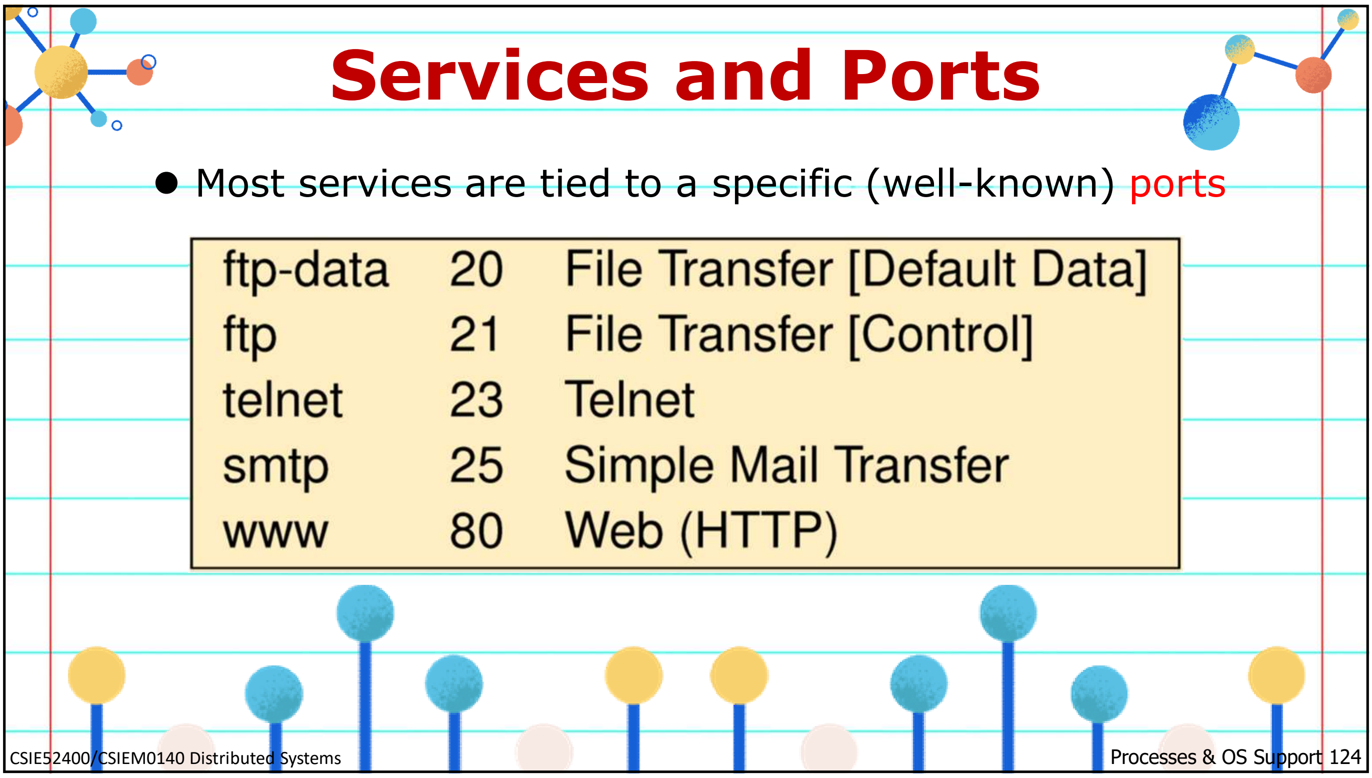
Other Client-Server Examples

- Mail servers and clients.
- File servers and terminal systems.
- DNS (Domain Name Server)
- Database clients and DB server
- Remote Desktop
- Video Streaming (e.g., YouTube app)
- VoIP (e.g., Skype)
- Cloud Storage (e.g., Dropbox)
- …



Servers: Design Issues

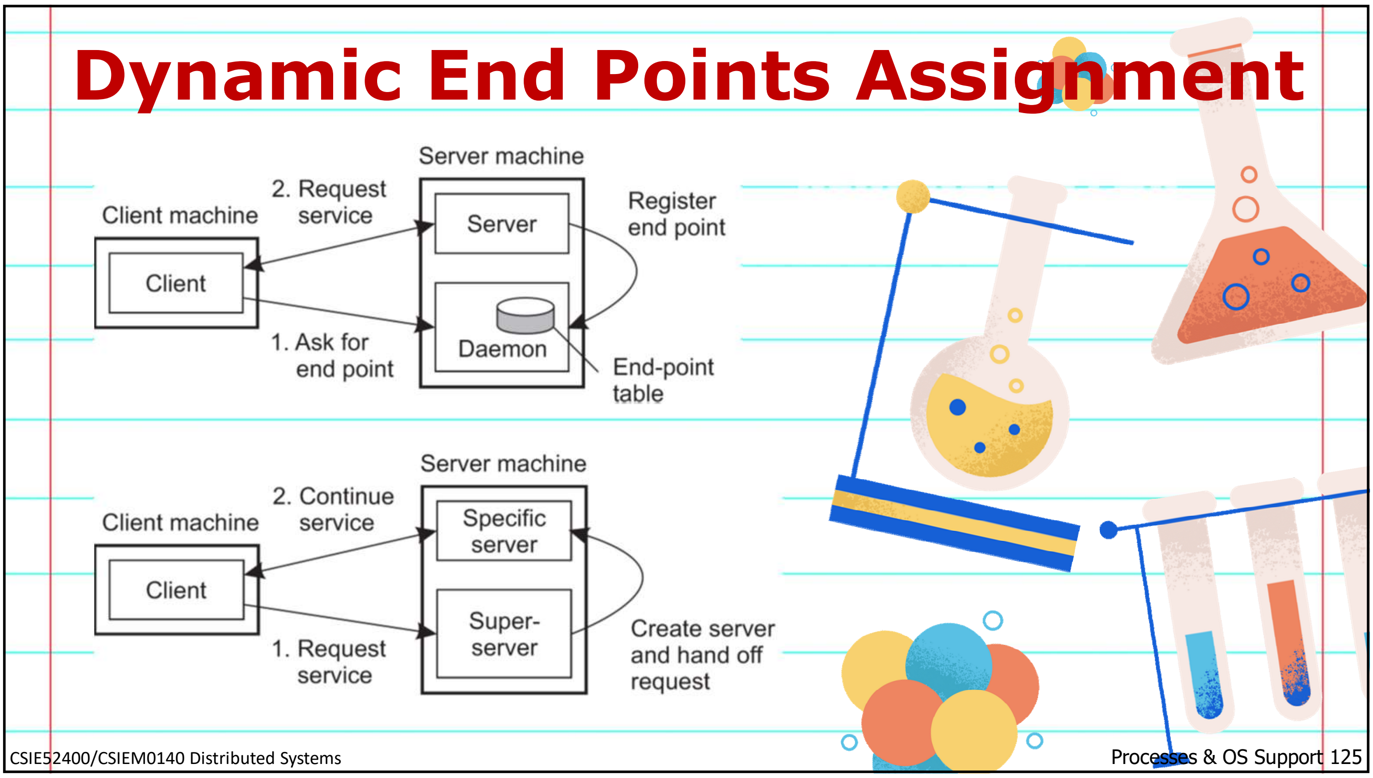
- Iterative or concurrent server
- Service identification (next slide)
- End points assignment (more later)
- How a server can be interrupted?
  - user exit
  - use out-of-band data
- Stateless or stateful server
- Implementing stateful server
  - keep records of clients at the server
  - use cookies stored at the clients and sent along with the request



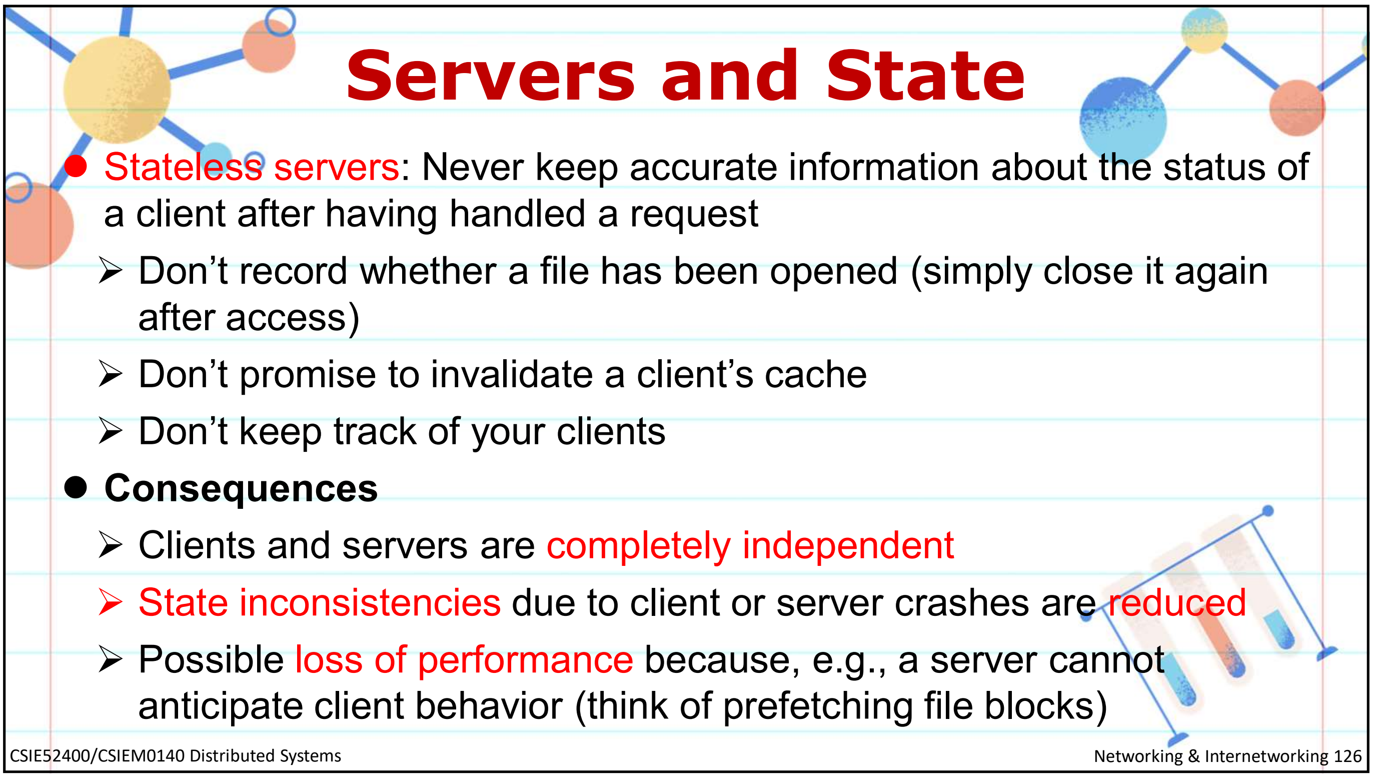
Services and Ports

- Most services are tied to a specific (well-known) ports

ftp-data	20	File Transfer [Default Data]
ftp	21	File Transfer [Control]
telnet	23	Telnet
smtp	25	Simple Mail Transfer
www	80	Web (HTTP)



