Parallelism

Master 1 International & Data Science

Andrea G. B. Tettamanzi
Université de Nice Sophia Antipolis
Département Informatique
andrea.tettamanzi@unice.fr

Lecture 1

Introduction

About This Class

Web Page:

- http://www.i3s.unice.fr/~tettaman/Classes/ConcPar

Workload:

- 6 ECTS

Grading:

- Written Intermediate Test (weight: 30%)
- Written Final Test (weight: 30%)
- Assignments and Lab Work (weight: 40%)

Aims

Familiarize ourselves with the main concepts and techniques of:

- Concurrency
  - Multithreading
  - Concurrent programming
- Parallelism
  - Designing massively parallel algorithms
  - Parallel programming
- Distribution
  - Designing distributed systems
  - Distributed programming

What is Concurrency?

- One physical machine (single- or multi-core)
- Multiple processes (or execution threads)
  - Single-core CPU → time sharing
  - Multi-core CPU → simultaneous execution
- Designing and implementing concurrent software systems
- Concurrency = property of a system in which computations are
  - executing simultaneously
  - potentially interacting with each other
- Mathematical models
  - Petri Nets
  - Process calculi

What is Parallelism?

- Also known as “Parallel Computing”
- Parallel and Massively parallel machines
  - Multiple (= potentially a large number of) processing units
  - Communication via dedicated high-speed bus/network
  - Shared memory
- Design and implementation of algorithms
  - Decompose large problems into smaller ones
  - Solve them in parallel
  - Minimize communication
- Parallel Programming Languages
- Aim: obtain maximum speed-up

What is Distribution?

- Also known as “Distributed Computing”
- Distributed systems:
  - components physically located on distinct machines
  - connected through a network (usually the Internet)
  - often (but not necessarily) in geographically distinct locations
- Distributed programming
- Problems:
  - Communication
  - Synchronization
  - Architectures (layered, data-centered, event-based, P2P)
  - Consistency and Replication, Security

Common Background

- Three slightly different programming contexts
- Need to abandon programming determinism
- Address problems such as
  - Synchronization
  - Latency
  - Indeterministic execution order
  - Consistency
  - Scalability and Speed-Up, etc.
- Principled approach to programming
  - Develop and use appropriate theoretical models
- Need for specific languages, libraries, and frameworks.

History and Motivation

- Recurring Cycles:
  - The first computers were strictly sequential
  - Then came multi-tasking OSs, and concurrent programming
  - Supercomputers and parallel programming
  - Clusters, the Internet, and the Web: distributed programming
  - GPUs: parallel programming strikes back
- Why are Concurrency, Parallelism, and Distribution important?
  - Inherent speed limits of sequential processors
  - Many interesting problems are hard
  - Need to harness the power of parallel and networked h/w
  - Big data

Plan

1) Introduction [conc + distr + par]
   Processes and Threads [conc]
2) Communication [conc + distr]
3) Parallel Architectures [par]
   Describing Concurrent and Parallel Algorithms [conc + par]
4) Theoretical Models [par]
5) Languages and libraries [distr + par]
   Throughput-Oriented Architectures [par]

---

6) Distributed architectures [distr]
   Synchronization [distr]
7) Distributed Computing and DB Systems for the Big Data [distr]
   Consistency and replication [distr]

Bibliography

- Mordechai Ben-Ari. *Principles of Concurrent and Distributed Programming (2nd Edition)*. Addison-Wesley, 2006.
- Andrew S. Tannenbaum and Maarten van Steen. *Distributed Systems: Principles and Paradigms (2nd Edition)*. Prentice Hall, 2007.
- Ananth Grama, George Karypis, Vipin Kumar, and Anshul Gupta. *Introduction to Parallel Computing (2nd Edition)*. Addison-Wesley, 2003.

Lecture 1

Processes and Threads

Introduction to Threads

Basic idea

We build virtual processors in software, on top of physical processors:

