Week 10

Recall: Concurrent Processes

```
int main()
{
        pid_t pid;
        int x = 1;

        pid = fork();
        if (pid == 0) {
                printf("child : x=%d\n", ++x);
                exit(0);
        }

        printf("parent : x=%d\n", --x);
        exit(0);
}
```

Recall: Concurrent Processes

```
int main()
{
        pid_t pid;
        int x = 1;

        pid = fork();
        if (pid == 0) {
                printf("child : x=%d\n", ++x);
                exit(0);
        }

        printf("parent : x=%d\n", --x);
        exit(0);
}
```

Main Process
pid = pid
x = 1

```
if (pid == 0) {
 printf("child : x=%d\n", ++x);
 exit(0);
 }
printf("parent: x=%d\n", --x);
exit(0);
```

Child Process
pid = 0
x = 1

```
if (pid == 0) {
 printf("child : x=%d\n", ++x);
 exit(0);
 }
printf("parent: x=%d\n", --x);
exit(0);
```

```
>> parent : x=0
>> child : x=2
```

Recall: Concurrent Processes

- fork() duplicates the **context** of the parent, only changing the PID of the child
  - **context**: registers, instruction counter, condition codes
- Each process has a separate **address space**, creating a safe environment but inhibiting communication
  - Child and parent both have their own “x”
- Process which terminate go into suspended (zombie) state until reaped
  - waitpid(pid_t, int *status, int options)

Threads

- No children but “peers” sharing virtual memory of a process
  - Main thread -> peer threads
  - Less hierarchy
- Each thread has its own **context** and **stack**
  - The rest of the address space is shared
  - Same scheduling procedure as processes, but the context switch excludes VM

Traditional View of a Process

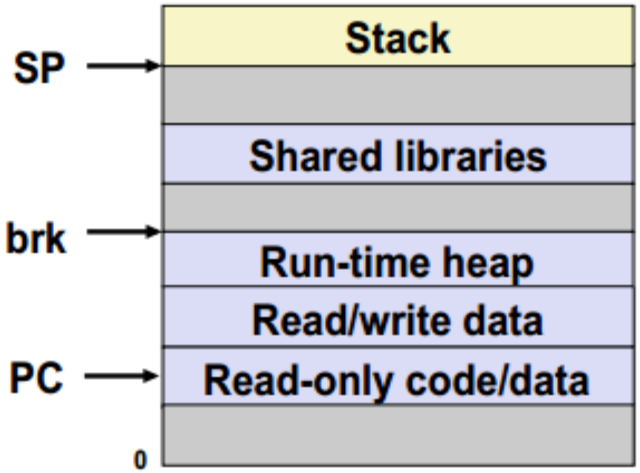
- Process = process context + code, data, and stack

Process context

Program context:
- Data registers
- Condition codes
- Stack pointer (SP)
- Program counter (PC)

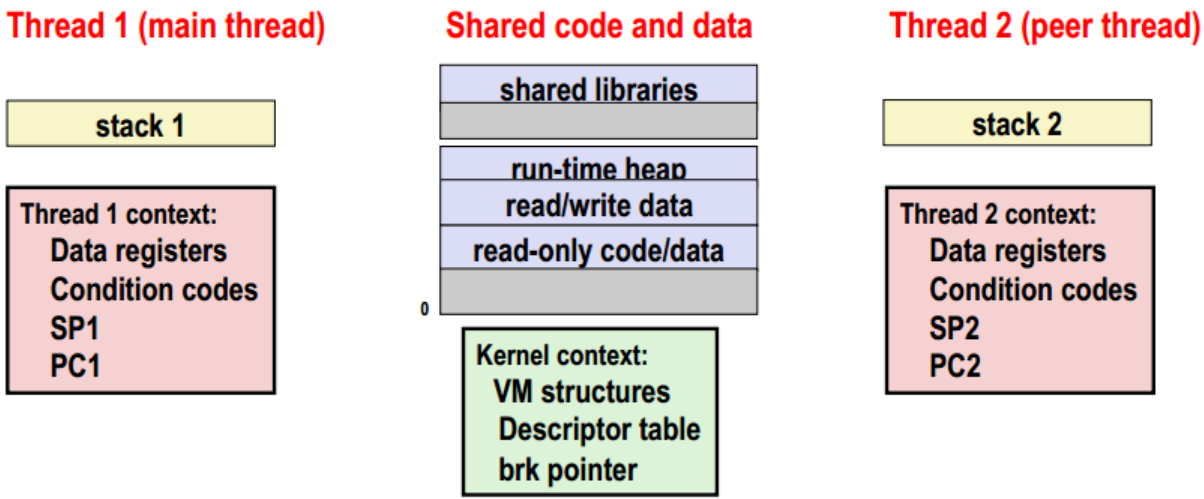
Kernel context:
- VM structures
- Descriptor table
- brk pointer

Code, data, and stack



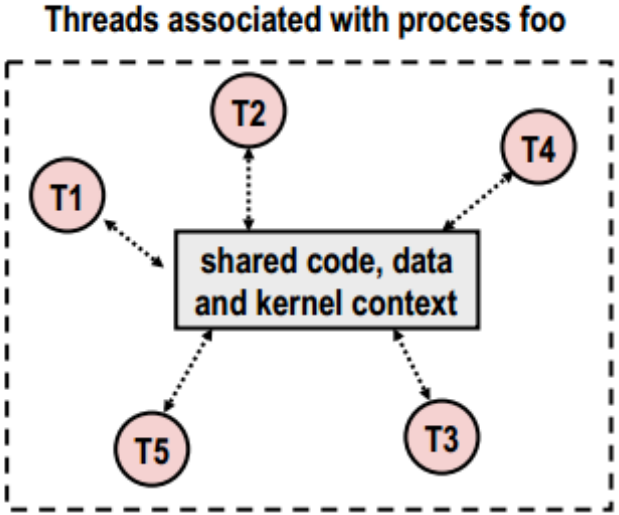
Alternate View of a Process

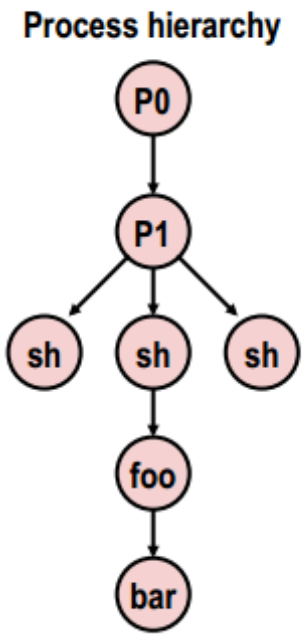
- Process = thread + code, data, and kernel context



Logical View of Threads

- **Threads associated with process form a pool of peers**
  - Unlike processes which form a tree hierarchy





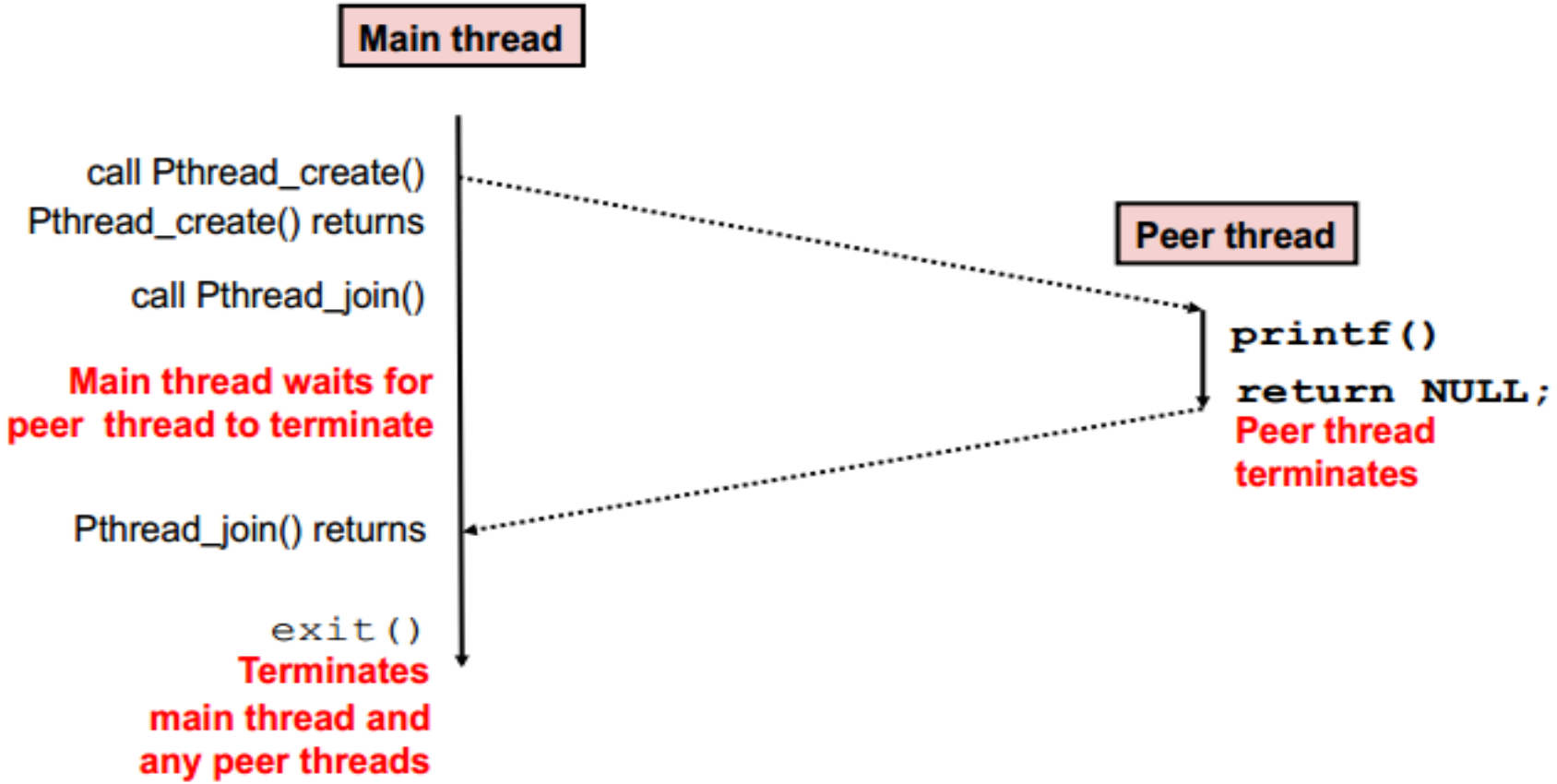
Simple Posix Threads (-pthread)

- int pthread_create(pthread_t *tid, pthread_attr_t *attr, func *f, void *arg);
  - tid = thread id
  - attr = thread attributes (less important for this class)
  - f = base function for thread
  - arg = argument struct: can only pass 1 argument, so (void *) gives you the most flexibility
  - returns 0 if OK, -1 if error.
- int pthread_join(pthread_t tid, void **retval);
  - Waits for tid to terminate and reaps the thread
  - Gets a pointer to a return value. Also a (void *) for maximum flexibility

Example

```
#include <pthread.h>
#include <stdio.h>

void *thread(void *vargp) {
   printf("Hello, world!\n");
   return NULL;
 }

int main() {
   pthread_t tid;
   pthread_create(&tid, NULL, thread, NULL);
   pthread_join(tid, NULL);
   exit(0);
 }
```

Execution of Threaded "hello, world"



More pthreads

- Many ways to terminate threads
  - implicit termination by returning
  - self can call: void pthread_exit(void *retval);
    - It can then be reaped by a peer calling pthread_join() on it
  - another peer can call: void pthread_cancel(pthread_t tid);
  - exit(): ends the entire process, terminating all threads
- void pthread_detach(pthread tid);
  - “unpeers” the thread. Does not need to be reaped, and can no longer be cancelled

Pros and Cons of Thread-Based Designs

- **+ Easy to share data structures between threads**
  - e.g., logging information, file cache
- **+ Threads are more efficient than processes**

- **– Unintentional sharing can introduce subtle and hard-to-reproduce errors!**

Concurrent Programming is Hard!

- **Classical problem classes of concurrent programs:**
  - ***Races:*** outcome depends on arbitrary scheduling decisions elsewhere in the system
    - Example: who gets the last seat on the airplane?
  - ***Deadlock:*** improper resource allocation prevents forward progress
    - Example: traffic gridlock
  - ***Livelock / Starvation / Fairness***: external events and/or system scheduling decisions can prevent sub-task progress
    - Example: people always jump in front of you in line

Shared Variables in Threaded C Programs

- **Question: Which variables in a threaded C program are shared?**
  - The answer is not as simple as "*global variables are shared*" and "*stack variables are private*"

- **Requires answers to the following questions:**
  - What is the memory model for threads?
  - How are instances of variables mapped to memory?
  - How many threads might reference each of these instances?

