Distributed Systems
Principles and Paradigms

Maarten van Steen

VU Amsterdam, Dept. Computer Science
steen@cs.vu.nl

Chapter 03: Processes

Version: November 1, 2012

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.

Processes 3.1 Threads

2 / 34

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).

Processes 3.1 Threads

3 / 34

Context Switching

Observations

1. Threads share the same address space. Thread context switching can be done entirely independent of the operating system.
2. Process switching is generally more expensive as it involves getting the OS in the loop, i.e., trapping to the kernel.
3. 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 problem is (or used to be) the loss of efficiency due to the fact that each thread operation requires a trap to the kernel.

Conclusion – but

Try to mix user-level and kernel-level threads into a single concept, however, performance gain has not turned out to outweigh the increased complexity.

Threads and 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 a linear speed-up.

Processes 3.1 Threads

7 / 34

Threads and 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.

Processes 3.1 Threads

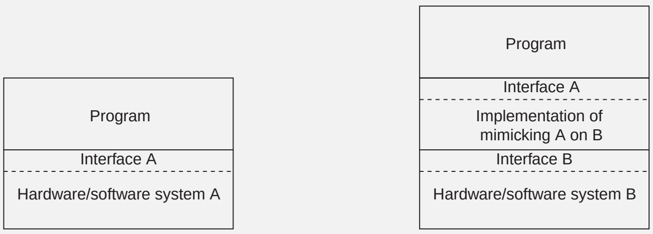
8 / 34

Virtualization

Observation

Virtualization is becoming increasingly important:

- Hardware changes faster than software
- Ease of portability and code migration
- Isolation of failing or attacked components



(a) (b)

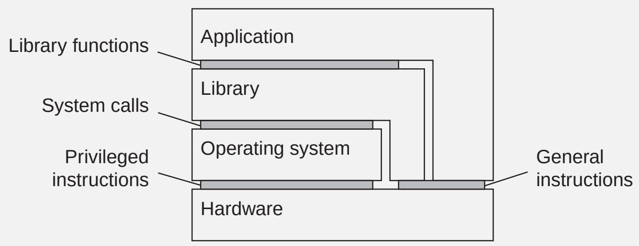
Processes 3.2 Virtualizaton

9 / 34

Architecture of VMs

Observation

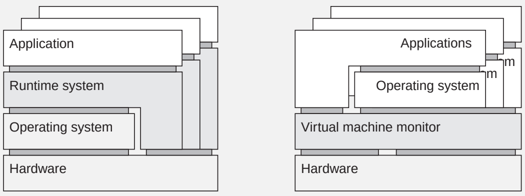
Virtualization can take place at very different levels, strongly depending on the interfaces as offered by various systems components:



Processes 3.2 Virtualizaton

10 / 34

Process VMs versus VM Monitors



(a) (b)

- Process VM: A program is compiled to intermediate (portable) code, which is then executed by a runtime system (Example: Java VM).
- VM Monitor: A separate software layer mimics the instruction set of hardware ⇒ a complete operating system and its applications can be supported (Example: VMware, VirtualBox).

Processes 3.2 Virtualizaton

11 / 34

VM Monitors on operating systems

Practice

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 orginal kernel (think of system calls, or privileged instructions).
- Sensitive instructions are replaced with calls to the VMM.

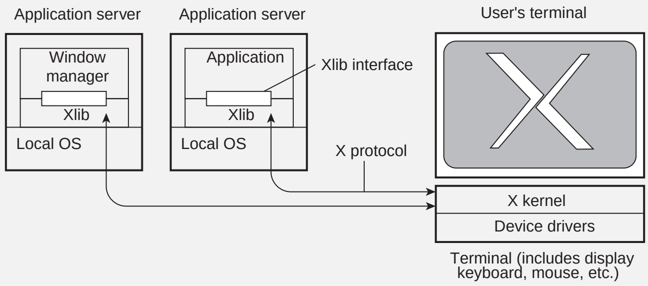
Processes 3.2 Virtualizaton

12 / 34

Clients: User Interfaces

Essence

A major part of client-side software is focused on (graphical) user interfaces.



