Processes

Topics to be covered

Threads
Clients and Servers
Code Migration
Mobile Agents

Introduction: Processes

To execute a program, the OS creates a number of virtual processors

Process table
Process: program in execution

Concurrency Transparency

Kernel: supervisor mode
Other processes in user mode
System calls

Each process an address space (a collection of ranges of virtual memory locations)

Introduction: Processes vs Threads

Execution environment:
- an address space
- higher level resources
- CPU context

Process context: CPU context (register values, program counter, stack pointer), registers of the memory management unit (MMU)

- Expensive creation
- Expensive context switch (may also require memory swap)

Split a process in threads

Introduction: Processes

To execute a program, the OS creates a number of virtual processors

Process table
Process: program in execution

Concurrency Transparency

Kernel: supervisor mode
Other processes in user mode
System calls

Each process an address space (a collection of ranges of virtual memory locations)

Introduction: Processes vs Threads

Execution environment:
- an address space
- higher level resources
- CPU context

Process context: CPU context (register values, program counter, stack pointer), registers of the memory management unit (MMU)

- Expensive creation
- Expensive context switch (may also require memory swap)

Split a process in threads
**Introduction: Processes vs Threads**

- Threads: no attempt to provide concurrency transparency executed in the same address space.

  - **Threads of the same process share the same execution environment.**

- **Thread context:** only the CPU context (+information for thread management)
  - No performance overhead, but
  - Harder to use, require extra effort to protect against each other.

**More on process creation**

- Two issues:
  - Creation of an execution environment (and an initial thread within it)
  - Share, Copy, Copy-on-Write

  - Choice of target host (in distributed systems)
  - Transfer Policy (local or remote host)
  - Location Policy (to which node to transfer)
  - Load sharing (sender vs receiver initiated)

**Creating a new thread within an existing process is cheaper than creating a process (~10-20 times)**

- **Traditional Unix process**
  - Child processes created from a parent process using the command `fork`.

- **Drawbacks:**
  - `fork` is expensive: Memory is copied from a parent to its children. Logically a child process has a copy of the memory of the parent before the fork (with copy-on-write semantics).
  - Communication after the fork is expensive: Inter process communication is needed to pass information from parent to children and vice versa after the fork has been done.

- **Threads:**
  - Creation 10 to 100 times faster than process creation
  - Shared memory: all threads within a given process share the same memory and files.

- **Switching to a different thread within the same process is cheaper than switching between threads belonging to different processes (5-50 times)**

  - Threads within a process may share data and other resources conveniently and efficiently compared with separate processes (without copying or messages).
  - Threads within a process are not protected from one another.

**Threads versus multiple processes**

- State associated with execution environments and threads

<table>
<thead>
<tr>
<th>Execution Environment</th>
<th>Thread</th>
</tr>
</thead>
<tbody>
<tr>
<td>Address space tables</td>
<td>Saved processor registers</td>
</tr>
<tr>
<td>Communication interfaces (e.g., sockets), open files</td>
<td>Priority and execution state (such as SLEEPING)</td>
</tr>
<tr>
<td>Semaphores, other synchronization objects</td>
<td>Software interrupt handling information</td>
</tr>
<tr>
<td>List of thread identifiers</td>
<td>Execution environment identifier</td>
</tr>
<tr>
<td>Pages of address space resident in memory, hardware cache entries</td>
<td></td>
</tr>
</tbody>
</table>

**Threads concept and implementation**

- A process can have many threads sharing an execution environment.

- **Process**
  - **Thread**
  - **Activation stack** (parameters, local variables)
  - **Heap** (dynamic storage, objects, global variables)
  - System-provided resources (sockets, windows, open files)
Thread Implementation

Generally provided in the form of a thread package

Operations to create and destroy threads as well as operations on synchronization variables (e.g., mutexes and condition variables)

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

Thread implementation

Threads can be implemented:
- in the OS kernel (Win NT, Solaris, Mach)
- at user level (e.g. by a thread library: C threads, pthreads), or in the language (Ada, Java)
  - lightweight - no system calls
  - modifiable scheduler
  - low cost enables more threads to be employed
  - not pre-emptive, cannot schedule threads within different procedures
  - Cannot exploit multiple processors
  - Blocking system calls (page fault) blocks the process and thus all threads

Java can be implemented either way
- hybrid approaches can gain some advantages of both
- user-level hints to kernel scheduler
- hierarchic threads (Solaris)
- event-based (SPIN, FastThreads)

User-space solution:

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

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.