- ***Def:* A variable x is *shared* if and only if multiple threads reference some instance of x.**

Threads Memory Model

- **Conceptual model:**
  - Multiple threads run within the context of a single process
  - Each thread has its own separate thread context
    - Thread ID, stack, stack pointer, PC, condition codes, and GP registers
  - All threads share the remaining process context
    - Code, data, heap, and shared library segments of the process virtual address space
    - Open files and installed handlers
- **Operationally, this model is not strictly enforced:**
  - Register values are truly separate and protected, but…
  - Any thread can read and write the stack of any other thread

The mismatch between the conceptual and operation model is a source of confusion and errors

Mapping Variable Instances to Memory

- **Global variables**
  - *Def:* Variable declared outside of a function
  - **Virtual memory contains exactly one instance of any global variable**
- **Local variables**
  - *Def:* Variable declared inside function without `static` attribute
  - **Each thread stack contains one instance of each local variable**
- **Local static variables**
  - *Def:* Variable declared inside function with the `static` attribute
  - **Virtual memory contains exactly one instance of any local static variable.**

Mapping Variable Instances to Memory

Global var: 1 instance (`ptr [data]`)

Local vars: 1 instance (`i.m, msgs.m`)

```
char **ptr;  /* global var */

int main()
{
    long i;
    pthread_t tid;
    char *msgs[2] = {
        "Hello from foo",
        "Hello from bar"
    };

    ptr = msgs;
    for (i = 0; i < 2; i++)
        Pthread_create(&tid,
            NULL,
            thread,
            (void *)i);
    Pthread_exit(NULL);
}
```

sharing.c

Local var: 2 instances (
`myid.p0` [peer thread 0's stack],
`myid.p1` [peer thread 1's stack]
)

```
void *thread(void *vargp)
{
    long myid = (long)vargp;
    static int cnt = 0;

    printf("[%ld]:  %s (cnt=%d)\n",
        myid, ptr[myid], ++cnt);
    return NULL;
}
```

Local static var: 1 instance (`cnt [data]`)

Shared Variable Analysis

- Which variables are shared?

Variable instance	*Referenced by main thread?*	*Referenced by peer thread 0?*	*Referenced by peer thread 1?*
`ptr`	yes	yes	yes
`cnt`	no	yes	yes
`i.m`	yes	no	no
`msgs.m`	yes	yes	yes
`myid.p0`	no	yes	no
`myid.p1`	no	no	yes

- Answer: A variable `x` is shared iff multiple threads reference at least one instance of `x`. Thus:
  - `ptr`, `cnt`, and `msgs` are shared
  - `i` and `myid` are *not* shared

badcnt.c: Improper Synchronization

```c
/* Global shared variable */
volatile long cnt = 0; /* Counter */

int main(int argc, char **argv)
{
    long niters;
    pthread_t tid1, tid2;

    niters = atoi(argv[1]);
    Pthread_create(&tid1, NULL,
        thread, &niters);
    Pthread_create(&tid2, NULL,
        thread, &niters);
    Pthread_join(tid1, NULL);
    Pthread_join(tid2, NULL);

    /* Check result */
    if (cnt != (2 * niters))
        printf("BOOM! cnt=%ld\n", cnt);
    else
        printf("OK cnt=%ld\n", cnt);
    exit(0);
}
```

badcnt.c

```c
/* Thread routine */
void *thread(void *vargp)
{
    long i, niters =
                *((long *)vargp);

    for (i = 0; i < niters; i++)
        cnt++;

    return NULL;
}
```

```
linux> ./badcnt 10000
OK cnt=20000
linux> ./badcnt 10000
BOOM! cnt=13051
linux>
```

cnt should equal 20,000.

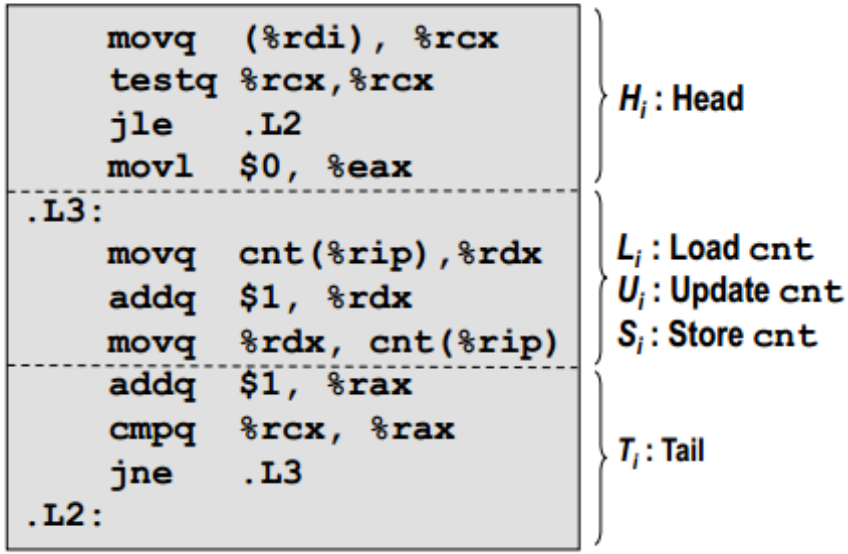
What went wrong?

Assembly Code for Counter Loop

C code for counter loop in thread i

```
for (i = 0; i < niters; i++)
    cnt++;
```

Asm code for thread i

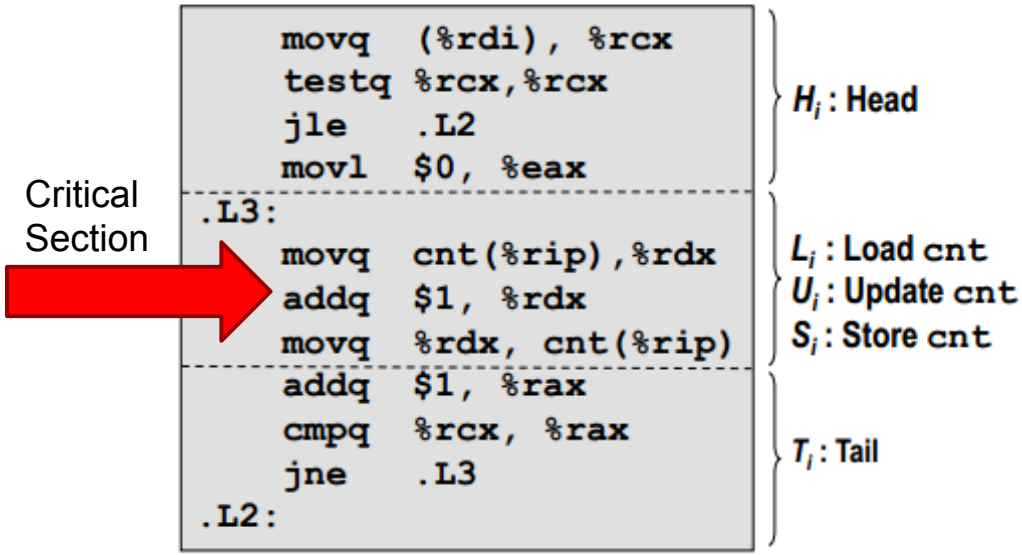


Assembly Code for Counter Loop

C code for counter loop in thread i

```
for (i = 0; i < niters; i++)
    cnt++;
```

Asm code for thread i



Concurrent Execution

- ***Key idea:*** **In general, any sequentially consistent interleaving is possible, but some give an unexpected result!**
  - I_i denotes that thread i executes instruction I
  - $\%rdx_i$ is the content of %rdx in thread i's context

i (thread)	$instr_i$	$\%rdx_1$	$\%rdx_2$	cnt
1	H_1	-	-	0
1	L_1	0	-	0
1	U_1	1	-	0
1	S_1	1	-	1
2	H_2	-	-	1
2	L_2	-	1	1
2	U_2	-	2	1
2	S_2	-	2	2
2	T_2	-	2	2
1	T_1	1	-	2

Thread 1 critical section

Thread 2 critical section

OK

Concurrent Execution (cont)

- Incorrect ordering: two threads increment the counter, but the result is 1 instead of 2

i (thread)	$instr_i$	$\%rdx_1$	$\%rdx_2$	cnt
1	H_1	-	-	0
1	L_1	0	-	0
1	U_1	1	-	0
2	H_2	-	-	0
2	L_2	-	0	0
1	S_1	1	-	1
1	T_1	1	-	1
2	U_2	-	1	1
2	S_2	-	1	1
2	T_2	-	1	1

Oops!

Enforcing Mutual Exclusion

- ***Question:*** **How can we guarantee a safe trajectory?**
- **Answer: We must *synchronize* the execution of the threads so that they never have an unsafe trajectory.**
  - i.e., need to guarantee ***mutually exclusive access*** to critical regions
- **Classic solution:**
  - Semaphores (Edsger Dijkstra)
- **Other approaches (out of our scope)**
  - Mutex and condition variables (Pthreads)
  - Monitors (Java)

Semaphores

- ***Semaphore:*** **non-negative global integer synchronization variable. Manipulated by *P* and *V* operations.**
- **P(s): *Proberen* (to probe)**
  - If *s* is nonzero, then decrement *s* by 1 and return immediately.
  - If *s* is zero, then suspend thread until *s* becomes nonzero and the thread is restarted by a V operation.
  - After restarting, the P operation decrements *s* and returns control to the caller.
- ***V(s): Verhogen* (to increment)**
  - Increment *s* by 1.
  - If there are any threads blocked in a P operation waiting for *s* to become non-zero, then restart exactly one of those threads, which then completes its P operation by decrementing *s*.
- **Semaphore invariant: *(s >= 0)***

C Semaphore Operations

Pthreads functions:

```
#include <semaphore.h>

int sem_init(sem_t *sem, 0, unsigned int val);} /* s = val */

int sem_wait(sem_t *s);  /* P(s) */
int sem_post(sem_t *s);  /* V(s) */
```

CS:APP wrapper functions:

```
#include "csapp.h"

void P(sem_t *s); /* Wrapper function for sem_wait */
void V(sem_t *s); /* Wrapper function for sem_post */
```

Can we improve badcnt.c with semaphores?

goodcnt.c: Proper Synchronization

- Define and initialize a mutex for the shared variable `cnt`:

```
volatile long cnt = 0;   /* Counter */
sem_t mutex;             /* Semaphore that protects cnt */

Sem_init(&mutex, 0, 1); /* mutex = 1 */
```

- Surround critical section with *P* and *V*:

```
for (i = 0; i < niters; i++) {
    P(&mutex);
    cnt++;
    V(&mutex);
 }
```

goodcnt.c

```
linux> ./goodcnt 10000
OK cnt=20000
linux> ./goodcnt 10000
OK cnt=20000
linux>
```

Another worry: Deadlock

- **Def: A process is *deadlocked* iff it is waiting for a condition that will never be true**

- **Typical Scenario**
  - Processes 1 and 2 needs two resources (A and B) to proceed
  - Process 1 acquires A, waits for B
  - Process 2 acquires B, waits for A
  - Both will wait forever!

***Solution: Be sure that transactions involving multiple resources are always locked/unlocked in the same order**

Readers-Writers Problem

- **Generalization of the mutual exclusion problem**

- **Problem statement:**
  - *Reader* threads only read the object
  - *Writer* threads modify the object
  - Writers must have exclusive access to the object
  - Unlimited number of readers can access the object

- **Occurs frequently in real systems, e.g.,**
  - Online airline reservation system
  - Multithreaded caching Web proxy

Variants of Readers-Writers

- ***First readers-writers problem* (favors readers)**
  - No reader should be kept waiting unless a writer has already been granted permission to use the object
  - A reader that arrives after a waiting writer gets priority over the writer
- ***Second readers-writers problem* (favors writers)**
  - Once a writer is ready to write, it performs its write as soon as possible
  - A reader that arrives after a writer must wait, even if the writer is also waiting
- ***Starvation* (where a thread waits indefinitely) is possible in both cases**

Solution to First Readers-Writers Problem

Readers:

```
int readcnt;     /* Initially 0 */
sem_t mutex, w; /* Both initially 1 */

void reader(void)
{
  while (1) {
    P(&mutex);
    readcnt++;
    if (readcnt == 1) /* First in */
      P(&w);
    V(&mutex);

    /* Reading happens here */

    P(&mutex);
    readcnt--;
    if (readcnt == 0) /* Last out */
      V(&w);
    V(&mutex);
  }
}
```

Writers:

```
void writer(void)
{
  while (1) {
    P(&w);

    /* Writing here */

    V(&w);
  }
}
```

rw1.c

Crucial concept: Thread Safety

- **Functions called from a thread must be *thread-safe***
- ***Def:* A function is *thread-safe* iff it will always produce correct results when called repeatedly from multiple concurrent threads**
- **Classes of thread-unsafe functions:**
  - Class 1: Functions that do not protect shared variables
  - Class 2: Functions that keep state across multiple invocations
  - Class 3: Functions that return a pointer to a static variable
  - Class 4: Functions that call thread-unsafe functions

Thread-Unsafe Functions (Class 1)

- **Failing to protect shared variables**
  - Fix: Use *P* and *V* semaphore operations
  - Example: `goodcnt.c`
  - Issue: Synchronization operations will slow down code

Thread-Unsafe Functions (Class 2)

- **Relying on persistent state across multiple function invocations**
  - Example: Random number generator that relies on static state

```
static unsigned int next = 1;

/* rand: return pseudo-random integer on 0..32767 */
int rand(void)
{
    next = next*1103515245 + 12345;
    return (unsigned int)(next/65536) % 32768;
}

/* srand: set seed for rand() */
void srand(unsigned int seed)
{
    next = seed;
}
```

Thread-Safe Random Number Generator

- **Pass state as part of argument**
  - and, thereby, eliminate static state

```
/* rand_r - return pseudo-random integer on 0..32767 */

int rand_r(int *nextp)
{
    *nextp = *nextp*1103515245 + 12345;
    return (unsigned int)(*nextp/65536) % 32768;
}
```

- **Consequence: programmer using `rand_r` must maintain seed**

Thread-Unsafe Functions (Class 3)

- **Returning a pointer to a static variable**
- **Fix 1. Rewrite function so caller passes address of variable to store result**
  - Requires changes in caller and callee
- **Fix 2. Lock-and-copy**
  - Requires simple changes in caller (and none in callee)
  - However, caller must free memory.

