Idle Thread Talk

Gabriel Somlo

Consider the humble 410 idle thread

```
int main()
{
        while (1) {
        }               // a.k.a. for (;;);
}
```

Consider the humble 410 idle thread

```
FOO:
        jmp FOO
```

- Funny, I expected at least a NOP in there
  - But obviously that wasn't necessary
- Run around in circles until scheduler gets invoked, and may pick something better

Let's run P3 kernel on QEMU-KVM

- 'qemu' process uses 100% of a host core
- 'idle' thread is a LOT less idle than one might expect...

Let's look through the P3 handout

- On page 31, it casually mentions:
  - “... Or, you may choose to hand-craft an idle program w/o reference to an executable file.”
- Why on ${Deity}'s Green Earth would I want to do that for ?
- And what's this about “without ... an executable file” ?

How about a less hyperactive idle ?

```
FOO:
        hlt
        jmp FOO
```

- The CPU core actually stops for a while
- Great for saving energy
- On QEMU, host CPU drops below 10%
- Interrupts wake the core, same chance to schedule something better as before

Why no executable file ?

```
FOO:
        hlt
        jmp FOO
```

- HLT is a privileged instruction
- New idle thread can never drop into user mode
- That's OK, saves us a few userspace pages (.txt, stack, etc.)

So we're done, right ?

- Idle thread now heavily sedated
- But what about SMP ?
  - Sadly, I missed the SMP P4 by one semester :)
- What if a core can “find work” for another ?

Reaction time vs. energy efficiency

- With spinning idle, we can

  FOO:

  <look_for_something_useful>

  jne FOO

  <switch_to_something_useful>

- With HLT-based idle, work accumulates
  - Until an interrupt wakes the sleeping core
  - Could be an IPI from another (awake) core
  - Way slower than spinning + mem. access

Reaction time vs. energy efficiency

- Linux used to default to HLT (see arch/x86/kernel/process.c)
  - Spinning available as option with SMP (poll_idle)
- Burning energy is the price for eternal vigilance... Or is it ?
- If only we had a compromise solution
  - Stay mellow, save energy
  - Wake without delay when needed

Enter MONITOR & MWAIT

- MONITOR: start watching a (write-back) memory location for writes
- MWAIT: turn off core until “something” writes to MONITORed memory location
- Originally intended for thread synch
  - Memory may hold some kind of lock
- Reminds me a bit of deschedule() and make_runnable() from 410...

MONITOR

```
void __monitor(const void *memaddr,
                        unsigned ext, unsigned hint);

__monitor:
        mov  <memaddr>, %rax
        mov <ext>,          %ecx  // leave 0
        mov <hint>,         %edx  // leave 0
        monitor

// a write to <memaddr> will “trigger” the
// “armed” monitoring hardware
```

MWAIT

```
void __mwait(unsigned ext, unsigned hint);
__mwait:
        mov <ext>,  %ecx   // ignore IF==0
        mov <hint>, %eax   // C-state > C1
        mwait
// sleep while memory monitor is “armed”
// wake when “triggered”, or on interrupt
// act as NOP when monitor not armed
```

Using MONITOR & MWAIT

sleeper thread

```
while (*flag == 0) {

    __monitor(flag, 0, 0);

    if (*flag == 0)

        __mwait(0, 0);

}
```

waker-upper thread

```
*flag = 1;
// while loop never entered
```

Using MONITOR & MWAIT

sleeper thread	waker-upper thread
while (*flag == 0) {	
	*flag = 1;
__monitor(flag, 0, 0);	// arming after change
if (*flag == 0)	// luckily, we check again
	// before going to sleep
__mwait(0, 0);	
}	

Using MONITOR & MWAIT

sleeper thread	waker-upper thread
while (*flag == 0) {	
__monitor(flag, 0, 0);	// trigger right after arming
	*flag = 1;
if (*flag == 0)	// we don't mwait, but it
	// would have been a NOP
__mwait(0, 0);	// regardless
}	

Using MONITOR & MWAIT

sleeper thread	waker-upper thread
while (*flag == 0) {	
__monitor(flag, 0, 0);	
if (*flag == 0)	// trigger before mwait
	*flag = 1;
__mwait(0, 0);	// mwait acts as NOP
}	

Using MONITOR & MWAIT

sleeper thread	waker-upper thread
while (*flag == 0) {	
__monitor(flag, 0, 0);	
if (*flag == 0)	
__mwait(0, 0);	// first sleep, then trigger
	*flag = 1;
}	// canonical use case

How do MONITOR & MWAIT work ?