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 trip to the kernel.

Conclusion: Try to mix user-level and kernel-level threads into a single concept.

User-level vs Kernel-level Threads (summary)

(-) threads within a process can take advantage of a multiprocessor
(-) A thread that takes a page fault blocks the entire process and all threads within it
(-) Threads within different processes cannot be scheduled according to a single scheme of relative prioritization
(+) Certain thread operations (e.g switching) cost less: do not involve a system call
(+) Thread scheduling can be customized
(+) More many threads can be supported

Thread Implementation (Solaris)

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

An LWP runs in the context of a single (heavy-weight) process * a user-level thread package (create, destroy threads, thread synchronization)

Assign a thread to an LWP (hidden from the programmer)

When an LWP is created (through a system call) gets its own stack and executes a scheduling routine (that searches for a thread to execute)

If many LWPs, each executes the scheduler, they share a thread table (with the current set of threads), synchronization among them in user space.

When an LWP finds a thread, it switches context to it
Thread Implementation (LWP)

When a thread calls a blocking user-level operation (e.g., blocks on a mutex or a condition variable), it calls the scheduling routine, when another runnable thread is found, a context switch is made to that thread which is then bound to the same LWP (the LWP that is executing the thread need not be informed).

When a user-level thread does a blocking 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 there are no threads to schedule, an LWP may remain idle, and may even be removed (destroyed) by the kernel.

Thread Implementation

Most modern OSs support threads, either with their own thread library or through POSIX pthreads.

Each OS uses a different technique to support threads.

- X-to-Y model. The mapping between LWPs and Threads.

  - Solaris uses the many-to-many model. All CPUs are mapped to any number of LWPs which are then mapped to any number of threads. The kernel schedules the LWPs for slices of CPU time.

  - Linux uses the one-to-one model. Each thread is mapped to a single LWP. Linux LWPs are really lightweight and thus LWP creation is not as expensive as in Solaris. In Linux, the scheduler gives a 1 point boost to “processes” scheduled which are in the same thread family as the currently running process.

Why threads?

In a single-threaded system process whenever a blocking system call is executed, the process as a whole is blocked.

Exploit parallelism when executing a program on a multiprocessor system (assign each thread to a different CPU)

Multithreaded Servers

Example

Each request takes on average 2msecs of processing and 8 msecs of I/O delay (no caching)

Maximum server throughput (measured as client requests handled per sec)

- Single thread
  - Turnaround time for each request: 2 + 8 = 10 msecs
  - Throughput: 100 req/sec

- Two threads
  - (if disk requests are serialized)
  - Turnaround time for each request: 8 msecs
  - Throughput: 125 req/sec

Assume disk block caching, 75% hit rate

Example (continued)

Two threads

- Mean I/O time per request: 0.75 * 0 + 0.25 * 8 msecs = 2msecs
- Throughput: 500 req/sec

But the processor time actually increases due to caching, say to 2.5
- Throughput: 400 req/sec
Example (continued)

Assume shared memory multiprocessor

Two processors, one thread at each

Two threads

Mean I/O for each request remains: 0.75 * 0 + 0.25 * 8 msec = 2msec
Processing time per request: 2.5msec
But two process executed in parallel -> ?? (444 req/sec prove it?)

Throughput:

More than two threads

Bounded by the I/O time (2msec per process) thus,
Max throughput: 500 req/sec

Multithreaded Servers

The worker pool architecture

The server creates a fixed pool of "worker" threads to process the requests when it starts up.

One I/O thread receives requests from a collection of ports and places them on a shared request queue for retrieval by the workers.

Prioritize multiple queues in the worker pool

Disadvantage: high level of switching between the I/O pool and the workers; limited number of worker threads

Multithreaded Servers

One thread-per-request architecture

The I/O thread spawns a new worker thread for each request.
The worker destroys itself when it has processed the request.