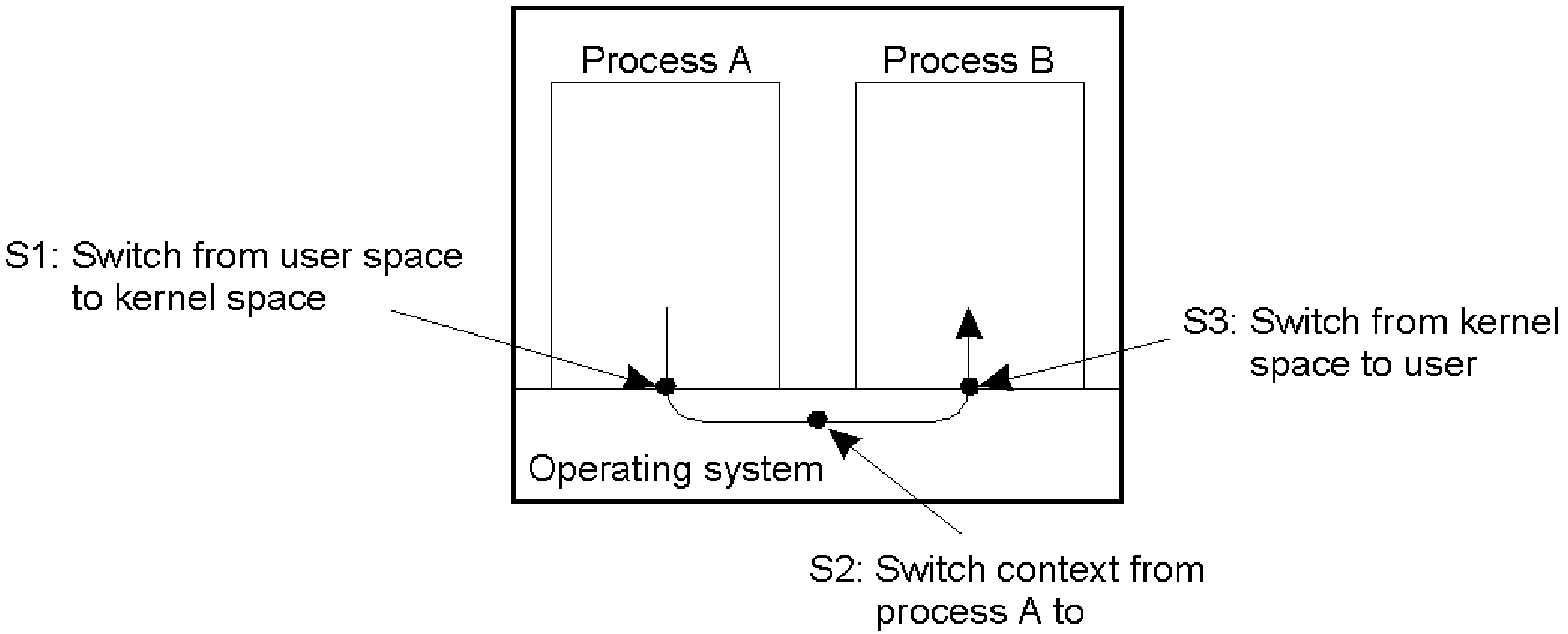
- Processor: Provides a set of instructions along with the capability of automatically executing a series of those instructions.
- Thread: A minimal software processor in whose context a series of instructions can be executed. Saving a thread context implies stopping the current execution and saving 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.

Context Switching

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).

