Chapter 3
Processes
Overview

- Multithreading (for higher performance)
- Virtualization (for portability and failure isolation)
- Clients
- Servers
- Code Migration
Advantages of Threads

- Improves application responsiveness by allowing them to not block

- Allows for parallel computation resulting in higher speed on many-core systems

- Easier to structure many applications as a collection of cooperating threads

- Higher performance compared to multiple processes since switching between threads takes less time
Thread Usage in Nondistributed Systems

Figure 3-1. Context switching in processes as the result of IPC.
Thread Implementation

- **User-space threads:**
  - Creating/destroying threads is inexpensive
  - Switching context is very fast
  - But invocation of a blocking system call blocks all threads…
  - Cannot make use of multiple cores

- **Kernel-space threads:**
  - Overcomes the last two issues with user-space threads but loses performance

- **Hybrid model:** *Light-Weight Processes* (LWP). Multiple LWP/threads run inside a single (heavy-weight) process. In addition, the system offers a user-level threads package
Thread Implementation

Figure 3-2. Combining kernel-level lightweight processes and user-level threads.
Multithreaded Clients

Multiple threads can be used to hide delays in network communications.

- For example, a web browser can start up several threads, one for downloading the HTML source of the page, one each for images on the page, one each for animations/applets etc.

- Replicated web servers along with multi-threaded clients can result in shorter download times.
Multithreaded Servers (1)

Figure 3-3. A multithreaded server organized in a *dispatcher/worker* model.
Multithreaded Servers (2)

<table>
<thead>
<tr>
<th>Model</th>
<th>Characteristics</th>
</tr>
</thead>
<tbody>
<tr>
<td>Threads</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>

Figure 3-4. Three ways to construct a server.
Virtualization

Virtualization is the creation of a virtual (rather than actual) version of something, such as a hardware platform, operating system, a storage device or network resources.

- Eases administration of large number of servers (or resources)
- Helps with scalability and better utilization of hardware resources
- The driver behind cloud computing and utility computing
- Has been around for decades. IBM mainframes have used this technique very successfully for a long time
The Role of Virtualization in Distributed Systems

Figure 3-5. (a) General organization between a program, interface, and system. (b) General organization of virtualizing system A on top of system B.
Architectures of Virtual Machines (1)

Interfaces at different levels

- An interface between the hardware and software consisting of *machine instructions* that can be invoked by any program.

- An interface between the hardware and software, consisting of machine instructions that can be invoked only by privileged programs, such as an operating system.
Architectures of Virtual Machines (2)

Interfaces at different levels

- An interface consisting of *system calls* as offered by an operating system.
- An interface consisting of library calls
  - generally forming what is known as an *application programming interface* (API).
  - In many cases, the aforementioned system calls are hidden by an API.
Architectures of Virtual Machines (3)

Figure 3-6. Various interfaces offered by computer systems.
Architectures of Virtual Machines (4)

Virtualization can be implemented at two levels.

- **Process Virtual Machine**: An abstract instruction set that is to be used for executing applications. For example: Java runtime, Windows emulation (Wine) on Unix/Linux/MacOS.

- **Virtual Machine Monitor**: A layer completely shielding the original hardware but offering the complete instruction set of that same (or other hardware) as an interface. Makes it possible to have multiple instances of different operating systems run simultaneously on the same platform. Examples: VMWare, VirtualBox, Xen, VirtualPC, Parallels etc.
Architectures of Virtual Machines (5)

Figure 3-7. (a) A process virtual machine, with multiple instances of (application, runtime) combinations.
Architectures of Virtual Machines (6)

Figure 3-7. (b) A virtual machine monitor, with multiple instances of (applications, operating system) combinations.
Design Issues for Clients

- For each remote service the client machine can have a separate counterpart that can contact the service over the network.

- Provide direct access to remote services by only offering a convenient user interface. The client machine is used only as a terminal with no need for local storage, leading to an application neutral solution (*Thin-client* approach).

- **Transparency for clients**: Access (via stubs/interfaces), location, migration, relocation, replication, failure, concurrency (intermediate layer like a transaction monitor), persistence (server-side).
Networked User Interfaces (1)

Figure 3-8. (a) A networked application with its own protocol.
Figure 3-8. (b) A general solution to allow access to remote applications.
Example: The XWindow System

Figure 3-9. The basic organization of the XWindow System.
Client-Side Software for Distribution Transparency

Figure 3-10. Transparent replication of a server using a client-side solution.
Design Issues for Servers

- A server is a process implementing a specific service on behalf of a collection of clients.
- A server can be *iterative* or *concurrent*. Concurrent servers can be multi-threaded or multi-process.
- How does a client find a server? Need to know the *end-point* or *port* and the host address.
  - Statically assigned like well known servers like HTTP on port 80.
  - Look-up service provided by a special *directory* server.
  - Using a *superserver* that selects on multiple ports and forks off the appropriate server when a request comes in.
General Design Issues (1)

Figure 3-11. (a) Client-to-server binding using a daemon.
General Design Issues (2)

Figure 3-11. (b) Client-to-server binding using a superserver.
Servers: Further Design Issues

- How to handle communication interrupts? Use *out-of-band data*. Example: to cancel the upload of a huge file
  - Server listens to separate endpoint, which has higher priority, while also listening to the normal endpoint (with lower priority)
  - Send urgent data on the same connection. Can be done with TCP, where the server gets a signal (SIGURG) on receiving urgent data

