Processes

Amir H. Payberah
amir@sics.se

Amirkabir University of Technology
(Tehran Polytechnic)

Based on slides by Maarten Van Steen

Introduction

- Processor: provides a set of instructions along with the capability of automatically executing a series of those instructions.

Introduction

- 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.

Introduction

- 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.

- Process: a software processor in whose context one or more threads may be executed.

Context

- 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).

Context

- 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.

Context

- 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.
- Process context: the minimal collection of values stored in registers and memory, used for the execution of a thread.

Context Switching

- Threads share the same address space. Thread context switching can be done entirely independent of the operating system (OS).

Context Switching

- Threads share the same address space. Thread context switching can be done entirely independent of the operating system (OS).

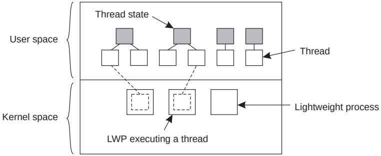
- Process switching is generally more expensive as it involves getting the OS in the loop, i.e., trapping to the kernel.

Context Switching

- Threads share the same address space. Thread context switching can be done entirely independent of the operating system (OS).
- 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.

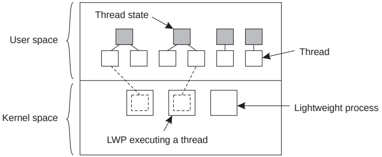
Threads and Operating Systems (1/4)

- Question: should an OS kernel provide threads, or should they be implemented as user-level packages?



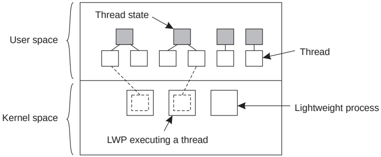
Threads and Operating Systems (2/4)

- User-space solution.



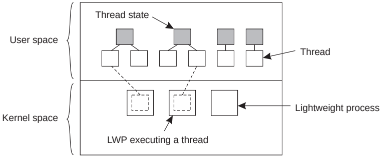
Threads and Operating Systems (2/4)

- User-space solution.
  - All operations can be completely handled within a single process ⇒ implementations can be extremely efficient.



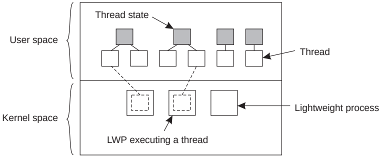
Threads and Operating Systems (2/4)

- User-space solution.
  - All operations can be completely handled within a single process $\Rightarrow$ implementations can be extremely efficient.
  - All services provided by the kernel are done on behalf of the process in which a thread resides $\Rightarrow$ if the kernel decides to block a thread, the entire process will be blocked.



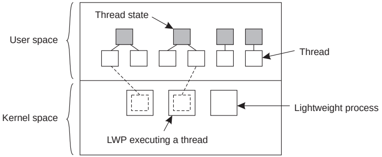
Threads and Operating Systems (2/4)

- 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.



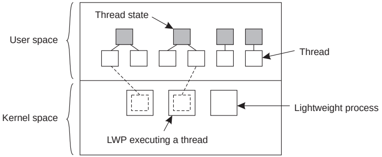
Threads and Operating Systems (3/4)

- Kernel solution: the kernel contains the implementation of a thread package. This means that all operations return as system calls.



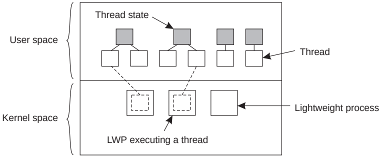
Threads and Operating Systems (3/4)

- Kernel solution: the kernel contains 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.



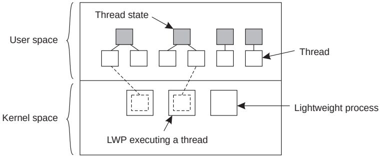
Threads and Operating Systems (3/4)

- Kernel solution: the kernel contains 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 schedules the thread associated with the event.



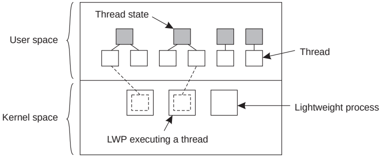
Threads and Operating Systems (3/4)

- Kernel solution: the kernel contains 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 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.



Threads and Operating Systems (4/4)

- Conclusion
  - Try to mix user-level and kernel-level threads into a single concept.



Threads and Distributed Systems (1/4)

- Multithreaded web client: hiding network latencies.

Threads and Distributed Systems (1/4)

- 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.

Threads and Distributed Systems (2/4)

- Multiple request-response calls to other machines (RPC).

Threads and Distributed Systems (2/4)

- 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.