Threads do not contend for a shared queue and as many workers as outstanding requests

Disadvantage: overhead of creating and destroying threads

Multithreaded Servers

Thread-per-connection

The server creates a new worker thread when a client makes a connection and destroys the thread when the client closes the connection. In between the client can make many requests over the connection.

Lower thread management

Clients may delay while a worker has several requests but another thread has no work to perform.

Multithreaded Servers

Alternative multi-server architectures (summary)

Implemented by the server-side ORB in CORBA

(a) would be useful for UDP-based service, e.g. NTP

(b) most commonly used; matches the TCP connection model

(c) used where the service is encapsulated as an object. E.g. could have multiple shared whiteboards with one thread each. Each object has only one thread, avoiding the need for thread synchronization within objects.
Multithreaded Servers

<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.

Main issue is improved performance and better structure

改善性能
- 开始一个线程来处理传入请求比启动一个新的进程便宜得多。
- 具有单线程服务器的系统无法简单地将服务器扩展到多处理器系统。
- 隐藏网络延迟，当之前的请求被回复时，立即开始下一个请求。

更好地结构
- 大多数服务器有很高的I/O需求。使用简单、易于理解的阻塞调用简化了整体结构。
- 多线程程序往往更小、更容易理解，因为控制流程更简单。

Multithreaded Clients

主问题在于隐藏网络延迟

Multithreaded Web client
- 网页浏览器扫描传入的HTML页面，并发现需要更多的文件。
- 每个文件由一个单独的线程处理，每个线程执行一个（阻塞）HTTP请求。
- 文件到达时，浏览器显示它们。

Multiple RPCs
- 一个客户可以同时执行多个RPC，每个RPC由不同的线程执行。
- 第一个线程生成结果，然后通过（阻塞）RPC调用传递给服务器。
- 第二个线程执行RPC调用。
- 两个线程共享队列。

Concurrent Programming

并发编程
- 概念：
  - 赛道条件
  - 临界部分
  - 互斥锁
  - 条件变量
  - 信号量
- 在常规语言如C中，通过线程库（POSIX）或Java线程类实现。

Threads Programming

线程编程
- 互斥锁
  - 避免了修改一个变量，该变量已经在另一个线程中被修改或读取到旧值的风险。
  - 附带锁以访问资源。锁的产生和释放。
  - 死锁。
  - 优先级反转。
  - 多个读取锁，写者饥饿。
  - 死锁。
  - 优先级反转。
  - 多个读取锁，写者饥饿。
Race conditions occur when multiple threads share data and at least one of the threads accesses the data without going through a defined synchronization mechanism. Could result in erroneous results. Whether a library call is safe to use in reentrant code (reentrant code means that a program can have more than one thread executing concurrently).

Thread Synchronization Primitives besides mutexes

Condition Variables
- Allow threads to synchronize to a value of a shared resource
- Provide a kind of notification system among threads
- `wait` on the condition variable
- Other threads `signal` this condition variable or `broadcast` to signal all threads waiting on the condition variable

Thread Synchronization Primitives besides mutexes

Condition Variables
- Allow threads to synchronize to a value of a shared resource
- Provide a kind of notification system among threads
- `wait` on the condition variable
- Other threads `signal` this condition variable or `broadcast` to signal all threads waiting on the condition variable

Threads Programming

Spinlocks
- Frequently in the Linux kernel; less commonly used at the user-level
- A spinlock basically spins on a mutex. If a thread cannot obtain the mutex, it will keep polling the lock until it is free.
- If a thread is about to give up a mutex, you don’t have to context switch to another thread. However, long spin times will result in poor performance.
- Should never be used on uniprocessor machines. Why?

Threads Programming

Spinlocks
- Frequently in the Linux kernel; less commonly used at the user-level
- A spinlock basically spins on a mutex. If a thread cannot obtain the mutex, it will keep polling the lock until it is free.
- If a thread is about to give up a mutex, you don’t have to context switch to another thread. However, long spin times will result in poor performance.
- Should never be used on uniprocessor machines. Why?

