Chapter 3

Processes

October, 1st

Introduction

- The notion of process is fundamental
- Originally introduced in operating systems
- “A process is a program in execution”
- Processes and threads
- Processes vs processors
- Related concepts: virtualization

Processes and Threads

Processes

Introduction to processes

- Virtual processors created by the OS
- Process table (per virtual processor)
  - Process context: CPU register values, memory maps, open files, ...
- Process is a program executed by a virtual processor
- Concurrent processes are *isolated* from each other
  - Special hardware enabling protection
  - ✓ Sharing of CPU and hardware resources (RAM) is transparent
  - ✗ High performance price

Processes

Introduction to processes

The price of process concurrency

- Creating a new process
  - OS must create an independent address space
- Switching between two processes
  - Save CPU context (registers, program counter, stack pointer, ...)
  - Modify registers of the memory management unit (MMU)
  - Invalidate translation lookaside buffer (TLB)
  - Maybe swap processes between main memory and disk

Threads

Introduction to threads

The consequences of thread concurrency:

- Less transparency between threads (than between processes)
- Thread context
  - CPU context and some other management data (**not memory**)
  - e.g., mutex variables
- Implications
  - ✓ multithreaded applications are highly efficient
  - ✗ more difficult to program *correctly* and *securely*

Threads

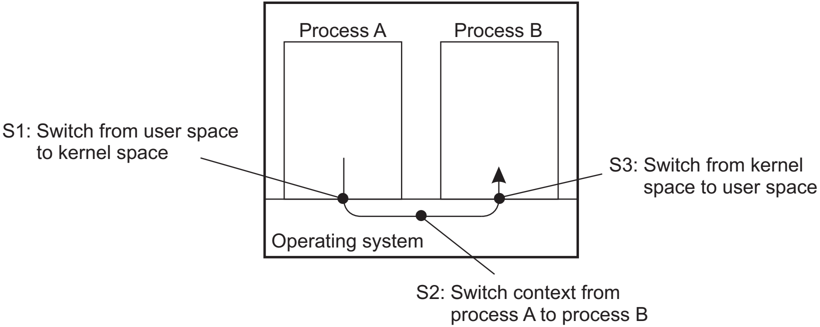
Threads usage in “non-distributed systems”

- ✓ Better performance from non-blocking requests
  - Reading from I/O and computing
- ✓ Better performance in parallel systems
  - Each thread is assigned to a processor
- ✓ Some applications are simpler to program
  - e.g., word processor (Microsoft Word):
    - thread for handling input
    - thread for spellchecking
    - thread for document layout
  - e.g., integrated development environment (VSCode):
    - thread for handling input
    - thread for analyzing the source code
    - thread for highlighting
    - thread for executing tests

Threads

Threads usage in “non-distributed systems”

- Threads reduce the number of context switches in complex applications using IPC (e.g., pipes, message queues)
- Multithread communication by means of shared memory



Context switching as the result of IPC

Threads

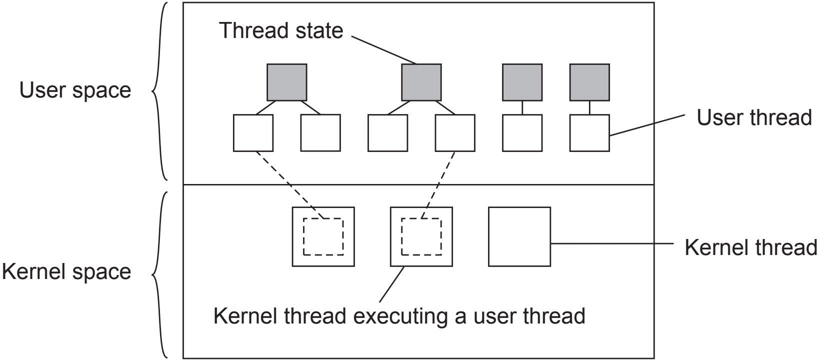
Thread implementation

- User-level threads: threads exist in user space only
  - ✓ Creating and destroying threads is cheap
  - ✓ Fast thread switching (only CPU registers)
  - ✗ **Process blocks on blocking system calls**
  - e.g., GNU Portable Threads
- Operating system threads
  - More similar to processes
  - Native POSIX Thread Library (NPTL) for Linux (POSIX Threads)

