Processes

Topics to be covered
- Threads
- Clients and Servers
- Code Migration
- Mobile Agents

Introduction: Processes vs Threads

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

Execution environment:
- an address space
- higher level resources
- CPU context
- Process table
- Process: program in execution

Concurrency transparency

Process context: CPU context (register values, program counter, stack pointer), registers of the memory management unit (MMU)
- Expensive context switch (may also require memory swap)

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

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.

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

<table>
<thead>
<tr>
<th>Execution environment</th>
<th>Thread</th>
</tr>
</thead>
<tbody>
<tr>
<td>Address space tables</td>
<td>Kernel registers</td>
</tr>
</tbody>
</table>
| Communication interfaces, open files | Priority and execution state (such as
| Semaphore, other synchronization objects | UNIX SECU |
| List of thread identifiers | Software interrupt handling information |
| Pages of address space resident in memory; hardware cache entries | Execution environment identifier |

Threads concept and implementation

Process

- Activation stacks (parameters, local variables)
- Heap (dynamic storage, objects, global variables)
- System-provided resources (sockets, windows, open files)

Threads 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 as user-level packages?
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.

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 (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)
Each LWP executes a scheduling routine in search of a thread to execute (if many LWPs, they share the thread table, synchronization among them in user space).

When a thread calls a blocking user-level operation (e.g., blocks on a mutex or a condition variable), and 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.

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. Why? 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).

Example
Each request takes on average 2 ms of processing and 8 ms 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 ms
  - Throughput: 100 req/sec
- Two threads
  - (if disk requests are serialized)
  - Turnaround time for each request: 8 ms
  - Throughput: 125 req/sec

Example (continued)
Assume disk block caching, 75% hit rate.
- Two threads
  - Mean I/O time per request: 0.75 * 0 + 0.25 * 8 = 2 ms
  - 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 = 2 ms
  - Processing time per request: 2.5 ms
  - But two process executed in parallel -> ?? (444 req/sec prove it!)
  - Throughput:

More than two threads
- Bounded by the I/O time (2 ms per process) thus,
  - Max throughput: 500 req/sec

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.
- N-arrived multiple queues in the worker pool
- Disadvantage: high level of switching between the I/O pool and the workers; limited number of worker threads.

A pool of "worker" threads to process the requests
One dispatcher thread receives requests from a collection of ports and places them on a shared request queue for retrieval by the workers.

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.
- N-arrived multiple queues in the worker pool
- Disadvantage: high level of switching between the I/O pool and the workers; limited number of worker threads.

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.
- N-arrived multiple queues in the worker pool
- Disadvantage: high level of switching between the I/O pool and the workers; limited number of worker threads.

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.
- N-arrived 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**

- Server process creates a new thread per connection.
- The server creates a new thread when a client makes a connection and destroys the thread when the client closes the connection.
- Clients may delay while a worker has several requests but another thread has no work to perform.

**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.

### Multithreaded Servers

**Alternative multi-server architectures (summary)**

<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</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.

### Multithreaded Clients

**Main issue is hiding network latency**

- Main issue is hiding network latency

**Multithreaded Web client**

- 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.

**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

Thread 1 generates results

Thread 2 makes requests to server

Server

N threads

Requests

Receipt & queuing

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 accessed 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

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?
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.

POSIX pthreads

The pthread library can be found on almost any modern OS.

```c
#include <pthread.h>

#define _REENTRANT
```

In your Makefile: gcc links against -lpthread

Optional: add `-D_POSIX_PTHREAD_SEMANTICS` to your Makefile

POSIX pthreads

Pthread Mutexes

```c
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

```c
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.

Java threads

Methods of objects that inherit from class Thread

```java
public class Thread extends Object implements Runnable {

    Thread(ThreadGroup group, Runnable target, String name){
        * Creates a new thread in the SUSPENDED state, which will belong to group and be named as name, the thread will execute the run() method of target.
        * Set the thread's priority.
        * Set the thread's name.
        * A thread executes the run() method of its target object, if it has one, and otherwise its own run() method (Thread implements Runnable).
        * Change the state of the thread from SUSPENDED to RUNNABLE.
        * Cause the thread to enter the SUSPENDED state for the specified time.
        * Enter the READY state and invoke the scheduler.
        * Destroy the thread.

    }
```

Java thread lifetimes

New thread created in the same JVM as its creator in the SUSPENDED state

```java
start() makes it runnable
```

It executed 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.

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

thread.interrupt()
- Interrupts the calling 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

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

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 do not need to 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).
Client-Side Software

More than just interfaces, often focused on providing distribution transparency

- Access transparency: client-side stubs for RPCs and RMI
- 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 send requests to an endpoint (port) at the machine where the server is running. Each server listens to a specific endpoint.

1. Globally assigned endpoints (examples: TCP port 21 for Internet FTP, TCP port 80, for the HTTP server for the www)
2. Have a special daemon on each machine that runs servers, a client first contacts the daemon(a)
3. Superserver (b)
   - 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 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: 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?

Object Servers

Object server: a server for supporting distributed objects
Provides only the means to invoke the local objects, no 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 ports 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 or 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 (as 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

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

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)

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 shared)</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>

MV: move the resource
GR: establish a global system-wide 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)

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>Adaptive</td>
<td>No</td>
<td>Capable of learning</td>
</tr>
</tbody>
</table>

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 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>
<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 given object</td>
<td>Expression</td>
</tr>
<tr>
<td>CFP</td>
<td>Ask for a proposal</td>
<td>Proposal specific</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>

Header | Actual Content

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

Header: purpose, 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>Sender</td>
<td>elke@iiop://royalty-watcher.uk:5623</td>
</tr>
<tr>
<td>Receiver</td>
<td>max@<a href="http://fanclub-beatrix.royalty-spotters.nl:7239">http://fanclub-beatrix.royalty-spotters.nl:7239</a></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>