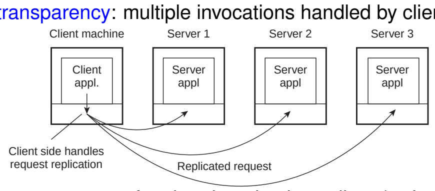
Processes 3.3 Clients

13 / 34

Client-Side Software

Generally tailored for distribution transparency

- access transparency: client-side stubs for RPCs
- location/migration transparency: let client-side software keep track of actual location
- replication transparency: multiple invocations handled by client stub:



- failure transparency: can often be placed only at client (we're trying to mask server and communication failures).

Processes 3.3 Clients

14 / 34

Servers: General organization

Basic model

A server is a process that waits for incoming service requests at a specific transport address. In practice, there is a one-to-one mapping between a port and a service.

ftp-data	20	File Transfer [Default Data]
ftp	21	File Transfer [Control]
telnet	23	Telnet
	24	any private mail system
smtp	25	Simple Mail Transfer
login	49	Login Host Protocol
sunrpc	111	SUN RPC (portmapper)
courier	530	Xerox RPC

Processes 3.4 Servers

15 / 34

Servers: General organization

Type of servers

Superservers: Servers that listen to several ports, i.e., provide several independent services. In practice, when a service request comes in, they start a subprocess to handle the request (UNIX *inetd*)

Iterative vs. concurrent servers: Iterative servers can handle only one client at a time, in contrast to concurrent servers

Processes 3.4 Servers

16 / 34

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 is 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

Processes 3.4 Servers

17 / 34

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)

Processes 3.4 Servers

18 / 34

Servers and state

Question

Does connection-oriented communication fit into a stateless design?

Processes 3.4 Servers

19 / 34

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

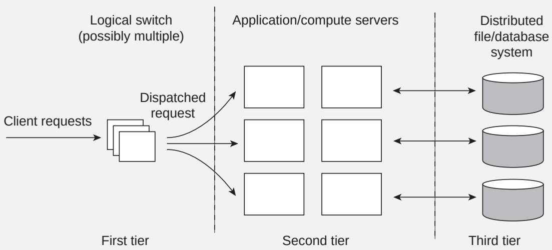
Observation

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

Processes 3.4 Servers

20 / 34

Server clusters: three different tiers



Crucial element

The first tier is generally responsible for passing requests to an appropriate server.

Processes 3.4 Servers

21 / 34

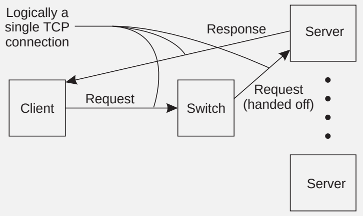
Request Handling

Observation

Having the first tier handle all communication from/to the cluster may lead to a bottleneck.

Solution

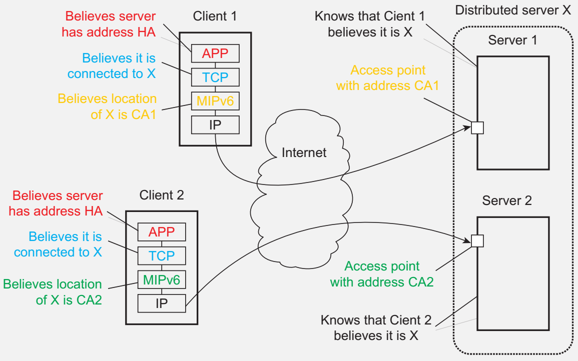
Various, but one popular one is TCP-handoff



Processes 3.4 Servers

22 / 34

Distributed servers with stable IPv6 address(es)



Processes 3.4 Servers

23 / 34

Distributed servers: addressing details

Essence

Clients having MobileIPv6 can transparently set up a connection to any peer:

- Client *C* sets up connection to IPv6 home address *HA*
- *HA* is maintained by a (network-level) home agent, which hands off the connection to a registered care-of address *CA*.
- *C* can then apply route optimization by directly forwarding packets to address *CA* (i.e., without the handoff through the home agent).

