CKRM:

Class-based Prioritized Resource Control in Linux

Hubertus Franke, Shailabh Nagar, Jonghyuk Choi, Mike Kravetz, Chandra Seetharaman, Vivek Kashyap, Nivedita Singhvi, Scott Kaplan Haoquiang Zheng, Jiantao Kong

IBM T.J. Watson Research Center
IBM Linux Technology Center
Amherst College
Outline

- Motivation
- Framework
- Classification
- CPU
- Memory
- I/O
- Network
- Conclusions
Linux Kernel Resource Management

• Process centric
  - nice value for cpu
  - rss limits for memory
  - Flexible outbound network QoS

• Performance isolation mechanisms are weak

• New “fairshare” proposals still process/user centric
  • fairshare cpu scheduler (RvR)
  • Complete Fair Queueing I/O (Jens Axboe/AA)
Server Requirements

- “Work” (users, transaction, appl, ..) has varying levels of importance unknown to the kernel
- Different “work” is colocated in a single system
- Need ability to classify work by importance
- Need ability to differentiate service provided
  - QoS is typically based on end-user goals
    - Transaction latency, bandwidth, response time
  - Resource share needs to be specified external to kernel
- Monitor resource consumption by “work”
What is Class-based Kernel Resource Management?

- Attempt to make Linux kernel meet said server requirements better

- Driving Principles for CKRM
  - Flexible, dynamic grouping of processes into classes
  - Resource shares for each class
  - Kernel enforcement of shares for each phys resource
    - Requires scheduler (cpu, mem, i/o, net) modifications
  - Grouping rules and shares specified externally through a system wide policy
Target Scenario 1: Enterprise Server Configuration

- Class determined by
  - who, how, what
- Different expected QoS for each class:
  - Response time, bandwidth utilization

- Example Stock trading:
  - **Gold**: high volume trader initiating a transaction
  - **Silver**: all other stock trading
  - **Bronze**: mutual fund transactions quotes
Target Scenario 2: Virtual Hosting

- Virtual Hosting using UML, apps run as processes under host system together with guest OS
- Every system resource needs to be regulated
- Service guarantees for each UML instance
Target Scenario 3: Desktop

• More control over performance isolation of activities:
  – Compile code while (emailing, listening to music ..)
  – Scheduled Backup disk / Virus check while working
  – Limitations for ftp / telnet sessions
Why kernel changes?

- From user space certain QoS can not be done
  - Some limits do not exist (e.g. I/O bandwidth)
  - Some hard to specify for dynamic workloads

- Kernel is central agent for resource control
  - Natural place to do “this kind of stuff”
  - Thesis: this can be done with modest changes to code and performance of existing resource schedulers
CKRM : Key Concepts

• Class
  – Policy-defined grouping of tasks/mm's doing work at a common importance level e.g.
    • All web requests from customer X
    • All work initiated by user Y
  – Tasks can dynamically change classes

• Share
  – Portion of a resource that a class can use
  – Dynamically specified by entity external to OS
    • Direct specification by sys-admin/user
    • Indirectly through a root-level userland control program
Classification

- Classification rule
  - \{ \ [(attr,value)]+ \ -> \ class \}
  - Attrs of task: uid, gid, executable, application tag

- Policy
  - classes + classification rules

- Application tags
  - Additional flexibility for grouping based on application specific criteria

- Classification takes place
  - fork(), exec(), setuid(), setgid(), explicit call
**CKRM Framework**

- **Classification Engine**
  - Policy (classes/rules)
  - Classification Engine
    - T2
    - T3
    - T4

- **Control**
  - Share (per resource)
  - Resource Request
  - Class-aware resource allocation
  - Class A
    - T2
    - T3
    - T4

- **Monitoring**
  - Share (per resource)
  - Usage
  - Class-aware resource allocation
  - Resource Schedulers
    - CPU, Disk I/O, Network, Memory

- **User**
  - System Administrator
  - Monitoring events
    - do_timer()
    - end_request()
    - ...

