Linux 2.6 Virtual Memory

Linux Kernel Hacking Free Course, third edition 2006

Andrea Arcangeli
andrea(at)cpushare.com

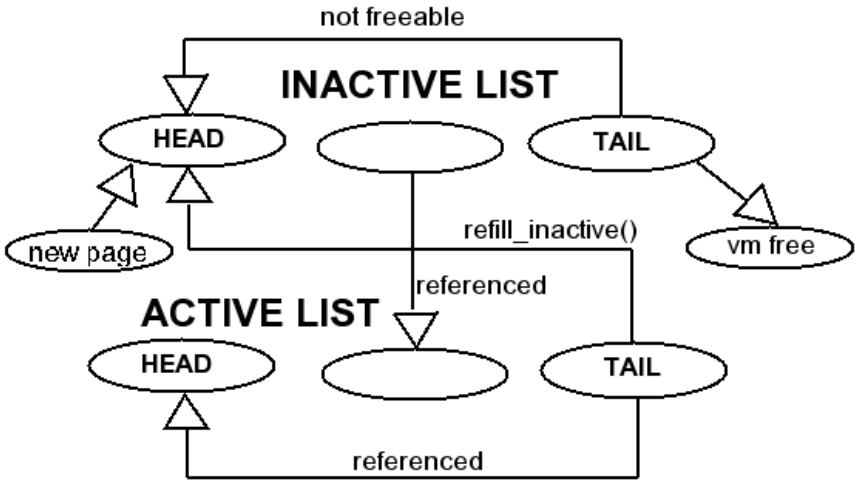
20060412 - Rome

Topic last time

- Last time I covered the 2.5 VM and the original rmap based on pte_chains
- There were problems in the performance of the pte_chains
- Infact no attractive solution to avoid the pte_chains overhead was found until mid 2004
- Today the VM is overall quite optimal
- There are no huge margins for optimizations for the basic workloads on regular hardware

Cache replacement is the same

- Interestingly many things have changed during 2.5 and 2.6 (especially the implementation is almost completely rewritten), but the active/inactive list design for cache replacement introduced in 2.4.10 is still the same today in 2.6.16