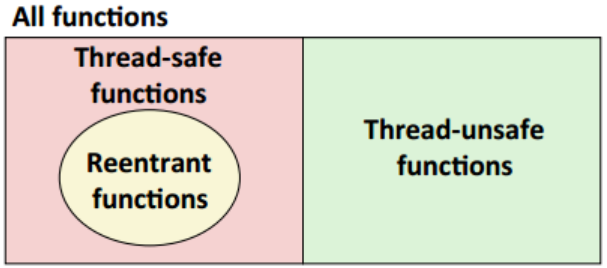
```
/* lock-and-copy version */
char *ctime_ts(const time_t *timep,
               char *privatep)
{
    char *sharedp;

    P(&mutex);
    sharedp = ctime(timep);
    strcpy(privatep, sharedp);
    V(&mutex);
    return privatep;
}
```

Thread-Unsafe Functions (Class 4)

- **Calling thread-unsafe functions**
  - Calling one thread-unsafe function makes the entire function that calls it thread-unsafe
  - Fix: Modify the function so it calls only thread-safe functions ☺

Reentrant Functions

- A function is reentrant if it accesses no shared variables when called by multiple threads.
- The function can be interrupted and called again in the middle of its execution, and it will assume correct operation when the original function is “*re-entered”.*



Final Review

Floating Point Review

- 0 0000 000

  s e f
- Normalized (e is not all 0's or all 1's):
  - bias = $2^{sizeof(e) - 1} - 1$
  - E = e - bias
  - V = $(-1)^s * 2^E * (1.f)_2$
- Denormalized (e is all 0's):
  - bias = $2^{sizeof(e) - 1} - 1$
  - E = 1 - bias
  - V = $(-1)^s * 2^E * (.f)_2$

Floating Point Review

- 0 0000 000

  s e f
- Infinity (e is all 1's and f is all 0's):
- NaN (e is all 1's and f is not all 0's):

Floating Point Review

Consider the following 9-bit floating point representations based on the IEEE floating-point format:

0 000 00000
s e f

Floating Point Review

- What's the smallest possible range between any two representable values?

Floating Point Review

- What's the smallest possible range between any two representable values?
- Smallest difference caused by least significant bit
- 0 000 00000 to 0 000 00001
- bias = 3
- E = 1 – bias = -2
- 0 to $2^{-2}*2^{-5}$
- 0 to 2^{-7}

Floating Point Review

- What is the range of numbers where the difference between any two representable numbers is > 2^{-5}

Floating Point Review

- What is the range of numbers where the difference between any two representable numbers is > 2^{-5}
- The least significant bit will always represent the minimum distance between numbers with the same E.
- $2^E * (1 + 1/2^5) – 2^E * 1 >= 2^{-5}$
- $2^E / 2^5 >= 2^{-5}$
- E >=0
- e – bias >= 0
- E >= 3
- |X| >= 0 011 00000 <===> |X| >= 1

Floating Point Review

- 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 Review

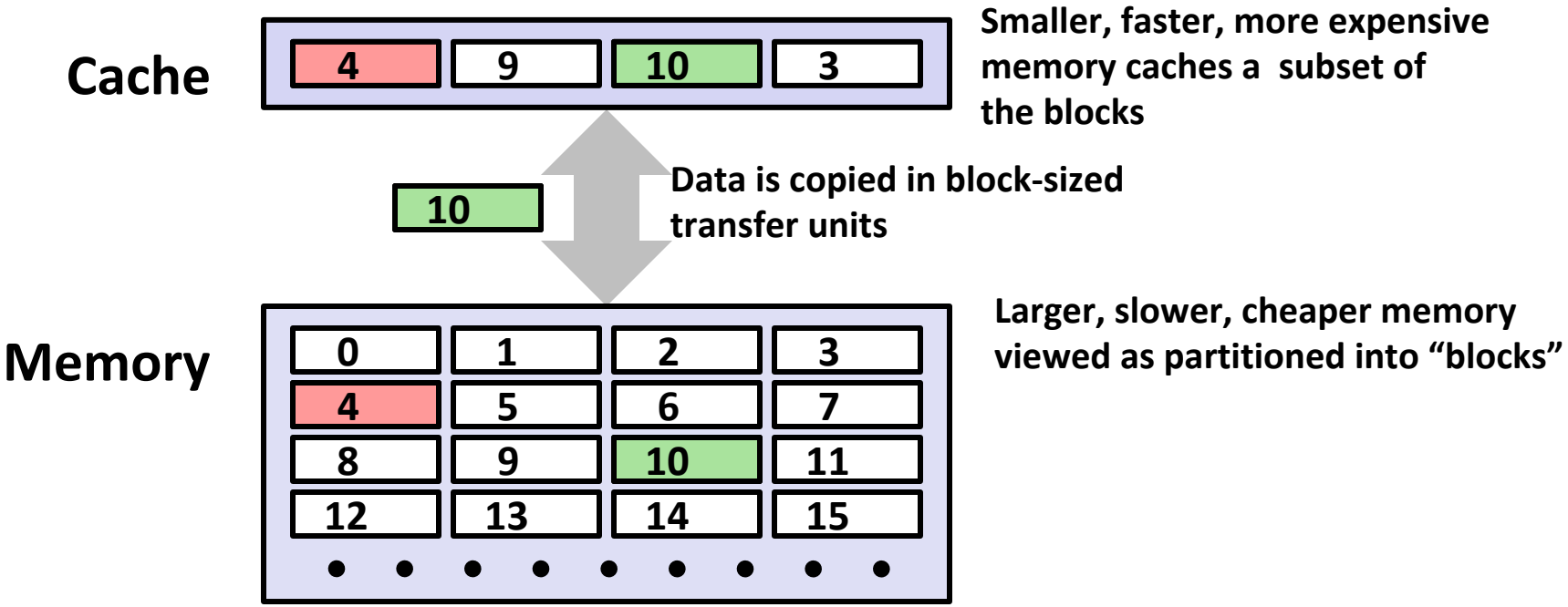
- Format A:
  - 0 000 10101
  - Bias = 3
  - E = 1 – Bias = -2 (Denormalized)
  - $V = 2^{-2} * (1/2 + 1/2^3 + 1/2^5) = 2^{-3} * (1 + 1/2^2 + 1/2^4) = .1640625$
- Format B:
  - Format B has enough range in the exponent bits to represent the full exponent range of format A.
  - Bias = 15
  - E = -3 = e – Bias
  - e = 12
  - We want this V to equal the V from format A ($2^{-3} * (1 + 1/2^2 + 1/2^4)$)

Floating Point Review

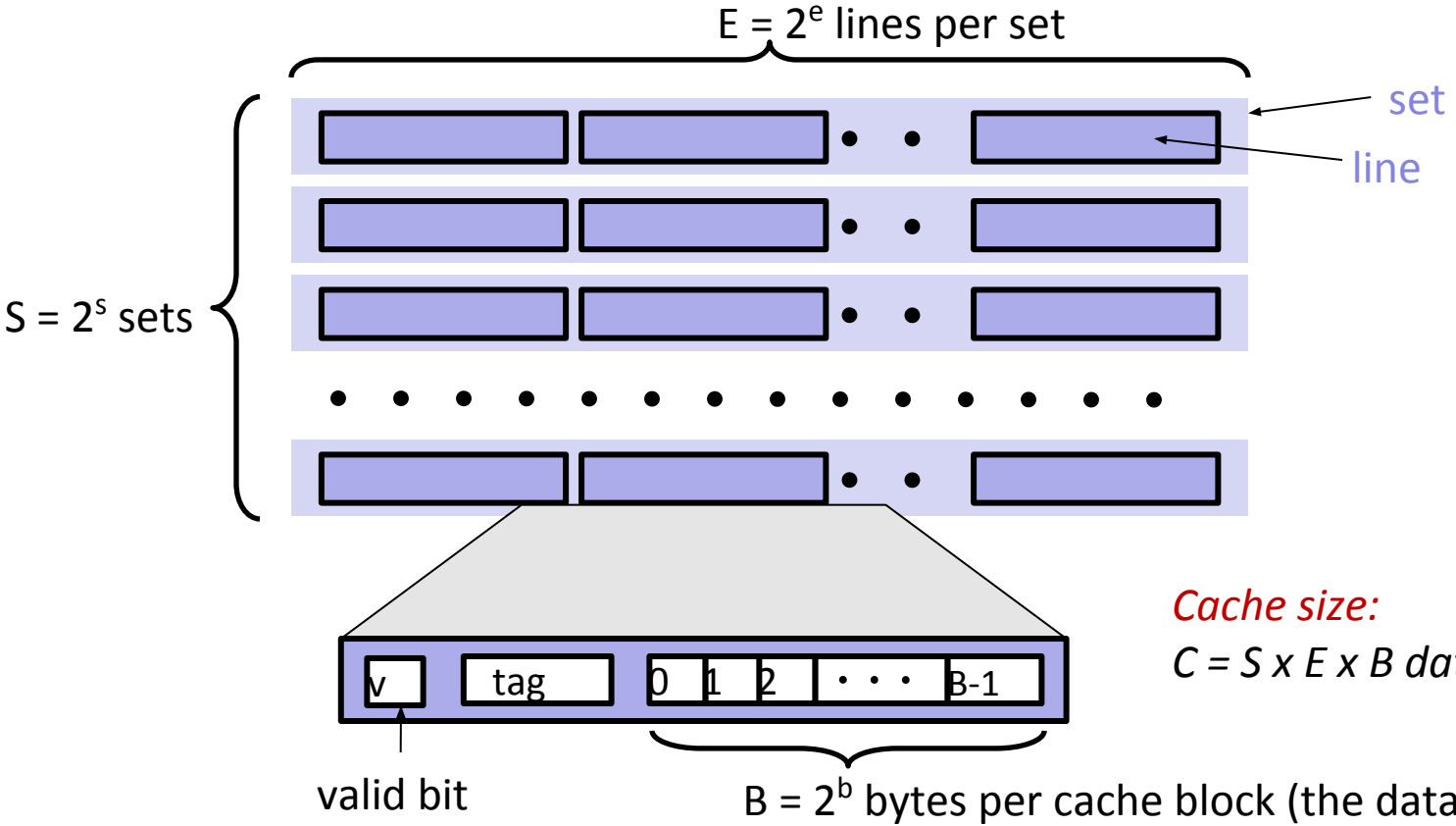
- However, it is impossible to represent this in format B
  - $V = 2^{-3} * (1 + 1/2^2 + 1/2^4)$
- Format B only has three fractional bits and cannot represent a fractional contribution of $1/2^4$
- Rounding down solution:
  - $V = 2^{-3} * (1 + 1/2^2) = .15625$
  - 0 01100 010
- Rounding up solution:
  - $V = 2^{-3} * (1 + 1/2^2 + 1/2^3) = .171875$
  - 0 01100 011

Cache

General Cache Concept



General Cache Organization (S, E, B)

