Operating Systems Design (CS 423)

Elsa L Gunter

2112 SC, UIUC

http://www.cs.illinois.edu/class/cs423/

Based on slides by Roy Campbell, Sam King, and Andrew S Tanenbaum

Simulator

- CPU state --- data structure within your sim.
  - Includes registers, IDT, etc.
- Memory --- malloc
  - Guest physical address 0 == beginning of malloc
- Disk --- file
  - Disk block 1 = file offset 1*(size of disk block)
- Display --- window
- Within simulator, does the software “know” it is being simulated?

VMM environment

- Duplicate
  - Virtual machine == physical machine
- Efficient
  - Runs almost as fast as real machine
- Isolated
  - VMM has control over resources
  - What are the resources the VMM controls?
- Which properties does Simulator have?

Key observation

- If the virtual ISA == physical ISA
  - CPU can perform simulation loop
- Can we set host PC to address we want to start at and let it run?
- What property are we violating?
- What else goes wrong?

Main issues

- Privileged instructions
- Instructions operate on physical state, not virtual state

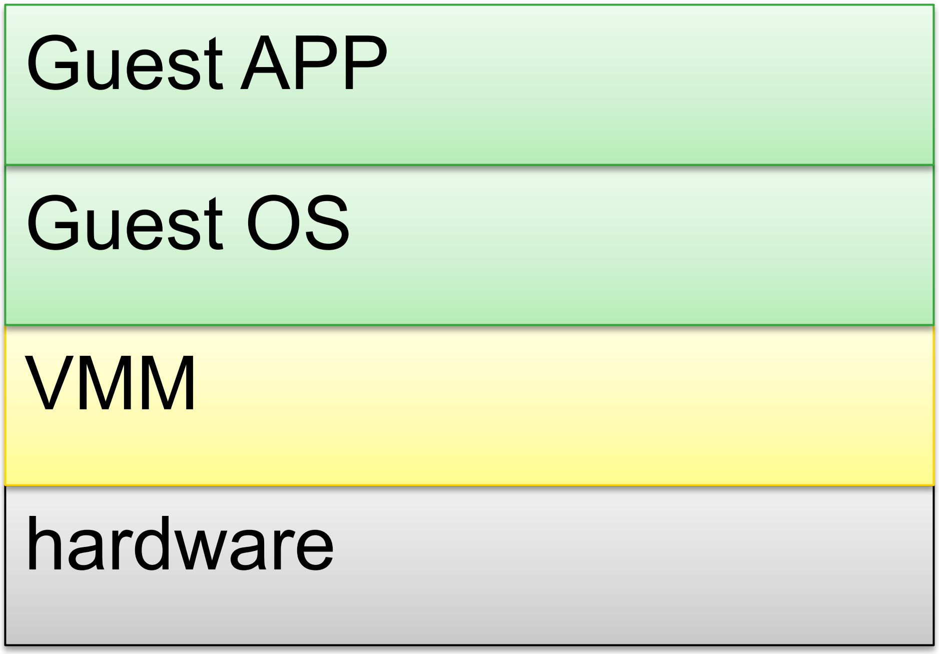
Privileged instructions

- CPU divided into supervisor and user modes
- Part of the CPU ISA only accessible by “supervisor” code
- Allowing guest OS execute these would violate isolation
    - E.g., I/O instructions to write disk blocks
- Solution: run guest OS in user mode
    - CPU user mode: only non-privileged instr
    - CPU supervisor mode: all instr

Example: interrupt descriptor table (IDT)

- x86 processor have an IDT register
  - CPU uses to find interrupt service routines
- Set the IDT register using the lidt instruction
- VMM must handle all interrupts to maintain control
- Goal: allow the guest OS to set the virtual IDT register without affecting the physical CPU

Guest OS priv. inst.