Thread Usage in Nondistributed Systems



Context switching as the result of IPC

Context Switching : Observations

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

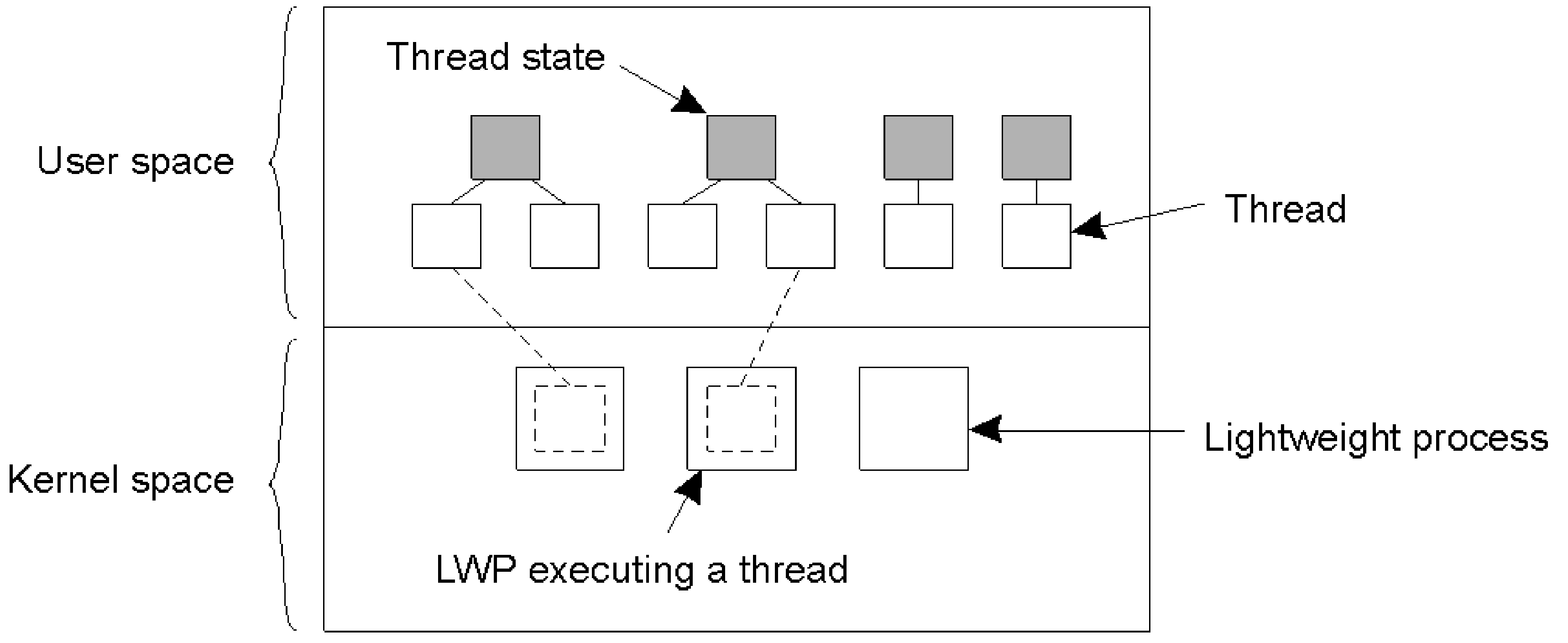
Threads and Operating Systems

- Main issue : Should an OS kernel provide threads, or should they be implemented as user-level packages?
- User-space solution
  - All operations can be completely handled within a single process ⇒ implementations can be extremely efficient.
  - All services provided by the kernel are done on behalf of the process in which a thread resides ⇒ if the kernel decides to block a thread, the entire process will be blocked.
  - Threads are used when there are lots of external events: threads block on a per-event basis ⇒ if the kernel can't distinguish threads, how can it support signaling events to them?

Threads and Operating Systems

- Kernel solution : The whole idea is to have the kernel contain the implementation of a thread package. This means that 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 big problem is the loss of efficiency due to the fact that each thread operation requires a trap to the kernel.
- Conclusion: Try to mix user-level and kernel-level threads into a single concept.

Solaris Threads



Combining kernel-level lightweight processes and user-level threads.

Solaris Thread Operation

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

- **Note:** UNIX System V and its modern derivatives IRIX, SCO OpenServer, HP-UX and IBM AIX allow a many-to-many mapping between user threads and LWPs.

Java Threads

- The Java platform supports concurrent programming natively
- Since v. 5.0, it includes high-level concurrency APIs
  - Package: `java.util.concurrent`
- Basic concurrency support:
  - The Thread class:
    - A constructor which takes a runnable object
    - Methods: start(), interrupt(), join()
  - The Runnable interface: void run();
- Two “idioms” to create a new thread:
  - Call the constructor while providing a runnable object;
  - Subclass Thread and override run() – simpler but less general

Sleeping, Yielding and Interrupts

- Static method Thread.sleep() pauses the calling thread
- Method t.join() waits for thread t to terminate
- A thread is interrupted by a call to its interrupt() method
- InterruptedException is thrown by sleep(), join() if interrupted
- Method interrupted() checks if an interrupt has been received
- Static method Thread.yield() yields processor use to scheduler

Synchronization

- Two basic synchronization idioms:
  - Synchronized methods
  - Synchronized statements
- Monitor lock: every object has a monitor
  - A thread that needs exclusive access acquires the monitor
  - Requests queued and the first executed on monitor release
- A method may be declared as “synchronized”
  - The calling thread automatically acquires the monitor
- A code block may acquire the monitor of object o with construct

```
synchronized(o) { <statement> }
```

- More on this subject in a forthcoming lecture...

Threads in Distributed Systems

Multithreaded Web client

Hiding network latencies:

- Web browser scans an incoming HTML page, and finds that more files need to be fetched.
- Each file is fetched by a separate thread, each doing a (blocking) HTTP request.
- As files come in, the browser displays them.