Threads and Distributed Systems (3/4)

- Improve performance

Threads and Distributed Systems (3/4)

- Improve performance
  - Starting a thread is much cheaper than starting a new process.
  - Having a single-threaded server prohibits simple scale-up to a multi-processor system.
  - As with clients: hide network latency by reacting to next request while previous one is being replied.

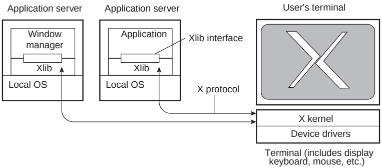
Threads and Distributed Systems (4/4)

- 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.

Clients

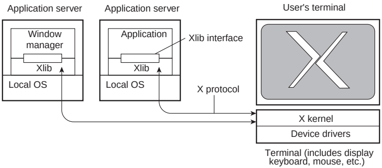
Clients

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



Clients

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



- User interface is application-aware:
  - Drag-and-drop: move objects across the screen to invoke interaction with other applications
  - In-place editing: integrate several applications at user-interface level (word processing + drawing facilities)

Client-Side Software

- Generally tailored for distribution transparency.

Client-Side Software

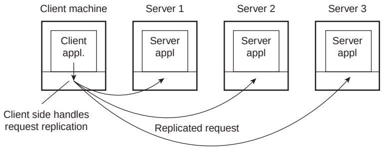
- Generally tailored for distribution transparency.
- Access transparency: client-side stubs for RPCs

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

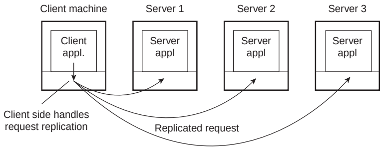
Client-Side Software

- 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.



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).

Servers

Servers

- A server is a process that waits for incoming service requests at a specific transport address (port). 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
smtp	25	Simple Mail Transfer
login	49	Login Host Protocol
sunrpc	111	SUN RPC (portmapper)
courier	530	Xerox RPC

Types 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.

Types 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.

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

Server and State (1/2)

- Stateless and stateful servers

Server and State (1/2)

- Stateless and stateful servers

- Stateless servers: never keep accurate information about the status of a client after having handled a request:

Server and State (1/2)

- Stateless and stateful servers

- 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

Server and State (1/2)

- Stateless and stateful servers

- 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)

Server and State (2/2)

- Stateful servers: keeps track of the status of its clients.

Server and State (2/2)

- 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

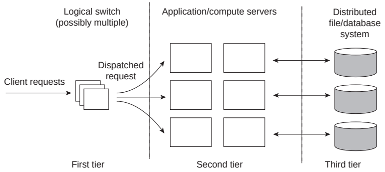
Server and State (2/2)

- 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

- 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.

Server Clusters (1/2)

- Three different tiers.
- The first tier is generally responsible for passing requests to an appropriate server.

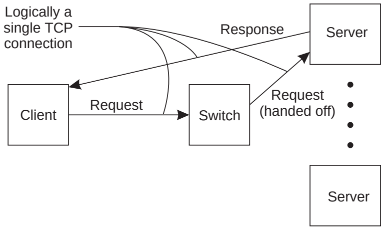


Server Clusters (2/2)

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

Server Clusters (2/2)

- 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



Example: Planet Lab (1/2)

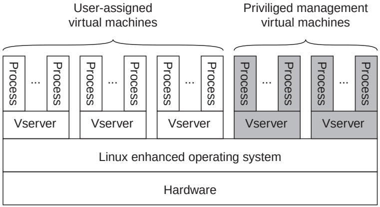
- Different organizations contribute machines, which they subsequently share for various experiments.

- Problem: we need to ensure that different distributed applications do not get into each other's way $\Rightarrow$ virtualization

Example: Planet Lab (2/2)

- 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).



Virtualization

Virtualization

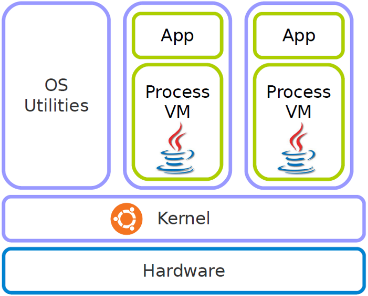
- Technique for hiding the physical characterizes of computing resources from the way other systems, applications or end users interact with them.
- Offer a different interface.
- Virtualized interface is not necessarily simpler.

Different Types of Virtualization

- Process-level virtualization
- OS-level virtualization
- System-level virtualization

Process-Level Virtualization (1/2)

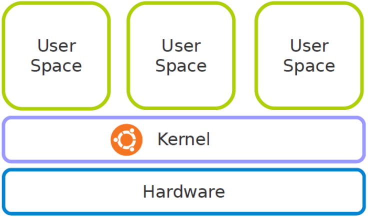
- Usually implemented on top of an OS.
- Application has to be written specifically for the VM.
- The virtual machine runs one application (one process).



