



How to Deal with Lock-Holder Preemption

Thomas Friebe

June 2008

Spinlock Basics



Spinlocks wait actively as opposed to sleeping locks

Used for short critical sections



Month ##, 200#

How to Deal with Lock-Holder Preemption

Spinlocks are a common kernel synchronization primitive.

Spinlocks wait actively: Use CPU to wait.

Sleeping locks sleep: Thread is blocked when lock is found in locked state; thread is woken up when lock is available again.

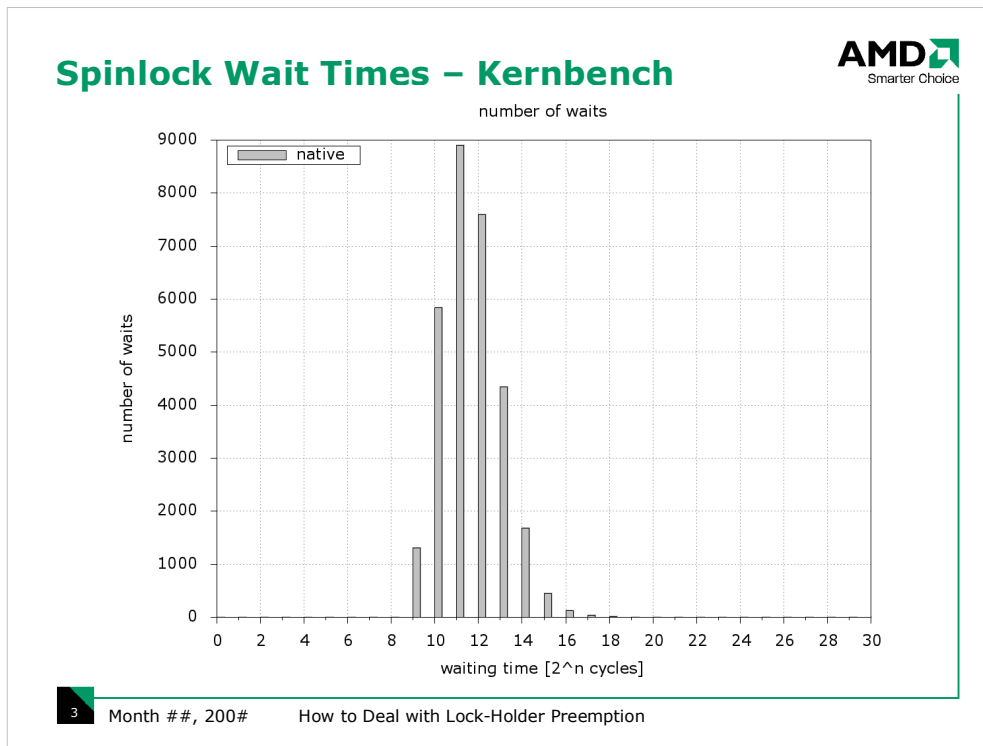
Why use spinlocks at all?

Sleeping locks: Going to sleep and waking up has a cost.

If critical section short sleep/wake cost is relatively high, so instead use spinlocks.

In certain kernel parts, eg interrupt handler, sleeping not allowed; use spinlocks.

What is a 'short' critical section for Linux developers?



Kernbench run natively. All experiments done on the following machine:

- 16 cores: 4 sockets equipped with quad-cores
- 2.2 GHz
- 8 GB RAM, benchmark data in a RAM disk (Linux tmpfs)

Histogram shows distribution for wait time for spinlocks:

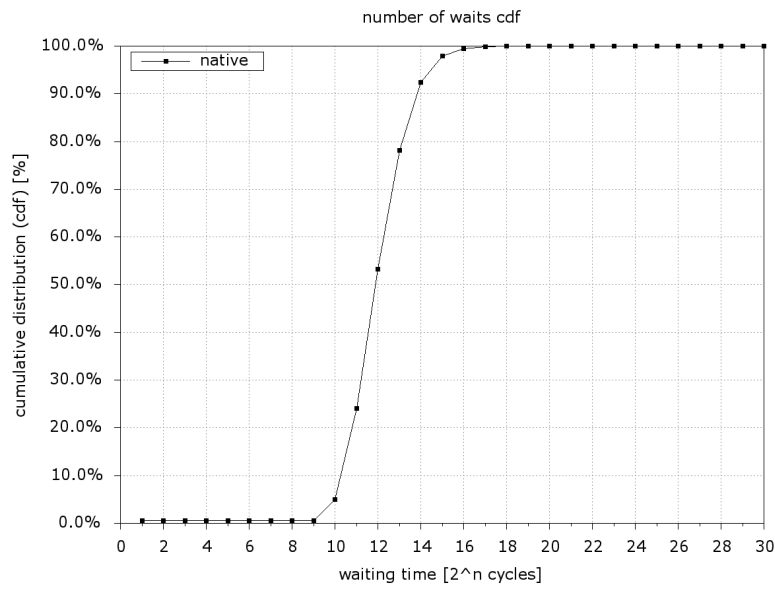
X axis: waiting time in 2^n cycles (logarithmic scale)

Y axis: number of lock acquisitions after waiting between 2^n and $2^{(n+1)}$ cycles

Results:

- Waiting time for spinlocks short (less than 30 microseconds)
- Most lock waits around 2^{12} cycles

Spinlock Wait Times – Kernbench



4 Month ##, 200# How to Deal with Lock-Holder Preemption

Same data, cumulative distribution function:

After waiting 2^{16} cycles almost always lock acquired (97.8%). So 2^{16} cycles or less is 'short'.

Spinlocks and Virtualization

5

Month ##, 200#

How to Deal with Lock-Holder Preemption

What is different when running in a virtualized environment?

Spinlocks and Virtualization



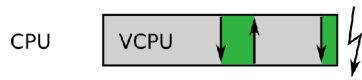
VCPU running on physical CPU

down: lock acquisition

green: critical section

up: lock release

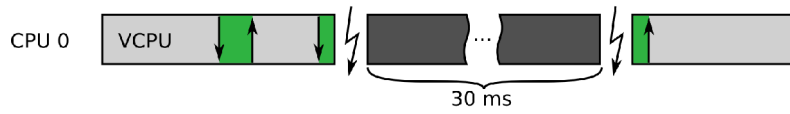
Spinlocks and Virtualization



Even if critical sections are short: Preemption (eg. end of VCPU time slice) inside a critical section is possible.

→ This is lock-holder preemption (LHP)

Spinlocks and Virtualization

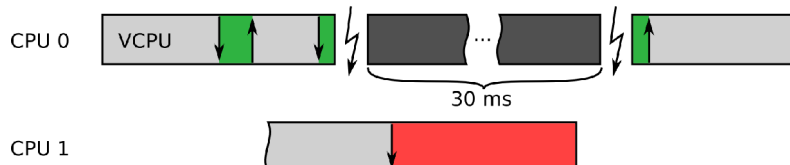


A different VCPU (maybe even from another guest) gets scheduled; This VCPU runs until it blocks or its time slice ends. Currently credit based scheduler time slice is 30 msecs.

When the first VCPU is scheduled again it ends its critical section and releases the lock.

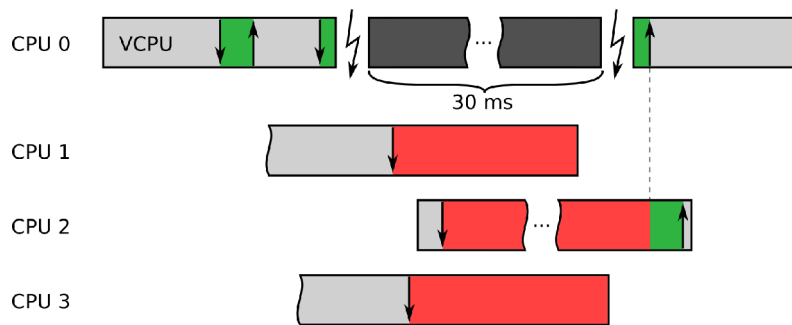
→Lock held for tens of milliseconds instead of tens of microseconds.

Spinlocks and Virtualization



A VCPU trying to acquire the same lock will have to wait much longer than in the native case... again tens of msecs instead of tens of usecs. It might already be descheduled at the time the lock is released.

