MonitorU: vSMP

Ole Agesen
August 17, 2010

What, how, why?

- What is it?
- How is it done?
- Why is it this way?

Goals and non-goals

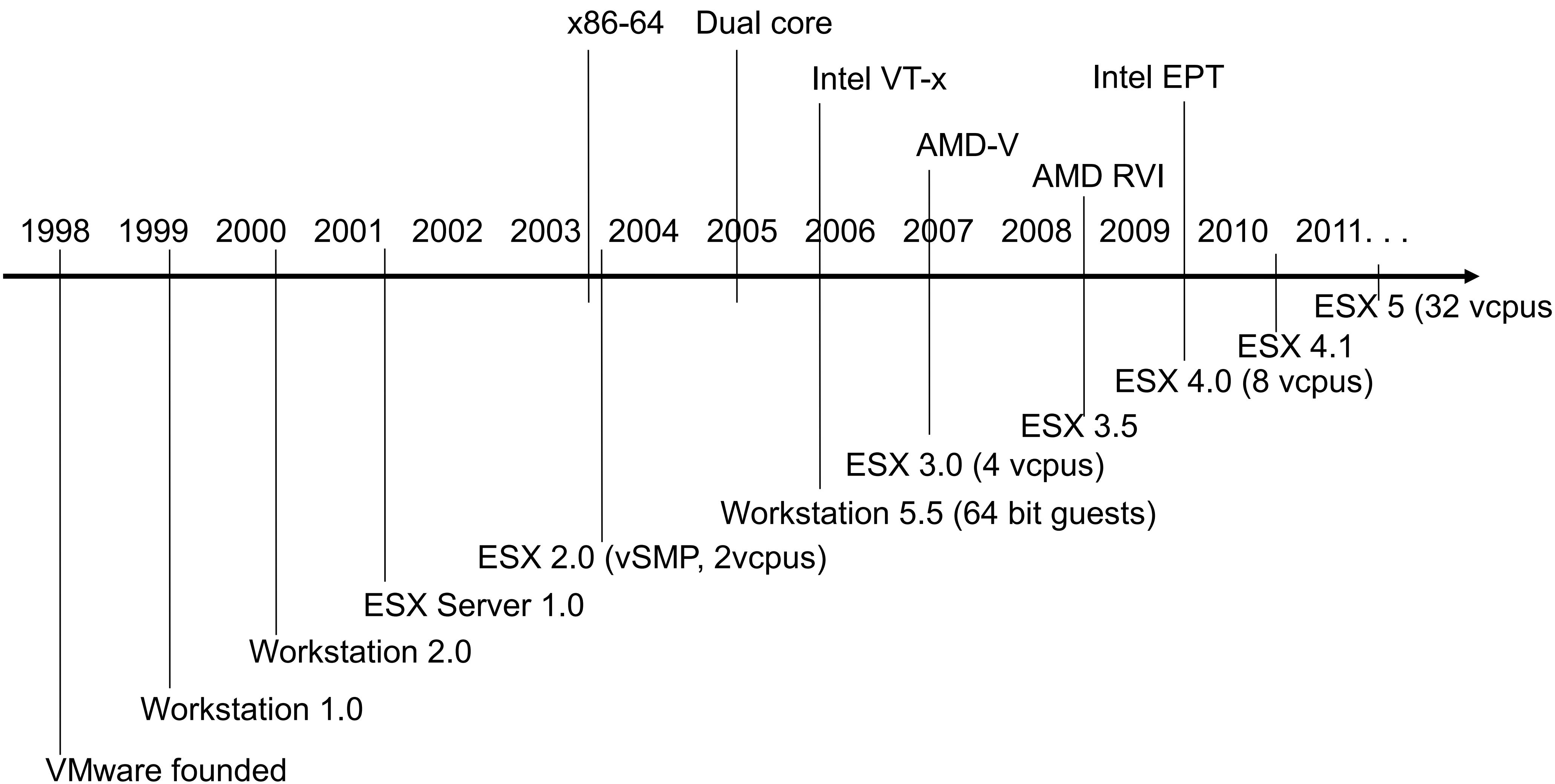
- Goals are to explain
  - the vmm/vmx environment
  - the mechanisms we have built
  - the constraints that apply
- It is not a goal to explain
  - how x86 SMP systems work
  - cache coherency protocols
  - memory models
  - what an IPI or an APIC is
  - how cross-cpu code (or GDT/pg-table) modification “acts”