Threads Programming

Scheduling
- Preemptive: a thread may be suspended at any point to allow another thread to proceed even when the preemted thread would otherwise continue running
- Non-preemptive: a thread runs until it makes a call to a threading system (eg, a system call), when the system may de-schedule it and schedule another thread to run

POSIX pthreads
- The pthread library can be found on almost any modern OS.
- 1. Add include `<pthread.h>` in your `.c` or `.h` header file(s)
- 2. Define the keyword _REENTRANT macro somewhere in a common `.h` or `.c` file
- 3. In your Makefile, check that gcc links against `-lpthread`
- Optional: add `-D_POSIX_PTHREAD_SEMANTICS` to your Makefile (gcc flag) for certain function calls like `sigwait()`


### POSIX Pthreads

A thread is represented by the type `pthread_t`.

```c
int pthread_create(pthread_t *thread, pthread_attr_t *attr, void *(*start_routine)(void *), void *arg);
```

**Example:**

```c
pthread_create(&pt_worker, &thread_attributes, thread_function, (void *)thread_args);
```

Create a `pthread pt_worker` with thread attributes defined in `thread_attributes`. The thread code is contained in the function `thread_function` and is passed in arguments `thread_args`. That is, the thread function prototype would look like `void *thread_function(void *args);`

### POSIX Pthreads

**Pthread Mutexes**

- `int pthread_mutex_lock(pthread_mutex_t *mutex);`
- `int pthread_mutex_trylock(pthread_mutex_t *mutex);`
- `int pthread_mutex_unlock(pthread_mutex_t *mutex);`

`pthread_mutex_lock()` is a blocking call. `pthread_mutex_trylock()` will return immediately if the mutex cannot be locked.

To unlock a mutex: `pthread_mutex_unlock()`.

**Pthread Condition Variables**

- `int pthread_cond_wait(pthread_cond_t *cond, pthread_mutex_t *mutex);`
- `int pthread_cond_signal(pthread_cond_t *cond);`
- `int pthread_cond_broadcast(pthread_cond_t *cond);`

`pthread_cond_wait()` puts the current thread to sleep. `pthread_cond_wait()` and `pthread_cond_broadcast()` signals one (all) threads waiting on a condition.

### Java threads

Methods of objects that inherit from class `Thread`

```java
Thread(ThreadGroup group, Runnable target, String name)
setPriority(int newPriority), getPriority()
start()
sleep(int millisecs)
yield()
destroy()
```

New thread created in the same JVM as its creator in the SUSPENDED state. `start()` makes it runnable. It executes the `run()` method of an object designated in its constructor.

A thread ends its life when it returns from `run()` or when its `destroy()` method is called.

Threads in groups (e.g., for security)

Execute on top of the OS.

### Java thread synchronization calls

Each thread's local variables in methods are private to it. However, threads are not given private copies of static (class) variables or object instance variables.

`synchronized` methods (and code blocks) implement the monitor abstraction. Guarantee that at most one thread is executing within it at any time.

The operations within a synchronized method are performed atomically with respect to other synchronized methods of the same object. `synchronized` should be used for any methods that update the state of an object in a threaded environment.

- `object.wait(long millisecs, int nanosecs)`
  - Blocks the calling thread for up to the specified time until thread has terminated.
  - Blocks the calling thread for up to the specified time until thread is interrupted, or the specified time has elapsed.
  - A thread that needs to block awaiting a certain condition calls `wait()`

- `object.wait(long millisecs, int nanosecs)`
  - Blocks the calling thread for up to the specified time until thread has terminated.
  - Blocks the calling thread for up to the specified time until thread is interrupted, or the specified time has elapsed.
  - A thread that needs to block awaiting a certain condition calls `wait()`

- `object.notify()`, `object.notifyAll()`
  - Wakes, respectively, one or all of any threads that have called `wait()` on `object`.

- `object.wait()`, `object.sleep()` and `object.notify()` are very similar to the semaphore operations. E.g. a worker thread would use `queue.wait()`, `queue.notify()` to wait for incoming requests.

- `object.wait()` and `object.notify()` are very similar to the semaphore operations. E.g. a worker thread would use `queue.wait()`, `queue.notify()` to wait for incoming requests.
Clients

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

The X Window System

- **X**: controls bit-mapped terminals (monitor, keyboard, mouse), part of the OS that controls the terminal, contains the terminal-specific device drivers
- **X kernel**: contains terminal-specific device drivers

Offers a low-level interface for controlling the screen and for capturing events from the keyboard and mouse.