Threads

Thread implementation

- Combining user-level and kernel-level threads
  - Kernel-level threads execute user-level threads



Threads

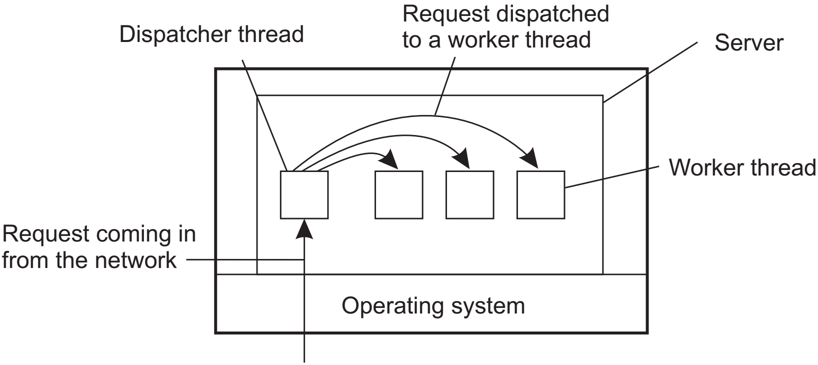
Threads in distributed systems

- Main advantages:
  - ✓ not blocking on *communication* calls
  - ✓ exploiting parallelism in multi-processor systems
- Multithreaded clients
  - hide large latencies in wide-area communication
  - e.g., web browser:
    - display layout before fetching media
    - fetch multiple objects at the same time
- Multithreaded servers...

Threads

Multithreaded servers

- Simplifies server design
- Easier to develop high performance servers (concurrency)

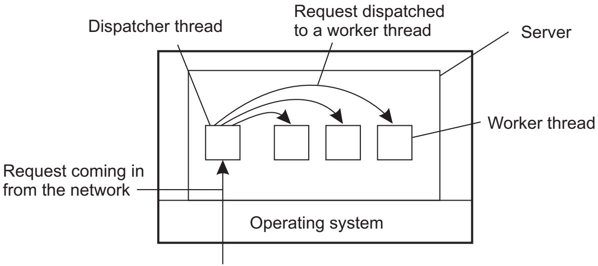


A multithreaded server organized in a dispatcher/worker model

Multithreaded servers

Dispatcher/worker model workflow

- **Dispatcher** (one thread) reads incoming requests for the service, examines the request and chooses an idle worker thread
- **Worker** threads execute requests and reply to the clients
- Worker thread blocks $\Rightarrow$ others workers continue to execute
- How to implement the server without threads?



Threads

Multithreaded servers

How to implement the server without threads?

- Single-thread model
    - One thread receives requests, executes them and replies
    - Sequential processing
- Finite-state machine model (simplified by async programming)
    - One thread executes requests and replies, if execution is quick
    - Blocking system calls are replaced by non-blocking calls
    - Multiple requests are handled simultaneously

Multithreaded servers

How to implement the server without threads?

