

Analysis of the Intel Pentium's Ability Ability to Support a Secure Virtual Virtual Machine Monitor

Based on "Analysis of the Intel Pentium's Ability to Support a Secure Virtual Machine Monitor"
Monitor"

John Scott Robin (U.S. Air Force) and Cynthia E. Irvine (Naval Postgraduate School)
USENIX Security Symposium, Denver, Colorado, August 2000

Contents

- 01** Introduction and motivation
- 02** VMM types and hardware requirements
- 03** Pentium architecture analysis
- 04** The 17 sensitive, unprivileged instructions
- 05** Pentium-based VMM security problems
- 06** A better approach and conclusions

01

Introduction and motivation

What is a virtual machine monitor?

"a hardware-software duplicate of a real existing computer system in which a statistically dominant subset of the virtual processor's instructions execute on the host processor in native mode"

— Goldberg [12], 1972

- Configure the environment. A system manager can configure each VM differently from the real machine — e.g., give a VM only 8 MB of a 32 MB machine to test resource-constrained behaviour.
- Concurrent OS execution. Different operating systems run simultaneously on the same hardware. Application development and testing across platforms is easier.
- Isolate untrusted applications. A program downloaded from the internet can be tested in a VM. If it contains a virus, the virus is isolated to that VM.
- Legacy OS upgrade. A legacy OS runs in a VM while a new OS version runs in another VM on the same hardware.

Three VMM characteristics (Popek & Goldberg)

ONE

Identical environment

Any program executed on a VM should run the same as on an unvirtualized machine. Exceptions result from differences in resource availability, timing dependencies, and attached I/O devices.

TWO

Resource control

No program running on a VMM can access any resource not explicitly allocated to it by the VMM. The VMM can also regain control of previously allocated resources.

THREE

Efficiency

A large percentage of the virtual processor's instructions must be executed by the machine's real processor without VMM intervention. Instructions that cannot be executed directly are interpreted by the VMM.

"Can the Intel Pentium support a secure VMM?"

A secure VMM for the Intel Pentium would be very desirable: a single machine could implement critical security policies while also running popular Win32 operating systems and applications.

Although the x86 processor family had been used as the base for many highly secure systems — Boeing MLS Lan (A1), Gemini Trusted Network Processor (A1), Verdix VSLAN (B2), TIS Trusted Xenix (B2) — it had not been considered as a VMM base.

This paper presents an analysis to determine whether the Intel Pentium architecture can support a highly secure VMM without sacrificing user convenience.

02

VMM types and hardware requirements

Three types of virtual machine monitor

TYPE I VMM

Runs directly on bare machine hardware. It is an operating system with virtualization mechanisms. Performs scheduling and resource allocation for all VMs. Requires device drivers for all hardware peripherals. Best performance and security isolation.

TYPE II VMM

Runs as an application on a host operating system. The host OS controls the real hardware. Each OS run in the virtual environment is a "guest OS." Relies on the host OS for memory management, scheduling, and device drivers. Security is bounded by the host OS.

HYBRID VMM (HVM)

Interprets every privileged instruction in software; all non-privileged instructions execute directly. Possible on a larger class of hardware than a VMM. Functionally equivalent to the real machine but with lower performance due to increased interpretation.

Hardware requirements for a Type I VMM

Requirement 1	The method of executing non-privileged instructions must be roughly equivalent in both privileged and user mode. A processor cannot use an additional bit when in privileged mode that is not available in user mode.
Requirement 2	There must be a protection system or address translation system to protect the real system and any other VMs from the active VM.
Requirement 3A	There must be a way to automatically signal the VMM when a VM attempts to execute an instruction that changes or references the mode of the VM or the state of the machine.
Requirement 3B	Instructions that read or change sensitive registers and/or memory locations — such as a clock register or interrupt registers — must trap.
Requirement 3C	Instructions that reference the storage protection system, memory system, or address relocation system — including any that would allow a VM to access a location not in its virtual memory — must trap.
Requirement 3D	All I/O instructions must trap.

03

Pentium architecture analysis

The Pentium meets three of the four Goldberg features

Goldberg identified four key architectural features of third generation hardware pertinent to virtual machines. The Intel Pentium provides all four:

Two processor modes

The Pentium has four rings (CPL 0–3). Ring 0 is the most privileged (OS). Ring 3 is least privileged (applications). This maps onto the two-mode model — OS in ring 0, VMs in ring 3.

Non-privileged → privileged calls

The Pentium uses the call gate mechanism to control transfer of program execution between privilege levels, so non-privileged tasks can call privileged system routines.

Memory protection

The Pentium uses both paging and segmentation. Segments have a Descriptor Privilege Level (DPL) 0–3 to control access.

Asynchronous interrupts

The Pentium uses both interrupts and exceptions. The architecture has 16 predefined interrupts/exceptions and 224 user-defined maskable interrupts.

Requirement 3 fails: sensitive, unprivileged instructions

The Pentium instruction set contains sensitive, unprivileged instructions. The processor will execute these without generating an interrupt or exception. Thus, a VMM will never have the opportunity to simulate the effect of the instruction.

- 17 instructions violate Requirement 3 after examining all ~250 documented Pentium instructions.

17

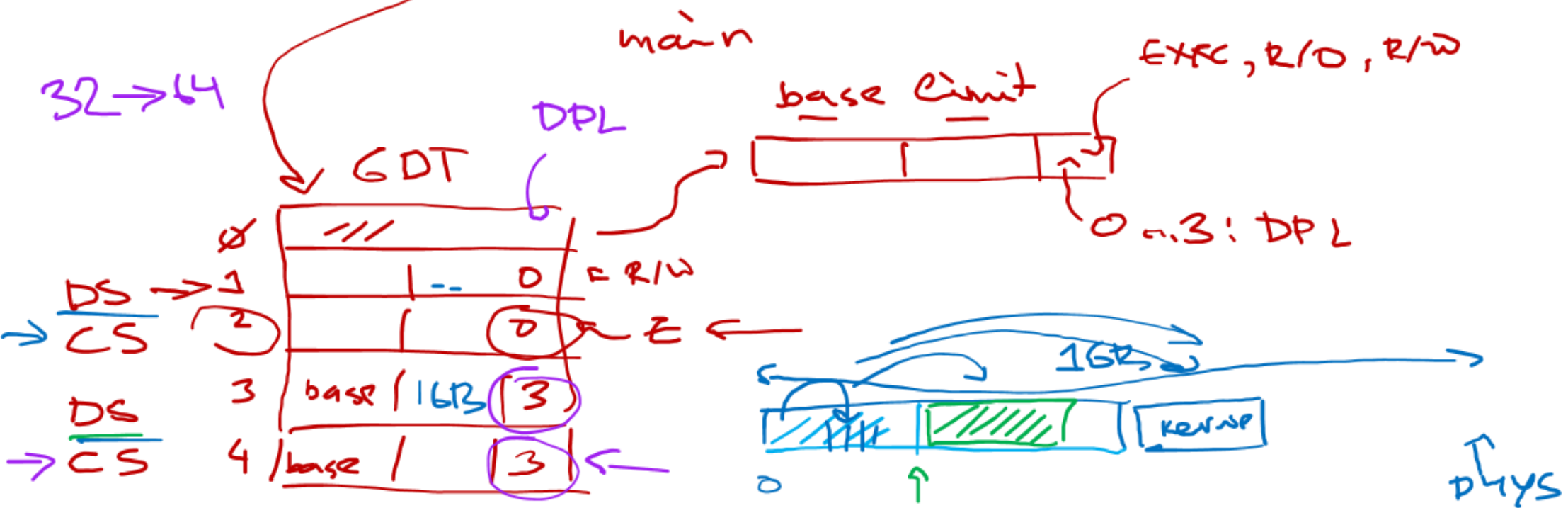
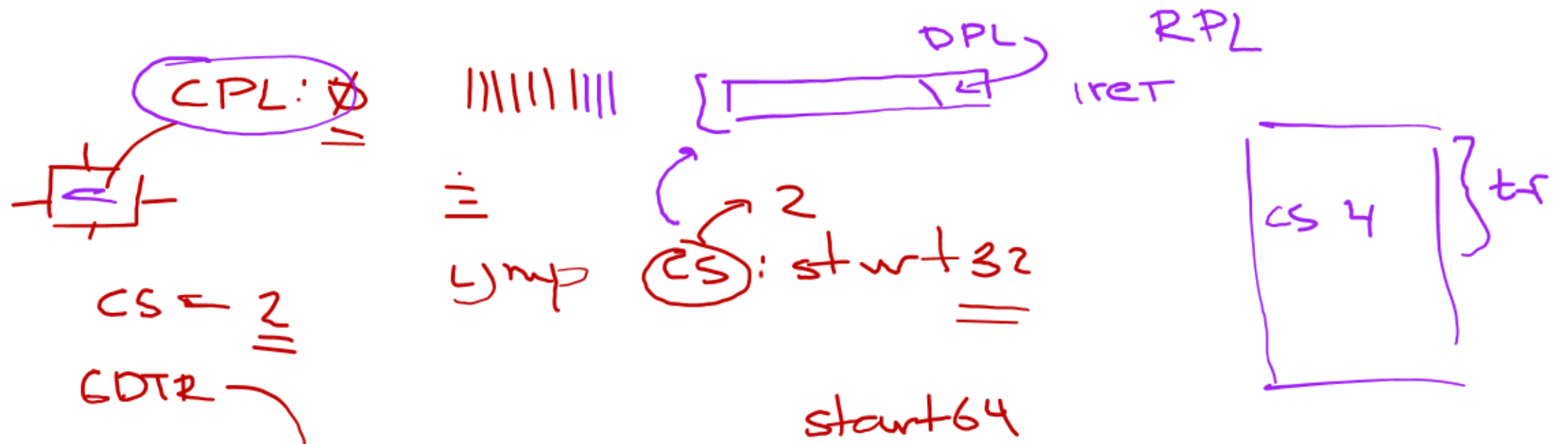
- All 17 violate either Requirement 3B (sensitive registers) or Requirement 3C (protection system references).
- This makes the Intel processor non-virtualizable according to Goldberg's hardware rules.

04

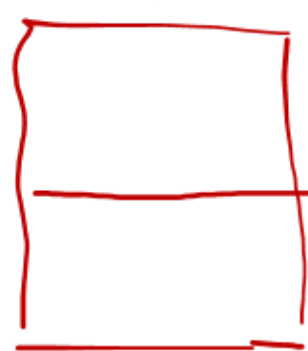
The 17 sensitive, unprivileged instructions

Detour

Recalling core x86 isolation mechanisms



USTACK



CS
IP RFLAG

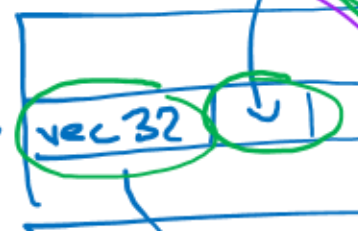
RSP

ESP

←

32

IDT



CS

KERNEL
CODE

IP

EFLAGS

SS

RSP

PL=1

MOV

main()

=

IRQ
TIMER 32

→ mov RSP, x7z

ALIE

PUSH DDDD

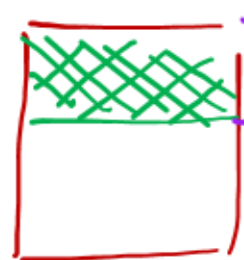
CPL:3

ring 3

ring 0

LF

KSTACK



CPL:0

IDT

Root PG TABLE

kernel

CPL:0

TSS

ring 0

RSP

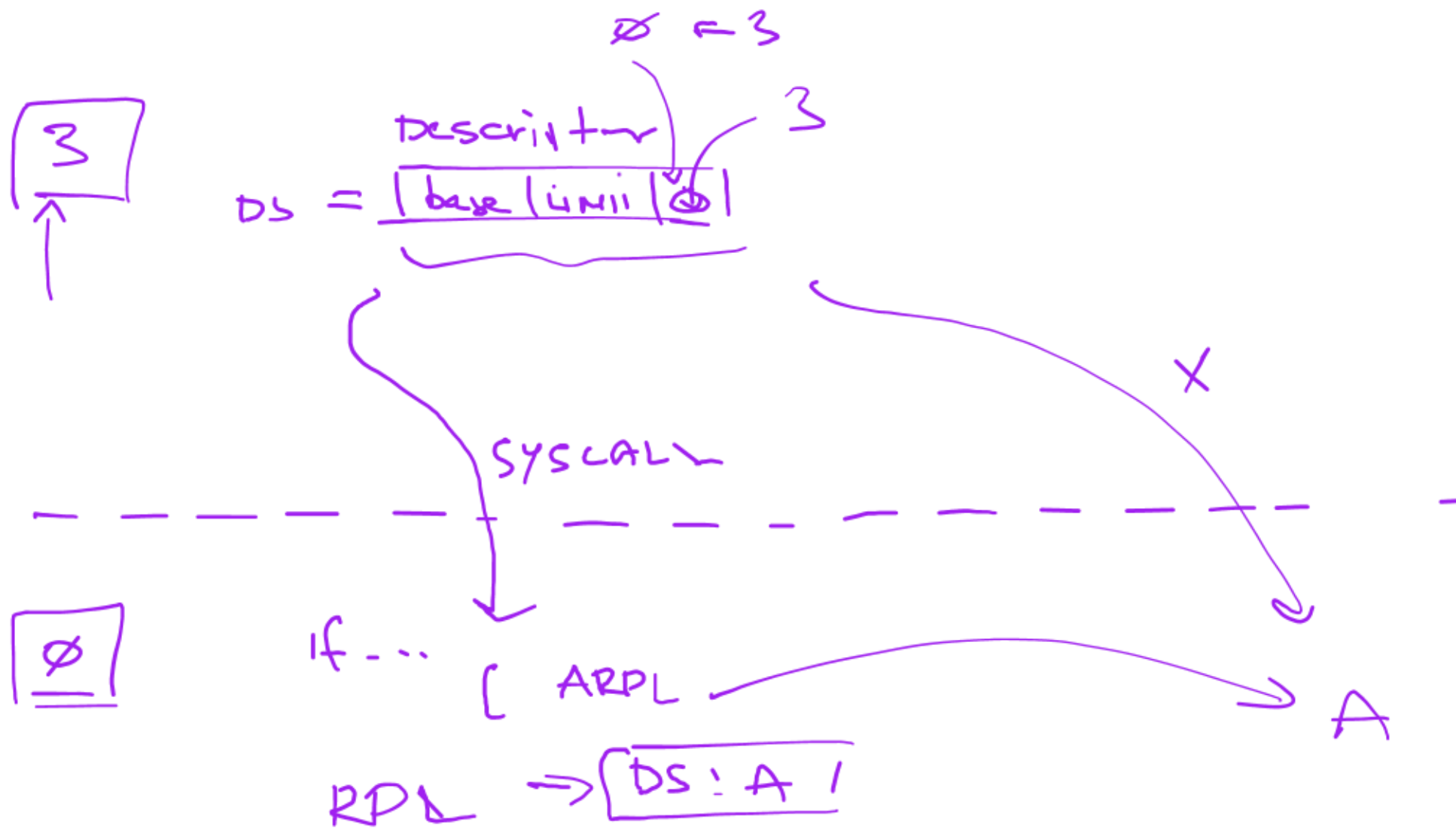
IRET

GPF

DDD

vector 32

IP



End of detour

SMSW — store machine status word

What the instruction does

SMSW stores bits 0–15 of Control Register 0 (CR0) into a general-purpose register or memory. These bits are the Machine Status Word: bit 0 (PE) enables protected mode vs real mode; bit 1 (MP) controls coprocessor monitoring; bit 2 (EM) signals FPU emulation mode; bit 3 (TS) is set on a task switch; bit 5 (NE) controls numeric error reporting. The PE bit is the most critical — it tells the reader which fundamental operating mode the processor is in.

Why it is sensitive and cannot be trapped

A VMM running in protected mode may host a guest that believes it is in real mode. The guest reads SMSW to check PE and sees PE=1, concluding the machine is in protected mode. It may halt or refuse to run. Because SMSW exists only for Intel 286 backwards compatibility and was deliberately made non-privileged, there is no CPL check — the VMM has no trap hook and cannot return a virtualised MSW.

The fix the paper proposes

Robin and Irvine note that SMSW could simply be removed. Programs written for the Intel 386 and later are supposed to use the privileged MOV CR0 instruction, which does trap and can be emulated. Only systems requiring 286 backwards compatibility would notice SMSW's absence, and application software (as opposed to OS code) should never call it. Removing it would cost nothing for modern workloads.