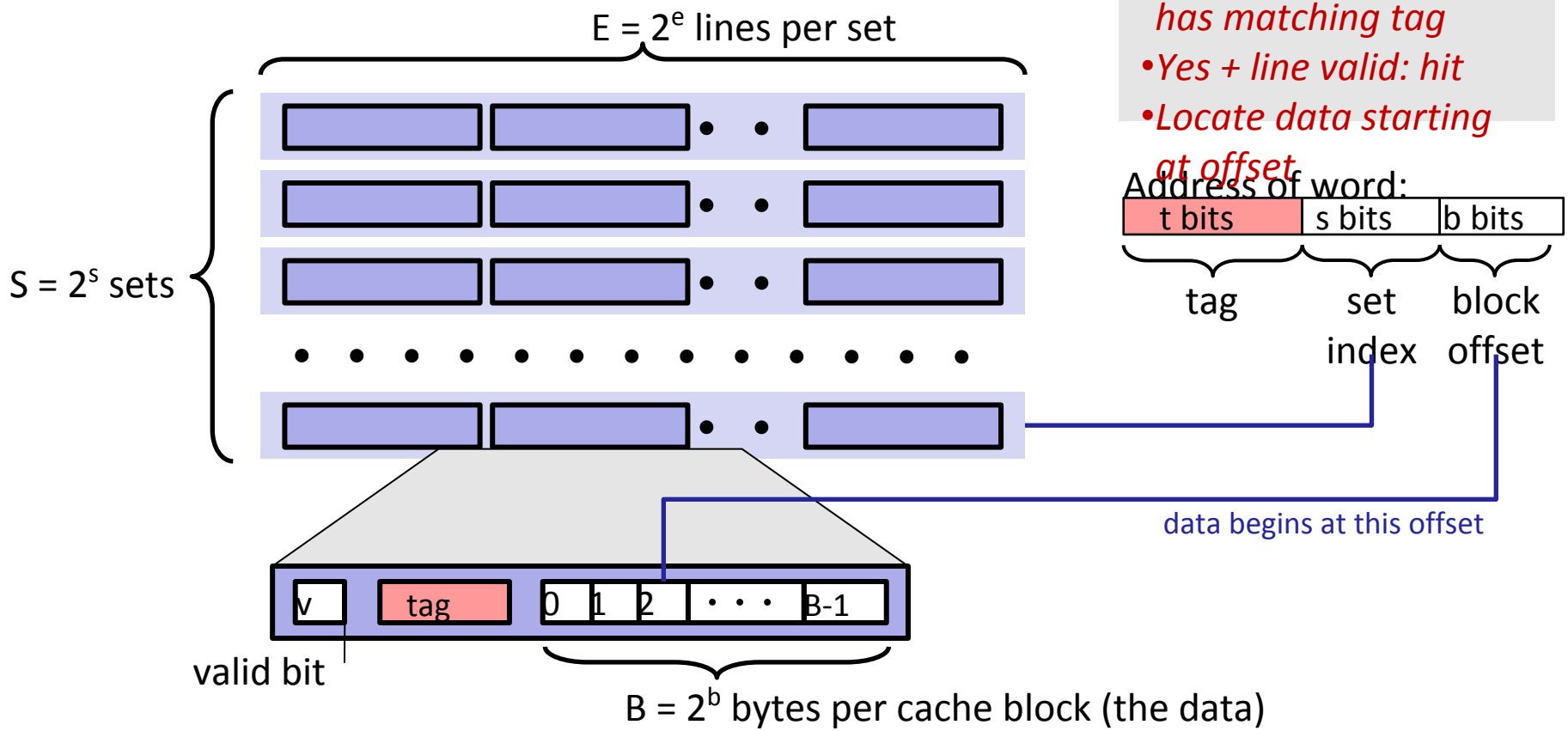


Cache size:

C = S x E x B data bytes

Cache Read

- *Locate set*
- *Check if any line in set has matching tag*
- *Yes + line valid: hit*
- *Locate data starting at offset*



What about writes?

- **Multiple copies of data exist:**
  - L1, L2, L3, Main Memory, Disk
- **What to do on a write-hit?**
  - Write-through (write immediately to memory)
  - Write-back (defer write to memory until replacement of line)
    - Need a dirty bit (line different from memory or not)
- **What to do on a write-miss?**
  - Write-allocate (load into cache, update line in cache)
    - Good if more writes to the location follow
  - No-write-allocate (writes straight to memory, does not load into cache)
- **Typical**
  - Write-through + No-write-allocate
  - **Write-back + Write-allocate**

Cache memories. Consider the following matrix transpose function

```
typedef int array[2][2];

void transpose(array dst, array src) {
  int i, j;

  for (j = 0; j < 2; j++) {
    for (i = 0; i < 2; i++) {
      dst[i][j] = src[j][i];
    }
  }
}
```

running on a hypothetical machine with the following properties:

- `sizeof(int) == 4`.
- The `src` array starts at address 0 and the `dst` array starts at address 16 (decimal).
- There is a single L1 data cache that is direct mapped and write-allocate, with a block size of 8 bytes.
- Accesses to the `src` and `dst` arrays are the only sources of read and write accesses to the cache, respectively.

Suppose the cache has a total size of 16 data bytes (i.e., the block size times the number of sets is 16 bytes) and that the cache is initially empty. Then for each `row` and `col`, indicate whether each access to `src[row][col]` and `dst[row][col]` is a hit (h) or a miss (m). For example, reading `src[0][0]` is a miss and writing `dst[0][0]` is also a miss.

`src` array		
	col 0	col 1
row 0	m	
row 1		

`dst` array		
	col 0	col 1
row 0	m	
row 1		

B. Repeat part A for a cache with a total size of 32 data bytes.

`src` array		
	col 0	col 1
row 0	m	
row 1		

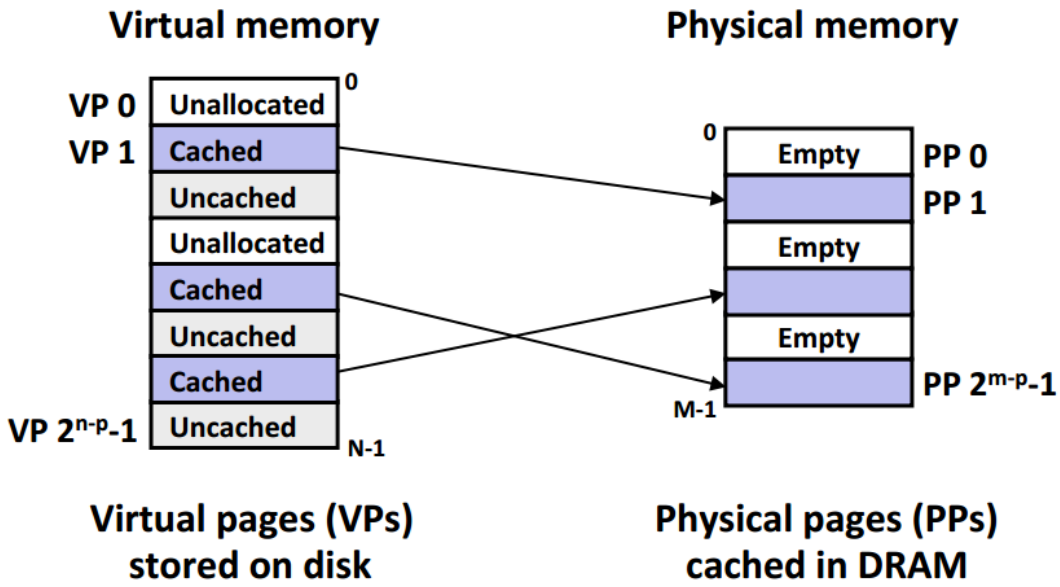
`dst` array		
	col 0	col 1
row 0	m	
row 1		

src		dst	
M	M	M	M
M	H	M	M

src		dst	
M	H	M	H
M	H	M	H

Virtual Memory

Remember how VM works



Some basics

Virtual Address Space is 8 GB = 8 * 2^30 B.

How many bits do I need in my virtual address?

_____

Some basics

Virtual Address Space is 8 GB = 8 * 2^30 B.

How many bits do I need in my virtual address?

33

Some basics

Virtual Address Space is 8 GB = 8 * 2^30 B.

How many bits do I need in my virtual address?

33

If I want pages of size 4KB = 4 * 2^10 B, then

How many bits do I need for my Virtual Page Offset?

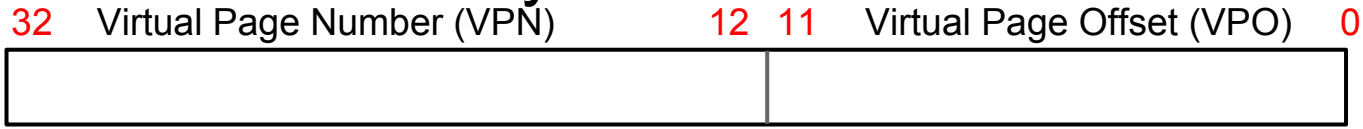
How many bits do I need for my Virtual Page Number?

Some basics

Virtual Address Space is 8 GB = 8 * 2^30 B.

How many bits do I need in my virtual address?

33



If I want pages of size 4KB = 4 * 2^10 B, then

How many bits do I need for my Virtual Page Offset?

12

How many bits do I need for my Virtual Page Number?

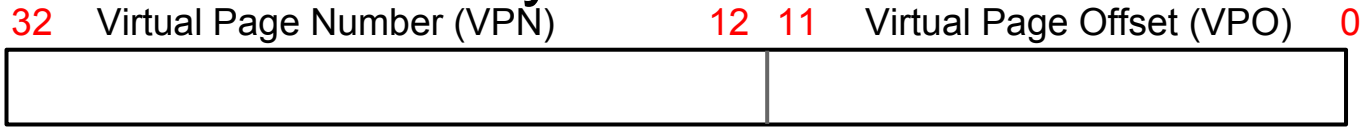
21

Some basics

Virtual Address Space is 8 GB = 8 * 2^30 B.

How many bits do I need in my virtual address?

33



If I want pages of size 4KB = 4 * 2^10 B, then

How many bits do I need for my Virtual Page Offset?

12

How many bits do I need for my Virtual Page Number?

21 Then how many pages are there? 2^21

VM Address Translation

Need to use a **page table** to translate a virtual memory address into a physical memory 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 = ?
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 = ?

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 = ?
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 = ?

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
	...

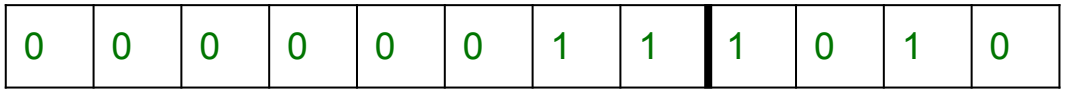
bits VPO = 4
bits PPO = 4

bits VPN = 8
bits PPN = 6

Physical Address:

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

Virtual Address: **0x03A**



Page Table:

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

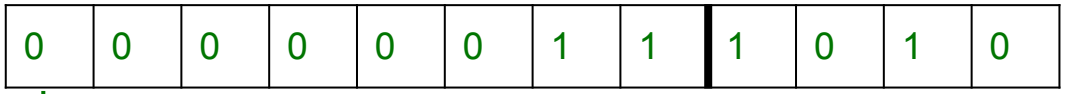
bits VPO = 4
bits PPO = 4

bits VPN = 8
bits PPN = 6

Physical Address:

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

Virtual Address: **0x03A**



Page Table:

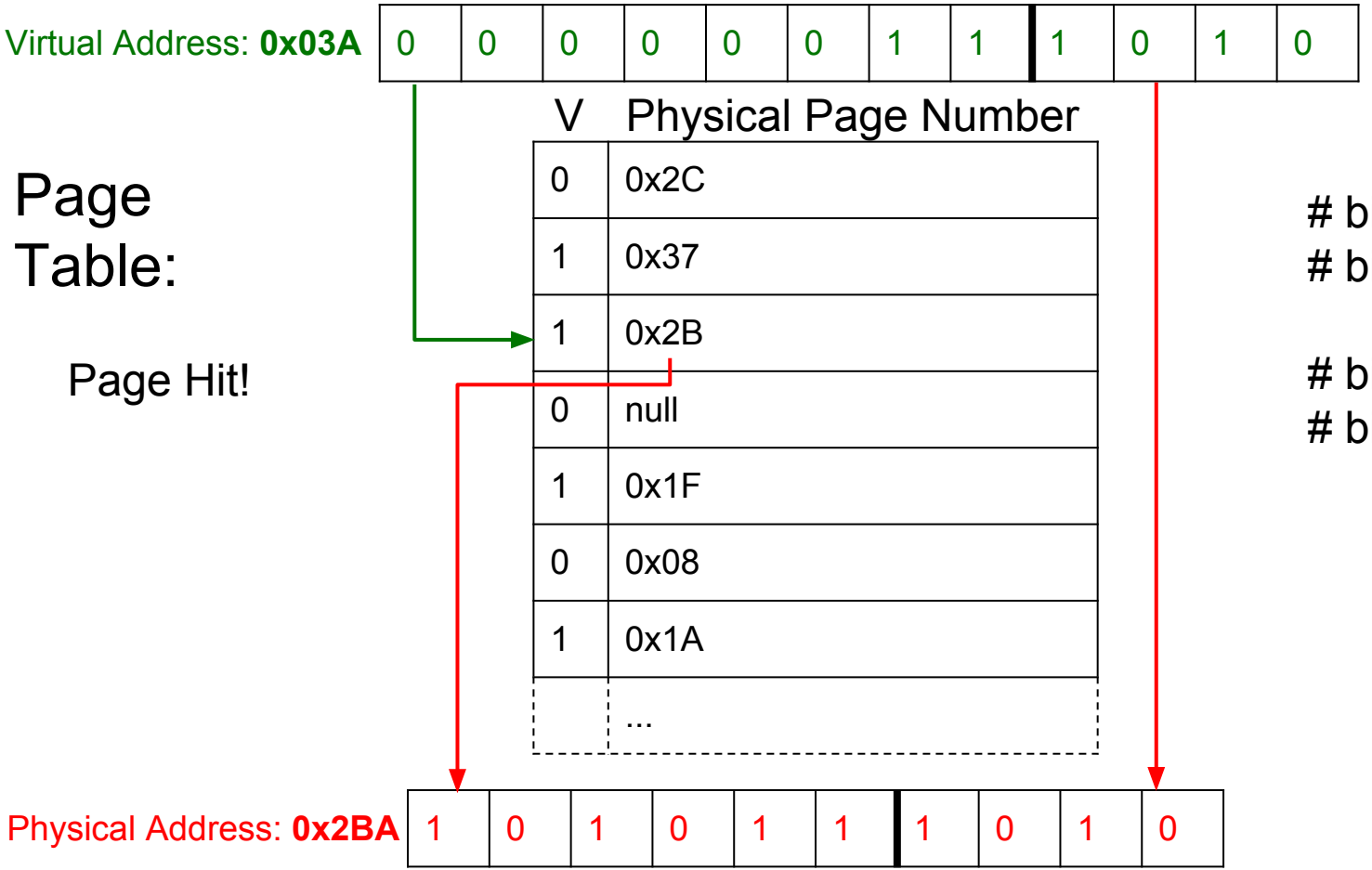
V	Physical Page Number
0	0x2C
1	0x37
1	0x2B
0	null
1	0x1F
0	0x08
1	0x1A
	...

