

SPIN: An extensible operating system

Bershad et al.
For CMSC 838Y
Spring 2003

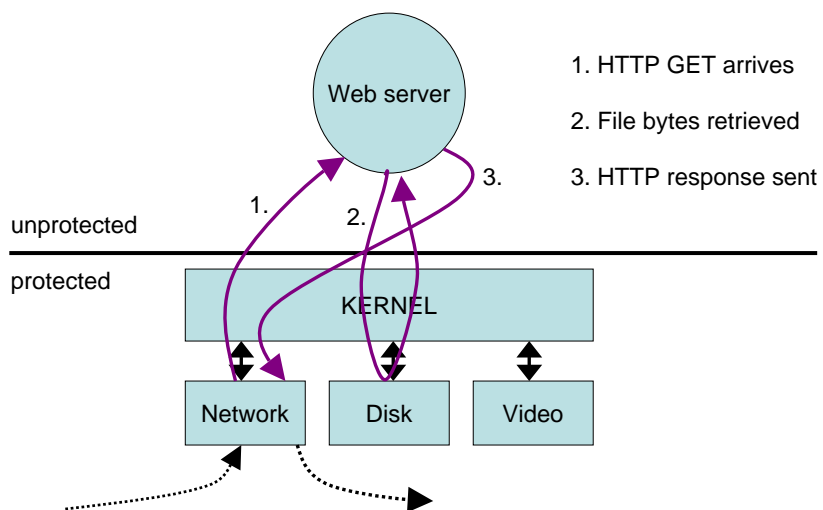
Papers

- **Extensibility, Safety, and Performance in the SPIN operating system, SOSP 1995.**
 - Overview of the entire system
- **Dynamic Binding in an Extensible System, OSDI 1996.**
 - Focus on the extensibility model, and enabling language and system mechanisms.

Motivation

- Typical operating system extensibility mechanism is the **process**.
- Hardware protection mechanisms (virtual memory and operating modes) are expensive, so
 - interfaces not well-matched to applications
 - communication tends to be coarse-grained

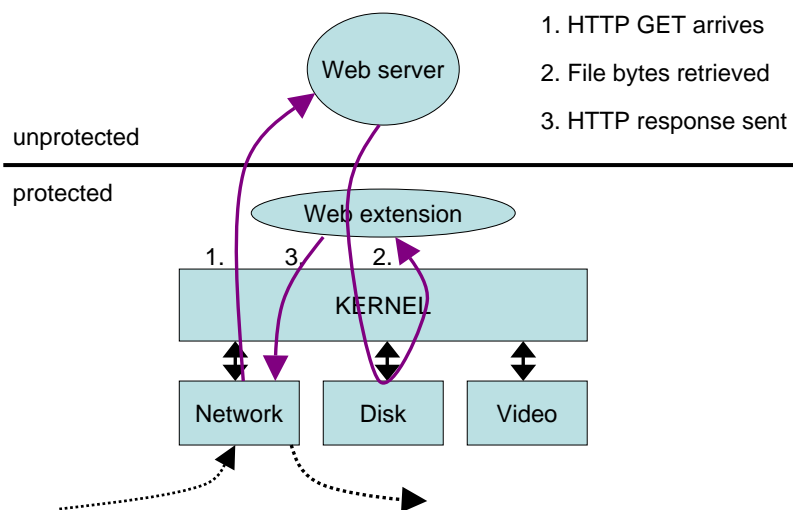
Example: Webserver



Example: Webserver

- Three syscalls required:
 - Wait for and obtain connection request.
 - Retrieve file from filesystem.
 - Send out response.
- Syscalls are expensive
 - much more than a procedure call
- Interface is coarse-grained
 - Just general communication, not tailored to a webserver

