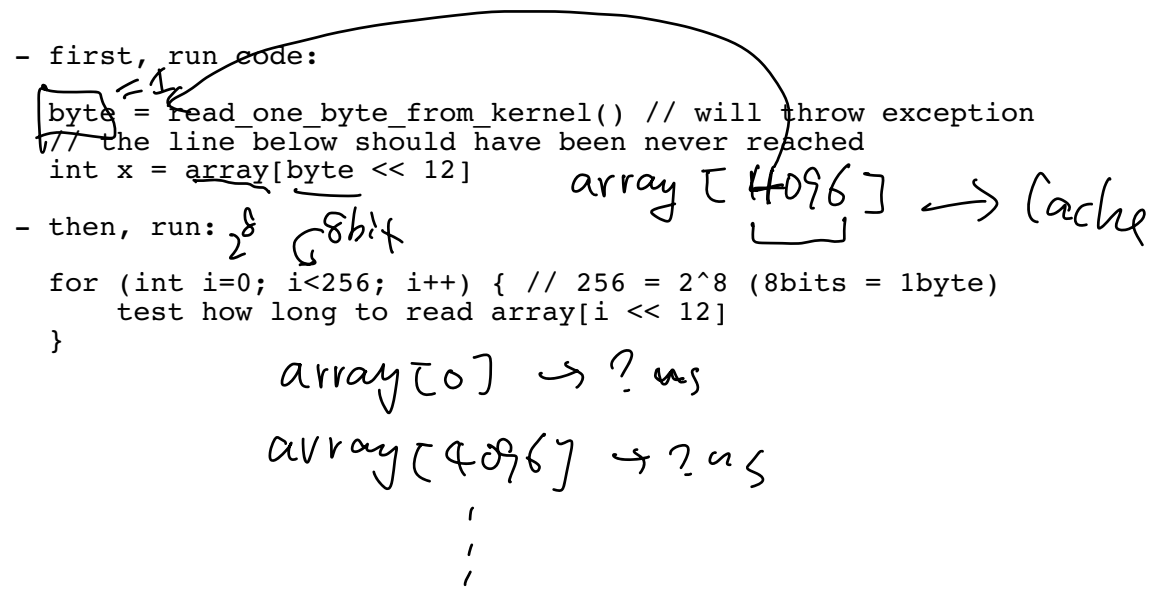
Meltdown and Spectre (2018).
https://meltdownattack.com/

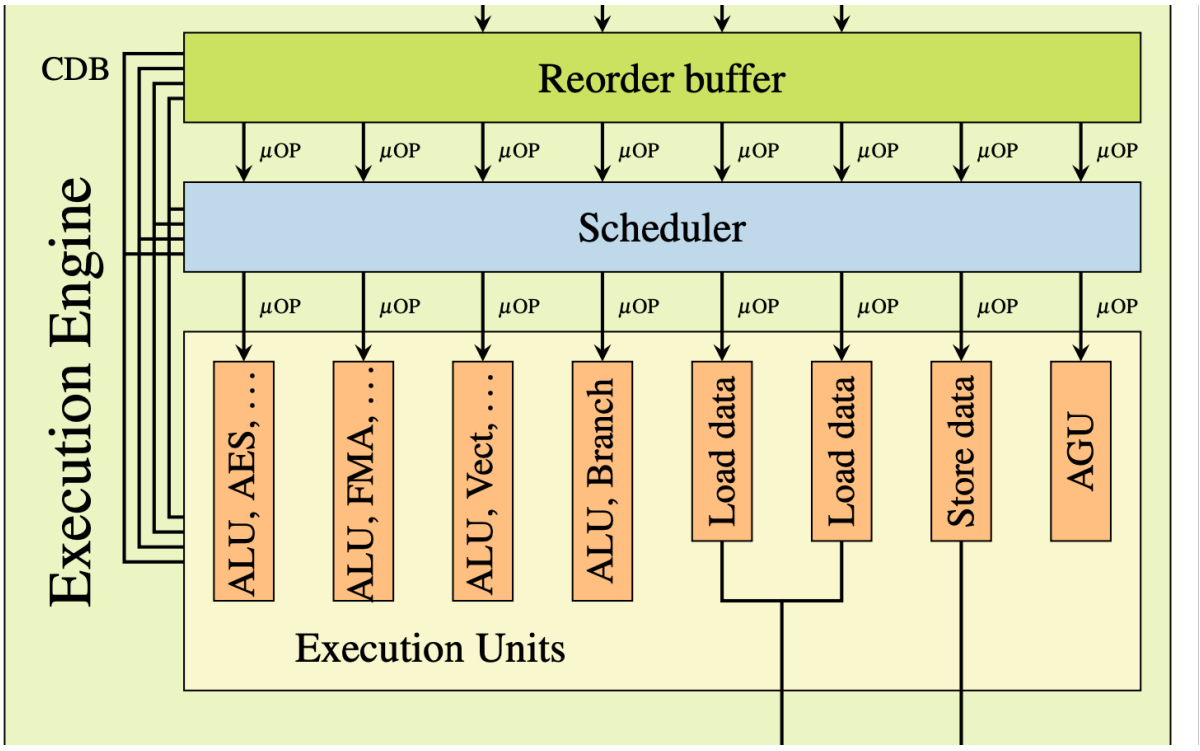
"""
Q: Am I affected by the vulnerability?
A: Most certainly, yes.