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.



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**

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: **0x047**

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

Page Table:

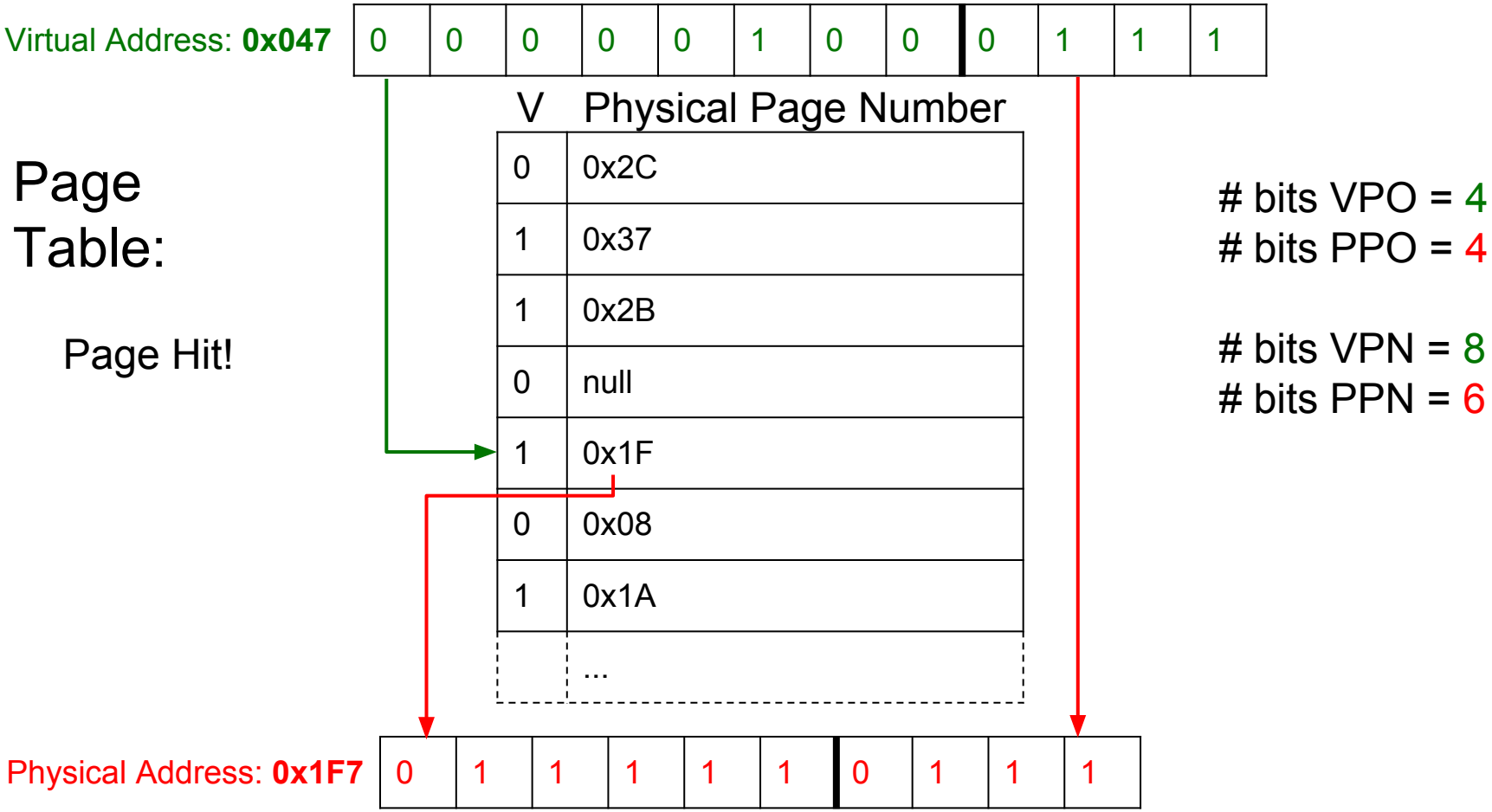
V	Physical Page Number
0	0x2C
1	0x37
1	0x2B
0	null
1	0x1F
0	0x08
1	0x1A
	...

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.



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

Virtual Address: **0x05F**

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
	...

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
0	0x08
1	0x1A
	...

bits VPO = 4
bits PPO = 4

bits VPN = 8
bits PPN = 6

Page Fault!
=(
Page is not resident in Memory

What do we do?

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

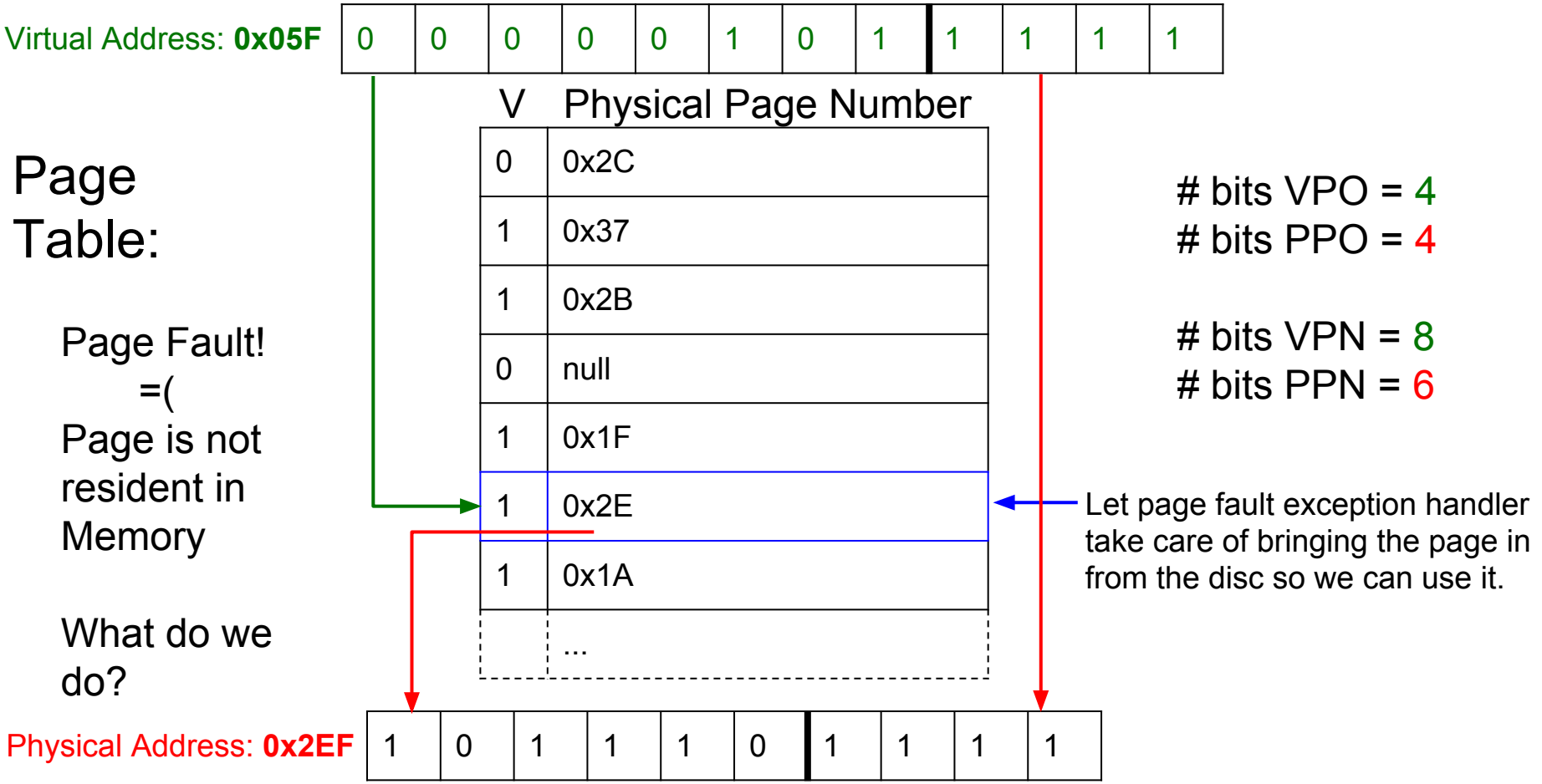
Page Fault!
=(
Page is not resident in Memory

What do we do?

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

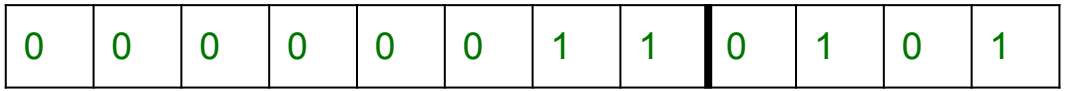
Physical Address:

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



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

Virtual Address: **0x035**



Page Table:

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

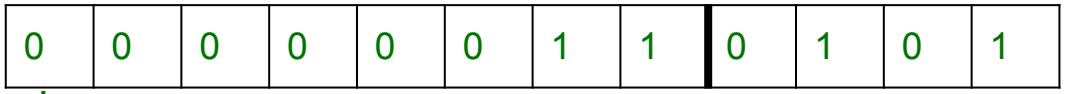
bits VPO = 4
bits PPO = 4

bits VPN = 8
bits PPN = 6

Physical Address:

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

Virtual Address: **0x035**



Page Table:

null ???

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: **0x035**

0	0	0	0	0	0	1	1	0	1	0	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 in Memory OR on disc.
It's unallocated

What do we do?

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: **0x035**

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

Page Table:

V	Physical Page Number
0	0x2C
1	0x37
1	0x2B
1	0x14
1	0x1F
0	0x08
1	0x1A
	...

Page Fault!
=(
Page is not in Memory OR on disc.
It's unallocated

What do we do?

