CS202

Agenda

1. Finish Virtual Memory

2. Context Switching

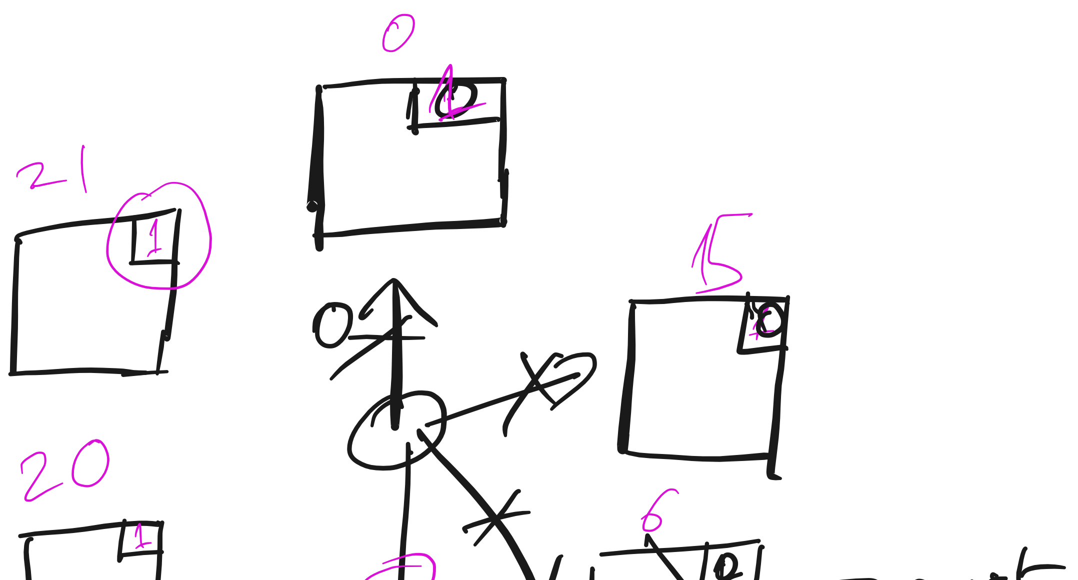
Virtual Memory

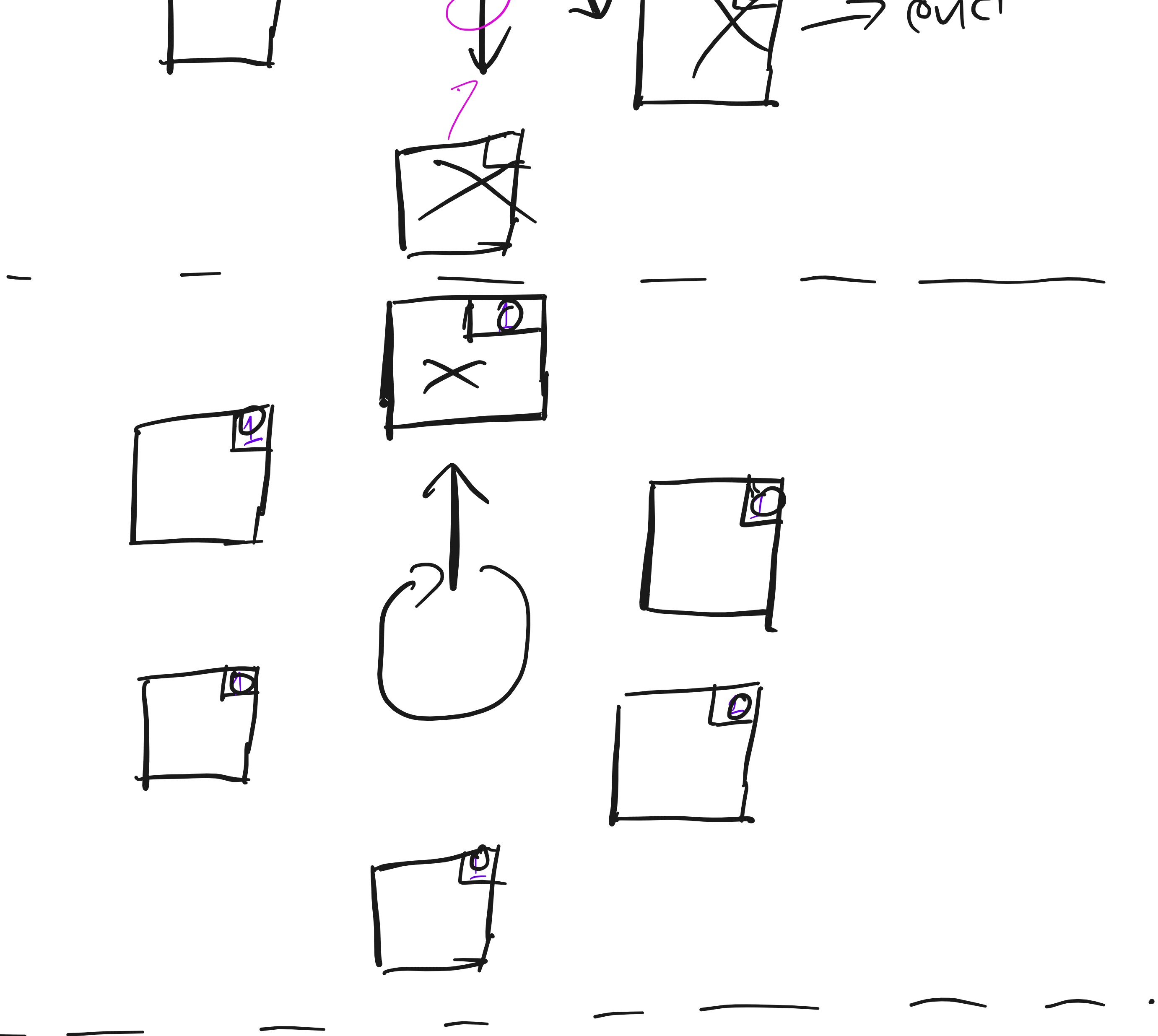
- Demand Paging
  - ↳ Mechanism