Outline

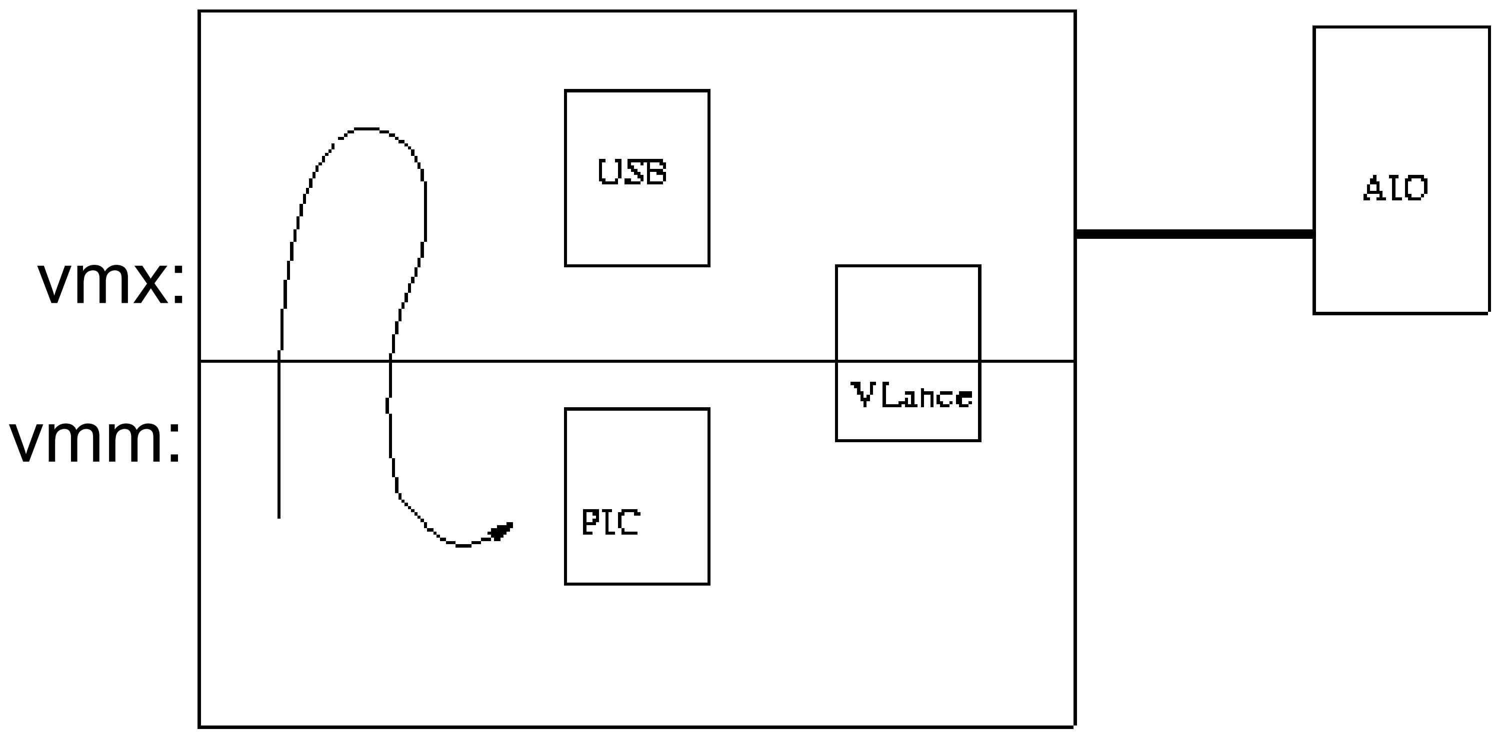
- Historical context
- State and threading model
- Synchronization
- Communication
- Q&A