Dynamic End Points Assignment



Servers and State

- Stateless servers: Never keep accurate information about the status of a client after having handled a request
  - ➢ Don’t record whether a file has been opened (simply close it again after access)
  - ➢ Don’t promise to invalidate a client’s cache
  - ➢ Don’t keep track of your clients
- **Consequences**
  - ➢ Clients and servers are completely independent
  - ➢ State inconsistencies due to client or server crashes are reduced
  - ➢ Possible loss of performance because, e.g., a server cannot anticipate client behavior (think of prefetching file blocks)

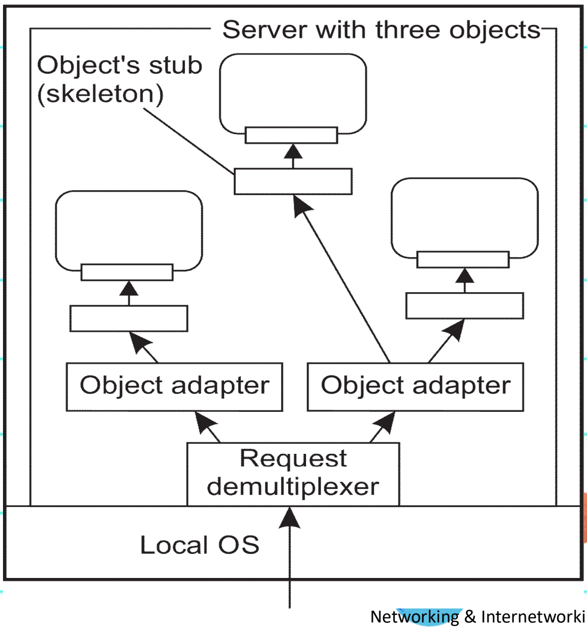
Servers and State

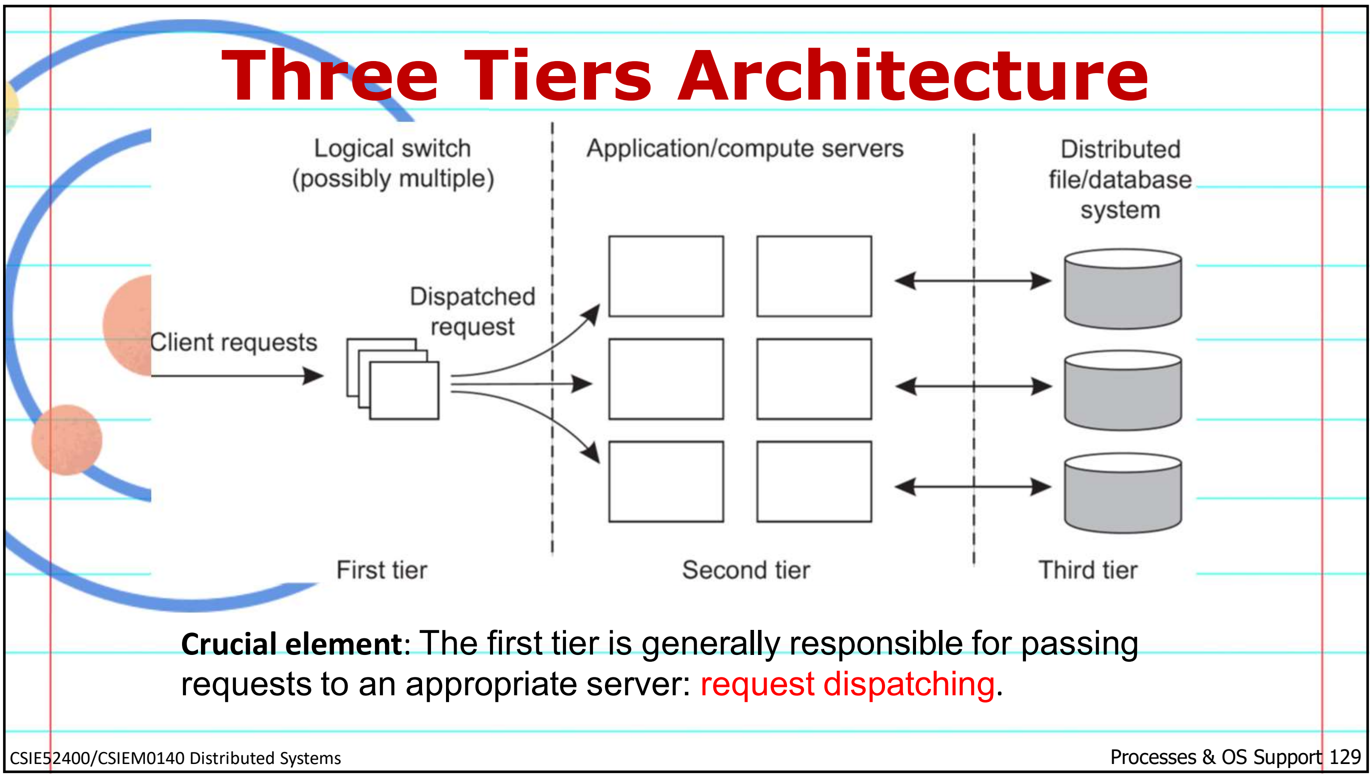
- Stateful servers: Keeps track of the status of its clients
  - ➢ Record that a file has been opened, so that prefetching can be done
  - ➢ Knows which data a client has cached, and allows clients to keep local copies of shared data

- **Observation**: The performance of stateful servers can be extremely high, provided clients are allowed to keep local copies. As it turns out, reliability is often not a major problem.

Object Servers

- Activation policy: which actions to take when an invocation request comes in:
  - ➢ Where are code and data of the object?
  - ➢ Which threading model to use?
  - ➢ Keep modified state of object, if any?
- Object adapter: implements a specific activation policy





Three Tiers Architecture

Logical switch (possibly multiple)

Application/compute servers

Distributed file/database system

Client requests

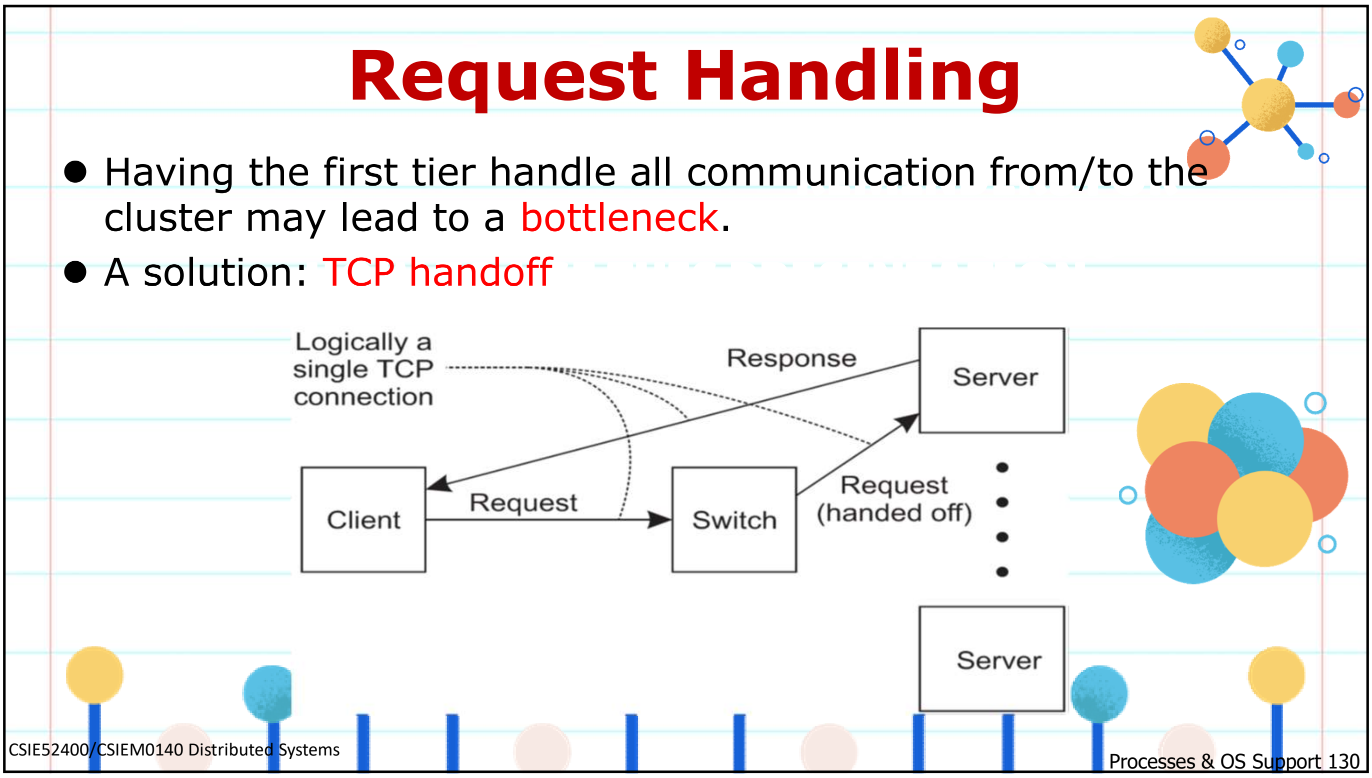
Dispatched request

First tier

Second tier

Third tier

Crucial element: The first tier is generally responsible for passing requests to an appropriate server: request dispatching.



Request Handling

- Having the first tier handle all communication from/to the cluster may lead to a bottleneck.
- A solution: TCP handoff

Logically a single TCP connection

Response

Server

Client

Request

Switch

Request (handed off)

