ECE151 – Lecture 4

Chapter 3
Processes
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.
Virtualization
Context Switching

**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).
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 than thread switching; 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

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

**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.
Thread Usage in Nondistributed Systems

Context switching as the result of IPC

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

Combining kernel-level lightweight processes and user-level threads.

[Diagram showing the relationship between thread states and lightweight processes.]
Solaris Threads

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.
User Interfaces

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

**Compound documents:** Make the user interface application-aware to allow interapplication 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)
The X-Window System

- Server machine
  - Application
  - Xlib
  - Xlib interface
  - X protocol
  - Terminal (includes display keyboard, mouse, etc.)

- Client machine
  - X kernel
  - Device drivers
Client-Side Software

**Essence:** Often focused on providing distribution transparency. Access transparency: client-side stubs for RPCs and RMIs. Location/migration transparency: let client-side software keep track of actual location. Replication transparency: multiple invocations handled by client stub. Failure transparency: can often be placed only at client (we’re trying to mask server and communication failures).
Client-Side Software for Distribution Transparency

A possible approach to transparent replication of a remote object using a client-side solution.
**Servers**

**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
  
  24 any private mail system
- **smtp** 25 Simple Mail Transfer
- **login** 49 Login Host Protocol
- **sunrpc** 111 SUN RPC (portmapper)
- **courier** 530 Xerox RPC
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 (UNIX).

Iterative vs. concurrent servers: Iterative servers can handle only one client at a time, in contrast to concurrent servers.
Servers: General Design Issues

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 (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

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

**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 Servers

**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

Organization of an object server supporting different activation policies.
Object Adapter

/* Definitions needed by caller of adapter and adapter */
#define TRUE
#define MAX_DATA 65536

/* Definition of general message format */
struct message {
    long source /* senders 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* unsigned*, 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 */
Object 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 of 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

The main part of an adapter that implements a thread-per-object policy.
Reasons for Migrating Code

The principle of dynamically configuring a client to communicate to a server. The client first fetches the necessary software, and then invokes the server.

1. Client fetches code
2. Client and server communicate
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

Mobility mechanism

Weak mobility
- Sender-initiated mobility
  - Execute at target process
  - Execute in separate process

- Receiver-initiated mobility
  - Execute at target process
  - Execute in separate process

Strong mobility
- Sender-initiated mobility
  - Migrate process
  - Clone process
- Receiver-initiated mobility
  - Migrate process
  - Clone process
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

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

Actions to be taken with respect to the references to local resources when migrating code to another machine.
Migration in Heterogeneous 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.
What’s an Agent?

**Definition:** An autonomous process capable of reacting to, and initiating changes in its environment, possibly in collaboration with users and other agents

- **collaborative agent:** collaborate with others in a multiagent system
- **mobile agent:** can move between machines
- **interface agent:** assist users at user-interface level
- **information agent:** manage information from physically different sources
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 general model of an agent platform (adapted from [fipa98-mgt]).

**Management:** Keeps track of where the agents on this platform are (mapping agent ID to port)

**Directory:** Mapping of agent names & attributes to agent IDs

**ACC:** Agent Communication Channel, used to communicate with other platforms
Agent Communication Languages (1)

Examples of different message types in the FIPA ACL [fipa98-acl], giving the purpose of a message, along with the description of the actual message content.

<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>
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 ACL message sent between two agents using Prolog to express genealogy information.
Overview of Code Migration in D'Agents (1)

A simple example of a Tel agent in D'Agents submitting a script to a remote machine (adapted from [gray.r95])

```
proc factorial n {
    if ($n \leq 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)
```
Overview of Code Migration in D'Agents (2)

An example of a Tel agent in D'Agents migrating to different machines where it executes the UNIX `who` command (adapted from [gray.r95])

```tcl
all_users $machines

proc all_users machines {
    set list ""
    foreach m $machines {
        agent_jump $m
        set users [exec who]
        append list $users
    }
    return $list
}

set machines …
set this_machine

# 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 (1)

The architecture of the D'Agents system.

<table>
<thead>
<tr>
<th></th>
<th>Agents</th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>5</td>
<td></td>
<td>Tcl/Tk interpreter</td>
<td>Scheme interpreter</td>
<td>Java interpreter</td>
</tr>
<tr>
<td>4</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>Common agent RTS</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>Server</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>TCP/IP</td>
<td></td>
<td>E-mail</td>
<td></td>
</tr>
</tbody>
</table>
Implementation Issues (2)

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

<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>