Highly related/overlapping concepts

History

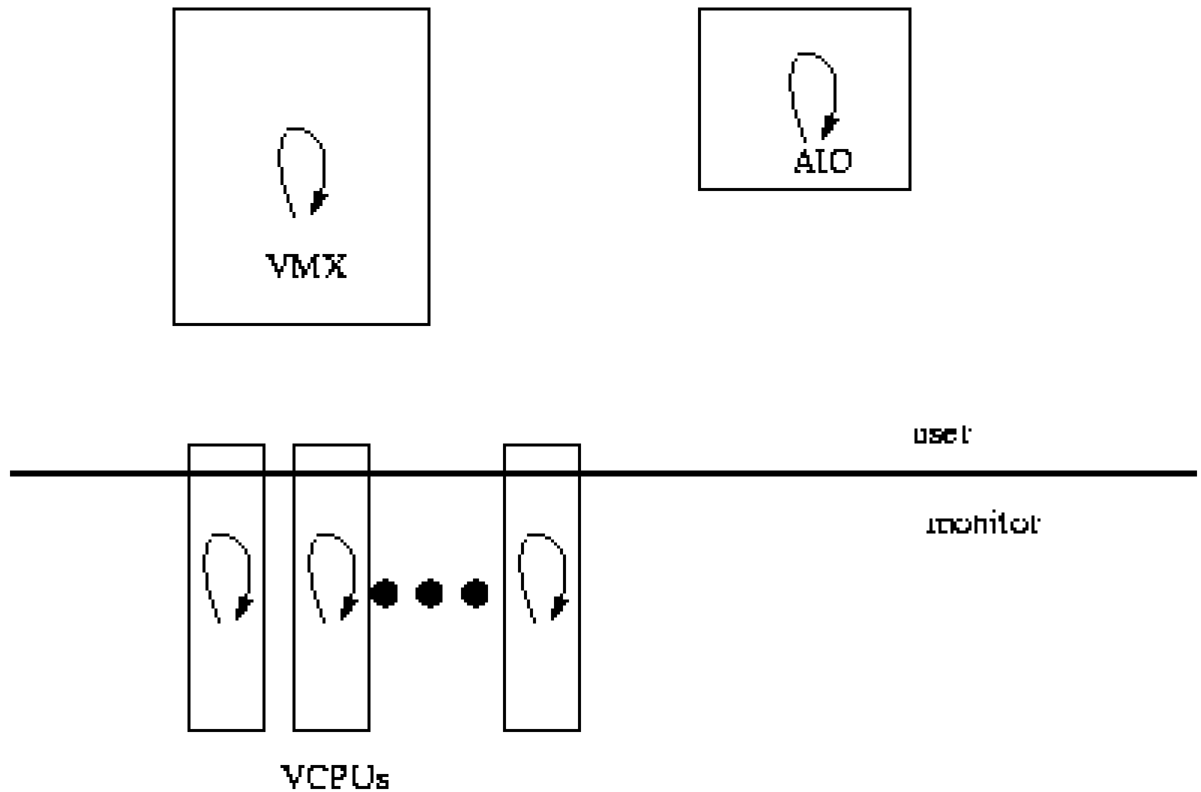

x86-64
Dual core
Intel VT-x
Intel EPT
AMD-V
AMD RVI
1998 1999 2000 2001 2002 2003 2004 2005 2006 2007 2008 2009 2010 2011. . .
ESX 5 (32 vcpus)
ESX 4.1
ESX 4.0 (8 vcpus)
ESX 3.5
ESX 3.0 (4 vcpus)
Workstation 5.5 (64 bit guests)
ESX 2.0 (vSMP, 2vcpus)
ESX Server 1.0
Workstation 2.0
Workstation 1.0
VMware founded

Threading model, pre-vSMP



- One vmx/vcpu thread

Now: separate vmx/vcpu threads



- vmx/vcpu concurrency (UP and vSMP)
- vcpu/vcpu concurrency (vSMP only)

VMM perspective

- One vmm/monitor per VM
- One thread per vcpu
  - No other threads (today)
  - But have discussed introduction of “helper threads”
- Concurrent programming model
  - Some state is shared: busmem, gphys, ...
  - Some state is per-vcpu: MMU, TC, DT, ...