- Page Replacement Policy
  - What Page To Evict

- FIFO, MIN, LRU

- Clock





THRASHING

Small Physical

Bad access pattern

Lot of allocated virtual memory

Belady's Anomaly

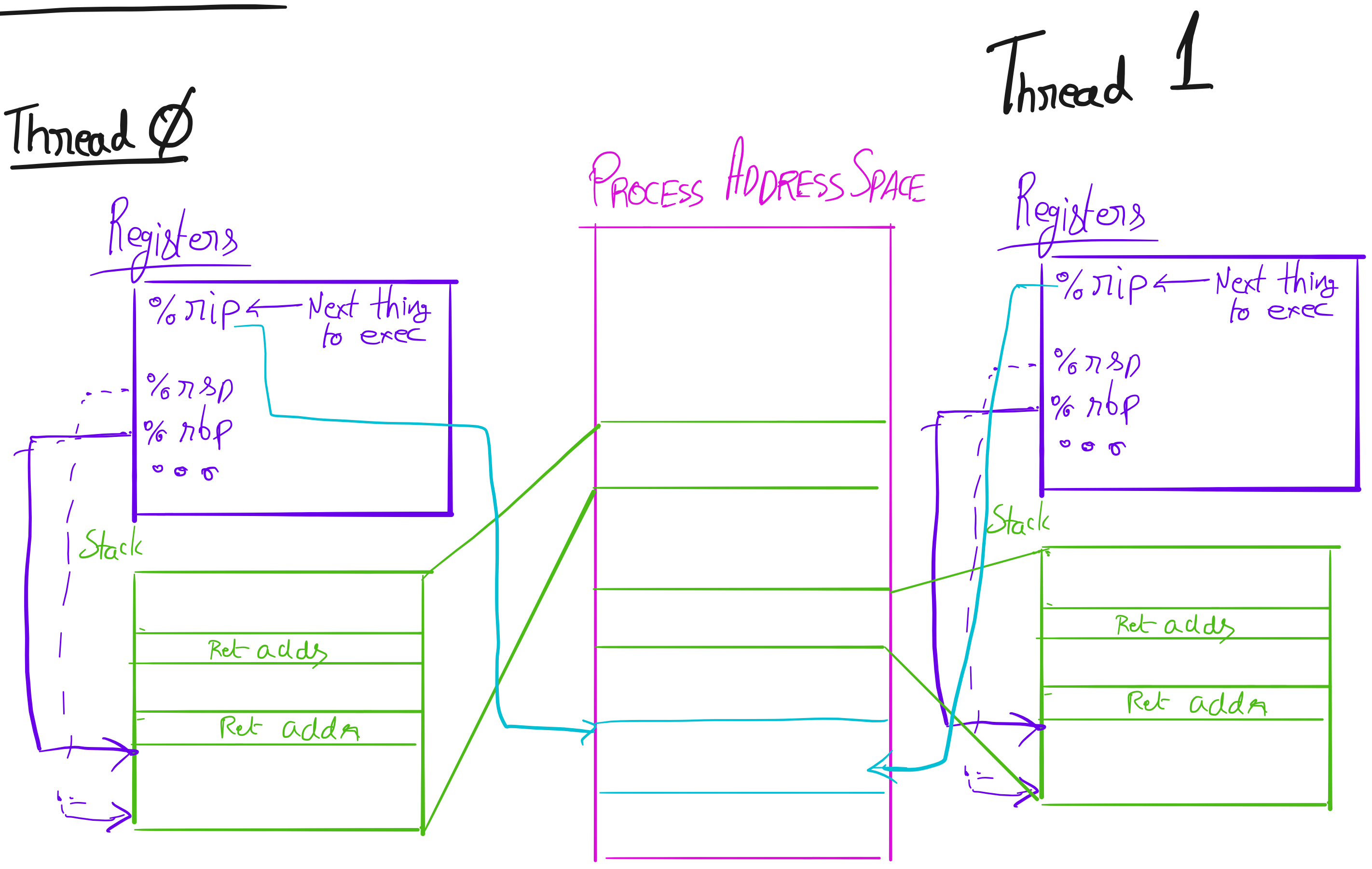