Guest APP

Guest OS

VMM

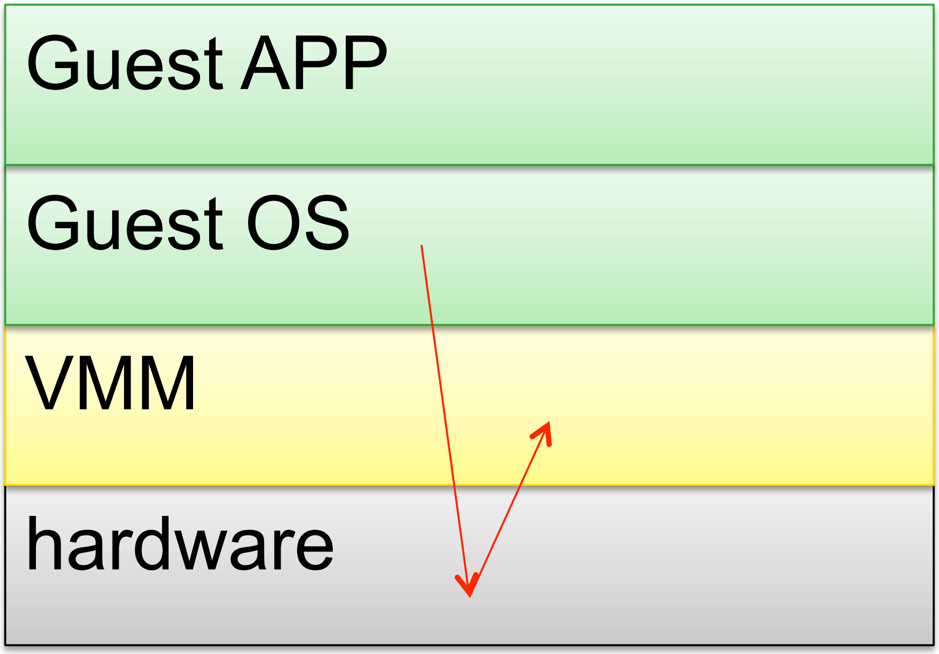
hardware

🟩 User mode

🟨 Supervisor mode

- lidt 0x1234
- vcpu->idtr = 0x4321
- idtr = 0x5555

Guest OS priv. inst.

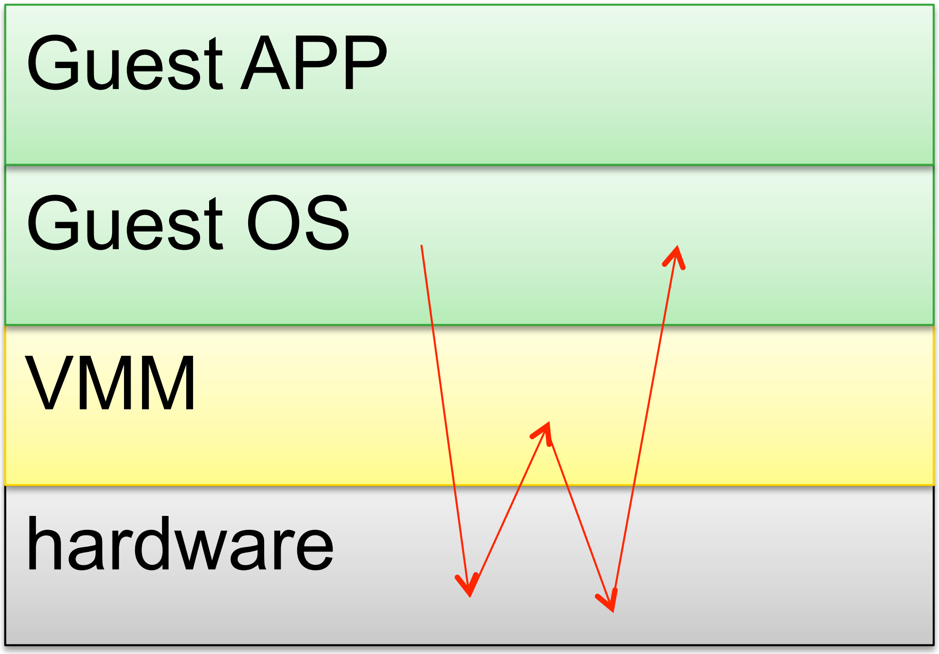


🟩 User mode

🟨 Supervisor mode

- lidt 0x1234
- vcpu->idtr = 0x4321
- idtr = 0x5555

Guest OS priv. inst.