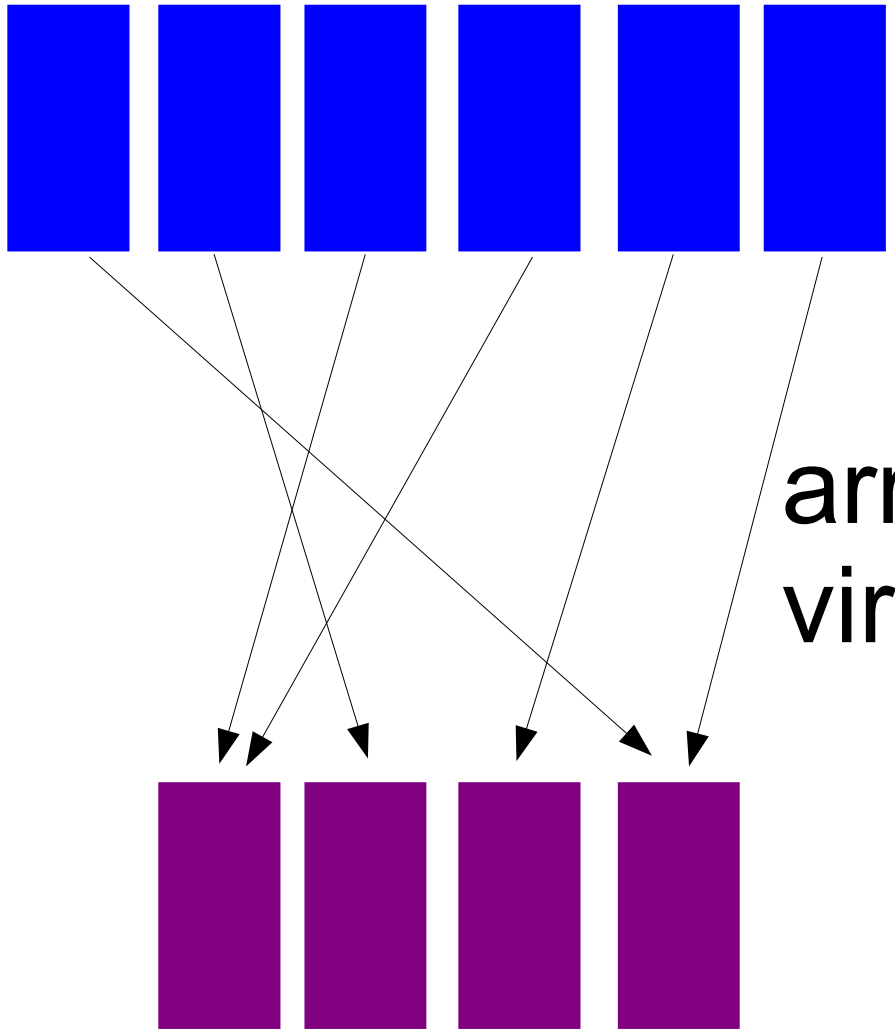
objrmap+anon-vma design

- One significant VM change happened around 2.6.7 has been the introduction of objrmap and anon-vma to replace the pte_chains
- We'll see in the next slides why the pte_chains were useful and how the new objrmap+anon_vma model works

VM basics

- You know userspace memory is virtual
- Pagetables are set by the kernel and they tell the cpu how to translate from virtual addresses to physical addresses
- Before a page of RAM can be swapped out to disk, we must find all the pagetables (i.e. virt addresses) that map to the page and mark them non-present
- Each virtual address maps to only one phys addr but different virtual addresses can point to the same phys addr (SHM)

VM basics (greatly simplified)



Virtual pages
They cost “nothing”

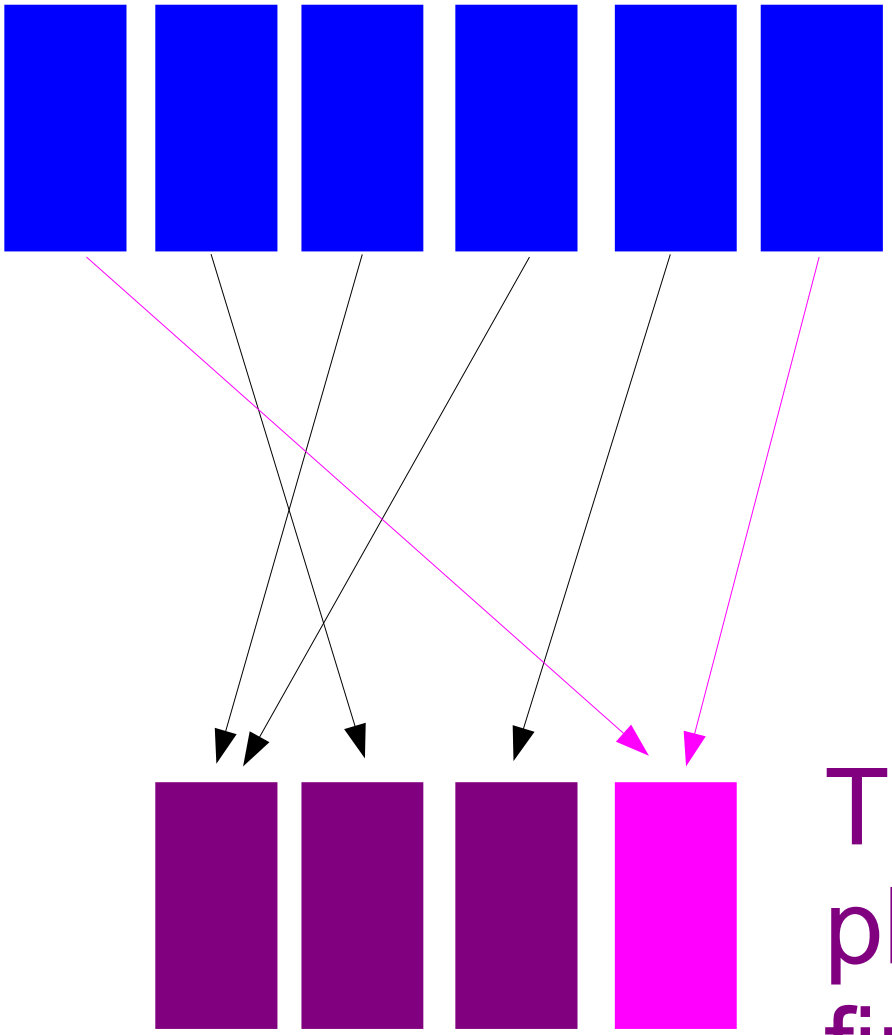
arrows = pagetables
virtual to physical mapping

