Processes

• Threads
• Clients
• Servers
• Code migration
• Software agents

Threads

• Program in execution:
  – Virtual processor
• Processor context
  – Values needed to execute an instruction
• Thread context
  – Values needed to execute a series of instructions
• Process context
  – Values needed to manage a series of instructions in the operating system
Context Switching

• 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 vs. OS

• User-space solution:
  – 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 vs. OS

• Kernel solution:
  – have the kernel contain the implementation of a thread package, i.e. 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 vs. OS**

- **Compromise solution:**
  - two-level threading approach: lightweight processes that can execute user-level threads.

**LWP**

- 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.
Threads and Distributed Systems

- **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 RPCs:**
  - A client does several RPCs at the same time, each one by a different thread
  - It then waits until all results have been returned
  - Note: if RPCs are to different servers, we may have a linear speed-up compared to doing RPCs one after the other

Threads and Distributed Systems

- **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
Multithreaded Servers (1)

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>

Three ways to construct a server.
Clients

- User interface
  - Ex: X Window System
- Transparency issues

The X-Window System

Xlib uses network-aware X protocol
Compound documents

- Make the user interface application-aware to allow inter-application communication:
  - **drag-and-drop**: move objects to other positions on the screen, possibly invoking interaction with other applications
  - **in-place editing**: integrate several applications at user-interface level (word processing + drawing facilities)

Client-Side Software

A possible approach to transparent replication of a remote object using a client-side solution.
Other distribution transparencies

- access transparency:
  - client-side stubs for RPCs and RMIs
- location/migration transparency:
  - let client-side software keep track of actual location
- failure transparency:
  - can often be placed only at client (we’re trying to mask server and communication failures).

Servers

- Server design issues
- Object servers
Server design

• **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:
    – 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
    – imap 143 mail access
    – http 80 webserver

Server design

• **Iterative vs. concurrent servers:**
  – Iterative servers can handle only one client at a time, in contrast to concurrent 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
Super Servers

(a) Client-to-server binding using a daemon as in DCE
(b) Client-to-server binding using a superserver as in UNIX

Out-of-Band Communication

**Issue:** Is it possible to *interrupt* a server once it has accepted a service request?

**Solution 1:**
Use a separate port for urgent data:
- Server has a separate thread waiting for incoming urgent messages
- When urgent message comes in, associated request is put on hold
- Note: requires OS support of high-priority scheduling

**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

- **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?

- **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.
Object Server

**Servant:**
- The actual implementation of an object, sometimes containing only method implementations:
  - Collection of C or COBOL functions, that act on structs, records, database tables, etc.
  - Java or C++ classes

**Skeleton:**
- Server-side stub for handling network I/O:
  - Unmarshalls incoming requests, and calls the appropriate servant code
  - Marshalls results and sends reply message
  - Generated from interface specifications

**Object adapter:**
- The “manager” of a set of objects:
  - Inspects (as first) incoming requests
  - Ensures referenced object is activated (requires identification of servant)
  - Passes request to appropriate skeleton, following specific activation policy
  - Responsible for generating object references

Object Adapter (1)

Organization of an object server supporting different activation policies.
/* Definitions needed by caller of adapter and adapter */
#define TRUE
#define MAX_DATA 65536

/* Definition of general message format */
struct message {
    long source; /* sender's identity */
    long object_id; /* identifier for the requested object */
    long method_id; /* identifier for the requested method */
    unsigned size; /* total bytes in list of parameters */
    char **data; /* parameters as sequence of bytes */
};

/* General definition of operation to be called at skeleton of object */
typedef void (*METHOD_CALL)(unsigned, char[], char[]);

long register_object (METHOD_CALL call); /* register an object */
void unregister_object (long object_id); /* unregister an object */
void invoke_adapter (message *request); /* call the adapter */

The header.h file used by the adapter and any program that calls an adapter.

typedef struct thread THREAD; /* hidden definition of a thread */

thread *CREATE_THREAD (void (*body)(long tid), long thread_id);
/* Create a thread by giving a pointer to a function that defines the actual */
/* behavior of the thread, along with a thread identifier */

void get_msg (unsigned *size, char **data);
void put_msg(THREAD *receiver, unsigned size, char **data);
/* Calling get_msg blocks the thread until a message has been put into its */
/* associated buffer. Putting a message in a thread's buffer is a nonblocking */
/* operation. */

The thread.h file used by the adapter for using threads.
Object Adapter (4)

The main part of an adapter that implements a thread-per-object policy.

Code Migration

- Approaches to code migration
- Migration and local resources
- Migration in heterogeneous systems
Reasons for Migrating Code

Goal: bring code to data and/or compute capability

Strong and Weak Mobility

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.
Models for Code Migration

Alternatives for code migration.

Managing Local Resources

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)
Migration and Local Resources

### Resource-to machine binding

<table>
<thead>
<tr>
<th>Process-to-resource 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>
</tr>
<tr>
<td>By value</td>
<td>CP (or MV, GR)</td>
<td>GR (or CP)</td>
<td>GR</td>
</tr>
<tr>
<td>By type</td>
<td>RB (or GR, CP)</td>
<td>RB (or GR, CP)</td>
<td>RB (or GR)</td>
</tr>
</tbody>
</table>

Actions to be taken with respect to the references to local resources when migrating code to another machine.

---

**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)
- Existing languages: allow migration at specific “transferable” points, such as just before a function call.
Migration in Heterogeneous Systems

The principle of maintaining a migration stack to support migration of an execution segment in a heterogeneous environment.