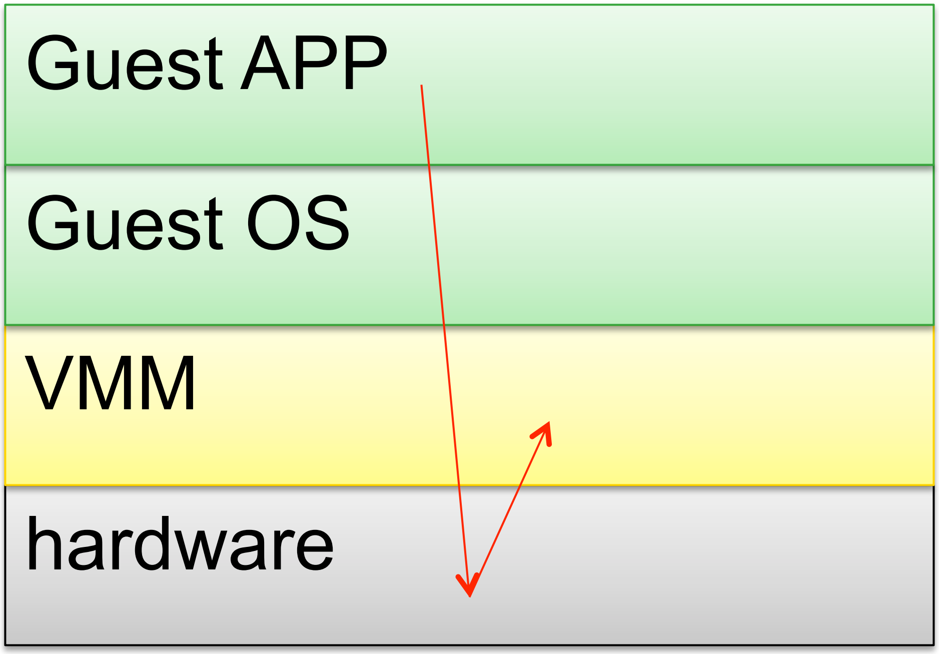


🟩 User mode

🟨 Supervisor mode

- lidt 0x1234
- vcpu->idtr = 0x1234
- idtr = 0x5555

Guest OS priv. inst.