Physical pages
They cost money!
This is the RAM

VM basics (greatly simplified)

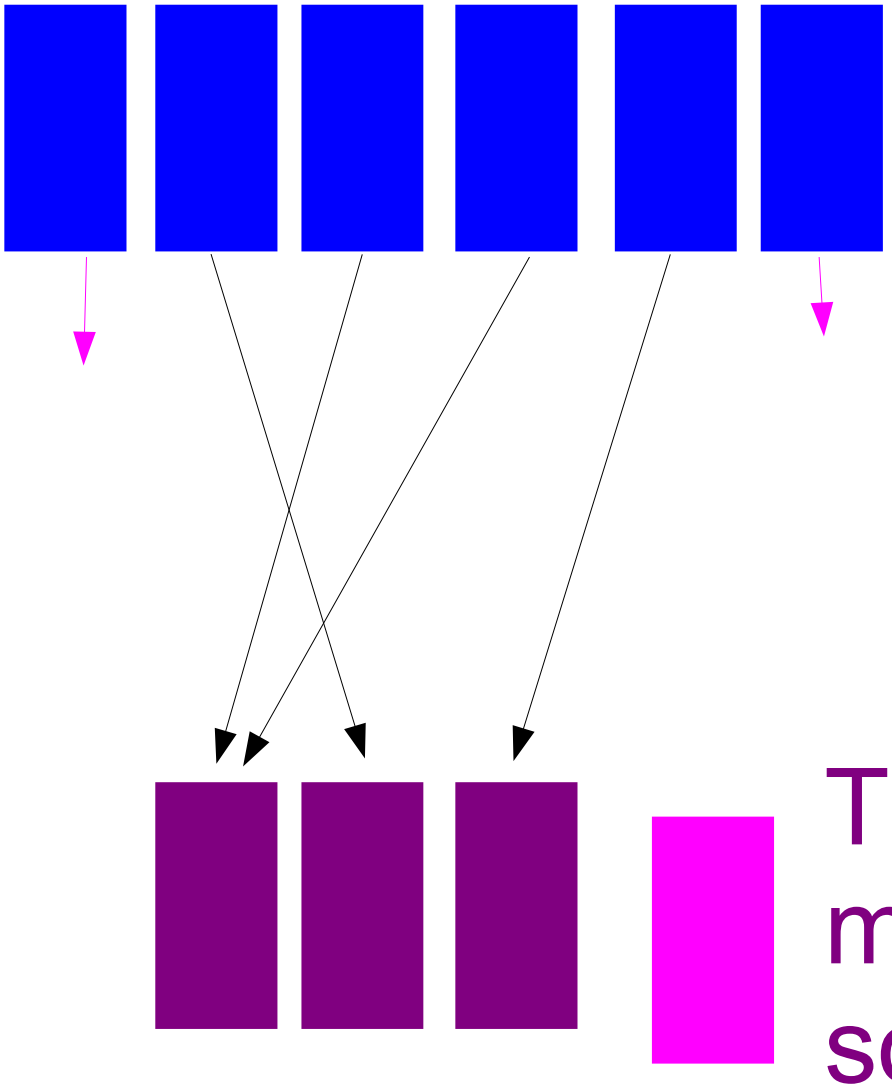
- Finding all pagetables (i.e. virtual addresses) that map to a certain physical page is not easy (but needed to swap)
- 2.4 in short scanned all pagetables O(N)
- 2.6 has an API called “rmap” that given a certain physical address allows the common code to reach all ptes that maps to it
- This rmap API is used by the paging methods to swap and/or unmap pages more efficiently than 2.4 did

VM basics (greatly simplified)



To swap to disk the last physical page we must first drop the two arrows (mark pte non-present)