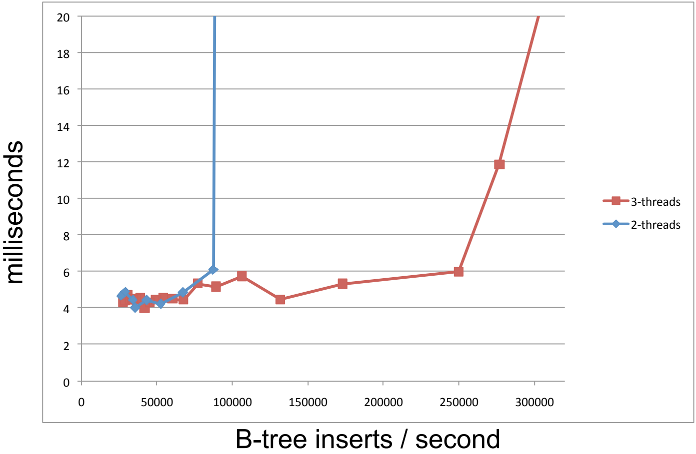
- Single-thread model
  - One thread receives requests, executes them and replies
  - Sequential processing
- Finite-state machine model (simplified by async programming)
  - One thread executes requests and replies, if execution is quick
  - Blocking system calls are replaced by non-blocking calls
  - Multiple requests are handled simultaneously

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

Three ways to construct a server

Threads

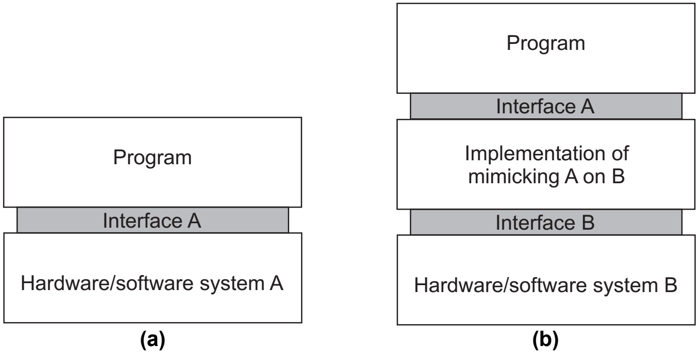
Multithreaded servers



Virtualization

The role of virtualization in distributed systems

- Resource virtualization (not only CPU)
- Virtualization extends or replaces existing interfaces
  e.g., to mimic the behavior of other systems



(a) General organization between a program, interface, and system.
(b) General organization of virtualizing system A on top of system B.

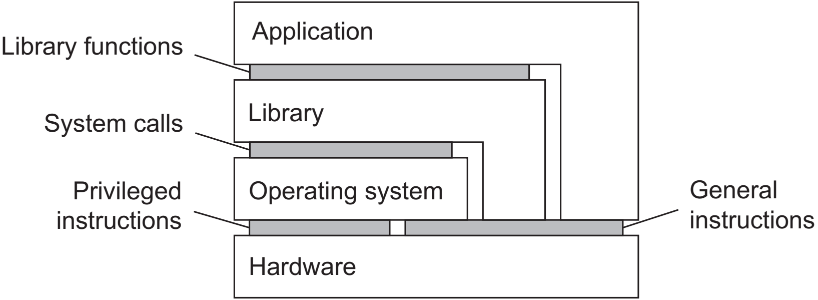
The role of virtualization in distributed systems

Reasons for virtualization

- Allow legacy software to run on expensive mainframe hardware (e.g., IBM 370 in the 1970s)
- Porting legacy interfaces to new platforms
  e.g., compatibility in Windows
- Make hardware platforms *appear* more homogeneous
- Security (by isolating systems running on the same servers)

Architectures of virtual machines

Interfaces offered by computer systems



Various interfaces offered by computer systems

Architectures of virtual machines

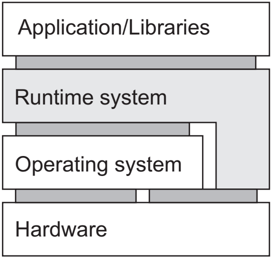
Interfaces offered by computer systems

- hardware/software: CPU instructions invokable by any program
- hardware/software: CPU instructions invokable only by privileged programs (e.g., operating systems)
- operating system/software: system calls
- library/software: functions generally forming what is known as an application programming interface (API)

How to implement virtualization

Run-time system with abstract instruction set

- Process virtual machine
- Instructions are interpreted (e.g., Java) or emulated (e.g., Windows on Unix)