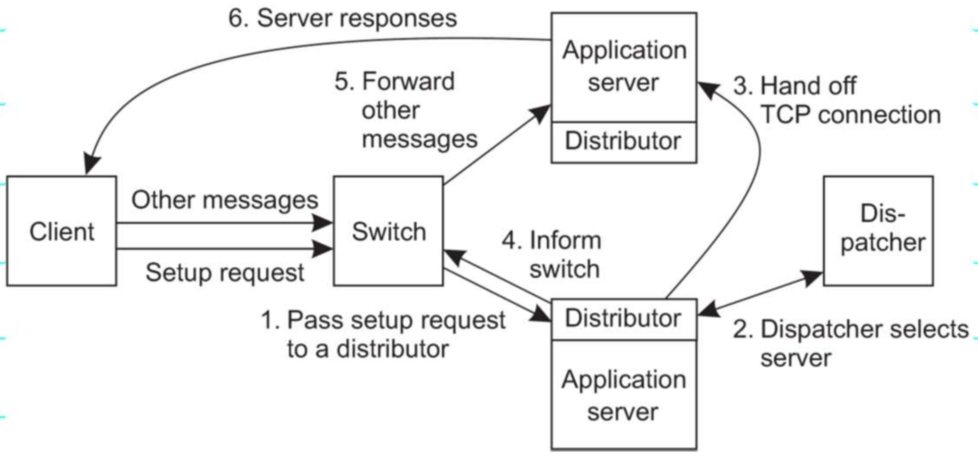
Server

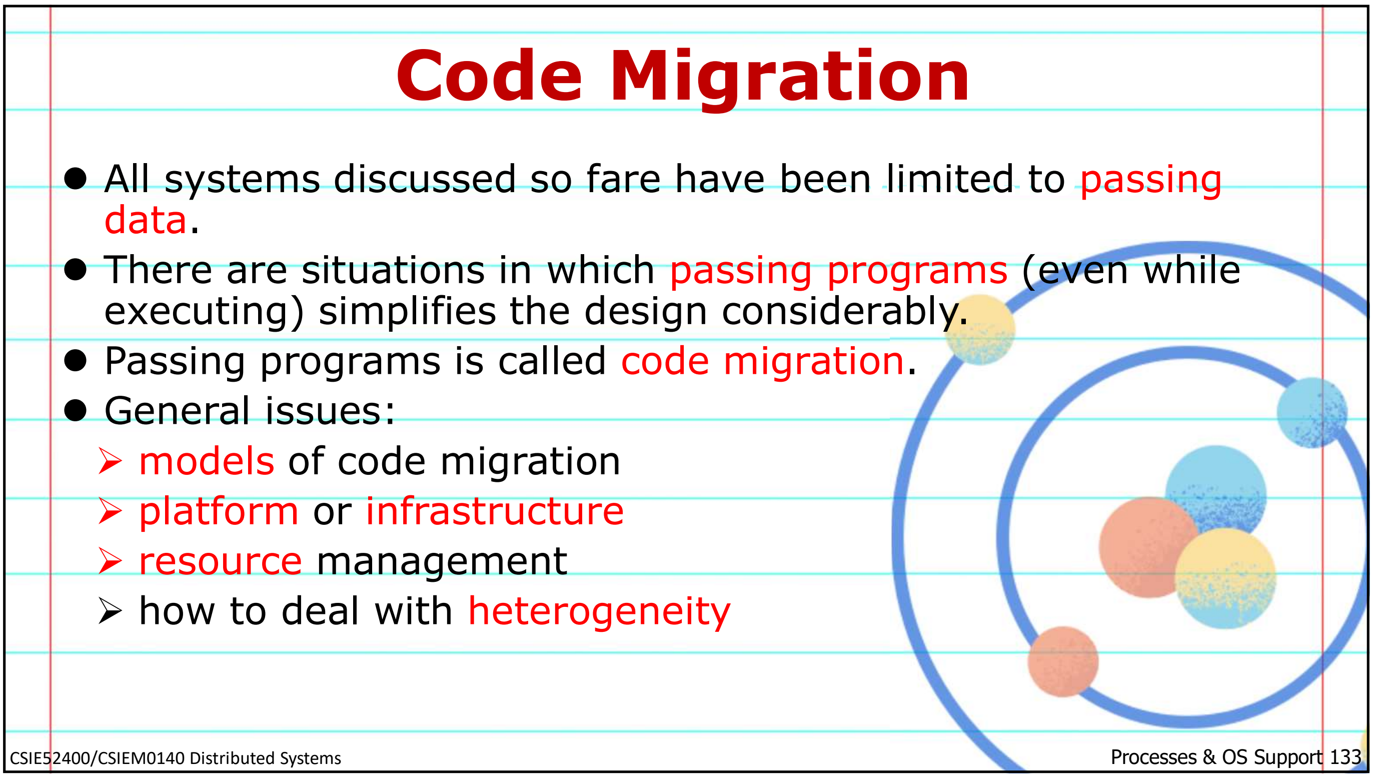
Server Clusters

- The front end may easily get overloaded.
- Transport-layer switching: Front end simply passes the TCP request to one of the servers.
- Content-aware distribution: Front end reads the request content and selects the best server.

Server Clusters

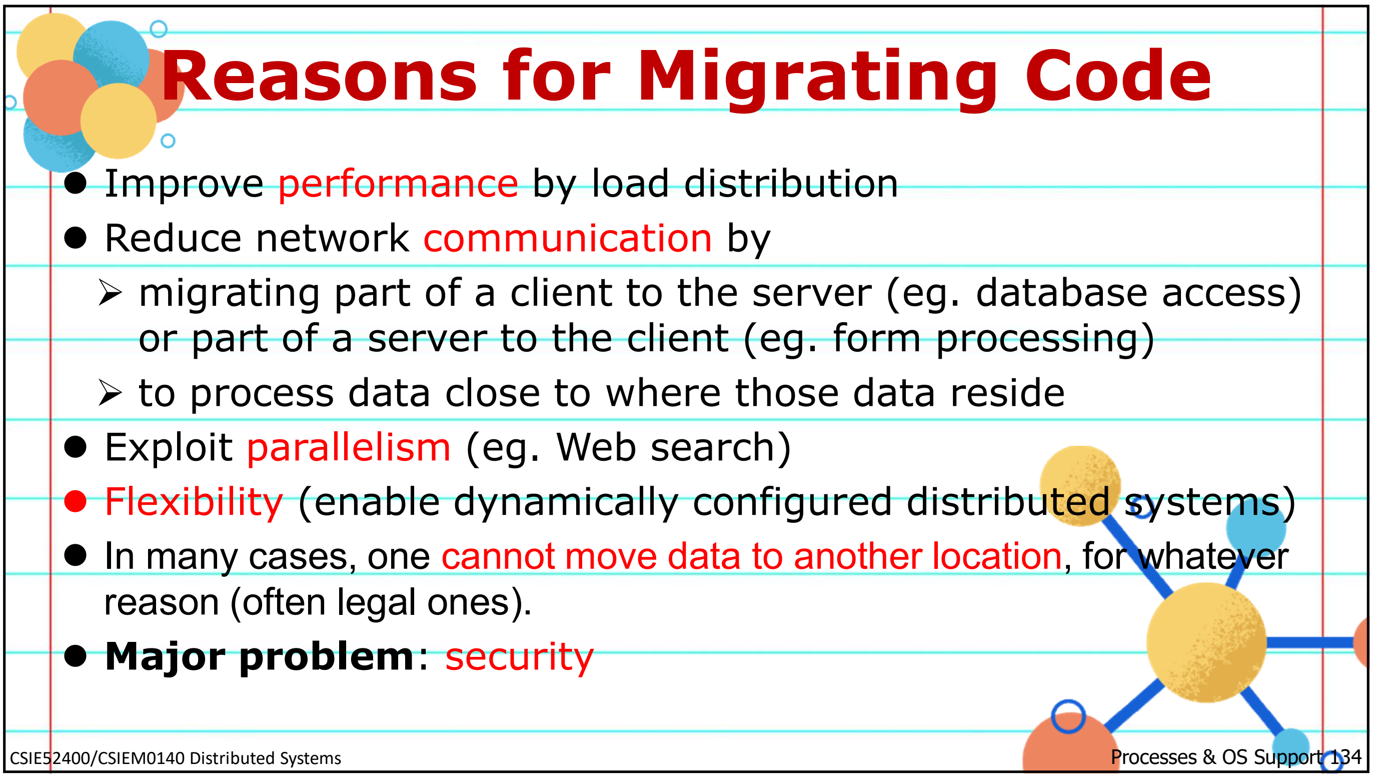
- Combining two solutions:





Code Migration

- All systems discussed so fare have been limited to passing data.
- There are situations in which passing programs (even while executing) simplifies the design considerably.
- Passing programs is called code migration.
- General issues:
  - models of code migration
  - platform or infrastructure
  - resource management
  - how to deal with heterogeneity

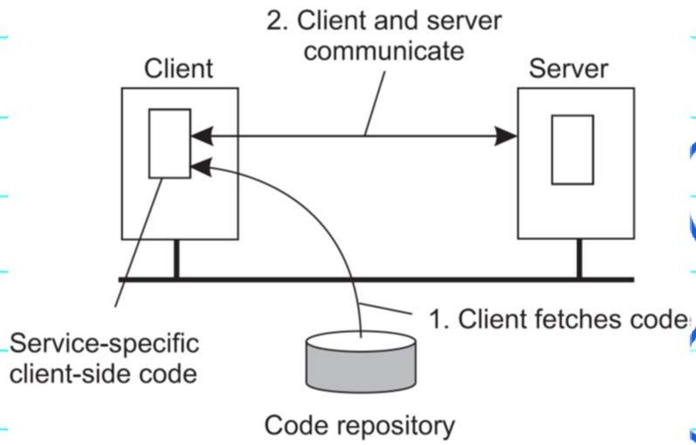


Reasons for Migrating Code

- Improve performance by load distribution
- Reduce network communication by
  - migrating part of a client to the server (eg. database access) or part of a server to the client (eg. form processing)
  - to process data close to where those data reside
- Exploit parallelism (eg. Web search)
- Flexibility (enable dynamically configured distributed systems)
- In many cases, one cannot move data to another location, for whatever reason (often legal ones).
- **Major problem**: security

Dynamic Configuration of a Client

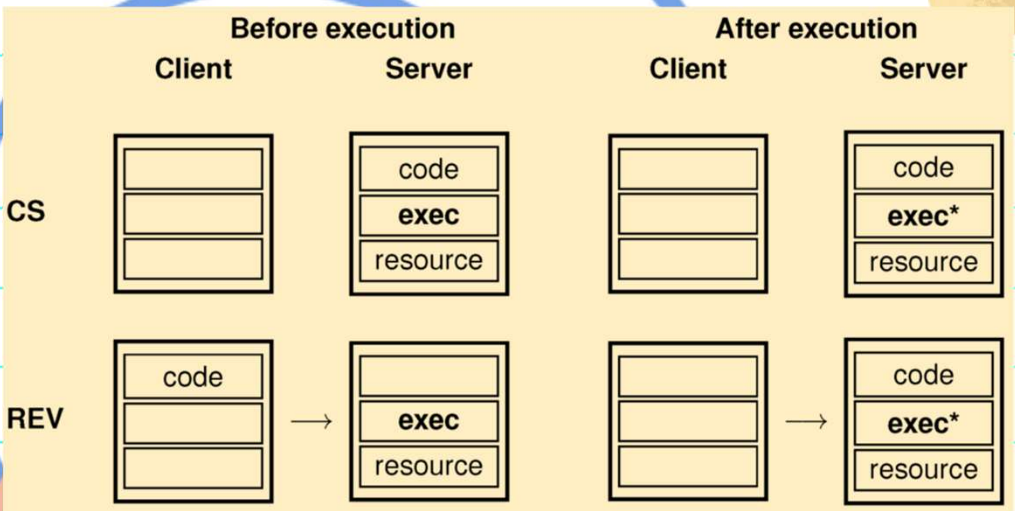
- The principle of dynamically configuring a client to communicate to a server. The client first fetches the necessary software, and then invokes the server.


