Advanced Topics in Computer Systems, CS262B

Prof Eric A. Brewer

Spin
Leading OS Research Astray?

January 27, 2004

I. Extensibility, Safety and Performance in the SPIN Operating System

Goal: extensible OS that has both safety and good cross-domain performance

Why build an extensible OS?

- Performance: a general-purpose OS handles all apps equally poorly. E.g, multimedia apps, passing buffers between network and disk, new network protocols,...
- 3rd-party development: untrusted third-parties can write extensions without need kernel source code, and without compromising the safety/security of the kernel. In turn, this should speed up innovation.
- risk management: faster and safer to develop extensions, rather than kernel mods.
- change is the norm! better to make a changeable system rather than a monolithic one

Note that these goals apply to plug-ins, applets, and Active X as well. In all cases, we enable innovation and third-party development that ideally should be safe relative to the base system. Databases have similar extension mechanisms (often called “blades”).

How to extend:

- insert trusted code into the kernel dynamically -- key claim: this is necessary for performance reasons (wins by avoiding kernel crossings)
- Use a type-safe language (Modula 3) to enforce namespaces securely. I.e., extension run in the kernel address space, but can’t muck with anything that they can’t explicitly name. They call this enforced modularity.
- Type-safe => no dangling references => must have automatic storage management (garbage collection)
- Also need dynamic linking, since we don’t want to reboot for every new extension
- Spin has basic modules for VM and scheduling, and uses events to interact with extensions

Alternative frameworks have different answers: applets are safe and use dynamic linking, Active X is dynamic but not safe (but is authenticated); plug-ins are neither safe nor dynamic, same with database blades.

Signed modules:

- Modula-3 does not have a byte-code verifier => modules must be signed by the compiler
- How does the compiler have a secret key? this is unclear, but it can't just be in the executable, since an attacker could steal it.
- solution 1: compile everything locally, so that the key is only valid for the local machine (and hence damage due to a stolen key is limited to the local machine as well). Stealing the key should be just as hard as getting root access, and thus introduces little additional weakness. Java kind of does this by verifying everything locally; the compiler output is not actually trusted.
- solution 2: have a compiler service that is shared. Stealing its key would allow a bad module to exist in many places, but the centralization makes it possible to secure the key well. (this is like verisign for certificates)
- The bad case occurs when you load modules signed by compilers from other machines; can you trust that machine has not been compromised?

Boundaries: although not in this paper, Spin makes a good venue to talk about boundaries. By boundary, I mean two sides of an API that have different trust levels. Extension boundaries and kernel crossing are examples of boundaries, as are plug-ins and applets. In each case, there is the base side that is trusted, and the extension side that is not (at least not as much). In this section, I outline all of the issues that you have to think about whenever you have a boundary; there is a huge number of them and they are rarely explicit or even discussed. Spin covers most of them well, but never views the issues all together as the set of "boundary problems".

- Shared protection domain: are the two sides in the same protection domain? If not, there is less trust needed. Spin uses a shared address space but depends on Modula-3 to limit access. Nemesis is an unusual case in that it has one address space, but separate domains. Plug-ins are same address space and are trusted but not always trustworthy.
- Shared storage: even when using separate domains, there may be shared memory regions or shared storage. These regions can clearly be damaged by an extension. The usual answer to this is to have the base system check the shared regions before use. A simple example is argument checking for kernel calls. Similarly, shared files can be checked for validity before being used by the base system.
- Are there pass by reference arguments across the boundary? If so, there is clearly shared storage. It is literally shared (extension updates are immediately visible in the base), or it is really pass-by-copy-result (pass by value, then copy the result back). LRPC is confused about this issue. There are two directions to worry about: 1) the extension passes in a reference -- can it then change the value while the base is using it? 2) the base passes a reference to the extension -- does the extension have read/write or read-only access to the base value? There is another more subtle issue in the Spin case: if we pass a pointer out to an extension, it might pass it back in with a different type, thus defeating the type safety. Spin avoids this by "externalizing" the reference, which basically means make a table of pointers and pass out the index rather than the pointer. The extension can pass back in the reference (integer), but the table securely stores the real type.
- What is the lifetime of a reference passed across the boundary? This is a tricky one

that is almost always left out. The default answer is that a reference is good forever and the base cannot move, or really even use, the shared object. (The base could lock the object, but these is really no reason to think the extension will follow the locking discipline.) The two simplest solutions are to either avoid pass by reference (only pass values), or to have references expire (leases). With expiration, there needs to be a way to detect the use of expired references, which implies a level of indirection -- you can't use pointers directly, nor can you pass in the timestamp, unless it was signed by the base. In theory, one could garbage collect references across a boundary, but requires not only a shared GC system, but that the extension can't pass references outside the GC boundary (such as storing them in a file or giving them to a non-GC system). Faults brake the GC model as well: if an extension becomes disconnected when do you GC references it had? To summarize, unless you're within one protection domain, GC probably doesn't make sense, leases work OK, and pass by value is the simplest. The problem with leases is that it is hard to build systems that work robustly when leases expire during correct use.