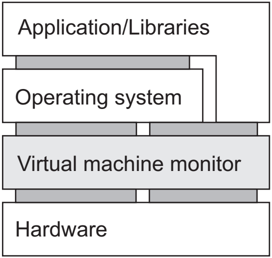


A process virtual machine

How to implement virtualization

Layer shielding original hardware

- Virtual machine monitor (VMM), (e.g., VMware)
- Offer complete set of instructions of same or different hardware

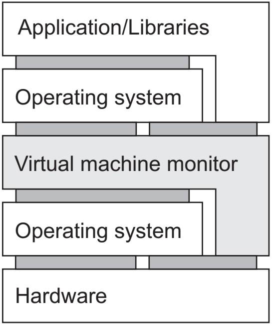


A virtual machine monitor

How to implement virtualization

A hosted virtual machine monitor

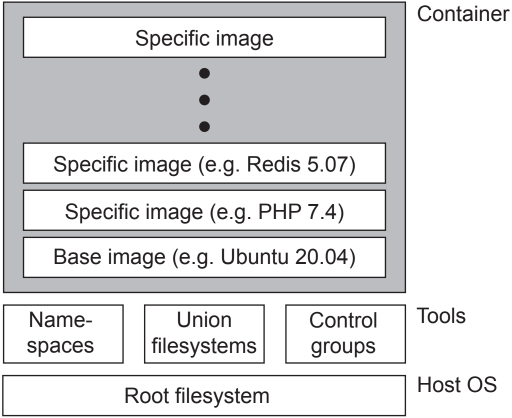
- Virtual machine monitor (VMM), (e.g., VirtualBox)
- Uses interfaces provided by the host operating system



A (hosted) virtual machine monitor

Containers

- applications stable in operating system and CPU architecture
- need *isolated software environment*



Structure of a typical running container

Clients and Servers

Clients

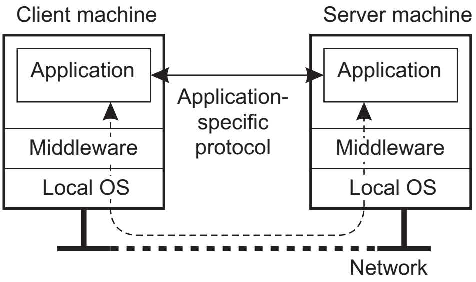
Networked user interfaces

- Client machines
  - Provide the means for users to *interact with remote servers*
- Application-specific protocol
  - For each remote service, the client machine has a separate counterpart that contacts the service over the network
- Application-independent protocol
  - Direct access to remote services through a convenient interface
  - Client machine used only as a terminal, no local state

Clients

Application-specific protocol

- Some application state resides on the client machine
- e.g., calendar app synchronizes with a remote agenda

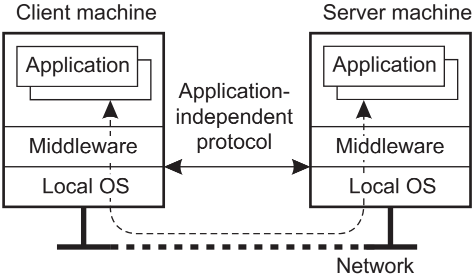


A networked application with its own protocol

Clients

Application-independent protocol

- Thin-client approach
- Everything processed and stored on the server



A general solution to allow access to remote applications

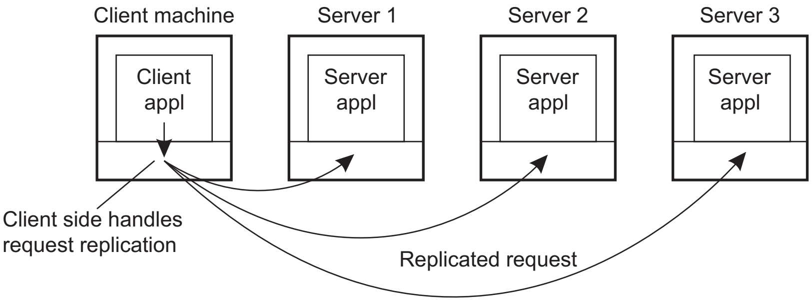
Clients

Client-side solution for distributed transparency

- In many applications, clients are more than user interfaces
    - e.g., embedded devices (ATMs, TV set-top boxes, ...)
    - e.g., web applications
    - e.g., smartphones
- Mechanism for achieving distribution transparency
    - e.g., users should not be able to tell that service is remote
- Client-side solution for replication transparency
    - i.e., users cannot tell that multiple replicas of a server exist