- **Kernel**
  - Resource Management Middleware
  - load

- **User**
  - Load

- **Kernel**
  - Classification Engine
    - shared
    - exec

- **Resource Schedulers**
  - Resource Schedulers
    - CPU, Disk I/O, Network, Memory
Physical resources controlled

- CPU: timer ticks
- Memory: \#physical pages used
- I/O: \#bytes transferred per disk
  - Separate share for each disk visible to OS
- Inbound network: \#connections accepted
Monitoring

- Assess utilization
  - Capacity planning
- Accurate billing
  - Benefit independent of ability to regulate usage
- Feedback for control settings
- Operates at different time scales
  - Cumulative since policy load
  - Since last invocation of “get data”
- Currently no monitoring API exposed
Outline

• Motivation
• Framework
• Classification

• CPU Control in CKRM
• Memory
• I/O
• Network
• Conclusions
Linux 2.5 Scheduler

- Ordered by process priority
- Operations (enqueue, dequeue, get_next_task) are O(1)
- Priority and interactiveness are determined by nice value and sleep_average
- time_slice is determined by nice value, task will expired after time_slice ticks consumed
- Interactive jobs will not expire if they don't starve other processes
- Switch active and expired queue when all tasks expired
Class Fair Share Scheduler

- Each class has its own runqueue
- Minimal changes to the existing scheduler:
  - same runqueue structure
  - same way to calculate time_slice, sleep_average and prio, etc.
  - same O(1) behavior within class
- `get_next_task()` now makes 2 decision
  - First selects the next class to run
  - Then, within that class select the top priority task just as today

**Class Selection:**
- Based on accumulative normalized time per class
  - \( \text{ecp}(C) = \Sigma \text{ticks}(C)/\text{share}(C) \)
  - monotonic increasing function
- Select class \( C \) with \( \text{min}(\text{ecp}(C)) \)
- Consider finite sliding window \( \text{CWIN} \) [min..min+WS]
  - \( \text{min} = \text{min}(\text{ecp}(C)); \)  \( \text{WS} \sim 128,256 \)
- When a class is reactivated (task is rescheduled)
  - \( \text{if} \) (min \( \leq \) \( \text{ecp}(C) < \) min+WS)
    - \( \text{then} \) insert \( C \) at \( \text{WIN}[\text{ecp}(C)] \)
    - \( \text{else} \) insert \( C \) at \( \text{WIN}[\text{min}] \).
- Provides fairness (shares) only

**Urgency** (Interactivity)
- \( \text{ecp}(C) = (\Sigma\text{ticks}(C)/\text{share}(C)) \ast \text{scale} + \text{top_prio} \)
- High priority in class gives a short term boost

**Scheduler maintains O(1) characteristics**
Throughput Measurement

- 4 classes, with share of (60,30,9,1) respectively
- Each class has 5*3=15 cpu bound jobs with nice value of (-20, -10, 0, 10, 19) respectively
- Fair Sharing among classes: The CPU time received by the classes are proportional to its share (60:30:9:1) during the 30 minutes run.
- Fair to processes within a class: CPU time is proportional to its time_slice (200:151:102:54:10)
- Behavior exactly as desired
  - Same as O(1) within a class
  - Observing shares
Interactiveness Measurement

- Experimental Setting
  - 4 classes, with share of (60,30,9,1) respectively
  - Run cpu bound jobs on gold, blonze and best effort class
  - Run one interactive job in silver class (30%).
    - The interactive job will run for N ms;
    - then sleep for 200ms.
    - N varies from 50 to 500ms.
Using CFS the cpu usage of the interactive job is roughly 30% 
Class Fair Scheduler receive much smooth service because of performance isolation
Scheduling Overhead

- Measured using Lmbench
  - `lat_ctx -s 0 $N, N=(2..256)`
- Scalability: the overhead of Class Fair Scheduler increases at about the same pace as Linux 2.5 Scheduler
- The static overhead (class – linux) varies from 0.14us to 0.63us during the measurement
- Since class selection is O(1), i.e. Independent of #classes, there are no scalability concerns with #classes
- Code optimization might further reduce the static overhead
SMP / Load Balancing

- Achieve global shares per class
- Maintain fairness within class (nice ratios)
- Tasks in same class/nice need similar progress
- Balancing runqueue length insufficient
- Solution: pressure based balancing
- We DO NOT try to attempt to achieve class shares on each cpu
- Progress: ticks/EPOCH
- Estimated progress of class on cpu
  - \( EP(C,cpu) = \sum ts \times ia(ts) \) (maintained)
  - \( P(C,cpu) = EP(C,cpu) / cpu\_usage(C,cpu) \)
  - \( cpu\_usage(C,cpu) \) maintained by MovAvg
  - Ensure that \( P(C,cpu-i) = P(C,cpu-j) \)

![CPU-0: 20 80 0, CPU-1: 30 50 20](image)

Goal: 30 60 10

Possible
SMP and Load Balancing

- Simulation Result
  - (8 CPU, 300 CPU bound jobs)
  - Similar test setting as throughput test
  - Proportional sharing among classes maintained (60:30:9:1)
  - CPU time of tasks (same class, different nice) is proportional to time_slice
  - Tasks (same class, same nice) receive roughly the same service (diff < 4%)
Load Balance (cont.)