Q: Can I detect if someone has exploited Meltdown or Spectre against me?
A: Probably not. The exploitation does not leave any traces in
traditional log files.

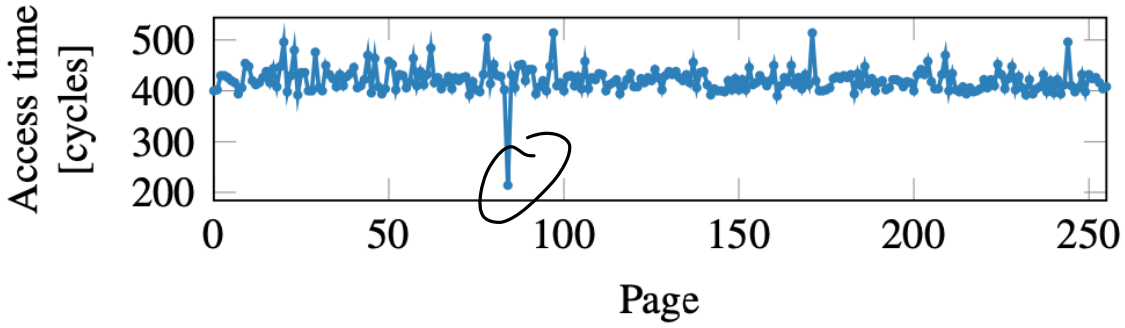
Q: What can be leaked?
A: If your system is affected, our proof-of-concept exploit can read
the memory content of your computer. This may include passwords
and sensitive data stored on the system.

Q: Has Meltdown or Spectre been abused in the wild?
A: We don't know.
"""



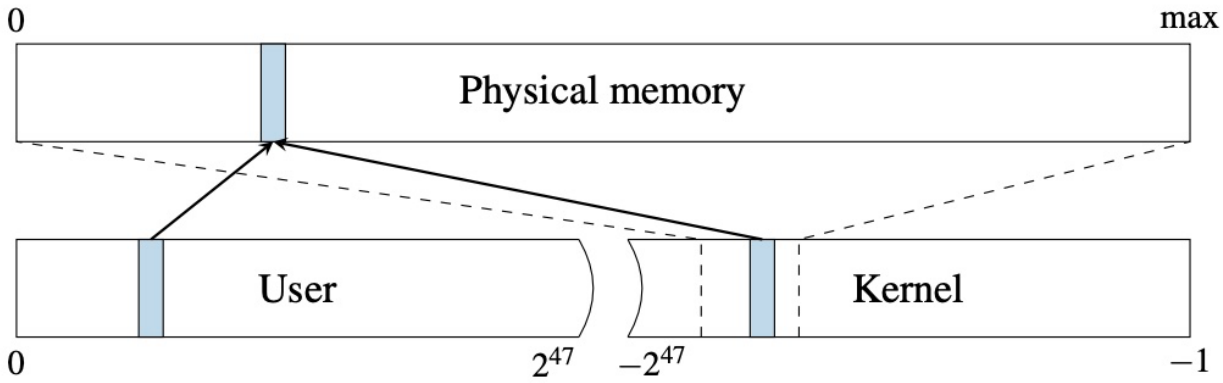


(Partial view of CPU internals: execution engine)