SMSW — store machine status word (not used by xv6)

TRAPS → VMM can emulate

✓ (no case traps)

SMSW is unconditionally non-privileged. The processor executes it in any ring without any privilege check.

DOES NOT TRAP → VMM has no hook

✗ Always returns real CR0 low bits

The PE bit and other mode flags are always real hardware state — not the virtualised state the VMOS believes it is in.

✗ Breaks emulated real-mode guests

A VMOS in emulated real mode reads PE=1 and concludes it is in protected mode. It may halt.

✗ Category: behavior-sensitive (mode)

Result depends on the current processor mode — mode-sensitive by Popek-Goldberg definition.

SMSW is a legacy 286 instruction. The paper notes it could simply be removed; only 286-compatibility systems would be affected.

PUSHF / PUSHFD — push flags register

What the instruction does and what it exposes

PUSHF pushes the lower 16 bits of EFLAGS onto the stack; PUSHFD pushes the full 32 bits. EFLAGS contains fields the OS actively manages: IF (bit 9) controls whether the CPU acknowledges hardware interrupts; IOPL (bits 12–13) sets the I/O privilege level; VM (bit 17) enables Virtual-8086 mode; TF (bit 8) enables single-step debugging. All of these are the VMM's real hardware state, not the virtualised state the guest believes it has set.

Why the guest reading EFLAGS is a problem

A guest OS that saves flags before a critical section — a standard kernel pattern — calls PUSHF and then inspects the saved value. It sees the real IOPL and IF. IOPL will not be 0 as expected (the VMM must keep IOPL at 3 to let user mode run), and IF may be in a state the VMOS did not set. The VMOS concludes something is wrong with the hardware or directly detects that it is not running at ring 0 with full flag control.

Why there is no mitigation on an unmodified Pentium

PUSHF has no privilege check of any kind — pushing registers is a fundamental operation. The VMM has no trap to catch. It cannot intercept the push and substitute a virtualised EFLAGS with fake IOPL/IF. The only remedy is binary translation: scan guest code before it runs, rewrite every PUSHF into a sequence that reads from a per-guest EFLAGS shadow rather than from the real hardware register.

PUSHF / PUSHFD — push EFLAGS

R ✓ *REAL*

TRAPS → VMM can emulate

✓ (no case traps)

PUSHF/PUSHFD are unconditionally non-privileged. They execute at any CPL without any check.

DOES NOT TRAP → VMM has no hook

✗ Always pushes real EFLAGS

EFLAGS includes the real IOPL and IF flag. These are pushed exactly as they are in hardware — no virtualisation.

✗ Guest can read IOPL and IF

The VMOS sees $\text{IOPL} \neq 3$ or IF in an unexpected state, revealing it is not at ring 0 with full interrupt control.

✗ Category: behavior-sensitive (mode)

Values of IOPL and IF in pushed EFLAGS depend on the real privilege state, which the VMOS did not control.

A VMM would need to intercept PUSHF and push a virtualised EFLAGS — but it has no opportunity to do so.

POPF / POPFD — pop flags register

What the instruction does and the CPL=3 silent failure

POPF pops the top of the stack into EFLAGS. At CPL=0 it writes every non-reserved flag — including IF, IOPL, and VM — with full effect. At CPL=3 it silently ignores writes to IOPL, VIF, VIP, and VM. IF is only updated if $CPL \leq IOPL$, which is false when the guest runs at CPL=3 with IOPL=0. Critically, no exception is raised and no trap fires. The instruction completes normally and returns to the guest, which has no way to know the flags were not updated.

The concrete damage: broken critical sections

Kernel code disables interrupts before a critical section with CLI (or a POPF that clears IF), does sensitive work, then restores the old flags with POPF. Inside a VMM, the POPF at the end of the critical section appears to succeed but IF remains cleared. Interrupts stay disabled indefinitely. The guest kernel deadlocks or data structures are corrupted by a race condition — and the VMM was never notified any of this happened.

Why this is the canonical Popek-Goldberg failure

Popek and Goldberg named this exact pattern in 1974: a control-sensitive instruction (it attempts to change IF, a mode-control bit) that executes at user level without trapping — a "user-mode NOP." Their theorem states that if any such instruction exists, a VMM cannot be constructed by trap-and-emulate. POPF on x86 is that instruction, present and unchanged 26 years after the theorem identified it as a fatal flaw.

POPF / POPFD — pop EFLAGS

TRAPS → VMM can emulate

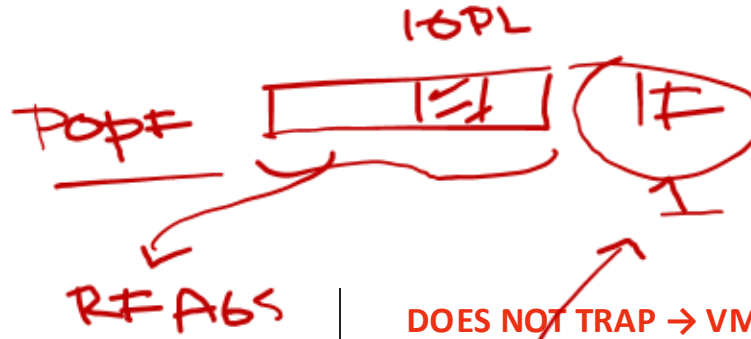
✓ CPL=0: all flags updated normally

At ring 0, POPF updates every non-reserved flag including IF, IOPL, VM.
Works correctly for a real kernel.

$IOPL = \emptyset, 1, 3$

↑

{ in ←
out ←



DOES NOT TRAP → VMM has no hook

✗ CPL=3: silently ignores sensitive fields

Writes to IOPL, VIF, VIP, VM silently dropped. IF only updated if $CPL \leq IOPL$. No exception raised.

✗ Guest believes it disabled interrupts

VMOS executes POPF to clear IF. Instruction appears to succeed — but IF is unchanged. VMM never knew.

✗ Undetectable: no trap, data corruption

VMOS proceeds with interrupts enabled. Race conditions and deadlocks follow inside the guest.

This is the "user-mode NOP" Popek and Goldberg named in 1974. POPF on x86 at CPL=3 is exactly that — 26 years later.

Xv6 example: PUSHFQ in xv6 — readrflags() breaks the spinlock interrupt mask

xv6 source — x86.h + spinlock.c

```
// x86.h:152
readrflags() {
    asm volatile(
        "pushfq; popq %0"    // ← PUSHFQ — not privileged
        : "=r" (rflags));
}

// spinlock.c:104
pushcli() {
    rflags = readrflags(); // ← reads REAL RFLAGS
    cli();                 // privileged, traps to VMM
    if (ncli == 0)
        intena = rflags & FL_IF; // stores REAL IF bit
    ncli++;
}

// spinlock.c:115
popcli() {
    if (ncli==0 && intena)
        sti(); // re-enables interrupts
}
```

RFLAGS

Failure under virtualization

① cli() is privileged — traps to VMM

VMM intercepts the trap and sets its virtual-IF flag to 0. But VMM does NOT clear the real RFLAGS.IF — it only tracks a shadow variable.

② PUSHFQ executes without trapping

readrflags() runs pushfq unintercepted. The real RFLAGS.IF is still 1 (never cleared). xv6 reads IF = 1 from the real hardware.

③ intena = 1 stored in per-CPU state

pushcli() records that interrupts were enabled before the cli(). But this is the REAL IF, not the VMM's virtual state.

④ popcli() calls sti() to restore

When the critical section ends, popcli() sees intena = 1 and calls sti(). VMM intercepts sti() with virtual-IF = 1 — overriding whatever it had.

⑤ All spinlock critical sections broken

Every acquire()/release() pair calls pushcli()/popcli(). The PUSHFQ vs. VMM-emulated-CLI mismatch corrupts interrupt masking silently — no trap, no error.

Every acquire() and release() calls pushcli()/popcli() — readrflags()/PUSHFQ runs on every kernel spinlock. The entire locking mechanism is corrupted silently.

xv6-64 source — x86.h:152, spinlock.c:104

Recap: IOPL — I/O Privilege Level field in RFLAGS

What IOPL controls

in / out / ins / outs

Direct hardware I/O port instructions are permitted only when $CPL \leq IOPL$. With $IOPL=0$ (xv6 default), user space at $CPL=3$ always faults — only kernel code at $CPL=0$ can access I/O ports.

cli / sti

Disabling and enabling interrupts also requires $CPL \leq IOPL$. This is why cli/sti are privileged in xv6 — they trap to the VMM under ring compression, letting the VMM maintain its virtual interrupt flag.

Who can change IOPL

Only $CPL=0$ code can modify the IOPL field — via popfq (restores RFLAGS from stack) or iretq (restores RFLAGS on trap return). At $CPL>0$, the IOPL bits in the stack value are silently ignored.

Why IOPL breaks virtualization

① PUSHFQ reads real IOPL

PUSHFQ pushes real RFLAGS — including the real IOPL set by the VMM (0). If the guest OS wants $IOPL=3$ for its user processes, it cannot detect that IOPL is wrong — PUSHFQ returns the VMM's IOPL silently.

② IRETQ cannot restore IOPL

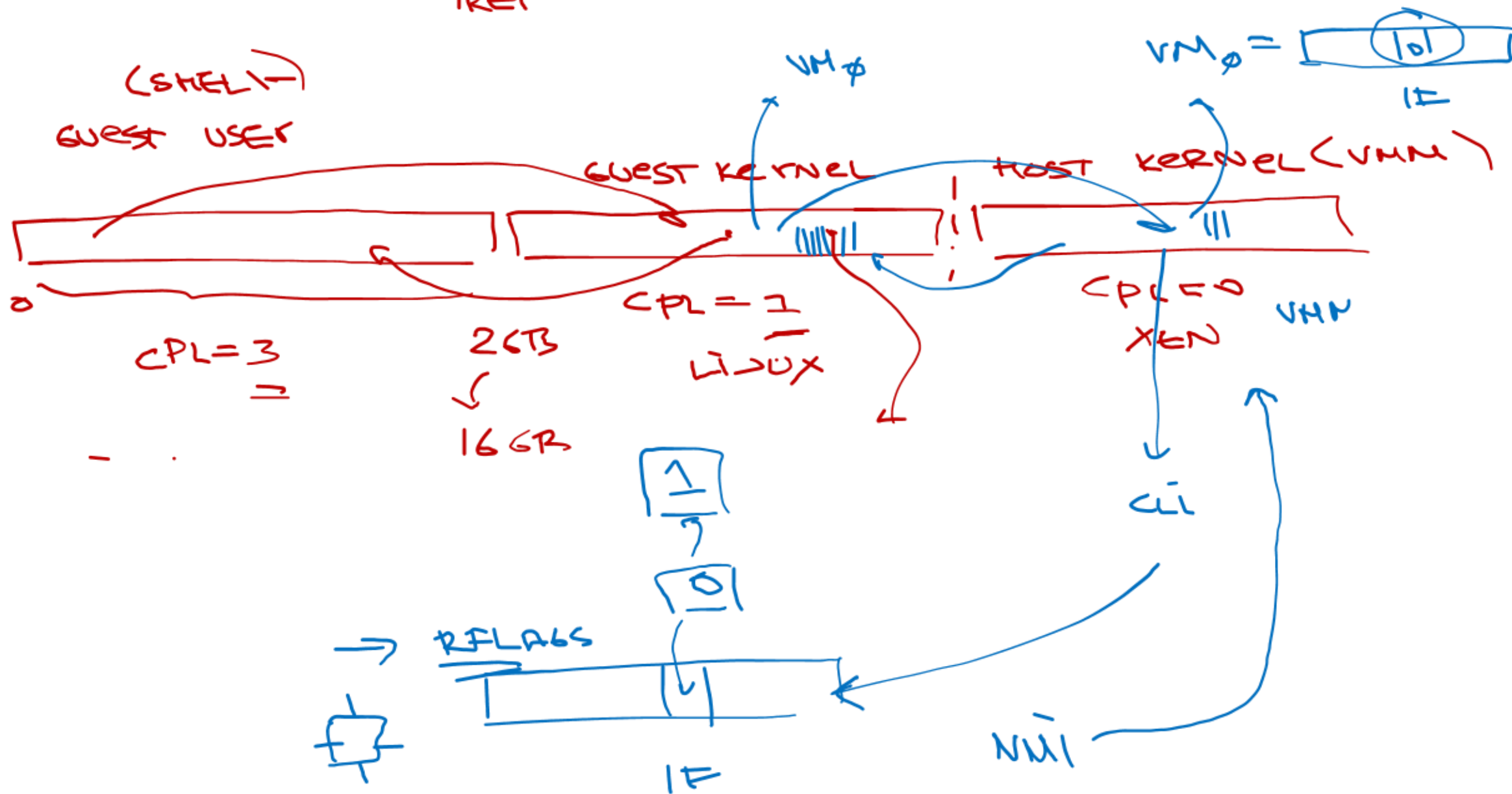
A guest OS at real $CPL=1$ executing iretq with $IOPL=3$ in the saved RFLAGS: the IOPL bits are silently dropped. User processes never get I/O access even if the guest OS explicitly grants it. No fault, no error — just silent failure.

IOPL is the link between three non-virtualizable instructions: PUSHFQ reads it, POPFQ cannot change it at $CPL>0$, and IRETQ cannot restore it at $CPL>0$. The VMM controls IOPL via the real RFLAGS, but the guest OS has no way to know its view of IOPL is wrong.

XEN

IRET

Shutdown



LAR, LSL — load access rights / segment limit (not used in xv6)

What the instructions do and the privilege check they perform

LAR loads the access-rights byte of a segment descriptor (type, DPL, present bit, etc.) into a general-purpose register. LSL loads the segment limit. Both locate the descriptor by looking up the supplied selector in the GDT or LDT. Before returning data they check $\max(\text{CPL}, \text{RPL_of_selector}) \leq \text{DPL_of_descriptor}$. If the check fails, ZF is cleared and the destination register is zeroed. If it passes, ZF is set and the data is returned. No exception is raised in either case.

Why the silent failure breaks virtualisation

A guest OS runs at CPL=3 but its kernel was compiled assuming CPL=0. When the VMOS calls LAR on one of its own kernel segment descriptors (DPL=0), the check $\max(3, \text{RPL}) \leq 0$ fails: ZF=0, register zeroed, no exception. The VMOS was not written to handle ZF=0 from LAR on its own segments — it assumes unconditional access to its own descriptors. The result is incorrect decisions about segment attributes, crashes, or silent data corruption with no VMM notification.

The additional leak on the success path

For DPL=3 segments the check passes and real descriptor data is returned to the guest — leaking the actual layout of host descriptor tables including base addresses, limits, and type information that belong to the VMM. Neither the failure path (no trap) nor the success path (no trap) gives the VMM an interception point. The guest gets either wrong data silently or real data it should not see.

LAR, LSL — load access rights / segment limit

TRAPS → VMM can emulate

✓ DPL=3 segment: succeeds, returns data

Check passes. Returns real descriptor data — leaking host descriptor layout to the guest.

DOES NOT TRAP → VMM has no hook

✗ DPL=0 segment: ZF=0, no result, no exception

Check fails. ZF cleared, register zeroed. No #GP. Instruction silently reports failure.

✗ VMOS assumes CPL=0 access to all segments

Guest OS queries kernel descriptors expecting success. Gets nothing back, no handler for this case.

✗ Category: behavior-sensitive (mode) + protection system

Both the returned value and whether the instruction "succeeds" depend on CPL. No VMM hook on either path.

The trappable path does not exist. Both outcomes — silent wrong result and silent leak — happen without a trap.

VERR, VERW — verify segment readable / writable (not used in xv6)

What the instructions do

VERR checks whether a segment can be read from the current privilege level and sets ZF=1 if yes, ZF=0 if no. VERW does the same for writability. Both use the same privilege check as LAR: $\max(\text{CPL}, \text{RPL}) \leq \text{DPL}$. Neither instruction raises an exception under any circumstance — the result is communicated solely through ZF. These are used by OS code to sanity-check that a segment is accessible before using it, typically when handling a user-supplied selector.