Spinlocks and Virtualization



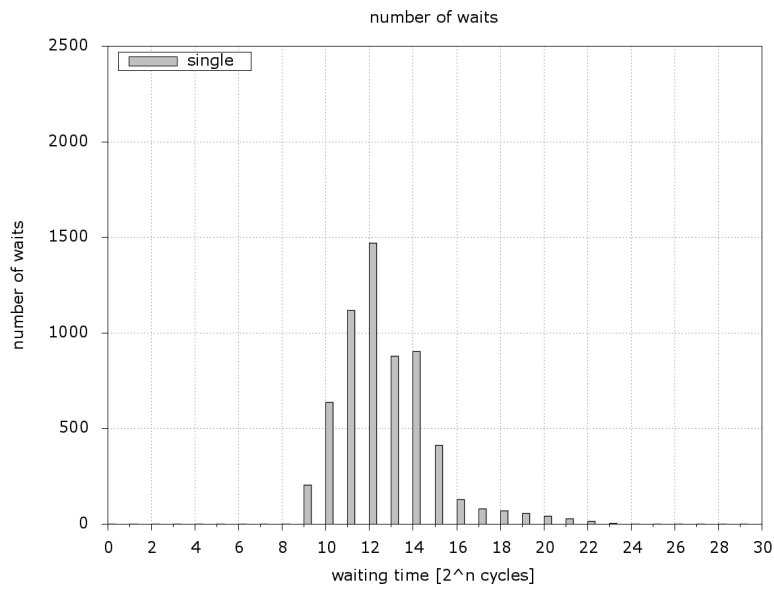
This can happen to a potentially high number of VCPUs, wasting time spinning, waiting for a lock which can not be released by the owning VCPU.

The more VCPUs the more VCPUs which can be preempted.
The more CPUs the more VCPUs which can run concurrently and spin.
But: Critical sections are short; preemption inside a critical section is unlikely.

So, is the problem already serious with current 16core 4p machines?

Is lock-holder preemption problematic?

Kernbench in a Guest



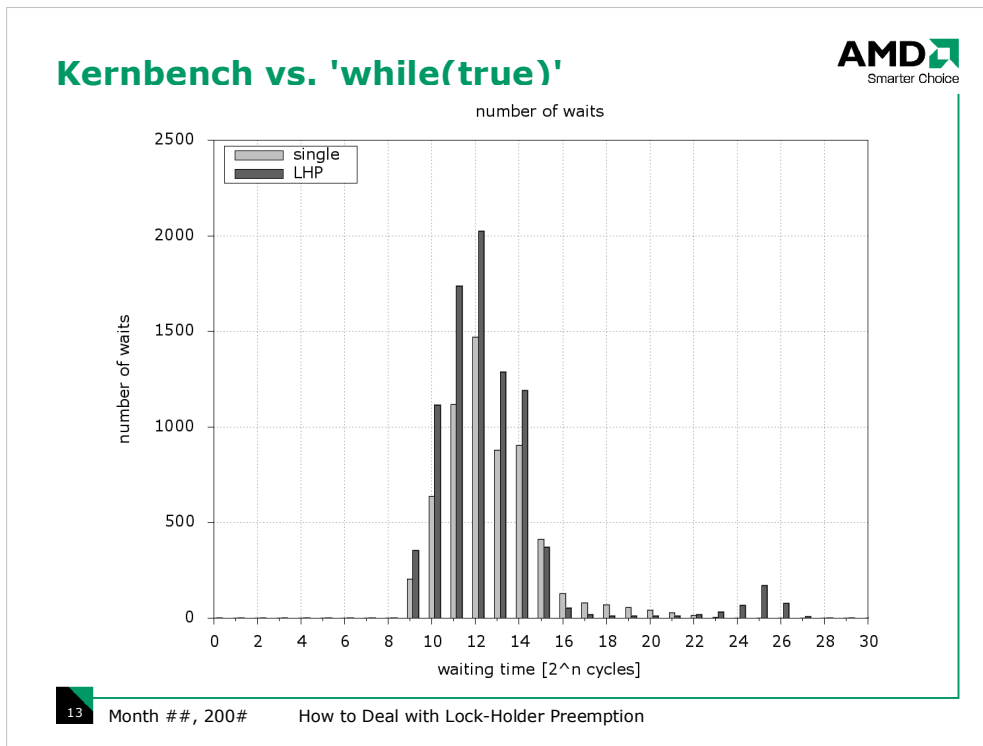
12

Month ##, 200#

How to Deal with Lock-Holder Preemption

Redo kernbench in a VM. One single guest, doing kernbench:

Behaviour about the same as native, almost no waits longer than 2^{16} cycles. Histogram's center at about 2^{12} .