- **Stateless** servers. A stateless server does not remember anything from one request to another. For example, a HTTP server is stateless. *Cookies* can be used to transmit information specific to a client with a stateless server. Easy to recover from a crash

- **Stateful** servers. Maintains information about its clients. Performance improvement over stateless servers is often the reason for stateful servers. Needs to recover its entire state as it was just before crash. Can be quite complex for distributed servers

- **Soft state**: The server promises to maintain state on behalf of the client, but only for a limited time
Server Clusters (1)

Figure 3-12. The general organization of a three-tiered server cluster.
Figure 3-13. The principle of TCP handoff (requires IP forwarding and IP spoofing).
Server Clusters (3)

- Switch can hand off connections in round-robin and thus be oblivious of the service being provided
- Switch can handoff request based on type of service requested to the appropriate server
- Switch can handoff request based on server loads
- Switch can handoff request by being aware of the content

- Single point of access can be made better using DNS to map one hostname to several servers. But the client still has to try multiple servers in case some are down.
Distributed Servers

- A *distributed server* is a possibly dynamically changing set of machines, with also possibly varying access points, but which nevertheless appears to the outside world as a single, powerful machine.

- A *stable access point* across a distributed server can be implemented using mobility support for IPv6 (MIPv6):
  - *home network*: where a mobile node normally resides
  - *home address (HoA)*: stable address for a node in its home network
  - *care of address (CoA)*: when a mobile node attaches to a foreign network, its sets up a forwarding from its HoA to the CoA
  - *route optimization* from MPIv6 is used to make different clients believe they are communicating with a single server where, in fact, each is communicating with a different member node of the distributed server
Distributed Servers

Figure 3-14. Route optimization in a distributed server.
Managing Server Clusters

- An administrator manually logs in to each server to manage them. Or they use shell scripts or simple tools to manage a collection of servers.
- An management interface is provided at one system to collectively manage the cluster of servers.
- However to go to the level of thousands or more of servers, most administration approaches are ad hoc and still an active area of research.
- Large server cluster farms are increasing rapidly! Google is said to operate over a million servers in 12 farms.
Managing Server Clusters

Example: PlanetLab

Figure 3-15. The basic organization of a PlanetLab node.
Reasons for Migrating Code

- Improve computing performance by moving processes from heavily-loaded machines to lightly loaded machines

- Improve communication times by shipping code to systems where large data sets reside. E.g. a client ships code to a database server or vice versa

- **Mobile Agents**: small piece of code that moves from site to site for a web search. Several copies can be made to improve performance

- **Flexibility to dynamically configure distributed systems.** E.g. a server can provide interface code to a client dynamically. This does the require that the protocol for downloading and initializing the code is standardized. Allows the interface to be changed as often as desired without having to rebuild applications or servers
Figure 3-17. The principle of dynamically configuring a client to communicate to a server. The client first fetches the necessary software, and then invokes the server.
Code Migration Models

- A process consists of three segments: code segment, resource segment, execution segment.

- **Weak mobility**: Only the code segment (and some initialization data) can be transferred. Transferred program always starts at one of the predefined starting positions. E.g. java applets.

- **Strong mobility**: Code and execution segments can both be transferred.

- **Sender-initiated** versus **receiver-initiated**: Uploading code to a server versus downloading code from a server by a client.
Models for Code Migration

Figure 3-18. Alternatives for code migration.
Process to Resource Binding

Three ways to handle migration (which can be combined)

- **Binding by identifier**: A process requires precisely the referenced resource. For example using a URL for a website or a FTP server

- **Binding by value**: The process can continue to execute if it gets the same value from another resource. E.g. A standard library

- **Binding by type**: The process only needs a resource of specific type. E.g. a printer
Resource to Machine Binding

Three ways to handle migration (which can be combined)

- **Unattached resources**: can be easily moved between machines. E.g. Data files

- **Fastened resource**: Can only be moved at a high cost. E.g. a local database, a web site

- **Fixed resource**: Intimately bound to a specific machine and environment and cannot be moved. E.g. Local devices, local communication end point (a socket bound to an address and port)
### Migration and Local Resources

<table>
<thead>
<tr>
<th>Process-to-resource binding</th>
<th>Resource-to-machine binding</th>
<th>Unattached</th>
<th>Fastened</th>
<th>Fixed</th>
</tr>
</thead>
<tbody>
<tr>
<td>By identifier</td>
<td>MV (or GR)</td>
<td>GR (or MV)</td>
<td>GR</td>
<td></td>
</tr>
<tr>
<td>By value</td>
<td>CP (or MV,GR)</td>
<td>GR (or CP)</td>
<td>GR</td>
<td></td>
</tr>
<tr>
<td>By type</td>
<td>RB (or MV,CP)</td>
<td>RB (or GR,CP)</td>
<td>RB (or GR)</td>
<td></td>
</tr>
</tbody>
</table>

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

**Figure 3-19.** Actions to be taken with respect to the references to local resources when migrating code to another machine.
Migration in Heterogeneous Systems

Use virtual machines. Then we have three ways to handle migration (which can be combined)

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

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

- Letting the new virtual machine pull in new pages as needed, that is, let processes start on the new virtual machine immediately and copy memory pages on demand.