Why the output is consistently wrong under a VMM

A guest OS running at CPL=3 calls VERR on its own kernel segment descriptors (DPL=0) expecting ZF=1 (accessible). The check $\max(3, \text{RPL}) \leq 0$ fails: ZF=0 is returned — "not accessible." No exception, no trap. The VMOS was not written to handle ZF=0 on its own segments. It may refuse to use a segment it should use, fail to map memory correctly, or incorrectly accept a user-supplied selector it should reject — all producing silent data corruption.

The security implication: wrong access decisions with no VMM visibility

VERR and VERW are used for security decisions: "is this selector safe to use on behalf of this privilege level?" Inside a VMM the answers are consistently wrong for any privileged segment. An OS using VERR to validate a user-supplied pointer before dereferencing it at kernel privilege will reach a wrong answer. The VMM has no hook on the instruction in either outcome — it cannot correct the result or even know the check occurred.

VERR, VERW — verify segment readable / writable

TRAPS → VMM can emulate

✓ (no case raises an exception)

VERR and VERW never generate a #GP. They always execute to completion and report via ZF only.

DOES NOT TRAP → VMM has no hook

✗ DPL=0 segment: ZF=0, no exception

Check fails silently. ZF=0 tells VMOS the segment is inaccessible. No trap fires.

✗ DPL=3 segment: ZF=1, but data reflects real CPL

Reports accessible — but result reflects CPL=3 environment, not CPL=0 the VMOS assumes.

✗ VMOS cannot reach its own kernel segments

OS using VERR to check its kernel data segment receives ZF=0. No handler for this case exists.

VERR/VERW are behavior-sensitive: their output (ZF) depends on the real CPL, which differs from what the VMOS believes.

POP (segment registers) — loading DS, ES, FS, GS, SS (not used in xv6)

What the instruction does and the check it performs

POP loads a value from the top of the stack into a segment register (DS, ES, FS, GS, or SS — not CS). Before writing, the processor validates the selector: locates its descriptor in the GDT or LDT, checks that the segment type is compatible, and performs $\max(\text{CPL}, \text{RPL_of_selector}) \leq \text{DPL_of_descriptor}$. On failure a #GP is raised, which the VMM can catch and emulate. On success the register is loaded — and the loaded selector has the real CPL in bits 0–1.

The case that traps vs the case that does not

DPL=0 selector: $\max(\text{CPL}=3, \text{RPL}) > 0 \rightarrow \#GP$ fires $\rightarrow$ VMM catches and emulates. This case works. DPL=3 selector: check passes $\rightarrow$ register loaded $\rightarrow$ no trap $\rightarrow$ VMM has no hook. But the loaded selector has RPL=3 in bits 0–1. If the VMOS reads DS or SS back via PUSH or MOV, it sees CPL=3. A real kernel sees 0. The VMOS detects virtualisation or makes incorrect privilege decisions.

The structural problem: the success path is the problem

The DPL=0 failure case traps and is emulatable — you identified this correctly. The DPL=3 success case is the actual violation: the instruction executes without a trap, loads the register with the real CPL encoded in bits 0–1, and the VMM never finds out. Any OS code that reads a segment register and inspects bits 0–1 to check its privilege level — which is the hardware-intended use of RPL — will discover it is at ring 3.

POP (segment registers) — loading DS, ES, FS, GS, SS

TRAPS → VMM can emulate

✓ DPL=0 selector: #GP fires

$\max(\text{CPL}=3, \text{RPL}) > \text{DPL}=0$ — privilege check fails, general protection fault.
VMM catches and can emulate the correct load.

DOES NOT TRAP → VMM has no hook

✗ DPL=3 selector: succeeds, RPL=3 visible in register

Check passes. Register is loaded. RPL=3 in bits 0–1. VMOS reads DS or SS and sees CPL=3.

✗ Problem is the success path, not the failure path

The DPL=0 failure traps (emulatable). The DPL=3 success exposes real privilege. No trap, no VMM hook.

✗ Category: behavior-sensitive (mode) + protection system

Loaded register value encodes real CPL. VMOS can discover it is not at the expected privilege level.

You were right the DPL=0 failure traps. The problem is the DPL=3 success leaking CPL into the loaded selector.

PUSH CS, PUSH SS — pushing segment registers onto the stack (not used in xv6)

What the instruction does and why bits 0–1 matter

PUSH copies a segment register onto the stack. CS and SS have the real CPL stored in their bits 0–1 — this is how the Pentium hardware tracks the current ring. Every privilege transition writes the new CPL into CS. So PUSH CS always pushes the real current ring level onto the stack. There is no privilege check of any kind. PUSH executes identically at every CPL — it is a fundamental register-copy operation.

The direct CPL readout and its consequences

The VMOS executes PUSH CS then reads the value back (via POP AX or by examining the stack). It masks bits 0–1 with AND AX, 3. The result is 3. A real kernel at ring 0 sees 0. The VMOS concludes it is not running at the privilege level it expects. Depending on the OS this causes a halt, an error recovery path, or an assertion failure. Any OS that self-checks its privilege level this way — which is common practice — detects the VMM.

Why there is no mitigation

The VMM has no trap hook because PUSH has no privilege check. It cannot intercept the push and substitute a fake CS with RPL=0. The only solution is hardware-level interception (VT-x's VMX non-root mode causes PUSH CS in the guest to VM-exit, allowing the hypervisor to return a virtualised value) or binary translation (rewriting PUSH CS into a call that returns a shadow value). Neither is available on an unmodified Pentium.

PUSH CS, PUSH SS — exposing CPL via stack

TRAPS → VMM can emulate

✓ (no case traps)

PUSH is unconditionally non-privileged for all segment registers. No CPL check of any kind is performed.

DOES NOT TRAP → VMM has no hook

✗ CS bits 0–1 always contain the real CPL

PUSH CS pushes the full CS selector including RPL=3 in bits 0–1. VMOS reads the stack and sees CPL=3.

✗ SS bits 0–1 contain the real stack segment RPL

PUSH SS similarly exposes the real RPL. A kernel expecting RPL=0 finds RPL=3.

✗ Direct CPL leak, no mitigation possible

The VMM has no interception point. PUSH CS; POP AX; AND AX,3 is a reliable VMM detection sequence.

MOV reg, CS and MOV reg, SS have the same problem and are listed separately in the paper as the MOV (segment registers) instruction.

CALL, JMP (far, cross-privilege) — gate-based privilege transitions

How call gates work and why all kernel gates are unreachable

A far CALL to a different privilege level must pass through a call gate descriptor in the GDT. The gate's DPL controls who may invoke it: the check $\max(\text{CPL}, \text{RPL_of_gate_selector}) \leq \text{gate DPL}$ must pass. A conventional OS sets all system call gates with DPL=0 — only ring-0 callers should use them. Inside a VMM the VMOS runs at CPL=3, so $\max(3, \text{RPL}) \leq 0$ fails with #GP for every kernel gate. The VMOS simply cannot make system calls through the gate mechanism it was built for.

The DPL=3 success case and the privilege-elevation failure

If the VMOS attempts a far CALL through a gate with DPL=3, the privilege check passes. But a ring-3 to ring-3 gate call produces no privilege elevation — the destination code runs at CPL=3. The VMOS expects to have elevated to ring 0 and to be able to execute privileged instructions. Instead it is still at ring 3. The CS loaded at the destination has RPL=3 in bits 0–1, which the handler can inspect to detect the failure.

Task switches and the broader impact

JMP through a task gate initiates a full task switch: the processor loads a new TSS and performs extensive CPL/DPL/RPL checks at every stage. From CPL=3, access to CPL=0 task gates fails with #GP. Task switches that do succeed restore segment registers from the TSS, but CS will have RPL=3, and all the PUSH CS / POP problems apply to the restored context. The VMOS's scheduling and context-switching code is structurally broken.

CALL, JMP (far, cross-privilege) — gate access

TRAPS → VMM can emulate

✓ Gate DPL=0: #GP fires

$\max(\text{CPL}=3, \text{RPL}) > \text{gate DPL}=0$ — access check fails, #GP. VMM can catch and emulate.

✓ Gate DPL=3: succeeds but stays at CPL=3

Check passes. Control transfers. But no privilege elevation — destination runs at CPL=3, not CPL=0.

DOES NOT TRAP → VMM has no hook

✗ VMOS cannot make privileged cross-ring calls

All kernel call gates have DPL=0. VMOS cannot use them at CPL=3. Its system call architecture is broken.

✗ Successful same-level call exposes CPL

DPL=3 far call succeeds but CS at destination has RPL=3 — VMOS detects it did not elevate privilege.

✗ Task switches: CPL/DPL/RPL checks all fail

From CPL=3, access to CPL=0 task gates fails with #GP at every stage.

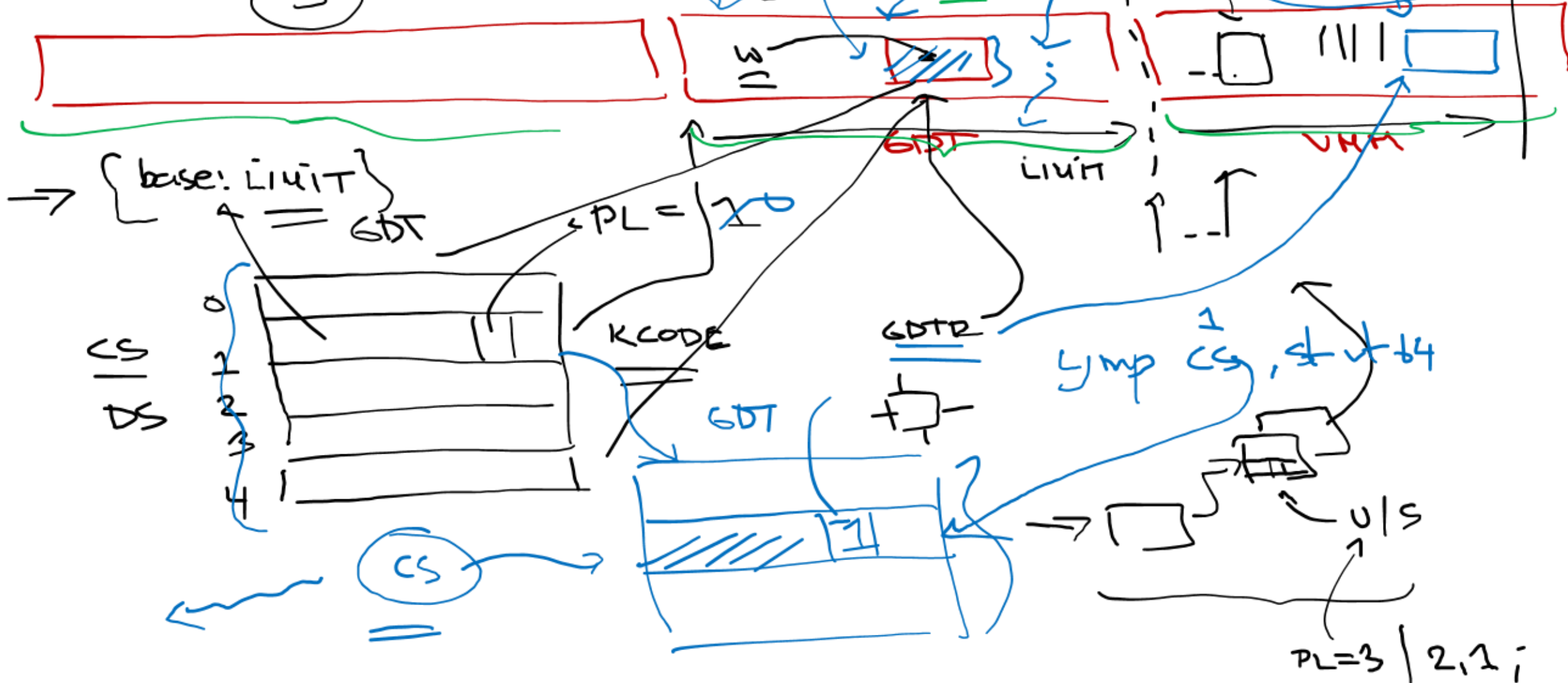
The trappable cases (DPL=0 gates) can be emulated. The structural problem: the VMOS simply cannot reach ring-0 gates at all.