FIFO

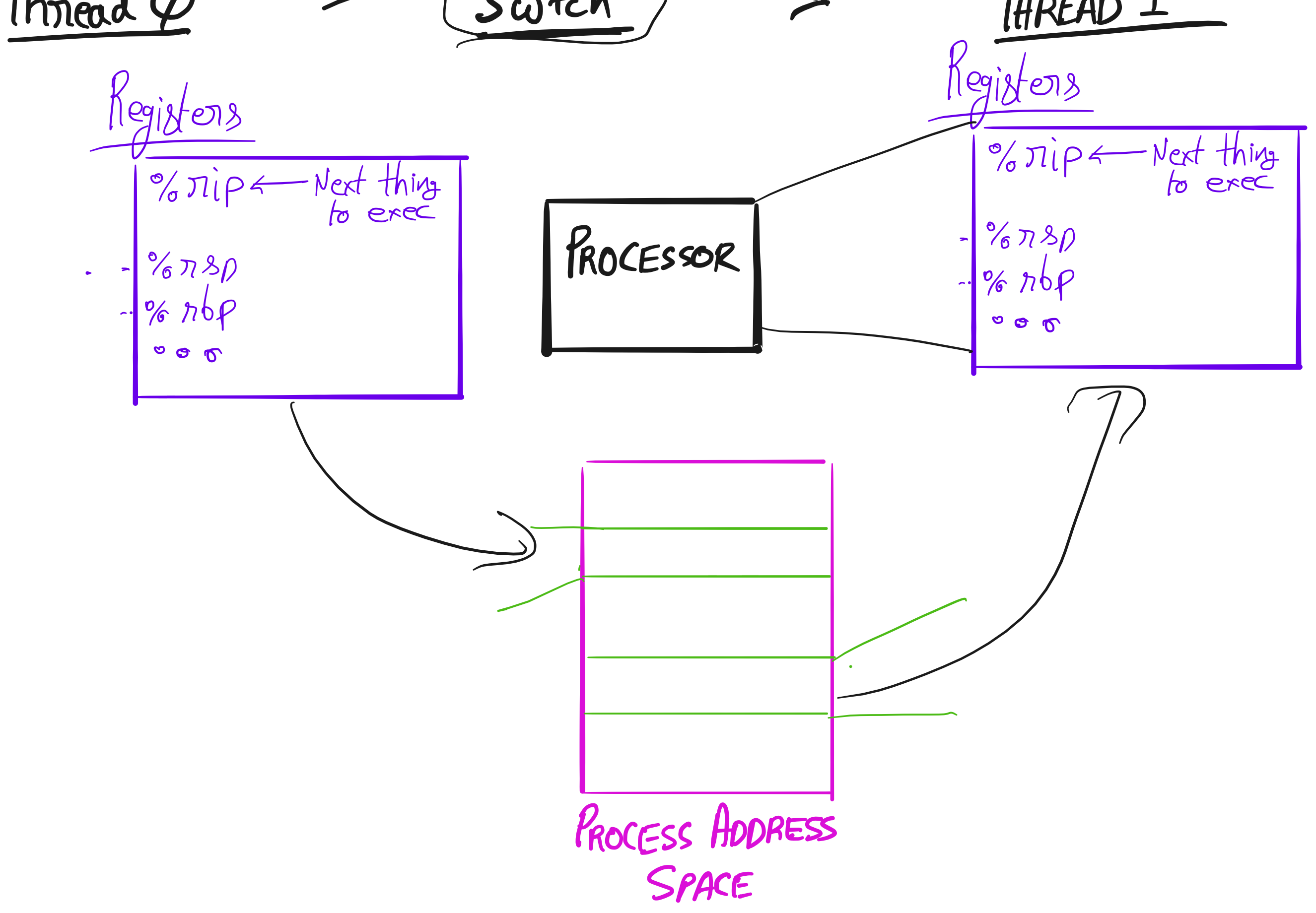
hits =

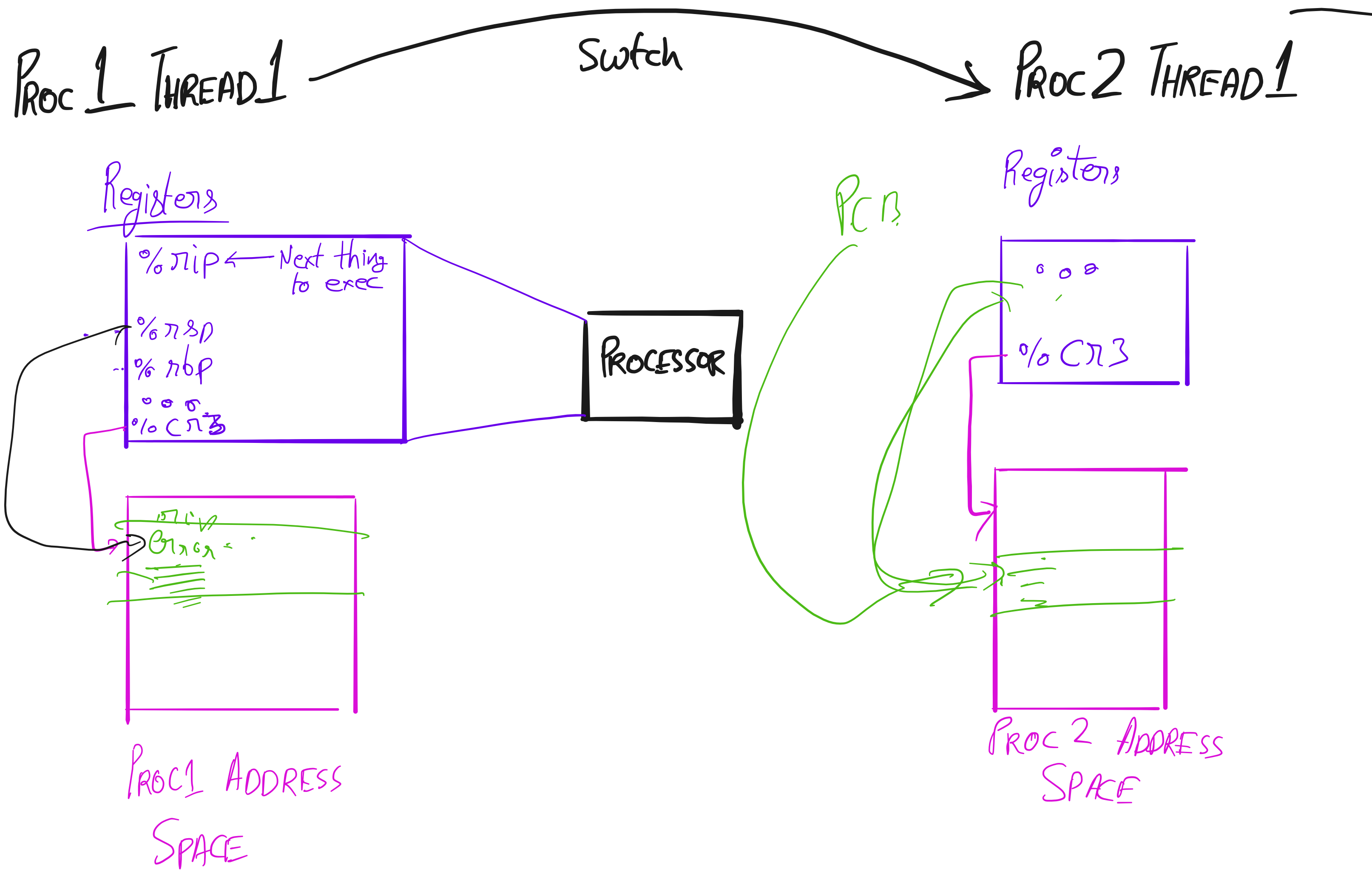
	A	B	C	D	A	B	E	A	B	C	D	E
P_0	A			D			E					h
P_1		B			A			h		C		
P_2			C			B			h		D	