Made available to applications through the Xlib library.

Compound documents

A collection of documents possibly of different kinds that are seamlessly integrated at the user-interface level – the user interface hides the fact that different applications operate at different parts of the document.

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).

Servers

Implement a service for a number of clients.

Basic model: A server is a process that waits for incoming service requests at a specific transport address.

Iterative server: the server itself handles the request.

Concurrent server: does not handle the request itself, but passes it to a separate thread or another process, and then immediately waits for the next request.

Client-Side Software

More than just interfaces, 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 are trying to mask server and communication failures) (e.g., retry, return cached values)

The X Window System

X distinguishes between normal applications and window managers.

- Normal applications request (through Xlib) the creation of a window on the screen. When a window is active, all events are passed to the application.
- Window managers manipulate the entire screen. Set restrictions (e.g., windows not overlap).

The X kernel and the X applications need not reside on the same machine.

X protocol: network-oriented communication protocol between an instance of Xlib and the X kernel.

X terminals (run only the X kernel).

Clients and Servers

Client

Servers

Object servers

Compound documents:

- A collection of documents possibly of different kinds that are seamlessly integrated at the user-interface level.
- The user interface hides the fact that different applications operate at different parts of the document.
- 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).
Endpoints
Clients send requests to an endpoint (port) at the machine where the server is running. Each server listens to a specific endpoint.

How does a client know the endpoint for a service?
1. Globally assigned endpoints (examples: TCP port 21 for Internet FTP, TCP port 80, for the HTTP server for the www) need to know just the machine.
2. Have a special daemon on each machine that runs servers, a client first contacts the daemon (a) how to find the daemon? What about passive servers?
3. SuperServer (b) servers that listen to several ports, i.e., provide several independent services. One server per endpoint, when a service request comes in, they start a subprocess to handle the request (UNIX inetd daemon).

Interrupting a Service
Is it possible to interrupt a server once it has accepted (or is in the process of accepting) a service request?
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
   • Requires OS supports high-priority scheduling of specific threads or processes
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

Stateless Servers
Stateless server: does not keep information of the state of its clients and can change its own state without informing its clients (e.g., a web server)

Examples:
• 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
• 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)

Stateful server: maintains information about its clients

Examples:
• 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
• 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
Object server: a server for supporting distributed objects
Provides only the means to invoke the local objects, not specific services

A place where object lives
Provides the means to invoke local objects

Object: data (state) + code (implementation of its methods)

Issues:
• Are these parts separated?
• Are method implementations shared among multiple objects?
• A separate thread per object or a separate thread per invocation?

Invoking Objects
Activation policy: decisions on how to invoke an object
Object adapter of object wrapper: a mechanism to group objects per policy

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

Object adapter: the “manager” of a set of objects
• Inspects (at first) incoming requests
• Ensures referenced object is activated (requires identification of servant)
• Passes request to appropriate skeleton, following a specific activation policy
• Responsible for generating object references
### Code Migration

#### Reasons for Migrating Code
- **Performance**
  - Load balancing
  - Process data close to where they reside
- **Flexibility/dynamic configuration**
  - Dynamically downloading client-side software

#### Models for Code Migration

**What is moved?**

- **Weak mobility**: move only the code segment (plus perhaps some initialization data)
  - Always start from its initial state
  - Example: Java applets
  - Code shipping (push) vs. code fetching (pull)
- **Strong mobility**: move also the execution segment
  - The process resumes execution from where it was left off
  - Harder to implement

**Where/How is the code executed?**

- **Weak mobility**
  - The migrated code:
    - Executed by the target process, or
    - A separate process is initiated
  - Example: Java applets executed in the Web browsers address space
- **Strong mobility** can be supported by remote cloning
  - Cloning yields an exact copy of the original process, executed in parallel

**Who initiates the movement?**

- **Sender-initiated**: migration is initiated at the machine where the code currently resides or is being executed
  - Example: uploading programs, sending programs across the Internet
  - Simpler to implement
- **Receiver-initiated**: the initiative for migration is taken by the target machine
  - Example: Java Applets