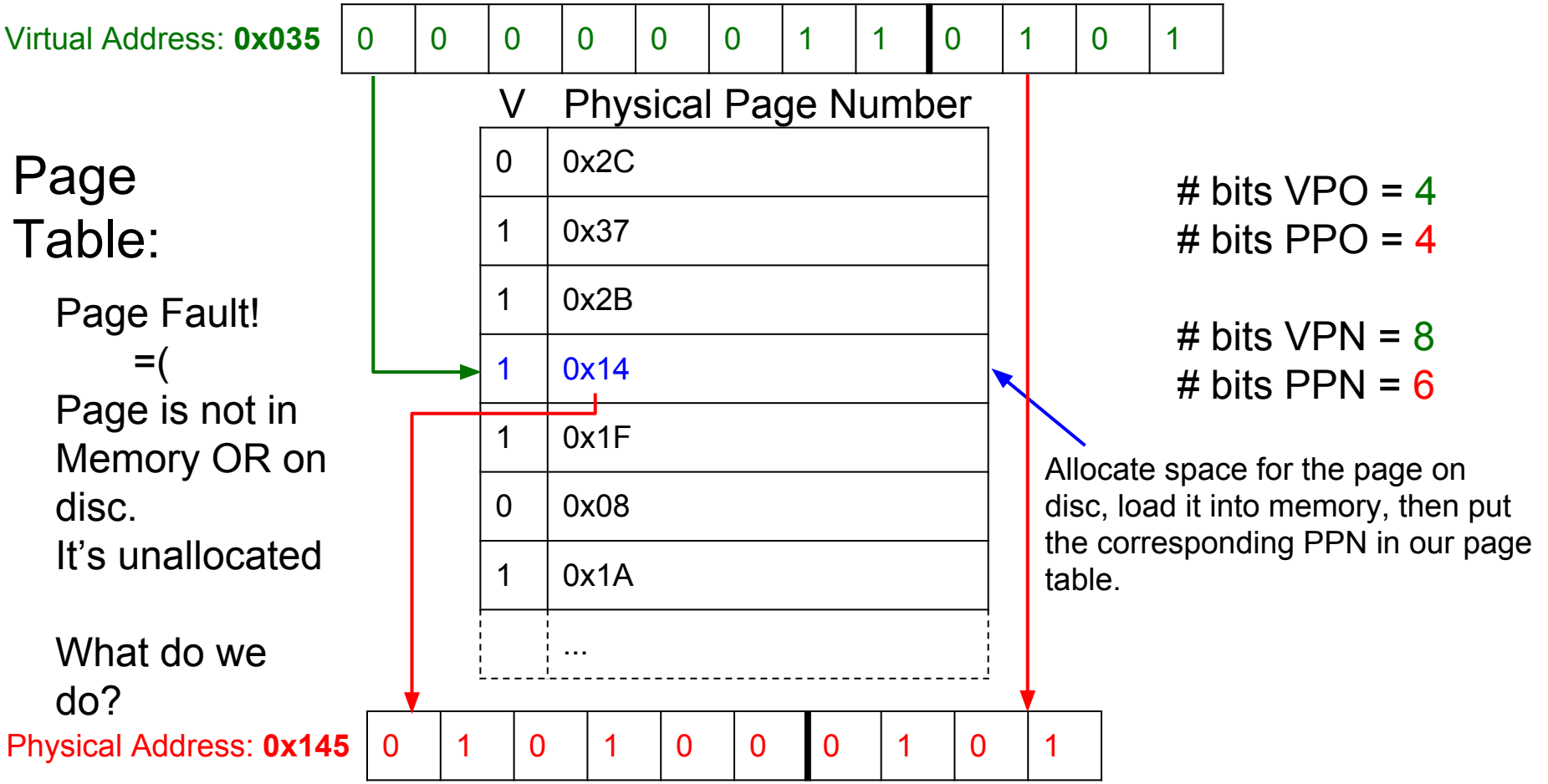
Allocate space for the page on disc, load it into memory, then put the corresponding PPN in our page table.

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 Fault!
=(
Page is not in Memory OR on disc.
It's unallocated

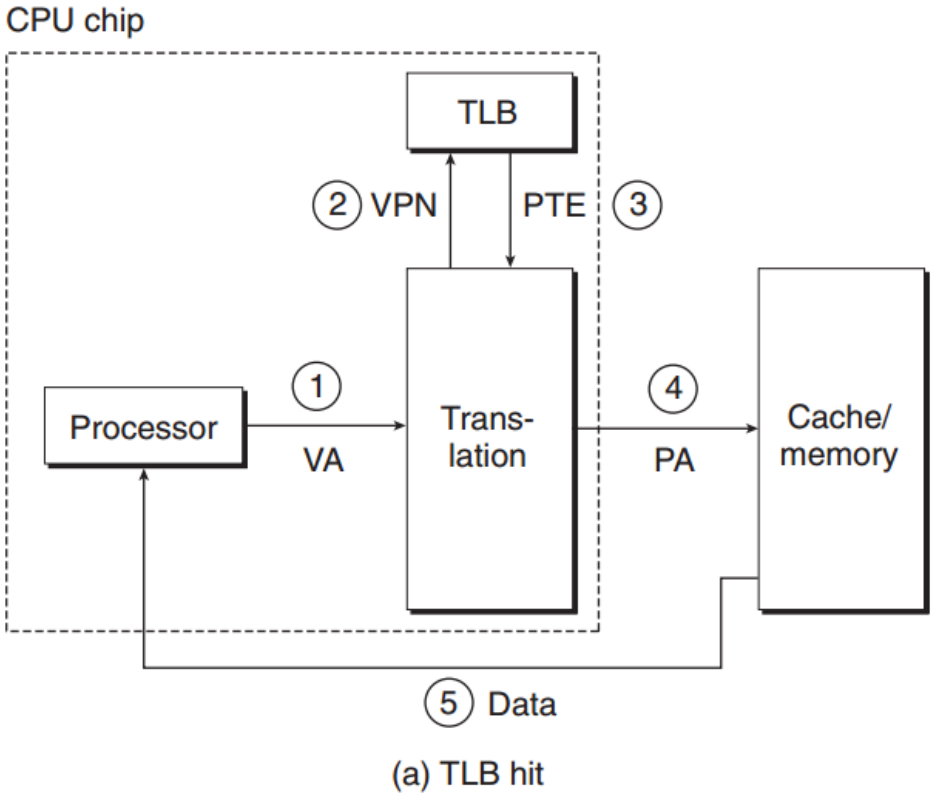
What do we do?

bits VPO = 4
bits PPO = 4

bits VPN = 8
bits PPN = 6

Allocate space for the page on disc, load it into memory, then put the corresponding PPN in our page table.

Translation Lookaside Buffer (TLB)

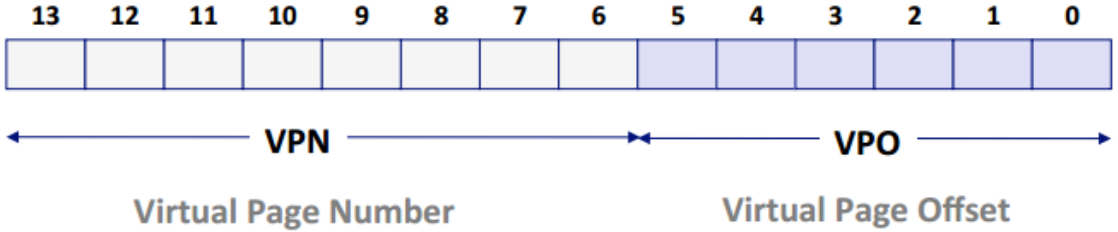


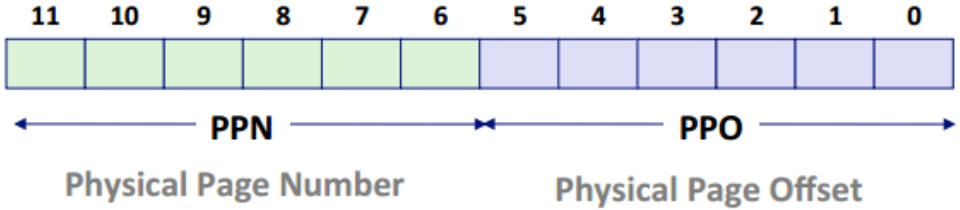
(a) TLB hit

1. CPU sends Virtual Address to MMU
2. Send Virtual Page Number to TLB
3. If TLB hit, send Page Table Entry back to MMU.
4. Send Physical Address to Main Memory/hierarchy of caches.
5. Memory returns data to CPU

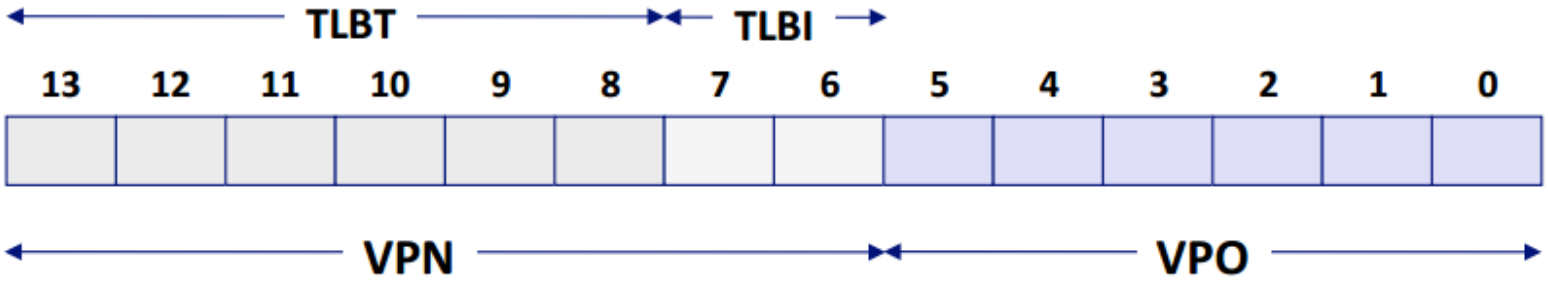
Simple Memory System Example

- **Addressing**
  - 14-bit virtual addresses
  - 12-bit physical address
  - Page size = 64 bytes





4-way set associative TLB



Set	Tag	PPN	Valid	Tag	PPN	Valid	Tag	PPN	Valid	Tag	PPN	Valid
0	03	–	0	09	0D	1	00	–	0	07	02	1
1	03	2D	1	02	–	0	04	–	0	0A	–	0
2	02	–	0	08	–	0	06	–	0	03	–	0
3	07	–	0	03	0D	1	0A	34	1	02	–	0

Address Translation

VA = 0x03D4 PA = ?

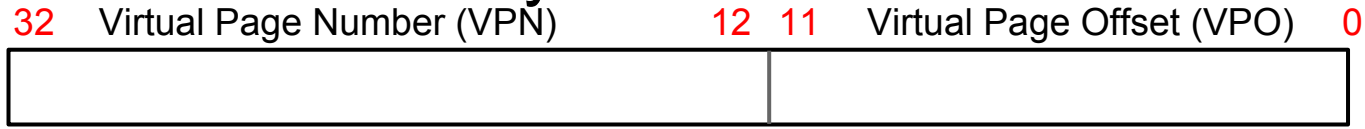
VA = 0x0020 PA = ?

Back to basics.

Virtual Address Space is 8 GB = 8 * 2^30 B.

How many bits do I need in my virtual address?

33



If I want pages of size 4KB = 4 * 2^10 B, then

How many bits do I need for my Virtual Page Offset?

12

How many bits do I need for my Virtual Page Number?

21 Then how many pages are there? 2^21

Question

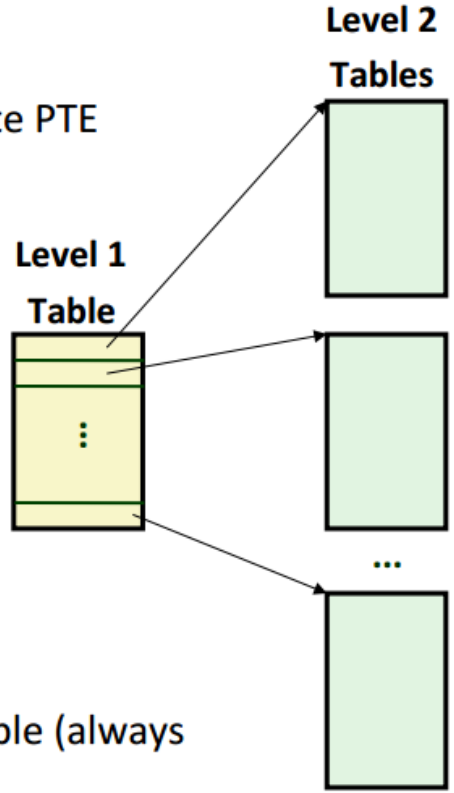
If there are 2^21 VPs, then don’t I have to store 2^21 entries in my page table??

That would take up 2MB of space!