VMX perspective

- Single vmx thread
  - Assistant for vcpus and devices
  - Polls for work (timers, IO completions, etc)
- vcpu threads can “visit” through user-RPC
- Limited concurrency
  - Big User Level Lock (BULL)
- Complex devices have their own threads
  - mks, sound

State in the VMM

- Global variables (including “static”)
  - One instance per-vcpu (“thread local”)
  - No access to other vcpus' variables
- SHARED_PER_VMM:
  - One shared instance
  - Visible to all vcpus, vmx, vmkernel
- SHARED_INTER_VCPU:
  - One instance per vcpu
  - Visible to other vcpus (using “array-like” accessor)

Synchronization

- pthreads: mutexes, condvars, barriers, semaphores
- “Stop”
- Atomics

Synchronization primitives (“pthreads-like”)

- Mutexes (can self-deadlock)
  - **`MX_Lock()`**, **`MX_Unlock()`**, **`MX_TryLock()`**
- Recursive mutexes (owner can reacquire)
  - **`MX_LockRec()`**, **`MX_UnlockRec()`**, **`MX_IsLockedBy()`**
- Condition variables (rarely used)
  - **`MX_Signal()`**, **`MX_Broadcast()`**, **`MX_Wait()`**
- Barriers
  - **`MX_EnterBarrier()`**
- Semaphores (counting and binary)

Locks (“MX”)

- Locks guard “critical sections”

```
MX_Lock(&myLock);
...critical section...
MX_Unlock(&myLock);
```

- Locks protect shared state
  - Hold lock when reading/modifying state
  - No concurrency/interleaving reasoning
  - No memory model dependency
  - No compiler optimizer dependency

Properties of “MX” locks

- Blocking (with bounded spin)
- Uncontested locking *very* fast (tens of cycles)
- Work across vmm/vmx domains
- But:
  - vmkernel has its own locks
    - Cannot use vmkernel locks in monitor
    - Cannot use vmm/vmx locks in vmkernel
  - User-level locks
    - Cannot use in vmm/vmkernel

Deadlocks

- Large systems have many locks (dozens for vmm)
- Lock ordering consistency hard to ensure

```
lock(a)
...
lock(b)
...
```

Thread 1

```
lock(b)
...
lock(a)
...
```

Thread 2

Deadlock!!!!

Deadlock prevention: lock ranks

- All locks have a rank
- Threads can acquire locks in ascending rank *only*
- Implication: no deadlocks
  - Either “lock(a); lock(b)” or
    “lock(b); lock(a)” produces rank violation
  - or both do if “rank(a) == rank(b)”
- Ranks are checked in debug- builds only
  - Implementation: per-thread linked list of held locks

How do ranks help?

- Without ranks
  - to find all deadlocks: must exercise all interleavings
  - thread-interaction problem, timing-sensitive
- With ranks
  - to find all potential deadlocks: exercise all code paths (regardless of interleavings)
  - thread-local problem, timing-independent
  - Dramatic reduction in testing burden
  - But still a runtime property
- Rank alternatives:
  - Be clever/careful – just too hard
  - Static checkers: not really up to the job yet

Ranks constrain code

- **`mutexRank.h`** defines all ranks
- Ranks establish partial order on locks
- Every so often we may have to rework ranks to make room for new lock
  - It can be very difficult