	A	B	C	D	A	B	E	A	B	C	D	E
P_0	A				h		E				D	
P_1		B				h		A				E
P_2			C						B			
P_3				D						C		

Context Switch



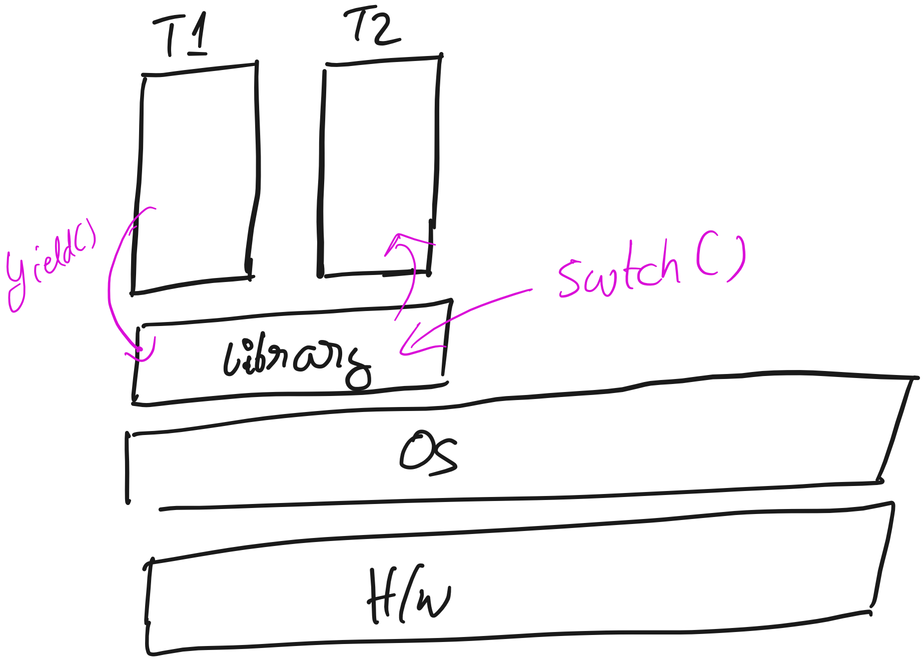




Switching & Scheduling in WeensyOS

(Switch sheets)

User Level Threading



Where?

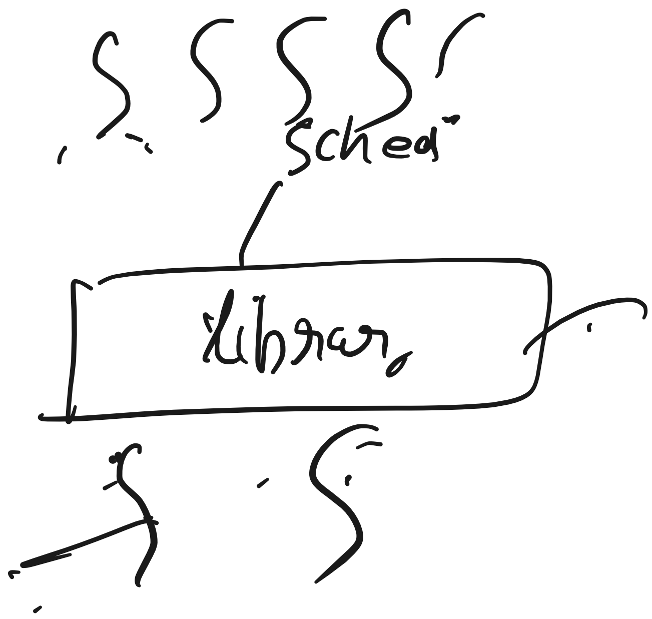
- Go
- Rust
- ...

How?