2. Client and server communicate
Client
Server
Service-specific client-side code
1. Client fetches code
Code repository

Models for Code Migration

- Traditionally, code migration in distributed systems is termed process migration.
- For sake of migration, a process can be considered as consists of three segments:
  - ➢ code segment – the program code
  - ➢ resource segment – references to resources needed by the process
  - ➢ execution segment – current execution state
- Different models for code migration may migrate different segments of a process

Models for Code Migration

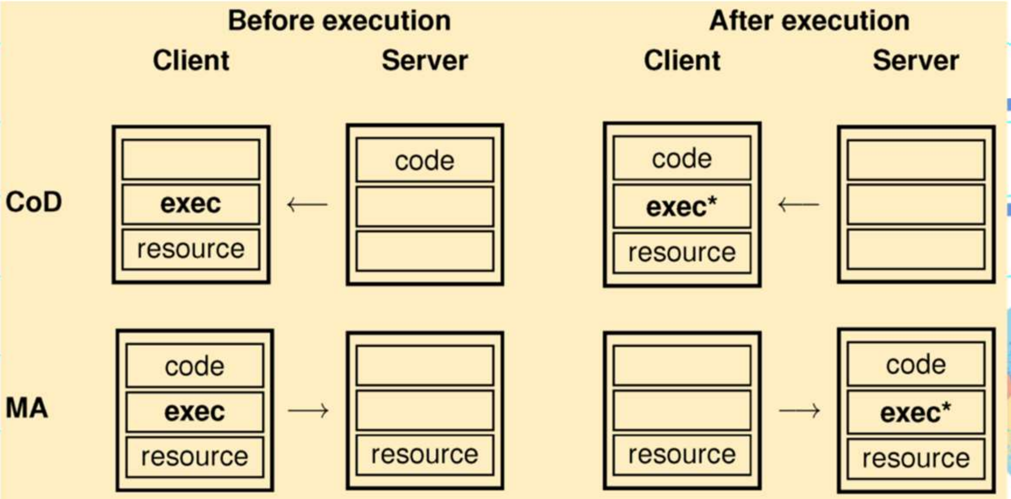


	Before execution			After execution		
	Client		Server	Client		Server
CS			code exec resource			code exec* resource
REV	code	→	exec resource		→	code exec* resource

CS: Client-Server

REV: Remote evaluation

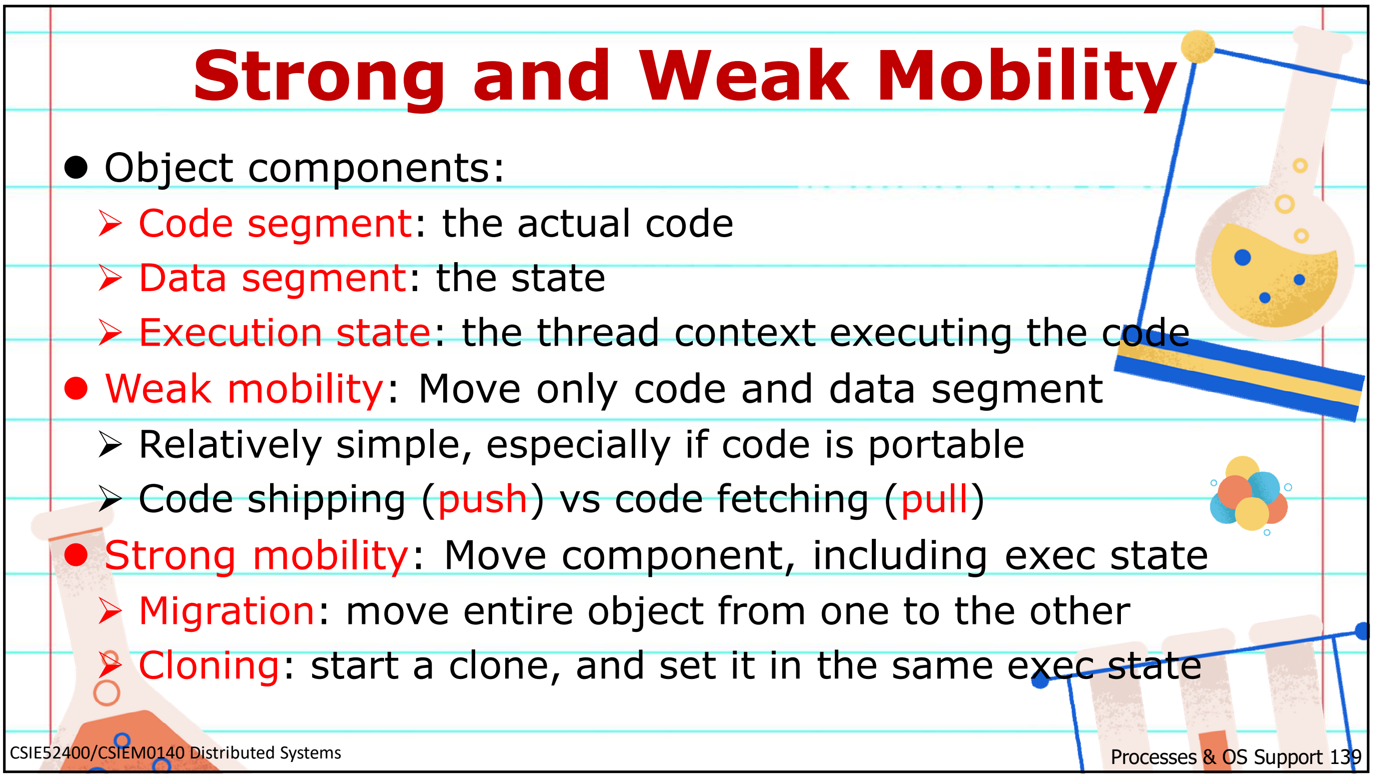
Models for Code Migration



	Before execution			After execution		
	Client		Server	Client		Server
CoD	exec resource	←	code	code exec* resource	←	
MA	code exec resource	→	resource	resource	→	code exec* resource

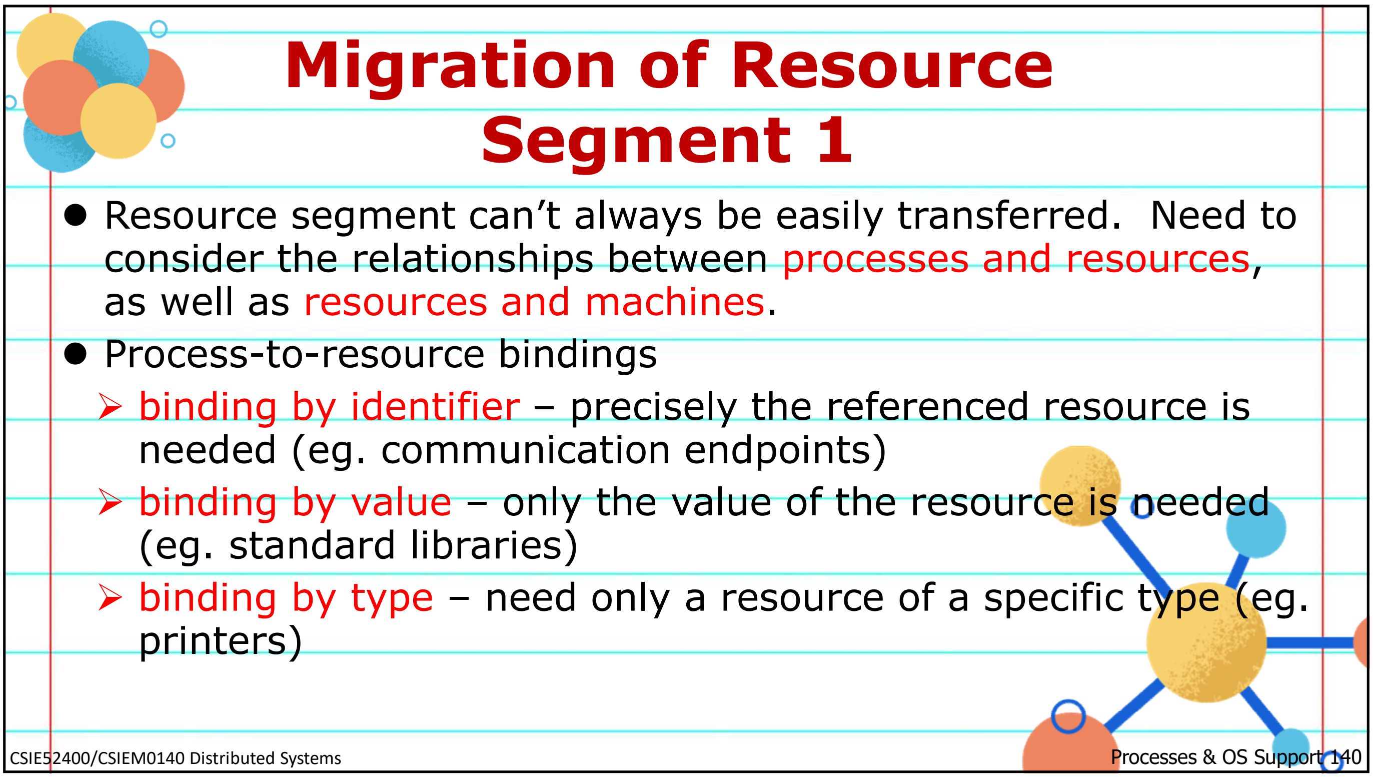
CoD: Code-on-demand

MA: Mobile agents



Strong and Weak Mobility

- Object components:
  - Code segment: the actual code
  - Data segment: the state
  - Execution state: the thread context executing the code
- Weak mobility: Move only code and data segment
  - Relatively simple, especially if code is portable
  - Code shipping (push) vs code fetching (pull)
- Strong mobility: Move component, including exec state
  - Migration: move entire object from one to the other
  - Cloning: start a clone, and set it in the same exec state



Migration of Resource Segment 1

- Resource segment can't always be easily transferred. Need to consider the relationships between processes and resources, as well as resources and machines.
- Process-to-resource bindings
  - binding by identifier – precisely the referenced resource is needed (eg. communication endpoints)
  - binding by value – only the value of the resource is needed (eg. standard libraries)
  - binding by type – need only a resource of a specific type (eg. printers)

Migration of Resource Segment 2

- Resource-to-machine bindings:
  - unattached resources – can be easily moved between machines (eg. data files associated with the migrated code)
  - fastened resources – can be moved, but only at relative high costs (eg. databases, Web sites)
  - fixed resources – intimately bound to a specific machine/environment and cannot be moved (eg. local devices, communication endpoints)
- Combining the two bindings to get 9 combinations to cope with.

Migration and Local Resources

- Actions to be taken with respect to the references to local resources when migrating code to another machine.

