Processes

Topics to be covered

Threads
Clients and Servers
Code Migration
Mobile Agents

Threads

Introduction: Processes

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

Process table
Process: program in execution

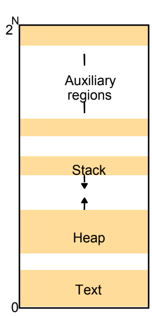
Concurrency transparency

Kernel: supervisor mode
Other processes in user mode

System calls

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

Process address space



2^{32} - 2^{64} bytes
Divided in regions

Regions can be shared
- kernel code
- libraries
- shared data & communication
- copy-on-write

A stack per thread

*

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)

Introduction: Processes vs Threads

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

Lightweight processes:
• Creation 10 to 100 times faster than process creation
• Shared memory: all threads within a given process share the same memory and files.

Introduction: Processes vs Threads

• 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

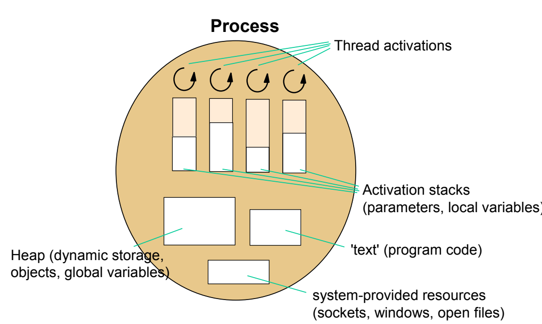
Threads versus multiple processes

State associated with execution environments and threads

Execution environment	*Thread*
Address space tables	Saved processor registers
Communication interfaces (eg sockets), open files	Priority and execution state (such as *BLOCKED)*
Semaphores, other synchronization objects	Software interrupt handling information
List of thread identifiers	Execution environment identifier
Pages of address space resident in memory; hardware cache entries	

Threads concept and implementation

A process has many threads sharing an execution environments



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., mutuxes and condition variables)

Main issue: Should an OS kernel provide threads, or should they be implemented as 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)

*

Thread Implementation

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.

Thread Implementation

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 Implementation

User-level vs Kernel-level Threads (summary)

(-) threads within a process can take advantage of a multiprocessor

(-) A tread 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 (eg switching) cost less; do not involve a system call

(+) Thread scheduling can be customized

(+) Many more 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 execute *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)

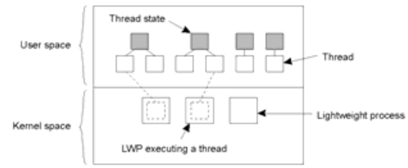
When a thread calls a blocking user-level operation (e.g., blocks on a mutex or a condition variable), it calls the scheduling routine, when another runnable thread is found, a context switch is made to that thread which is then bound to the *same LWP* (the LWP that is executing the thread need not be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

Lightweight Process (LWP) can be thought of as a virtual CPU where the number of LWPs is usually greater than the number of CPUs in the system. Thread libraries communicate with LWPs to schedule threads. LWPs are also referred to as *kernel threads*.



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.

Multithreaded Servers

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

Multithreaded Servers

Example (continued)

Assume disk block caching, 75% hit rate

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

Multithreaded Servers
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 msecs = 2msecs
Processing time per request: 2.5msec

But two process executed in parallel -> ?? (444 req/sec prove it!)

Throughput:

More than two threads

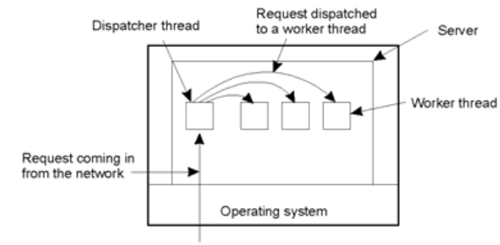
Bounded by the I/O time (2msecs per process) thus,

Max throughput: 500 req/sec

Multithreaded Servers

A pool of "worker" threads to process the requests

One I/O (dispatcher) thread: receives requests from a collections of ports and places them on a shared request queue for retrieval by the workers



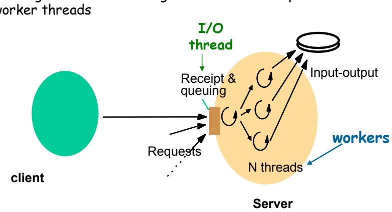
Multithreaded Servers
The worker pool architecture

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 collections of ports and places them on a shared request queue for retrieval by the workers

Priorities: multiple queues in the worker pool

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



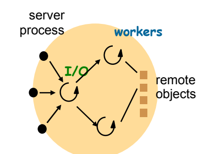
Multithreaded Servers
One thread-per-request architecture

The I/O thread spawns a new worker thread for each request

The worker destroys itself when it has processed the request

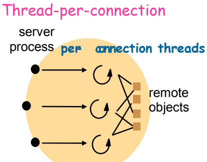
Threads do not contend for a shared queue and as many workers as outstanding requests

Disadvantage: overhead of creating and destroying threads



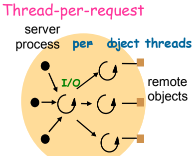
Multithreaded Servers

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 request over the connection.