Clients

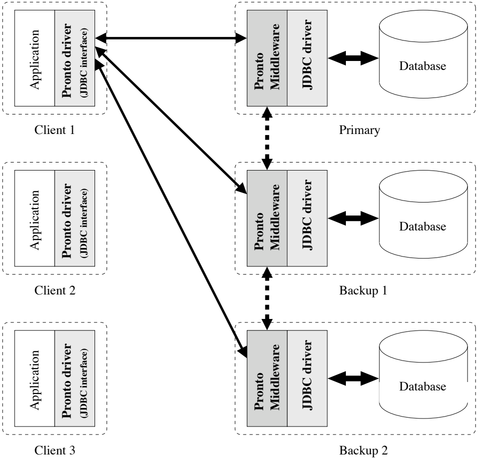
Client-side solution for replication transparency



Transparent replication of a server using a client-side solution

Clients

Example: Pronto architecture



Servers

General design issues

- Typical server organization
  - Waits for an incoming request from a client
  - Takes care of the request (e.g., executes the request)
  - Sends the response back to the client

Servers

General design issues

- Typical server organization
  - Waits for an incoming request from a client
  - Takes care of the request (e.g., executes the request)
  - Sends the response back to the client
- Iterative versus concurrent servers

Servers

General design issues

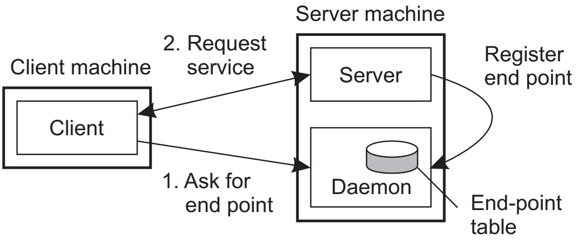
- Typical server organization
  - Waits for an incoming request from a client
  - Takes care of the request (e.g., executes the request)
  - Sends the response back to the client
- Iterative versus concurrent servers
  - Iterative: one request handled at a time
  - Concurrent: a separate thread or process executes the request
    - e.g., by forking a separate process

Servers

General design issues

- Typical server organization
  - Waits for an incoming request from a client
  - Takes care of the request (e.g., executes the request)
  - Sends the response back to the client
- Iterative versus concurrent servers
  - Iterative: one request handled at a time
  - Concurrent: a separate thread or process executes the request
    - e.g., by forking a separate process
- How clients contact a server?
  - Requests are sent to an end point (a port) at the server machine
  - Finding the end point
    - “Well-known” ports (e.g, FTP, TCP port 21, HTTP, TCP port 80)
    - Special service provides port number (daemon)

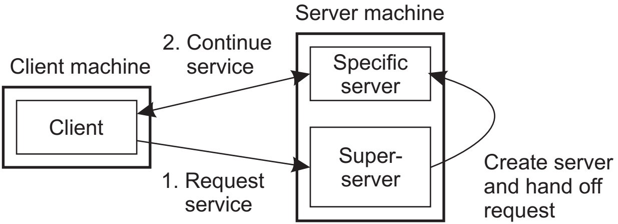
Finding the end point: through a daemon



Client-to-server binding using a daemon

Finding the end point: superservers

- Listen on many ports and fork a process to execute service
- e.g., inetd in Unix