Multiple request-response calls to other machines (RPC)

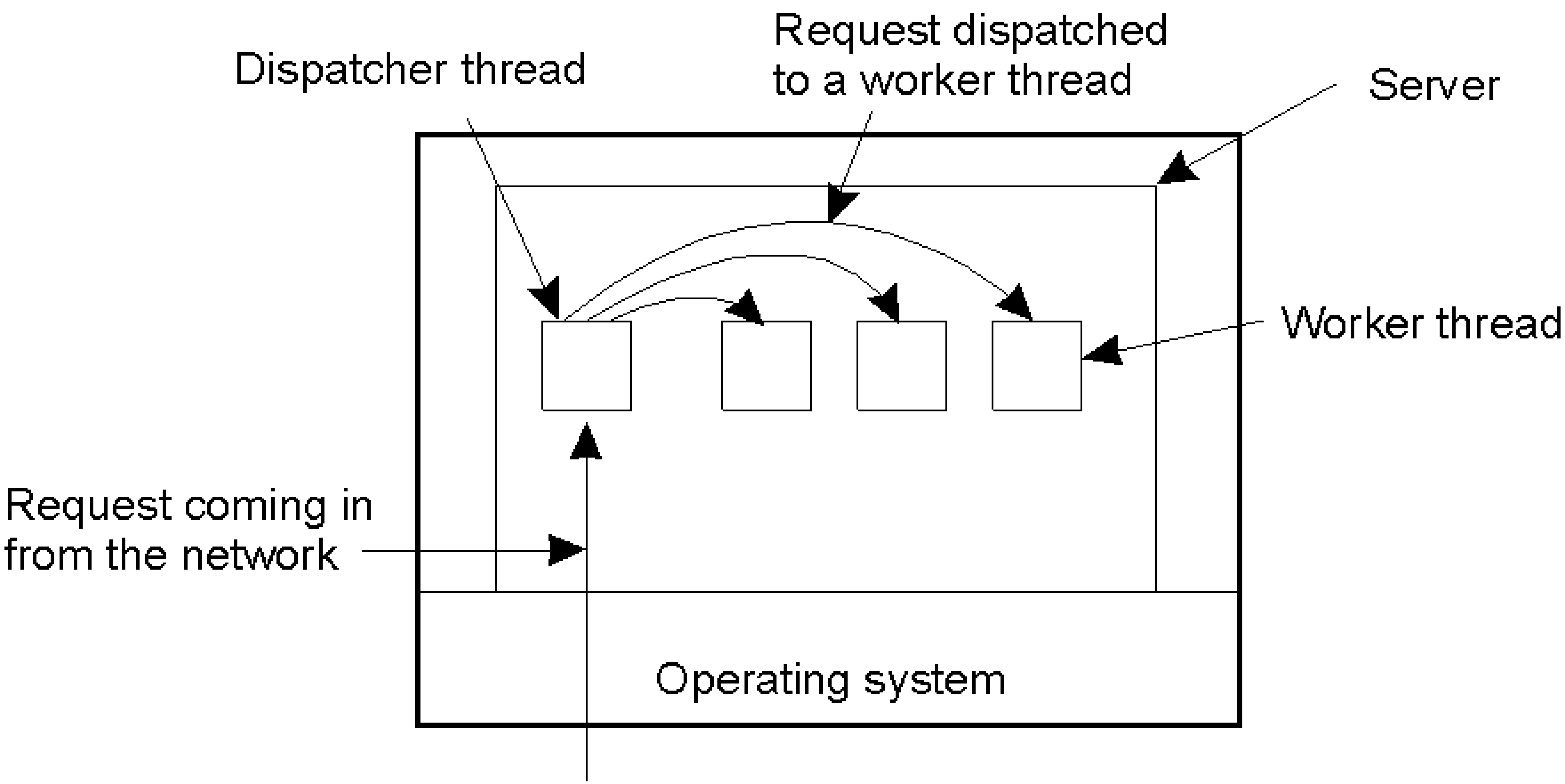
- A client does several calls at the same time, each one by a different thread.
- It then waits until all results have been returned.

Note: if calls are to different servers, we may have linear speed-up.

Threads in Distributed Systems

- Improve performance
  - Starting a thread is much cheaper than starting a new process.
  - Having a single-threaded server prohibits 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 overall structure.
  - Multithreaded programs tend to be smaller and easier to understand due to simplified flow of control.

Multithreaded Servers (1)



A multithreaded server organized in a dispatcher/worker model.

