SOLUTION NOTES

Advanced Systems Topics 2003 Paper 8 Question 5 (SMH)

(*a*) (*i*) Motivation for extensibility

There are several motivations. The most applicable is to allow application-specific behaviour/specialisation. In the more general sense, extensibility also allows the fixing of mistakes, the support of new features, etc.

(*ii*) Three techniques for extensibility

The most straightforward (conceptually at least) is to allow applications to 'download' code into the kernel. This is the approach taken by e.g. SPIN and Vino; certain well defined pieces of functionality (thread scheduling, page replacement, prefetching heuritics) are offered for replacement. Performance in this case is great, at least once the 'graft' or 'spindle' has been verified (code is executed directly in kernel with little or no additional overhead). Major downside is trying to ensure safety; in SPIN this is done by a combination of language-level techniques and a 'trusted' compiler (digital signatures); in Vino sandboxing and lightweight 'transactions' are supposed to solve this. Doesn't really work though. Granularity of extensibility is good here, but is pre-defined (i.e. if haven't thought about a particular graft point cannot add it later without rebuilding OS – need 'meta extensibility').

A second scheme is to use a virtual machine monitor or hypervisor. This allows each user to in principle run whatever operating system they like. Examples of this approach include Disco, VMWare, Denali, and XenoServers. In terms of performance these always have both space and time overhead; the former since the VMM requires its own code and data, the latter since essentially an extra layer of indirection is present in accessing most hardware resources. The 'para virtualization' techniques used by the impurse VMMs (Cellular Disco, Denali, XenoServers) can mitigate this cost substantially however. The protection model here is nice and clean since resource multiplexing is done at very low level, although need to ensure have decent level of 'isolation' in the VMM (e.g. by using SRT scheduling techniques, admission control, etc). The granularity of extensibility is rather coarse however; one needs to replace the entire operating system in most cases.

A third scheme involves moving the protection boundary lower while leaving abstraction boundaries higher. This is the approach taken by the Exokernel and Nemesis systems; extensibility is provided by having most OS functionality

in user-space libraries. These are then easy to replace. The granularity of extensibility is somewhere between the previous two – generally entire libraries (Exokernel) or modules (Nemesis) need be replaced. However different applications can choose different, overlapping subsets without major problems (providing interfaces are defined correctly). Performance here can be really good in some cases; the Exokernel people in particular showed excellent performance using a specialised 'library OS' for web serving. Protection is pretty good, being a pretty low level. As with VMM approaches, however, need to be sure have good levels of isolation.

(*b*) (*i*) Extensible VMs

Desirable to allow application specific control over abstractions provided at too general a level e.g thread scheduling, heap placement, garbage collection, etc.

(*ii*) Building XVM

This could be achieved in a number of ways; for example one could extend the definition of the virtual machine to allow additional bytecode operations; one could expose a selection of implementations of functionality within the JVM as explicit objects and allow selection (perhaps via classloading functions); or one could add 'callbacks' or 'upcalls' to enable application threads to make specific calls. The key thing here is that programs are guaranteed to be well typed and cannot e.g. get dangerous references, unlike in the operating system case.