Guest APP

Guest OS

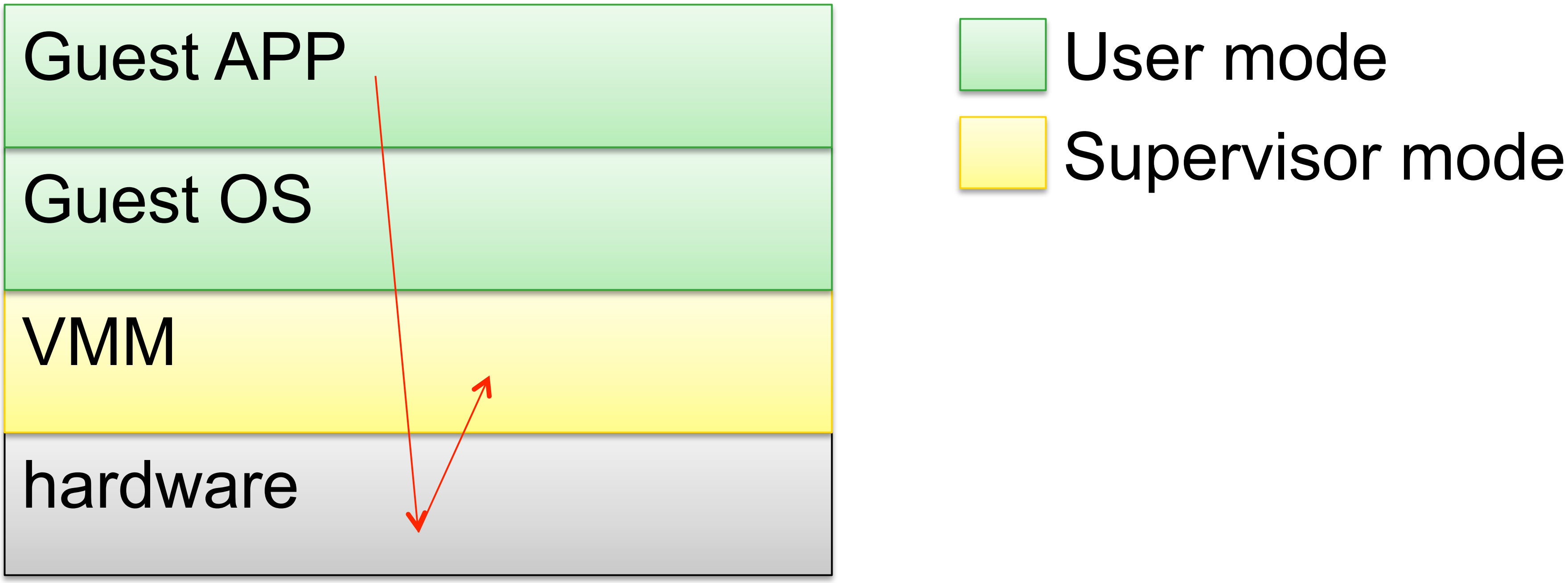
VMM

hardware

🟩 User mode

🟨 Supervisor mode

Guest OS priv. inst.



Guest APP

Guest OS

VMM

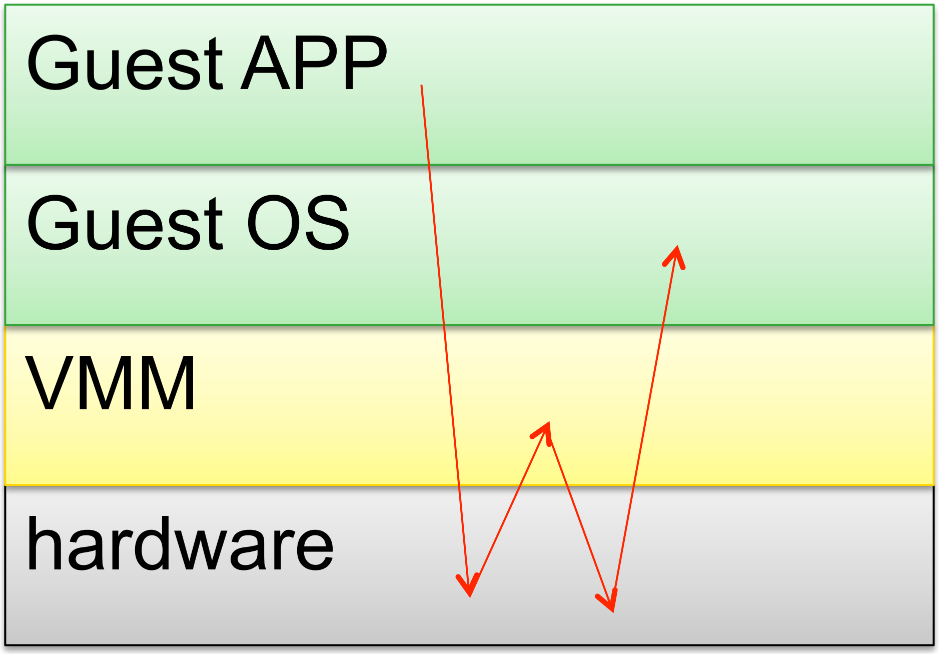
hardware

🟩 User mode

🟨 Supervisor mode

- vcpu->supervisor = false

Guest OS priv. inst.