Client-to-server binding using a superserver

Servers

State management

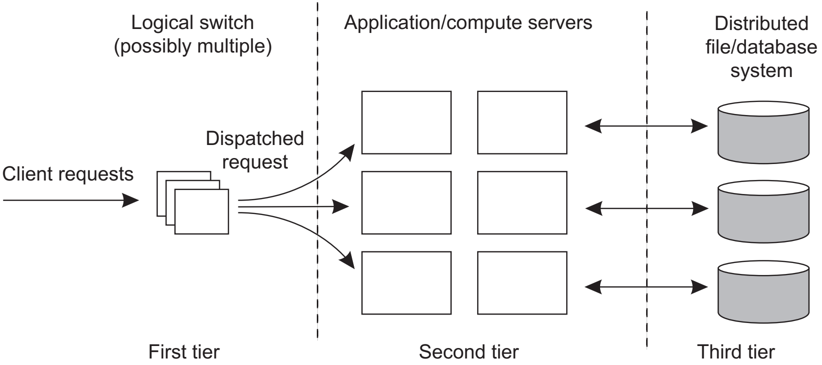
- Stateless server
  - does not keep information about the state of clients
    e.g., web server: executes request and forgets about client
  - sometimes server keeps information that can be recreated
    cf. RESTful APIs
- Stateful server
  - Maintains persistent information about clients (e.g., file server)

  ✓ Better performance (e.g., data pre-fetching)

  ✗ Problematic in case of failures

Server clusters

Collection of machines connected through a network



The general organization of a three-tiered server cluster

Servers

Server clusters

General organization:

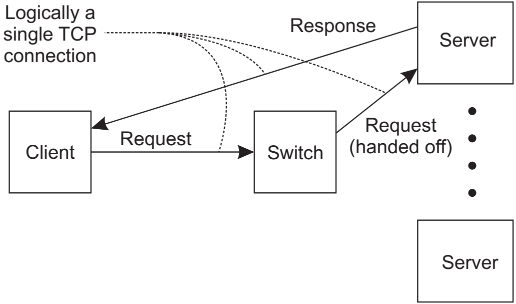
- First tier: logical switch
  - Receives client requests and routes them to servers
- Second tier: application servers
  - High-end servers, if service is computationally intensive
  - Low-end servers if storage is the bottleneck
- Third tier: data-processing servers
  - e.g., file/database systems

Servers

Server clusters

Hiding the existence of multiple servers:

- Unique access point (logical switch)
- Switch accepts TCP connection and hands it off to server
- Server replies using *switch's* IP address
- How to do load balancing?



The principle of TCP handoff

Servers

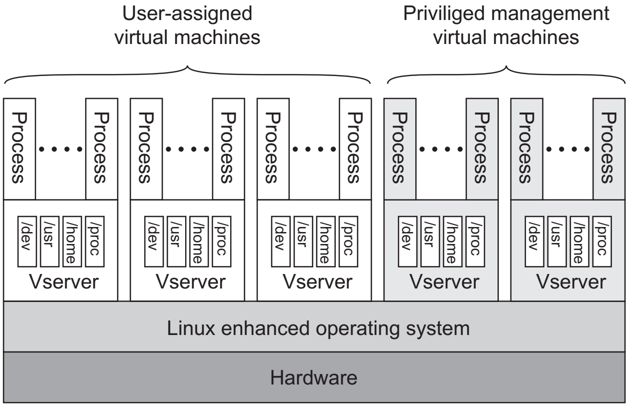
PlanetLab

- Collaborative distributed system
- Different organizations donate servers to the main system
- Testbed for geographically distributed systems

Basic entities:

- **Virtual machine monitor (VMM)**: enhanced Linux OS
- **Vserver**: isolated environment (like a container)
- **Slice**: set of Vservers, each running on a different node

PlanetLab



The basic organization of a PlanetLab node

Servers

PlanetLab

Management issues:

- Nodes belong to different organizations. Organization should be allowed to specify who can run applications on their nodes.
- Monitoring tools assume a very specific combination of hardware and software. All tailored to be used within a single organization.
- Programs from different slices but running on the same node should not interfere with each other.