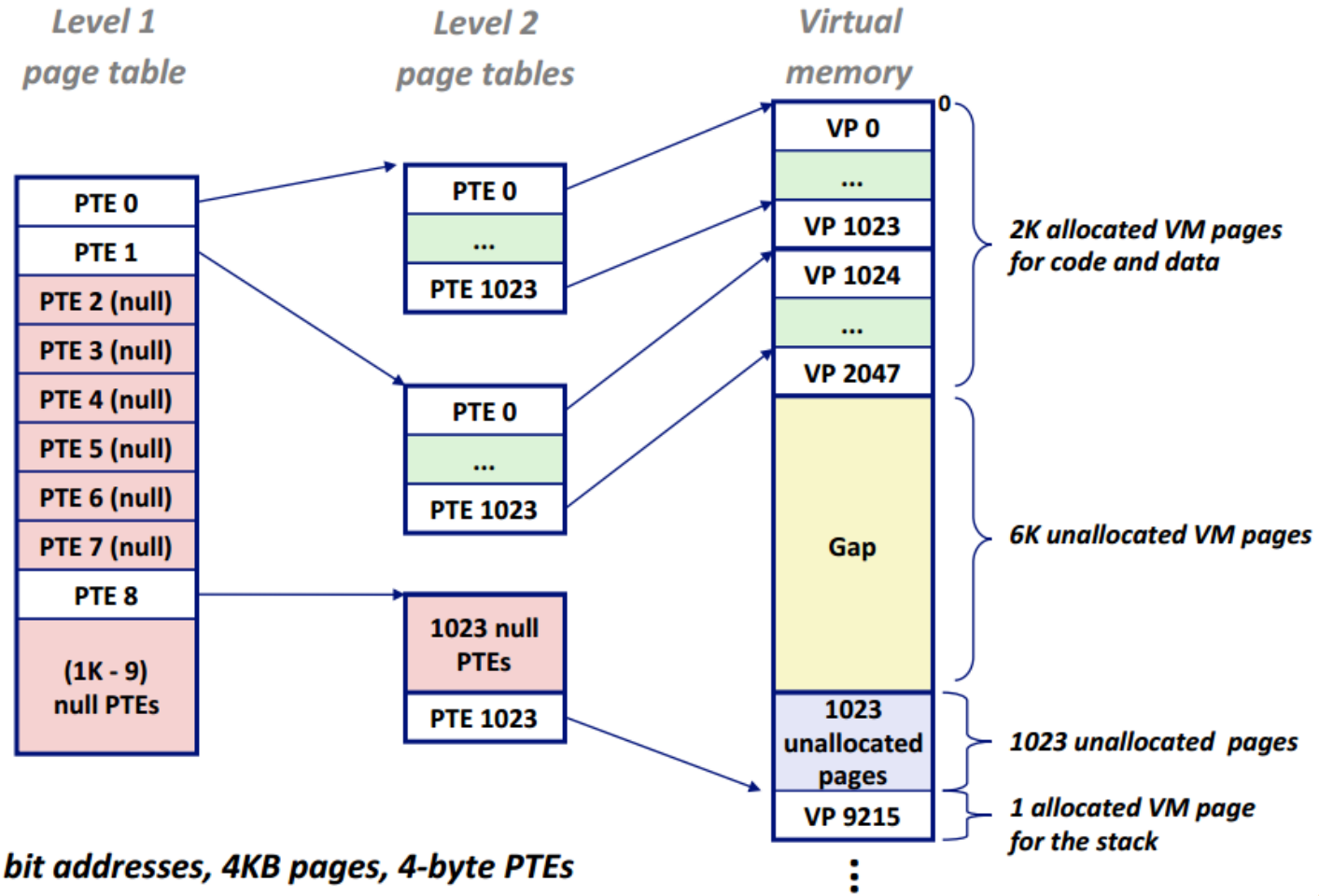
Solution: Multi-leveled paging

Multi-Level Page Tables

- **Suppose:**
  - 4KB (2^{12}) page size, 64-bit address space, 8-byte PTE
- **Problem:**
  - Would need a 32,000 TB page table!
    - $2^{64} * 2^{-12} * 2^{3} = 2^{55}$ bytes
- **Common solution:**
  - Multi-level page tables
  - Example: 2-level page table
    - Level 1 table: each PTE points to a page table (always memory resident)
    - Level 2 table: each PTE points to a page (paged in and out like any other data)

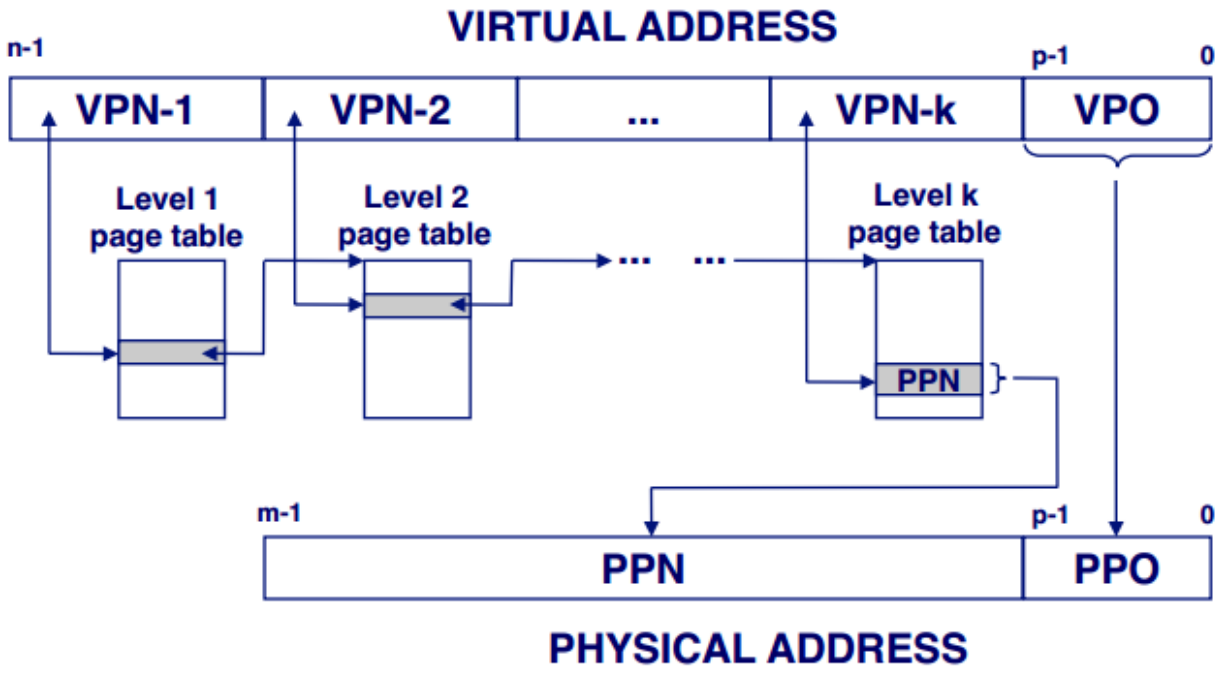


A Two-Level Page Table Hierarchy



32 bit addresses, 4KB pages, 4-byte PTEs

Translating with a k-level Page Table



Questions on Virtual Memory?

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

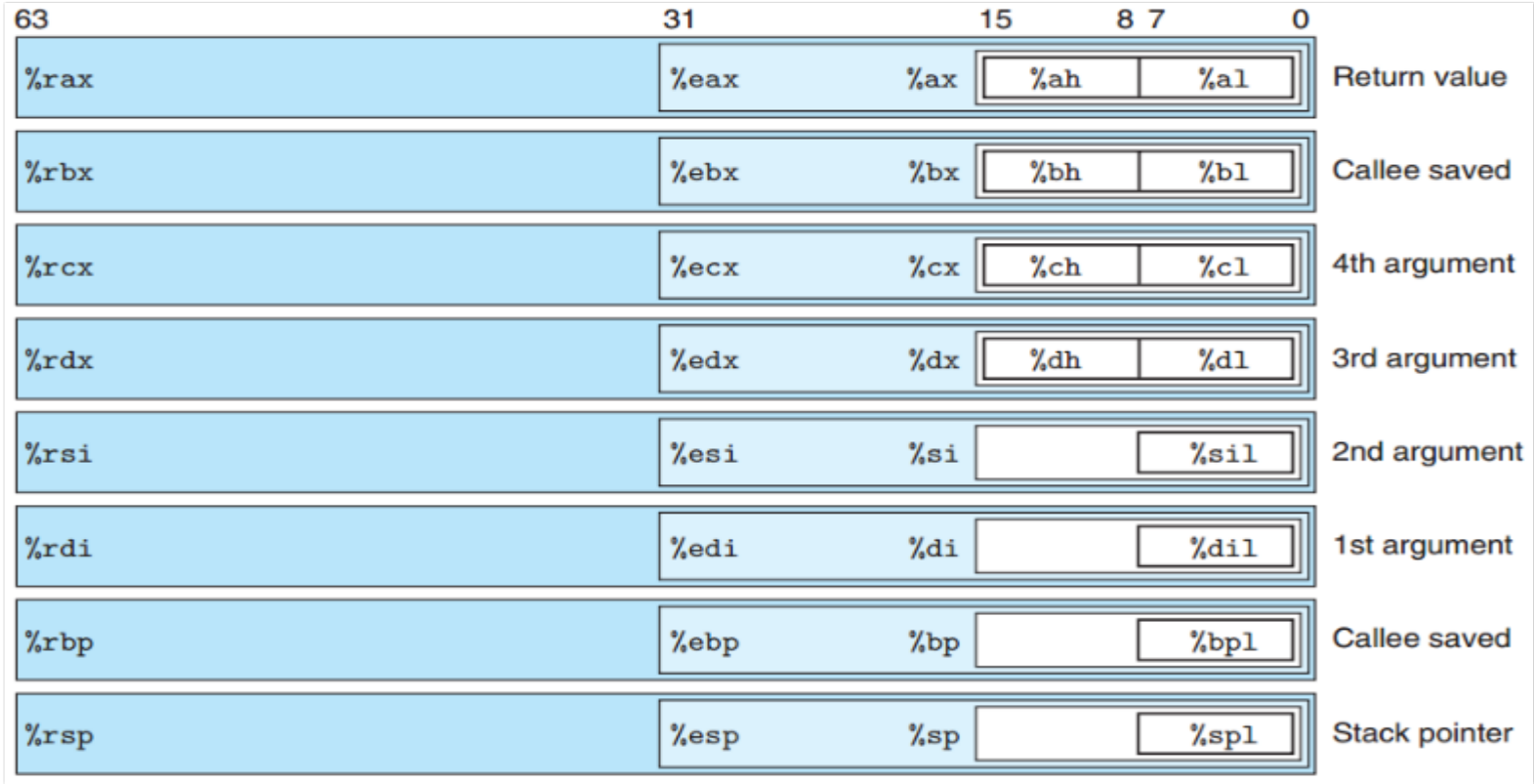
  sar_ %al, %edi

  sub_ 0x63(%eax), %esi

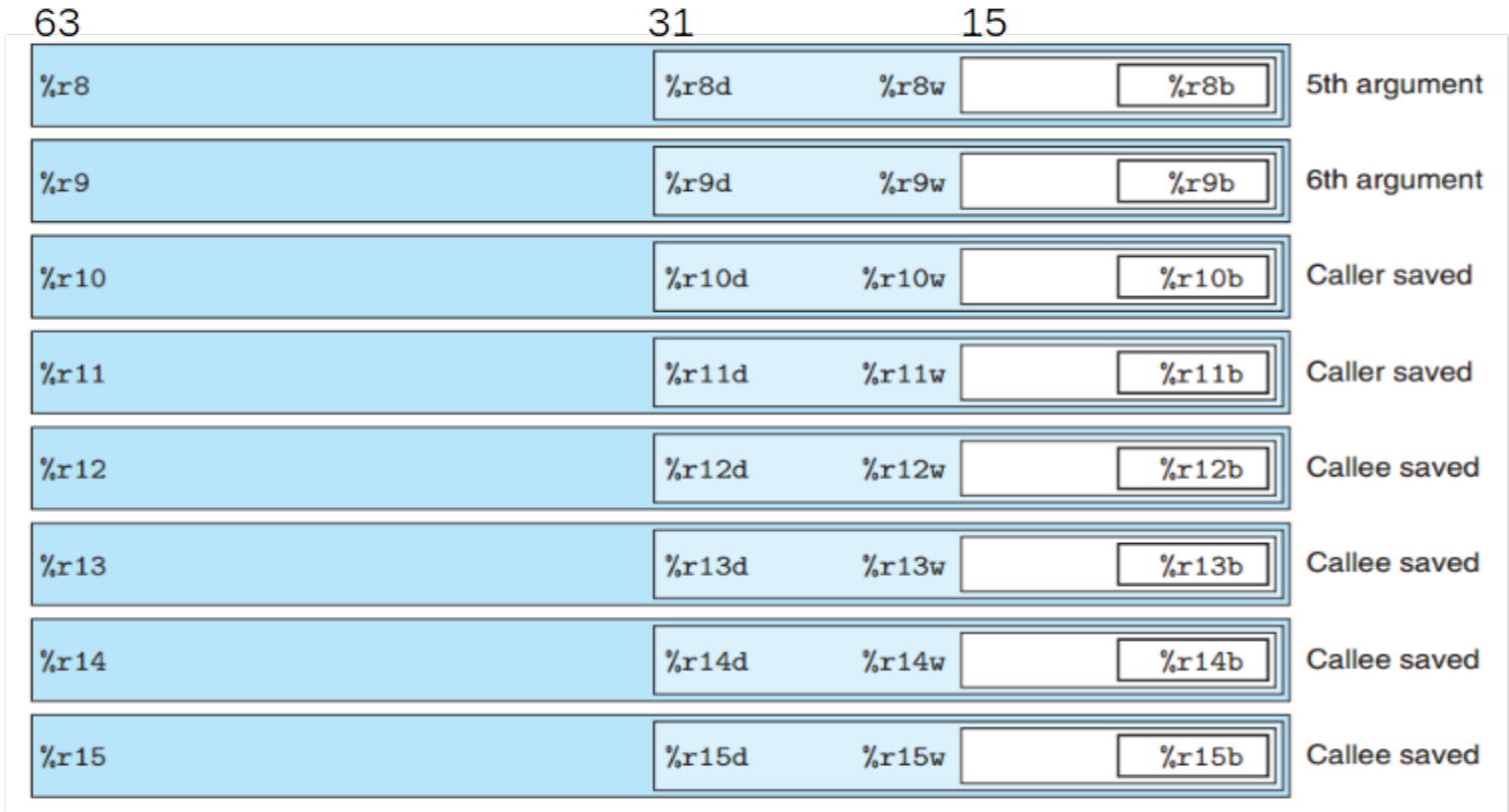
- When destination is memory, it can be ambiguous

