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 (1/2)

**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 (2/2)

**Observation 1**: Threads share the same address space. Thread context switching can be done entirely independent of the operating system.

**Observation 2**: Process switching is generally more expensive as it involves getting the OS in the loop, i.e., trapping to the kernel.

**Observation 3**: Creating and destroying threads is much cheaper than doing so for processes.
Threads and Operating Systems (1/2)

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

User-space solution:

- We’ll have nothing to do with the kernel, so 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. Requires messy solutions.
- In practice we want to use threads 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 (2/2)

Kernel solution: The whole idea is to have the kernel contain the implementation of a thread package. This does mean 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 big problem is the loss of efficiency due to the fact that each thread operation requires a trap to the kernel.

Conclusion: Try to mix user-level and kernel-level threads into a single concept.
Solaris Threads (1/2)

**Basic idea:** Introduce a two-level threading approach: lightweight processes that can execute user-level threads.

![Diagram of thread states and processes]

- When a user-level thread does a system call, the LWP that is executing that thread, **blocks**. The thread remains **bound** to the LWP.
- The kernel can simply schedule another LWP having a runnable thread bound to it. Note that this thread can switch to **any** other runnable thread currently in user space.
- When a thread calls a blocking user-level operation, we can simply do a context switch to a runnable thread, which is then bound to the same LWP.
- When there are no threads to schedule, an LWP may remain idle, and may even be removed (destroyed) by the kernel.

Solaris Threads (2/2)
Multithreaded clients: Main issue is hiding network latency.

Multithreaded Web client:
- 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 compared to doing calls one after the other.

Multithreaded servers: Main issue is improved performance and better structure.

Improve performance:
- Starting a thread to handle an incoming request is much cheaper than starting a new process.
- Having a single-threaded server prohibits simply scaling the server 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.
Virtualization

**Observation:** Virtualization is becoming increasingly important:

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

![Diagram of Virtualization](attachment:image.png)

---

Architecture of VMs (1/3)

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

![Diagram of Architectural Components](attachment:image.png)
Architecture of VMs (2/3)

Note: Make a distinction between (a) process virtual machines and (b) virtual machine monitors:

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

Architecture of VMs (3/3)

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 original kernel (think of system calls, or privileged instructions).
- Sensitive instructions are replaced with calls to the VMM
Clients: User Interfaces

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

<table>
<thead>
<tr>
<th>Application server</th>
<th>Application server</th>
<th>User's terminal</th>
</tr>
</thead>
<tbody>
<tr>
<td>Window manager</td>
<td>Application</td>
<td>X protocol</td>
</tr>
<tr>
<td></td>
<td>Xlib interface</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Local OS</td>
<td>X kernel</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Device drivers</td>
</tr>
<tr>
<td>Terminal (includes display keyboard, mouse, etc.)</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**Compound documents:** User interface is application-aware ⇒ interapplication communication:

- **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

**Essence:** Often 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:

<table>
<thead>
<tr>
<th>Client machine</th>
<th>Server 1</th>
<th>Server 2</th>
<th>Server 3</th>
</tr>
</thead>
<tbody>
<tr>
<td>Client side handles request replication</td>
<td>Replicated request</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

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

<table>
<thead>
<tr>
<th>Port</th>
<th>Service</th>
</tr>
</thead>
<tbody>
<tr>
<td>20</td>
<td>File Transfer [Default Data]</td>
</tr>
<tr>
<td>21</td>
<td>File Transfer [Control]</td>
</tr>
<tr>
<td>23</td>
<td>Telnet</td>
</tr>
<tr>
<td>24</td>
<td>any private mail system</td>
</tr>
<tr>
<td>25</td>
<td>Simple Mail Transfer</td>
</tr>
<tr>
<td>49</td>
<td>Login Host Protocol</td>
</tr>
<tr>
<td>111</td>
<td>SUN RPC (portmapper)</td>
</tr>
<tr>
<td>530</td>
<td>Xerox RPC</td>
</tr>
</tbody>
</table>

**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.

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 (possibly per service request):

- Server has a separate thread (or process) waiting for incoming urgent messages
- When urgent message comes in, associated request is put on hold
- Note: we require OS supports high-priority scheduling of specific threads or processes

**Solution 2:** Use out-of-band communication facilities of the transport layer:

- Example: TCP allows to send urgent messages in the same connection
- Urgent messages can be caught using OS signaling techniques
Servers and State (1/2)

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)

Question: Does connection-oriented communication fit into a stateless design?

Servers 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

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.
**Server Clusters**

**Observation:** Many server clusters are organized along three different tiers:

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

**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:
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:**

- **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**).

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Hardware</td>
<td>Linux enhanced operating system</td>
<td>Hardware</td>
<td>Linux enhanced operating system</td>
<td>Hardware</td>
<td>Linux enhanced operating system</td>
<td>Hardware</td>
<td>Linux enhanced operating system</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**Code Migration**

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

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

**Weak mobility**: Move only code and data segment (and start execution from the beginning) after migration:

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

**Strong mobility**: Move component, including execution state

- **Migration**: move the entire object from one machine to the other
- **Cloning**: simply 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.

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)

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)

Managing Local Resources (2/2)

<table>
<thead>
<tr>
<th>ID</th>
<th>Unattached</th>
<th>Fastened</th>
<th>Fixed</th>
</tr>
</thead>
<tbody>
<tr>
<td>Value</td>
<td>MV (or GR)</td>
<td>GR (or MV)</td>
<td>GR</td>
</tr>
<tr>
<td>Type</td>
<td>CP (or MV, GR)</td>
<td>GR (or CP)</td>
<td>GR</td>
</tr>
<tr>
<td></td>
<td>RB (or MV, GR)</td>
<td>RB (or GR, CP)</td>
<td>RB (or GR)</td>
</tr>
</tbody>
</table>

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

Current solutions:
- Interpreted languages running on a virtual machine (Java/JVM; scripting languages)
- Virtual machine monitors, allowing migration of complete OS + apps.