Discussion 9 (week 10)

Concurrent Programming and Final Review

Brandon Wu

Reminders

- Final is Tuesday 12/16 8:00AM – 11:00AM
- Super office hour Monday 10AM-4PM in 4670 Boelter Hall
- Course Evaluations – please evaluate me!

Concurrent Processes

- Process is an abstraction of a running program
- A process **context** describes its state
  - Registers
  - Virtual address space
- Fork spawns a child process
  - Duplicates the context

Processes are Expensive!

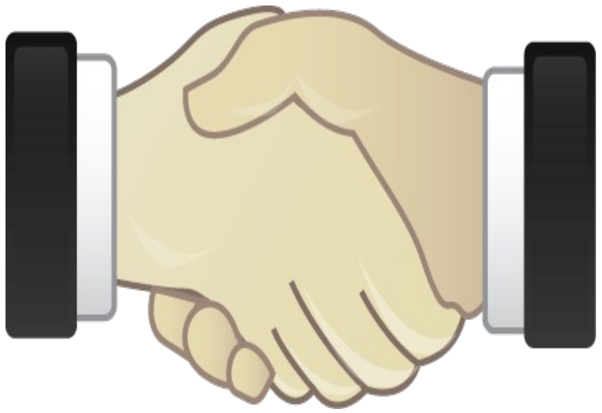
- OS overhead in creating a process
- Sharing state information is tough
  - Process have separate Virtual Address space
  - Must communicate via OS IPC (interprocess communication
- On the other hand, processes are hard to screw up, cannot accidentally step on memory of another process

Threads

- **Threads** are logical flow that share context of program
- Has its own "agenda"
  - Can execute independently of its peers
  - Has private set of registers and stack
- Shares <u>process context</u> with all other threads within process
  - Share the same addr space

Thread context and Sharing

- Threads have their own context
  - Stack
  - General purpose registers
- Can share global variables with other threads within the process



OS <u>does not</u> guarantee order of execution across threads!

OS also <u>does not</u> guarantee exclusion

```
volatile int count = 0;
int main() {
    ...
    pthread_create(&tid1, NULL, thread, &arg);
    pthread_create(&tid2, NULL, thread, &arg);
    ...
    return 0;
}
void* thread(void *arg) {
    num = *(int* ) arg);
    for (int i=0; i<num; i++)
        count++;
}
```

Thread concurrency

- After tid1 and tid2 terminate, we expect `count = 2*arg` but it doesn't
- Threads can execute concurrently, even in parallel
- We don't guarantee exclusion of shared resources
- Unguarded modification of shared data results in **race conditions**

Mutual Exclusion: Semaphores

- A global variable used only to control access to other global variables
- Two operations on a semaphore s:
  - **Down**: P(s) return (s > 0) ? -- s : RETRY
  - **Up**: V(s) performs ++s
- In C library:

```
int sem_init(sem_t* sem, 0, unsigned val);
int sem_wait(sem_t* s);  /* P(s) */
int sem_post(sem_t* s);      /* V(s) */
```

Up and **Down** operations must be **atomic**.
WHY?

Semaphores

- Semaphores with value = 1 are **<u>Locks</u>**
- s == 1 → unlocked
- s == 0 → locked
- P(s) → attempt to acquire lock s
- V(s) → unlock s

- Binary semaphores sometimes called **<u>Mutexes</u>**

Fixing the multi-threaded Counting Problem

```
volatile int count = 0;
sem_t mutex;
...
void* thread(void* arg) {
    sem_wait(&mutex);     /* P(&mutex) */
    count++;
    sem_post(&mutex);     /* V(&mutex) */
}
```

Readers-Writers Problem

- Two types of players: **readers** and **writers**
- Many readers can read the value at the same time
- Modifying (writing) the value requires <u>exclusive access</u>

Readers-Writers Problem

```
/* GLOBAL VARIABLES */
int readcnt = 0;
sem_t rc_mutex, wlock;   /* initially 1 */
```

The Reader

```
void reader( ) {
    sem_wait(&rc_mutex); // P(&rc_mutex)
    readcnt++;
    if (readcnt == 1)
        sem_wait(&wlock);                // P (&wlock)
    sem_post(&rc_mutex);     // V(&rc_mutex)
    // Reading happens...
    sem_wait(&rc_mutex);      //R(&rc_mutex);
    readcnt --;
    if (readcnt == 0)
        sem_post(&wlock);               // V(&wlock);
    sem_post(&rc_mutex);      //V(&rc_mutex);
```

The Reader

- `rc_mutex` protects access to readcnt
  - Readers modify this
- `wlock` is the lock required to obtain exclusive access to the data
  - Writer cannot have this while there are active readers

The Writer

```
void writer( ) {
    sem_wait(&wlock);   // Acquire write lock
    … do some writing
    sem_post(&wlock);   // Release write lock
}
```

First Readers-Writers Problem

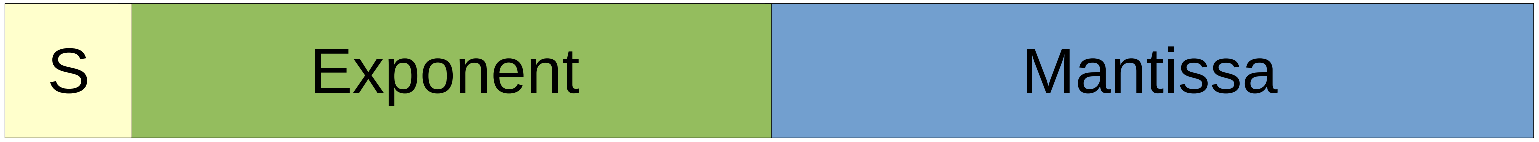
- This implementation favors **readers**
- While there is at least one reader, writer must wait
- If readers keep coming, writer never executes
  - This is an example of **starvation**

CHAPTER 2: DATA REPRESENTATION

Floating Point Conversions

IEEE Floating Point Recap

- Decimal value x = (-1)^s * 2^(E-BIAS) * M



- Where BIAS = 2^(k-1)-1
- Normalized FP:
  - When E != 0
- Denormalized
  - E == 0, exponent = 1 - BIAS

Floating Point Example

- Consider 9 bit floating point
- 3 bit exponent, 5 bit mantissa
- What is the bias
- What is the smallest possible range beweteen two representatable values?
- What is the range of numbers where the difference between any two representable numbers is > 2^-5

Example from Uen-Tao

Consider the two floating point representations:

A: 0 000 00000 (three exp, five fraction)

B: 0 00000 000 (five exp, three fraction)

What are the closest Format B approximations for the Format A value:

0 000 10101

Floating Point Example

- Format A:
- 0 000 10101
- Bias = 3
- E = 1 – Bias = -2 (Denormalized)
- V = 2-2 * (1/2 + 1/23 + 1/25) = 2-3 * (1 + 1/22 + 1/24) = .1640625
- Format B:
- Format B has enough range in the exponent bits to represent the full exponent range of format A.
- Bias = 15
- E = -3 = e – Bias
- e = 12
- We want this V to equal the V from format A (2-3 * (1 + 1/22 + 1/24))

Floating Point Example

However, it is impossible to represent this in format B

V = 2-3 * (1 + 1/22 + 1/24)

Format B only has three fractional bits and cannot represent a fractional contribution of 1/24

Rounding down solution:

V = 2-3 * (1 + 1/22) = .15625

0 01100 010

Rounding up solution:

V = 2-3 * (1 + 1/22 + 1/23) = .171875

0 01100 011

CHAPTER 3: MACHINE CODE

Reverse Engineering and Stack Discipline

Inferring Operation Suffix

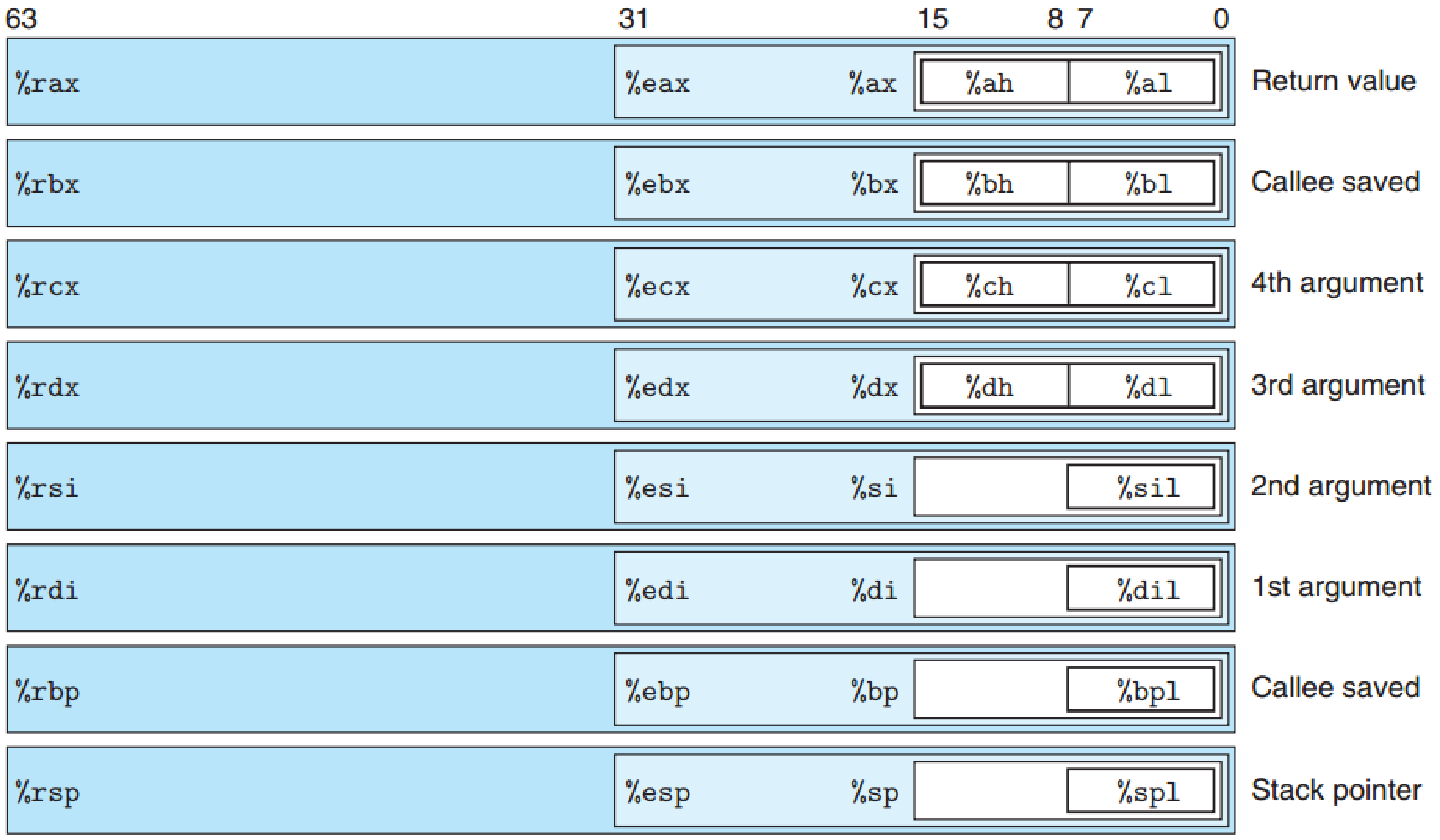
- If you can, look to the destination size

```
mov_   $0x7, %ebx
add_   %ecx, %edx
sar_ %al, %edi
sub_   0x63(%eax), %esi
```

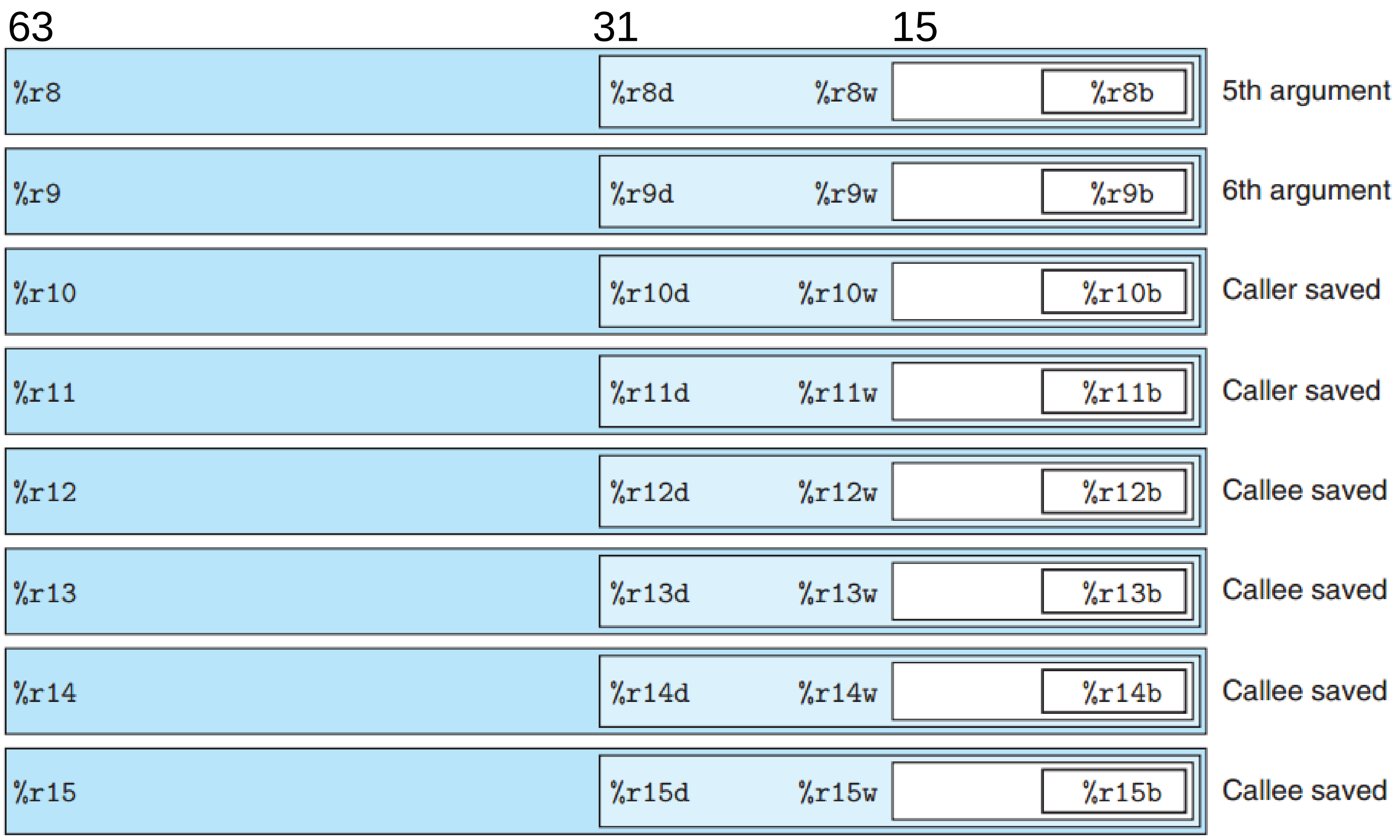
- When destination is memory, it can be ambiguous

```
mov_ $0xfed3, $0xc(%ebp)
```

Registers in x86



More Registers



x86 Reverse Engineering Problems

- Unless it specifies otherwise, assume function parameters make use of registers (for x86 problems)
  - i.e. first parameter stored in %rdi
  - Second parameter stored in %rsi
  - Similar to %ebp+8 and %ebp+12 convention in IA32

Reverse Engineering: Problem 3.59

```
int switch_prob(int x, int n) {
    int result = x;
    switch(n) {
        /* Fill in the code here */
    }
    return result;
}
```

```
(gdb) x/6w 0x80485d0
0x80485d0: 0x8048438  0x8048448  0x8048438  0x804843d
0x80485e0: 0x8048442  0x8048455
```

```
8048420 <switch_prob>:
8048420: 55                        push %ebp
8048421: 89 e5                     mov %esp,%ebp
8048423: 8b 45 08                  mov 0x8(%ebp),%eax
8048426: 8b 55 0c                  mov 0xc(%ebp),%edx
8048429: 83 ea 32                  sub $0x32,%edx
804842c: 83 fa 05                  cmp $0x5,%edx
804842f: 77 17                     ja 8048448 <switch_prob+0x28>
8048431: ff 24 95 d0 85 04 08      jmp *0x80485d0(,%edx,4)
8048438: c1 e0 02                  shl $0x2,%eax
804843b: eb 0e                     jmp 804844b <switch_prob+0x2b>
804843d: c1 f8 02                  sar $0x2,%eax
8048440: eb 09                     jmp 804844b <switch_prob+0x2b>
8048442: 8d 04 40                  lea (%eax,%eax,2),%eax
8048445: 0f af c0                  imul %eax,%eax
8048448: 83 c0 0a                  add $0xa,%eax
804844b: 5d                        pop %ebp
804844c: c3                        ret
```

```
                     8048420 <switch_prob>:
                     8048420: 55                         push %ebp
                     8048421: 89 e5                      mov %esp,%ebp
                     8048423: 8b 45 08                   mov 0x8(%ebp),%eax
                     8048426: 8b 55 0c                   mov 0xc(%ebp),%edx
                     8048429: 83 ea 32                   sub $0x32,%edx
                     804842c: 83 fa 05                   cmp $0x5,%edx
                     804842f: 77 17                      ja 8048448 <switch_prob+0x28>
                     8048431: ff 24 95 d0 85 04 08       jmp *0x80485d0(,%edx,4)
JTAB[0] JTAB[2]      8048438: c1 e0 02                   shl $0x2,%eax
                     804843b: eb 0e                      jmp 804844b <switch_prob+0x2b>
        JTAB[3]      804843d: c1 f8 02                   sar $0x2,%eax
                     8048440: eb 09                      jmp 804844b <switch_prob+0x2b>
        JTAB[4]      8048442: 8d 04 40                   lea (%eax,%eax,2),%eax
                     8048445: 0f af c0                   imul %eax,%eax
        JTAB[1]      8048448: 83 c0 0a                   add $0xa,%eax
                     804844b: 5d                         pop %ebp
                     804844c: c3                         ret
```

```
int switch_prob(int x, int n) {
    int result = x;
    switch(n) {
    case 50:
    case 52:
        result = x << 2;
        break;
    case 53:
        result = x >> 2;
        break;
    case 54:
        result = 3*x;
        result *= result;
    default:
        result += 10;
    }
    return result;
```