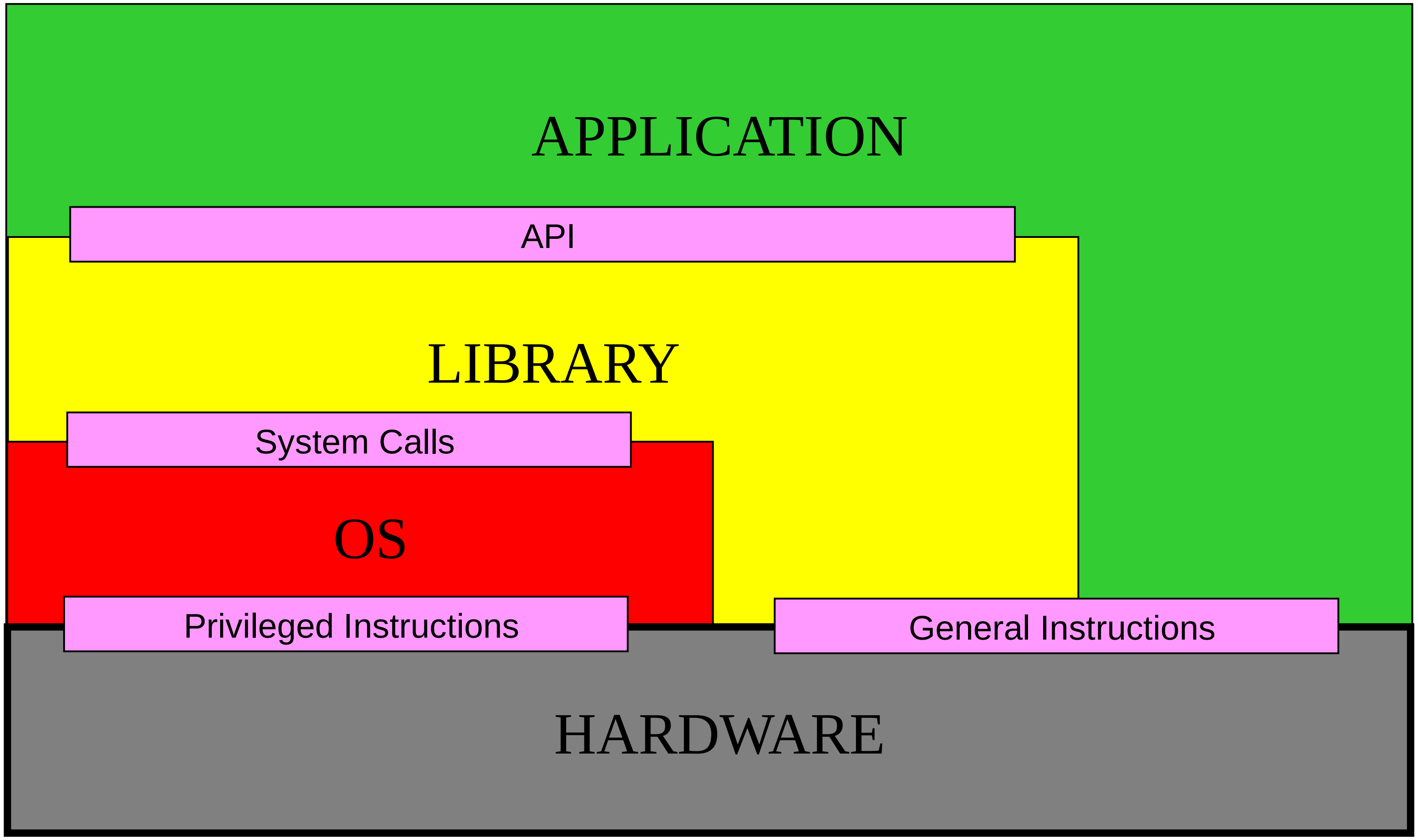
Three Ways to Program a Server

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

Virtualization

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

Architecture of Virtual Machines



Types of Virtual Machines