Rank constraits (mutexRank.h)

```
*       barrier              == 1
*       stopLock             < stopCrossCall
*       stopCrossCall        < RANK_stopResponderSema
*       crossUserCall        < any lock or semaphore acquired from userlevel
*       afterTraces          < beforeTraceInstall
*       afterTraces          < memspaceDevice
*       beforeTraceInstall   < memspaceDevice
*       beforeTraceInstall   < busMemLock
*       memspaceDevice       < afterTraceInstall
*       tcCohLock            < afterTraceInstall
*       memspaceDevice       < busMemLock
*       afterTraceInstall    < beforeTraces
*       afterTraceInstall    < busMemLock
*       beforeTrace          < busMemLock
*       busmem               < busmemCB
*       busmemCB             < userlevelLock
*       busmemCB             < memrefDevice
*       ioSpaceLock          < intrLock
*       splitDevice          < intrLock
*       partialTrace         < busMemLock
*       traceAlotLock        < busMemLock
*       crossCallLock        < busMemLock
*       traceAlotLock        < userlevelLock
*       traceTableLock       < userlevelLock
*       timer                < intrLock
*       timer                < timeTracker
*       asyncUserCall        <= anything userlevel can acquire
*       mksLock              > anything non-leaf the VMX holds
*       busMemLock           < mmuInfoLock
*       mmuInfoLock          < monitor leaf rank
```

Actual ranks

```
#define RANK_MONITOR_LEAF        (RANK_userlevelLock - 1)

#define RANK_barrier             1
#define RANK_stopLock            1
#define RANK_stopCrossCall       3
#define RANK_stopResponderSema   4
#define RANK_crossUserCall       4
#define RANK_executeCB           5

#define RANK_tcTraces            10
#define RANK_beforeTraceInstall  20
#define RANK_memspaceDevice      30 /* needs traceInstall+busMem */
#define RANK_tcCohLock           (RANK_afterTraceInstall - 1)
#define RANK_mwaitLock           (RANK_afterTraceInstall - 1)
#define RANK_vnptLock            (RANK_afterTraceInstall - 1)
#define RANK_afterTraceInstall   40
#define RANK_traceALotLock       50
#define RANK_memrefDevice        60
#define RANK_beforeTraces        70
#define RANK_busMemLock          80
#define RANK_busmemCB            90
/* def'n of RANK_userlevelLock   100  in mutex.h */
#define RANK_device              110
#define RANK_iospaceLock         120
#define RANK_mksLock             130 /* above BULL, dev & iospace */

#define RANK_mmuInfoLock         (RANK_MONITOR_LEAF - 1)
#define RANK_simpleMalloc        RANK_MONITOR_LEAF
```

“Stop-the-world”

- Used by language runtimes (garbage collectors stop all mutators)
- Stop protocol
  - One vcpu initiates “stop” operation
  - Other vcpus stop at their next “safe point”:
    - Base rank
    - At guest instruction boundary
  - “Stop master” can now request work by other vcpus
  - Finally release stopped vcpus
- “Stop” turns competitive environment into cooperative one
  - Allows for very fast fast-paths (but very slow slow-paths)
  - Properties that only change under “stop” stay constant between safe-points
- Stop is one notch above base rank

vm_atomic.h:

- Set of types for atomic operations
  - Atomic_uint32
  - Atomic_uint64
- Operations that operate with thread-to-thread atomicity (using <lck>) on these types:
  - CompareIfEqualExchange (CAS)
  - FetchAndAdd
- Use sparingly
  - When entire data structure is single atomic value
  - When lock/unlock fails to scale or ranks do not work out
  - May need to reason about memory model, optimizer

Communication

- Monitor actions (asynchronous)
- Crosscalls (synchronous)
- [Traces: not covered in this talk]

Monitor actions

- A way to request “something” from VCPU
- Each vcpu has “in-bound” monitor action data structure
  - Multiple producers, single consumer
- Post from vmx, other vcpu, vcpu itself, vmkernel, vmmon, vmnet
- Asynchronous but fairly prompt:
  - Once “noticed”: process at end of current guest instruction
  - *Some* actions use IPIs to pull dest out of “direct exec”
    - Not all actions (IPIs are expensive)
  - Still no guarantees in absolute time