Problem 3.58

```
Int switch3(int *p1, int *p2, mode_t action) {
    int result = 0;
    switch(action) {
    case MODE_A:
    case MODE_B:
    case MODE_C:
    case MODE_D:
    default:
    }
    return result;
}
```

```
.L17:                              MODE_E
  movl    $17, %edx
  jmp     .L19
.L13:                              MODE_A
  movl    8(%ebp), %eax
  movl    (%eax), %edx
  movl    12(%ebp), %ecx
  movl    (%ecx), %eax
  movl    8(%ebp), %ecx
  movl    %eax, (%ecx)
  jmp     .L19
.L14:                              MODE_B
  movl    12(%ebp), %edx
  movl    (%edx), %eax
  movl    %eax, %edx
  movl    8(%ebp), %ecx
  addl    (%ecx), %edx
  movl    12(%ebp), %eax
  movl    %edx, (%eax)
  jmp     .L19
.L15:                              MODE_C
  movl    12(%ebp), %edx
  movl    $15, (%edx)
  movl    8(%ebp), %ecx
  movl    (%ecx), %edx
  jmp     .L19
.L16:                              MODE_D
  movl    8(%ebp), %edx
  movl    (%edx), %eax
  movl    12(%ebp), %ecx
  movl    %eax, (%ecx)
  movl    $17, %edx
.L19:                              default
  movl    %edx, %eax               Set return value
```

Problem 3.58

```
result = -1;
switch(action) {
case MODE_A:
    result = *p1;
    *p1 = *p2;
    break;
case MODE_B:
    result = *p2;
    *p2 = *p1;
    break;
case MODE_C:
    *p2 = 15;
    result = *p1;
    break;
case MODE_D:
    *p2 = *p1;
    result = 17;
    break;
case MODE_E:
    result = 17
default:;}
return result;
```

Problem 3.65

```
typdef struct {
    short x[A][B];
    int y;
} str 1;
typedef struct {
    char array[A];
    int t;
    short s[B];
    int u;
    } str2;
```

```
void setVal(str1 *p,str2 *q){
    int v1 = q->t;
    int v2 = q->u;
    p->y = v1+v2;
}
```

Problem 3.65

```
movl 12(%ebp), %eax
movl 36(%eax), %edx
addl 12(%eax), %edx
movl 8(%ebp), %eax
movl %edx, 92(%eax)
```

What is the Value of A and B?

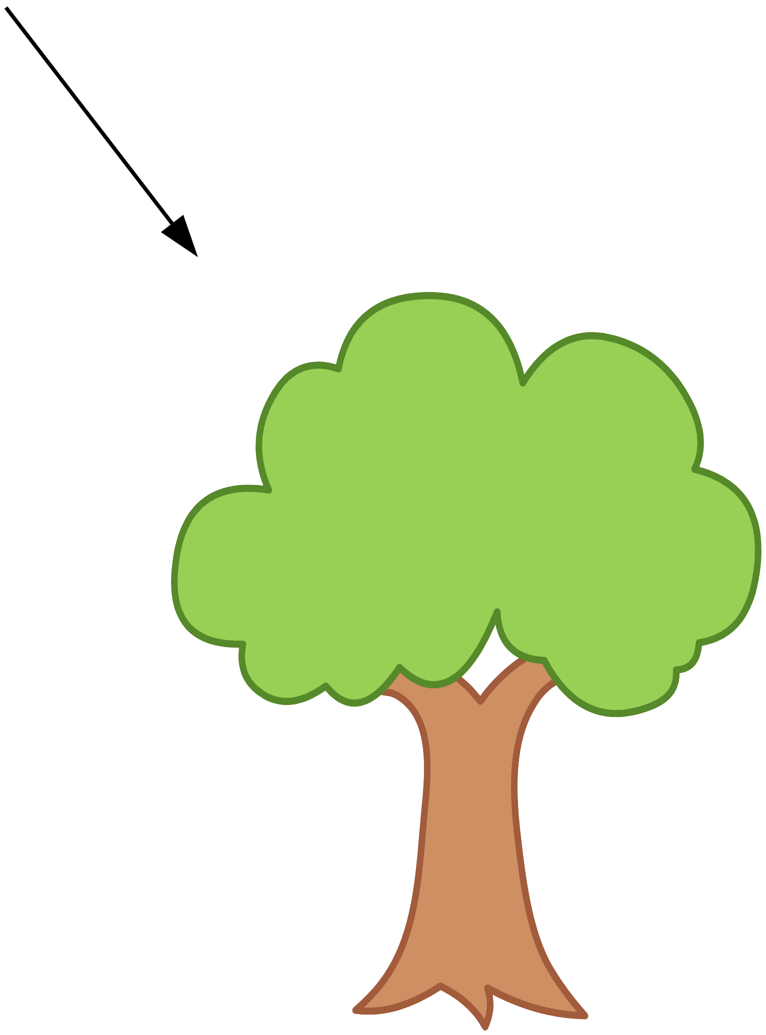
The solution is unique

Problem 3.69

```
typedef struct ELE *tree_ptr;
struct ELE {
    long val;
    tree_ptr left;
    tree_ptr right;
}
long trace(tree_ptr tp);     // What does this do?
```

```
trace:
    movl $0, %eax
    testq %rdi, %rdi
    je .L3
.L5:
    movq (%rdi), %rax
    movq 16(%rdi), %rdi
    testq %rdi, %rdi
    jne .L5
.L3
    rep
    ret
```

```
long trace(tree_ptr tp) {
    long eax = 0;
    while (tp != 0) {
        eax = tp-> val;
        tp = tp-> right;
    }
    return eax;
}
```



Stack Discipline

Suppose ebp = 4, what is return address?

0				4				8				12			
A0	D3	7F	00	10	00	00	00	36	D3	77	E4	3D	D3	FF	B4

16				20				24				28			
BB	88	7F	00	FF	FF	39	24	D9	FD	11	00	00	00	FF	4B

32				36				40				44			
9C	11	3F	10	48	D0	99	24	89	D9	43	58	80	00	88	44

What is first parameter, second parameter if they are both ints?

Stack Discipline

ebp = 4 → return address = 0xE477D336

0				4				8				12			
A0	D3	7F	00	20	00	00	00	36	D3	77	E4	3D	D3	FF	B4

16				20				24				28			
BB	88	7F	00	FF	FF	39	24	D9	FD	11	00	00	00	FF	4B

32				36				40				44			
9C	11	3F	10	48	D0	99	24	89	D9	43	58	80	00	88	44

Param1 = 0xB4FFD33D Param 2 = 0x7F88BB

Stack Discipline

ebp = 4. What is first parameter of the caller?

0				4				8				12			
A0	D3	7F	00	20	00	00	00	36	D3	77	E4	3D	D3	FF	B4

16				20				24				28			
BB	88	7F	00	FF	FF	39	24	D9	FD	11	00	00	00	FF	4B

32				36				40				44			
9C	11	3F	10	48	D0	99	24	89	D9	43	58	80	00	88	44

Stack Discipline

ebp = 4. What is first parameter of the caller?