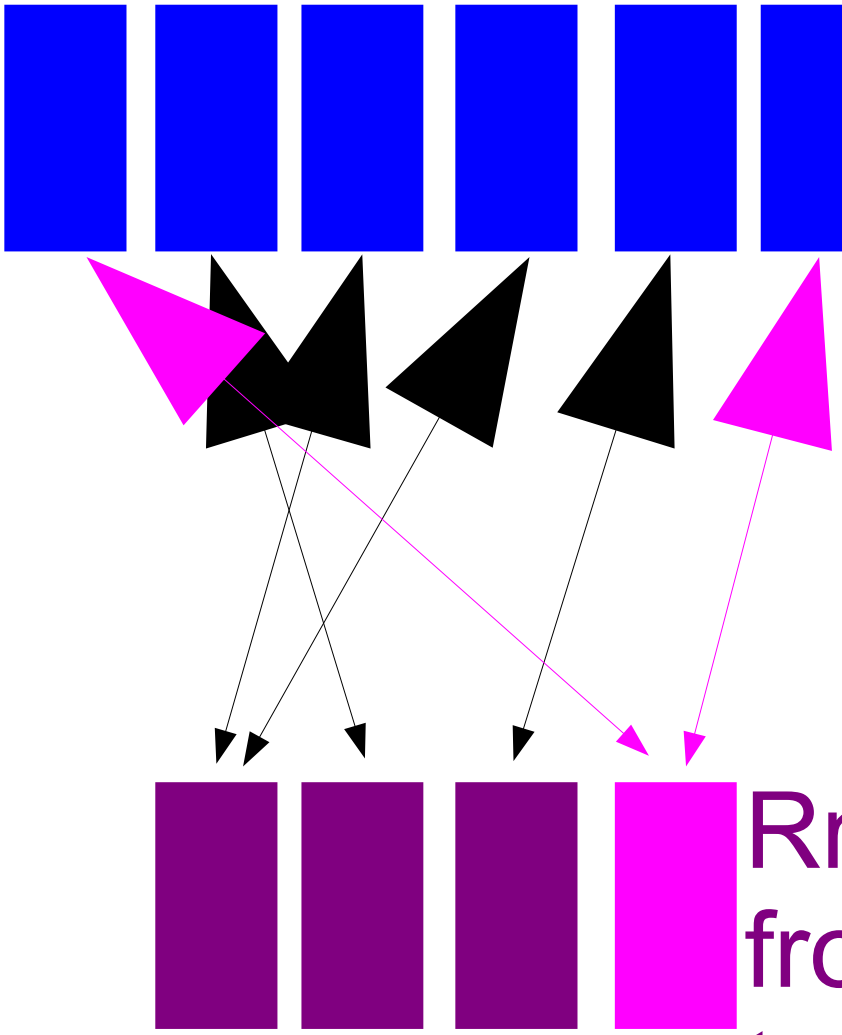
VM basics (greatly simplified)



Two pagetables have been set as non-present

The last page is not mapped anymore so we can swap it to disk now

rmap/objrmap VM



the big arrow is rmap
it allows us to drop the mapping fast by finding all virtual addresses (through the vmas)

Rmap API adds the arrow from phys to virtual too, so we can find the ptes faster than O(N)

rmap pros and cons

- rmap API clearly provides a benefit to the VM paging code by avoding to potentially scan all ptes to swap or drop a single page
- but keeping track of the back-arrow is expensive, and it has to happen in the cpu-bound fast paths
- swap is not a fast path and it's I/O bound
- the first 2.6 rmap design was exceptionally inefficient and it wasted lots of ram and CPU

rmap before 2.6.7

- Before 2.6.7, rmap pte-chains invalidated pte-highmem/highpte (rmap overhead is equal to the pte overhead, but pagetables go in highmem, rmap cannot)
- pte-highmem/highmem allowed to run some database workload on >4G boxes and the pte-chains broke that
- With old rmap code (with PAE) **8 bytes of lowmemory** were wasted **for each mapped page** (on a x86 there are only ~900M of lowmemory)

rmap before 2.6.7

- So with the old rmap design a workload with 1000 processes mapping 2G of shared memory each, would waste 1000*2*1024*1024*1024/4096*8/1024/1024/1024 = **3.9Gbytes** of lowmemory (and there are only ~900M of lowmemory available on a x86...)
- The performance of the page faults and of all other syscalls mangling the pagetables were also hurt by the pte-chains based rmap design

rmap since 2.6.7

- The solution was to replace the old rmap code based on the pte-chains with objrmap+anon_vma+prio_tree: that solved both the memory waste with SHM and the slowdown of the fast paths (like page faults and fork), but without sacrificing the paging scalability
- The first kernel out there brave enough to use objrmap+anon_vma has been the 2.6.5 SLES9 kernel and shortly later the new design has been merged in 2.6.7