Thread-per-request

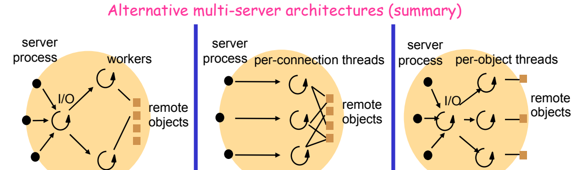


One thread for each remote object. There is an I/O thread that receives requests and queues them for the workers, but there is one queue per object.

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)



(a) Thread-per-request (b) Thread-per-connection (c) Thread-per-object

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

Model	Characteristics
Threads	Parallelism, blocking system calls
Single-threaded process	No parallelism, blocking system calls
Finite-state machine	Parallelism, nonblocking system calls

Three ways to construct a server.

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

▪ 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 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

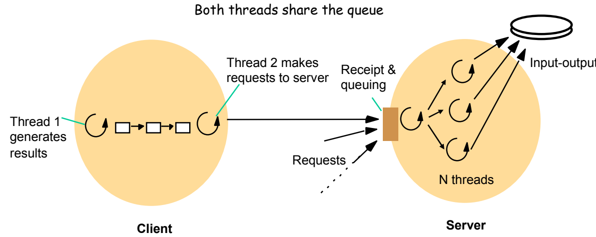
Client and server with threads

Client

1st thread generates results to be passed to the server through a (blocking) RPC call

2nd thread performs the RPC call

Both threads share the queue



The 'worker pool' architecture

Threads Programming

Concurrent Programming

Concepts:

Race conditions
Critical section
Monitor
Condition variable
Semaphore

In conventional languages such as C augmented with a thread library

pthreads (POSIX)

Java Thread class

Threads Programming

Mutual Exclusion

Avoid a thread modifying a variable that is already in the process of being modified by another thread or a dirty read (read an old value)

Attach **locks** to resources. Serialization of accesses

The code between the lock and unlock calls to the mutex, is referred to as the **critical section**. Minimizing time spent in the critical section allows for greater concurrency because it reduces the time other threads must wait to gain the lock.

Deadlocks

Priority Inversion

Multiple reader lock, Writers starvation

Threads Programming

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)

Threads Programming

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

Semaphores

Binary semaphores act much like mutexes, while counting semaphores can behave as *recursive mutexes*.

Counting semaphores can be initialized to any arbitrary value (lock depth): depending on the number of resources available for that particular shared data.

Many threads can obtain the lock simultaneously until the limit is reached.

Threads Programming

Scheduling

Preemptive: a thread may be suspended at any point to allow another thread to proceed even when the preempted 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 #define _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.

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

int pthread_attr_init(pthread_attr_t *attr);

Example:

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 stored in** 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() pthread_cond_broadcast() signals one (all) threads waiting on a condition

Java threads

Methods of objects that inherit from class Thread

Thread(ThreadGroup group, Runnable target, String name)
- Creates a new thread in the *SUSPENDED* state, which will belong to *group* and be identified as *name*; the thread will execute the *run()* method of *target*.

setPriority(int newPriority), getPriority()
- Set and return the thread's priority.

run()
- A thread executes the *run()* method of its target object, if it has one, and otherwise its own *run()* method (*Thread* implements *Runnable*).

start()
- Change the state of the thread from *SUSPENDED* to *RUNNABLE*.

sleep(int millisecs)
- Cause the thread to enter the *SUSPENDED* state for the specified time.

yield()
- Enter the *READY* state and invoke the scheduler.

destroy()
- Destroy the thread.

Java thread lifetimes

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 can execute 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.

Java thread synchronization calls

Allows threads to be blocked and woken up via arbitrary objects that act as condition variables

A thread that needs to block awaiting a certain condition calls wait()

thread.join(int millisecs)
- Blocks the calling thread for up to the specified time until *thread* has terminated.

thread.interrupt()
- Interrupts *thread*: causes it to return from a blocking method call such as *sleep()*.

object.wait(long millisecs, int nanosecs)
- Blocks the calling thread until a call made to *notify()* or *notifyAll()* on *object* wakes the thread, or the thread is interrupted, or the specified time has elapsed.