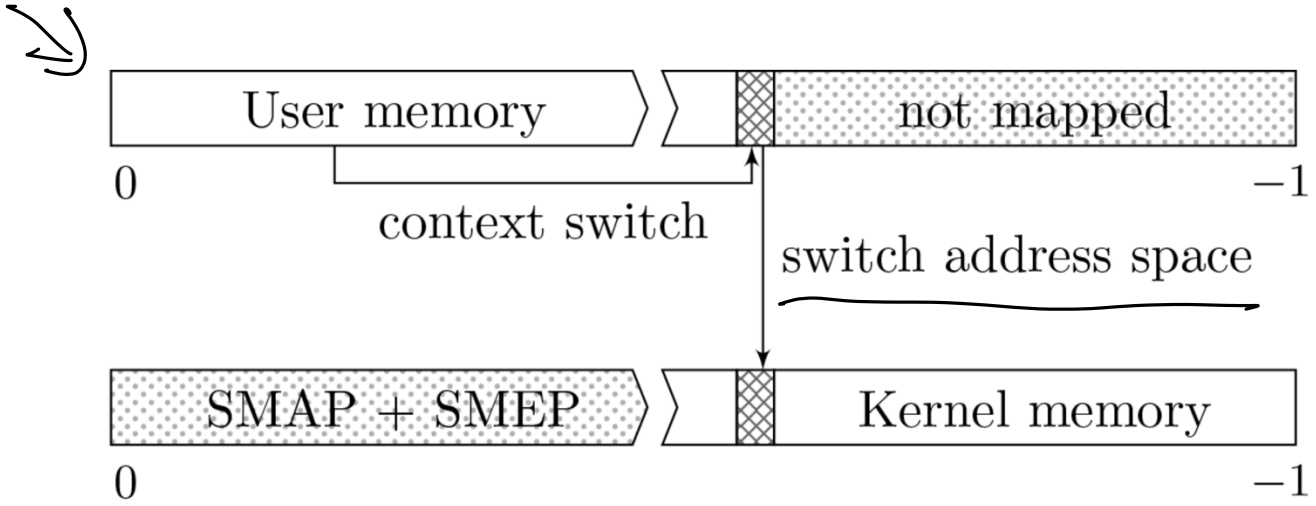


(Meltdown last step: checking which page has been cached)

Figures borrowed from Meltdown paper.