		Resource-to-machine binding		
		Unattached	Fastened	Fixed
Process-to-resource binding	By identifier	MV (or GR)	GR (or MV)	GR
	By value	CP (or MV, GR)	GR (or CP)	GR
	By type	RB (or GR, CP)	RB (or GR, CP)	RB (or GR)

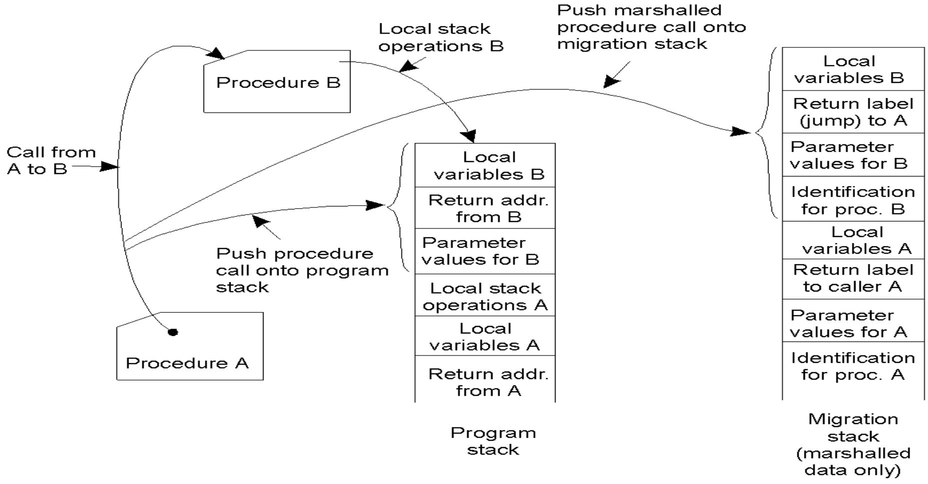
GR	establish a global system wide reference
MV	Move the resource
CP	Copy the value of the resource
RB	Rebind process to locally available resource

Migration in Heterogeneous Systems

- **Main problem**:
  - The target machine may not be suitable to execute the migrated code
  - The definition of process/thread/processor context is highly dependent on local hardware, operating system and runtime system
- **Solution**: abstract machine implemented on different platforms
  - Interpreted languages, effectively having their own VM
  - Virtual machine monitors
- **Observation**: As containers are directly dependent on the underlying OS, their migration in heterogeneous environments is far from trivial, to simply impractical, just as process migration is.

Migration in Heterogeneous Systems

- The principle of maintaining a migration stack to support migration of an execution segment in a heterogeneous environment

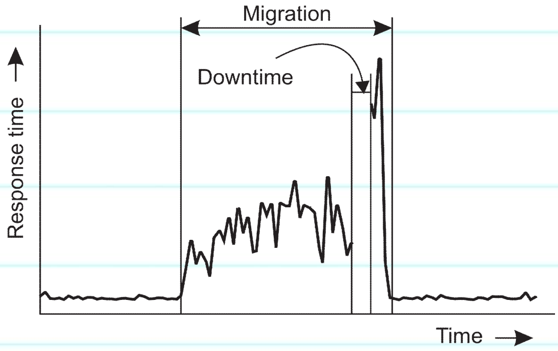


Migrating a Virtual Machine

- Migrating images: three alternatives
  1. Pushing memory pages to the new machine and resending the ones that are later modified during the migration process.
  2. Stopping the current virtual machine; migrate memory, and start the new virtual machine.
  3. Letting the new virtual machine pull in new pages as needed: processes start on the new virtual machine immediately and copy memory pages on demand.

Performance Issue

- A complete migration may actually take tens of seconds. We also need to realize that during the migration, a service will be completely unavailable for multiple seconds.
- Measurements regarding response times during VM migration



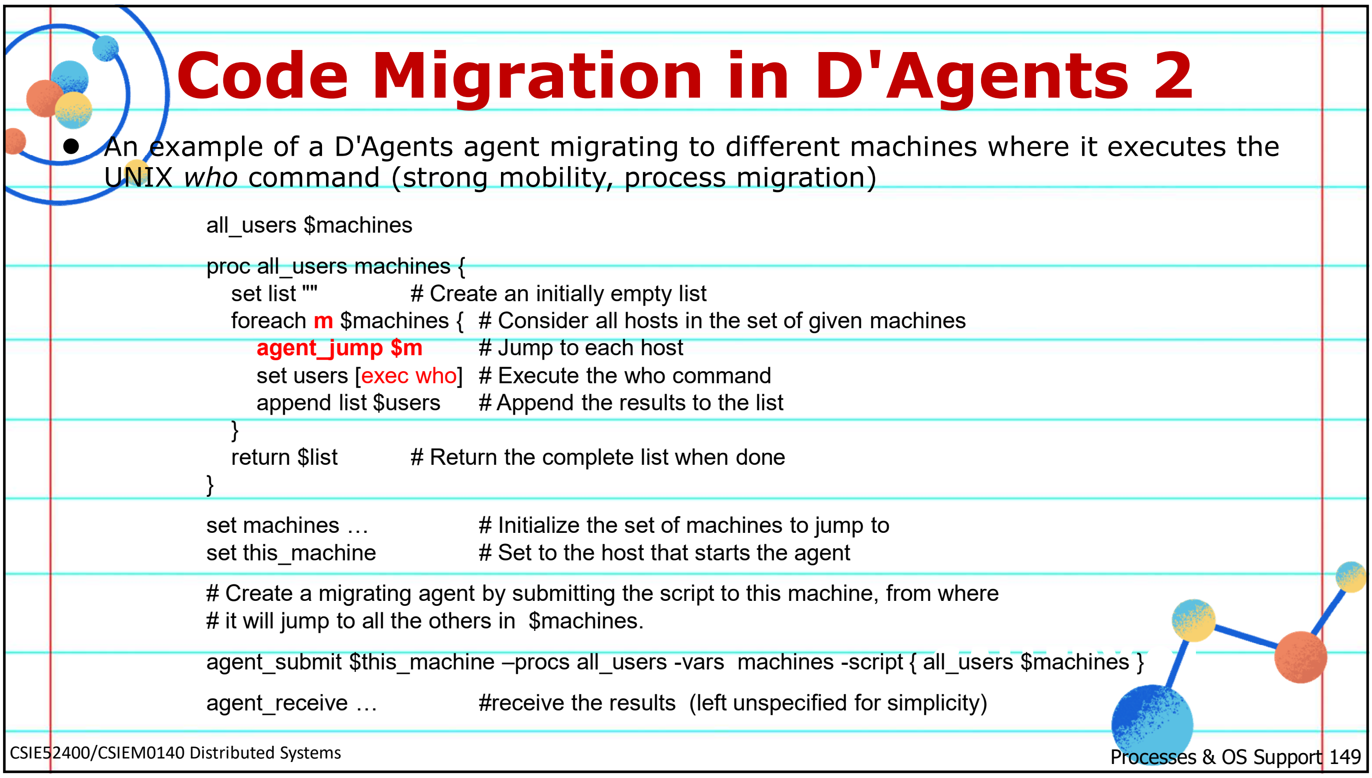
Example: D’Agents

- D’Agents (formally called Agent Tcl) can migrate programs in a heterogeneous system.
- Programs are written in an interpretable language (eg. Tcl, Java, or Scheme).
- Supports three types of mobility:
  - sender-initiated weak mobility
  - strong mobility by process migration
  - strong mobility by process cloning

Code Migration in D'Agents 1

- A simple example of a Tcl agent in D'Agents submitting a script to a remote machine (sender-initiated weak mobility)

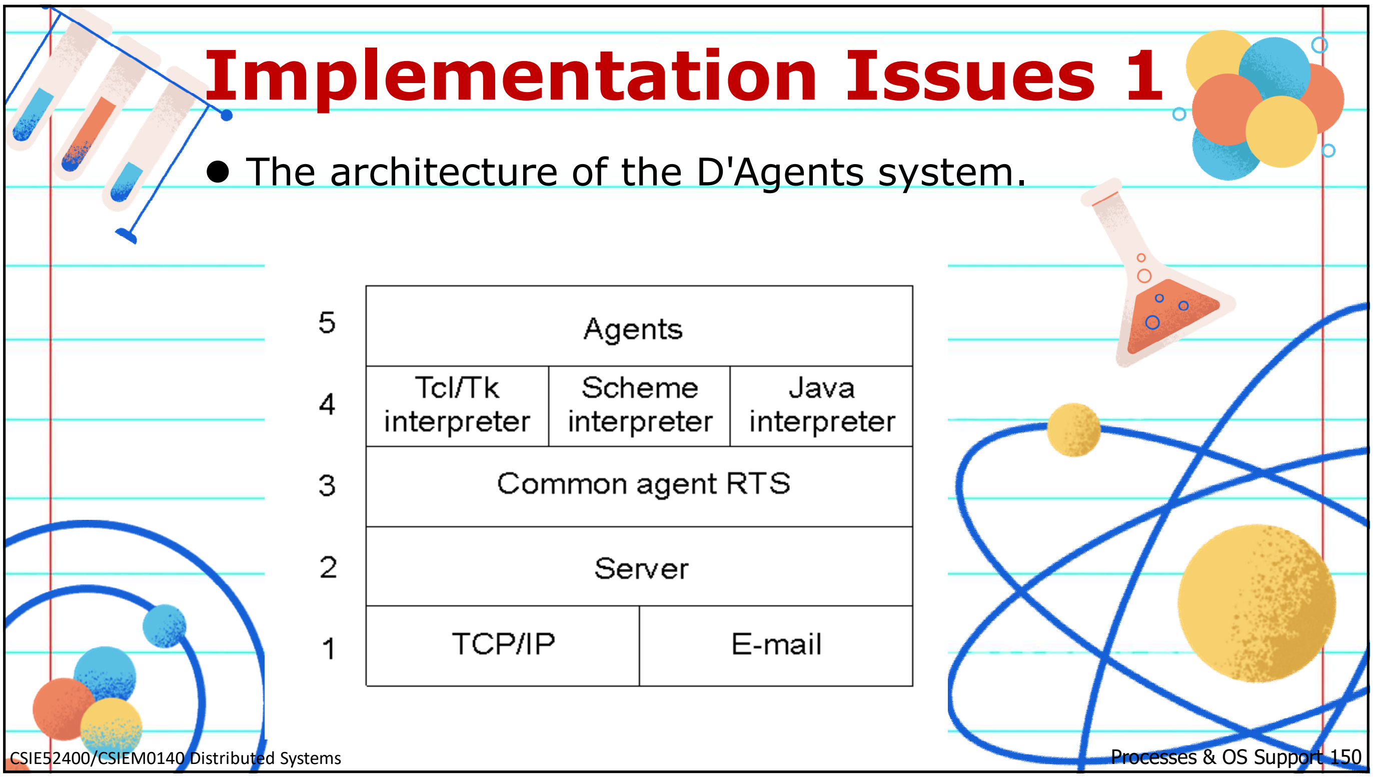
```
proc factorial n {
   if ($n ≤ 1) { return 1; }              # fac(1) = 1
   expr $n * [ factorial [expr $n – 1] ]      # fac(n) = n * fac(n – 1)
}

set number …  # tells which factorial to compute

set machine …       # identify the target machine

agent_submit $machine –procs factorial –vars number –script
{factorial $number }

agent_receive …    # receive the results (left unspecified for simplicity)
```



Code Migration in D'Agents 2

- An example of a D'Agents agent migrating to different machines where it executes the UNIX *who* command (strong mobility, process migration)

```
all_users $machines

proc all_users machines {
    set list ""              # Create an initially empty list
    foreach m $machines {  # Consider all hosts in the set of given machines
        agent_jump $m        # Jump to each host
        set users [exec who]  # Execute the who command
        append list $users    # Append the results to the list
    }
    return $list             # Return the complete list when done
}

set machines …                # Initialize the set of machines to jump to
set this_machine              # Set to the host that starts the agent

# Create a migrating agent by submitting the script to this machine, from where
# it will jump to all the others in  $machines.

agent_submit $this_machine –procs all_users -vars  machines -script { all_users $machines }

agent_receive …               #receive the results  (left unspecified for simplicity)
```



Implementation Issues 1

- The architecture of the D'Agents system.

