Extensible Systems and Virtual Machine Monitors

Papers:
- SPIN
- Exokernel
- Microkernel
- Cache Kernel
- V
- VM/370
- VM Survey paper
- Scheduler Activations

Questions:
1.) Why would you want an extensible system?
2.) What is bad (that we can fix) about a monolithic kernel?
3.) What are the problems to be tackled with extensible systems?
4.) What are the approaches? What is the "correct" abstraction?
5.) What is a VMM?
6.) What are the advantages/disadvantages of VMM?
7.) Comparison of extensible systems and VMM.

Q: What is bad about a monolithic kernel?
- not easy to add new features
- has to serve general purpose functionality - no optimization for specific needs (ex. scheduling)
- pay penalty even if you don't use the feature
- calls into kernel are expensive
- it's not easy to debug or verify
- protection between modules is difficult because supervisor mode has a far reach
- can't specialize functionality across several hw in order to get better resource utilization
- only one kernel can be directly on top of hw

Q: What can an extensible system give you over a monolithic kernel?
- tackles issues of kernel crossing by moving things to user level or by downloading functionality into kernel so cost of call is equal to procedure call
- write your own specialized functionality to improve performance
- fault/failure containment
    - maintain separation between programs
    - protection between os-like modules
- smaller kernel is easier to debug/verify
- very distinct os-modules can be running simultaneously

Q: What are the approaches? What is the "correct" abstraction?

Abstraction:
1.) Present basic core data structures/primities
    a.) Threads
    b.) Address spaces
    c.) IPC
    systems: Microkernel, Cache Kernel, V
2.) Present close to hw primitives that allow you to implement 1. as desired
    ex.) you get to implement whatever page table structure you want
    systems: SPIN, Exokernel

Q: How do you provide extensibility?
1.) Export functionality to user level processes
    -> cost of domain crossing is cheaper
    systems: V, Cache Kernel, Exokernel, Microkernel
2.) Download functionality into kernel
    -> functionality provided at cost of procedure call
    systems: SPIN, Exokernel
    - take into account how safe what you're downloading is
        SPIN - typed languages. extensions are given good seal of approval
        Exokernel - sandboxing and other techniques

Specific Systems
V:
- small os kernel on each node implementing basic protocols and services providing simple network-transparent process, address space, and communication model
- extensible in sense that you get to implement own server modules, where they must only follow the protocol dictated and respond to IPC protocol
- kernel has time, process, memory, communication, and device modules. however, most functionality for these handled in user space.
- server only manages local objects
- all communication done through IPC (no need for syscall)
- entire system hinges on multicast

Cache Kernel:
- Cache Kernel is underlying os
- provides threads, address spaces, and IPC
- basic function to cache specific objects (threads and address spaces and kernel objects) - active mappings
- kernel objects are user level resource managers (here's the extensibility)
- exceptions and traps forwarded by Cache Kernel to application kernel (Fig. 2)
- protection provided because users can extend supervisor-level

Microkernel:

- microkernels haven't worked before because you're all idiots. you implemented badly.
- basic abstractions are thread, address space, and IPC
- I/O is memory mapped
- only put things in kernel that had to be there for protection and independence
- interrupts transferred into msgs by kernel and passed to some user level process

Exokernel:
- untrusted, application-level resource managers
- minimal kernel that securely multiplexes available HW
- export resources to user level
    to do securely:
    - secure bindings
    - visible revocation
    - abort protocol
- design principles:
    - securely expose hw
    - expose allocation - libOS can ask for specific resources
    - expose revocaton - allows libOS to perform effective level resource mgmt.
- download code:
    - can be run when context switch to application not feasible (ASH)
    - runtime must be bounded

SPIN:
- safety - apps protected from one another. access controlled at same granularity os extensions defined
- performance - low overhead communication between extension and system
- dynamically link into kernel virtual address space these extensions (co-location)
- extensions can only access items they've explicitly been given access to
    - run in response to system events. events declared in interfaces and bound at runtime (cost is procedure call)
- extensions certified by compiler
- kernel resources referenced by capabilities (created through language)
- extension model:
    - defined in terms of events and handlers
    - extension installs handler on an event by registering handler with central dispatcher
    - handler can run as procedure call or separate thread
- building blocks (higher than hw) for abstractions:
    - strands
    - physical addresses
    - virtual addresses

- translations

Scheduler Activations:
Problem:
- user level threads exhibit poor behavior when multiprogramming and I/O are used
- kernel level threads too heavyweight and perform worse than user-level threads
- neither thread has enough info
- user level can't take advantage of multiprocessors

Idea:
- get best of both worlds but this means distributing control and scheduling info between kernel and app address spaces.
- each app given virtual MP (w/designated # of processors) which given complete control over
- app gets to decide what gets to run on those processors
- kernel decides how many processors each app gets. changes dynamically
- scheduler activation vectors control between two
- events vectored exactly at points where kernel would otherwise make scheduling decisions
- app notifies kernel when needs more processors or has idle ones

Key: each app gets to decide which threads to run (extensible)

Virtual Machines
Q: What is a VMM?
VMM - software layer that runs directly on hw
VM - each OS running on VMM thinks it has its own complete set of hw
VMM - allows different OSes to run concurrently

Advantages:
- can pretend you have diff hw. good for debugging and development of OS
- can run multiple OSes at once
- allows legacy code to be supported past hw/architecture changes
- allows you to write speciailized, simple OS that does not need full functionality
- one OS going down doesn't hurt any other OSes and their apps

Disadvantages:
- slow
- overhead
- VMM resource mgmt. may be counterproductive to OS mgmt.
- you might be less willing to change/optimize software because old code still works (backwards compatible)

Extensible Systems vs. VMM
- theoretically extensible systems give better performance

- if you consider tiny kernel = VMM, then protection is equivalent
- changes to specific mgmt. of resource in VMM still requires change of ugly OS. however, VMM allows you to implement really small OS that could do this specialized stuff.
- extensible system is just more fine-grained that VMM
- extensible system lets you make changes w/out root access
- can't run commodity OS on top of extensible system
- you don't have to put much effort into VMM approach. just slap OS on. systems require writing libOS stuff.