  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
```

```
1 8048420 <switch_prob>:
2 8048420: 55                    push %ebp
3 8048421: 89 e5                 mov %esp,%ebp
4 8048423: 8b 45 08              mov 0x8(%ebp),%eax
5 8048426: 8b 55 0c              mov 0xc(%ebp),%edx
6 8048429: 83 ea 32              sub $0x32,%edx
7 804842c: 83 fa 05              cmp $0x5,%edx
8 804842f: 77 17                 ja 8048448 <switch_prob+0x28>
9 8048431: ff 24 95 d0 85 04 08  jmp *0x80485d0(,%edx,4)
10 8048438: c1 e0 02             shl $0x2,%eax
11 804843b: eb 0e                jmp 804844b <switch_prob+0x2b>
12 804843d: c1 f8 02             sar $0x2,%eax
13 8048440: eb 09                jmp 804844b <switch_prob+0x2b>
14 8048442: 8d 04 40             lea (%eax,%eax,2),%eax
15 8048445: 0f af c0             imul %eax,%eax
16 8048448: 83 c0 0a             add $0xa,%eax
17 804844b: 5d                   pop %ebp
18 804844c: c3                    ret
```

```
                1 8048420 <switch_prob>:
                2 8048420: 55                    push %ebp
                3 8048421: 89 e5                 mov %esp,%ebp
                4 8048423: 8b 45 08              mov 0x8(%ebp),%eax
                5 8048426: 8b 55 0c              mov 0xc(%ebp),%edx
                6 8048429: 83 ea 32              sub $0x32,%edx
                7 804842c: 83 fa 05              cmp $0x5,%edx
                8 804842f: 77 17                 ja 8048448 <switch_prob+0x28>
                9 8048431: ff 24 95 d0 85 04 08  jmp *0x80485d0(,%edx,4)
JTAB[0] JTAB[2] 10 8048438: c1 e0 02             shl $0x2,%eax
               11 804843b: eb 0e                 jmp 804844b <switch_prob+0x2b>
     JTAB[3]   12 804843d: c1 f8 02              sar $0x2,%eax
               13 8048440: eb 09                 jmp 804844b <switch_prob+0x2b>
     JTAB[4]   14 8048442: 8d 04 40              lea (%eax,%eax,2),%eax
               15 8048445: 0f af c0              imul %eax,%eax
     JTAB[1]   16 8048448: 83 c0 0a              add $0xa,%eax
               17 804844b: 5d                    pop %ebp
               18 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

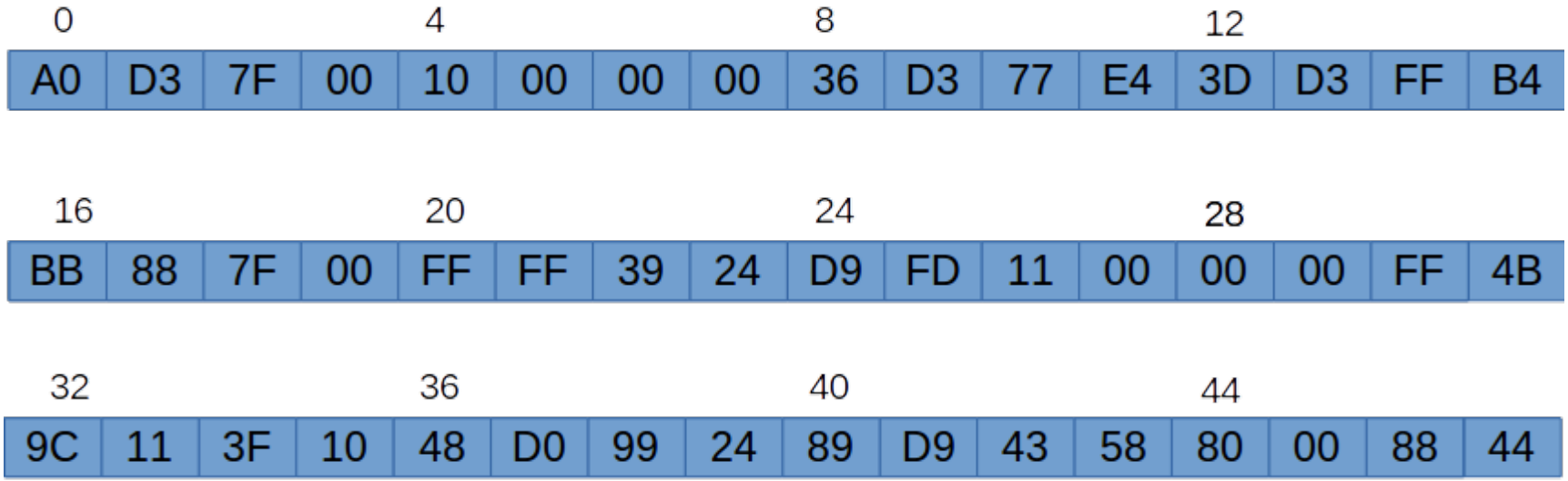
```
typedef struct ELE *tree_ptr;
struct ELE {
    long val;
    tree_ptr left;
    tree_ptr right;
}
long trace(tree_ptr tp);     // What does this do?
```

```
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

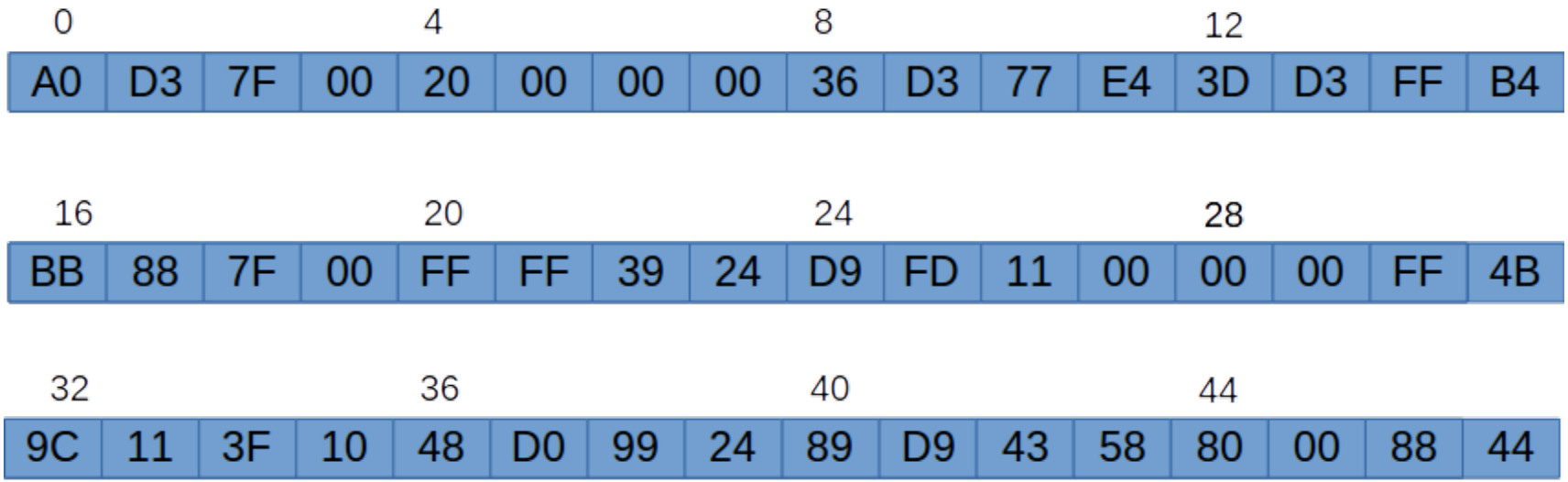
Suppose ebp = 4, what is return address?



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

Stack Discipline

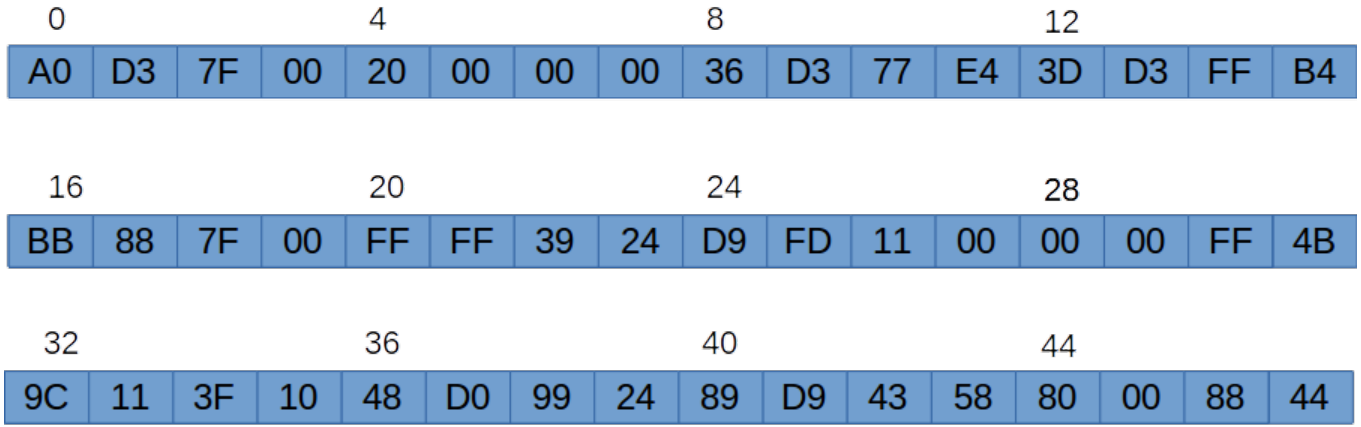
ebp = 4 → return address = 0xE477D336



Param1 = 0xB4FFD33D Param 2 = 0x7F88BB

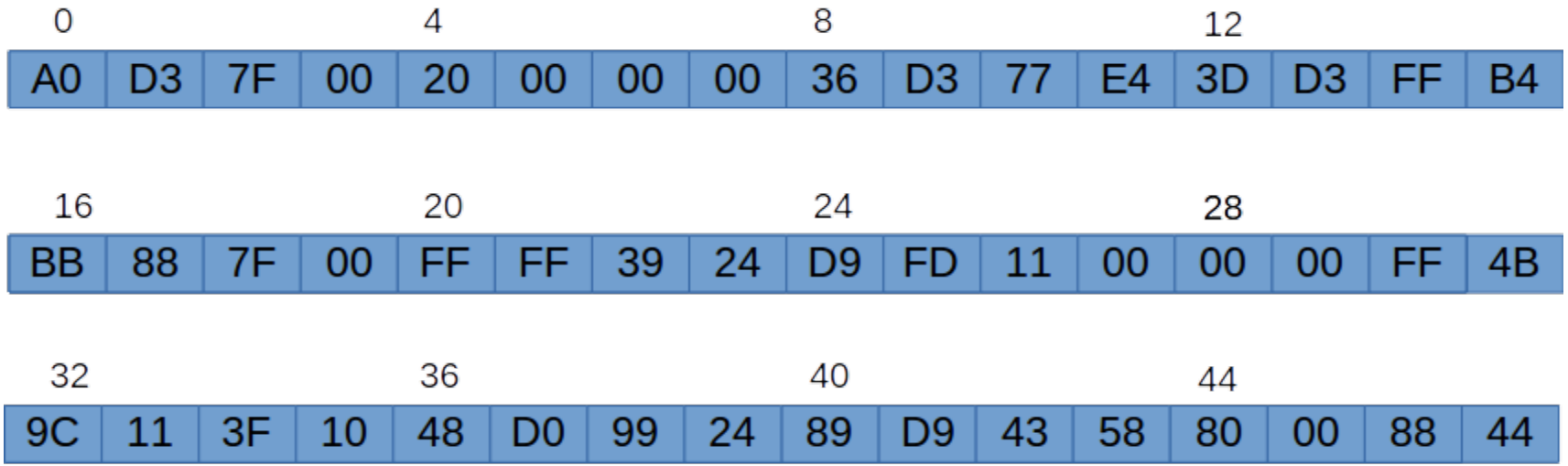
Stack Discipline

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



Stack Discipline

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



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