Processes
Distributed Systems
Santa Clara University 2016
Processes

• Issues OS: Management and scheduling of processes
• Issues DS:
  • Client Server Architectures
  • Moving processes between servers
  • Software agents
Threads

- **Process:**
  - Has own memory space
  - Has an entry in process table
  - Protected against other processes (Transparency)

- **Thread:**
  - Share memory space
  - Thread context is just its CPU context plus thread management
    - (e.g. thread is blocked because it waits on mutex)
  - No protection against other threads of the same process
Threads

• Multi-threaded application
  • does not block when system call is executed
  • easier to exploit parallelism of multiple CPU
• IPC requires context switching
• But thread communication can be done via shared memory
• Are easier to structure
Threads

- Thread package
  - Operation to create and destroy threads
  - Operations on synchronization variables

- Implementation:
  - Execute thread library in user space
  - Or: Have kernel be thread aware and schedule threads
Threads

• User-level threads:
  + Switching thread context takes only few instructions
    • Need to store / reload values of CPU registers
  - Blocking system call blocks the entire process
    • The thread that called and all others
Threads

• Implement threads in OS kernel
  + Blocking call only blocks one thread
  - All thread operations need now switch to kernel
Threads

• Hybrid version:
  - *lightweight processes (LWP)*
  - LWP runs in the context of a single process
  - Several LWPs per process
  - System can allow user-level thread package
Threads

• Each LWP runs its own thread
• Thread package has a single routine to schedule the next thread
• LWPs have their own stack, …
• LWPs share thread table
• LWP look for runnable threads and run them
• If a thread blocks, it does the necessary administration and calls scheduling routine
• LWP switches to another thread, but in user space
Threads

- Thread makes blocking system call
- Execution switches to kernel mode but retains current LWP context
- OS decides to switch context to another LWP, which switches back to user mode
Threads

• LWP:
  • Creating, destroying and synchronizing threads is cheap: no kernel intervention
  • Blocking system usually does not suspend the entire process
  • Application does not need to know about LWP
  • Easy to adopt to multiprocessing environments

• Drawback: Still need to create and destroy LWP
Threads

• Threads in distributed systems
  • Threads allow blocking system calls without blocking the entire process
  • Useful for communication and maintaining several logical connections
Threads

• Multithreaded clients
  • Hide communication latencies by immediately proceeding with something else
• Web browser
  • Display data while data is still coming in
  • Can use several connections at the same time
Threads

- Multithreaded server in the dispatcher/worker model
Threads

- If multithreading is not possible
  - Use a Finite-State Machine
    - When request or answer from resource is received
    - Change state

<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>
Servers

- Iterative server
  - Server handles request and returns response
- Concurrent server
  - Server passes request to separate thread or process
  - Returns to waiting for requests
Servers

- Clients send requests to endpoints (ports)
  - How do they find endpoints
    - Globally assigned endpoints
    - Daemon (DCE) runs at a well-known endpoint and keeps track of current endpoints
    - Super Server: Creates actual servers on demand
Servers

(a) Client machine

Client

1. Ask for endpoint

DCE daemon

2. Request service

Server

Register endpoint

Endpoint table

(b) Client machine

Client

1. Request service

Server machine

Super-server

Actual server

Create server for requested service

2. Continue service
Servers

- UNIX inetd is such a superserver
Servers

- How to interrupt a service
  - Kill application
  - Have out-of-band communication on different end point
  - Have out-of-band communication using same stream
    - URGENT flag in TCP
Servers

• Stateless and stateful servers

• How are stateful servers implemented in web servers?
Servers

• Object servers
  • Tailored to support distributed object
    • Server does not directly provide service
    • It's a place where objects live.
Servers

- Invoking objects
  - DCE: all objects have only one way to be invoked
  - Allow different policies
    - Transient objects:
      - Can be created at first invocation and destroyed when there are no longer clients bound to it
      - Can be created and kept around
    - Memory segments:
      - Objects do not share memory
      - Object share their code
  - Threading
    - Objects are single threaded
    - Objects are multithreaded
Servers

- Activation policies: How to invoke an object
- **Object adapter / wrapper**
  - Generic components to assist developers of distributed objects
  - Configured for a specific policy
- Usually several object adapters coexist
Servers

Object adapters don’t know the interface of the objects they control.
Servers

• Adapter:
  • expects that each skeleton implements the operation

\[
\text{invoke}(\text{unsigned in\_size}, \text{char in\_args[]}\text{, unsigned* out\_size, char* out\_args[]})
\]

• Actual format of array is only known to stub
Servers

/* Definitions needed by caller of adapter and adapter */
#define TRUE 1
#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*, 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 */

Header file for adapter and all programs that call an adapter
Code Migration

• Reasons for migrating code
  • Move processes from heavily to lightly loaded machines
  • Move processes from server to client
  • Dynamically configure distributed systems
Code Migration
Code Migration

• Process consists of:
  • code segment
  • resource segment
    • references to external resources
  • execution segment
    • current execution state of a process
      • private data
      • stack
      • program counter
Code Migration

- Weak mobility
  - Transfer only the code segment
  - maybe some initialization data
    - E.g.: Java applets
- Strong mobility
  - Execution segment can be transferred as well
  - E.g. D'Agent
Code Migration