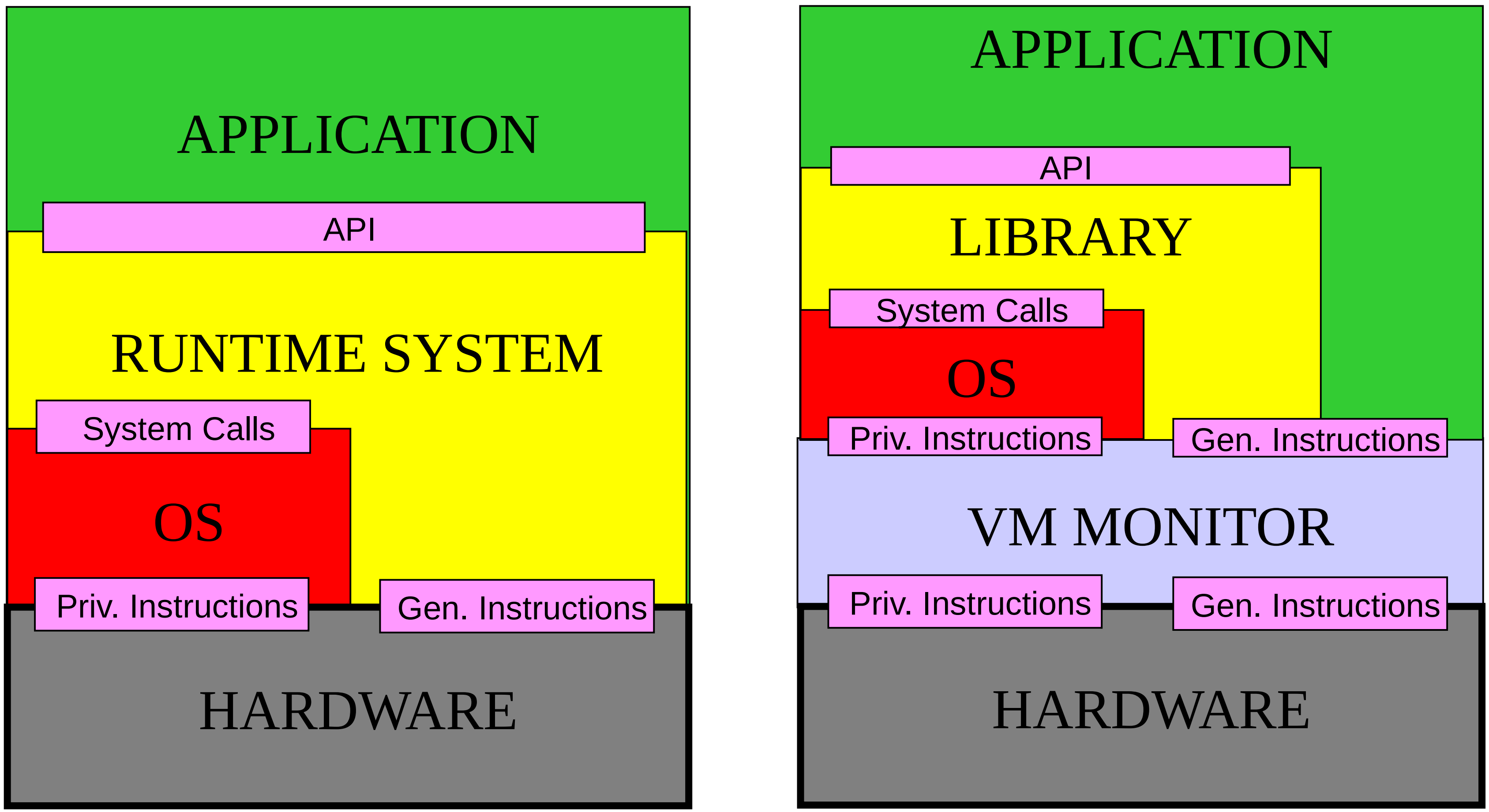
Process Virtual Machine

- One VM per process
- Runtime system
- Interpreted or emulated instructions

Virtual Machine Monitor

- One VM for more processes
- Layer that completely encapsulates the original h/w
- Interface to a virtual h/w