Monitor action kinds

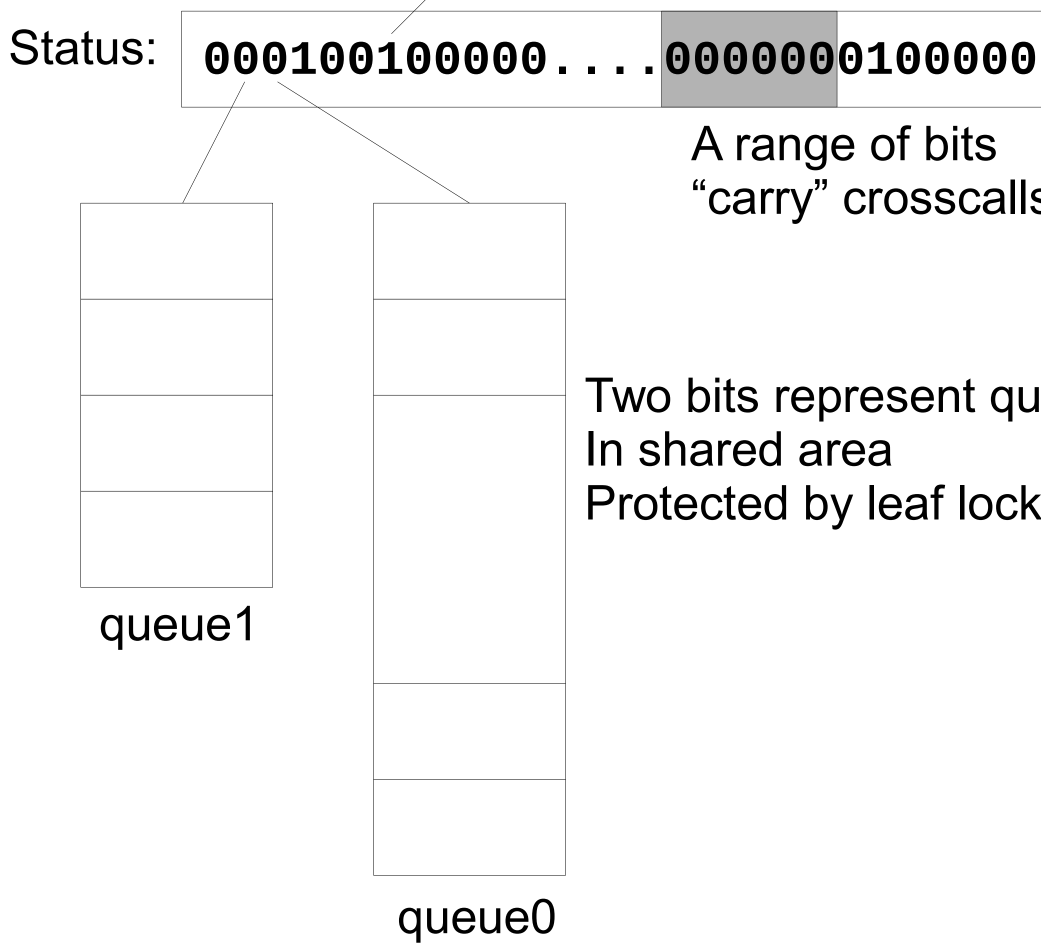
- Idempotent actions (“simple and fast”)
  - One bit only: is action present or not?
  - Examples:
    - MONACTION_MMU_ZAP (no point zapping twice)
    - MONACTION_10HZ: do “this” every 10'th of a second
- Queued actions (“richer and a bit slower”)
  - Data-carrying
  - Example:
    - Fire “write trace” on BPN 0x12345678

vcpu may post action to itself

- Process at next guest instruction boundary
- Why do this?
  - Defer operation to next guest instruction boundary
  - Side-step rank issue (e.g., Zap at high rank)
  - Currently in vmx, but work has to happen in vmm
- Deferral is not without issues
  - Must reason about *all* code between action post and processing w.r.t. deferred operation

Monitor Action implementation

Each set bit represents idempotent action
(set/clear bits using atomic ops)



A range of bits
“carry” crosscalls

Two bits represent queues (bit set if queue nonempty)
In shared area
Protected by leaf lock

Also have user-level overflow
queues (ULAQ).