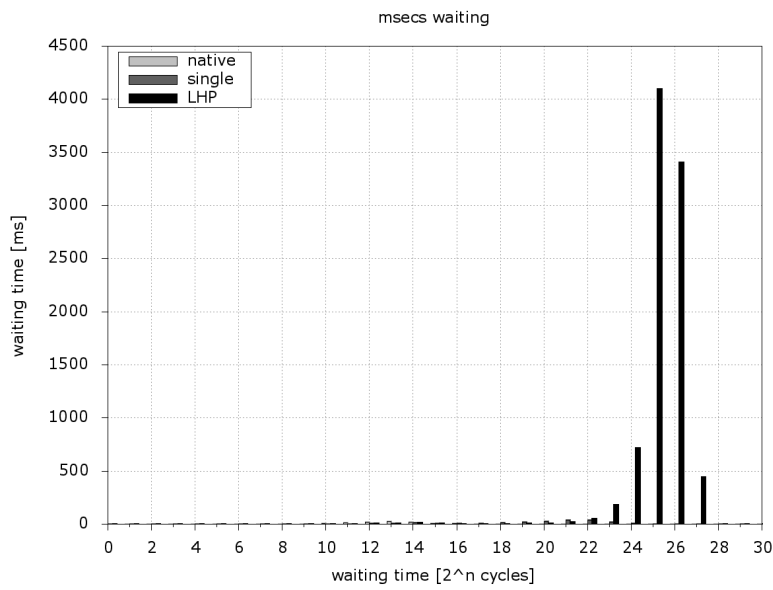
Adding another guest, executing while(true) on each of its VCPUs. This scenario is called 'LHP' in the histogram.

Histogram almost identical to 'single' case. Exception: A second group of about 400 waits at about 2^{25} cycles.

2^{26} cycles is about 30 msecs – the VMM time slice. This is caused by lock-holder preemption!

What is the effect of this small number (400) of long waits?

Time, not Times



14

Month ##, 200#

How to Deal with Lock-Holder Preemption

Same data. Y axis now time in msec.

Seconds spent spinning due to lock-holder preemption. The normal behaviour at 2^8 to 2^{16} insignificant, not even visible here.

Let's have a look at the kernbench performance.

And in Numbers?



	guest time [s]	time spent spinning [s]	[%]
single kernbench	109.0	0.2	0.2%
kernbench vs while(1)	117.3	9.0	7.6%
difference		7.6%	

15 Month ##, 200# How to Deal with Lock-Holder Preemption

Single kernbench: no lock-holder preemption (each VCPU has a CPU)

kb vs while(1): lock-holder preemption (2 VCPUs share one CPU)

LHP scenario: about 9 secs more guest time, about 9 secs spent spinning

7.6% guest time wasted due to lock-holder preemption!

What can we do about it?

Dealing with lock-holder preemption



LHP avoidance

- No spinlock held in userspace
- Idea: Avoid preempting guest in kernel space
- Postpone guest switch to kernel exit
- Problem: extraordinary long critical sections, e.g. Apache using sendfile()

Helping locks

- Instead of busy waiting, switch to preempted lock-holder
- Problem: finding the preempted lock-holder



Two approaches; I chose the helping locks approach.