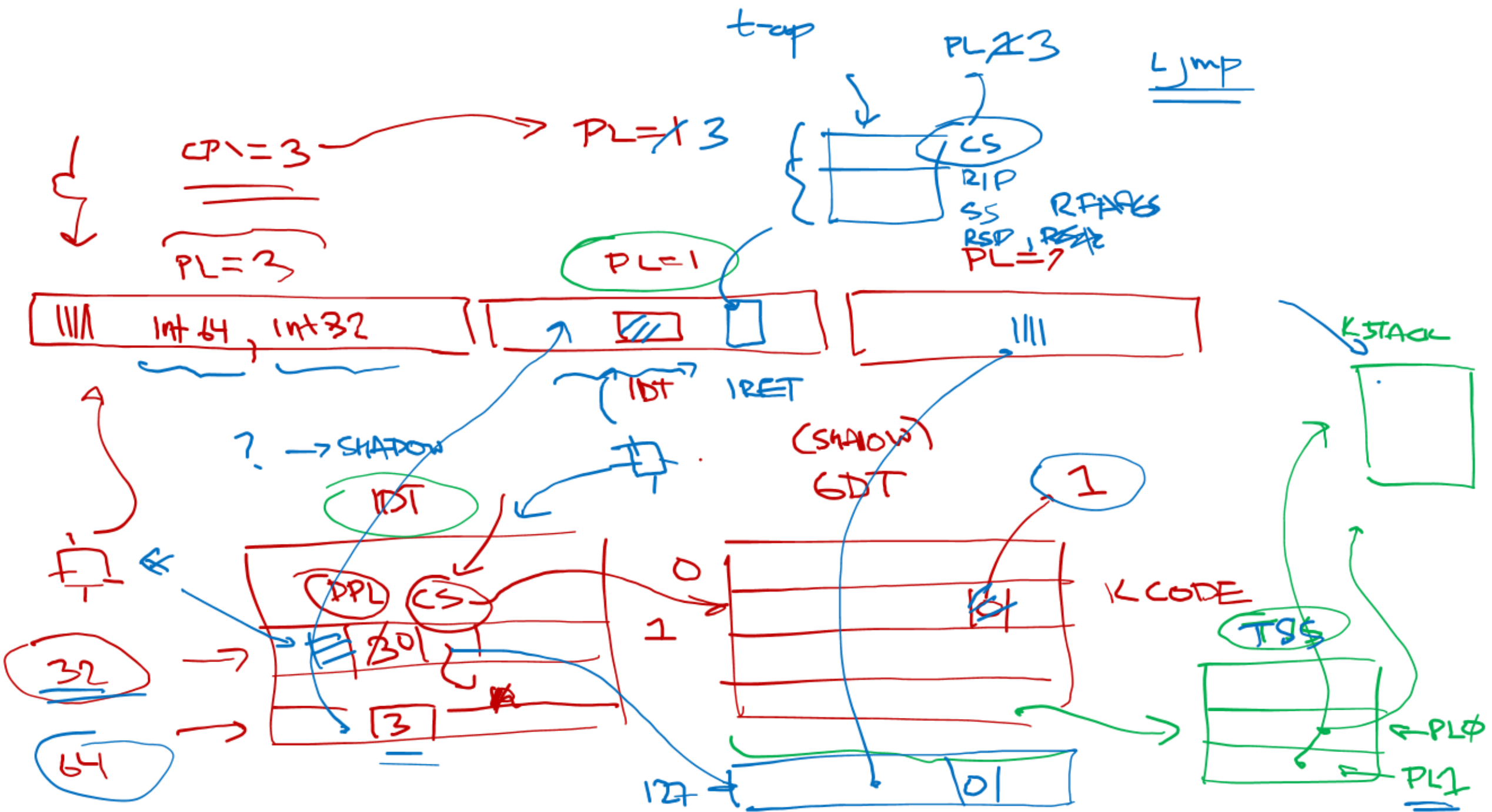
XEN 32bit → Intel → 64bit

3
AND

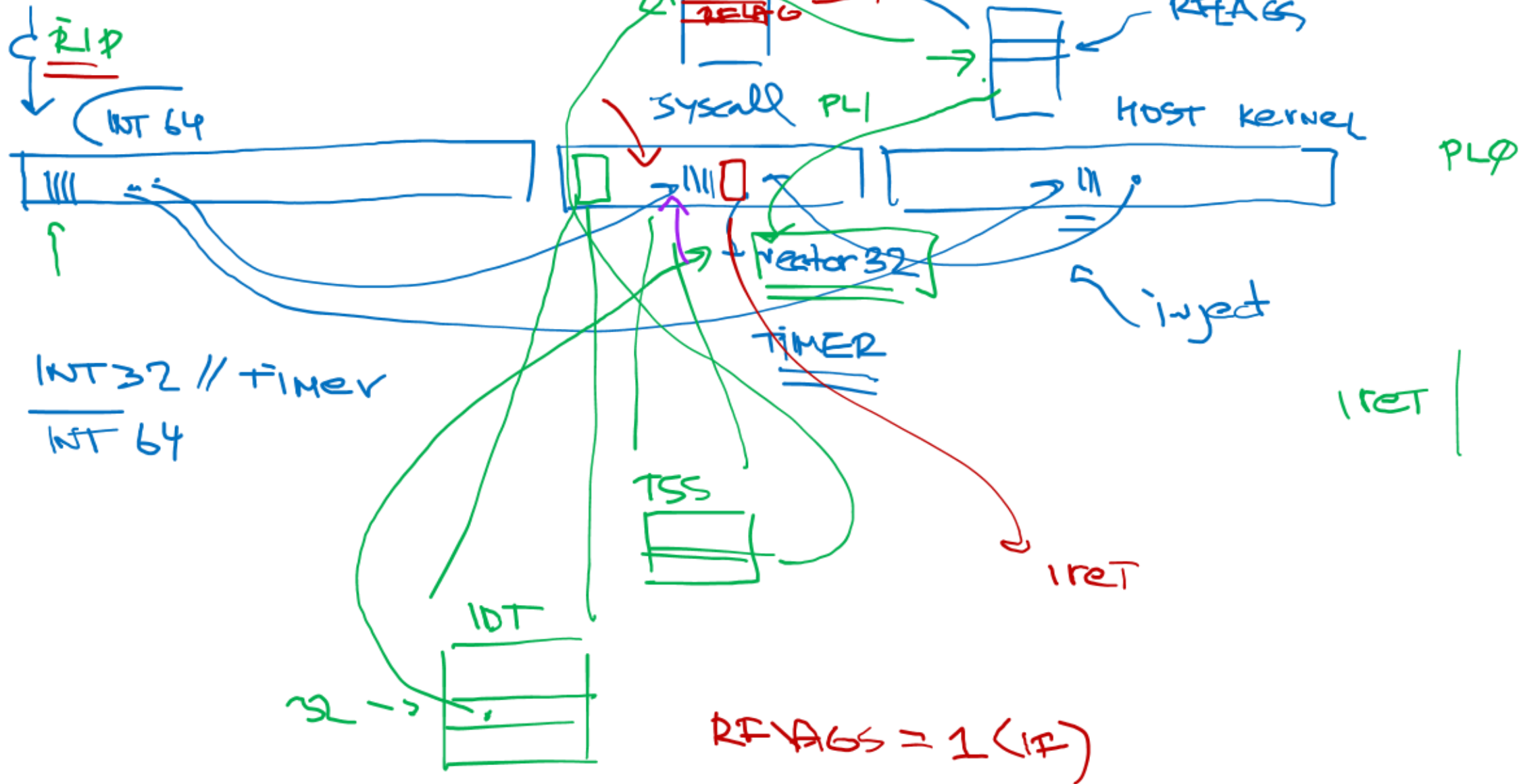
LDGDT
GUEST KERNEL
1
PL=1

SHADOW GDT





IF = 1



RFLAGS = 1 (IF)

Far JMP in xv6 — CS.RPL reveals real CPL, breaking user/kernel detection

xv6 source — entry.S + trap.c

```
// entry.S:71 — activate 64-bit long mode
lgdt (V2P_WO(gdt64desc)) // <- privileged, VMM intercepts
                        // shadow GDT: kernel DPL 0->1
ljmp $(SEG_KCODE<<3), // <- NOT privileged, silent
    $(V2P_WO(start64)) // CPL=1, DPL=1 -> succeeds
                        // CS.RPL = 1 (not 0)

// trap.c -- unexpected trap handler
if (myproc() == 0 || // no current process?
    (tf->cs & 3) == 0) { // <- kernel mode check
    panic("trap"); // CPL=0 native -> panic
} // CPL=1 VMM -> FALSE!
myproc()->killed = 1; // <- kernel bug silently
                    // kills user process
```

Two cases under virtualization

① Fault in scheduler — `myproc() == 0`

The first condition fires regardless of CS.RPL. xv6 panics correctly. The OR short-circuits before the CS check is even evaluated. This case survives the VMM. ✓ correct

② Fault in kernel — process is running

Native: CS.RPL=0 -> second condition TRUE -> panic
 VMM: CS.RPL=1 -> second condition FALSE
 myproc()!=0 -> first condition FALSE
falls through -> myproc()->killed = 1

A kernel bug (bad pointer, page fault) silently kills the current user process. Kernel continues in corrupted state -- no halt, no error.

ljmp is sensitive (reads GDT, sets CS.RPL from real CPL) but not privileged -- VMM never sees it. The fix -- patching CS.RPL to reflect virtual CPL=0 -- is impossible without hardware support.

xv6-64 entry.S:71 (every CPU, every boot) · trap.c default case (every unexpected kernel trap)

entry.S:71, trap.c:trap()

INT n — software interrupt

What the instruction does and its privilege check

INT n performs a software-initiated call to the interrupt or exception handler at vector n in the IDT. Like a far CALL through a gate, it checks the IDT gate's DPL against the current CPL. For kernel interrupt handlers (IDT gate DPL=0), the check $\max(\text{CPL}=3, \text{RPL}) \leq 0$ fails with #GP — the VMM can catch this. For DPL=3 gates, the instruction reaches the handler but at CPL=3: no privilege change occurs, and the handler cannot execute privileged instructions.

The EFLAGS push compounds the problem

INT pushes CS, EIP, and EFLAGS onto the stack before transferring control — the same EFLAGS values that PUSHF would expose. The handler receives the real IOPL and IF on its stack frame, not the virtualised values the VMOS believes it set. If the handler reads the saved EFLAGS to decide whether interrupts were enabled before the interrupt, it sees the wrong value and may re-enable or keep disabled interrupts incorrectly.

INT and IRET form a broken pair

The handler returns with IRET, which pops EIP, CS, and EFLAGS. The CS restore brings back RPL=3 (exposing CPL). The EFLAGS restore has the same silent-ignore behaviour as POPF at CPL=3: IF and IOPL are not updated. So INT pushes wrong EFLAGS state on entry and IRET silently fails to restore correct state on exit. The VMOS's entire interrupt handling subsystem operates on stale or incorrect flag state throughout.

INT n — software interrupt

TRAPS → VMM can emulate

✓ Gate DPL=0: #GP fires

$\max(\text{CPL}=3, \text{RPL_of_gate}) > 0$ — check fails, #GP. VMM can intercept and emulate the interrupt dispatch.

DOES NOT TRAP → VMM has no hook

✗ Gate DPL=3: executes but stays at CPL=3

Control reaches handler but no privilege change. Handler runs at CPL=3, cannot access ring-0 resources.

✗ EFLAGS pushed reflects real hardware state

Real IOPL and IF pushed to stack — same leak as PUSHF. Handler sees wrong interrupt state.

✗ Interrupt handlers assume CPL=0 entry

VMOS handlers were written for ring 0. At CPL=3 they cannot execute privileged instructions.

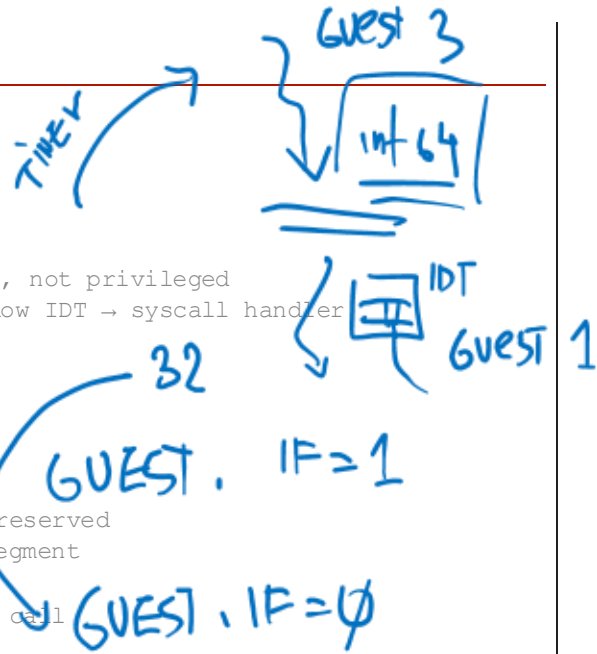
INT combines the call-gate privilege problem (CALL) with the EFLAGS exposure problem (PUSHF). Both compound inside the handler.

INT \$T_SYSCALL in xv6 — real EFLAGS pushed, IRETQ can't restore IF (but it works in xv6)

xv6 source — usys.S + trap.c

```
// usys.S — EVERY system call
#define SYSCALL(name) \
    .globl name; \
    name: \
        movq $SYS_##name, %rax; \
        int $T_SYSCALL; // ← sensitive, not privileged \
        ret // hits shadow IDT → syscall handler

// 21 system calls: fork, exec,
// read, write, open, close ...
// all go through int $T_SYSCALL
// trap.c — T_SYSCALL gate setup
SETGATE(idt[T_SYSCALL],
    1, // trap gate: IF preserved
    SEG_KCODE<<3, // → kernel code segment
    vectors[T_SYSCALL],
    DPL_USER); // DPL=3: user can call
```



What breaks under virtualization

① Entry — shadow IDT, DPL=3 gate

INT \$T_SYSCALL hits the shadow IDT. VMM correctly sets DPL=3 in the shadow gate so user space (CPL=3) can call it. Stack switch and register save happen correctly. The syscall handler runs at real CPL=1.

② Real EFLAGS on the trap frame

INT n pushes real RFLAGS to the trap frame — including the real IOPL (0, set by VMM) and real IF (1, interrupts were on in user space). The pushed IF=1 is correct. The problem is what happens on the way back.

③ Return — IRETQ silently drops IF

The syscall handler returns via iretq. At real CPL=1 with IOPL=0, IRETQ cannot restore IF from the trap frame. User process resumes with IF=0. After the very first system call, no more timer preemption — the process runs forever.

INT \$T_SYSCALL is only half the story — entry works, return is broken. Every one of the 21 xv6 system calls (fork, exec, read, write...) leaves the calling process with IF=0 on return. Multiplied across all processes, interrupt delivery collapses system-wide.

RET, IRET, IRETD — inter-privilege returns

What happens during an inter-privilege return

A far RET or IRET pops CS and EIP from the stack. If the RPL of the popped CS is greater than the current CPL, the processor treats this as a return to a less-privileged level. It additionally pops SS and ESP to restore the caller's stack. Then it checks DS, ES, FS, and GS: if any reference a segment whose DPL is less than the new CPL, the processor zeroes that register. This clearing is not an exception — it is a silent, defined behaviour that happens inside the instruction.

Why the segment register zeroing breaks the VMOS

Inside a VMM the VMOS runs at CPL=3 in both "kernel" and "user" contexts. When it executes what it believes is a kernel-to-user return, both sides are actually at CPL=3. The popped CS has RPL=3, which the processor reads as the new CPL=3. It then checks the segment registers against CPL=3 and zeroes any that point to DPL=0 segments — which includes the VMOS's kernel data segment (DS). The VMOS next tries to use DS, finds it zero, and faults in a way its exception handlers were not written to handle.

IRET additionally fails to restore EFLAGS

IRET pops EFLAGS from the stack (the value pushed when the original interrupt or INT fired). At CPL=3 the EFLAGS restore has the same silent-ignore behaviour as POPF: IF, IOPL, VIF, VIP, and VM are not updated. So IRET compounds two silent failures: segment registers are zeroed (no trap), and the interrupt flag is not restored (no trap). The VMOS returns from its interrupt handler with wrong segment state and wrong interrupt state, neither of which generates a VMM-visible event.

IOPL = 0
CPL = 1



IRETQ in xv6 — silently fails to restore IF on every trap return

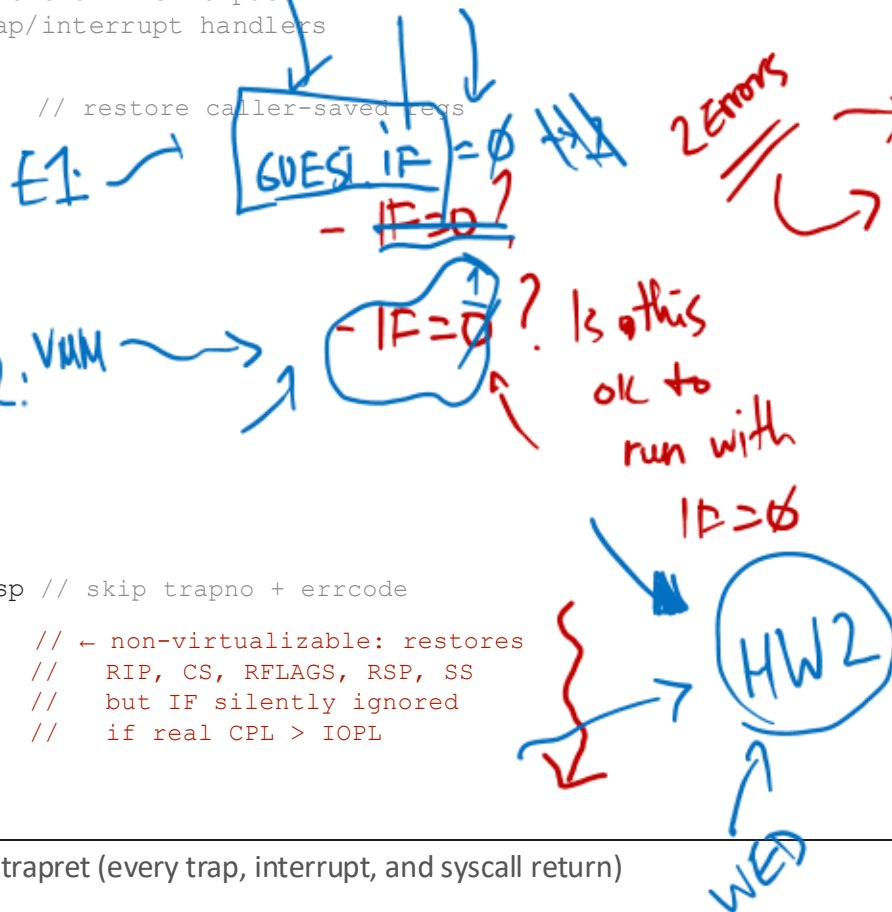
xv6 source — trapasm.S

```
// trapasm.S — the ONLY exit path
// from ALL trap/interrupt handlers
```

trapret:

```
popq %rax    // restore caller-saved regs
popq %rcx
popq %rdx
popq %rsi
popq %rdi
popq %r8
popq %r9
popq %r10
popq %r11
popq %rbx
popq %rbp
popq %r12
popq %r13
popq %r14
popq %r15
addq $16, %rsp // skip trapno + errcode
```

```
iretq        // ← non-virtualizable: restores
             // RIP, CS, RFLAGS, RSP, SS
             // but IF silently ignored
             // if real CPL > IOPL
```



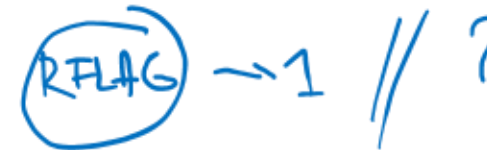
Two failure scenarios under virtualization

① User-space return: CPL=1 → 3

Saved RFLAGS.IF=1 on trap frame (user was running with interrupts on). IRETQ at real CPL=1 with IOPL=0 silently ignores IF. User process resumes with IF=0 — timer ticks never preempt it, device interrupts are lost.

② Kernel nested-trap return: CPL=1 → 1

scheduler() calls sti() — runs with IF=1. Timer fires mid-scheduler, handler returns via IRETQ. Saved RFLAGS has IF=1, but IRETQ at CPL=1 cannot restore it. Scheduler resumes with IF=0. Timer interrupts stop firing on this CPU. Scheduling halts silently.