5	Agents		
4	Tcl/Tk interpreter	Scheme interpreter	Java interpreter
3	Common agent RTS		
2	Server		
1	TCP/IP	E-mail	

Implementation Issues 2

- The parts comprising the state of an agent in D'Agents.

Status	Description
Global interpreter variables	Variables needed by the interpreter of an agent
Global system variables	Return codes, error codes, error strings, etc.
Global program variables	User-defined global variables in a program
Procedure definitions	Definitions of scripts to be executed by an agent
Stack of commands	Stack of commands currently being executed
Stack of call frames	Stack of activation records, one for each running command

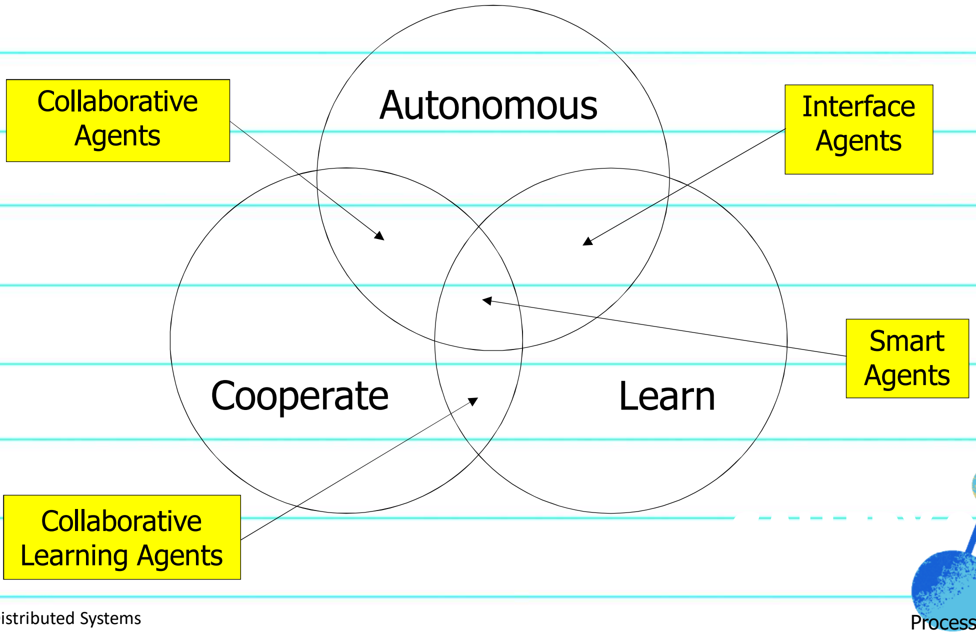
Example: Mobile-C

- IEEE FIPA (Foundation for Intelligent Physical Agents) standard compliant multi-agent platform for supporting C/C++ mobile agents
- Specifically designed for real-time and resource constrained applications with interface to hardware
- Has been ported to Raspberry Pi and ARM based computers
- Hosted in public git repository

Software Agents

- No universal agreement on the definition.
- An autonomous process capable of perceiving, reacting to, and initiating changes in its environment (may collaborate with users or other agents).
- Researchers can’t reach agreement on a single taxonomy either.
- Nevertheless, software agents are playing an increasingly important role in distributed systems.

A Taxonomy by Capabilities



Collaborative Agents

Autonomous

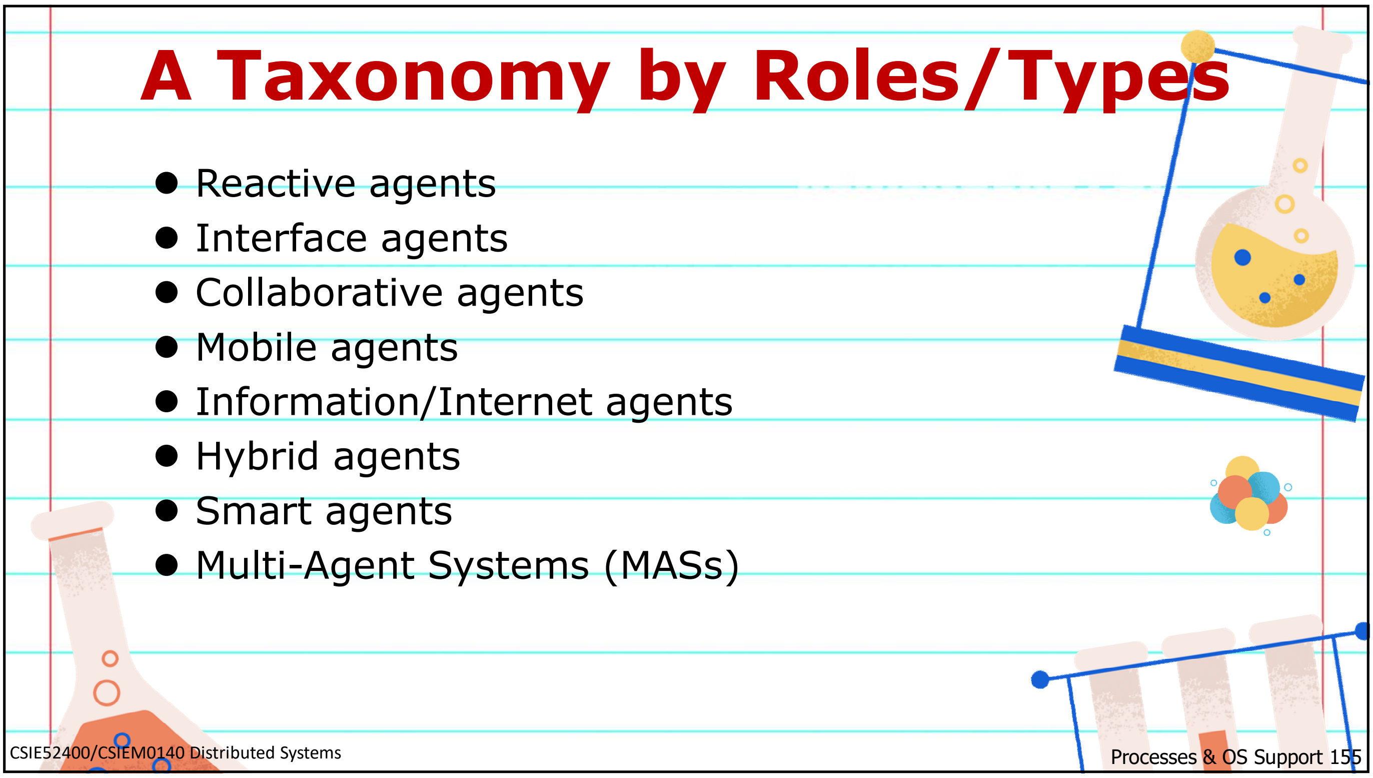
Interface Agents

Smart Agents

Cooperate

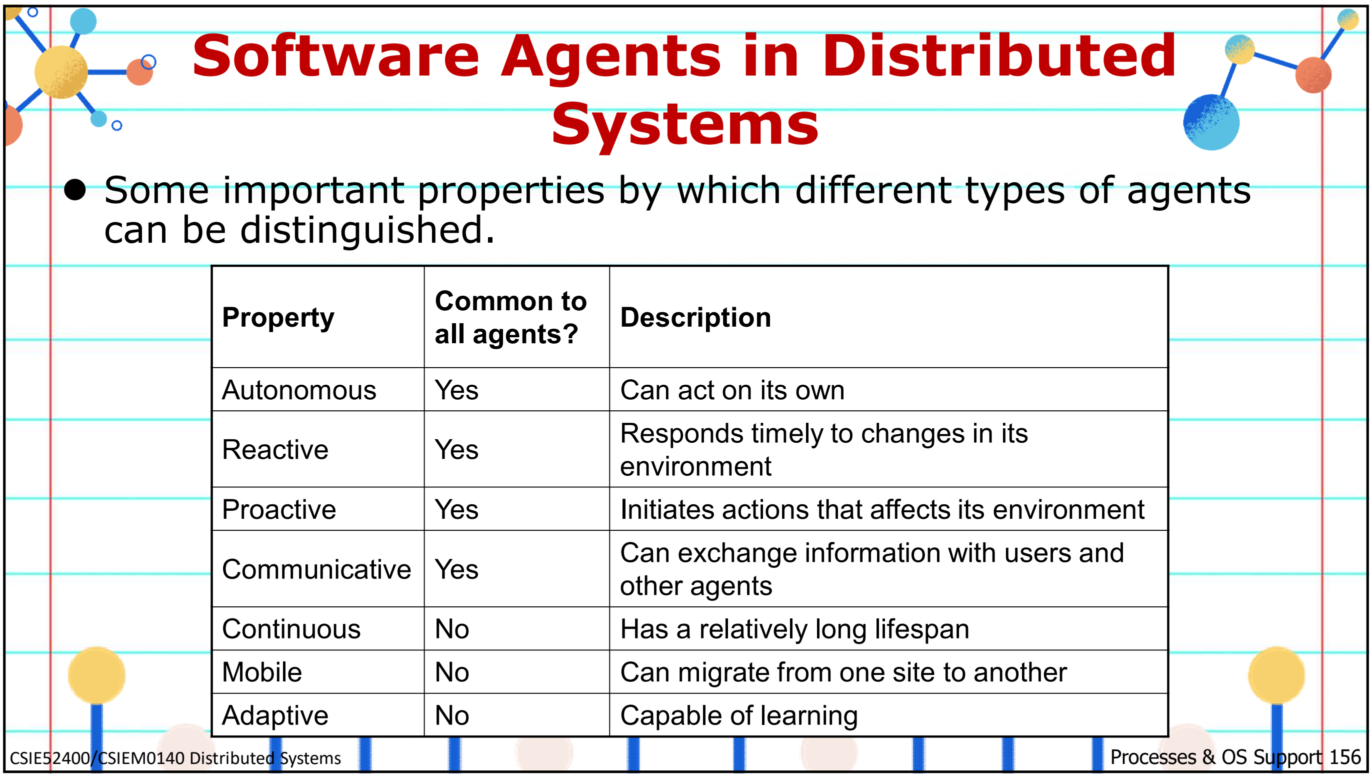
Learn

Collaborative Learning Agents



A Taxonomy by Roles/Types

- Reactive agents
- Interface agents
- Collaborative agents
- Mobile agents
- Information/Internet agents
- Hybrid agents
- Smart agents
- Multi-Agent Systems (MASs)



Software Agents in Distributed Systems

- Some important properties by which different types of agents can be distinguished.