#### Models for Code Migration (summary)
Migrating Resources

Three types of process-to-resource bindings:

- **binding by identifier**: requires precisely the references resource. Examples: URL address, Internet address of an FTP server, local communication endpoints
- **binding by value**: only the value of a resource is needed. Example: standard C or Java library
- **binding by type**: needs a resource of a specific type. Examples: printer, monitors

Three types of resource-to-machine bindings:

- **unattached resources**: can be easily moved between machines. Examples: local data files
- **fastened resources**: is possible to be moved but with high costs. Examples: local databases, complete web pages
- **fixed resources**: infeasible to be moved. Examples: printer, monitors, local communication endpoints

Migration and Local Resources

<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, if exists)</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)</td>
<td>GR</td>
</tr>
</tbody>
</table>

Migration in Heterogeneous Systems

- **Main problem**: (a) The target machine may not be suitable to execute the migrated code. (b) 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
- **Existing languages**: Code migration restricted to specific points in the execution of a program: only when a subroutine is called (migration stack)
- **Interpretated languages**: running on a virtual machine (Java, scripting languages)

Software Agents

- An autonomous process capable of reacting to, and initiating changes on, its environment, possibly in collaboration with users and other 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 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>Adaptable</td>
<td>No</td>
<td>Capable of learning</td>
</tr>
</tbody>
</table>

Agent Technology

- **FIPA (Foundation for Intelligent Physical Agents)**
- **Agent Platform**: provide the basic services needed by any multiagent system (create, delete, locate agents, interagent communication)
- **Naming service**: map a globally unique id to a local communication endpoint (for each agent)
- **Local directory service**: (similar to yellow pages) based on (attribute, value) pairs. Accessible by remote agents
Agent Communication Languages

Agents communicate by exchanging messages

**ACC (Agent Communication Channel):** provide reliable, order, point-to-point communication with other platforms

**ACL (Agent Communication Language):** application level communication protocol

**Distinction between Purpose - Content**

The purpose determines the receiver's reaction

---

### Table: Message Content and Purpose

<table>
<thead>
<tr>
<th>Message Content</th>
<th>Message Purpose</th>
</tr>
</thead>
<tbody>
<tr>
<td>INFORM</td>
<td>Inform that a given proposition is true</td>
</tr>
<tr>
<td>QUERY-IF</td>
<td>Query whether a given proposition is true</td>
</tr>
<tr>
<td>QUERY-REF</td>
<td>Query for a given object</td>
</tr>
<tr>
<td>CFP</td>
<td>Ask for a proposal</td>
</tr>
<tr>
<td>PROPOS</td>
<td>Provide a proposal</td>
</tr>
<tr>
<td>PROPOSE-PROPOSAL</td>
<td>Tell that a given proposal is accepted</td>
</tr>
<tr>
<td>REJECT-PROPOSAL</td>
<td>Tell that a given proposal is rejected</td>
</tr>
<tr>
<td>REQUEST</td>
<td>Request that an action be performed</td>
</tr>
<tr>
<td>SUBSCRIBE</td>
<td>Subscribe to an information source</td>
</tr>
</tbody>
</table>

---

**Header | Actual Content**

Actual Content specific to the communicating agents (no prescribed format)

Header: purpose id, server, receiver, language or encoding scheme, ontology (maps symbols to meaning)

<table>
<thead>
<tr>
<th>Field</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Purpose</td>
<td>INFORM</td>
</tr>
<tr>
<td>Receiver</td>
<td><a href="mailto:female@beatrix.royalty-spotters.nl">female@beatrix.royalty-spotters.nl</a>:7239</td>
</tr>
<tr>
<td>Sender</td>
<td><a href="mailto:elke@ecp.royalty-watcher.uk">elke@ecp.royalty-watcher.uk</a>:8225</td>
</tr>
<tr>
<td>Language</td>
<td>Prolog</td>
</tr>
<tr>
<td>Ontology</td>
<td>genealogy</td>
</tr>
<tr>
<td>Content</td>
<td>female, (beatrice.parent, beatrice, juliana, bernhard)</td>
</tr>
</tbody>
</table>