- Partial failure: can the other side fail without this side failing? If so all crossings have non-transparent exceptional conditions, and all crossing probably need a timeout. If an extension fails, can you just restart it? How does the base clean-up the state of the lost upcalls? LRPC covers these cases pretty well, RMI less so.
- Are there shared threads? I.e. does a thread cross the boundary or is there some kind of handoff? RMI and LRPC clearly have a handoff and thus use separate threads. Extensions share threads; Spin kernel crossings do not share threads. Can an extension swallow a thread and never return it? This is usually the fear that drives for separate threads an each side.
- Evolution: can the two sides evolve independently? For long-lived systems this is a prerequisite that is normally ignored. It is usually easy to evolve the extension, but how do you evolve the base without breaking all of the extensions? Mac OS has probably done the best at this. One answer is to never evolve a particular interface -- make interfaces immutable and add new interfaces to evolve the system. The old ones are left just to support legacy extensions. Who decides that a new version of the base is backward compatible (and what if they're wrong)? For big upgrades, it is common to force evolution of all of the extensions (e.g. for Windows 2000, most drivers must be updated, at least those from Windows 95/98). Microsoft COM has unique IDs for every interface so that an evolved interface is clearly marked as different; conversely, if you get the same ID, then you know you have the same interface (but not necessarily the same implementation!)
- Admission control: is there a limit on crossings, or on concurrent crossings? RMI has this problem in spades -- every new call typically spawns a new thread and runs concurrently, eating up resources until the system falls down. Is the admission control general or per-client? In general, the way to solve this is with explicit queues for incoming crossings; excess crossings can be dropped or failed immediately. A count of outstanding calls may also work, as long as it doesn't leak (which seems to be surprisingly hard in practice). For synchronous interface, the number of threads limits the number of crossings, but for async the limit must be enforced some other way.
- Resource allocation: how do extensions get resources and how are their resource limits defined? This comes up in network/disk buffers, physical memory (ACPM has a nice solution), and sockets among others. In general, it is hard for a server to correctly account for the resource usage of its clients (on a per-client basis). Hard cases including

shared overhead in the VM or interrupt system for example.

- Can you uninstall an extension? Ideally, installing an extension would be transactional, and if you aborted it, everything would return to normal. I don't really know of any systems that achieve this, but DLLs clearly do not. What if applications are using the extension? Typically would you stop new apps from using it, and then wait for the current ones to stop. Files systems have to do this for deleted files.
- upcall summary: use pass-by-value, or externalized references. It is useful to have timeouts for calls, and expirations for references passed out. Externalized references are really opaque tokens that have no use to the extension except as something to pass back later.
- downcall summary: decide if it is sync or async; check args on the way down; don't accept true pass by reference (but pass by value-result is OK, since there is no shared region). It is useful to enable app-specific "closures" to be passed in. These are tokens that are opaque to the base, but allow the application to have context when a result is returned, and thus simplifies reply matching for the asynchronous case.

In Spin, there are two kinds of boundaries: extension boundaries and the user-level boundary. The latter is a fairly typical kernel crossing with arg checking. Externalized references for the user-level boundary, while type-safety is for the extension boundary. Note that references are OK across the extension boundary because there is a shared GC system (as well as shared memory and address space). Extensions shares threads, but threads never cross the user-level boundary -- during a blocking call there are two threads involved, one for user level (which blocks) and one for the kernel.

Unix Interface:

- Mostly written in C on top of C-Threads implementation of the Strand interface
- Small Modula-3 core underneath the Unix C code
- Similar to Exokernel LibOS concept

Capabilities:

- just type-safe pointers!
- enforced mostly at compile time (e.g. except for array bounds checks)
- externalize these pointers if you need to give them to untrusted code, just file descriptors in Unix (an array of pointers and you pass out the index).

Domains:

- Just of set of (name, object) bindings used for dynamic linking
- Resolve looks up unresolved symbols in 'target' against exported symbols in 'source'
- Use two "resolves" to cross-link domains
- Linked domains are in the same address space: the new links are new capabilities
- Can further restrict these links with run-time checks

Event dispatch:

- the default case is one handler for each source of events

- But may have many handlers for an event -- need to order them and combine them
- Generates combination code dynamically as handlers are added
- Each handler may have a guard, so that it only applies in some situations -- this is the way to demultiplex events (e.g. network protocol dispatch)
- Also need to figure out “return value” for an event, which is not clear since it may cause many callees to execute. Default policy is the return value of the last handler executed, but the order is undefined (!).

II. Extensible Kernels are Leading OS Research Astray

Claims made by extensible OS researchers:

- safely achieve high performance for wide range of apps
- rapidly deploy OS innovations
- contain/manage increasing OS complexity

Counterclaim: only the first one has been addressed, and complexity is in fact much worse

Basic problems:

- don’t need general-purpose extensibility -- best ideas can just be added to a regular OS without the need for new mechanisms for safety
- easily modified boundaries are not a good idea long term: hinders compatibility, interoperability, and the ability to evolve... (must you support everything that was extended in the past?)
- safe extensions introduce new complex problems that are hard to solve (witness XN disk block scheme, trusted compilers, etc.)

Alternatives are OK:

- simple “little language” extensions (packet filters)
- existing techniques: ACPM, scheduler activations, U-Net are all safe and much simpler

Example: web server

- TCP implementation: just fix it directly (done now)
- TCP protocol optimization: better interface is sufficient
- forking/switching: better threads (Capriccio)
- filesystem: several solutions, including specialized file systems for web caches
- VM/cache control: better interface is sufficient
- copying overhead: U-net or IO-Lite
- double buffering: ACPM
- TCP checksum calculation: network layer can cache checksums itself (more reliable)