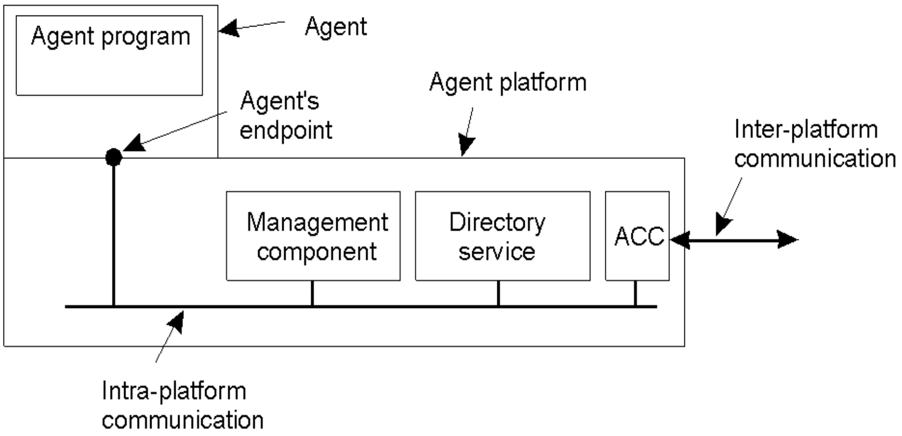
Property	Common to all agents?	Description
Autonomous	Yes	Can act on its own
Reactive	Yes	Responds timely to changes in its environment
Proactive	Yes	Initiates actions that affects its environment
Communicative	Yes	Can exchange information with users and other agents
Continuous	No	Has a relatively long lifespan
Mobile	No	Can migrate from one site to another
Adaptive	No	Capable of learning

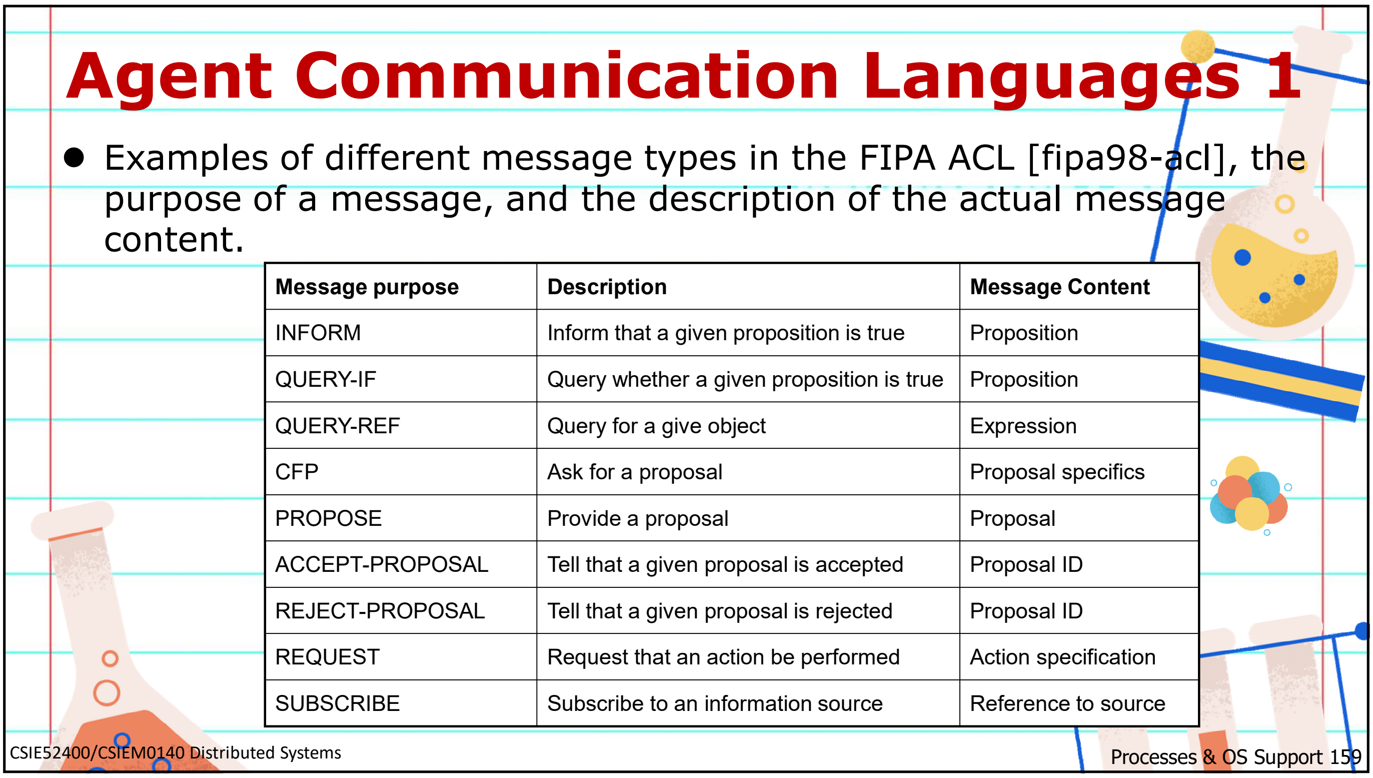
Agent Technology

- The Foundation for Intelligent Physical Agent (FIPA) is developing a general model for software agents.
- In this model, agents are registered at, and operate under the management of an agent platform:
  - **Management**: Keeps track of where the agents are.
    - creating and deleting agents.
    - mapping globally unique agent ID to a local communication endpoint (port)
  - **Directory**: Mapping of agent names and attributes to agent IDs
  - **ACC**: Agent Communication Channel, used to communicate with other platforms
    - Communication between ACCs on different platforms follows Internet Inter-ORB Protocol (IIOP).
    - Example: server in D'Agents

FIPA Agent Platform

- The general model of an agent platform (adapted from [fipa98-mgt]).

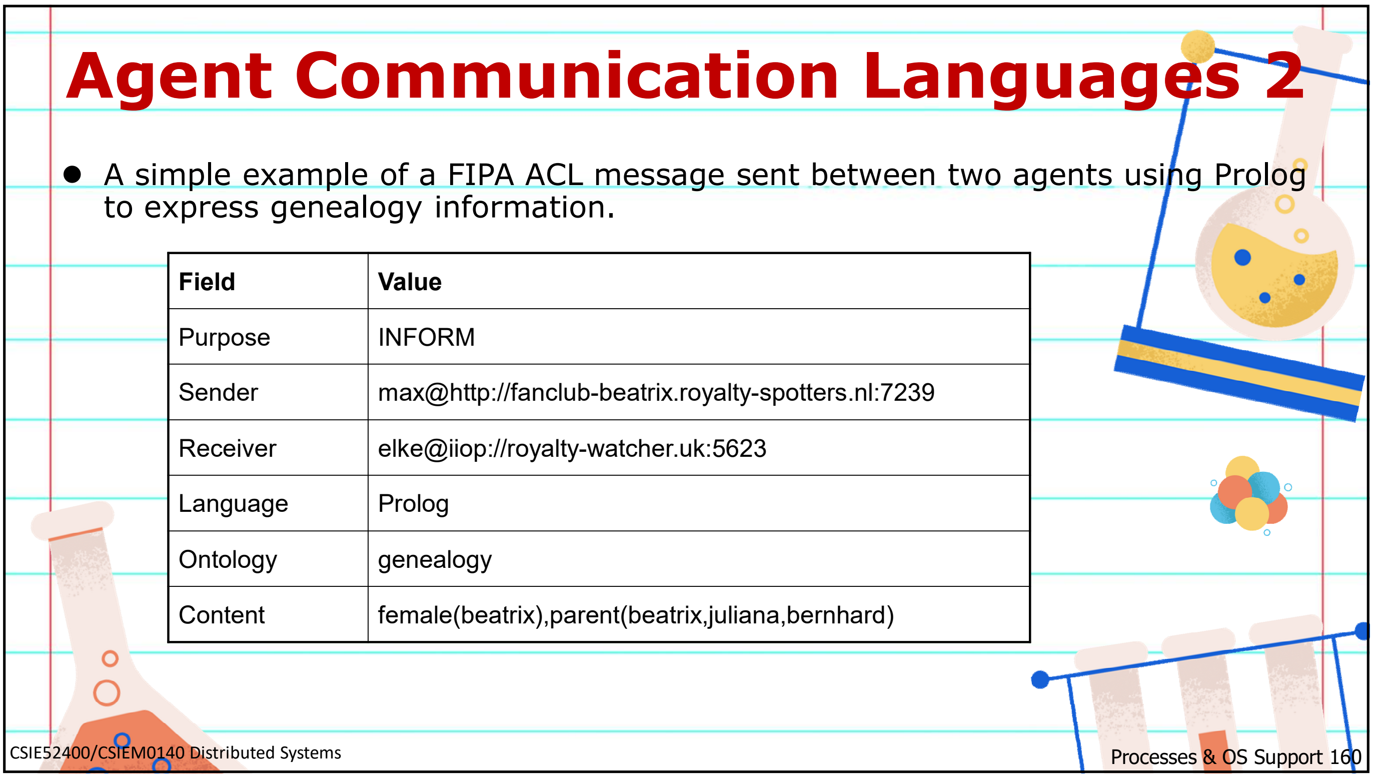




Agent Communication Languages 1

- Examples of different message types in the FIPA ACL [fipa98-acl], the purpose of a message, and the description of the actual message content.

Message purpose	Description	Message Content
INFORM	Inform that a given proposition is true	Proposition
QUERY-IF	Query whether a given proposition is true	Proposition
QUERY-REF	Query for a give object	Expression
CFP	Ask for a proposal	Proposal specifics
PROPOSE	Provide a proposal	Proposal
ACCEPT-PROPOSAL	Tell that a given proposal is accepted	Proposal ID
REJECT-PROPOSAL	Tell that a given proposal is rejected	Proposal ID
REQUEST	Request that an action be performed	Action specification
SUBSCRIBE	Subscribe to an information source	Reference to source



Agent Communication Languages 2

- A simple example of a FIPA ACL message sent between two agents using Prolog to express genealogy information.

