VIRTUALIZATION AND CLOUD PLATFORMS

George Porter
Feb 1, 2019
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)
  - Andrew Moore, Univ. of Cambridge
  - The Datacenter as a Computer: An Introduction to the Design of Warehouse-Scale Machines, 2nd ed., by Barroso, Clidaras, and Hölszle
ANNOUNCEMENTS

Project 1 due Monday

Gradescope invitation code: 97EGV3

Material for today can be found in van Steen and Tanenbaum 3.1 and 3.2
ABOUT STD::THREAD::JOINABLE()

- C++11/14/17 supports in-language threads
  - Vs. 3rd party library like pthreads
  - ...but uses pthreads internally
- Std::thread::joinable() semantics
  - Is the thread running?
  - **Not** if the thread has finished
- Result:
  - Testing for joinable() in main thread produces code that can only handle one client at a time

Solution 1: “Proper solution”
- Use of std::promise and std::future (beyond the scope of this course)

Solution 2: ‘hacky method’
- Call thread.detach() after creating the thread
- Not ideal in general

Solution 3: Allocate threads on the heap
- Never call joinable() or join()
- Would cause resource leak in real system, shouldn’t be a problem for 10-100 requests
log->info("Waiting for connections...\n");

while (true) {
    vector<thread*>* workers;

    sin_size = sizeof(their_addr);

    clientSock = accept(servSock, (struct sockaddr *)&their_addr,
                        &sin_size);

    if (clientSock == -1) {
        log->error("accept");
        continue;
    }

    thread * t = new thread(handle_client, clientSock, doc_root);
    workers.push_back(t);

/*
   for (thread &t: workers) {
       if (t.joinable()) {
           log->info("Just joined a thread");
           t.join();
       }
   }
*/
}
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 \( \{T_1, T_2, \ldots, T_n\} \)

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

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

• 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
  • ...
Multiple execution units enable progress to be made simultaneously

<table>
<thead>
<tr>
<th>Processor 1</th>
<th>Processor 2</th>
</tr>
</thead>
<tbody>
<tr>
<td>Time 0: 1&lt;sup&gt;st&lt;/sup&gt; 25% of Task1</td>
<td>Time 0: 1&lt;sup&gt;st&lt;/sup&gt; 25% of Task2</td>
</tr>
<tr>
<td>Time 1: 2&lt;sup&gt;nd&lt;/sup&gt; 25% of Task1</td>
<td>Time 1: 2&lt;sup&gt;nd&lt;/sup&gt; 25% of Task2</td>
</tr>
<tr>
<td>Time 2: 3&lt;sup&gt;rd&lt;/sup&gt; 25% of Task1</td>
<td>Time 2: 3&lt;sup&gt;rd&lt;/sup&gt; 25% of Task2</td>
</tr>
<tr>
<td>Time 3: 4&lt;sup&gt;th&lt;/sup&gt; 25% of Task1</td>
<td>Time 3: 4&lt;sup&gt;th&lt;/sup&gt; 25% of Task2</td>
</tr>
<tr>
<td>Time 4: 1&lt;sup&gt;st&lt;/sup&gt; 25% of Task3</td>
<td>Time 4: 1&lt;sup&gt;st&lt;/sup&gt; 25% of Task4</td>
</tr>
</tbody>
</table>
FLASH TRAFFIC

- USGS Pasadena, CA office Earthquake site
- Oct 16, 1999 earthquake
• Too much parallelism causes thrashing, excessive switching, lower performance
Outline

• Terminology: Parallelism vs Concurrency
• Virtualization
• Datacenters
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.
Multithreaded clients: does it help?

Thread-level parallelism: TLP

Let $c_i$ denote the fraction of time that exactly $i$ threads are being executed simultaneously.

$$ TLP = \frac{\sum_{i=1}^{N} i \cdot c_i}{1 - c_0} $$

with $N$ the maximum number of threads that (can) execute at the same time.
Multithreaded clients: does it help?

Thread-level parallelism: TLP

Let $c_i$ denote the fraction of time that exactly $i$ threads are being executed simultaneously.

$$TLP = \frac{\sum_{i=1}^{N} i \cdot c_i}{1 - c_0}$$

with $N$ the maximum number of threads that (can) execute at the same time.

Practical measurements

A typical Web browser has a TLP value between 1.5 and 2.5 $\Rightarrow$ threads are primarily used for logically organizing browsers.
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>
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
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.
Processes: Virtualization

Principle of virtualization

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).
Condition for virtualization

**Necessary condition**

*For any conventional computer, a virtual machine monitor may be constructed if the set of sensitive instructions for that computer is a subset of the set of privileged instructions.*

**Problem: condition is not always satisfied**

There may be sensitive instructions that are executed in user mode without causing a trap to the operating system.

**Solutions**

- Emulate all instructions
- Wrap nonprivileged sensitive instructions to divert control to VMM
- **Paravirtualization**: modify guest OS, either by preventing nonprivileged sensitive instructions, or making them nonsensitive (i.e., changing the context).
Three different tiers

Common organization

- Logical switch (possibly multiple)
- Application/compute servers
- Distributed file/database system

Client requests → Dispatched request → First tier
- Second tier
- Third tier

Crucial element
The first tier is generally responsible for passing requests to an appropriate server: request dispatching
Request Handling

Observation

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

A solution: TCP handoff

Logically a single TCP connection
Migrating a virtual machine

Migrating images: three alternatives

1. Pushing memory pages to the new machine and resending the ones that are later modified during the migration process.

2. Stopping the current virtual machine; migrate memory, and start the new virtual machine.

3. Letting the new virtual machine pull in new pages as needed: processes start on the new virtual machine immediately and copy memory pages on demand.
Performance of migrating virtual machines

Problem
A complete migration may actually take tens of seconds. We also need to realize that during the migration, a service will be completely unavailable for multiple seconds.

Measurements regarding response times during VM migration

<table>
<thead>
<tr>
<th>Time</th>
<th>Migration</th>
<th>Downtime</th>
<th>Response time</th>
</tr>
</thead>
<tbody>
<tr>
<td>Migration</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Downtime</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Response time</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Time

Response time

Migration

Downtime

Time

47 / 47
Outline

- Terminology: Parallelism vs Concurrency
- Processes, threads, and OS-level mechanisms
- Datacenters
DATACENTERS ARE NOT EXACTLY NEW...

EDSAC, 1949
“ROWS” OF SERVERS IN A DATACENTER
“RACKS” MAKING UP ONE ROW
A SINGLE RACK

• 20-40 “pizza box” servers per rack

• Each rack has a “top of rack” network switch that connects it to the rest of the datacenter network
CONNECTING RACKS TOGETHER

- “Aggregation” and “Core” network switches provide connectivity between racks
BROCADE REFERENCE DESIGN
**CISCO REFERENCE DESIGN**

Internet

- **S** = network switch
- **AR** = aggregation router
- **CR** = core router

```
~ 40-80 servers/rack
```

S = network switch
AR = aggregation router
CR = core router
DATACENTER PERFORMANCE

- Ideal: Homogeneous performance
  - Uniform bandwidth/latency between all servers
- Reality (typical): Heterogeneous performance
  - Two servers in the same rack
    - Very high bandwidth/very low latency
  - Two servers in same row (not same rack)
    - Medium bandwidth / medium latency
  - Two servers in different rows
    - Low bandwidth / high latency
EXTREME MODULARITY

- Containers filled with a 2 or 4 rows of servers

- Many containers
EFFECT OF THE NETWORK ON PERFORMANCE

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
UC San Diego