0				4				8				12			
A0	D3	7F	00	20	00	00	00	36	D3	77	E4	3D	D3	FF	B4

16				20				24				28			
BB	88	7F	00	FF	FF	39	24	D9	FD	11	00	00	00	FF	4B

32				36				40				44			
9C	11	3F	10	48	D0	99	24	89	D9	43	58	80	00	88	44

Old ebp = 0x00000020 = 32

1st param = 0x5843D989

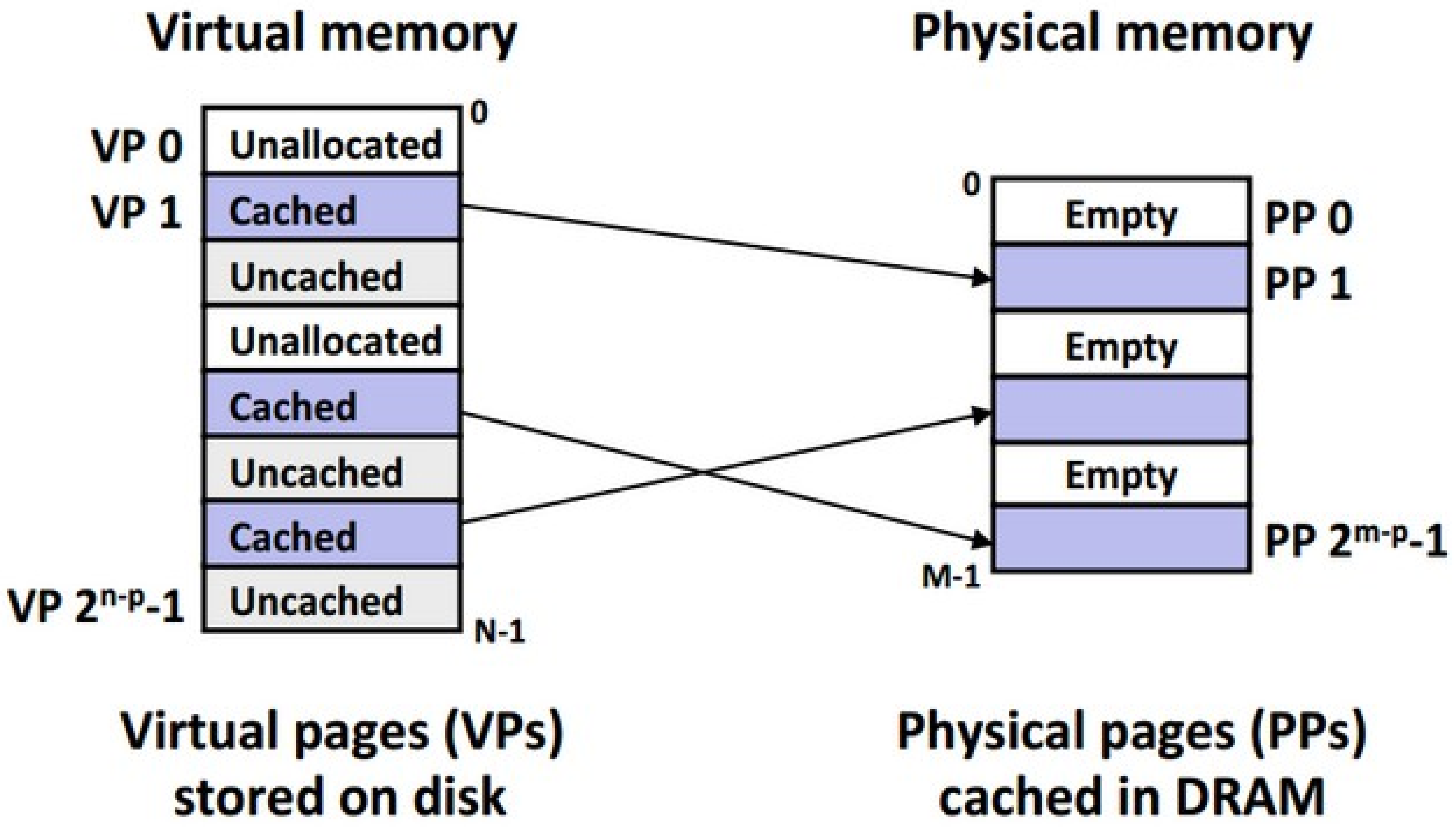
Other Misc Notes

- Linux alignment policy: all 2B data types (short) are aligned on multiples of 2B
- Any primitive larger than 2B is aligned on multiples of 4B
  - Even 8B primitives like doubles and long longs

CHAPTER 6: Virtual Memory

Page Table and TLB

Remember how VM works



Slide made by Garrett

VM basics

- Virtual Address (VA) space is 8GB
  - How long is a virtual address?
- Page size is 4KB
  - How many bits needed to represent page offset
- How many bits to represent the Virtual page number?
- What is the virtual page number for this address:
  - &x = 0x13477fad7

VM basics

- Virtual Address (VA) space is 8GB = (2^3)*(2^30)B
  - 33 bits
- Page size is 4KB = (2^2)*(2^10) B
  - 12 bits for the page offset
- Virtual page number is 33 – 12 = 21 bits
- What is the virtual page number for this address:
  - &x = 0x13477fad7
  - Page number is 0x13477f

VM basics

- VA space is 8GB, page size is 4KB
- How many pages in the PT?

VM basics

- VA space is 8GB, page size is 4KB
- How many pages in the PT?
  - 2^21 possible virtual pages == 2^21 page table entries

Assume 12 bit Virtual address, 10 bit Physical Address, Page size 16 B.

Virtual Address:

Page Table:

V	Physical Page Number
0	0x2C
1	0x37
1	0x2B
0	null
1	0x1F
0	0x08
1	0x1A
	...

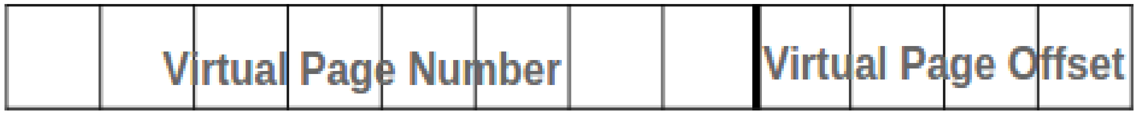
bits VPO = ?
bits PPO = ?

bits VPN = ?
bits PPN = ?

Physical Address:

Assume 12 bit Virtual address, 10 bit Physical Address, Page size 16 B.

Virtual Address:



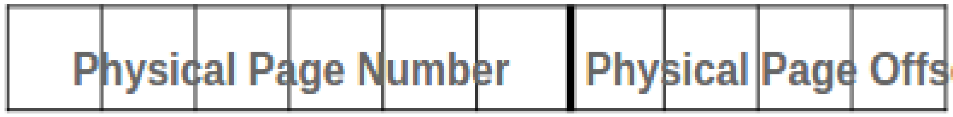
Page Table:

V	Physical Page Number
0	0x2C
1	0x37
1	0x2B
0	null
1	0x1F
0	0x08
1	0x1A
	...

bits VPO = 4
bits PPO = 4

bits VPN = 8
bits PPN = 6

Physical Address:



Assume 12 bit Virtual address, 10 bit Physical Address, Page size 16 B.

Virtual Address: **0x03A**

0	0	0	0	0	0	1	1	1	0	1	0

Page Table:

V	Physical Page Number
0	0x2C
1	0x37
1	0x2B
0	null
1	0x1F
0	0x08
1	0x1A
	...

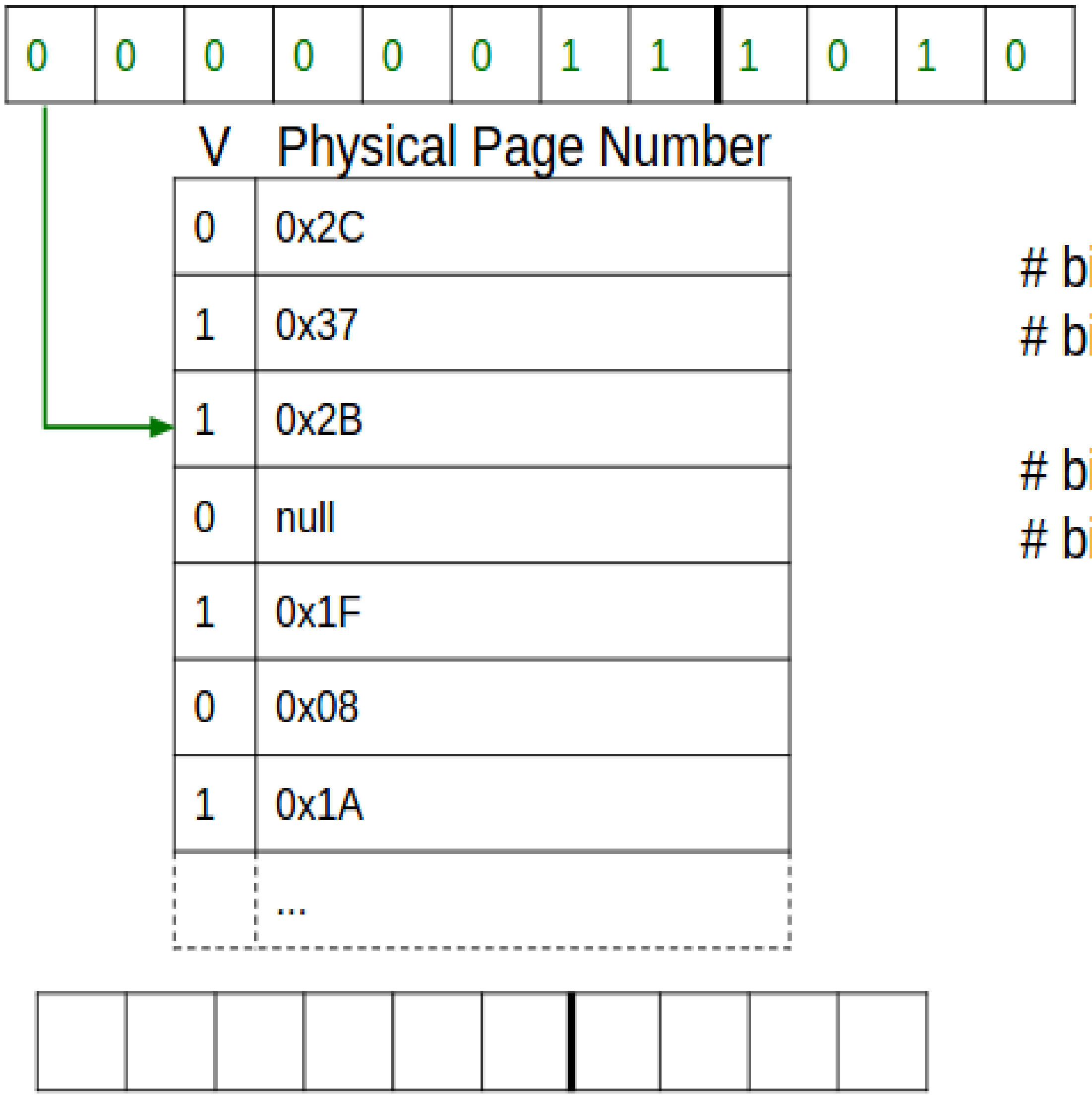
bits VPO = 4
bits PPO = 4

bits VPN = 8
bits PPN = 6

Physical Address:

Assume 12 bit Virtual address, 10 bit Physical Address, Page size 16 B.