Unlike CLI/STI (privileged — VMM intercepts), IRETQ executes without trapping. By the time the VMM re-enters, IF is already wrong. Nothing to fix up.

xv6-64 trapasm.S:trapret (every trap, interrupt, and syscall return)

xv6-64 trapasm.S:trapret (every trap, interrupt, and syscall return)

IRETQ in xv6 — cannot restore virtual-IF; VMM must keep real IF = 1 (actually works in xv6)

xv6 source — trapasm.S

```
// trapasm.S — the ONLY exit path
// from ALL trap/interrupt handlers

trapret:
    popq %rax      // restore caller-saved regs
    popq %rcx
    popq %rdx
    popq %rsi
    popq %rdi
    popq %r8
    popq %r9
    popq %r10
    popq %r11
    popq %rbx
    popq %rbp
    popq %r12
    popq %r13
    popq %r14
    popq %r15
    addq $16, %rsp // skip trapno + errcode

    iretq          // ← non-virtualizable: restores
                  //   RIP, CS, RFLAGS, RSP, SS
                  //   but IF silently ignored
                  //   if real CPL > IOPL
```

xv6-64 trapasm.S:trapret (every trap, interrupt, and syscall return)

xv6-64 trapasm.S:trapret (every trap, interrupt, and syscall return)

Corrected: failure is about virtual-IF, not real IF

① VMM isolation invariant: real IF = 1 always

VMM cannot let real IF=0 — a guest looping with IF=0 would freeze the physical CPU. Timer fires → VMM handles at CPL=0, injects to guest via shadow IDT, real IF stays 1. IRETQ is non-virtualizable because it does not trap: VMM cannot update virtual-IF from the saved frame. The failure: kernel exits IRETQ with virtual-IF=0 and saved RFLAGS.IF=1 → VMM stops delivering timer interrupts to user space.

② In xv6: PUSHFQ failure accidentally masks the IRETQ failure

VMM tracks virtual-IF via CLI/STI traps. After IRETQ, VMM's virtual-IF = last CLI/STI value — IRETQ does not trap, so the saved frame's IF is ignored. In xv6: PUSHFQ reads real IF=1 (VMM keeps it 1) → intena=1 → popcli calls STI before every trapret → virtual-IF=1. Saved frame also has IF=1 — they match. PUSHFQ's wrong read accidentally compensates for IRETQ's silent drop.

IRETQ is non-virtualizable: modifies IF without trapping, so VMM cannot compensate. But VMM keeps real IF=1 (isolation invariant), so the breakage is virtual-IF staleness after IRETQ. In xv6, PUSHFQ reading real IF=1 causes STI before every return — the two bugs partially cancel each other out.

RET, IRET, IRETD — inter-privilege returns

TRAPS → VMM can emulate

✓ Segment registers inaccessible at new CPL: cleared

If DS/ES/FS/GS point to DPL < new CPL, processor zeroes them. Defined behaviour, not an exception.

DOES NOT TRAP → VMM has no hook

✗ No trap — registers silently zeroed

Clearing happens inside the RET instruction without raising any exception. No VMM interception point.

✗ VMOS finds its own registers unexpectedly zeroed

After return VMOS uses DS, finds it zero, faults in a way its handlers were not written to handle.

✗ IRET additionally restores wrong EFLAGS

At CPL=3, EFLAGS restore has same silent-ignore behaviour as POPF — IF, IOPL not updated.

RET/IRET compound two problems: silent register zeroing (no trap) and silent EFLAGS restore failure (same as POPF).

STR — store task register

What the instruction does and what is in the result

STR stores the segment selector from the Task Register (TR) into a general-purpose register or memory. The TR holds a selector pointing to the Task State Segment of the currently executing task. Like all selectors, the TR selector has an index into the GDT, a TI bit, and RPL in bits 0–1. On a real system running a kernel task, the TR selector has RPL=0. Inside a VMM, where the guest kernel is running as a CPL=3 task, the TR selector has RPL=3. STR has no privilege check and executes at any CPL.

Why the VMOS cannot correctly identify its own task

OS code uses STR to find the current task — to locate the TSS for per-task state updates, or to verify which task is scheduled. The VMOS calls STR expecting RPL=0. It gets RPL=3. Any code that uses the RPL field for comparison or privilege decisions reaches wrong conclusions. Code that compares the returned selector to an expected value will fail to match and take a fatal error path. Code that reads the TSS via the returned selector will address the wrong memory.

STR as a simple VMM detection primitive

STR is one of the most reliable VMM detection instructions available. The sequence STR AX; AND AX, 3 returns 0 on real hardware and 3 inside a VMM. No timing side channel, no exception, no privilege required. Any guest that performs this check will immediately know it is virtualised. Without hardware support (VT-x makes STR in non-root mode VM-exit, allowing the hypervisor to return a virtualised value) there is nothing the VMM can do to prevent it.

STR — store task register (not used in xv6)

TRAPS → VMM can emulate

✓ (no case traps)

STR is unconditionally non-privileged. It executes at any CPL without any privilege check.

DOES NOT TRAP → VMM has no hook

✗ Always stores real task register selector

TR selector has real RPL in bits 0–1. At CPL=3 this is 3. VMOS reads it and discovers it is not the ring-0 task.

✗ Breaks VMOS task management

OS code using STR to identify the current task expects RPL=0. Sees RPL=3, may corrupt task data structures.

✗ Category: behavior-sensitive (mode)

Stored value reflects real CPL. Like PUSH CS, it is a direct CPL readout with no possible VMM mitigation.

Like PUSH CS, STR is a direct CPL readout. Any guest OS that uses STR to sanity-check its privilege level detects virtualisation immediately.

MOV (segment registers) — store and load

The store form: unconditional CPL leak

MOV reg, CS and MOV reg, SS copy the full segment selector — including bits 0–1 which contain the real CPL — into a general-purpose register. There is no privilege check of any kind. This is identical in effect to PUSH CS / PUSH SS. Any code that executes MOV EAX, CS; AND EAX, 3 has a direct, immediate readout of the real current privilege level. On a real kernel the result is 0. Inside a VMM it is 3. There is no way to intercept this on an unmodified Pentium.

The load form: split behaviour

MOV SS, reg (or MOV DS/ES/FS/GS, reg) performs the same privilege check as POP: $\max(\text{CPL}, \text{RPL_of_selector}) \leq \text{DPL_of_descriptor}$. With a DPL=0 selector the check fails and #GP fires — the VMM can catch and emulate. With a DPL=3 selector the check passes and the register is loaded with RPL=3 in bits 0–1. The VMOS can read those bits back and see CPL=3. No trap on the success path, no VMM notification.

Two distinct violations in one opcode

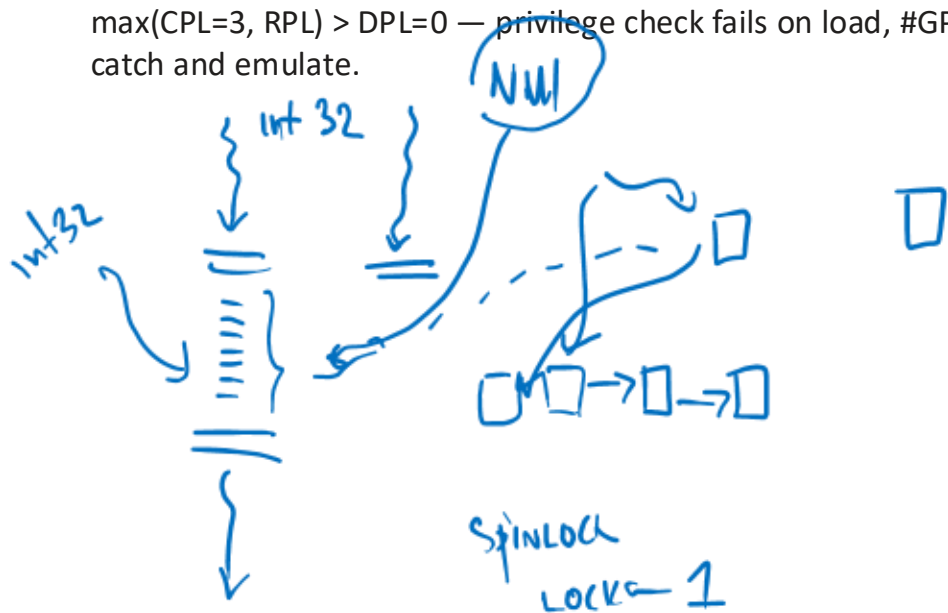
The paper lists MOV (segment registers) as having two separate sub-problems: the store variant violates Requirement 3B (reads the CPL, which is sensitive processor state, without trapping) and the load variant violates Requirement 3C (references the protection system in a CPL-dependent way with an untrapped success path). They share an opcode but represent fundamentally different failure modes — and together they account for two of the 17 problematic instruction slots.

MOV (segment registers) — store and load (not used in xv6, maybe during boot?)

TRAPS → VMM can emulate

✓ Loading SS with DPL=0 selector: #GP fires

$\max(\text{CPL}=3, \text{RPL}) > \text{DPL}=0$ — privilege check fails on load, #GP. VMM can catch and emulate.



DOES NOT TRAP → VMM has no hook

✗ Storing CS or SS: always leaks CPL

MOV reg, CS/SS always succeeds. Bits 0–1 of result are real CPL=3. Same direct leak as PUSH CS.

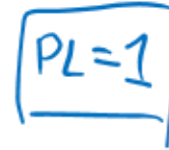
✗ Loading DS/ES/FS/GS with DPL=3: exposes RPL

Check passes. Register loaded with RPL=3 in bits 0–1. Guest reads CPL back from the loaded selector.

✗ Two distinct sub-problems in one instruction

Store variant: 3B violation (sensitive register read). Load variant: 3C violation (protection system reference).

MOV to/from segment registers is the instruction the paper identifies as having two separate violation modes under two different requirements.



Non-virtualizable instructions xv6-64 happens to avoid

Instruction → reason absent

MOV to %gs

loadgs() defined in x86.h but never called. The 64-bit port replaced GS-relative mycpu() with lapicid() + linear search through cpus[].

far CALL / far RET

Not used. Privilege transitions go through INT \$T_SYSCALL (user→kernel) and IRETQ (kernel→user). No call gates.

POPFQ

No writerflags() wrapper exists. Flag restoration always goes through IRETQ. POPFQ's IF-update failure is subsumed by the IRETQ bug.

~~SGDT~~ / ~~SIDT~~ / ~~SDT~~ / ~~STR~~

xv6 writes these tables (lgdt, lidt, ltr — all privileged, trap to VMM) but never reads them back with the unprivileged store variants.

SMSW

xv6 reads CR0 via rcr0() — MOV from CR0, which is privileged and traps. SMSW (the unprivileged CR0 read) is not used.

LAR / LSL / VERR / VERW

Segment descriptor inspection instructions. xv6 never queries descriptor access rights or limits at runtime.

ARPL

x86-32 only. Does not exist in 64-bit long mode. Irrelevant to xv6-64.

What this means

Lucky accidents, not safe design

xv6-64 avoids several non-virtualizable instructions not by design but because the 64-bit port simplified the kernel. The 32-bit original used GS for per-CPU data (MOV to %gs, broken under VMM). The 64-bit port rewrote mycpu() with lapicid() — accidentally sidestepping the problem.

The instructions that DO appear are the worst ones

PUSHFQ appears on every spinlock acquire and release. IRETQ appears on every single trap return. INT \$T_SYSCALL appears on every system call. Far JMP appears on every CPU at boot. These are not edge cases — they are the core kernel mechanisms. A VMM that ran xv6 would see these instructions thousands of times per second.

The 17 non-virtualizable instructions span the entire x86 protection model. Even a minimal OS like xv6 — with no call gates, no IOPL grants, no descriptor inspection — hits four of them constantly. A richer OS would hit more.

Robin & Irvine (2000): "Analysis of the Intel Pentium's Ability to Support a Secure Virtual Machine Monitor"
Intel VT-x (2005) added hardware VM exits for all 17 — the engineering fix Popek-Goldberg required

05

Pentium-based VMM security problems



Current "virtualization" techniques for the Pentium

Since the Pentium is not truly virtualizable, current VMMs must use "trickery." Each method must detect sensitive but unprivileged instructions before they are executed by a VM.

PURE EMULATION

Models the entire x86 instruction set in software — no instructions execute on real hardware. Approach used by Transmeta Crusoe (CSIM).
Disadvantage: significant performance degradation. Dynamic translation can help but will never reach Type I VMM performance.

OS/API EMULATION

Intercepts and emulates OS APIs using the underlying OS — used by Wine for Win32 on Linux/FreeBSD/Solaris. Achieves better performance than pure emulation. But only works for the specific OS family whose APIs are emulated, and is very complex.

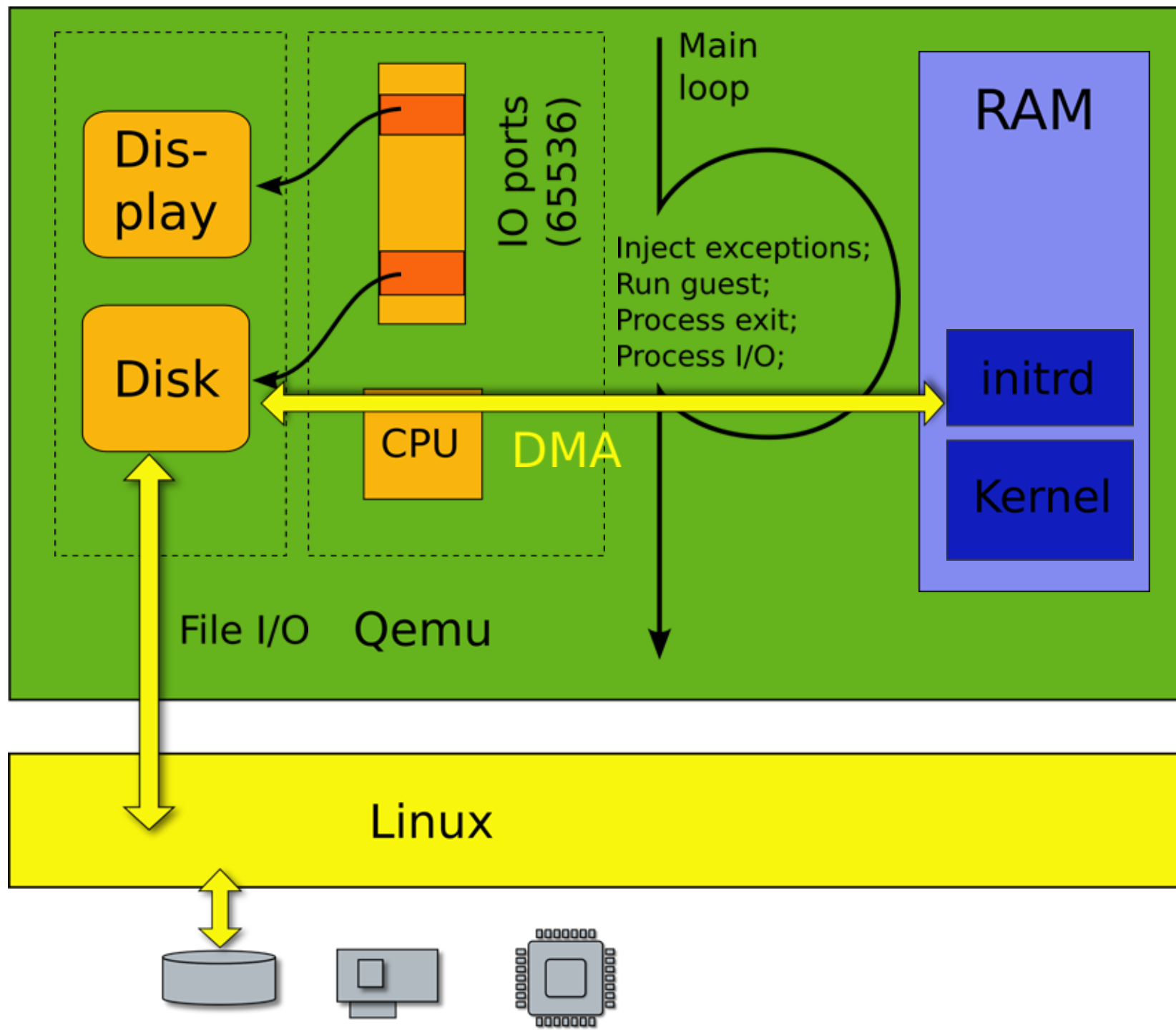
VIRTUALIZATION (with code scanning)

Runs code natively until a "problem instruction" or branch is encountered. Sets breakpoints before problem instructions. Used by FreeMWare (later Plex86). Complexity of this approach may make a high assurance VMM unachievable — all instructions must be checked before execution.