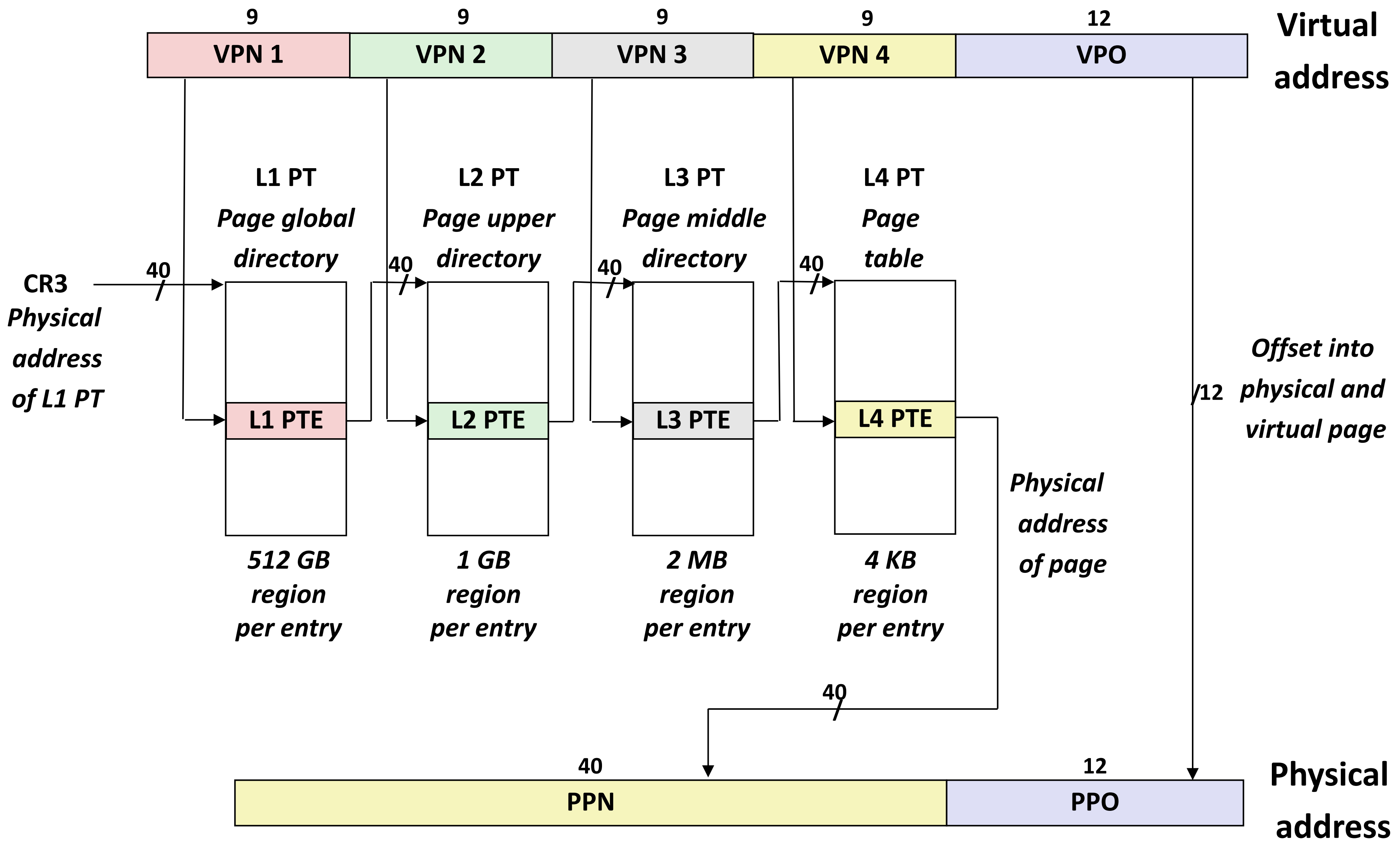
Somewhat similar to Weensy OS but

- No changing CR3
- Cannot access all registers

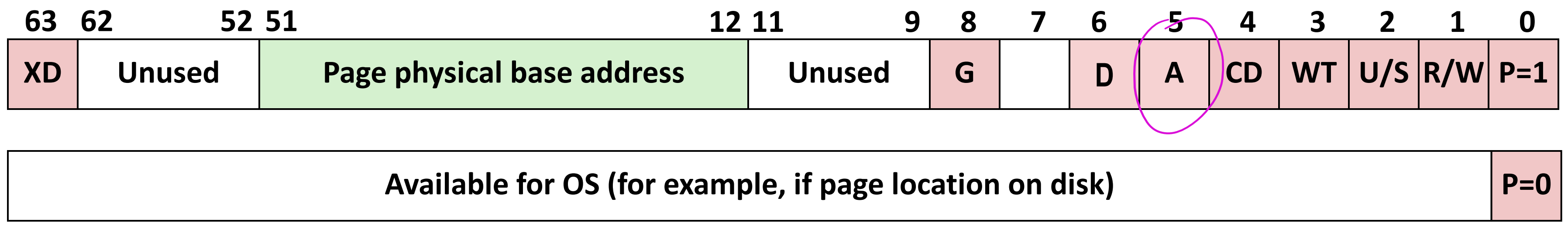
(switch to handout)



Core i7 Page Table Translation



Core i7 Level 4 Page Table Entries


63
62
52
51
12
11
9
8
7
6
5
4
3
2
1
0
XD
Unused
Page physical base address
Unused
G
D
A
CD
WT
U/S
R/W
P=1
Available for OS (for example, if page location on disk)
P=0

Each entry references a 4K child page. Significant fields:

P: Child page is present in memory (1) or not (0)

R/W: Read-only or read-write access permission for this page

U/S: User or supervisor mode access

WT: Write-through or write-back cache policy for this page

A: Reference bit (set by MMU on reads and writes, cleared by software)

D: Dirty bit (set by MMU on writes, cleared by software)

Page physical base address: 40 most significant bits of physical page address
(forces pages to be 4KB aligned)

XD: Disable or enable instruction fetches from this page.

Core i7 Level 1-3 Page Table Entries

63	62 52	51 12	11 9	8	7	6	5	4	3	2	1	0
XD	Unused	Page table physical base address	Unused	G	PS		A	CD	WT	U/S	R/W	P=1

Available for OS	P=0

Each entry references a 4K child page table. Significant fields:

P: Child page table present in physical memory (1) or not (0).

R/W: Read-only or read-write access access permission for all reachable pages.

U/S: user or supervisor (kernel) mode access permission for all reachable pages.

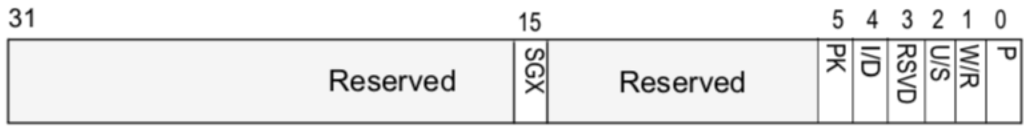
WT: Write-through or write-back cache policy for the child page table.

A: Reference bit (set by MMU on reads and writes, cleared by software).

PS: Page size: if bit set, we have 2 MB or 1 GB pages (bit can be set in Level 2 and 3 PTEs only).