Helping locks: Ingredients



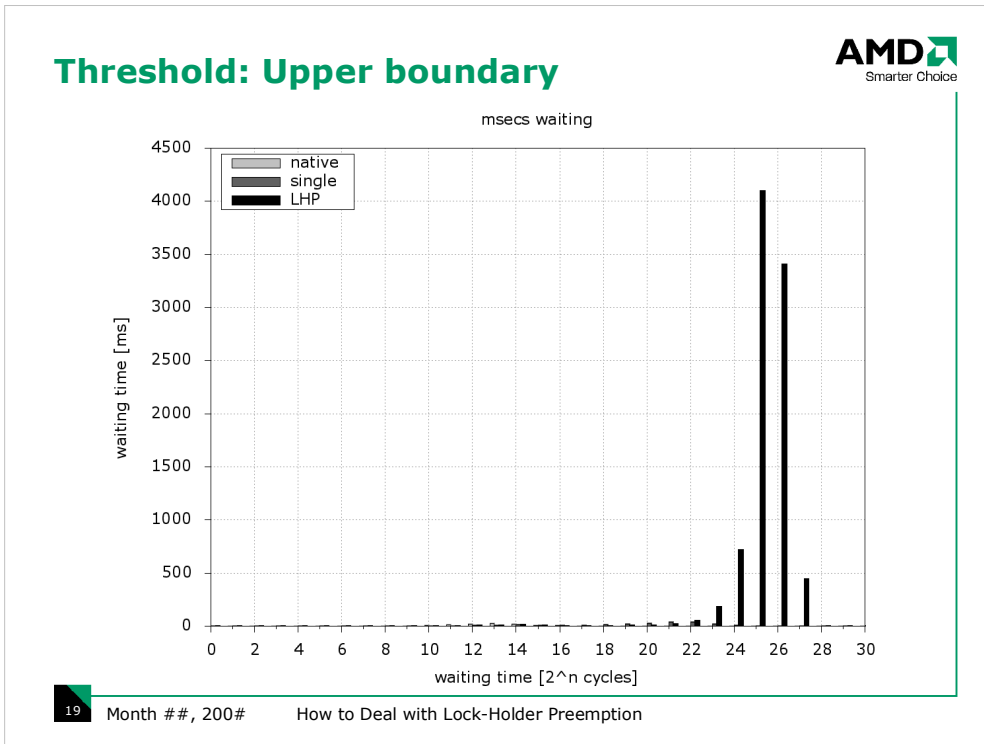
- 1) Guest kernel: new 'yield' hypercall when waiting unusually long
 - Modify spinlock loop
- 2) Reasonable threshold for 'unusually long'
 - Histograms help
- 3) Selecting which VCPU to switch to



Month ##, 200#

How to Deal with Lock-Holder Preemption

- 1: Detect 'long' waits
- 2: Detection threshold: What is 'long'?
- 3: What to do when a 'stuck' VCPU is detected?

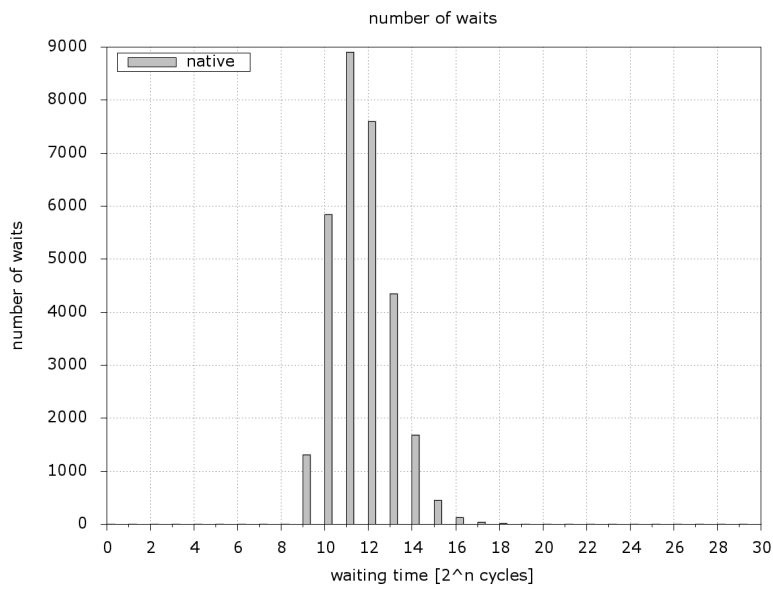


Searching for an upper boundary for the yield threshold.

Time we want to save: waits longer than 2^{24} cycles. But the earlier we interrupt spinning the better anyway.

But: Spinlocks are used for short critical sections, when sleeping is too costly. Don't set the threshold too low.

Threshold: Lower boundary



20

Month ##, 200#

How to Deal with Lock-Holder Preemption

Searching for a lower boundary:

The native kernbench histogram again: normal waits no longer than 2^{16} cycles. We do not want to interfere with normal behaviour.

Choose 2^{16} as threshold; yield to VMM after waiting (spinning) 2^{16} cycles.