Virtual Address: **0x03A**



0	0	0	0	0	0	1	1	1	0	1	0

Page Table:

V	Physical Page Number
0	0x2C
1	0x37
1	0x2B
0	null
1	0x1F
0	0x08
1	0x1A
	...

bits VPO = 4
bits PPO = 4

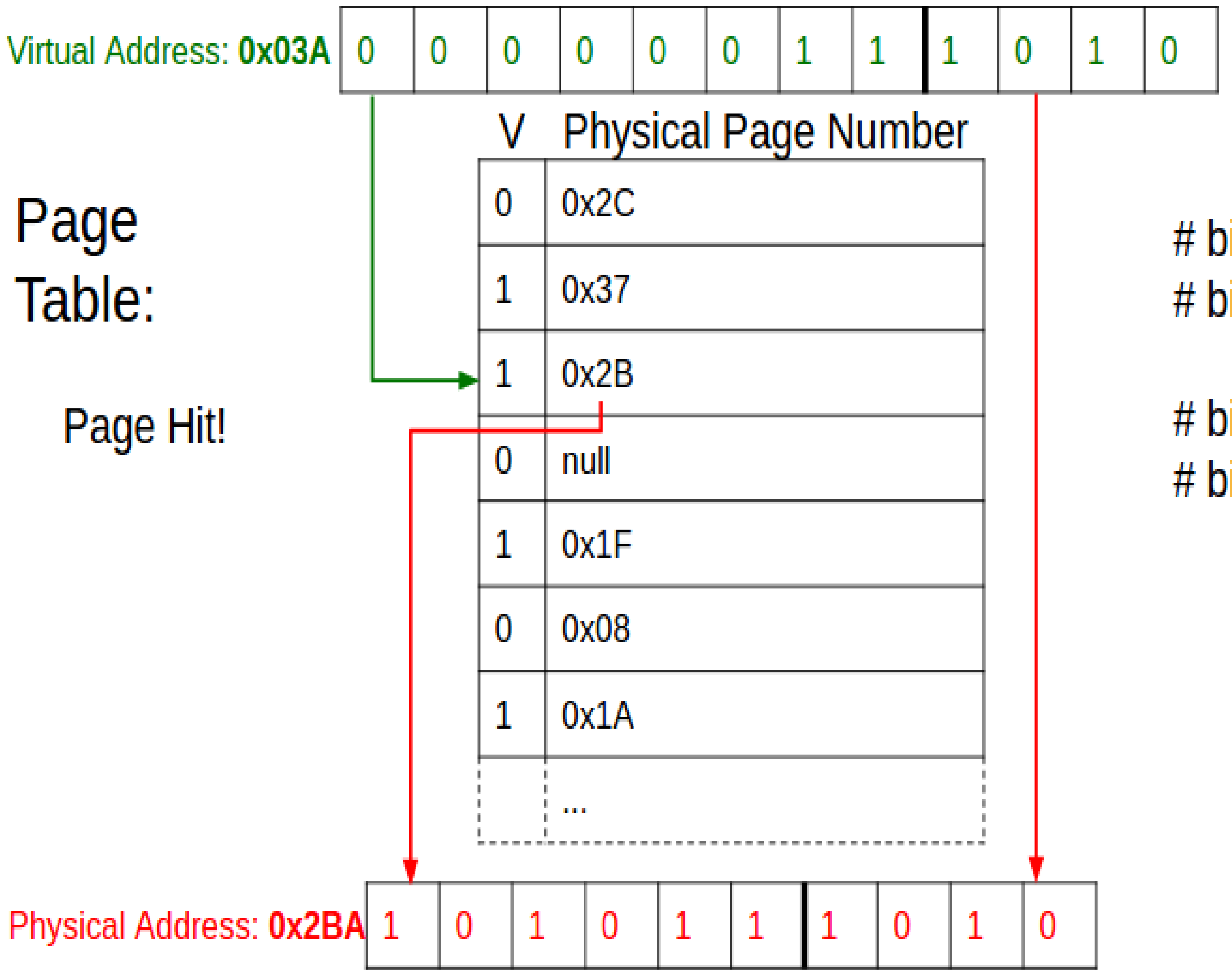
bits VPN = 8
bits PPN = 6

Physical Address:

Assume 12 bit Virtual address, 10 bit Physical Address, Page size 16 B.

Page Table:

Page Hit!



V	Physical Page Number
0	0x2C
1	0x37
1	0x2B
0	null
1	0x1F
0	0x08
1	0x1A
	...

# bits VPO = 4

# bits PPO = 4

# bits VPN = 8

# bits PPN = 6

Assume 12 bit Virtual address, 10 bit Physical Address, Page size 16 B.

Virtual Address: **0x047**

0	0	0	0	0	1	0	0	0	1	1	1

Page Table:

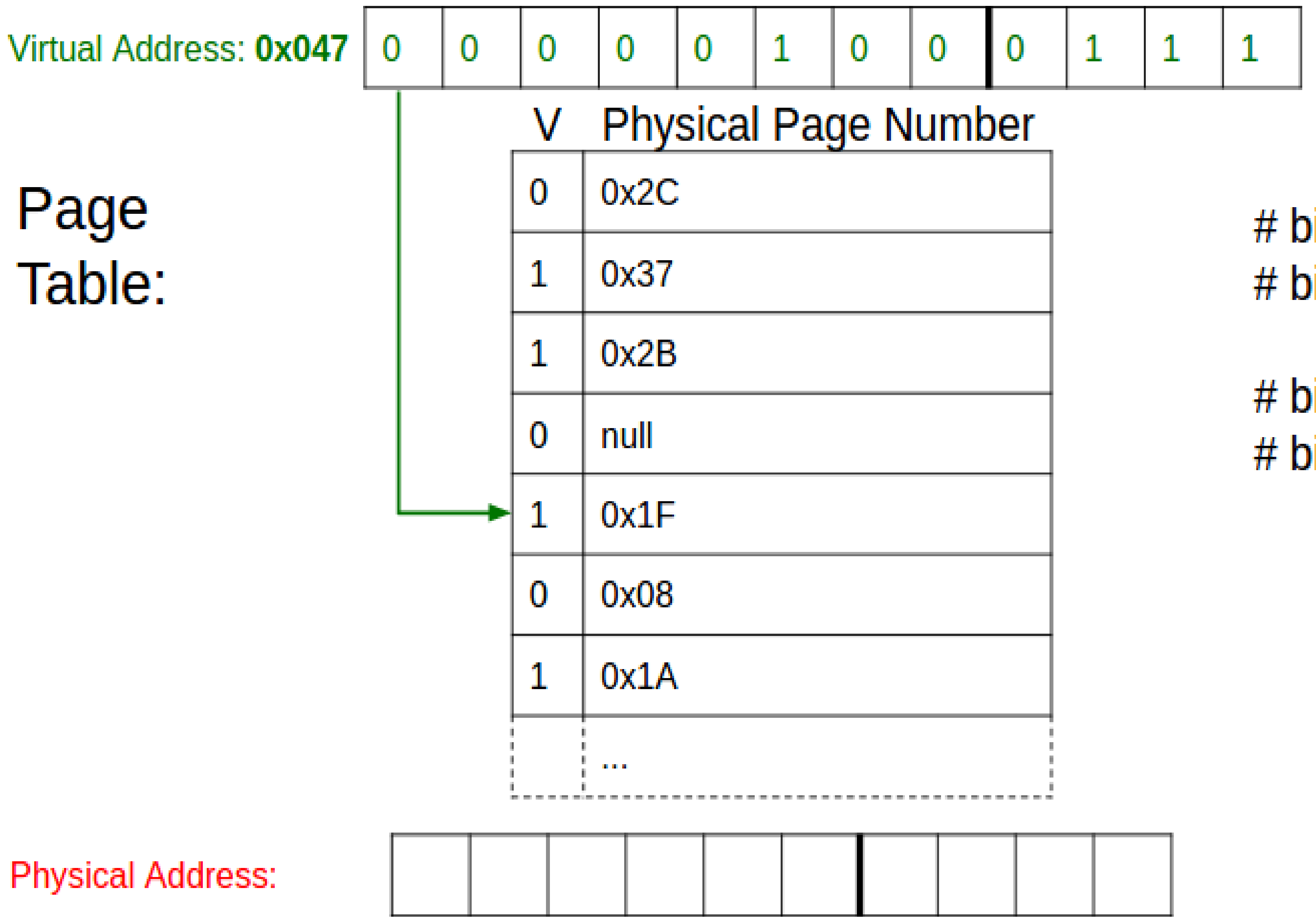
V	Physical Page Number
0	0x2C
1	0x37
1	0x2B
0	null
1	0x1F
0	0x08
1	0x1A
	...

bits VPO = 4
bits PPO = 4

bits VPN = 8
bits PPN = 6

Physical Address:

Assume 12 bit Virtual address, 10 bit Physical Address, Page size 16 B.