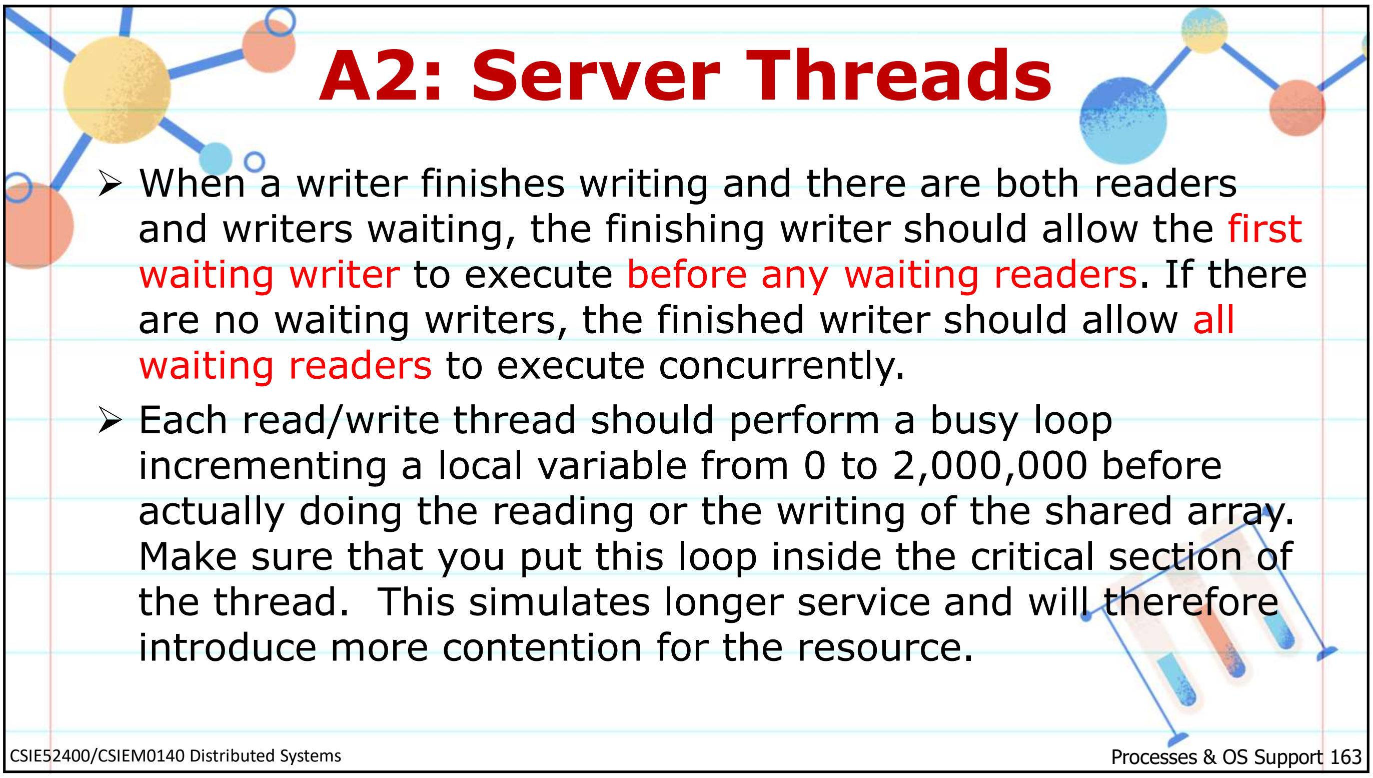
Field	Value
Purpose	INFORM
Sender	max@http://fanclub-beatrix.royalty-spotters.nl:7239
Receiver	elke@iiop://royalty-watcher.uk:5623
Language	Prolog
Ontology	genealogy
Content	female(beatrix),parent(beatrix,juliana,bernhard)

A2: Multi-Threaded Server

- You are to implement a multi-threaded server that provides shared conditional read/write access to an integer array of size 10.
- The server will maintain an array of 10 integers. It will accept two client operations:
  - `read cond` - will return the values of the integers in the array that satisfy the cond (in the format <op> <num> such as "> 10", "% 3")
  - `write num[10]` - will update the values of the integers in the array with the integers in `num`

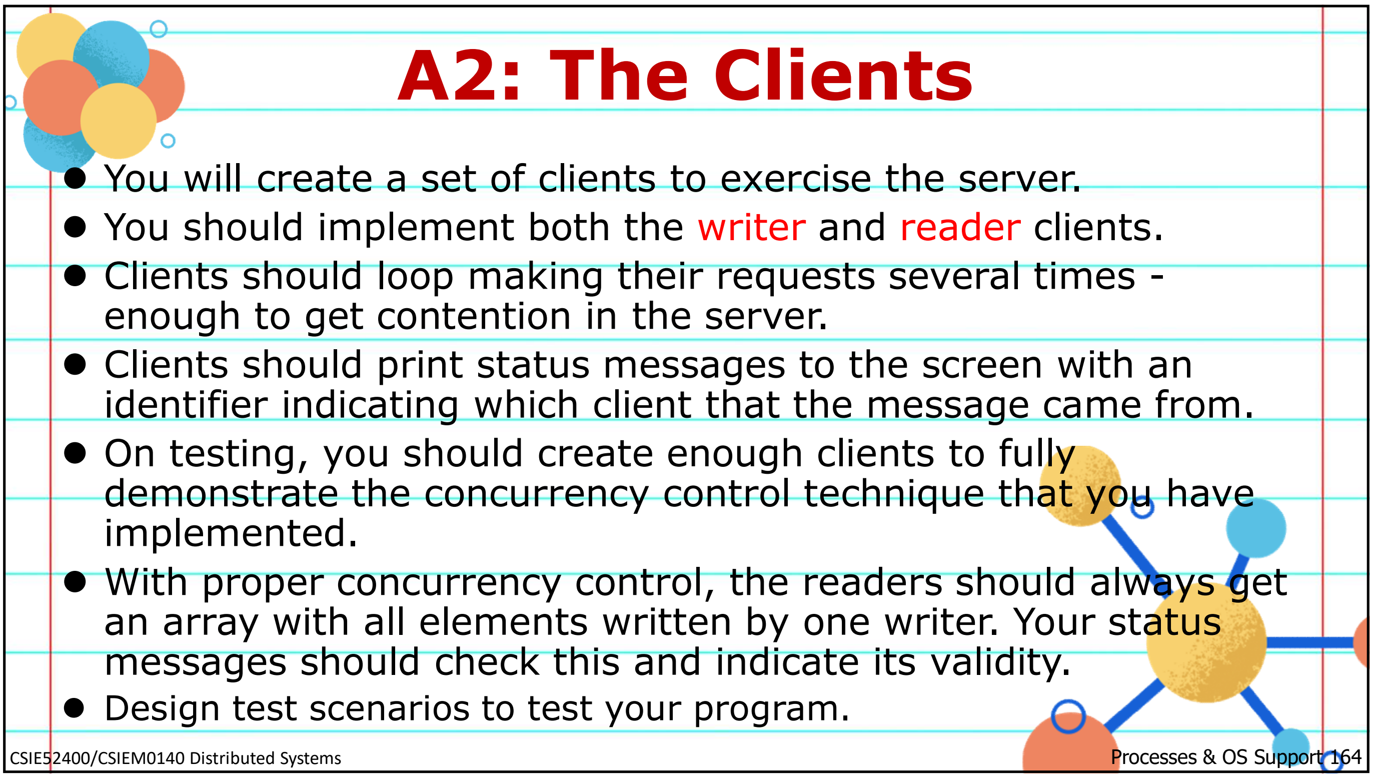
A2: Server Threads

- **Main thread**
  - Receive requests for read or write services.
  - Create a new thread to service each client request, then loop back to handle the next request.
- **Read/Write threads**
  - Communicate with associated clients. Will need the socket after the main thread has accepted the client.
  - Handle **concurrency control**. That is, once created, it is up to the new read or write thread to determine if it is "safe" to perform the operation.
  - Should allow multiple concurrent readers, but exclusive access for writers.



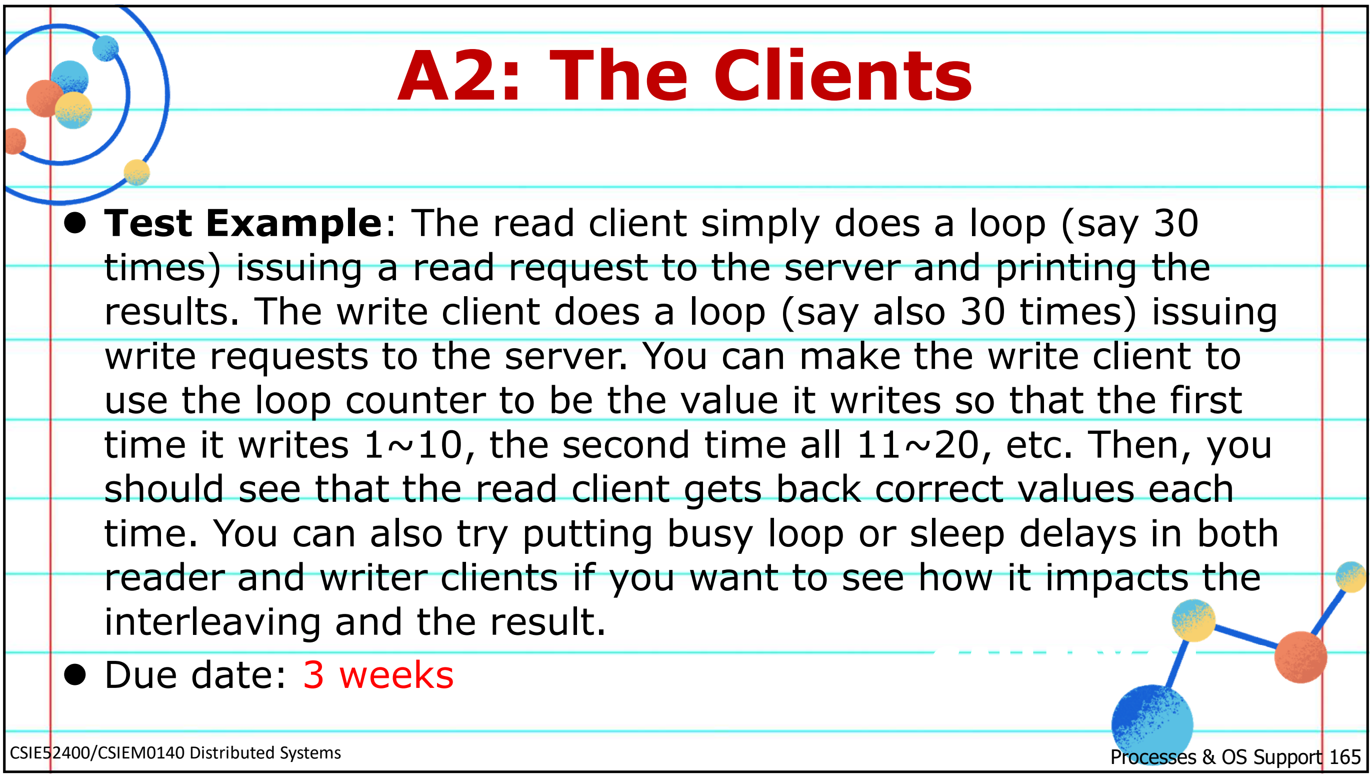
A2: Server Threads

- When a writer finishes writing and there are both readers and writers waiting, the finishing writer should allow the first waiting writer to execute before any waiting readers. If there are no waiting writers, the finished writer should allow all waiting readers to execute concurrently.
- Each read/write thread should perform a busy loop incrementing a local variable from 0 to 2,000,000 before actually doing the reading or the writing of the shared array. Make sure that you put this loop inside the critical section of the thread. This simulates longer service and will therefore introduce more contention for the resource.



A2: The Clients

- You will create a set of clients to exercise the server.
- You should implement both the writer and reader clients.
- Clients should loop making their requests several times - enough to get contention in the server.
- Clients should print status messages to the screen with an identifier indicating which client that the message came from.
- On testing, you should create enough clients to fully demonstrate the concurrency control technique that you have implemented.
- With proper concurrency control, the readers should always get an array with all elements written by one writer. Your status messages should check this and indicate its validity.
- Design test scenarios to test your program.



A2: The Clients

- **Test Example**: The read client simply does a loop (say 30 times) issuing a read request to the server and printing the results. The write client does a loop (say also 30 times) issuing write requests to the server. You can make the write client to use the loop counter to be the value it writes so that the first time it writes 1~10, the second time all 11~20, etc. Then, you should see that the read client gets back correct values each time. You can also try putting busy loop or sleep delays in both reader and writer clients if you want to see how it impacts the interleaving and the result.
- Due date: 3 weeks