Example: D’Agents

<table>
<thead>
<tr>
<th></th>
<th>Agents</th>
<th>Tcl/Tk interpreter</th>
<th>Scheme interpreter</th>
<th>Java interpreter</th>
</tr>
</thead>
<tbody>
<tr>
<td>5</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>4</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>3</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td></td>
<td>TCP/IP</td>
<td>E-mail</td>
<td></td>
</tr>
</tbody>
</table>

Layered architecture
D'Agents: weak mobility

```bash
proc factorial n {
    if ($n ≤ 1) { return 1; } # fac(1) = 1
    expr $n * [ factorial [expr $n – 1] ] # fac(n) = n * fac(n – 1)
}
set number … # tells which factorial to compute
set machine … # identify the target machine
agent_submit $machine –procs factorial –vars number –script {factorial $number }
agent_receive … # receive the results (left unspecified for simplicity)
```

D'Agents: strong mobility

```bash
all_users $machines
proc all_users machines {
    set list ** # Create an initially empty list
    foreach m $machines { # Consider all hosts in the set of given machines
        agent_jump $m # Jump to each host
        set users [exec who] # Execute the who command
        append list $users # Append the results to the list
    }
    return $list # Return the complete list when done
}
set machines … # Initialize the set of machines to jump to
set this_machine # Set to the host that starts the agent
# Create a migrating agent by submitting the script to this machine, from where
# it will jump to all the others in $machines.
agent_submit $this_machine –procs all_users -vars machines
    -script { all_users $machines }
agent_receive … # receive the results (left unspecified for simplicity)
```
Implementation Issues

<table>
<thead>
<tr>
<th>Status</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Global interpreter variables</td>
<td>Variables needed by the interpreter of an agent</td>
</tr>
<tr>
<td>Global system variables</td>
<td>Return codes, error codes, error strings, etc.</td>
</tr>
<tr>
<td>Global program variables</td>
<td>User-defined global variables in a program</td>
</tr>
<tr>
<td>Procedure definitions</td>
<td>Definitions of scripts to be executed by an agent</td>
</tr>
<tr>
<td>Stack of commands</td>
<td>Stack of commands currently being executed</td>
</tr>
<tr>
<td>Stack of call frames</td>
<td>Stack of activation records, one for each running command</td>
</tr>
</tbody>
</table>

The parts comprising the state of an agent in D'Agents.

Software Agents in Distributed Systems

<table>
<thead>
<tr>
<th>Property</th>
<th>Common to all agents?</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Autonomous</td>
<td>Yes</td>
<td>Can act on its own</td>
</tr>
<tr>
<td>Reactive</td>
<td>Yes</td>
<td>Responds timely to changes in its environment</td>
</tr>
<tr>
<td>Proactive</td>
<td>Yes</td>
<td>Initiates actions that affects its environment</td>
</tr>
<tr>
<td>Communicative</td>
<td>Yes</td>
<td>Can exchange information with users and other agents</td>
</tr>
<tr>
<td>Continuous</td>
<td>No</td>
<td>Has a relatively long lifespan</td>
</tr>
<tr>
<td>Mobile</td>
<td>No</td>
<td>Can migrate from one site to another</td>
</tr>
<tr>
<td>Adaptive</td>
<td>No</td>
<td>Capable of learning</td>
</tr>
</tbody>
</table>

Some important properties by which different types of agents can be distinguished.
Agent Technology

The FIPA model of an agent platform

Agent Communication Languages (1)

<table>
<thead>
<tr>
<th>Message purpose</th>
<th>Description</th>
<th>Message Content</th>
</tr>
</thead>
<tbody>
<tr>
<td>INFORM</td>
<td>Inform that a given proposition is true</td>
<td>Proposition</td>
</tr>
<tr>
<td>QUERY-IF</td>
<td>Query whether a given proposition is true</td>
<td>Proposition</td>
</tr>
<tr>
<td>QUERY-REF</td>
<td>Query for a give object</td>
<td>Expression</td>
</tr>
<tr>
<td>CFP</td>
<td>Ask for a proposal</td>
<td>Proposal specifics</td>
</tr>
<tr>
<td>PROPOSE</td>
<td>Provide a proposal</td>
<td>Proposal</td>
</tr>
<tr>
<td>ACCEPT-PROPOSAL</td>
<td>Tell that a given proposal is accepted</td>
<td>Proposal ID</td>
</tr>
<tr>
<td>REJECT-PROPOSAL</td>
<td>Tell that a given proposal is rejected</td>
<td>Proposal ID</td>
</tr>
<tr>
<td>REQUEST</td>
<td>Request that an action be performed</td>
<td>Action specification</td>
</tr>
<tr>
<td>SUBSCRIBE</td>
<td>Subscribe to an information source</td>
<td>Reference to source</td>
</tr>
</tbody>
</table>

FIPA message types
Agent Communication Languages (2)

<table>
<thead>
<tr>
<th>Field</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Purpose</td>
<td>INFORM</td>
</tr>
<tr>
<td>Sender</td>
<td>max@<a href="http://fanclub-beatrix.royalty-spotters.nl:7239">http://fanclub-beatrix.royalty-spotters.nl:7239</a></td>
</tr>
<tr>
<td>Receiver</td>
<td>elke@iiop://royalty-watcher.uk:5623</td>
</tr>
<tr>
<td>Language</td>
<td>Prolog</td>
</tr>
<tr>
<td>Ontology</td>
<td>genealogy</td>
</tr>
<tr>
<td>Content</td>
<td>female(beatrix),parent(beatrix,juliana,bernhard)</td>
</tr>
</tbody>
</table>

A simple example of a FIPA message sent between two agents using Prolog to express genealogy information.