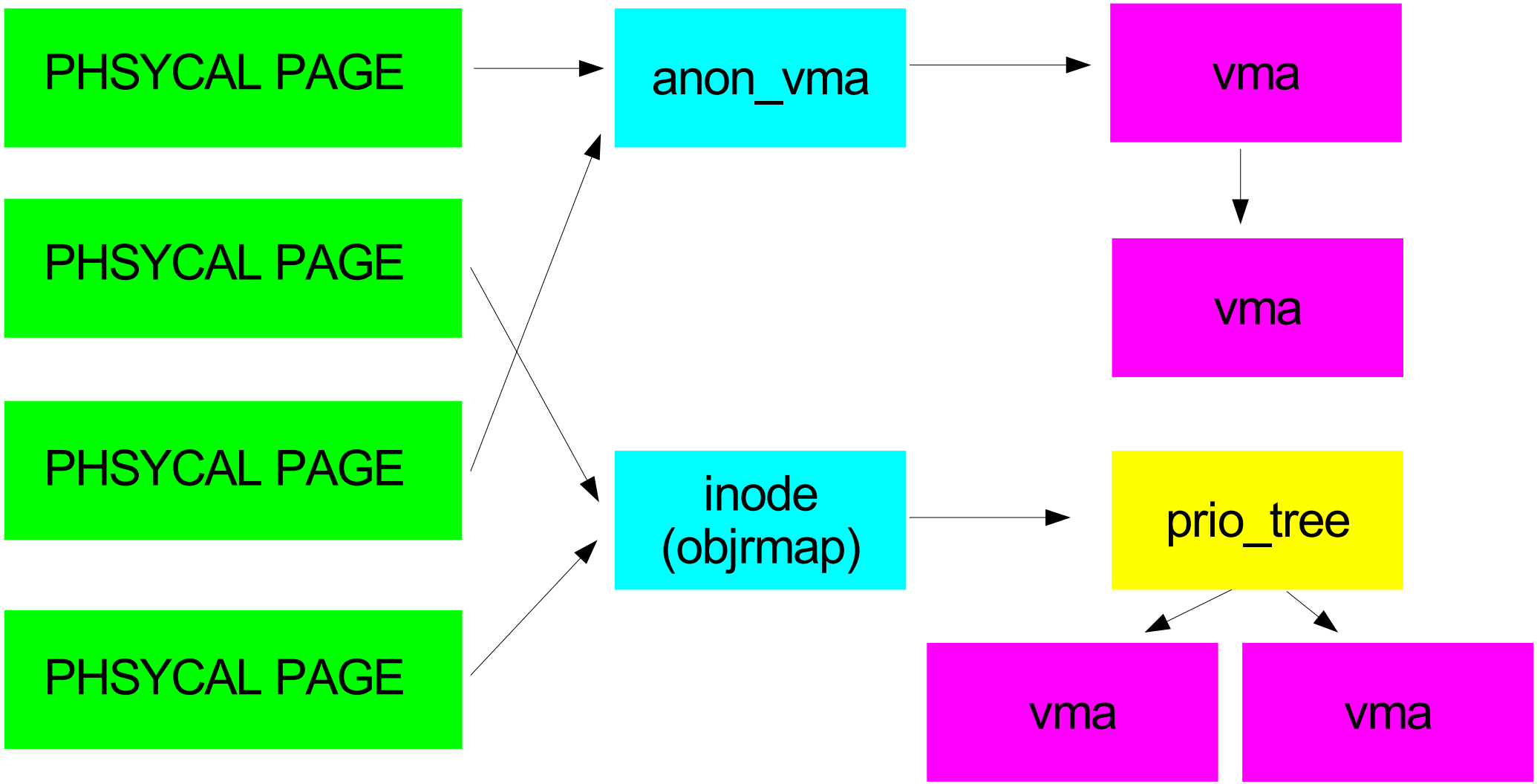
objrmap history

- objrmap has been researched by various developers (David Miller, Dave McCracken, Hugh Dickins), but it could never work well due the anonymous memory that still required rmap pte_chains or inefficient pagetable walking
- anon-vma solved the problem by creating a sort of filebacked address-space for the anonymous memory called “struct anon_vma”

objrmap history

- A single anon_vma object (8-bytes on UP 32bit) can reverse (back arrow) an unlimited amount of address space, while the old rmap pte_chains required a single object entry (8 bytes) for every single anonymous page being mapped
- anon_vma requires special locking and dynamic allocations only at the first page fault of a new vma (not at every page fault or pte modification like the old rmap code did)

objrmap/anon-vma visual



Reaching the relevant pagetables is a formality after we know all the “vmas”

