

Lecture Notes, 4/10/2008

K42

I. Background

Several assumptions in 1996:

- o *Windows would dominate everything except very high end (so focus there)*
Not true; shifted their strategy to promote/leverage Linux in 2000
Decided not to support multiple personalities => less need for customizability
- o *Multiprocessors would become ubiquitous; especially NUMA*
Manycore is here, but arrived slower than expected
NUMA not true yet: hardware trying to make systems mostly uniform; more clusters of UMA than NUMA shared memory
- o *Maintenance/development cost would dominate*
Generally true: still not very modular, which hinders development. Places that are modular, kernel-loadable modules, seem to have more innovation
- o *Will need to be customizable/extensible*
Has been useful, but complex to implement; VMs change the picture some as do the ability of clusters to handle the availability issues (so you can take down a node easily). IBM developed its own hypervisor for fault containment and to co-exist as a guest OS
- o *All machines moving to 64-bit*
Still coming, but really only at the high end so far. K42 spent a huge amount of time creating the 64-bit open source community and it still limits them some

Basics:

- o Aimed for small kernel, with much functionality in user-level libraries
 - enables customization/extensibility
 - enable multiple “personalities” over the same core (but now less needed with rise of Linux, VMs)
- o Extensive use of OOP
- o Aim for scalability to many cores/CPU's with shared memory
 - avoid global locks!
- o Multiplex multiple OSs in time -- seems like a bad idea compared to VMs
- o strong support for Linux ABI

II. Scalability

Two broad approaches:

K42

- o fine-grain locking (not generally true for other OSs)
- o memory locality
- o some per-CPU memory

Approaches:

- o protected procedure call (PPC):
 - cross-address space (client to server)
 - both sides run on the same processor (for memory locality)
 - each client request spawns a server thread (EB: might want to limit this with a thread pool); client thread blocks
- o Locality aware memory allocation (think free list for each processor)
- o Use of local, fine-grain objects to ensure fine-grain locking (per object); also enables customizability
- o Cluster objects (covered below)
- o In general, don't block in the kernel (like capriccio)

Memory techniques:

- o Partition state among CPUs; enables scalability and locality
- o Push page faults up to app (app blocks, but OS is event driven)
- o Processor specific memory (used for clustered objects among other things)

Clustered Objects:

- o Basic idea: a set of objects, one per processor, that work together to implement a service
- o Mechanism: indirect call using COIDs (clustered object IDs) -- each CPU has a COID to function pointer table to find the local object
- o Objects could be different, generally different instances of the same class; must at least have the same interface
- o $0 \leq \# \text{ objects} \leq \# \text{ CPUs}$
 - 0 because objects can be created lazily; invoking the object causes it to be created
- o Local object called the "rep"
- o Easy part: scales well, has fine-grained locking
- o Hard part: clustered objects must manage shared state among themselves
 - reps have pointer to "root" object that manages single-copy shared state
- o Nice point: can vary the # of objects over time based on load

K42

III. User-Kernel interface

Scheduling:

- o Dispatcher: manage user-level threads, handle events from the kernel
- o traditional problem: user-thread blocks in kernel, then app makes no progress -- need a way to switch to a different user thread
- o solution: user thread blocks, but dispatcher gets an upcall and starts a new thread
- o another upcall when blocked thread is runnable
- o dispatcher thus also handles page faults
- o kernel sees only a small number of dispatchers (usually 1) regardless of number of user-level threads
- o very similar in spirit to Capriccio

IPC:

- o handles both event-based async calls as well as same-CPU PPC
- o async call can cross processors as well as processes
- o system calls often done via “trap reflection” -- use a trap and then redirect back into user space (44% slower than a direct branch); reflection allows binary Linux compatibility

Dynamic composition:

- o relatively complex ability to hot-swap an object in a running system
- o must transfer active state
- o must have a level of indirection so that you can changes all references to it...
- o must quiesce the object