Solution space

- Parse the instruction stream and detect all sensitive instructions dynamically
 - Interpretation (BOCHS, JSLinux)
 - Binary translation (VMWare, QEMU)
- Change the operating system
 - Paravirtualization (Xen, L4, Denali, Hyper-V)
- Make all sensitive instructions privileged!
 - Hardware supported virtualization (Xen, KVM, VMWare)
 - Intel VT-x, AMD SVM

Basic blocks of a
virtual machine monitor:
QEMU example

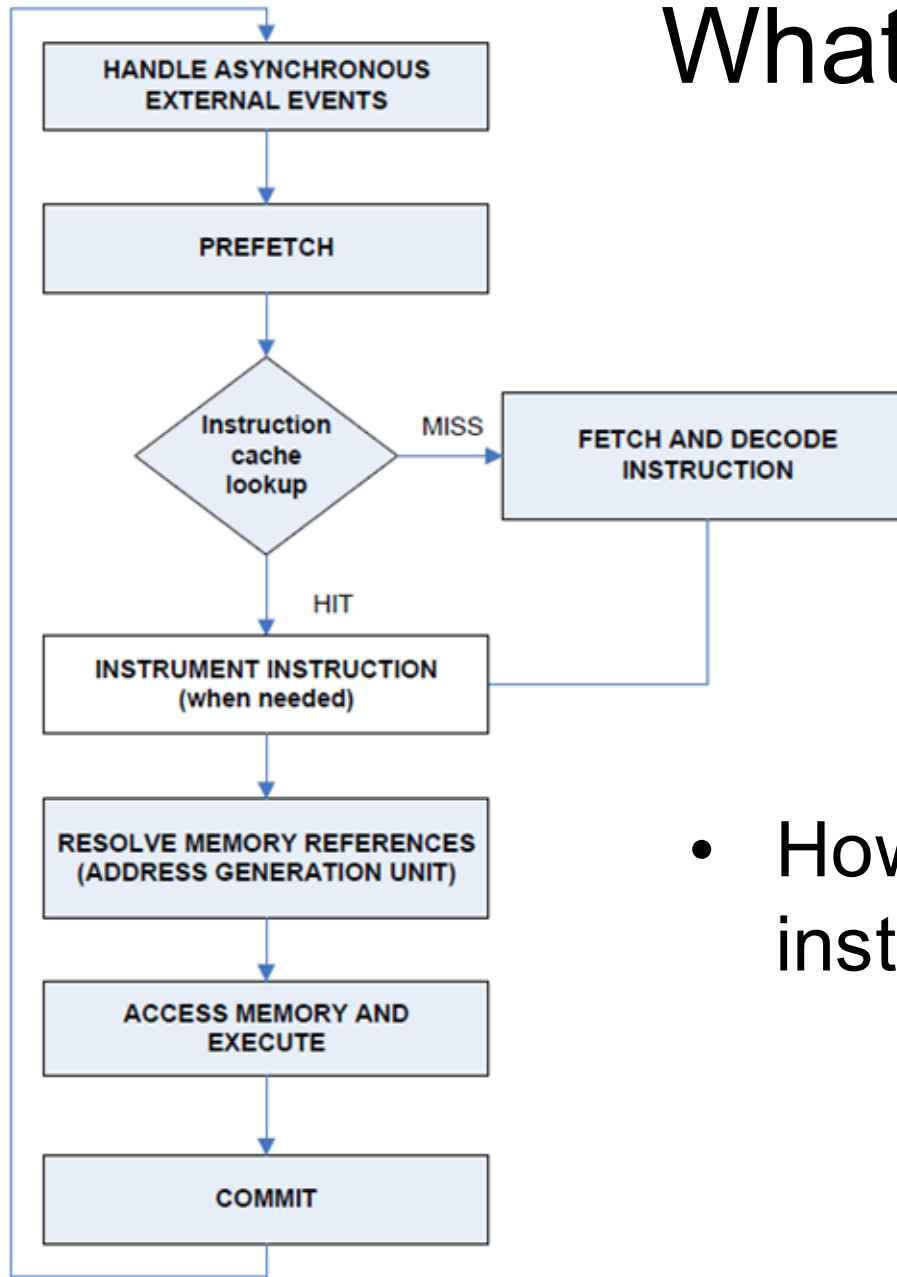


Interpreted execution:
BOCHS, JSLinux

What does it mean to run guest?

- Bochs internal emulation loop
- Similar to non-pipelined CPU like 8086

- How many cycles per instruction?



Binary translation:
VMWare/QEMU

```

int isPrime(int a) {
    for (int i = 2; i < a; i++) {
        if (a % i == 0) return 0;
    }
    return 1;
}

```

```

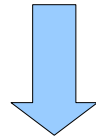
isPrime:  mov     %ecx, %edi ; %ecx = %edi (a)
          mov     %esi, $2   ; i = 2
          cmp     %esi, %ecx ; is i >= a?
          jge     prime      ; jump if yes
nexti:    mov     %eax, %ecx ; set %eax = a
          cdq                      ; sign-extend
          idiv    %esi        ; a % i
          test    %edx, %edx ; is remainder zero?
          jz      notPrime    ; jump if yes
          inc     %esi        ; i++
          cmp     %esi, %ecx ; is i >= a?
          jl      nexti       ; jump if no
prime:    mov     %eax, $1    ; return value in %eax
          ret
notPrime: xor     %eax, %eax ; %eax = 0
          ret

```

```

isPrime:  mov    %ecx, %edi ; %ecx = %edi (a)
          mov    %esi, $2  ; i = 2
          cmp    %esi, %ecx ; is i >= a?
          jge    prime     ; jump if yes
nexti:    mov    %eax, %ecx ; set %eax = a
          cdq                     ; sign-extend
          idiv   %esi        ; a % i
          test   %edx, %edx ; is remainder zero?
          jz     notPrime    ; jump if yes
          inc    %esi        ; i++
          cmp    %esi, %ecx ; is i >= a?
          jl     nexti      ; jump if no
prime:    mov    %eax, $1    ; return value in %eax
          ret
notPrime: xor    %eax, %eax ; %eax = 0
          ret

```



```

isPrime':  mov %ecx, %edi    ; IDENT
           mov %esi, $2
           cmp %esi, %ecx
           jge [takenAddr]  ; JCC
           jmp [fallthrAddr]

```

```

isPrime': *mov    %ecx, %edi    ; IDENT
          mov     %esi, $2
          cmp     %esi, %ecx
          jge     [takenAddr]  ; JCC
                                   ; fall-thru into next CCF
nexti':   *mov     %eax, %ecx    ; IDENT
          cdq
          idiv    %esi
          test    %edx, %edx
          jz      notPrime'     ; JCC
                                   ; fall-thru into next CCF
          *inc     %esi          ; IDENT
          cmp     %esi, %ecx
          jl      nexti'        ; JCC
          jmp     [fallthrAddr3]

notPrime': *xor     %eax, %eax    ; IDENT
          pop     %r11           ; RET
          mov     %gs:0xff39eb8(%rip), %rcx ; spill %rcx
          movzx   %ecx, %r11b
          jmp     %gs:0xfc7dde0(8*%rcx)

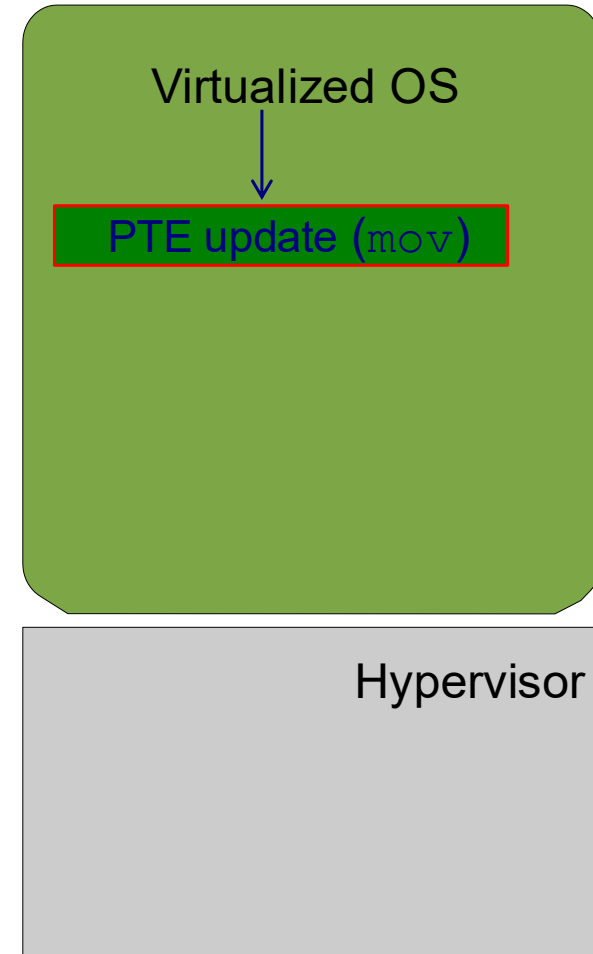
```

Paravirtualization:

Xen

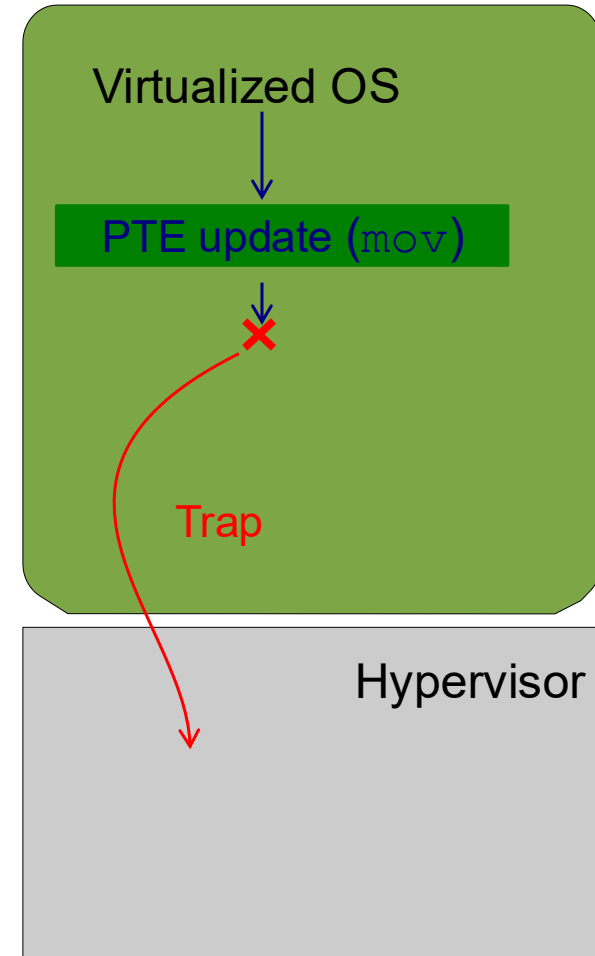
Full virtualization

- Complete illusion of physical hardware
- Trap all sensitive instructions
- Example: page table update



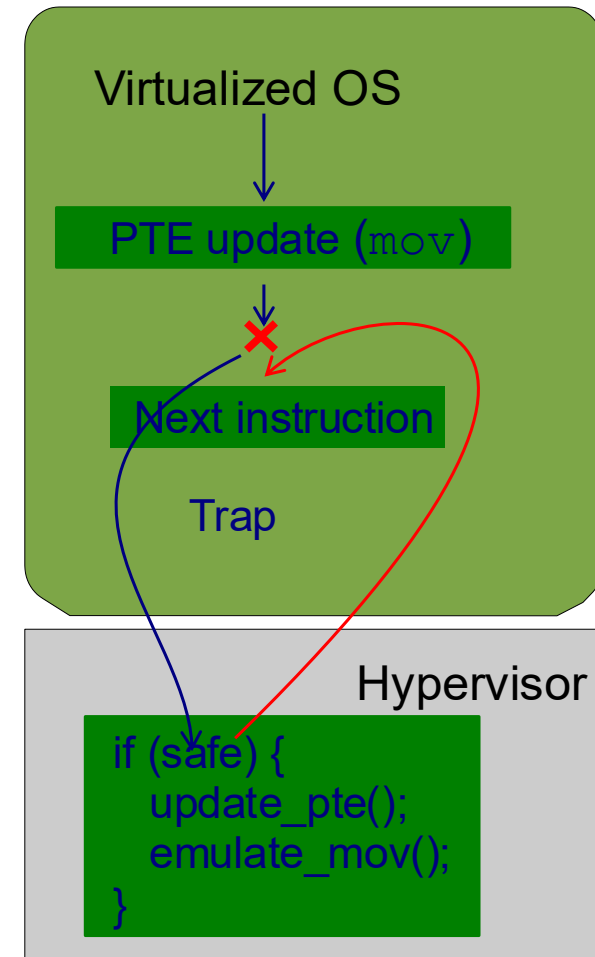
Full virtualization

- Complete illusion of physical hardware
- Trap all sensitive instructions
- Example: page table update



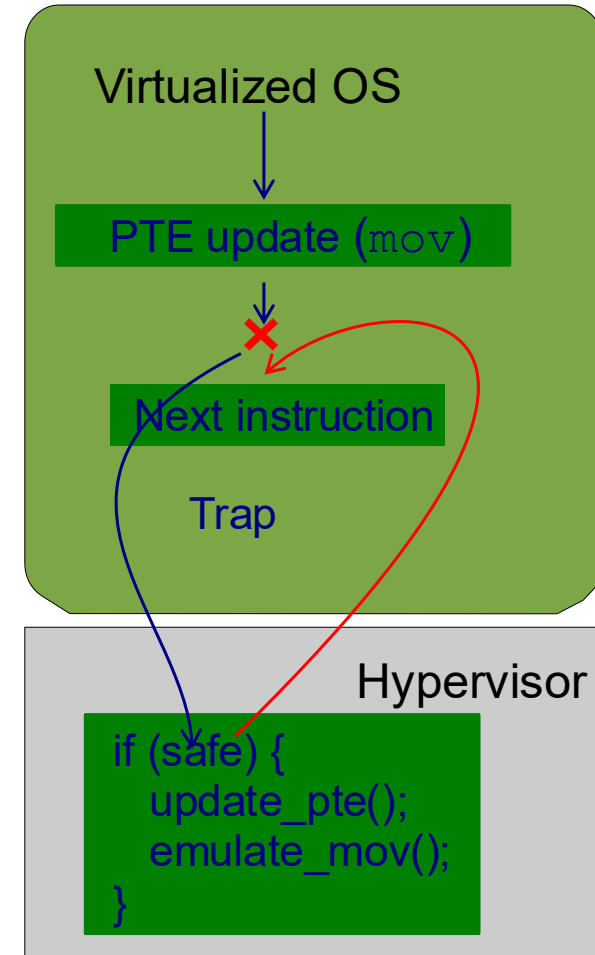
Full virtualization

- Complete illusion of physical hardware
- Trap all sensitive instructions
- Example: page table update



Performance problems

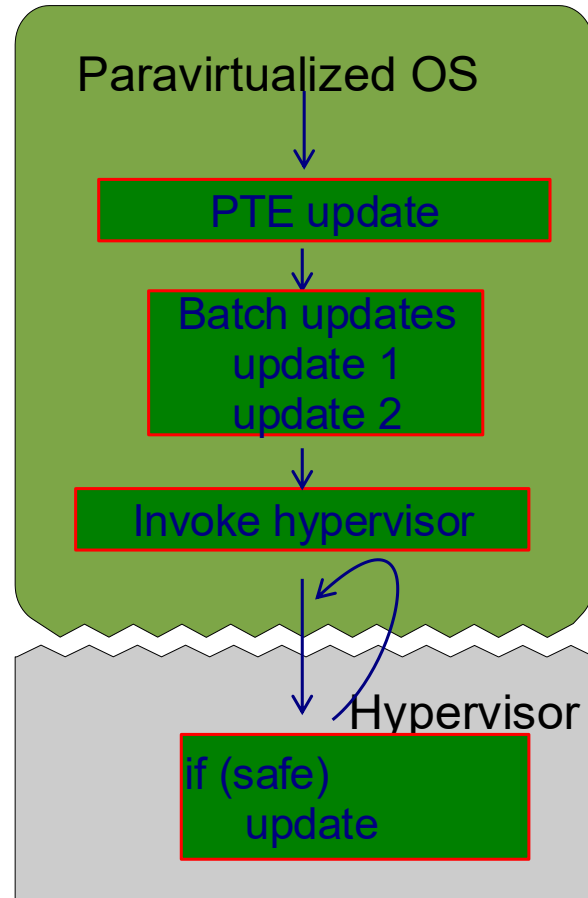
- Traps are slow
- Binary translation is faster
- For some events