- Sender initiated migration
- Receiver initiated migration
  - E.g. Java applets
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
  - Receiver-initiated mobility
    - Migrate process
    - Clone process
Code Migration

• Migration and local resources
  • Process to resource binding
    • Binding by identifier:
      • E.g. URL, FTP server by internet address
    • Binding by value:
      • Standard library is available on all servers, but its location differs
  • Binding by type:
    • E.g. references to monitors, printers, …
Code Migration

- Code migration needs to change references to resources
- Unattached resources can be easily moved
  - E.g. data files
- Fastened resources
  - Movement very difficult
  - E.g. local databases
- Fixed resources
  - Cannot be moved
Code Migration

<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 MV,CP)</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
Code Migration

• Moving the execution segment between heterogeneous machines
  • Only move when subroutines are called
  • Create a migration stack and send the migration stack
    • Need to marshal
Code Migration

Principle of maintaining a migration stack

[Diagram showing the process of code migration with labels for procedures A and B, local stack operations, and a migration stack.]
Code migration

• To use a migration stack
  • Compiler needs to
    • emit executables that update the migration stack
    • generate labels in the code to allow return from subroutine
  • Needs a suitable runtime system

• Example: Dimitrov and Rego (1998)
  • Migrate C/C++ programs by slightly modifying the language
  • Using only a preprocessor to insert the code to maintain the migration stack
Code Migration

• Historical perspective
  • 1970s: Porting Pascal to different machines
    • Generate intermediate code
      • Machine-independent
  • 1990s: Advent of scripting and highly portable languages
Code Migration

- Code migration is easier with
  - scripting languages
  - Java
- because they rely on a virtual machine
  - Data are represented the same way
  - Code is represented the same way
  - Architecture is the same
Code Migration

• Example: D’Agents
  • Middleware platform with support for code migration
  • Based on the idea of software agents

• Agent is a program that can migrate between different machines
Code Migration

• D’Agent implements:
  • Sender initiated weak mobility
  • Strong mobility by process migration
  • Strong mobility by process cloning
Code Migration

• D’Agent weak mobility:
  • agent_submit

• Parameters
  • target machine
  • script
Code Migration

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)
Code Migration

• D’Agent submit example
  • Simple Tcl agent submits script to a remote machine
  • Procedure factorial takes a single parameter
  • Variables number and machine need to be set by user
• D’Agent target sends result back to agent

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)
Code Migration

• D’Agent sender initiated mobility through process migration
  • Migrating an agent
    • Local machine
      • Agent calls $\text{agent\_jump}$
      • Suspend execution at sender
      • Marshal resource, code, and execution segment and send to the target machine
      • Agent exits
    • When received at the remote machine
      • New process with right interpreter is started
      • New process unmarshals message
      • Continues execution at instruction after $\text{agent\_jump}$
Code Migration

D’Agent migrates to different machines where it executes whois

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 ...

# Initialize the set of machines to jump to
# 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)
Code Migration

• D’Agent strong mobility by cloning
  • `agent_fork` command
    • Migrates a copy of the process to a remote machine
    • But local process continues
    • Behavior like UNIX `fork`
Code Migration

• D’Agent implementation
  1. Corresponds to Berkeley sockets
  2. Server for
     • Agent management
     • Authentication
     • Communication between agents:
       • Each agent has a location-unique identifier
       • Addresses are: (machine, local-id)
Code Migration

3. Language independent core
   • Start / end agent
   • various migration operations
   • facilities for inter-agent communication

4. Interpreters for different languages
   • Interpreter consists of
     • Component of language interpretation
     • Security module
     • Interface to core layer
     • Module to capture the state of a running agent
       • Needed for supporting strong mobility
Code Migration

- D’Agent implementation

5. Collection of agents written in a supported language
  - Each agent is executed in a separate process

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

- D’Agent: How to capture the state of a running agent
- Tcl: agent state contains

<table>
<thead>
<tr>
<th>State</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>
Code Migration

- D’Agent capturing state:
  - Agent is a series of Tcl commands
  - Commands are executed one by one
    - Basic command
      - Interpreter parses command building record
      - Interpreter pushes record on command stack
      - Record has all the information needed to execute the command
      - Interpreter passes execution to the component responsible for executing the command
    - Ergo: Command stack gives the precise account of current execution status
Code Migration

• D’Agent capturing state:
  • Agent is a series of Tcl commands
  • Commands are executed one by one
    • User defined procedure
      • use/create activation records / call frames
      • Call frames create reference to its calling command
    • Ergo: Call frame stack gives the precise account of current pending procedure calls
Code Migration

- D’Agent code migration
  - Complete state of the agent is in state
  - Marshaled into bytes
  - Sent to other machine
  - Unmarshaled to create copy / new clone of migrating process

<table>
<thead>
<tr>
<th>State</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>
Software Agents

• Definition of a software agent
  • No taxonomy exists
  • Important aspects:
    • Autonomy
    • Cooperation
Software Agents

- Mobile agent
  - can move to other machines
- Interface agents
  - Agents that assist an end user
  - Should have learning capability to better assist user
- Information agents
  - Help manage information from many different sources
# Software Agents

<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 affect 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 life span</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>