Scheduling Strategy



Good choices:

- VCPUs of the same VM to make progress locally
- (Potential) preempted lock-holders
- Cache-„near“ VCPUs

Neither/nor:

- VCPUs in user space

Bad choices:

- VCPUs which yielded recently



Month ##, 200#

How to Deal with Lock-Holder Preemption

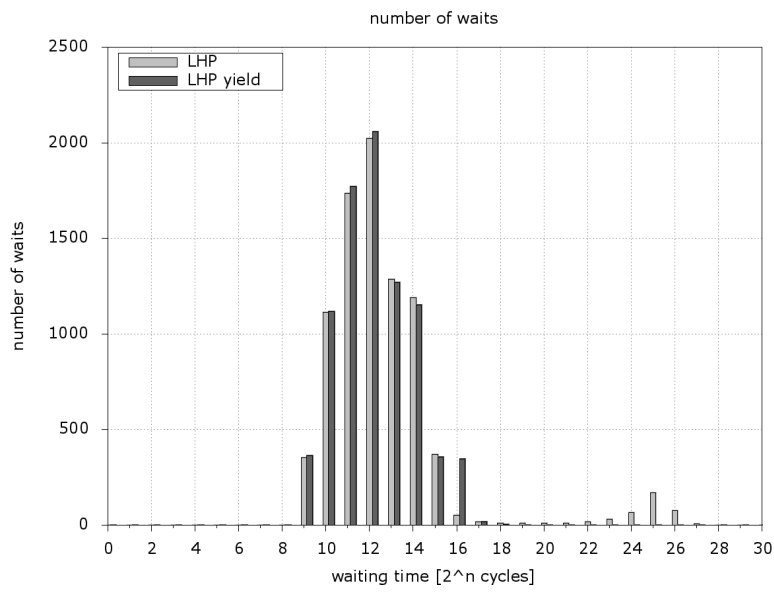
What to schedule next when VMM receives 'waiting unusually long' aka 'yield' hypercall?

My algorithm:

- search VCPU list of current domain. Ignore blocked, running, and 'yielders'.
- Schedule in-kernel VCPU if found; prefer cache-near over cache-remote (A VCPU is considered cache-near iff it last ran on a CPU sharing the L3 cache with the current CPU)
- Otherwise in-userland VCPU (cache-near before -remote, too)
- Otherwise try oldest yielder – maybe its lock is released since it yielded

What about performance?

Histogram with 'yield' hypercall



23

Month ##, 200#

How to Deal with Lock-Holder Preemption

With new 'yield' hypercall no more waits around 2^{25} , but more at 2^{16} – the yield threshold. Yielding works. And the performance impact?

Performance



	wall clock [s]	guest time [s]	time spent spinning [s]	spinning [%]
LHP	34.8	117.3	9.0	7.6%
yield	33.5	108.4	0.0	0.0%
difference	-3.9%	-7.6%		-7.6%

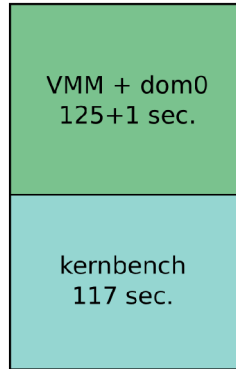
(Almost) no time spent spinning in yield scenario.
7.6% time was used for spinning, now 7.6% less CPU time for guest!
About 4% wall clock time saved -> 4% performance improvement.

...but, why not 7.6% wall clock saved? See next slides.

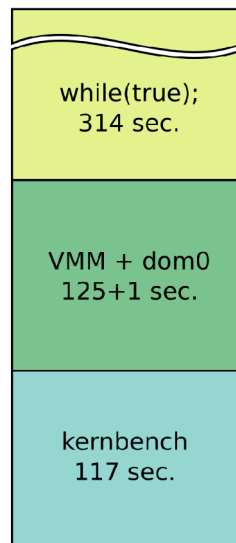
kernbench
117 sec.