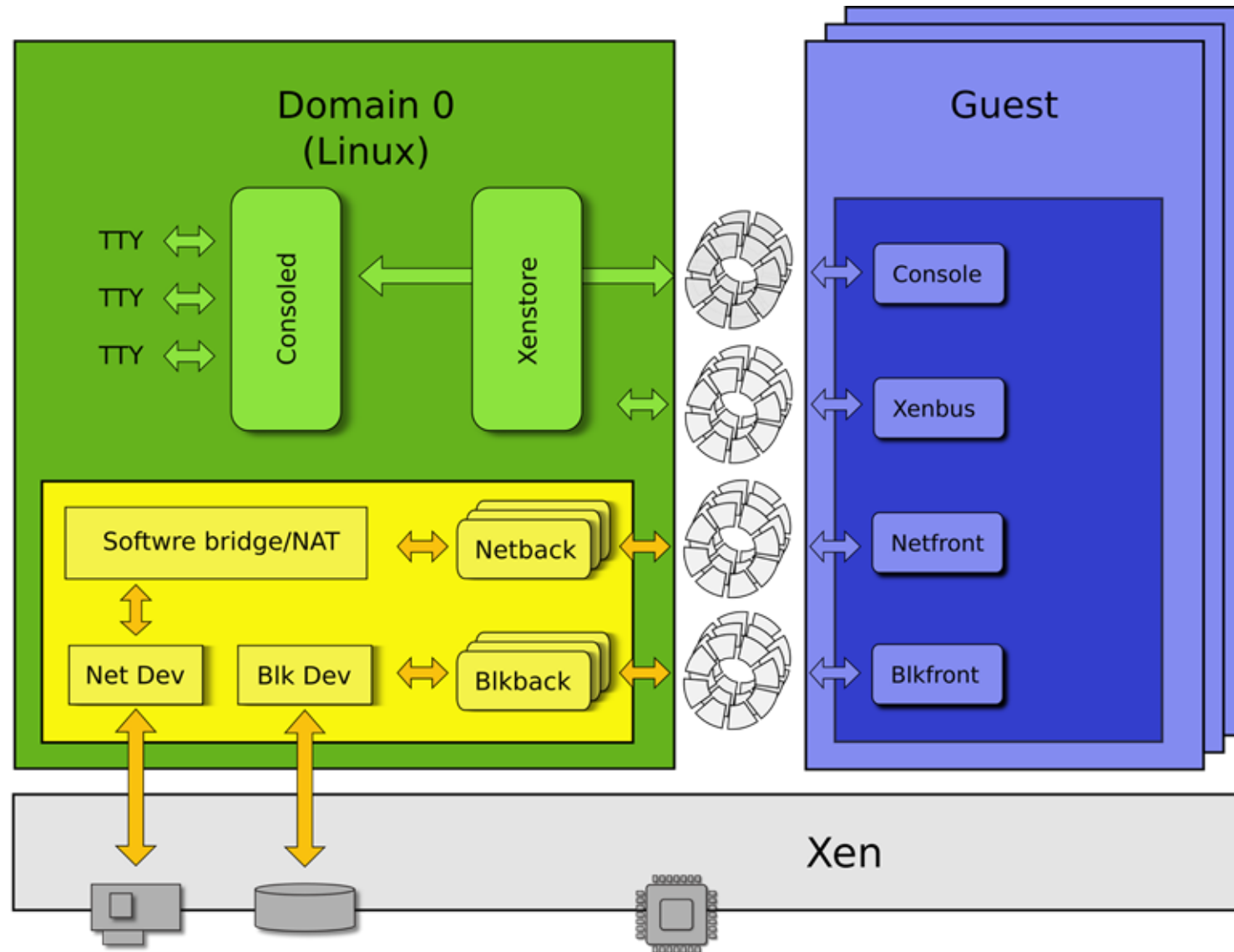
Paravirtualization

- No illusion of hardware
- Instead: paravirtualized interface
- Explicit hypervisor calls to update sensitive state
 - Page tables, interrupt flag
- But Guest OS needs porting
- Applications run natively in Ring 3

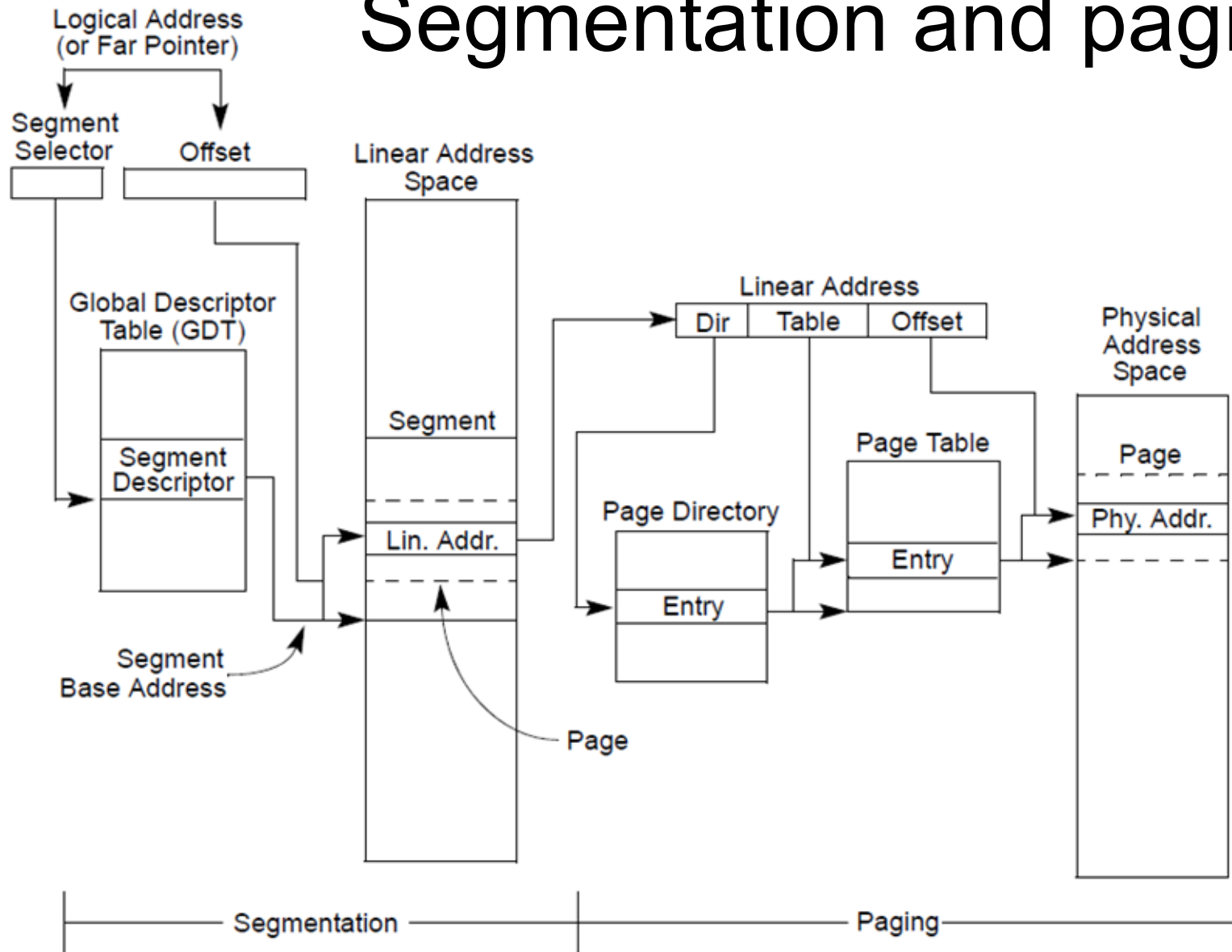
Paravirtualization



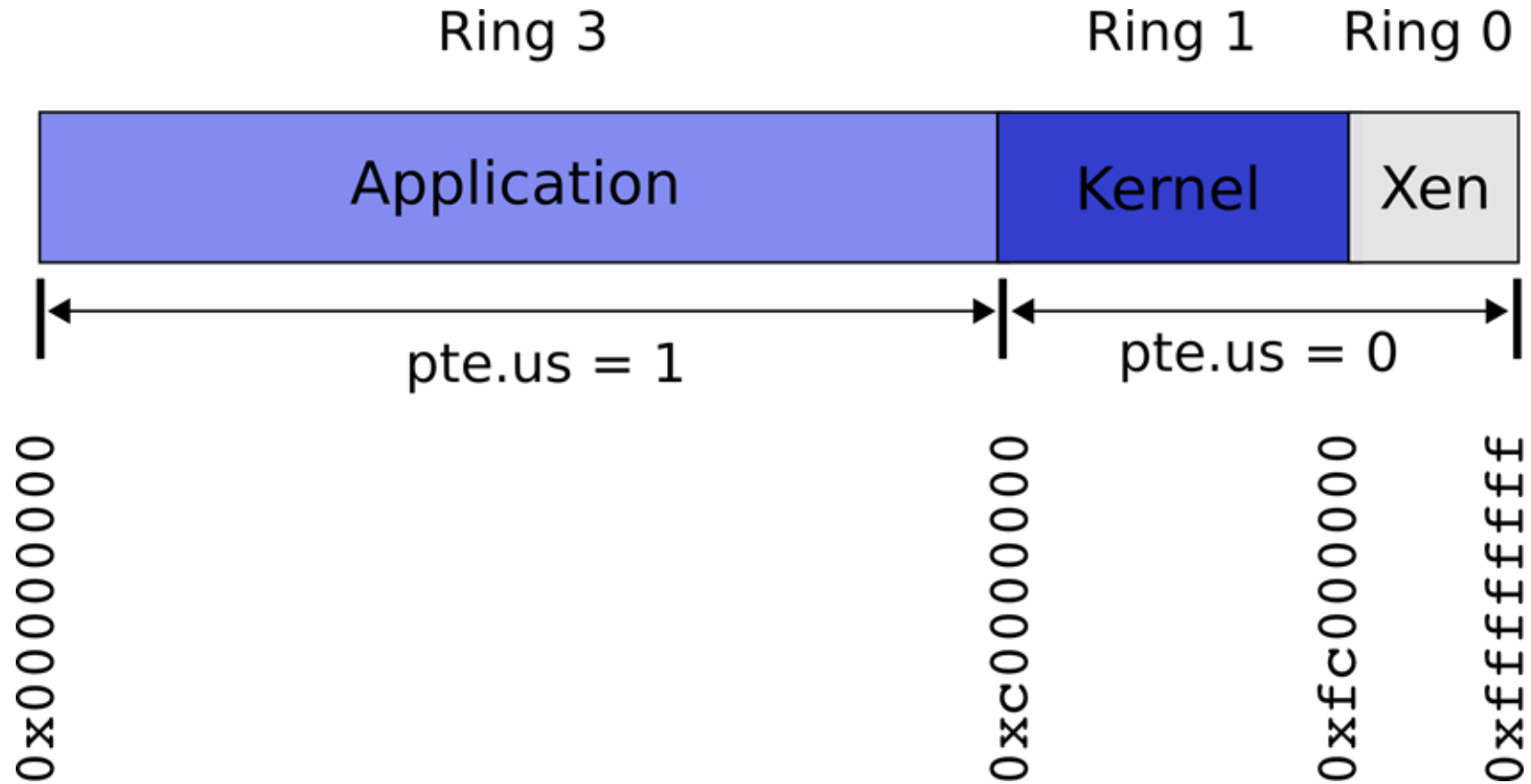
Xen



Segmentation and paging

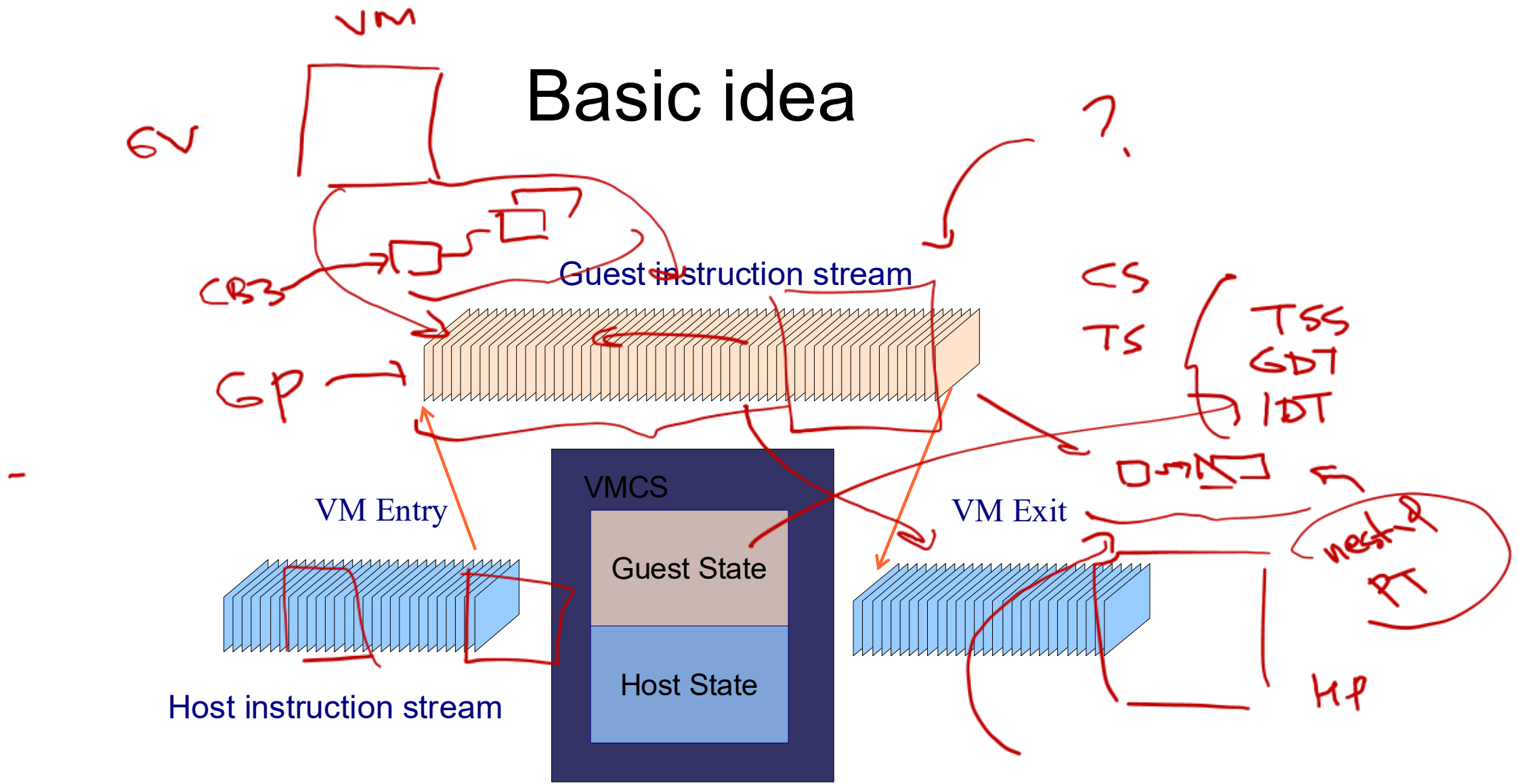


Hypervisor protection



Hardware support for virtualization:
KVM, Modern Xen, Modern Qemu

Basic idea





New mode of operation: VMX root

- VMX root operation
 - 4 privilege levels
- VMX non-root operation
 - 4 privilege levels as well, but unable to invoke VMX root instructions
 - Guest runs until it performs exception causing it to exit
- Rich set of exit events
 - Guest state and exit reason are stored in VMCS

Virtual machine control structure (VMCS)

- Guest State
 - Loaded on entries
 - Saved on exits
- Host State
 - Saved on entries
 - Loaded on exits
- Control fields
 - Execution control, exits control, entries control