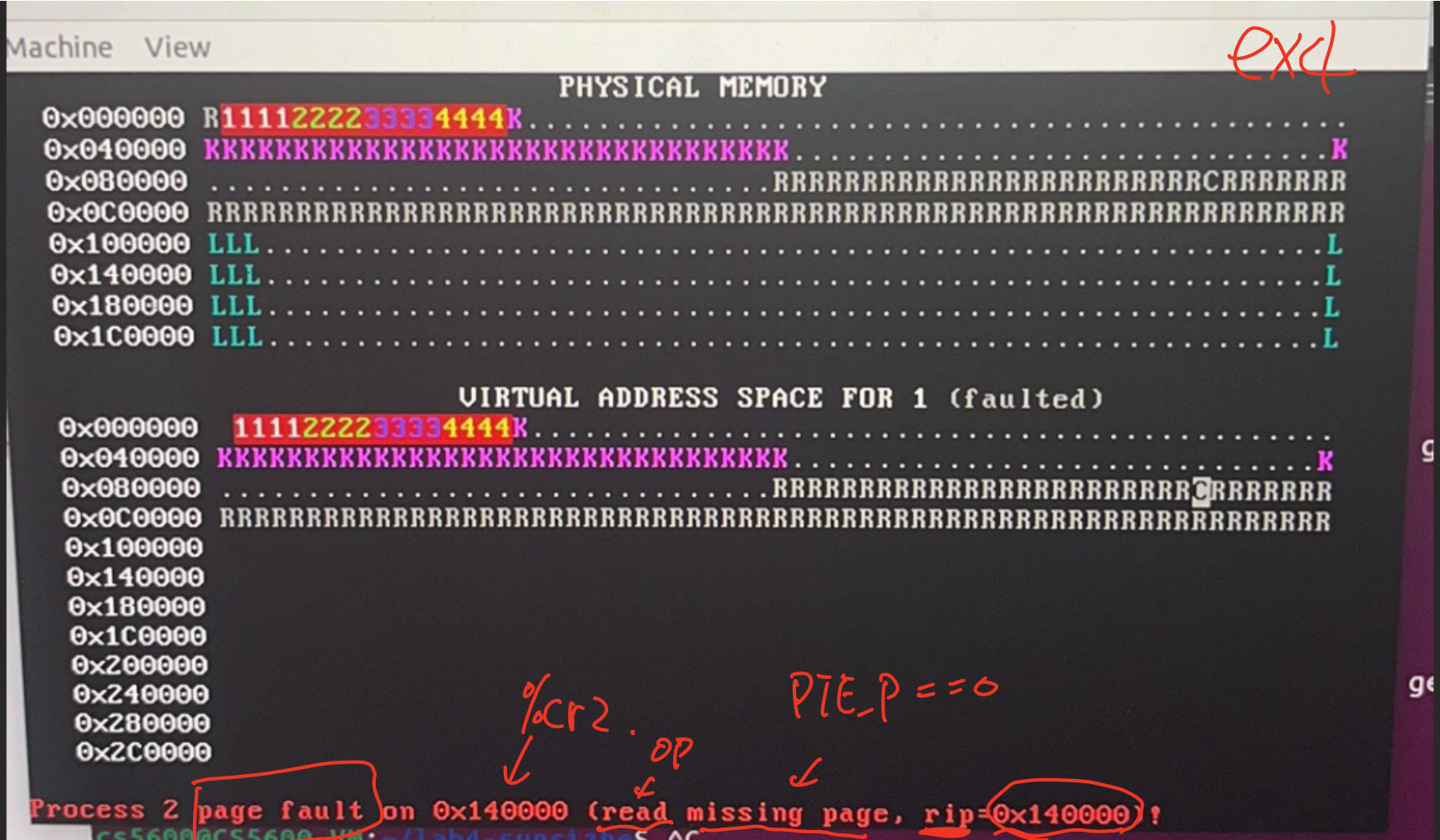


(Memory mapping pre-Meltdown)



(KAISER design: Kernel Address Isolation to have Side channels Efficiently Removed)

Figures borrowed from Meltdown and KAISER papers.


Machine View
PHYSICAL MEMORY
0x000000 R1111222233334444K
0x040000 KKKKKKKKKKKKKKKKKKKKKKKKKKKKKKKK
0x080000 RRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRR
0x0C0000 RRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRR
0x100000 LLL
0x140000 LLL
0x180000 LLL
0x1C0000 LLL
VIRTUAL ADDRESS SPACE FOR 1 (faulted)
0x000000 1111222233334444K
0x040000 KKKKKKKKKKKKKKKKKKKKKKKKKKKKKKKK
0x080000 RRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRR
0x0C0000 RRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRR
0x100000
0x140000
0x180000
0x1C0000
0x200000
0x240000
0x280000
0x2C0000
Process 2 page fault on 0x140000 (read missing page, rip=0x140000)!
%cr2
OP
PTE_P == 0
exc