🟩 User mode

🟨 Supervisor mode

- vcpu->supervisor = true

Sensitive non-privileged instructions

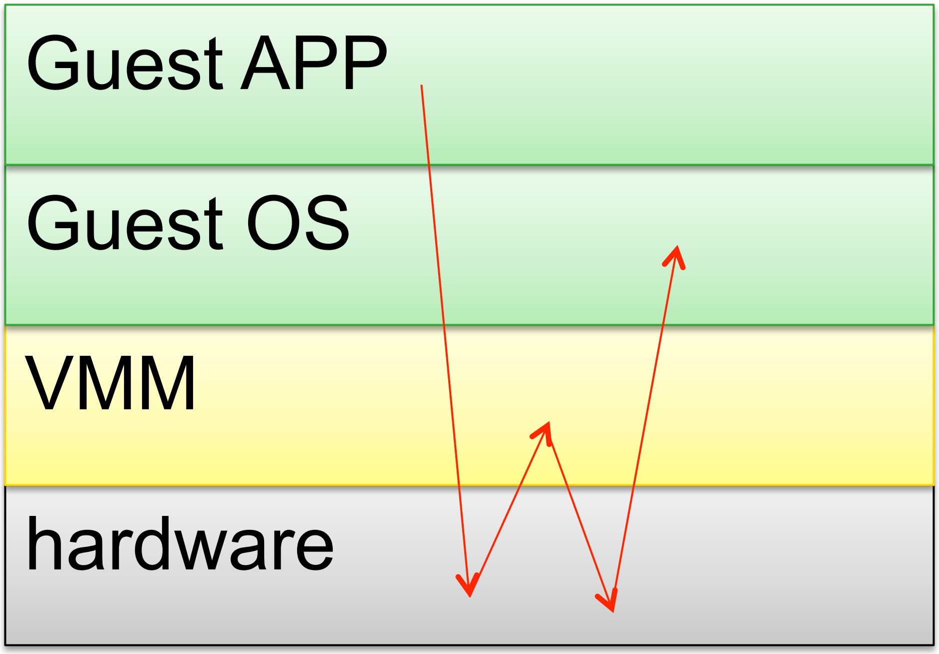
- Privileged instruction
  - Trap when called from CPU user mode
- Sensitive instruction
  - Leaks information about physical state of processor
  - Eg `sidt`
- Fully virtualizable processor
  - All sensitive instructions are privileged
- What about x86?

Sensitive non-privileged instructions

- x86 has a number of instructions that are sensitive, non-privileged instructions
  - Causes physical state of CPU to leak
  - Why is this a problem?

- What property does this violate?

Guest OS sensitive non-priv. inst.



🟩 User mode

🟨 Supervisor mode

- sidt = 0xabcd
- vcpu->idtr = 0x1234
- idtr = 0xabcd

Key insight

- Solution: simulate the OS, let user-mode code run natively
  - Simulation is flexible
  - Can interpose on all instructions
  - Problem: simulation is too slow
- Will work if guest OS calls SIDT, what about guest user mode?