Guest state

- Register state
- Non-register state
- Activity state:
 - active
 - inactive (HLT, Shutdown, wait for Startup IPI interprocessor interrupt)
- Interruptibility state

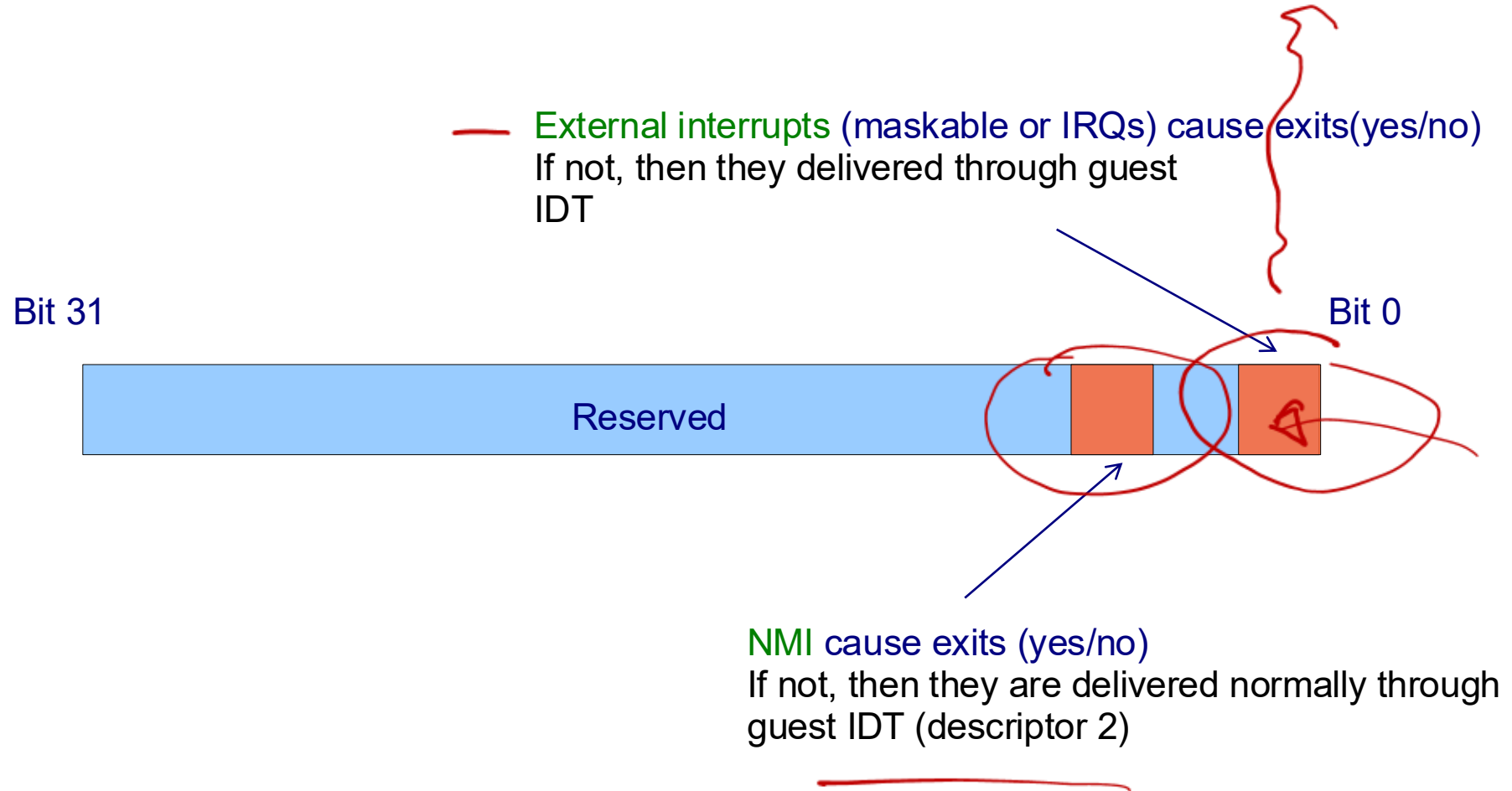
↪ IF

Host state

- Only register state
 - ALU registers,
 - also:
 - Base page table address (CR3)
 - Segment selectors
 - Global descriptors table
 - Interrupt descriptors table
- 

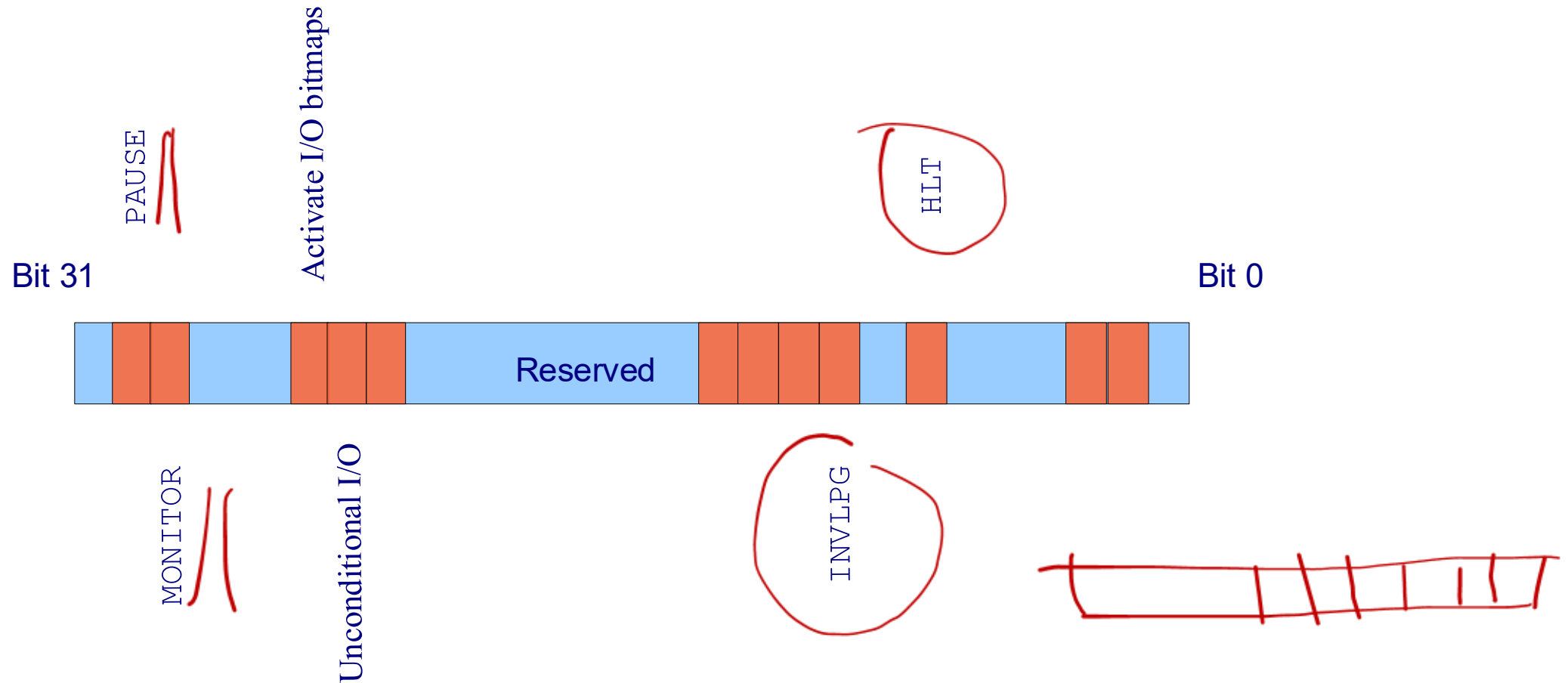
VM-execution controls

(asynchronous events control)



VM-execution controls

(synchronous events control, not all reasons are shown)

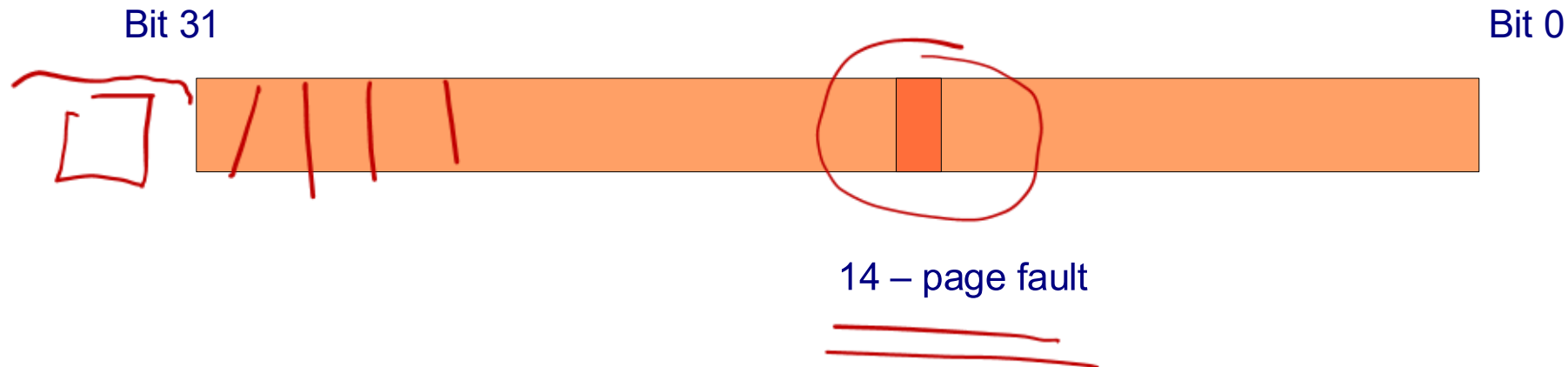


Exception bitmap

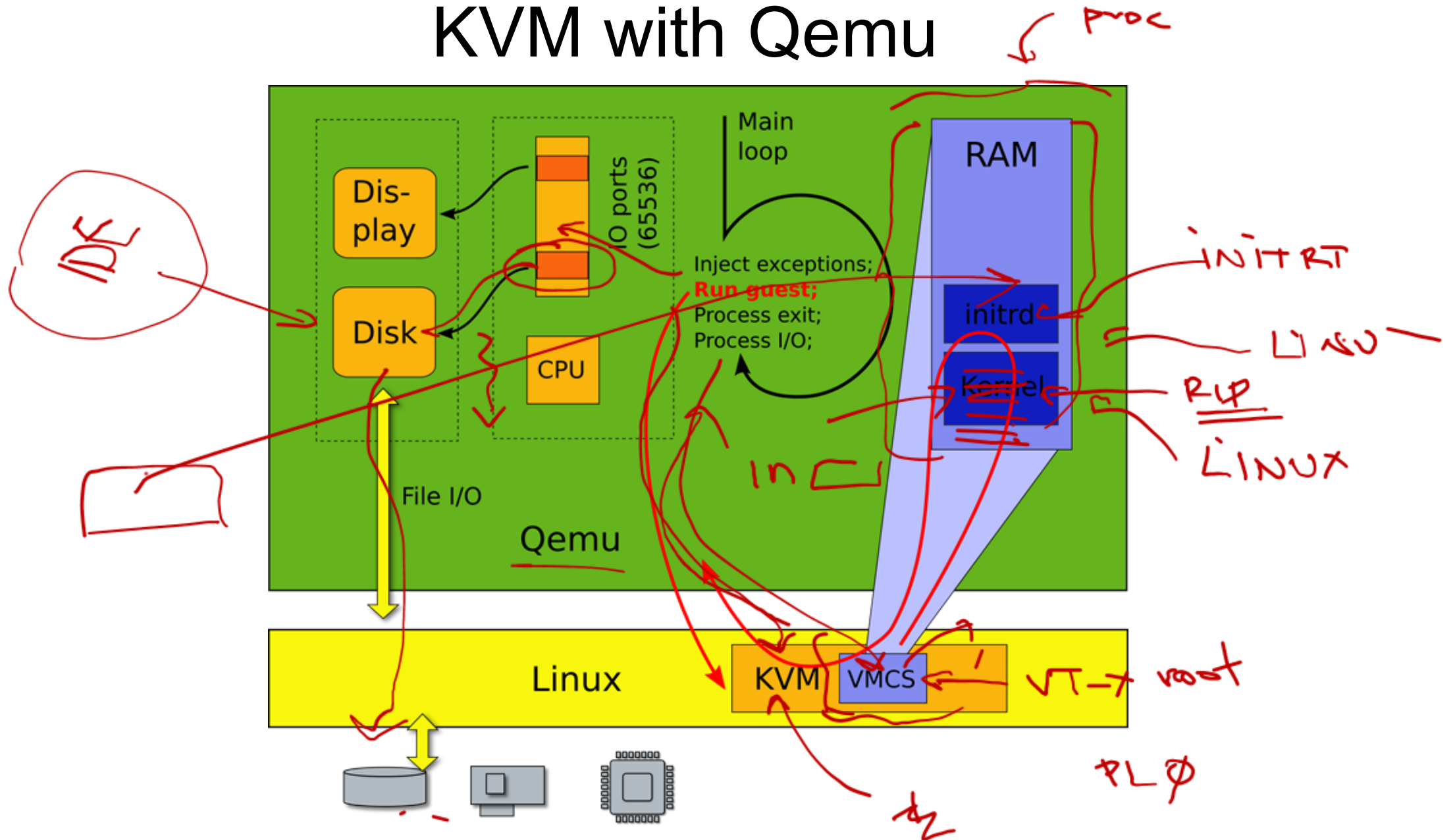
(one for each of 32 IA-32 exceptions)

- IA-32 defines 32 exception vectors (interrupts 0-31)
- Each of them is configured to cause or not VM-exit

0...>|



KVM with Qemu



06

A better approach and conclusions

Two paths to a virtualizable Pentium

PATH ONE

Change 17 instructions to privileged

Make all 17 sensitive, unprivileged instructions generate a general protection exception when executed at $CPL > 0$. All 0. All instructions trap naturally and the VMM can emulate them. Disadvantage: current operating systems that use these 17 instructions would now see unexpected traps — requires OS changes.

PATH TWO

Add a "trap on opcode" instruction

Add a new instruction that allows an OS to declare specific instructions that should be treated as if they were privileged. Makes virtualization possible without affecting current operating systems that do not invoke the new instruction. This is Goldberg's "trap on op-code" proposal [12].

The preferred approach for a high security VMM is a Type I VMM implemented as a microkernel — very small, meeting reference monitor verifiability requirements, and intercepting all low-level hardware functions from VMs.

Robin & Irvine — USENIX Security 2000

Conclusions

Seventeen instructions make the Pentium non-virtualizable

All 17 violate Requirement 3B (sensitive registers) or 3C (protection system references). They execute from user mode without generating traps, so a VMM can never intercept or simulate them.

Current Pentium VMMs cannot enforce critical security policies

"We conclude that current VMMs for the Intel architecture should not be used to enforce critical security policies." Implementing a high-assurance VMM as a Type II VMM on a generic commercial OS would not provide a high level of security.

The right approach: Type I VMM as microkernel

A small, formally verifiable Type I VMM microkernel provides isolation between VMs. It intercepts all hardware function attempts from VMs and controls all devices. Existing commercial OS and applications run unmodified in VMs.

Minor hardware modification would suffice

Either making the 17 instructions privileged, or adding a "trap on opcode" instruction, would make the Pentium fully virtualizable. The second option avoids breaking existing operating systems.

References

- Robin, J. S. and Irvine, C. E. Analysis of the Intel Pentium's ability to support a secure virtual machine monitor. USENIX Security Symposium, Denver, CO, August 2000.
- Popek, G. J. and Goldberg, R. P. Formal requirements for virtualizable third generation architectures. Communications of the ACM 17, 7 (July 1974), 412–421.
- Goldberg, R. P. Architectural Principles for Virtual Computer Systems. Ph.D. thesis, Harvard University, 1972.
- Karger, P. A. et al. A Retrospective on the VAX VMM Security Kernel. Transactions on Software Engineering, 17(11):1147–1165, November 1991.
- Hall, J. and Robinson, P. T. Virtualizing the VAX Architecture. ISCA, 1991.
- Intel Corporation. Intel Architecture Software Developer's Manual, Volumes 1 & 2. Santa Clara, CA, 1999.
- Bugnion, E. et al. Disco: Running Commodity Operating Systems on Scaleable Multiprocessors. ACM TOCS 15(4), November 1997.