VIRTUALIZATION AND CLOUD PLATFORMS

George Porter
Oct 18 and 25, 2018

ATTRIBUTION

• These slides are released under an Attribution-NonCommercial-ShareAlike 3.0 Unported (CC BY-NC-SA 3.0) Creative Commons license

• These slides incorporate material from:
  • Michael Freedman and Kyle Jamieson, Princeton University (also under a CC BY-NC-SA 3.0 Creative Commons license)
Outline

• Terminology: Parallelism vs Concurrency
• Processes, threads, and OS-level mechanisms
• Datacenters

CONCURRENCY VS PARALLELISM

• Both deal with doing a lot at once, but aren’t the same thing
  • Given set of tasks \{T_1, T_2, ..., T_n\}

• Concurrency:
  • Progress of multiple elements of the set overlap in time

• Parallelism:
  • Progress on elements of the set occur at the same time
CONCURRENCY

• Might be parallel, might not be parallel

• A single thread of execution can time slice a set of tasks to make partial progress over time
  • Time 0: Work on first 25% of Task 0
  • Time 1: Work on first 25% of Task 1
  • Time 2: Work on first 25% of Task 2
  • Time 3: Work on first 25% of Task 3
  • Time 4: Work on second 25% of Task 0
  • Time 5: Work on second 25% of Task 1
  • ...

PARALLELISM

Multiple execution units enable progress to be made simultaneously

**Processor 1**

• Time 0: 1st 25% of Task1
• Time 1: 2nd 25% of Task1
• Time 2: 3rd 25% of Task1
• Time 3: 4th 25% of Task1
• Time 4: 1st 25% of Task3

**Processor 2**

• Time 0: 1st 25% of Task2
• Time 1: 2nd 25% of Task2
• Time 2: 3rd 25% of Task2
• Time 3: 4th 25% of Task2
• Time 4: 1st 25% of Task4
FLASH TRAFFIC

- USGS Pasadena, CA office Earthquake site
- Oct 16, 1999 earthquake

THREADING AND PERFORMANCE

- Too much parallelism causes thrashing, excessive switching, lower performance
Outline

- Terminology: Parallelism vs Concurrency
- Processes, threads, and OS-level mechanisms
- Datacenters
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).
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).
# 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).
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 (somewhat) 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.
Why use threads

Some simple reasons

- **Avoid needless blocking**: a single-threaded process will block when doing I/O; in a multi-threaded process, the operating system can switch the CPU to another thread in that process.

- **Exploit parallelism**: the threads in a multi-threaded process can be scheduled to run in parallel on a multiprocessor or multicore processor.

- **Avoid process switching**: structure large applications not as a collection of processes, but through multiple threads.
Avoid process switching

Avoid expensive context switching

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

Trade-offs

- Threads use the same address space: more prone to errors
- No support from OS/HW to protect threads using each other’s memory
- Thread context switching may be faster than process context switching
The cost of a context switch

Consider a simple clock-interrupt handler

- **direct costs**: actual switch and executing code of the handler
- **indirect costs**: other costs, notably caused by messing up the cache

What a context switch may cause: indirect costs

- (a) before the context switch
- (b) after the context switch
- (c) after accessing block D.
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.
Using threads at the client side

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.
Using threads at the server side

### Improve performance

- Starting a thread is 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.
Why multithreading is popular: organization

Dispatcher/worker model

Overview

<table>
<thead>
<tr>
<th>Model</th>
<th>Characteristics</th>
</tr>
</thead>
<tbody>
<tr>
<td>Multithreading</td>
<td>Parallelism, blocking system calls</td>
</tr>
<tr>
<td>Single-threaded process</td>
<td>No parallelism, blocking system calls</td>
</tr>
<tr>
<td>Finite-state machine</td>
<td>Parallelism, nonblocking system calls</td>
</tr>
</tbody>
</table>
TREADING AND PROCESS CREATION

Outline

- Terminology: Parallelism vs Concurrency
- Processes, threads, and OS-level mechanisms
- Datacenters
CLOUD SERVICE MODELS

• Software as a Service (Saas)
  • Provider licenses applications to users as a service
  • e.g., customer relationship management, email, …
  • Avoid costs of installation, maintenance, patches, …

• Platform as a Service (Paas)
  • Provider offers software platform for building applications
  • e.g., Google’s App-Engine
  • Avoid worrying about scalability of platform

• Infrastructure as a Service (Iaas)
  • Provider offers raw computing, storage, and network
  • e.g., Amazon’s Elastic Computing Cloud (EC2)
  • Avoid buying servers and estimating resource needs
TOP-OF-RACK ARCHITECTURE