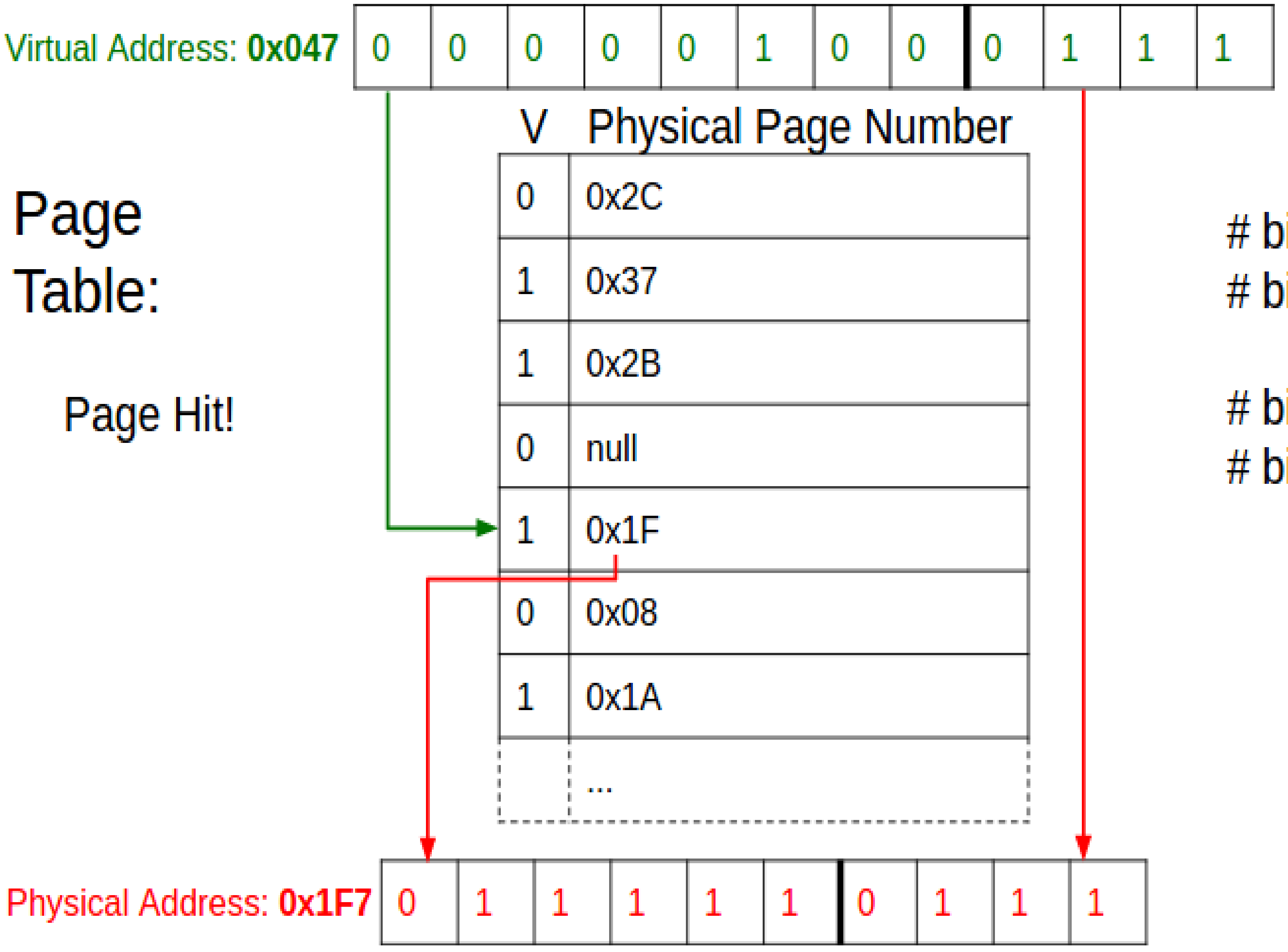


V	Physical Page Number
0	0x2C
1	0x37
1	0x2B
0	null
1	0x1F
0	0x08
1	0x1A
	...

bits VPO = 4
bits PPO = 4

bits VPN = 8
bits PPN = 6

Assume 12 bit Virtual address, 10 bit Physical Address, Page size 16 B.



V	Physical Page Number
0	0x2C
1	0x37
1	0x2B
0	null
1	0x1F
0	0x08
1	0x1A
	...

bits VPO = 4
bits PPO = 4

bits VPN = 8
bits PPN = 6

Assume 12 bit Virtual address, 10 bit Physical Address, Page size 16 B.

Virtual Address: **0x05F**

0	0	0	0	0	1	0	1	1	1	1	1

Page Table:

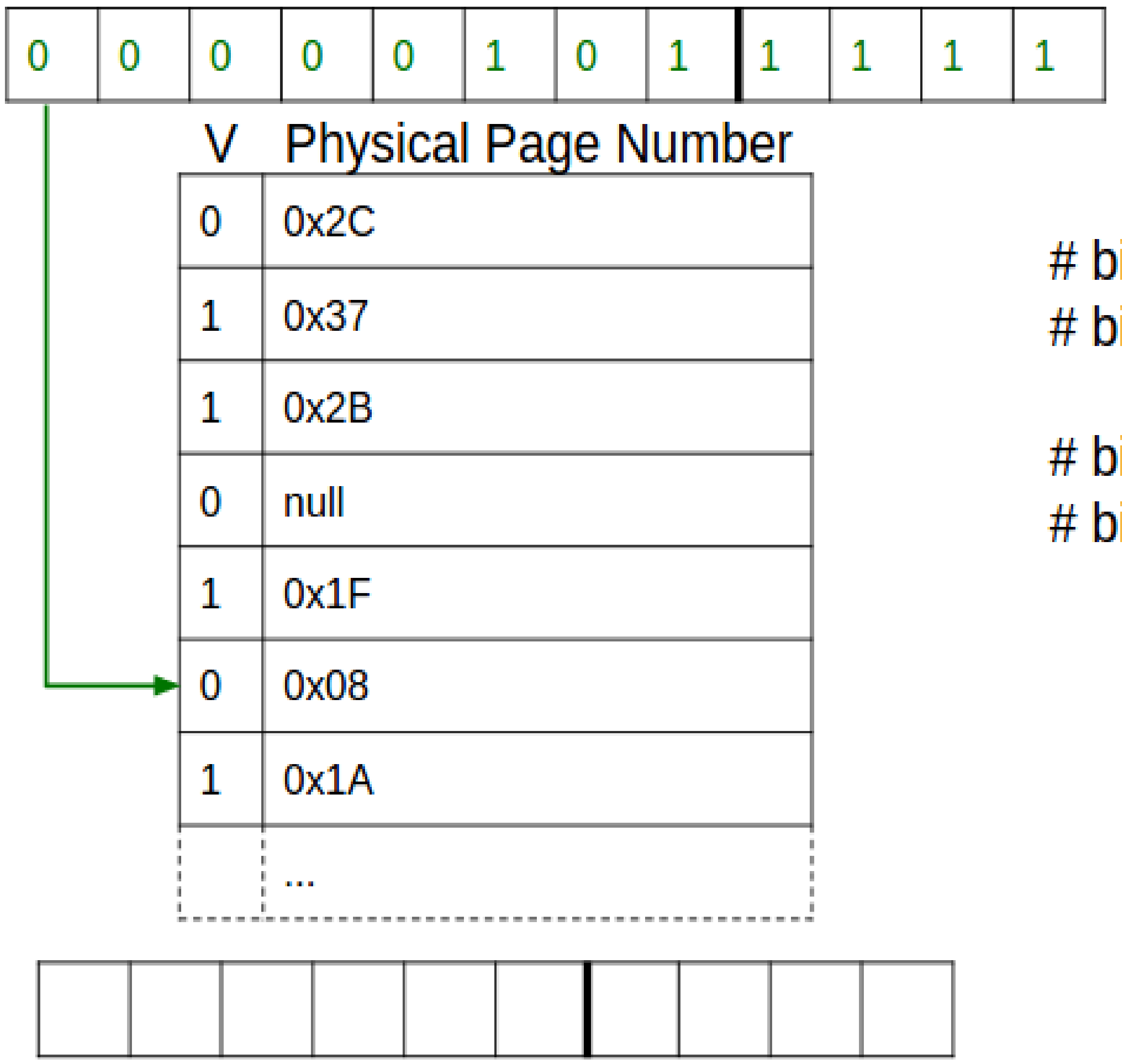
V	Physical Page Number
0	0x2C
1	0x37
1	0x2B
0	null
1	0x1F
0	0x08
1	0x1A
	...

bits VPO = 4
bits PPO = 4

bits VPN = 8
bits PPN = 6

Physical Address:

Assume 12 bit Virtual address, 10 bit Physical Address, Page size 16 B.



Virtual Address: **0x05F**

0	0	0	0	0	1	0	1	1	1	1	1

Page Table:

V	Physical Page Number
0	0x2C
1	0x37
1	0x2B
0	null
1	0x1F
0	0x08
1	0x1A
	...