What is CPU time actually used for? (numbers from the LHP scenario)

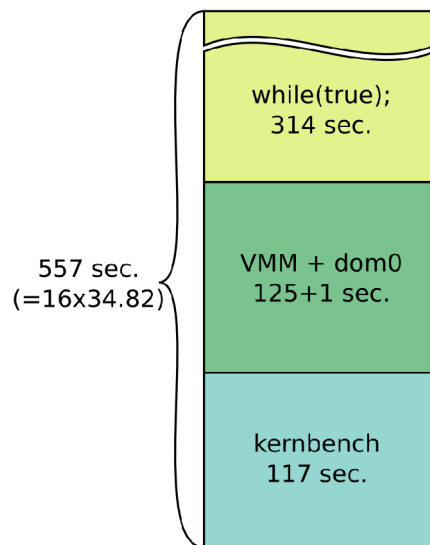
Efficiency



Efficiency

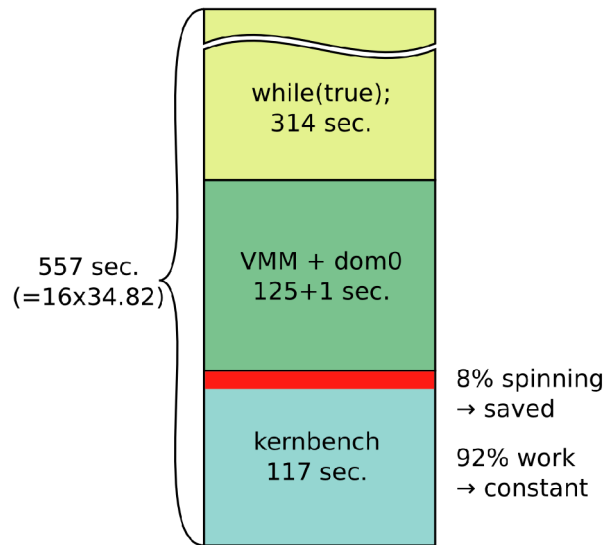


Efficiency



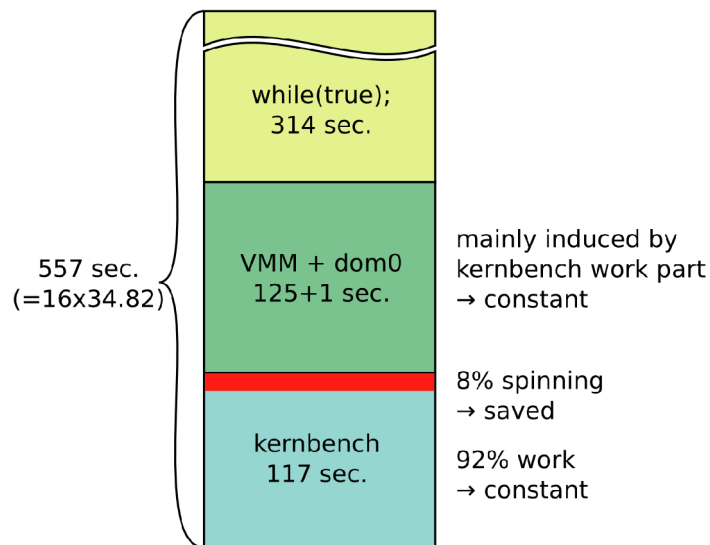
Both domains plus hypervisor sum up to 557 secs, which is 16 cores times 34.82 seconds wall clock time. -> no time spent idling (naturally)

Efficiency



Kernbench block: we can save about 8%

Efficiency



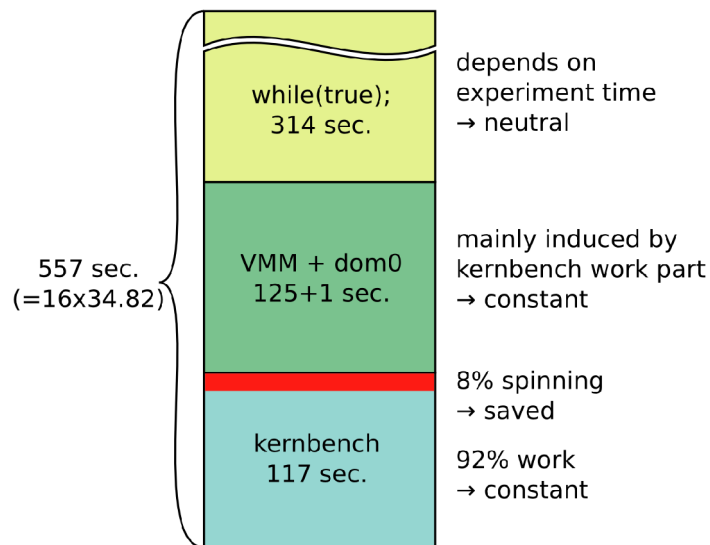
What does VMM spend time for?