Page table physical base address: 40 most significant bits of physical page table address (forces page tables to be 4KB aligned)

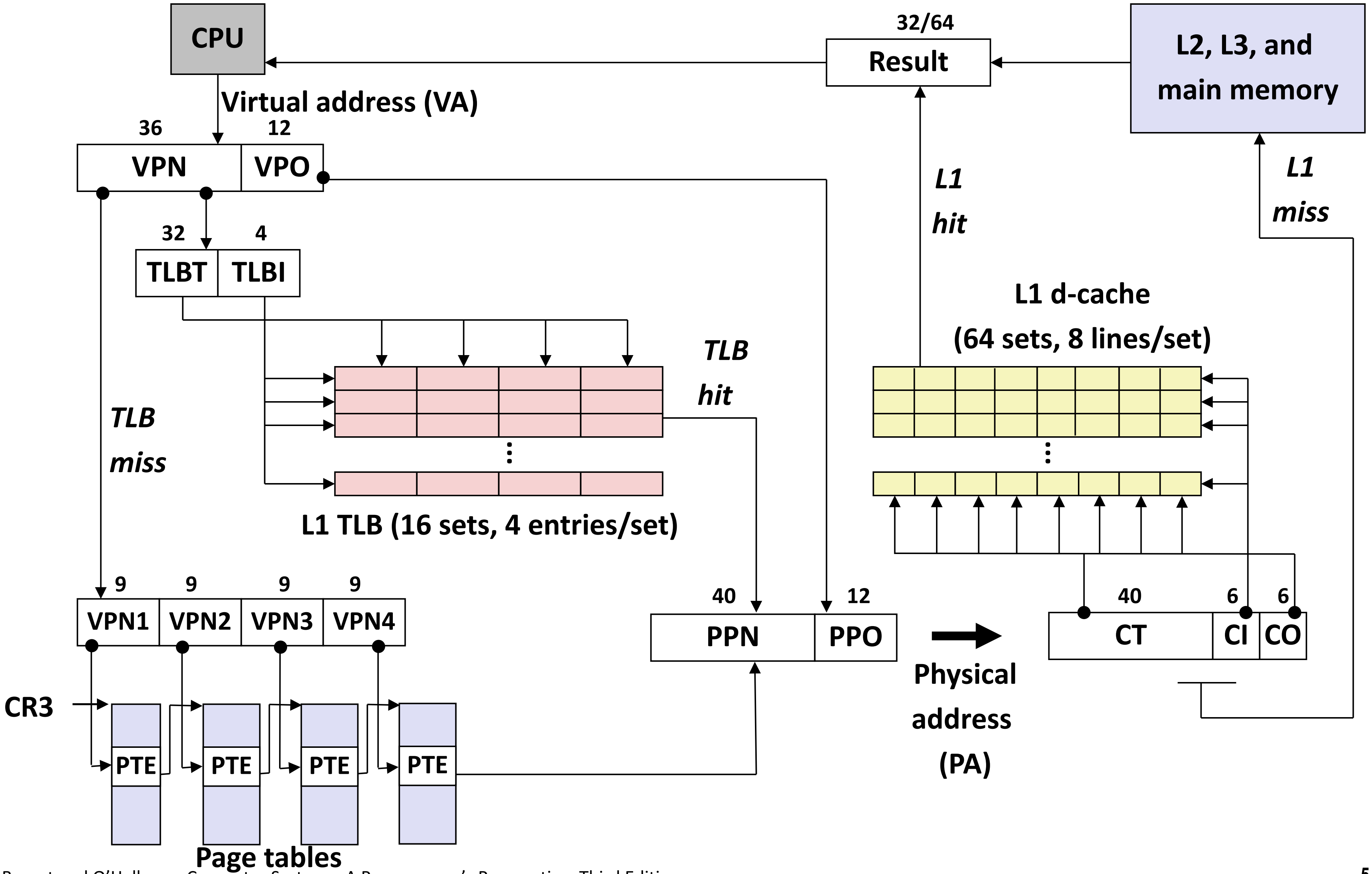
XD: Disable or enable instruction fetches from all pages reachable from this PTE.



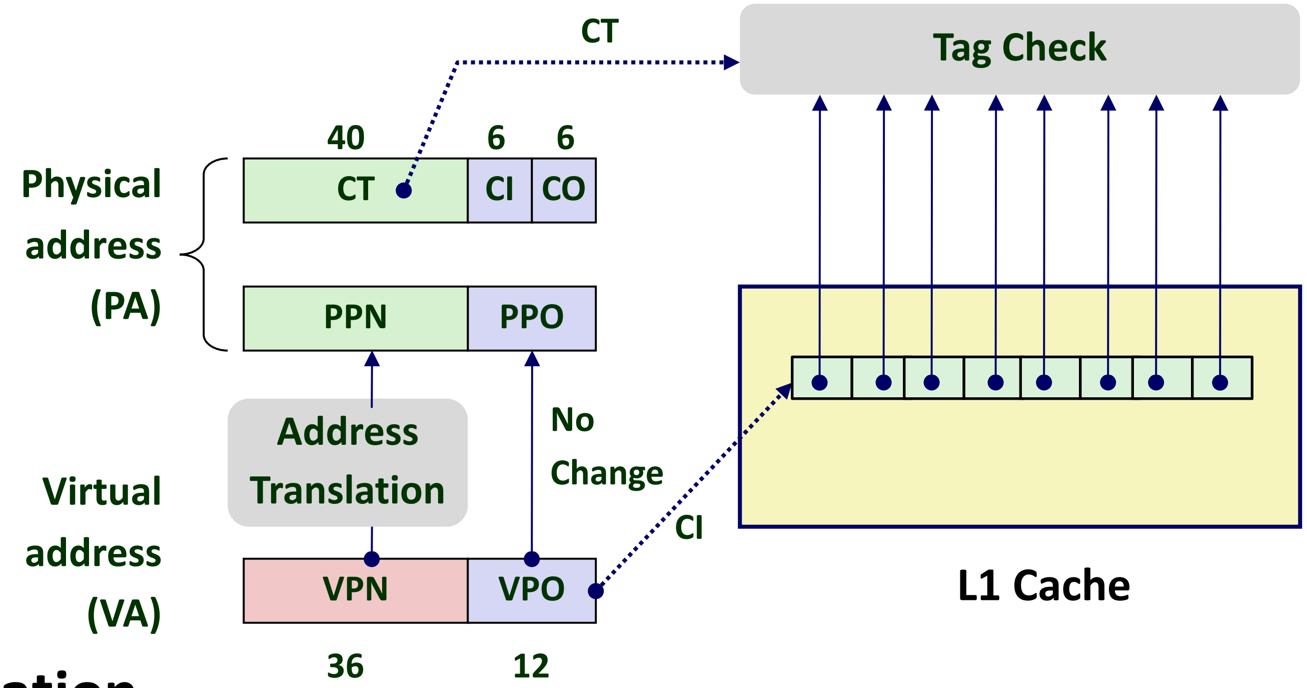
P	0	The fault was caused by a non-present page.
	1	The fault was caused by a page-level protection violation.
W/R	0	The access causing the fault was a read.
	1	The access causing the fault was a write.
U/S	0	A supervisor-mode access caused the fault.
	1	A user-mode access caused the fault.
RSVD	0	The fault was not caused by reserved bit violation.
	1	The fault was caused by a reserved bit set to 1 in some paging-structure entry.
I/D	0	The fault was not caused by an instruction fetch.
	1	The fault was caused by an instruction fetch.
PK	0	The fault was not caused by protection keys.
	1	There was a protection-key violation.
SGX	0	The fault is not related to SGX.
	1	The fault resulted from violation of SGX-specific access-control requirements.