Page Fault!
=(
Page is not resident in Memory

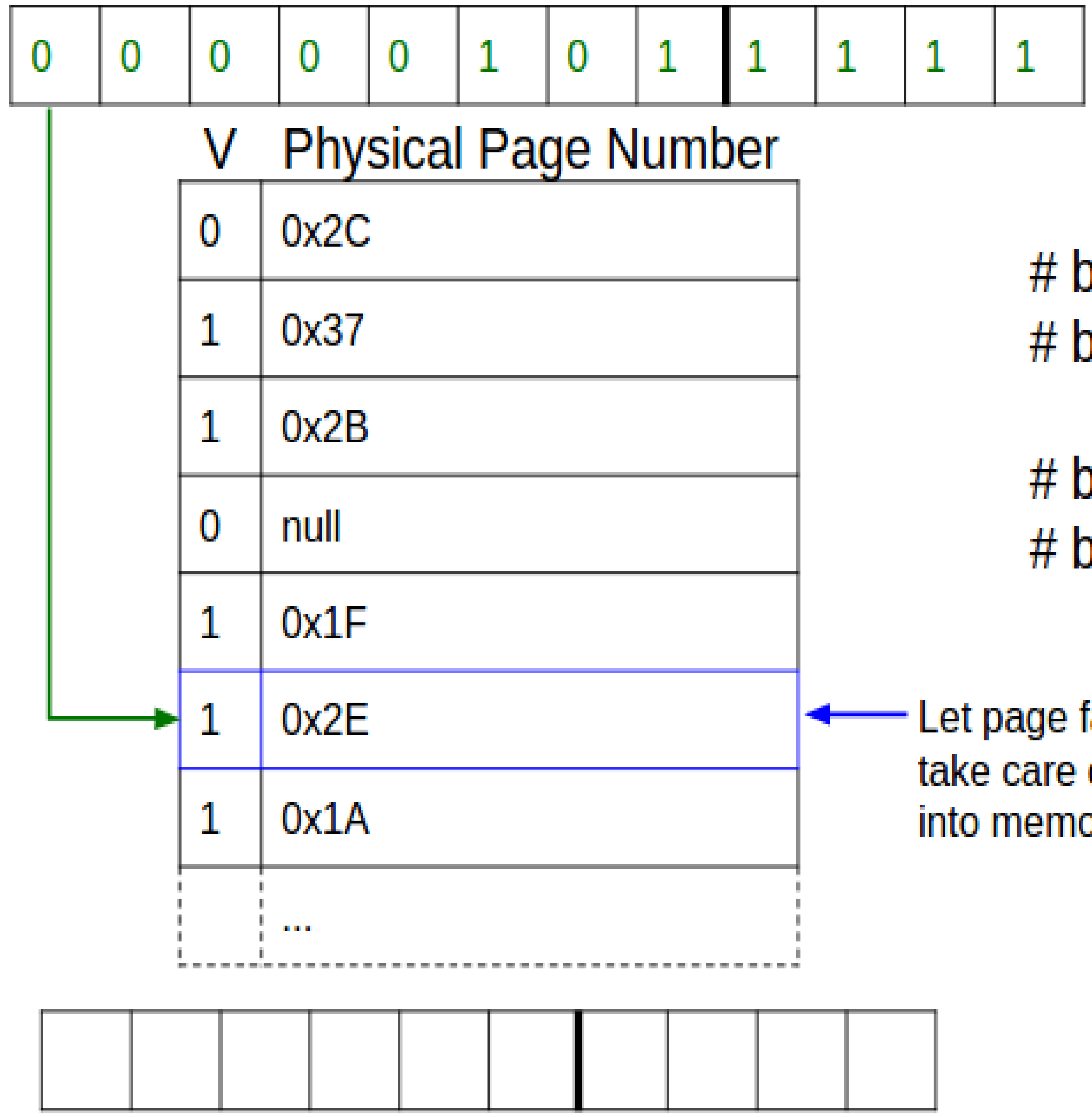
bits VPO = 4
bits PPO = 4

bits VPN = 8
bits PPN = 6

Physical Address:

Assume 12 bit Virtual address, 10 bit Physical Address, Page size 16 B.

Virtual Address: **0x05F**



0	0	0	0	0	1	0	1	1	1	1	1

Page Table:

V	Physical Page Number
0	0x2C
1	0x37
1	0x2B
0	null
1	0x1F
1	0x2E
1	0x1A
	...

# bits VPO = 4

# bits PPO = 4

# bits VPN = 8

# bits PPN = 6

Page Fault!

=(

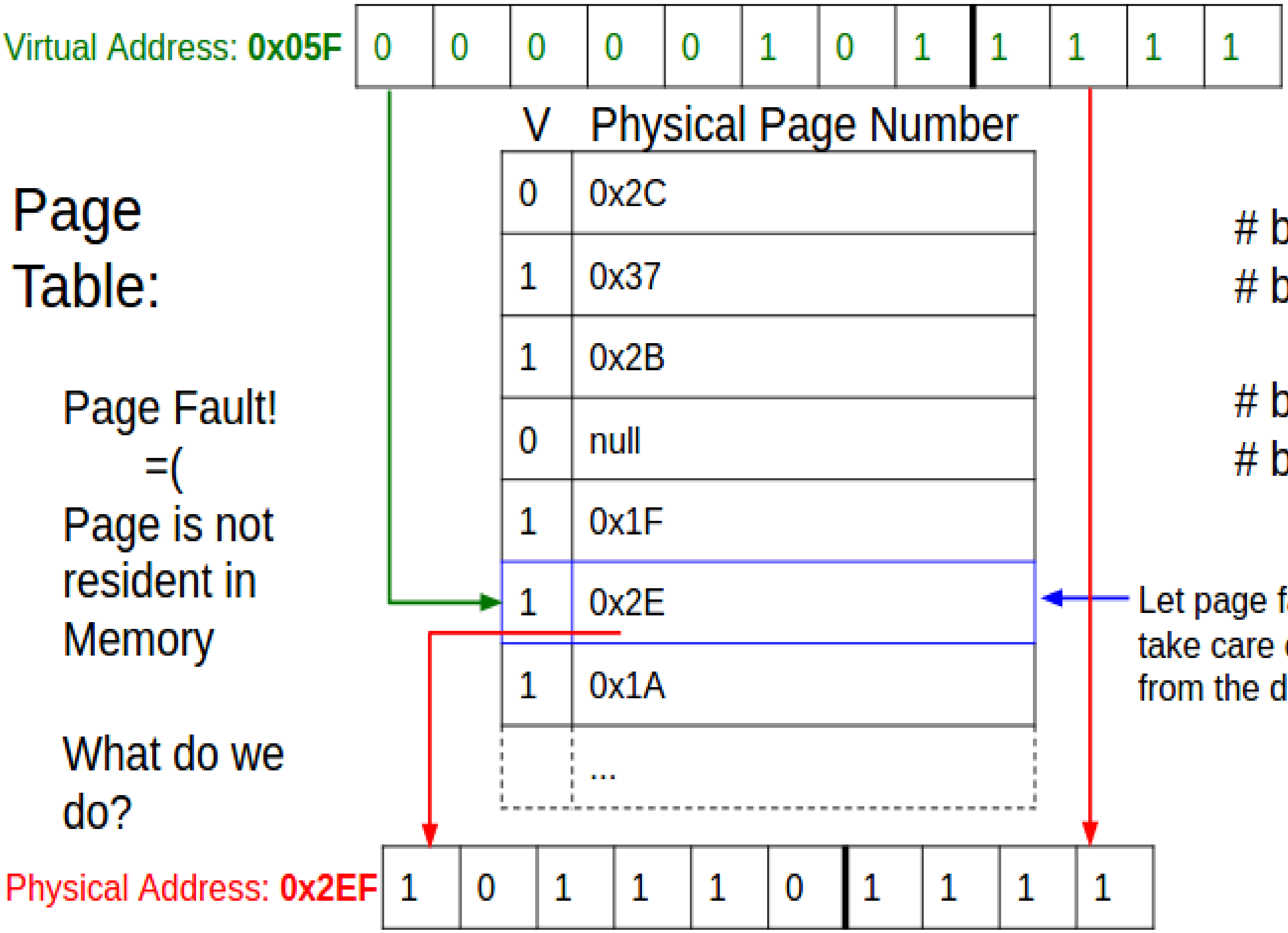
Page is not resident in Memory

Let page fault exception handler take care of bringing the page into memory, so we can use it.

What do we do?

Physical Address:

Assume 12 bit Virtual address, 10 bit Physical Address, Page size 16 B.



Page Table:

V	Physical Page Number
0	0x2C
1	0x37
1	0x2B
0	null
1	0x1F
1	0x2E
1	0x1A
	...

Page Fault!
=(
Page is not resident in Memory

What do we do?

bits VPO = 4
bits PPO = 4

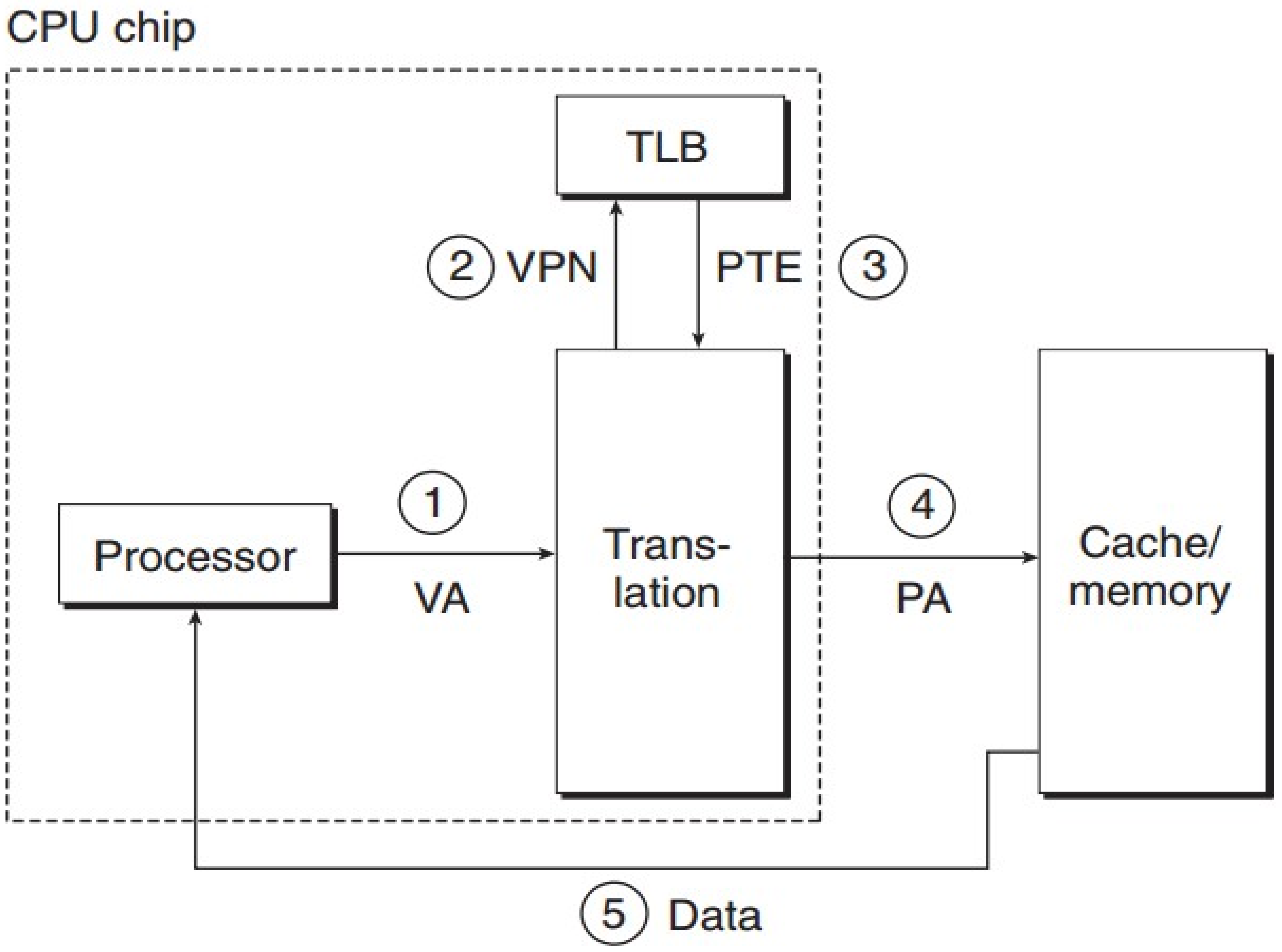
bits VPN = 8
bits PPN = 6

Let page fault exception handler take care of bringing the page in from the disc so we can use it.