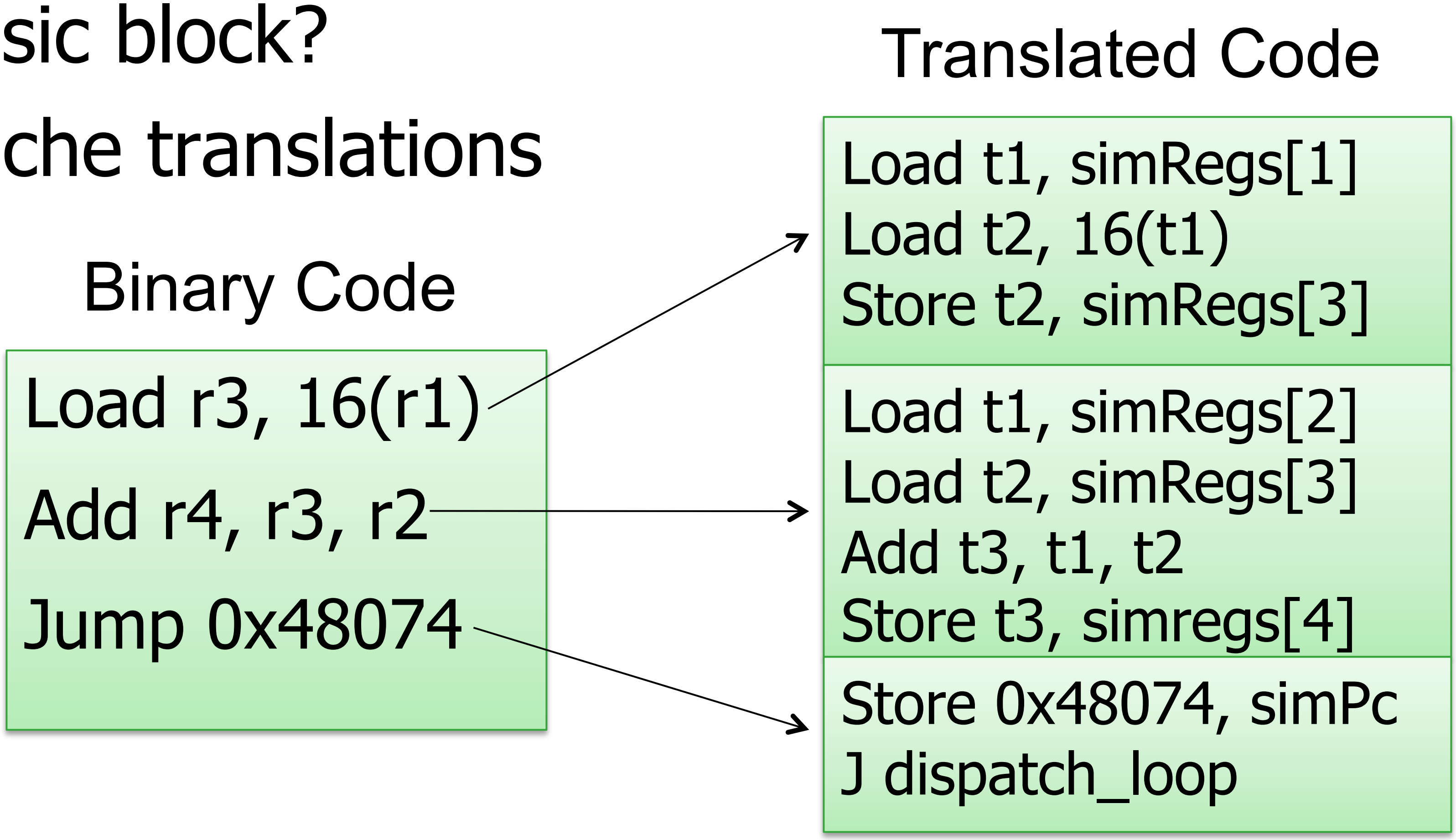
Simulation flexibility

- Normal simulations flexible, slow
- Can we simulate fast, still flexible?

```
while(1){
 inst = mem[PC]; // fetch
 if(inst == add) { // decode// execute
   reg[inst.reg1] = reg[inst.reg2] + reg[inst.reg3];
   PC++;
  }
 else if ...
 } // repeat
```

Dynamic binary translation

- Simulate basic block using host instructions. Basic block?
- Cache translations



Binary Code

```
Load r3, 16(r1)
Add r4, r3, r2
Jump 0x48074
```

Translated Code

```
Load t1, simRegs[1]
Load t2, 16(t1)
Store t2, simRegs[3]
```

```
Load t1, simRegs[2]
Load t2, simRegs[3]
Add t3, t1, t2
Store t3, simregs[4]
```

```
Store 0x48074, simPc
J dispatch_loop
```

Performance

```
while(1){
 inst = mem[PC]; // fetch
 //decode
 if(inst == add){
 //execute
 ...
 }
} // repeat
```

```
while(1){
 if(!translated(pc)){
  translate(pc);
 }
jump to pc2tc(pc);
} // repeat
```

- Why is this faster?

- Are there disadvantages?