Great collaboration on l-k

- Dave@IBM wrote objrmap
- I wrote anon-vma
- Rajesh@umich.edu wrote prio-tree
- Hugh@Veritas researched anonmm, audited the code and split and sorted it to make it easier for Andrew to merge it into mainline
- Andrew and Linus helped by checking and merging it
- Page to pagetable lookup problem is solved quite optimally now

VM tuning

- Luckily not an huge amount of tuning needed
- Most notable parameter is swappiness:
  - /proc/sys/vm/swappiness
  - default swappiness is 60
  - 100 max
  - 0 min
  - Higher means swap more
  - Lower means swap lessh

VM tuning

- Really be careful with lower swappiness value (I got reports of deadlocks with low swappiness values, that's a VM bug and should be fixed...)
- dirty_ratio
  - max percentage of memory dirty in the system (this doesn't account MAP_SHARED)
- min_free_kbytes
  - control the GFP_ATOMIC levels

VM tuning

- /proc/sys/vm/block_dump
  - debug why disk spin-up
- /proc/sys/vm/dirty_expire_centisecs
  - how long cache should be dirty
- dirty_writeback_centisecs
  - how frequently we check the dirty_expire_centisecs levels
- dirty_ratio
  - max percentage of memory dirty in the system (this doesn't account MAP_SHARED)

VM tuning

- min_free_kbytes
  - control the GFP_ATOMIC levels
- overcommit_memory
  - 0 default: non strict check, multiple mmaps will succeed even if the sum of the address space allocated largely exceedes swap + free + cache + buffers
  - 1: mmap always succeeds, overcommit fully enabled
  - 2: strict overcommit enabled

VM tuning

- /proc/sys/vm/overcommit_ratio
  - for mode 1: max percentage of memory committed
  - See /proc/meminfo:
  - grep Committed /proc/meminfo
  - Committed_AS: 243252 kB
  - This is the total amount of memory needed by userland to run including all mmaps and user stack
  - Still no guarantee of graceful -ENOMEM due to stack growsdown

/proc/<pid>/seccomp

- Secure computing mode allows running untrusted code as normal user
- syscalls allowed:
  - ***read***
  - ***write***
  - ***exit***
  - ***sigreturn***
- Supported archs by 2.6.12 and later:
  - x86_64
  - x86
  - ppc64

/proc/<pid>/seccomp applications

- Truly secure and fast grid computing
- No need of slow and memory hungry interpreters (or just in time compilers) to keep the cluster secure (i.e. no JVM waste), and no need of virtualization
- Safer and faster because much simpler and enforced by the kernel
- simd/sse2/ss3/altivec is allowed
- Secure decompression (bzip2, gzip, ogg, mp3, mpeg, mov etc..)
- jpeg decompression in web browsers

/proc/<pid>/seccomp API

- If “**echo 1 >/proc/self/seccomp**” returns ***no*** errors, it means it worked and you can depend on it
- Shall there be a security issue in mode = 1, the above write will fail and it will return an error, and the userland code should fallback to the normal decompression scheme for backwards compatibility on older kernels
- A userland lib could hide the kernel details and provide a friendly API

/proc/<pid>/seccomp API

- Given a proper intialization from a seccomp-loader, any untrusted or malicious bytecode can be later safely run inside the seccomp jail
- TSC is automatically disabled to prevent theoretical covert channels
- There is very little code involved with seccomp, so it's most secure approach
- All virtualization solutions that attempt to achieve similar objectives are order of magnitude more complicated

/proc/<pid>/seccomp API

- seccomp programming model is not friendly, malloc must be simulated etc...
- Good only for computing
- It could be adapted to the CELL SPU model which also has very limited capabilities
- I believe if people can write code for the CELL SPU they can as easily write code inside SECCOMP
- In the long term seccomp can run on top of virtualization (seccomp-hypervisor)