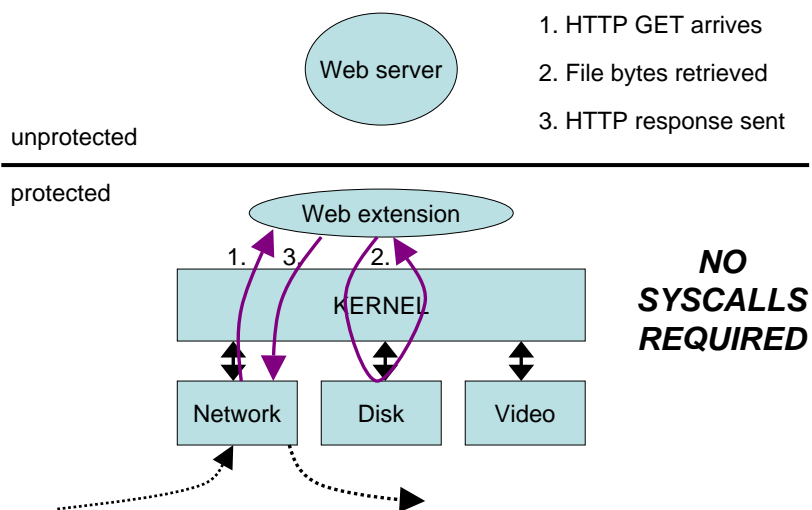
Example: SPIN Extension 1



What's the extension doing?

- It takes the name of a file and a connection file descriptor and sends the file contents along that connection
- Eliminates two syscalls
 - Only HTTP parsing done in server app
 - Could also handle CGI and dynamic content
- Extension could be useful to an FTP server, an NFS server, etc.

Example: SPIN Extension 2



SPIN elements

- Co-location
 - Extensions loaded into the kernel
- Enforced modularity
 - Type-safety guarantees no pointer forging
- Logical protection domains
 - Separate linking namespaces
- Dynamic call binding
 - Flexible, event-based system

What does this get you?

- Build specialized abstractions
 - Bake your own thread system, virtual memory management system, syscall interface, etc.
- Cut performance bottlenecks
 - Code accesses system services at low latency
- Be safe
 - Type safety prevents violating abstraction boundaries; traditionally been enforced by hardware
 - Caveat: what about extensions that abuse resources like memory, disk, CPU, etc.? More later ...

Language-based protection

- Modula-3 key features
 - Type safety
 - No unsafe casts, unchecked array accesses
 - Garbage collection (no manual freeing of memory)
 - Modularity mechanisms: interfaces and modules
 - Interfaces used to define *events* separate from a module's implementation
 - Allows for later (dynamic) extensions

Capabilities = Abstract Types

```
INTERFACE Console;  
TYPE T <: REFANY;  
  
PROCEDURE Open():T;  
PROCEDURE Write(t: T; msg: TEXT);  
PROCEDURE Read(t: T; VAR msg: TEXT);  
PROCEDURE Close(t: T);  
END Console;
```

Defines an abstract type

Provides a "capability"

Invalidates the "capability"

Any type-safe language would do...

```
module type Console =  
  sig  
    type t  
    open : unit → t  
    write : t → string → unit;  
    read : t → string ref → unit;  
    close : t → unit  
  end;
```

An OCaml module signature

Protection Domains

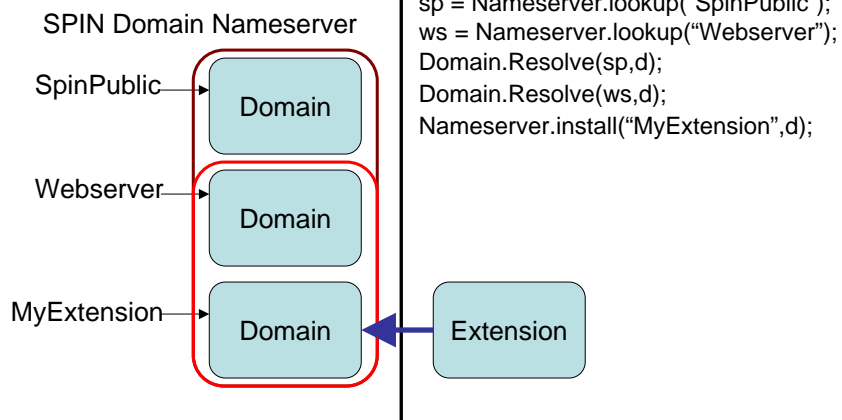
- Essentially a namespace
 - At the language level, using programming language names
 - Traditionally at the hardware level, using virtual addresses
- First-class object in SPIN
 - Has type Domain.T (abstract)
 - Facilitates definition of multiple namespaces