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

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

Clients and Servers

Client
Servers
Object servers

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

The X Window System

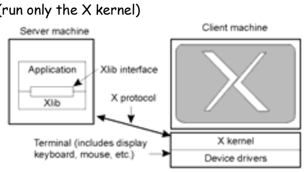
X distinguishes between normal applications and window managers

- 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)



Compound documents

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

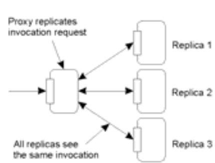
drag-and-drop: move objects to other positions on the screen, possibly invoking interaction with other applications

in-place editing: integrate several applications at user-interface level (word processing + drawing facilities)

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)



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

Endpoints

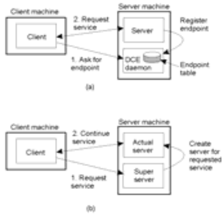
Clients sends requests to an endpoint (port) at the machine where the server is running. Each server listens to a specific endpoint.

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?

Say, you are downloading a large file

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: maintain 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.

- Cookies?

Stateless Servers

- Cookies?

The client stores in its browser information about its previous accesses
The client sends this information to the server

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

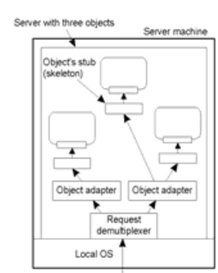
Activation policies: decisions on how to invoke an object

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:

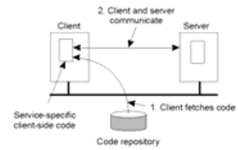
- Inspects (at first) incoming requests
- Ensures referenced object is activated (requires identification of servant)
- Passes request to appropriate skeleton, following a specific **activation policy**
- Responsible for generating **object references**



Code Migration

Reasons for Migrating Code

- Performance
  - Load balancing
  - Process data close to where they reside
  - Parallelism (e.g., web search)
- Flexibility/dynamic configuration
  - Dynamically downloading client-side software



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

Models for Code Migration

What is moved?

The three segments of a process:

- **Code segment**: the part that contains the set of instructions that make up the program that is being executed
- **Resource segment**: references to external resources needed by the process (e.g., files, devices, other processes)
- **Execution segment**: the current execution state of the process (program counter, stack, etc)

Weak mobility: move only the code segment (plus perhaps some initialization data)

- Always start from its initial state
- Example: Java applets
- code shipping (push) code fetching (pull)

Strong mobility: move also the execution segment

- The process resumes execution from where it was left off
- Harder to implement

Models for Code Migration

Where/How is the code executed?

Weak mobility

The migrated code:

- executed by the target process, or
- a separate process is initiated

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

Models for Code Migration

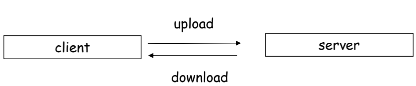
Who initiates the movement?

Sender-initiated: migration is initiated at the machine where the code currently resides or is being executed

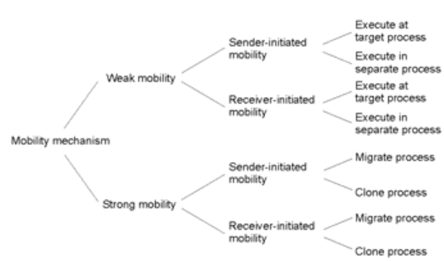
- Example: uploading programs, sending programs across the Internet
- simpler to implement

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 libraries
- **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, locla communication endpoints

Migration and Local Resources

Resource-to machine binding

Process-to-resource binding	Unattached	Fastened	Fixed
By identifier	MV (or GR, if shared)	GR (or MV)	GR
By value	CP (or MV, GR)	GR (or CP)	GR
By type	RB (or GR, CP)	RB (or GR, CP)	RB (or GR)

MV move the resource

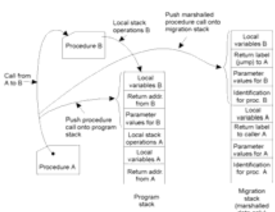
GR establish a global systemwide reference

CP copy the value of the resource

RB rebind process to locally available resource

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)
- **Interpreted languages**: running on a virtual machine (Java, scripting languages)



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

Software Agents

Software Agents

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

Properties

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

Functionality

Interface agents: agents that assist an end user in the use one of more applications

Information agents: manage (filter, order, etc) information for many resources

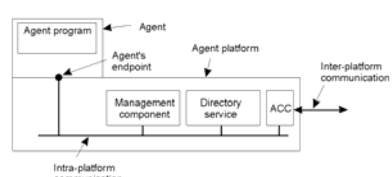
Agent Technology

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 an (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

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

Agent Communication Languages

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)

Field	Value
Purpose	INFORM
Sender	max@http://fanclub-beatrix.royalty-spotters.nl:7239
Receiver	elke@iiop://royalty-watcher.uk:5623
Language	Prolog
Ontology	genealogy
Content	female(beatrix),parent(beatrix,juliana,bernhard)