Process VMs vs. VM Monitors

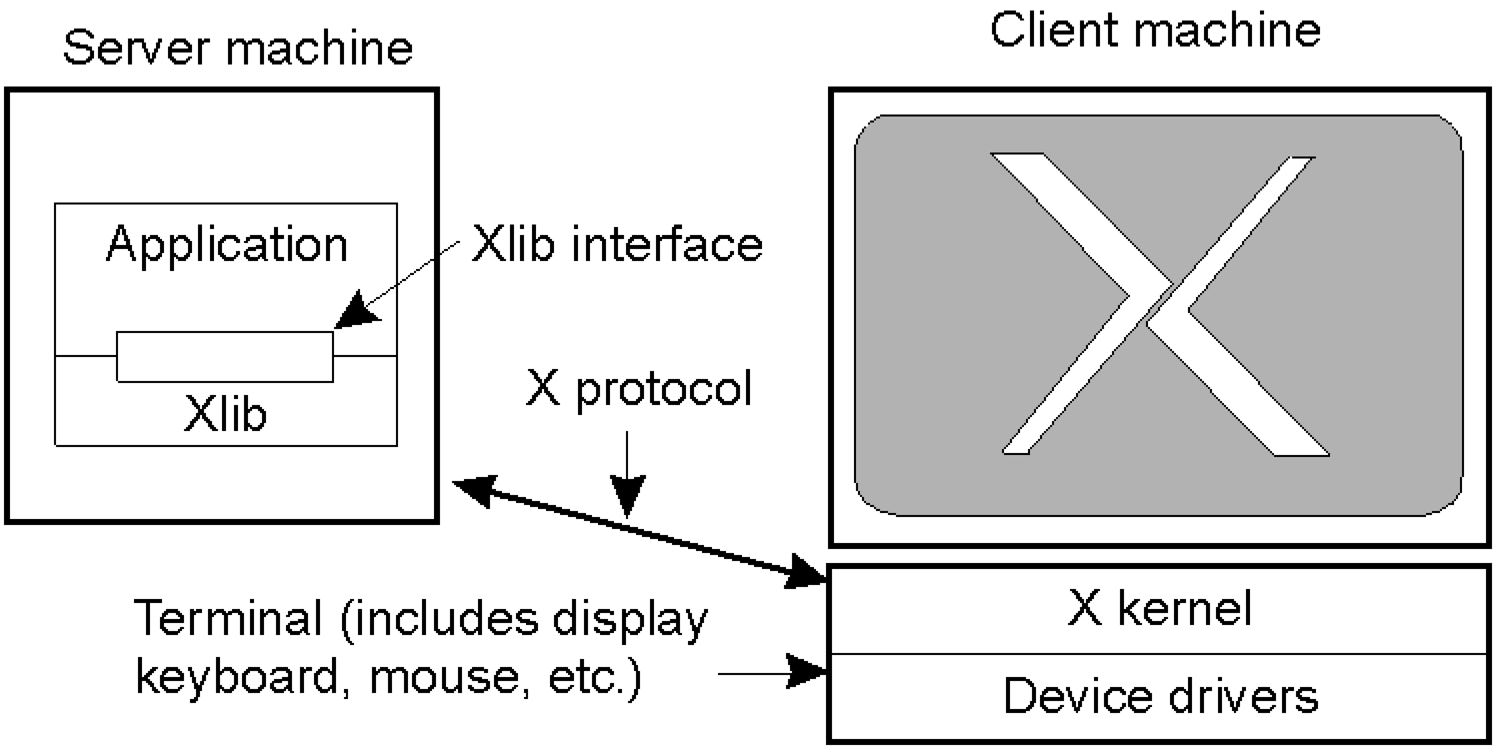


VM Monitors on Operating Systems

We're seeing VMMs run on top of existing operating systems.

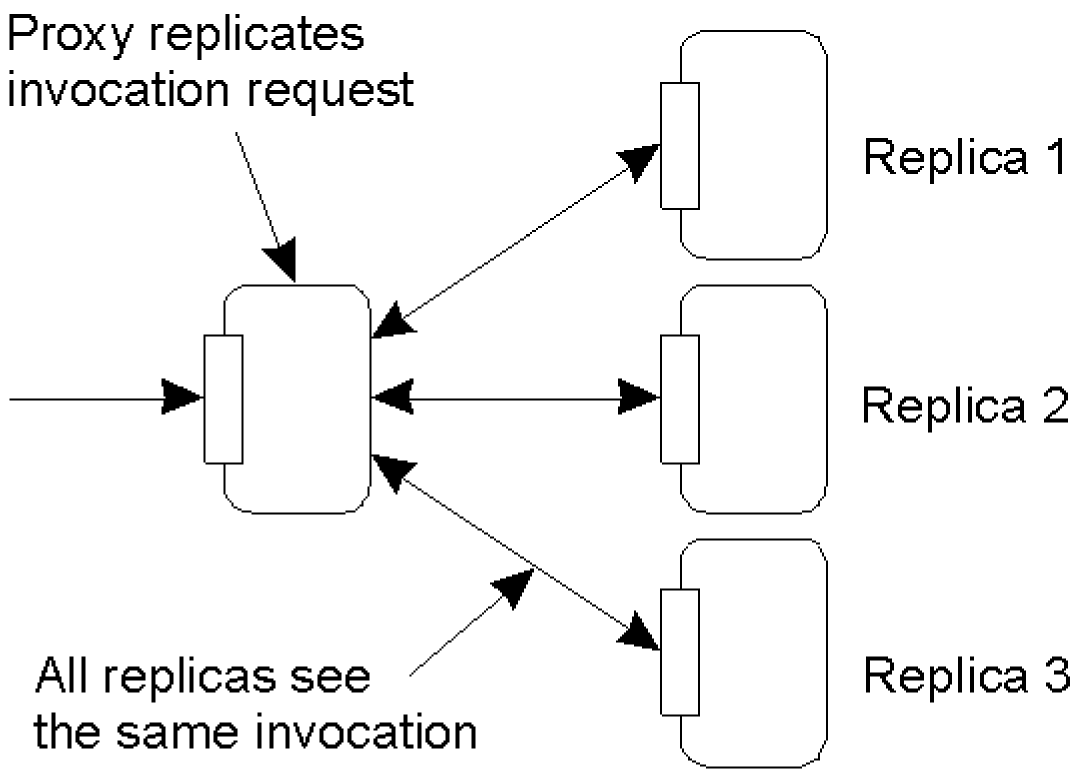
- Perform binary translation: while executing an application or operating system, translate instructions to that of the underlying machine.
- Distinguish sensitive instructions: traps to the original kernel (think of system calls, or privileged instructions).
- Sensitive instructions are replaced with calls to the VMM.