Process-Level Virtualization (2/2)

OS-Level Virtualization (1/2)

- The virtual machine runs a set of userland processes.
- Userland domains are separated.
- Kernel is the same for all userland domains.



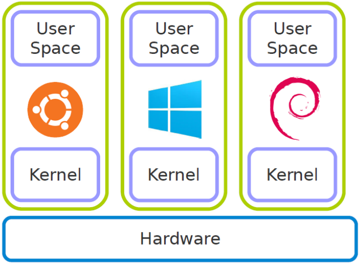
OS-Level Virtualization (2/2)

OpenVZ
Linux Containers

docker

System-Level Virtualization (1/3)

- Emulates a computer similar to a real physical one.
  - With CPU(s), memory, disk(s), network interface(s), etc.
- The virtual machine runs a full OS.



System-Level Virtualization (2/3)

- Full virtualization vs. Paravirtualization.

System-Level Virtualization (2/3)

- Full virtualization vs. Paravirtualization.
- Full virtualization: the guest OS is not aware it is being virtualized and requires no modification.

System-Level Virtualization (2/3)

- Full virtualization vs. Paravirtualization.
- Full virtualization: the guest OS is not aware it is being virtualized and requires no modification.
- Paravirtualization: the guest OS should be modified in order to be operated in the virtual environment.

System-Level Virtualization (3/3)

Hypervisor

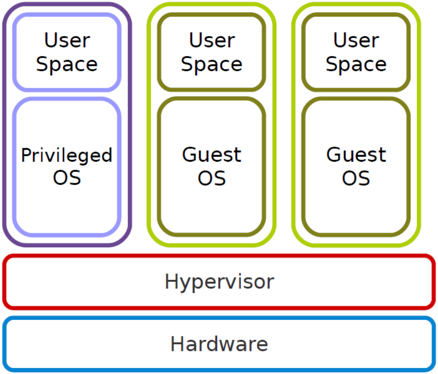
- In the system-level virtualization, virtual machines are managed by another software layer.

- It is called hypervisor or Virtual Machine Manager (VMM).

Hypervisor

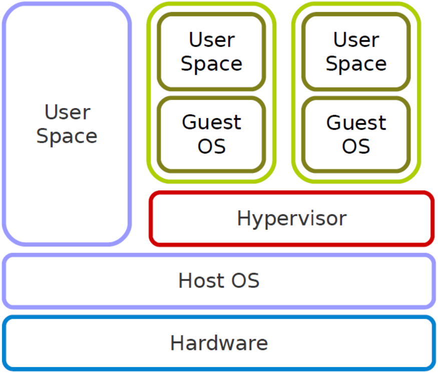
- In the system-level virtualization, virtual machines are managed by another software layer.
- It is called hypervisor or Virtual Machine Manager (VMM).
- Two types of hypervisors:
  - Type 1: runs directly on hardware (Native/Bare-Metal)
  - Type 2: hosted on top of another operating system (Hosted)

Bare Metal Hypervisor



- Xen, ...

Hosted Hypervisor



- VMWare, KVM, Virtualbox, ...

Code Migration

Strong and Weak Mobility (1/2)

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

Strong and Weak Mobility (2/2)

- 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 and Weak Mobility (2/2)

- 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.

Managing Local Resources (1/2)

- Problem: an object uses local resources that may or may not be available at the target site.

Managing Local Resources (1/2)

- 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.

Managing Local Resources (1/2)

- 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.

Managing Local Resources (1/2)

- 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).

Managing Local Resources (2/2)

- Object-to-resource binding:

Managing Local Resources (2/2)

- Object-to-resource binding:
  - By identifier: the object requires a specific instance of a resource (e.g., a specific database).

Managing Local Resources (2/2)

- 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., standard libraries).

Managing Local Resources (2/2)

- 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., standard libraries).
  - By type: the object requires that only a type of resource is available (e.g., a color monitor).

Migration in Heterogeneous Systems

- The target machine may not be suitable to execute the migrated code

Migration in Heterogeneous Systems

- 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, OS and runtime system.

Migration in Heterogeneous Systems

- 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, OS 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)

Summary

Summary

- Process and Threads
- Threads in OS: user-level vs. kernel-level implementations
- Threads in distributed systems: improve performance
- Clients
- Servers: stateless vs. stateful, server clusters
- Virtualization: process level, OS level, and system level
- Code migration
  - Weak vs. strong mobility
  - Local resources: fixed, fastened, and unattached
  - Object-to-resource-binding: by id, by value, by type

Reading

- Chapter 3 of the Distributed Systems: Principles and Paradigms.

Questions?