Figure 4-12. Page-Fault Error Code

End-to-end Core i7 Address Translation

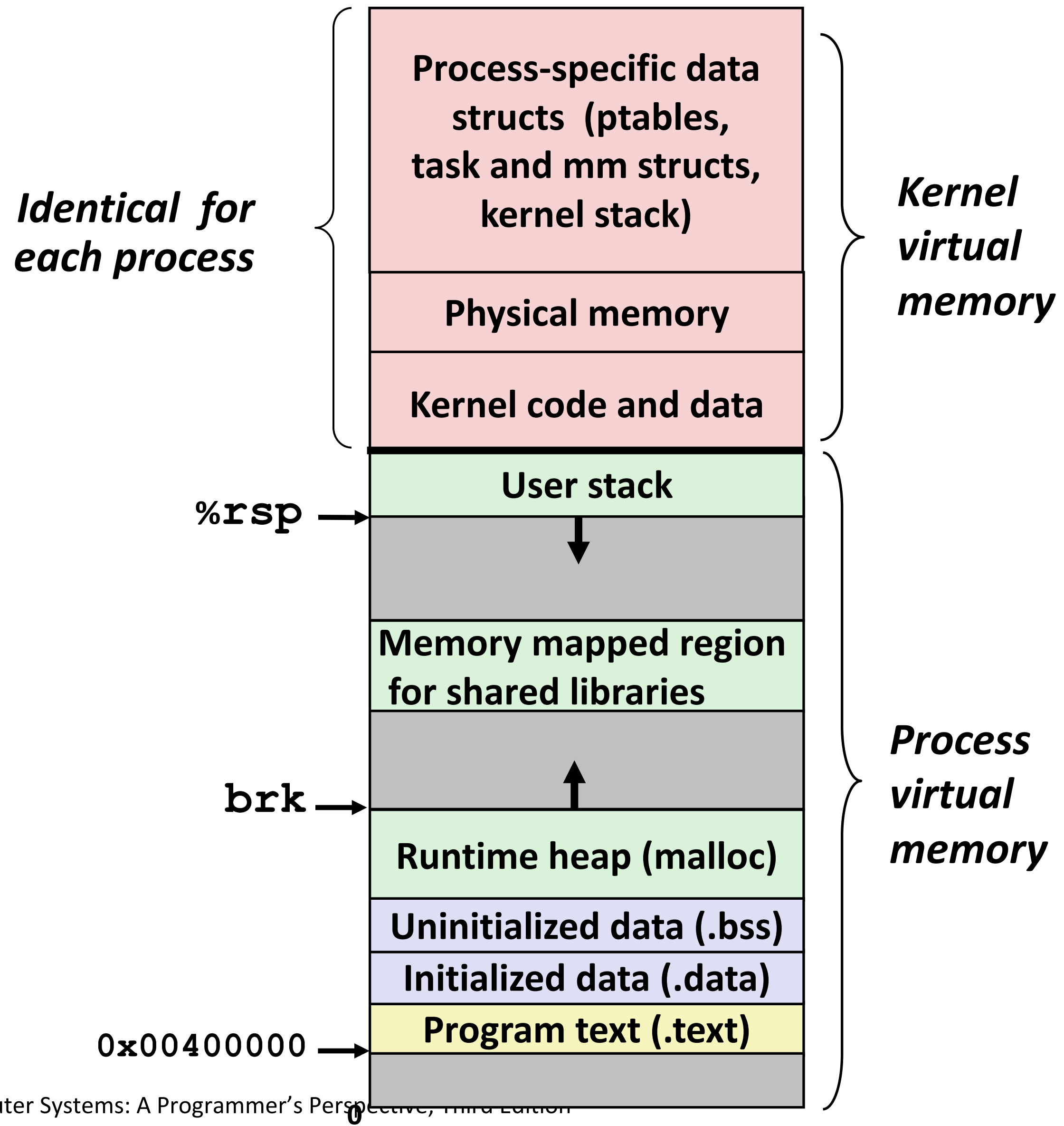


Cute Trick for Speeding Up L1 Access

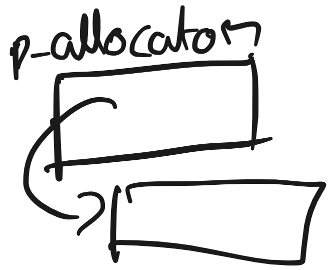


- **Observation**
  - Bits that determine CI identical in virtual and physical address
  - Can index into cache while address translation taking place
  - Cache carefully sized to make this possible: 64 sets, 64-byte cache blocks
  - Means 6 bits for cache index, 6 for *cache* offset
  - That's 12 bits; matches *VPO, PPO* → One reason pages are 2^{12} bits = 4 KB