Clients: User Interfaces



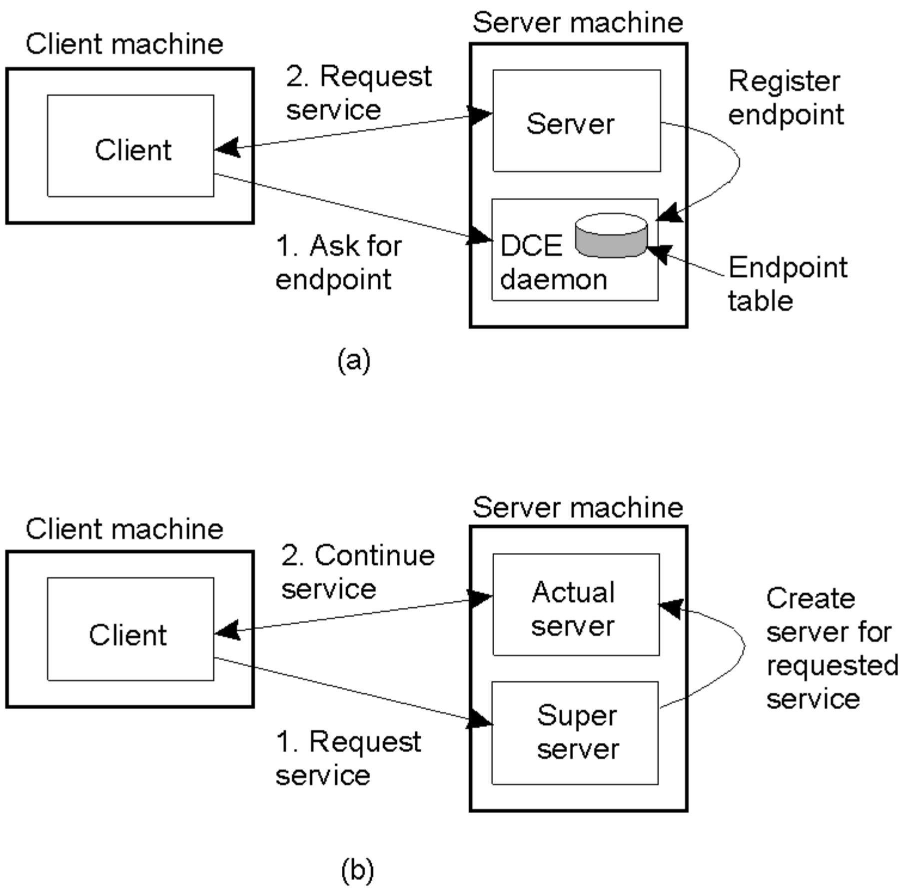
The basic organization of the X Window System

Client-Side Software for Distribution Transparency



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

Servers: General Design Issues



a) Client-to-server binding using a daemon as in DCE

b) Client-to-server binding using a superserver as in UNIX

Out-of-Band Communication

Issue: Is it possible to interrupt a server once it has accepted (or is in the process of accepting) a service request?

- Solution 1: Use a separate port for urgent data:
  - Server has a separate thread/process for urgent messages
  - Urgent message comes in ⇒ associated request put on hold
  - Note: we require OS supports priority-based scheduling
- Solution 2: Use out-of-band communication facilities of the transport layer:
  - Example: TCP allows for urgent messages in same connection
  - Urgent messages can be caught using OS signaling techniques

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: Keep track of the status of their clients:

- Record that a file has been opened, so that prefetching can be done
- Know which data a client has cached, and allow 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 not a major problem.

Thank you for your attention