- Why the while() loop in the example ?
  - MWAIT *may* also wake up when one looks at it funny
- Based on cache coherence protocol (wikipedia: MESI, also expects write-back)
- Armed on valid cache line(s)
- Triggered when cache line(s) invalidated
- Although size of monitored area is NOT equal (or even related) to size of cache line

How do MONITOR & MWAIT work ?

- CPU (via CPUID) will report size of monitored memory area
- Intel docs mention “cache coherence line” a few times, all on the same page (of 900+)
  - No definition, though
  - Obviously dependent on L1 coherence protocol implementation
  - Suspecting relationship to L2 line size
- On single CPU, MWAIT behaves like HALT (modulo DMA, which causes it to wake up)

Can we use MONITOR & MWAIT ?

- Per docs, one *must* check CPUID for availability before use !
- Linux: checks CPUID, and prefers MWAIT for its idle thread (over HLT or poll_idle)
- Windows: probably, but who cares ;)
- OS X: *blatantly* calls MONITOR & MWAIT *without* checking CPUID !
  - Because it *knows* they're there !
  - No MWAIT ? “Just Buy a Mac! (tm)”

Emulate MONITOR / MWAIT in KVM

- Start with a closer look at VMX (or SVM)
- VMCS: the VM “control structure”
  - *Very* fine-grained control of VM behavior, way beyond Popek-Goldberg
  - Per-VM list of “Things Which Will Trap” (or “cause VM exits”, in Intel-speak)
  - Some 20+ instructions are on the list, and may be *optionally* configured to cause a VM exit, or not
  - MONITOR & MWAIT are on the list

KVM vs. MONITOR & MWAIT

- When KVM initializes a VMCS, it asks for MONITOR & MWAIT to cause VM exits
- Current emulation handler for both is handle_invalid_op()
    - This causes an “invalid opcode” fault as observed by the guest
    - The guest virtual CPU's CPUID never claimed to support them
    - But OS X doesn't check CPUID !

KVM vs. MONITOR & MWAIT

- Existing KVM patch for OS X turns off VMCS flags asking for VM exit on MONITOR and MWAIT
  - Leave guest to run MONITOR & MWAIT
  - Assume architecturally same as NOP
  - Assume enough noise to keep MWAIT awake (theoretically a safety concern !)
    - On single-vcpu QEMU-KVM, should ≈ HLT
    - Burning 100% host CPU, meaning ≈ NOP !
    - At the mercy of underlying h/w details !

KVM vs. MONITOR & MWAIT

- A “better” KVM patch (tried and working)
  - Leave VM exit flags on
  - Modify emulation handler instead
  - Currently using handle_pause()
- PAUSE is “NOP for spin-wait loops”
  - Prevents false-positive “memory order violation detection” in spin-wait loops
  - *May* add a delay vs. “true” NOP
  - handle_pause() tries to yield to other vcpu

KVM vs. MONITOR & MWAIT

- Now I have MONITOR & MWAIT emulation
- MWAIT can do one of three things:
  - PAUSE (MONITOR as well)
    - safe & easy (functional patch already done)
    - downside is hyperactive idle
  - HLT (MONITOR remains ≈ PAUSE)
    - medium-range trickery (see next slide)
  - True emulation
    - probably hard
    - must trap memory writes by other VCPUs !

OS X vs. MONITOR & MWAIT

- Only one function in OS X uses them
  - Must be the idle thread !
  - From all available VCPUs
  - IOPL / CPL is 0 (kernel mode)
- IF is 0 (interrupts disabled)
  - No wonder emulating as HLT hangs !
  - MWAIT %ecx is 1 (wake on int if IF=0)
    - So it works on single-CPU systems
- Try: “Modified HLT + *always* wake on INT” !

Questions, Comments ?

- Thanks !