A Domain is a Module

- Has imports
 - References to externally defined symbols
 - Resolved against other domains using `Domain.Resolve`
- Has exports
 - References to externally visible symbols
 - Can be resolved against by `Domain.Resolve`
- Can be combined
 - *merge* in Cardelli terminology

Dynamic Linking in SPIN

Kernel



Two-level Namespace

- Nameserver defines global names
 - SpinPublic, Webserver, etc.
- Each domain defines its own separate namespace
 - Each domain's names are separate from all others, unless explicitly linked with Resolve
 - Avoids collisions between application names

Dynamic Binding of Extensions

- Procedures in interfaces define events
 - Each event has one or more *handlers*; the default implementation of that procedure is the *intrinsic handler*
 - To *raise* an event, you just invoke the procedure

Example

```
INTERFACE Console;
```

```
...
```

```
PROCEDURE write(t: T, msg: TEXT);
```

```
END Console.
```

Interface

```
MODULE Console;
```

```
...
```

```
PROCEDURE write(t: T, msg: TEXT) = ...
```

```
END Console.
```

*Default
Implementation*

```
...
```

```
Console.write(console, "hello there");
```

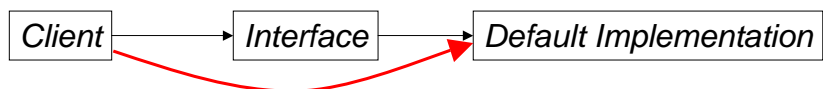
```
...
```

Client

Example

*Raises the event
Console.write(...)*

*"Intrinsic"
handler is invoked*



*Implemented
as a
procedure call*

Adding a handler

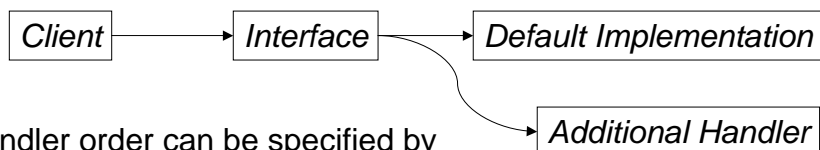
```
MODULE ConsoleLogger;  
...  
PROCEDURE writeLogger(t: Console.T, msg: TEXT) = ...  
...  
BEGIN (* initialization*)  
  Dispatcher.InstallHandler(  
    Console.write, ...,  
    writeLogger);  
END ConsoleLogger.
```

New handlers are registered with the SPIN dispatcher

Adding a handler

*Raises the event
Console.write(...)*

*Both handlers are
invoked*

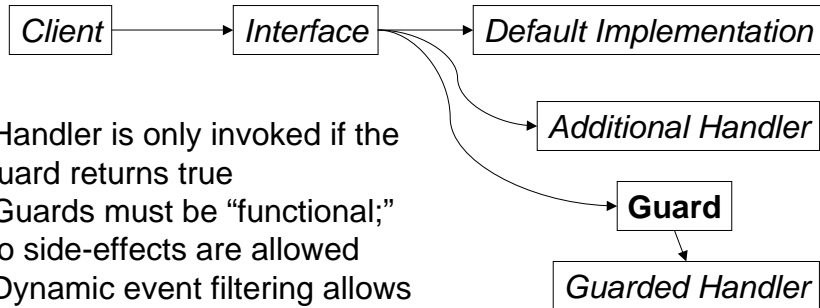


- Handler order can be specified by with ordering constraints.
- Handlers can modify arguments seen by subsequent handlers.
- Multiple results can be combined as specified by the user.

Adding a guarded handler

Raises the event
Console.write(...)