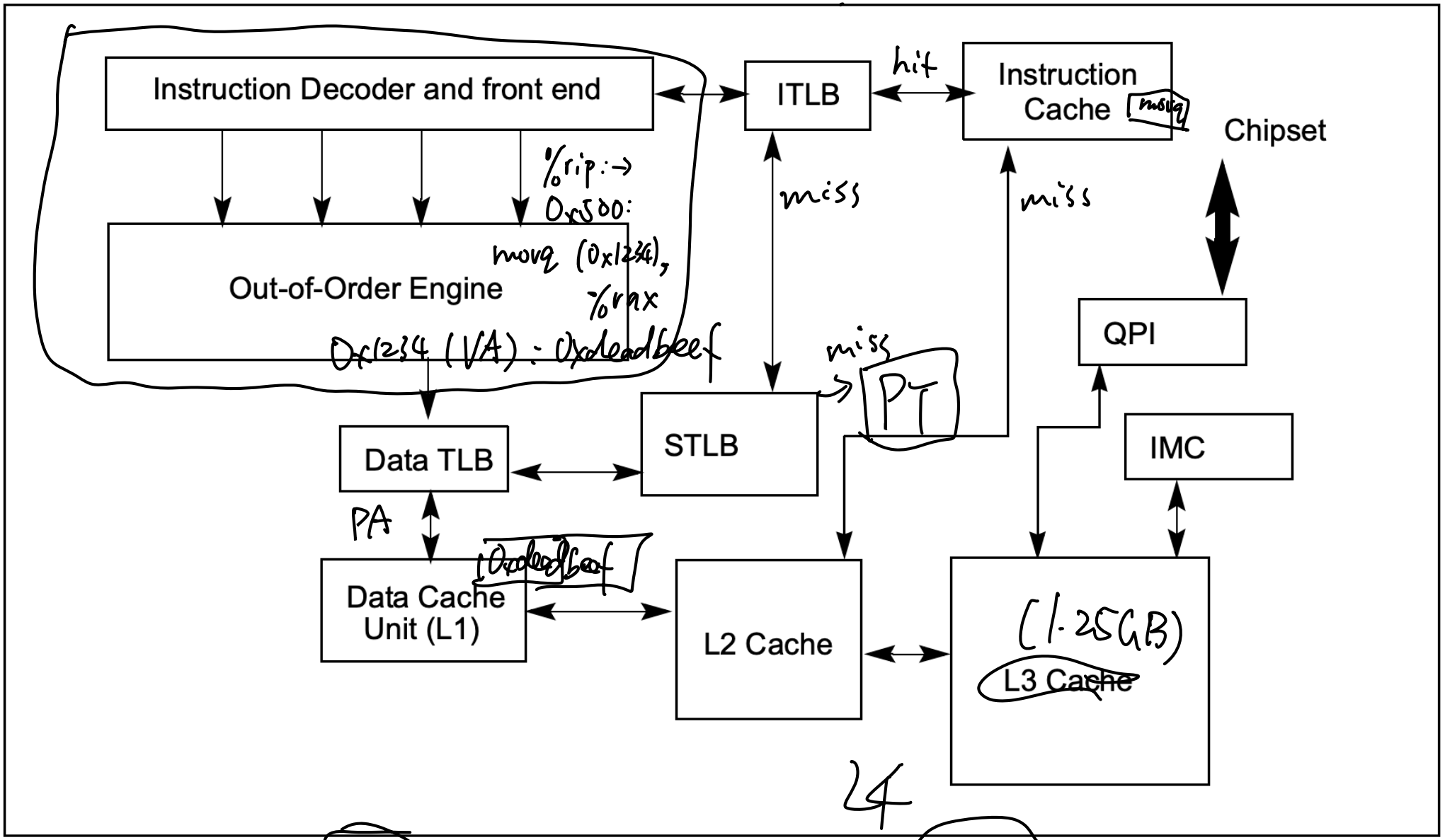


Figure 11-2. Cache Structure of the Intel Core i7 Processors

Week 11.a
CS 5600
03/20 2023

0. Last time
1. Page fault
2. Meltdown and Spectre
3. mmap

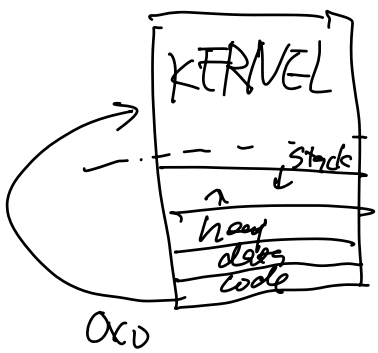
- CPU Cache . TLB
  - ↳ data
  - ↳ mapping

addr → [ ]

VPN → PPN

0xffff..

- Where does OS live?

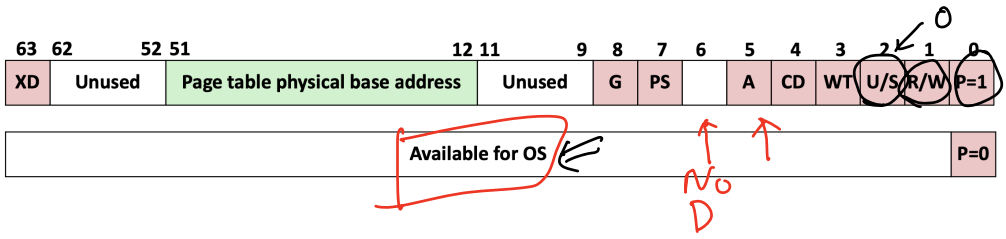


Q: What will happen if process touch KERNEL?

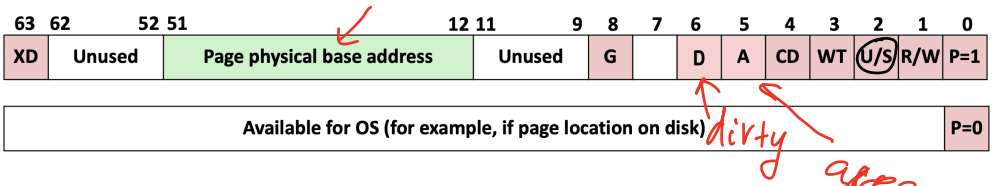
- Crash (page fault) ← permission violation
  - not_mapped (PTE_P)
  - (PTE_U, PTE_W)
  - → exception

→ movq (0x12345), %rax
%rip → add ...

