Advanced Topics in Computer Systems, CS262B

Prof. Eric A. Brewer

Exokernel

January 22, 2004

Key Ideas:

- separate protection from policy: normal kernel does both
- default policy might not be the best, but very hard to change
- challenge: how to structure protection for flexibility *and* sharing among untrusted processes
- move most of traditional OS to user-level library (a “lib OS”), to maintain compatibility

Extends the trend of better support for application control:

- ACPM: control over paging and replacement
- Scheduler Activations: control over threads and scheduling (also Capriccio)
- U-net and others: control over networking

Principles:

- Separate protection and management: use the lowest level required for protection, ideally the hardware level (e.g. disk blocks). Types of calls: allocation/revocation, sharing, tracking/changing ownership
- Explicit allocation and revocation
- Use physical names: avoid translation from virtual names (and its consistency problem)
- Expose information: typically read-only access to kernel state for making decisions in the libOS

Four basic tools: [slightly different than written in the paper!]

- software regions: like Multics segments -- sub-page protection and fault isolation
- hierarchical capabilities, which must be passed on most system calls
- bind hardware resources together (e.g. disk block and page frame that holds it)
- download code (that can be verified) to control some policies. E.g. wakeup predicates determine if a process is ready to run (without waking it up)

Multiplexing Storage -- extraordinarily complicated!

- goal: allow each app to have its own file system, but on the same disk, and share file systems when needed
- challenge: have to figure out which blocks go with which file system, for almost arbitrary file systems (that don’t trust each other!)

Some attempts that didn’t work:

- capability for each block (or extent): would work, except that you need to either put it in the block, which is a mess, or put it somewhere else have many more seeks... [would caching solve this?]
- self-describing metadata: so that XN can figure out where the references are without knowing all the details about the file system. Hard to find the right language for this.
- template-based metadata: use more complex language, but only once for each template. But still couldn't find the right language; i.e. flexible but simple

Actual solution: UDFs: *untrusted deterministic functions*

- idea: don't try to understand the metadata -- just make the file system tell you in a consistent way what the references are
- Usage: file system author writes a function for each template type, *owns-udf*$_T$, and gives it to the kernel, which verifies that it is deterministic. The kernel then uses this function to verify every proposed change to a block, so that libOS can't change references without the kernels permission.
- *owns-udf*$_T$(block) -> list of (block #, template type) pairs
  This returns the (deterministic) set of pointers in the input block
- Example: to add a data block, *d*, to an i-node, *i*, you call something like:
  XN_allocate(*i, d, changes to i*)
  XN would create *i'* by applying the changes to *i*, and would confirm that:
  owns-udf$_T$(*i'*) = *owns-udf*$_T$(*i*) + *d*

Access control for pages:

- done when a block is mapped into an address space -- which means you don't need to check it on every access (bind + fast access)

Must also deal with write ordering for persistence (read the paper for this)

- root list keeps the root block for each file system, from which XN can build the whole tree (using *owns-udf*$_T$ recursively)

Buffer cache registry:

- map of all pages and their access privileges (and dirty/clean)
- mapped read-only into libOSs
- any process can write back pages! (can see the data, but can force the writeback)

Fork is challenging:

- must copy your own process while you are running!
- idea: copy all of the pages that are not the active data/stack pages via copy-on-write, but handle the hard ones directly

Related Work:

- microkernels: help this problem via modularity -- easier to replace one part of the OS (to change its policy)
- faster system calls: help performance but not policy

- "info kernel" work: make more info available to user level, and then trick the OS into doing the right thing. For example, you can change the caching policy by issuing the right set of reads (from the lib OS layer). [from Univ. of Wisconsin, SOSP 2003]

Cheetah is compelling:

- high performance, highly tuned web server
- precompute TCP checksum in cache, so you can send them directly over the network
- avoid extra ack packets by knowing the protocol
- colocate files that are part of the same HTML page to reduce seeks
- about 4-8 x faster for small docs, maybe 2x for large docs

Some lessons:

- changing layers has more impact than optimizing the layers as they stand
- fast upcalls more useful than downloadable interrupt handlers (and easier)
- download code to extend policy (wakeup predicates)
- is it worth all this effort? how many applications will benefit? could most of the value by received via changes to Linux instead?