- Compare load balancing based on pressure (as describe before) vs runqueue length (used by Linux)
- Define fairness as: the max cpu time vs the min cpu time received by processes with the same class and same nice value.
- The figure shows the fairness achieved by linux vs by pressure under different workload and number of cpus
- Pressure is a better approach in general. The difference can be larger when workloads are interactive.
Outline

- Motivation
- Framework
- Classification
- CPU
  - CKRM Memory Control
- I/O
- Network
- Conclusions
Controlling Memory

• Average number of physical pages resident per-class
  – Does not correspond to page fault rate control

• Control points
  – Page allocation
    • Strict control similar to per-mm rss enforcement
  – Page reclamation
    • Looser control only done under memory pressure
Linux 2.5 Page Reclamation

1. Shrink Zones

2. Refill Inactive
   - 2.1 Recently Accessed

3. Non-reclaimable
   - 3.1 Recently Accessed
   - 3.2 Non-reclaimable
   - 3.3 Reclaim pages
     - if clean

4. Free Pages

PageCache (SwapCache)
CKRM Memory Control Design

• Share is #maximum physical pages used per class
  – hard/soft, min/max variants also possible
• Only control page reclamation
  – classes can exceed shares if no memory pressure
• No distinction between over-share classes
  – reclaim as many pages as needed by shrink_cache()
• Use global active/inactive lists
  – maintains global LRU order
  – overhead of repeated scans of under-share pages
CKRM Memory Control Implementation

Memory Pressure

1. Shrink Zones

Per Memory Zone

2. Choose victims

Arbitrator

Page Usage statistics
Class Share definition

CLASS

3. Refill Inactive

Active list

4. a Recently Accessed

4. b Non-reclaimable

Inactive list

4. c Reclaim pages
Memory Control Testbed

• Testbed
  – Uniproc: 2.4 GHz P4 uniprocessor, 512 MB memory
  – SMP: 8-way 700MHz PIII Xeon, 3 GB main memory

• “173.aplu”: SPEC CPU2000 Benchmark
  – Avg working set size ~ 184 Mbytes (46 Kilopages
  – Execution time (uniproc) ~ 7.85 minutes

• Microbenchmark
  – Working set size, memory access pattern determined by exponential probability distribution
  – Smoother degradation with memory share reduction
Uniproc, 368M memory, “173.applu”

- Two classes, one app per class
- Two scripts run each class/app in a loop for ~10 hrs (~20 minutes per run)
- 92 Kilopages needed, 90 available
- Memory usage for each class collected and averaged over entire expt
- Execution time = avg. for each run

Observations
- Share settings respected
- Execution time decreases by giving more memory share
- Degradation in execution time from no control to equal share case
  $\Rightarrow$ effect of page faulting on CPU scheduling.
Memory control affects CPU scheduling

- Measure memory usage over time for no control case
- Class B, starts second, gets much lower share, makes less progress due to increased page faults
  - improves after first run of Class A finishes
- “Batching” behaviour improves total execution time over equally penalized (equal share) case
Artificial Workload, RSS of 200Mbytes

- Exponential probability distribution
  - Memory access pattern
  - Memory footprint
- Cumulative footprint size with increasing number of page accesses shown above
SMP, 372M, Microbenchmark

- Two classes, one microbenchmark per class
  - Class A accesses memory twice as fast as Class B
- ~400 MB needed, 352MB available
- Memory usage and progress measured every 3 seconds, averaged over entire expt
- Progress rate normalized across classes

Observations
- Share settings respected
- Progress rate increases with more memory share
- System default behaviour and 192/162 share settings show very similar memory share and progress rate
  - Reduced effect of memory share on CPU scheduling on SMP
Advanced Page Reclalm Policies

- Memory share
  *Memory distribution among classes*
- Share max
  *Upper bound of memory usage under memory pressure*
- Share min
  *Guaranteed memory usage*
- Active set size
  *Real usage by each class*
  *Measured statistically by causing soft faults*
  *Can be used to tradeoff under, over share classes*

- Order of choosing victim classes
  *First, classes above share max.*
  *Second, classes having idle pages (Usage > AS)*
  *Third, classes above memory share*
  *Fourth, classes above share min*
Shared Memory Control

- Pages shared by multiple classes complicate accounting

- Shared address space
  - Create class hierarchy with notion of parent classes
  - Group shared pages into system-defined classes, each with multiple parents
    - each parent corresponds to a regular policy-defined class
  - Apportion page reclamation between system-defined class and parent classes appropriately

- Page cache, memory mapped files, shmep
  - Assign pages to (first, most recent, max share) class
  - Treat pages similar to shared address space case
Outline