Collaborative CDNs

Origin server maintains a home address, but hands off connections to address of collaborating peer ⇒ Origin server and peer appear as one server.

Processes 3.4 Servers

24 / 34

Example: PlanetLab

Essence

Different organizations contribute machines, which they subsequently share for various experiments.

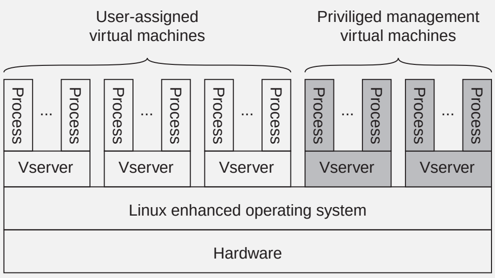
Problem

We need to ensure that different distributed applications do not get into each other's way ⇒ virtualization

Processes 3.4 Servers

25 / 34

Example: PlanetLab



Vserver: Independent and protected environment with its own libraries, server versions, and so on. Distributed applications are assigned a collection of vservers distributed across multiple machines (slice).

Processes 3.4 Servers

26 / 34

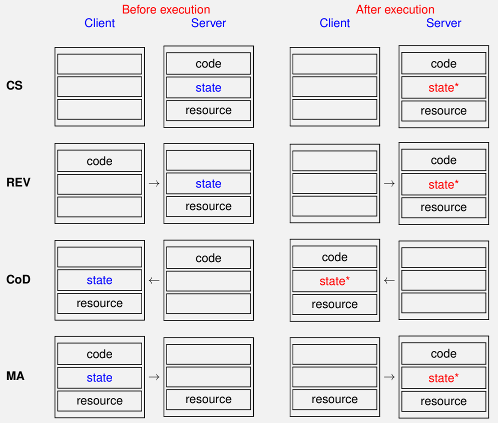
Code Migration

- Approaches to code migration
- Migration and local resources
- Migration in heterogeneous systems

Processes 3.5 Code Migration

27 / 34

Code Migration: Some Context



Processes 3.5 Code Migration

28 / 34

Processes 3.5 Code Migration

Strong and weak mobility

Object components

- Code segment: contains the actual code
- Data segment: contains the state
- Execution state: contains context of thread executing the object's code

Processes 3.5 Code Migration

29 / 34

Processes 3.5 Code Migration

Strong and weak mobility

Weak mobility

Move only code and data segment (and reboot execution):

- Relatively simple, especially if code is portable
- Distinguish code shipping (push) from code fetching (pull)

Strong mobility

Move component, including execution state

- Migration: move entire object from one machine to the other
- Cloning: start a clone, and set it in the same execution state.

Processes 3.5 Code Migration

30 / 34

Managing local resources

Problem

An object uses local resources that may or may not be available at the target site.

Resource types

- Fixed: the resource cannot be migrated, such as local hardware
- Fastened: the resource can, in principle, be migrated but only at high cost
- Unattached: the resource can easily be moved along with the object (e.g. a cache)

Processes 3.5 Code Migration

31 / 34

Processes 3.5 Code Migration

Managing local resources

Object-to-resource binding

- By identifier: the object requires a specific instance of a resource (e.g. a specific database)
- By value: the object requires the value of a resource (e.g. the set of cache entries)
- By type: the object requires that only a type of resource is available (e.g. a color monitor)

Processes 3.5 Code Migration

32 / 34

Processes 3.5 Code Migration

Managing Local Resources (2/2)

	Unattached	Fastened	Fixed
ID	MV (or GR)	GR (or MV)	GR
Value	CP (or MV, GR)	GR (or CP)	GR
Type	RB (or MV, GR)	RB (or GR, CP)	RB (or GR)

GR = Establish global systemwide reference
MV = Move the resource
CP = Copy the value of the resource
RB = Re-bind to a locally available resource

Processes 3.5 Code Migration

33 / 34

Migration in heterogenous 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

Only solution

Make use of an abstract machine that is implemented on different platforms:

- Interpreted languages, effectively having their own VM
- Virtual VM (as discussed previously)

Processes 3.5 Code Migration

34 / 34