- Rack of servers
  - Commodity servers
  - And top-of-rack switch

- Modular design
  - Preconfigured racks
  - Power, network, and storage cabling

- Aggregate to the next level

RACKS OF SERVERS (GOOGLE)
EXTREME MODULARITY

- Containers

- Many containers

VIRTUALIZATION
Virtualization

Observation

Virtualization is important:

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

Principle: mimicking interfaces

![Diagram showing the principle of virtualization](image)
Mimicking interfaces

Four types of interfaces at three different levels:

1. **Instruction set architecture**: the set of machine instructions, with two subsets:
   - Privileged instructions: allowed to be executed only by the operating system.
   - General instructions: can be executed by any program.
2. **System calls** as offered by an operating system.
3. **Library calls**, known as an application programming interface (API).
Ways of virtualization

(a) Process VM, (b) Native VMM, (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.
Zooming into VMs: performance

Refining the organization

- Application/Libraries
- Guest operating system
- Virtual machine monitor
- Host operating system
- Hardware

- Privileged instruction: if and only if executed in user mode, it causes a trap to the operating system
- Nonprivileged 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).
### VMs and cloud computing

#### Three types of cloud services

- **Infrastructure-as-a-Service** covering the basic infrastructure
- **Platform-as-a-Service** covering system-level services
- **Software-as-a-Service** containing actual applications

#### IaaS

Instead of renting out a physical machine, a cloud provider will rent out a VM (or VMM) that may possibly be sharing a physical machine with other customers ⇒ almost complete isolation between customers (although performance isolation may not be reached).
VMM VIRTUAL SWITCHES

THE STORAGE HIERARCHY

One Server
- DRAM: 16 GB, 100 ns, 20 GB/s
- Disk: 2 TB, 10 ms, 200 MB/s
- Flash: 128 GB, 100 us, 1 GB/s

Local Rack (80 servers)
- DRAM: 1 TB, 300 us, 100 MB/s
- Disk: 160 TB, 11 ms, 100 MB/s
- Flash: 20 TB, 400 us, 100 MB/s

Cluster (30 racks)
- DRAM: 30 TB, 500 us, 10 MB/s
- Disk: 4.80 PB, 12 ms, 10 MB/s
- Flash: 600 TB, 600 us, 10 MB/s
### HARDWARE COMPARISONS

**Table 3.1**: Server hardware configuration, benchmark results, and derived (unofficial) cost-efficiency metrics for a large SMP server and a low-end, PC-class server.

<table>
<thead>
<tr>
<th></th>
<th>HP INTEGRITY SUPERDOME-ITANIUM2</th>
<th>HP PROLIANT ML350 G5</th>
</tr>
</thead>
<tbody>
<tr>
<td>Processor</td>
<td>64 sockets, 128 cores</td>
<td>1 socket, quad-core,</td>
</tr>
<tr>
<td></td>
<td>(dual-threaded), 1.6 GHz</td>
<td>2.66 GHz X5355 CPU,</td>
</tr>
<tr>
<td></td>
<td>Itanium2, 12 MB</td>
<td>8 MB last-level cache</td>
</tr>
<tr>
<td></td>
<td>last-level cache</td>
<td></td>
</tr>
<tr>
<td>Memory</td>
<td>2,048 GB</td>
<td>24 GB</td>
</tr>
<tr>
<td>Disk storage</td>
<td>320,974 GB, 7,056 drives</td>
<td>3,961 GB, 105 drives</td>
</tr>
<tr>
<td>TPC-C price/performance</td>
<td>$2.93/tpmC</td>
<td>$0.73/tpmC</td>
</tr>
<tr>
<td>price/performance</td>
<td>$1.28/transactions per minute</td>
<td>$0.10/transactions per minute</td>
</tr>
<tr>
<td>(server HW only)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Price/performance</td>
<td>$2.39/transactions per minute</td>
<td>$0.12/transactions per minute</td>
</tr>
<tr>
<td>(no discounts)</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Barroso and Holzle, The Data Center as a Computer: An Introduction to the Design of Warehouse-Scale Machines
FIGURE 3.2: Performance advantage of a cluster built with high-end server nodes (128-core SMP) over a cluster with the same number of processor cores built with low-end server nodes (four-core SMP), for clusters of varying size.

Barroso and Holzle, The Data Center as a Computer: An Introduction to the Design of Warehouse-Scale Machines.