• Motivation
• Framework
• Classification
• CPU
• Memory

• I/O Control in CKRM

• Network
• Conclusions
Controlling I/O

- I/O bandwidth consumed by each class
  - Bandwidth measured by #bytes of I/O transfers initiated in either direction

- Per-disk shares

- Current design changes I/O scheduler (iosched)
  - Regulation at layers above (filesystem and VM) or below (device driver) also possible
  - iosched changes are simpler and good enough
Cello I/O Scheduler

- Two-level disk scheduler
  - Separate bandwidth from ordering
  - Work conserving
- Class-independent, coarse grain
  - Bandwidth allocation
- Class-specific, fine grain
  - Ordering within class
  - seek-optimizing, EDF
- Good results on Solaris
  - Linux implementations unstable or in progress
Deadline I/O Scheduler

- Improves average read response time
  - Disk utilization secondary
- Separate read/write input Q's
  - Requests sorted by sector (sort) and deadline (fifo)
- Batched transfers to dispatch queue
  - reduce seek overhead
- Implementation similar to Cello
CFQ I/O Scheduler

- Precedence of fairness over throughput
  - Each task has equal share
- Per-task request queues
- Dequeue function implements fairness
  - Roundrobin through non-empty queues
- Simple changes can implement priorities for dequeuing
I/O control requirements

- Weight/priority of I/O request submitter takes precedence over disk utilization
  - Already happening in 2.5 I/O schedulers
    - Anticipatory – per-task performance
    - Complete Fair Queuing (CFQ) – fairness

- Associate I/O request with class of submitter, not task/user
  - Weight of request = weight of submitting class
  - task/user based treatment can be done using classes
Costa I/O Scheduler

- Variant of Cello/CFQ
- Per-class input Q's
  - System queue for urgent/important requests (VM writeout)
- Deque requests using class weight
- Adding deadline
  - sort/fifo lists for each class
- Adding anticipation
  - service another request from same task, adjust class share
- Implementation planned
Outline

- Motivation
- Framework
- Classification
- CPU
- Memory
- I/O
- CKRM Inbound Network Control
- Conclusions
Network QoS

- DiffServ support in Linux provides Internet QoS
- Traffic Control, Netfilter
  - PHBs
    - classifier, marker, shaper/policer, meter
    - Implemented by traffic control / netfilter
- End server QoS support in Linux?
Inbound Network Control

● Motivation
  – Incoming connections initiate resource consumption
  – Head-of-line blocking of high priority connections under network load
    • Persistent connections exacerbate the problem
  – Application level control not enough under high load

● Prioritize acceptance of incoming connections
  – Classify connections using iptables or during in-kernel application protocol processing
  – Reorder socket accept queue
Prioritized Accept Queues

- Classify using (local, remote) x (IP, port)
  - Iptables rules defined
- Split single accept queue into prioritized queues
  - low priority conn requests moved back to SYN queue if accept queue full
  - SYN policing to avoid starving low prio conns
- Shown to prioritize connections effectively
- Drawbacks
  - Classification hard in presence of proxies and multiple classes on same remote host
Proportional Share Scheduling (PSS)

- Variant of PAQ with weights instead of strict priorities
  - Connections accepted from each queue in proportion of weight
- Only controls distribution, not amount of available bandwidth
PSS Experimental Results

- Httpperf clients, Apache web server

- Class 0 : Class 1 = 7:3
- From 300 reqs/sec, acceptance follows the weight
End Server Connection Control

• Head of Line Blocking
  – When service time of a connection is high (e.g. persistent connection)
  – High priority connection requests may block indefinitely

• Multi Accept Queues
  [Voigt01] Priority Accept Queue
  
  [Pradhan02] Proportional Share Accept Queue
design by IBM LTC (Nivedita, Vivek)
Outline

- Motivation
- Framework
- Classification
- CPU
- Memory
- I/O
- Network

- Conclusions
Conclusions

- There is a need for class-based control over all physical resources managed by the kernel.
- A design and implementation exists for CPU and Memory:
  - achieves major objectives
  - small modifications to existing code
- I/O and inbound network in development.
- Ideal candidate for a 2.7 feature.
Getting involved

• Open source project at Sourceforge
• Birds of Feather session
  – 30 minutes, same room
• Participation and feedback invited
CKRM:

Class-based Prioritized Resource Control in Linux

Hubertus Franke, Shailabh Nagar, Jonghyuk Choi, Mike Kravetz, Chandra Seetharaman, Vivek Kashyap, Nivedita Singhvi, Scott Kaplan Haoquiang Zheng, Jiantao Kong

IBM T.J. Watson Research Center
IBM Linux Technology Center
Amherst College