Scenario has nested paging off.

Kernbench creates lots of processes. -> Much shadow paging work in VMM, depending on the work part of kernbench. The spinning part does not exercise shadow paging. The while(true) domain neither.

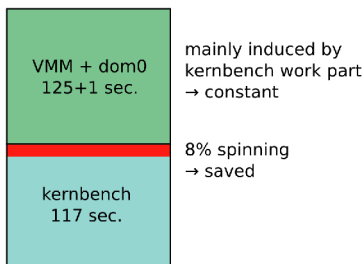
→ So no time saving in the VMM block.

Efficiency



Kernbench (and the VMM work for kernbench) determine duration of experiment. The while(true) domain has no influence. So while(true) domain is performance neutral.

Efficiency



$$\frac{117 \text{ sec}}{117 \text{ sec} + 126 \text{ sec}} \times 7.6\% = 3.7\%$$

- Real result of 3.9% is reasonable
- Highly efficient

Kernbench and VMM part almost equal. About 8% saving here, no saving there.

Principle: Amdahl's law.

Result: about 4% wall clock time saving. -> we have about 4%!

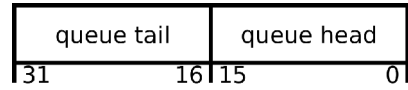
FIFO ticket spinlocks

Linux 2.6.25 introduced 'FIFO ticket spinlocks' for x86, replacing traditional spinlocks.

FIFO ticket spinlocks

Next ticket in dispenser: queue tail

„Now serving“ display at counter: queue head



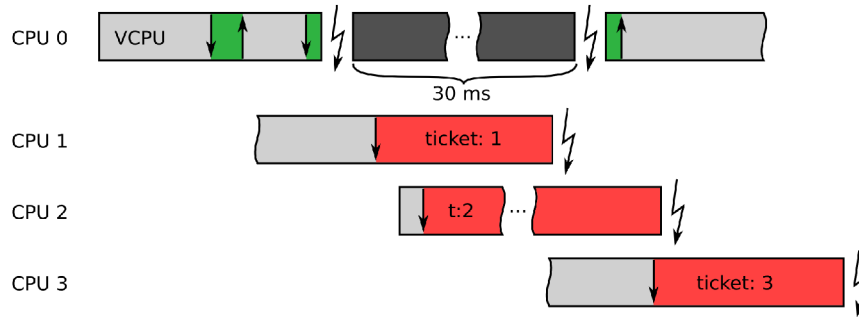
Lock: `atomic(ticket = tail++); while (head != ticket);`

Unlock: `atomic(head++);`



Ticket spinlocks work like shop with ticket dispenser and „now serving“ display to ensure fair strict FIFO order.

FIFO ticket spinlocks



Waiting VCPUs are strictly serialized. When lock is released only the VCPU holding the next ticket can acquire it.

What are the performance implications?

Ticket locks and virtualization



	wall clock [s]	guest time [s]	time spent spinning [s]	spinning [%]
LHP	2825.1	22434.2	22270.4	99.3%

The 33 seconds experiment now takes about 45 minutes!!!

What if we apply the yield hypercall again?

Ticket locks and virtualization



	wall clock [s]	guest time [s]	time spent spinning [s]	spinning [%]
LHP	2825.1	22434.2	22270.4	99.3%
yield	34.1	123.6	6.6	5.4%

With yielding back to the 30-40 seconds region. Problem solved.

6.6secs spent scheduling the yielders round-robin, trying to find the next ticket holder.

Conclusion



Lock-holder preemption quite serious:
7.6% guest time wasted

Helping locks:
3.9% system performance improvement!
(Amdahl's law explains why)

New ticket spinlocks:
30 secs kernbench takes 45 minutes

Helping locks help here, too



Trademark Attribution

AMD, the AMD Arrow logo and combinations thereof are trademarks of Advanced Micro Devices, Inc. in the United States and/or other jurisdictions. Other names used in this presentation are for identification purposes only and may be trademarks of their respective owners.

© 2008 Advanced Micro Devices, Inc. All rights reserved.