L1-3 PTE



L4 PTE



PF Mechanics:

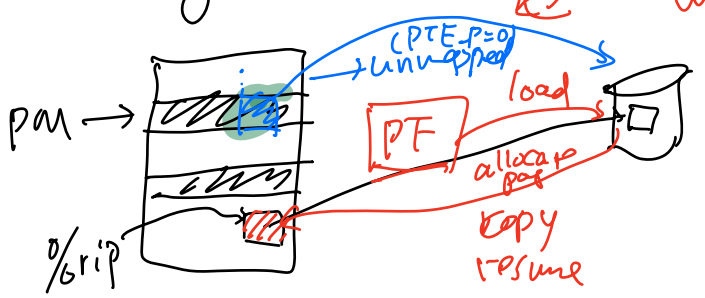
- CPU push a "trap frame" → CPU/HW
- run exception handler } kernel
  - handle PF.

PF use cases:

ramfs

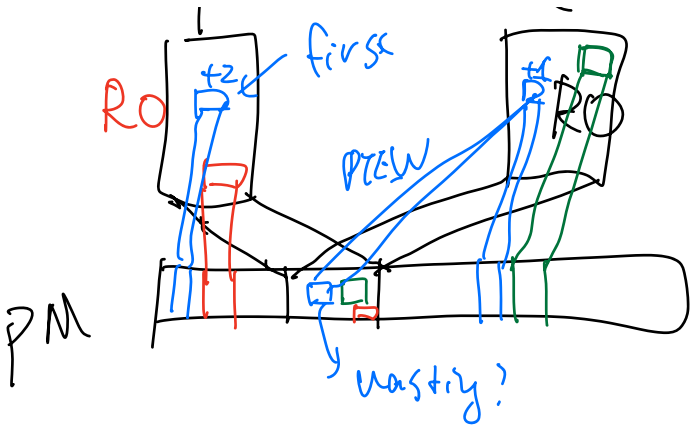
- overcommitting Mem

Q: Do we have to always write mem to disk?



- Copy-on-write
  [fork(), mmap()]

D C

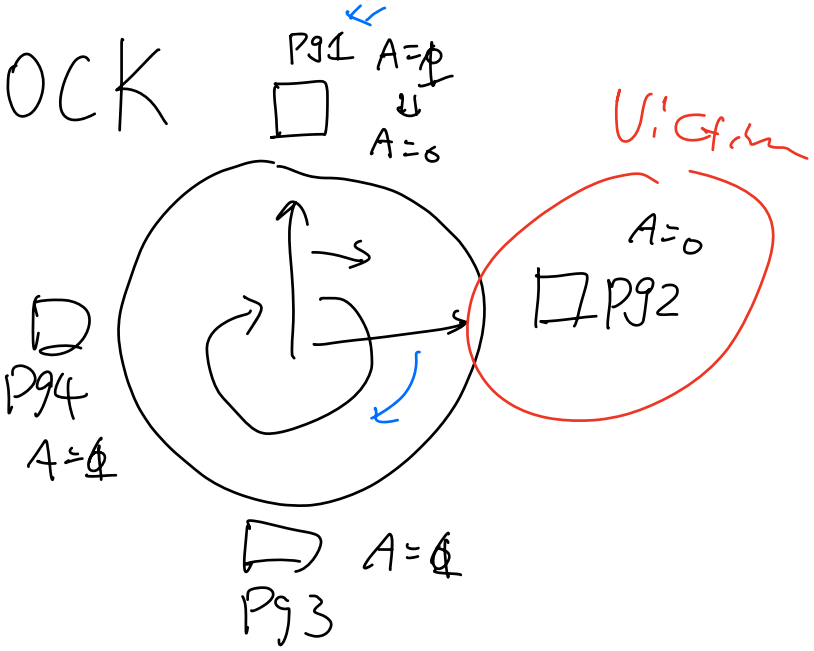


- page replacement policy

  Q: Which page we should kick off?