Improving Performance of Cache

- Last time we noticed that using a PT could be expensive
- Need one I/O to read the PT and retrieve addr translation
- Need one more I/O to fetch the data at the correct PA

Using TLB to Cache the PT

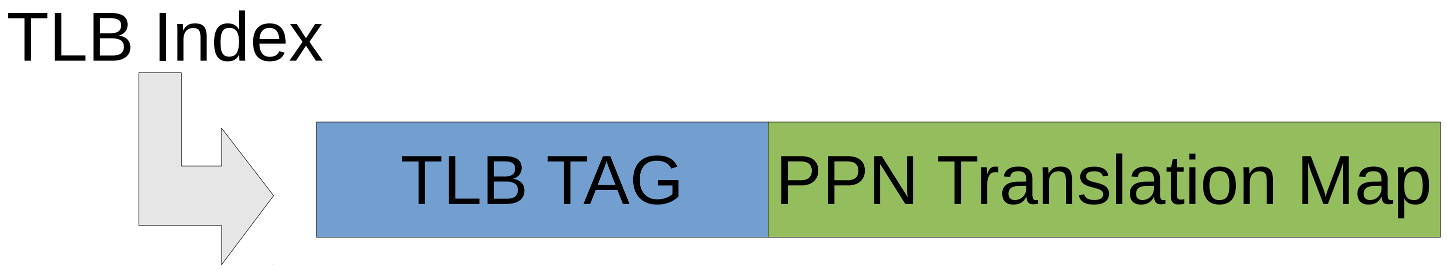


TLB is Just like a Cache

- Caches Page Table entries (e.g. caches Virtual → page number translation mappings)

TLB is Just like a Cache

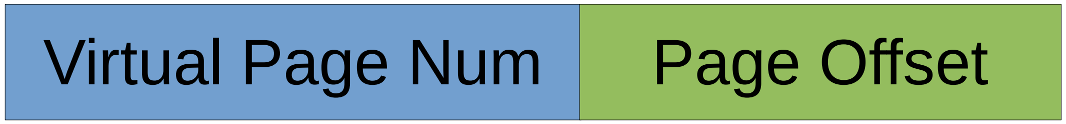
- Caches Page Table entries (e.g. caches Virtual → page number translation mappings)
- One Cache line contains one translation



- And TLB is typically highly associative

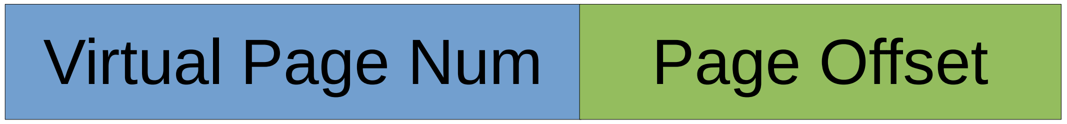
Determining TLB Index and Tag

- Before, we chopped our VA into 2 parts (VPN and VPO)

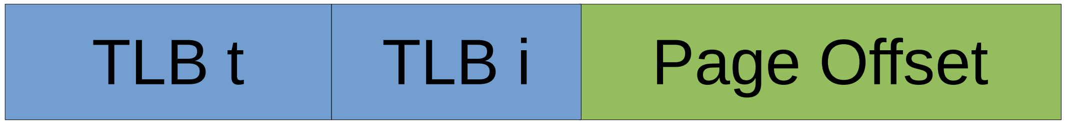


Determining TLB Index and Tag

- Before, we chopped our VA into 2 parts (VPN and VPO)



- Now, divide VPN into TLB tag and index



TLB Example

- |VA| = 20 bits, |PA| = 18 bits
- Page Size = 4KB
- TLB is direct mapped, with 8 sets
  - How many bits do I need for:
    - Virtual Page Number (VPN)
    - Physical Page Number (PPN)
    - TLB Tag
    - TLB index

TLB Example

- |VA| = 20 bits, |PA| = 18 bits
- Page Size = 4KB
- TLB is direct mapped, with 8 sets
  - How many bits do I need for:
    - Virtual Page Number (VPN) = 8 bits
    - Physical Page Number (PPN) = 6 bits
    - TLB Tag = 5 bits
    - TLB index = 3 bits

TLB Example

What is the addresss translation for **VA = 0xEE000**?

	VALID	1	PPN Mapping
0	1	0x0A	b100100
1	1	0x17	b101010
2	0	0x11	b000001
3	1	0x0F	b011111
4	1	0x03	b111000
5	1	0x01	b010001
6	1	0x1D	b110011
0	1	0x13	b111000

TLB Example

What is the addresss translation for **VA = 0xEE000**?

	VALID	1	PPN Mapping
0	1	0x0A	b100100
1	1	0x17	b101010
2	0	0x11	b000001
3	1	0x0F	b011111
4	1	0x03	b111000
5	1	0x01	b010001
6	1	0x1D	b110011
0	1	0x13	b111000

PPN = b110011
→ **PA = 0x33000**

TLB Lookup Procedure

1. Given Virtual address, chop off lower bits corresponding to <u>Page Offset</u>
2. The remaining bits specify <u>Virtual Page Number</u>. Separate into TLB Tag and TLB index (same procedure as in caching)
3. Use TLB set-index as an index lookup into the TLB
4. If there is a valid entry in the set with matching tags, take the contents of the block (the <u>Physical Page Number</u>)

TLB and Page Table

Small TLB

VALID	1	PPN Mapping
1	0x0A	b100100
1	0x17	b101010
0	0x11	b000001
1	0x0F	b011111
1	0x03	b111000
1	0x01	b010001
1	0x1D	b110011
1	0x13	b111000

PT entry 0xEE

VALID	PPN Mapping
	...
1	b110011
	...

BIG Page Table

VA = 0xEE000

→ VPN = 0xEE

→ TLB tag = 0x1D index = 6

TLB and Page Table

- Valid bits **do not** mean the same thing in PT and TLB
  - In Page Table, valid indicates whether that page is in memory
    - If not generate a **PAGE FAULT**
  - In TLB, valid indicates whether that cache line is valid
- e.g. a line in the TLB might be invalid, but PT lookup may yield a valid Physical Page translation
  - i.e. can be a **TLB miss** without generating a **Page Fault**
- If entry is not valid in PT, should **not** be valid in TLB

TLB Miss

- If the TLB lookup results in a miss, must fetch the underlying translation from the PT (<u>just like a cache miss</u>)
  - If the PT entry is invalid, must also generate a **PAGE FAULT**

TLB Miss

Small TLB

VALID	1	PPN Mapping
1	0x0A	b100100
1	0x17	b101010
0	0x11	b000001
1	0x0F	b011111
1	0x03	b111000
1	0x01	b010001
1	0x1D	b110011
1	0x13	b111000

Now Consider:

VA = 0xED000
→ **VPN = 0xED**
→ **TLB tag = 0x1D index = 5**

VALID	PPN Mapping
1	b000010
1	b110011
	...

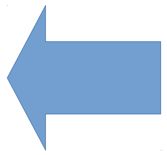
BIG Page Table

TLB Miss

Small TLB

VALID	1	PPN Mapping
1	0x0A	b100100
1	0x17	b101010
0	0x11	b000001
1	0x0F	b011111
1	0x03	b111000
1	0x01	b010001
1	0x1D	b110011
1	0x13	b111000

TLB MISS



VALID	PPN Mapping
1	b000010
1	b110011
	…

BIG Page Table

VA = 0xED000
→ VPN = 0xED
→ TLB tag = 0x1D index = 5

TLB Miss

Small TLB

VALID	1	PPN Mapping
1	0x0A	b100100
1	0x17	b101010
0	0x11	b000001
1	0x0F	b011111
1	0x03	b111000
1	0x01	b010001
1	0x1D	b110011
1	0x13	b111000

PT lookup
Entry 0xED

VALID	PPN Mapping
1	b000010
1	b110011
	...

BIG Page Table

VA = 0xED000
→ **VPN = 0xED**
→ **TLB tag = 0x1D index = 5**

TLB Miss

Small TLB

VALID	1	PPN Mapping
1	0x0A	b100100
1	0x17	b101010
0	0x11	b000001
1	0x0F	b011111
1	0x03	b111000
1	0x**1D**	b**000010**
1	0x1D	b110011
1	0x13	b111000

Block Move

VALID	PPN Mapping
1	b000010
1	b110011
	...

BIG Page Table

VA = 0xED000
→ **VPN = 0xED**
→ **TLB tag = 0x1D index = 5**

THE END

Good luck on Finals!!