Resource Isolation

1. Web server design
   a. Basic problems
      i. Lots of requests
      ii. Requests may require disk i/o (for static data) or network i/o (for proxy cache)
      iii. Data may not all arrive at once or be sent at once (if large)
      iv. how do you take advantage of sleep time while waiting for i/o?
      v. How do you store context of a request while you are waiting?
         1. On a stack?
         2. In an object?
      vi. How do you isolate independent requests?
         1. for reliability?
         2. For performance?
   b. Designs
      i. Fork-a-process (standard unix)
         1. listen in master process, fork off & call accept in worker
         2. client lives until connection goes away
         3. problem: for short connections (http 1.0 closes after every request), lots of overhead to fork processes
         4. Hard to fine tune scheduler with short lived process – doesn’t reach steady state
      ii. Pool of processes – becomes process per request not connection
         1. Master process listens, hands off socket to an existing worker process
         2. worker process does work
         3. worker adds self to queue waiting for next request
         4. problem: expensive context switching
      iii. Single process / thread per connection
         1. Thread creating for each request, handles request, dies
         2. Problem: lots of context switching (but not as bad)
3. Problem: lots of thread creation (expensive if need to do i/o in kernel)
4. Problem: loss of protection for dynamic data
5. Problem: thread scheduling by kernel, hard to prioritize

iv. Single process / thread per request pool
   1. Create pool of threads
   2. Thread handles a request, goes back in pool
   3. Most of same problems, but less overhead to create a request

v. Single process / single thread / Event driven
   1. All asynchronous / non-blocking I/O
   2. No scheduling in kernel
   3. app maintains queues of requests to handle at different stages
      a. receiving data
      b. sending data
      c. reading disk
   4. At each stage, app can choose what to do, when
   5. Can do lots of work on a stage to improve cache locality
   6. Problem: hard to write, can’t use OS scheduling to account for work done in kernel on behalf of a request

2. Accounting problems
   a. Process = unit of accounting for standard Unix, Linux, Windows
   b. Kernel work on behalf of a process not accounted to process
      i. Accounting done to currently executing thread (or nobody)
         1. paging – replacing a page for someone else to use
         2. network packet receipt
         3. block device transfer
   c. This doesn’t fit web servers
      i. Heterogeneous workload / different priorities per work item not visible at process level
      ii. Lots of work in web server happens in kernel (70% for apache with static web pages)
      iii. Work may encompass multiple threads in a process / multiple processes, or they may all be doing different things
d. Big Picture: to share things fairly, you need to measure accurately/correctly
   i. Example: sharing a network
      1. How deal with small packets (few bytes, high overhead)
      2. Disks: random vs sequential?

3. Goal:
   a. Flexible definition of an accounting unit
      i. Can be < 1 thread, > 1 thread, > 1 process
      ii. Include kernel work on behalf of an accounting unit
      iii. Use for all kernel scheduling – any time something is queued, it or given resources, it should be subject to this

4. Resource containers
   a. Like Opal, break apart an abstraction into finer grained, more flexible components
      i. Standard technique in systems – like exokernel
      ii. QUESTION: what are problems?
         1. Making sure new abstraction is right
         2. New one is usually finer-grained, need to minimize overhead
         3. Protection: process is well understood, need new model
   b. Concept:
      i. resource container == unit to which accounting happens, guides scheduling decisions instead of a process
      ii. Contains all system resources used by an activity
         1. cpu time
         2. kernel objects, buffers
      iii. Have attributes
         1. scheduling parameters
         2. resource limits
         3. network qos
      iv. Separate from security principals (users)
      v. QUESTION: how do you draw this architecturally? Is it a layer? What is the overhead cost at runtime? When do you pay this cost?
         1. A: already collecting stats for accounting to a process
         2. A: new overhead when consulting container for scheduling
         3. A: new overhead when changing containers
4. A: new overhead when accounting things not previously accounted (e.g. network packet receive)

c. Relation to threads / processes
   i. Threads start in a default container, can dynamically bind to others
   ii. Multiple threads / processes can be assigned to a container
   iii. usage: thread working on multiple things changes container as it changes activities
       1. e.g. thread pool per request, event driven model
   iv. Can pass between processes (like an FD) for cooperating processes

d. Kernel activity
   i. Kernel switch RC when processing a request
      1. Handling a network packet
      2. Processing a disk request
      3. Account interrupt for the process receiving the packet – record time, when figure out who gets it, account backwards to when started processing

e. Scheduling
   i. RC allows scheduling per activity not just per process / thread
   ii. Base case: thread is scheduled according container it is bound to
   iii. Threads bound to multiple containers use scheduler binding
       1. == set of containers it has been using
       2. combined to make decision (e.g. average, sum if not shared)
       3. NOT DETAILED
   iv. QUESTION: What if you have a single-threaded event system?
       1. Issue: Thread does scheduling of what event to handle
       2. Solution: use usage info from kernel to decide what to do locally
   v. ISSUE: what if you have user-mode threads?
       1. Same solution – use kernel information

f. Introspection
   i. Application can inspect accounting information for its own use
g. Nesting
   i. Like Opal resource groups, Nucleus
   ii. If give a proportion to a container, == sum of child containers
   iii. E.g., give 20% of CPU to cgi/bin – can then have a container per request in cgi/bin container

h. Network usage
   i. Can bind a socket to an RC
   ii. All packets on the socket are accounted to an RC
   iii. Socket can be a packet filter (group of addresses)
      1. Allows scheduling before a connection is established
      2. Allows giving higher priority to some group before they connect
      3. Allows classes of packets to be given no resources – black hole

i. Multiple request classes:
   i. Create a top container for each class
      1. Eg. Create new account
      2. Browse
      3. Buy something
   ii. Assign share/priority to each one
   iii. Create RC for each request
   iv. Switch RC once you figure out what the request is for
   v. QUESTION: What about requests that do multiple things?
      1. No DAG as in Lottery Scheduling

j. Protection:
   i. Who can set container parameters?
      1. Parent can set for child
      2. Child inherits portion of parent
      3. Child can lower self?
   ii. What if a cgi-bin script is malicious?

k. QUESTION: is this general purpose?
   i. Could you use it in a file server to allocate cache space, disk bandwidth, network bandwidth?
   ii. Does it apply in general to server applications with multiple clients?

l. QUESTION: What about TLB misses?
   i. Problem: in x86, no SW to run so cost not visible
   ii. Key approach in RC is accurate measurement/accounting
      1. TLB flush is not the only cost – impacts later usage
5. Uses
   a. Assign percent to cgi scripts
   b. limit runtime of cgi scripts
   c. DOS attack
      i. Goal: find a signature for attack packets, drop them early
      ii. Goal: prevent a single code path from being over used – e.g. slow cgi/bin path
      iii. limit code paths – e.g. all syn packets get % of time, then dropped
      iv. Ip-based filters – to black hole
      v. RC limit: filters are only on address, not packet contents.

6. Evaluation
   a. Cost: how expensive are primitives?
   b. QUESTION: what is missing?
      i. e.g. cost of having kernel use RC, frequency of RC invocation in kernel
   c. QUESTION: how evaluate?
      i. With single-process event driven server
      ii. Show that can constrain resources based on various classifications (e.g. max fraction, proportional share)

7. Relevance today
   a. MS has jobs – groups of processes that share scheduling / accounting / security privilege information
   b. Linux has CGROUPS – for CPU

8. Overall:
   a. Like VMs for resource isolation, but at less-than-a-thread granularity (much cheaper)