  - FIFO
  - LRU ←

    Least Recent Used
  - CLOCK



Before PF

After

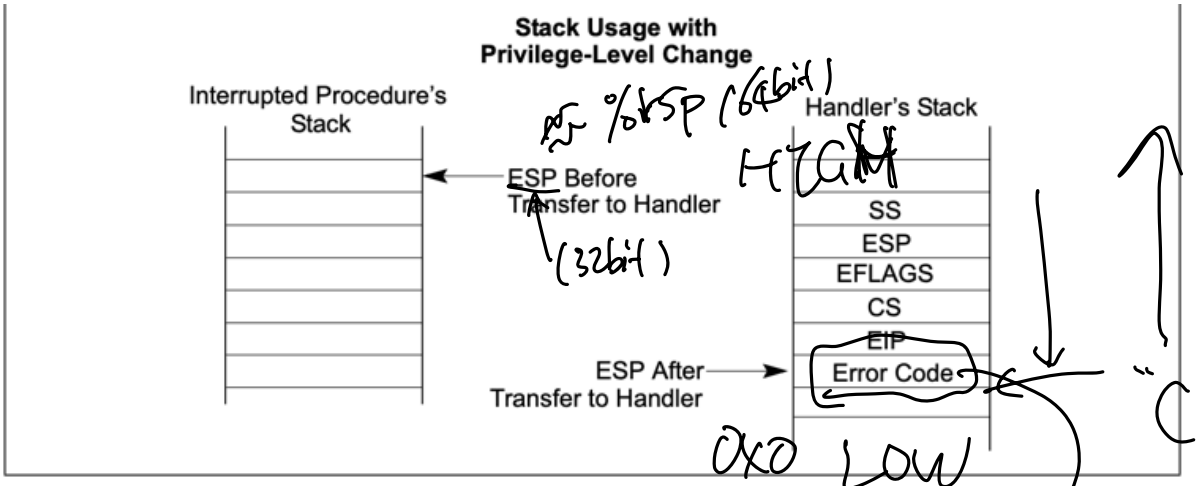


Figure 6-4. Stack Usage on Transfers to Interrupt and Exception-Handling Routines