Guards invoked first,
then handlers



- Handler is only invoked if the guard returns true
- Guards must be “functional;” no side-effects are allowed
- Dynamic event filtering allows handlers to be more generic

Access Control

- Which applications are allowed to
 - raise particular events?
 - add extensions to handle events?
- The defining module for an event defines authorization procedures for this
 - Which domains can link against this module
 - Which handlers can extend an event
 - May impose a guard to allow only partial access

Denial of Service

- How to deal with resource abuse?
 - Hogging the CPU (infinite loop, anyone?)
 - Allocating a ton of memory
 - You get the idea ...
- Traditional OS processes do this well
 - Can apply per-process limits
 - Scheduling policies (priorities, etc.)
 - Heap/stack limits
 - Virtual address spaces make for easy killing

Two techniques

- Handler asynchrony
 - Can specify that handlers run in their own thread.
Caller returns immediately.
- EPHEMERAL (killable) handlers
 - Specifies handlers that can be abruptly terminated.
 - Can only call EPHEMERAL procedures (e.g. new() is not EPHEMERAL).
 - Protects system from app, but not app from itself.
 - Could violate own datastructure invariants
 - FUNCTIONAL → EPHEMERAL

Summary of Extension Model

- “It must be clear, though, that a certain degree of vigilance and paranoia is required when designing the components of an extensible system.”
- This sounds like an opportunity for future work!

Implementing the Dispatcher

- One handler = normal procedure call
 - Catch: in Modula-3, a normal procedure call is indirected through the interface (I believe)
- Multiple handlers = specialized dispatch routine
 - At installation time, a fresh routine is generated and installed that unrolls the loop and invokes each guard+handler directly.

Performance

- Overhead is low. What did you expect?

Building an OS from Extensions

- What are the core abstractions that we want to extend?
- Low level
 - Memory management
 - Thread management
- Higher level
 - Network stacks
 - Filesystems

Memory Management

- Physical Addresses
 - Abstract type referring to physical storage
- Virtual Addresses
 - Abstract type referring to a virtual address range
- Translations
 - Map virtual addresses to physical storage

Example: UNIX address space

- Allocate a bunch of physical pages
- Allocate the corresponding virtual addresses
- Allocate a fresh translation between the two: this is the address space.
 - Need to allocate a fresh translation for each address space.
 - Multiple translations could map to the same physical pages to enable sharing.

Example: Handling Page Faults

- Install application handler to respond to page faults
 - Used by memory allocators to discover the “end of a page” so as to grow the heap
 - Response time to extension much faster
- Can have an order of magnitude speedup

Thread Management

- Strands
 - Name for some processing context
- Events
 - Block, Unblock
 - Raised by services to block execution (e.g. to wait on I/O)
 - Checkpoint, Resume
 - Raised by the scheduler (in response to block/unblock) to capture and restore thread state

Networking Extensions

- Kernel protocol handlers map directly to packet protocol stack
- Add handler to any element of the stack to customize it
 - Extend TCP.PktArrived event to
 - find HTTP packets, and redirect requests directly to the filesystem
 - implement a proxy by forwarding packets and connection requests
 - Extend UDP.PktArrived event to
 - find UDP video packets and redirect them directly to the video frame buffer.

Conclusions

- SPIN provides
 - Extensibility: add application-specific extensions directly to the kernel
 - Performance: extensions reduce application latency and are implemented with efficient services
 - Safety: language and run-time enforcement reduce effects of misbehaving extensions

Open Issues

- Denial of Service
 - EPHEMERAL provides a basic framework but is not robust in the face of untrusted extensions
- Trust
 - SPIN uses a trusted compiler; can we use proof-carrying code to reduce the TCB?
- Extension model
 - Extensions that are largely ignorant of one another are difficult to support. How can we coordinate extensions in an abstract, yet flexible manner?
- Authorization
 - The authorization scheme is fairly ad hoc in that information flow is fairly restricted (just the authorizer and a token). How can this be improved? How can it integrate with the extension model?

This document was created with Win2PDF available at <http://www.daneprairie.com>.
The unregistered version of Win2PDF is for evaluation or non-commercial use only.