`MonitorAction_DrainActionsSimple()`

- When draining all actions:
  - Invoked at base rank only
  - In *as few places as possible*
    - Action drain points are hard to reason about
    - “Anything” can happen
- Optional bit mask for selective draining
  - Optimization for DE/BT/HV transitions
  - Used for cross calls

Crosscalls

- Keith Adams' generalization of TLB shootdown
- Request from one vcpu to set of vcpus
  - Perform the “xyz” operation
  - Asap (crosscalls are “chased out” with an IPI)
  - Requester *waits* until
    - Crosscall completes (for most crosscalls), or
    - Crosscall request has been noticed and buffered
      - Known as “early release” crosscalls
      - More parallelism but harder to reason about
- Only vcpus can issue crosscalls

Crosscalls and ranks

- If vcpu X crosscalls vcpu Y...
  - vcpu X at rank 50
  - vcpu Y at rank 40
  - vcpu Y tries to take lock at rank 45
  - If vcpu X already holds the lock, we have deadlock!
- Solution: crosscallee inherits rank from caller
  - But: vcpus don't know their current rank in non-debug builds
  - Crosscalls must have rank
    - Debug builds: check caller rank against crosscall rank
- Blocking on lock with rank R: must serve crosscalls with rank >= R

Crosscalls and ranks

- If vcpu X crosscalls vcpu Y...
  - vcpu X at rank 50
  - vcpu Y at rank 40
  - vcpu Y tries to take lock at rank 45
  - If vcpu X already holds the lock, we have deadlock!
- Solution: crosscallee inherits rank from caller
  - But: vcpus don't know their current rank in non-debug builds
  - Crosscalls must have rank
    - Debug builds: check caller rank against crosscall rank
- Blocking on lock with rank R: must serve crosscalls with rank >= R

Ganesh's insight

Types of crosscalls

EXECUTE_LEAF_CB

EXECUTE_BUSMEM_CB

EXECUTE_CB

DT_TRACE_CROSSCALL

TC_TRACE_CROSSCALL

STOP_CROSSCALL

AFTER_TRACE_INSTALL

BEFORE_TRACE_INSTALL

Crosscall ranks and the BULL

```
#define RANK_MONITOR_LEAF         (RANK_userlevelLock - 1)

#define RANK_barrier              1
#define RANK_stopLock             1
#define RANK_stopCrossCall        3
#define RANK_stopResponderSema    4
#define RANK_crossUserCall        4
#define RANK_executeCB            5

#define RANK_tcTraces             10
#define RANK_beforeTraceInstall   20
#define RANK_memspaceDevice       30 /* needs traceInstall+busMem */
#define RANK_tcCohLock            (RANK_afterTraceInstall - 1)
#define RANK_mwaitLock            (RANK_afterTraceInstall - 1)
#define RANK_vnptLock             (RANK_afterTraceInstall - 1)
#define RANK_afterTraceInstall    40
#define RANK_traceALotLock        50
#define RANK_memrefDevice         60
#define RANK_beforeTraces         70
#define RANK_busMemLock           80
#define RANK_busmemCB             90
/* def'n of RANK_userlevelLock    100  in mutex.h */
#define RANK_device               110
#define RANK_iospaceLock          120
#define RANK_mksLock              130 /* above BULL, dev & iospace */

#define RANK_mmuInfoLock          (RANK_MONITOR_LEAF - 1)
#define RANK_simpleMalloc         RANK_MONITOR_LEAF
```

Crosscall implementation

- Carried with IPI-sending idempotent actions
  - IPIs expedite processing
  - IPIs expedite release of caller
  - We gain parallelism
- Crosscalls can nest
  - Only finitely -- why?
- Crosscall arguments
  - In global shared variables
  - With buffering for “early release” crosscalls
- Hooks mutex/semaphore blocking

Crosscalls and HLT: offlining

- When vcpu executes HLT
  - nothing much will happen for a while
  - we deschedule vcpu (to give cycles to others)
- Optimization opportunity:
  - Take vcpu “offline” (tell other vcpus about this)
  - Many crosscalls can now be deferred
  - Replace synchronous crosscall (expensive for descheduled vcpus) with queued action post
  - To go online: vcpu must drain all actions
- Offlining/onlining is guarded by monitor action locks

Q & A