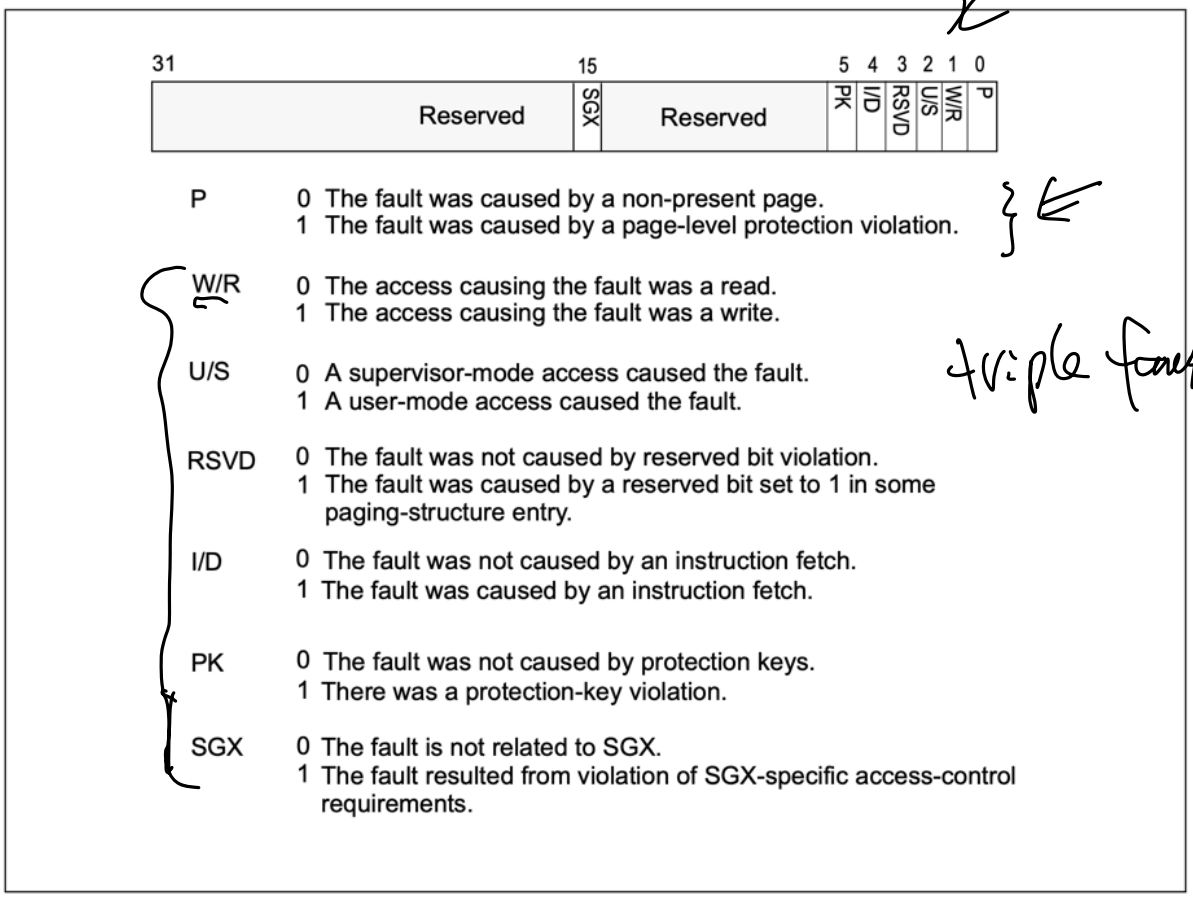


Figure 4-12. Page-Fault Error Code

```
// WeensyOS (Lab4): x86-64.h

typedef struct regstate {
    // General-purpose registers
    uint64_t reg_rax;
    uint64_t reg_rcx;
    uint64_t reg_rdx;
    uint64_t reg_rbx;
    uint64_t reg_rbp;
    uint64_t reg_rsi;
    uint64_t reg_rdi;
    uint64_t reg_r8;
    uint64_t reg_r9;
    uint64_t reg_r10;
    uint64_t reg_r11;
    uint64_t reg_r12;
    uint64_t reg_r13;
    uint64_t reg_r14;
    uint64_t reg_r15;
    uint64_t reg_fs;
    uint64_t reg_gs;

    uint32_t reg_intno;   // Interrupt number, -1 for syscall
    uint32_t reg_swapgs;  // Whether to `swapgs` on resume
    // Error code (supplied by hardware for some x86-64 exceptions)
    uint64_t reg_errcode;

    // Task status (pushed by exception mechanism, read by `iret`)
    uint64_t reg_rip;
    uint64_t reg_cs;
    uint64_t reg_rflags;
    uint64_t reg_rsp;
    uint64_t reg_ss;
} regstate;


// WeensyOS (Lab4): k-exception.S

// Exception entry point
//    Most exception handlers jump here.
.globl _Z15exception_entryv
_Z15exception_entryv:
    push %gs
    push %fs
    pushq %r15
    pushq %r14
    pushq %r13
    pushq %r12
    pushq %r11
    pushq %r10
    pushq %r9
    pushq %r8
    pushq %rdi
    pushq %rsi
    pushq %rbp
    pushq %rbx
    pushq %rdx
    pushq %rcx
    pushq %rax
    movq %rsp, %rdi

    // load kernel page table
    movq $kernel_pagetable, %rax
    movq %rax, %cr3

    call _Z9exceptionP8regstate
    // `exception` should never return.
```



```
// WeensyOS (Lab4): kernel.cc

void exception(regstate* regs) {
    // Copy the saved registers into the `current` process descriptor.
    current->regs = *regs;
    regs = &current->regs;

    …

    // Actually handle the exception.
    switch (regs->reg_intno) {

    case INT_IRQ + IRQ_TIMER:
        ++ticks;
        lapicstate::get().ack();
        schedule();
        break;                  /* will not be reached */

    case INT_PF: {
        // Analyze faulting address and access type.
        uintptr_t addr = rdcr2();
        const char* operation = regs->reg_errcode & PTE_W
                ? "write" : "read";
        const char* problem = regs->reg_errcode & PTE_P
                ? "protection problem" : "missing page";

        if (!(regs->reg_errcode & PTE_U)) {
            proc_panic(current, "Kernel page fault on %p (%s %s, rip=%p)!\n",
                       addr, operation, problem, regs->reg_rip);
        }
        error_printf(CPOS(24, 0), 0x0C00,
                     "Process %d page fault on %p (%s %s, rip=%p)!\n",
                     current->pid, addr, operation, problem, regs->reg_rip);
        current->state = P_FAULTED;
        break;
    }

    …

    // Return to the current process (or run something else).
    if (current->state == P_RUNNABLE) {
        run(current);
    } else {
        schedule();
    }
}
```