Virtual Address Space of a Linux Process



R1



Return path

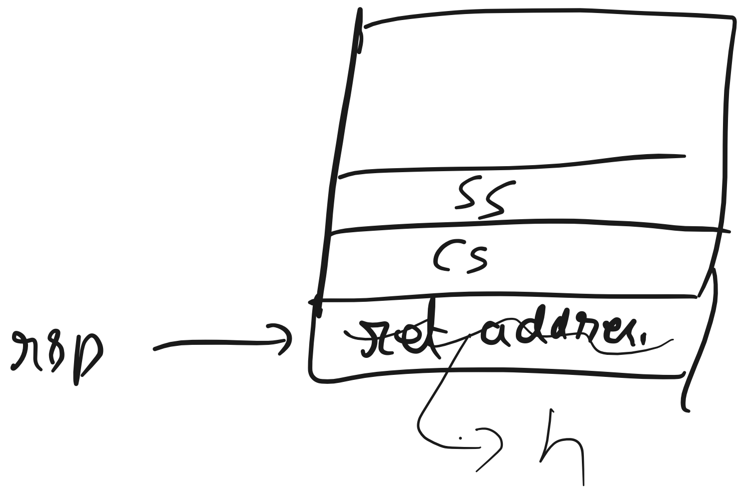
File: `kernel.c` line 26

```
// schedule
//    Pick the next process to run and then run it.
//    If there are no runnable processes, spins forever.

void schedule(void) {
    pid_t pid = current->p_pid;
    while (1) {
        pid = (pid + 1) % NPROC;
        if (processes[pid].p_state == P_RUNNABLE) {
            run(&processes[pid]);
        }
        // If Control-C was typed, exit the virtual machine.
        check_keyboard();
    }
}


// run(p)
//    Run process `p`. This means reloading all the registers from
//    `p->p_registers` using the `popal`, `popl`, and `iret` instructions.
//
//    As a side effect, sets `current = p`.

void run(proc* p) {
    assert(p->p_state == P_RUNNABLE);
    current = p;

    // Load the process's current pagetable.
    set_pagetable(p->p_pagetable);

    // This function is defined in k-exception.S. It restores the process's
    // registers then jumps back to user mode.
    exception_return(&p->p_registers);

 spinloop: goto spinloop;       // should never get here
}
```

wr,
cr3

File: `k-exception.S` line 158

```
exception_return:
        movq %rdi, %rsp
        popq %rax
        popq %rcx
        popq %rdx
        popq %rbx
        popq %rbp
        popq %rsi
        popq %rdi
        popq %r8
        popq %r9
        popq %r10
        popq %r11
        popq %r12
        popq %r13
        popq %r14
        popq %r15
        popq %fs
        popq %gs
        addq $16, %rsp
        iretq
```



Entry path

File: `k-exception.S` line 134

```
generic_exception_handler:
        pushq %gs
        pushq %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
        call exception
        # `exception` should never return.
```

File `kernel.c` line

```
void exception(x86_64_registers* reg) ...
```

File `x86-64.h` line 86

```
typedef struct x86_64_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;

    uint64_t reg_intno;         // (3) Interrupt number and error
    uint64_t reg_err;           // code (optional; supplied by x86
                                // interrupt mechanism)

    uint64_t reg_rip;       // (4) Task status: instruction pointer,
    uint16_t reg_cs;        // code segment, flags, stack
    uint16_t reg_padding2[3];   // in the order required by `iret`
    uint64_t reg_rflags;
    uint64_t reg_rsp;
    uint16_t reg_ss;
    uint16_t reg_padding3[3];
} x86_64_registers;
```

```
CS 202, Spring 2023
Handout 10 (Class 17)

1. User-level threads and swtch()

    We'll study this in the context of user-level threads.

    Per-thread state in thread control block:

        typedef struct tcb {
            unsigned long saved_rsp;    /* Stack pointer of thread */
            char *t_stack;              /* Bottom of thread's stack */
            /* ... */
            };

    Machine-dependent thread initialization function:

        void thread_init(tcb **t, void (*fn) (void *), void *arg);

    Machine-dependent thread-switch function:

        void swtch(tcb *current, tcb *next);

    Implementation of swtch(current, next):

        # gcc x86-64 calling convention:
        # on entering swtch():
        #  register %rdi holds first argument to the function ("current")
        #  register %rsi holds second argument to the function ("next")

        # Save call-preserved (aka "callee-saved") regs of 'current'
        pushq %rbp
        pushq %rbx
        pushq %r12
        pushq %r13
        pushq %r14
        pushq %r15

        # store old stack pointer, for when we swtch() back to "current" later
        movq %rsp, (%rdi)                       # %rdi->saved_rsp = %rsp
        movq (%rsi), %rsp                       # %rsp = %rsi->saved_rsp

        # Restore call-preserved (aka "callee-saved") regs of 'next'
        popq %r15
        popq %r14
        popq %r13
        popq %r12
        popq %rbx
        popq %rbp

        # Resume execution, from where "next" was when it last entered swtch()
        ret
```

```

2. Example use of swtch(): the yield() call.

    A thread is going about its business and decides that it's executed for
    long enough. So it calls yield(). Conceptually, the overall system needs
    to now choose another thread, and run it:

    void yield() {

        tcb* next    = pick_next_thread();  /* get a runnable thread */
        tcb* current = get_current_thread();

        swtch(current, next);

        /* when 'current' is later rescheduled, it starts from here */
    }

3. How do context switches interact with I/O calls?

    This assumes a user-level threading package.

    The thread calls something like "fake_blocking_read()". This looks
    to the _thread_ as though the call blocks, but in reality, the call
    is not blocking:

    int fake_blocking_read(int fd, char* buf, int num) {

        int nread = -1;

        while (nread == -1) {

            /* this is a non-blocking read() syscall */
            nread = read(fd, buf, num);

            if (nread == -1 && errno == EAGAIN) {
                /*
                 * read would block. so let another thread run
                 * and try again later (next time through the
                 * loop).
                 */
                yield();
            }
        }

        return nread;
    }
```