Advanced Topics in Computer Systems, CS262A
Prof. Eric Brewer

K42

I. Background

Several assumptions in 1996:

- *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
- *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
- *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
- *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
- *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:

- 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
- Extensive use of OOP
- Aim for scalability to many cores/CPUs with shared memory
  - avoid global locks!
- Multiplex multiple OSs in time -- seems like a bad idea compared to VMs

II. Scalability

Two broad approaches:

- fine-grain locking (not generally true for other OSs)
- memory locality

K42

- 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<= # objects <= #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

III. User-Kernel interface

Scheduling:

K42

- kernel schedule address spaces
- user-level schedules threads
- process = address space + one or more dispatchers
- multiple dispatchers for multiple cores or for different priorities/QoS
- threads can block for page faults (for example), but dispatcher retains control of the core
  - page fault => dispatcher receives upcall
  - halts offending thread
  - runs something else
- similarly, systems calls can block the thread without blocking the dispatcher
- priorities first, then lottery within one priority
- Posix can be on top of dispatchers

Message passing

- both sync and async messages between cores
- server process can export an object, and clients can call its methods via messages
- async calls have no reply and don't block the caller
- soft interrupt can be used to notify other dispatchers in the same address space (process)
-