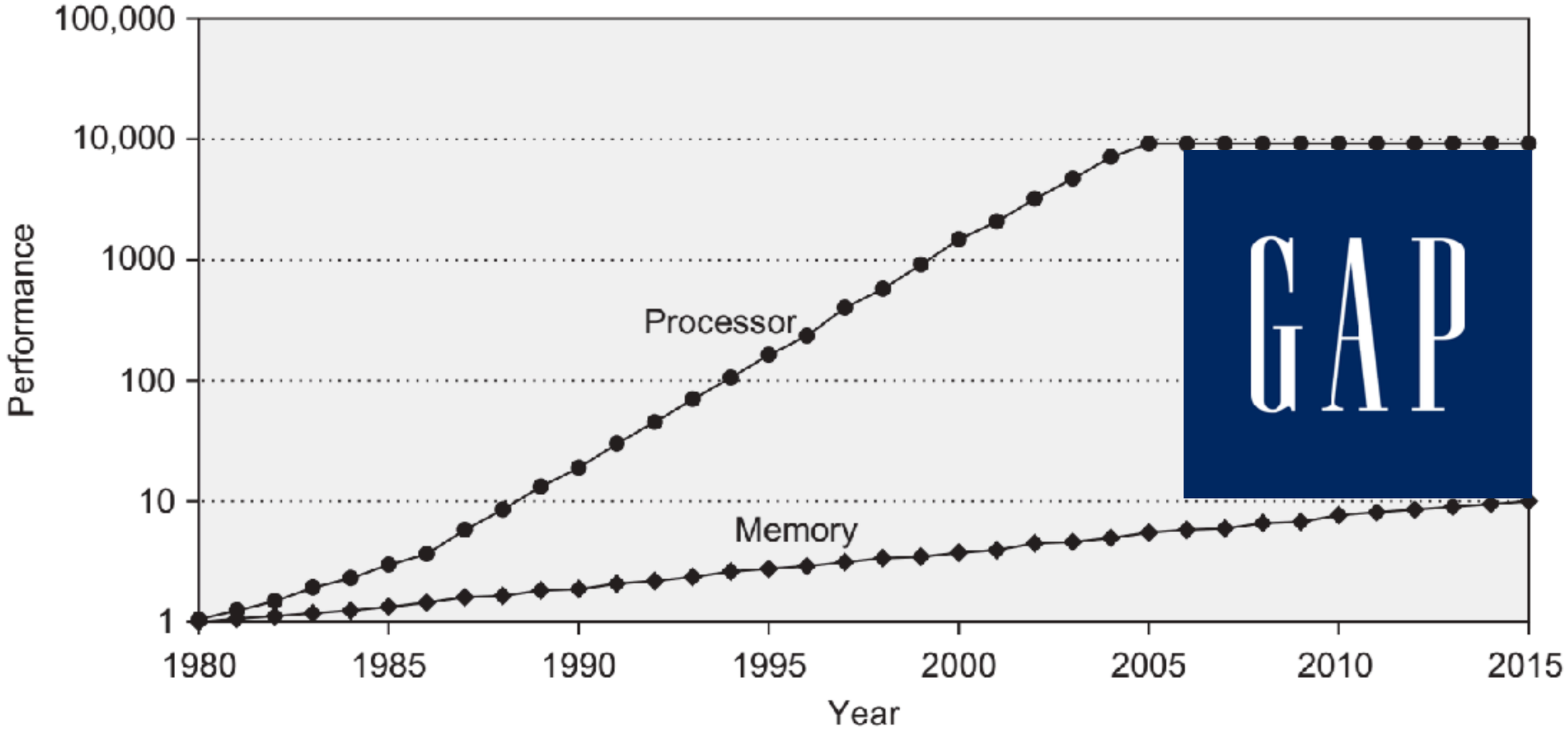
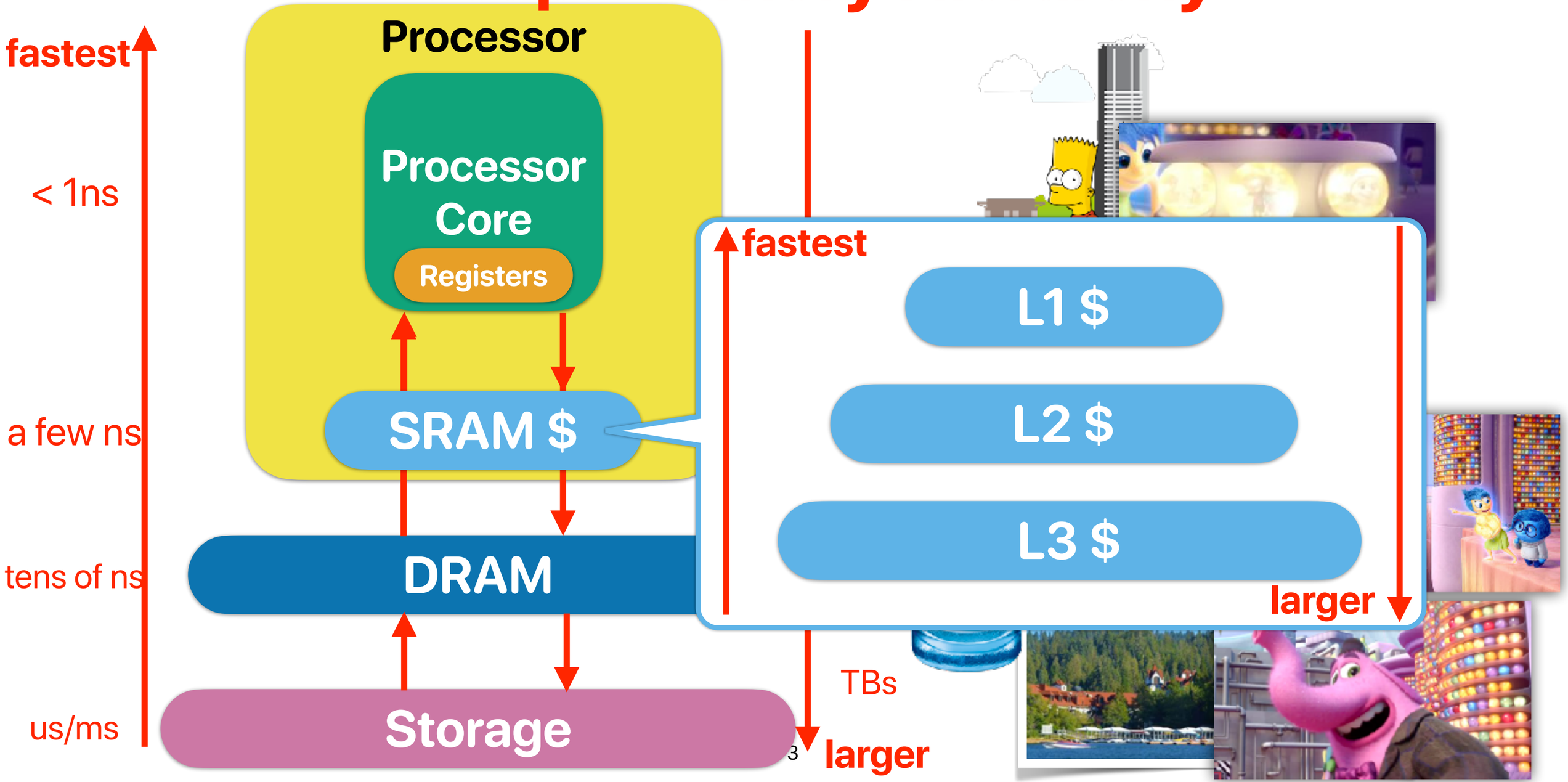
Virtual memory & memory hierarchy

Hung-Wei Tseng

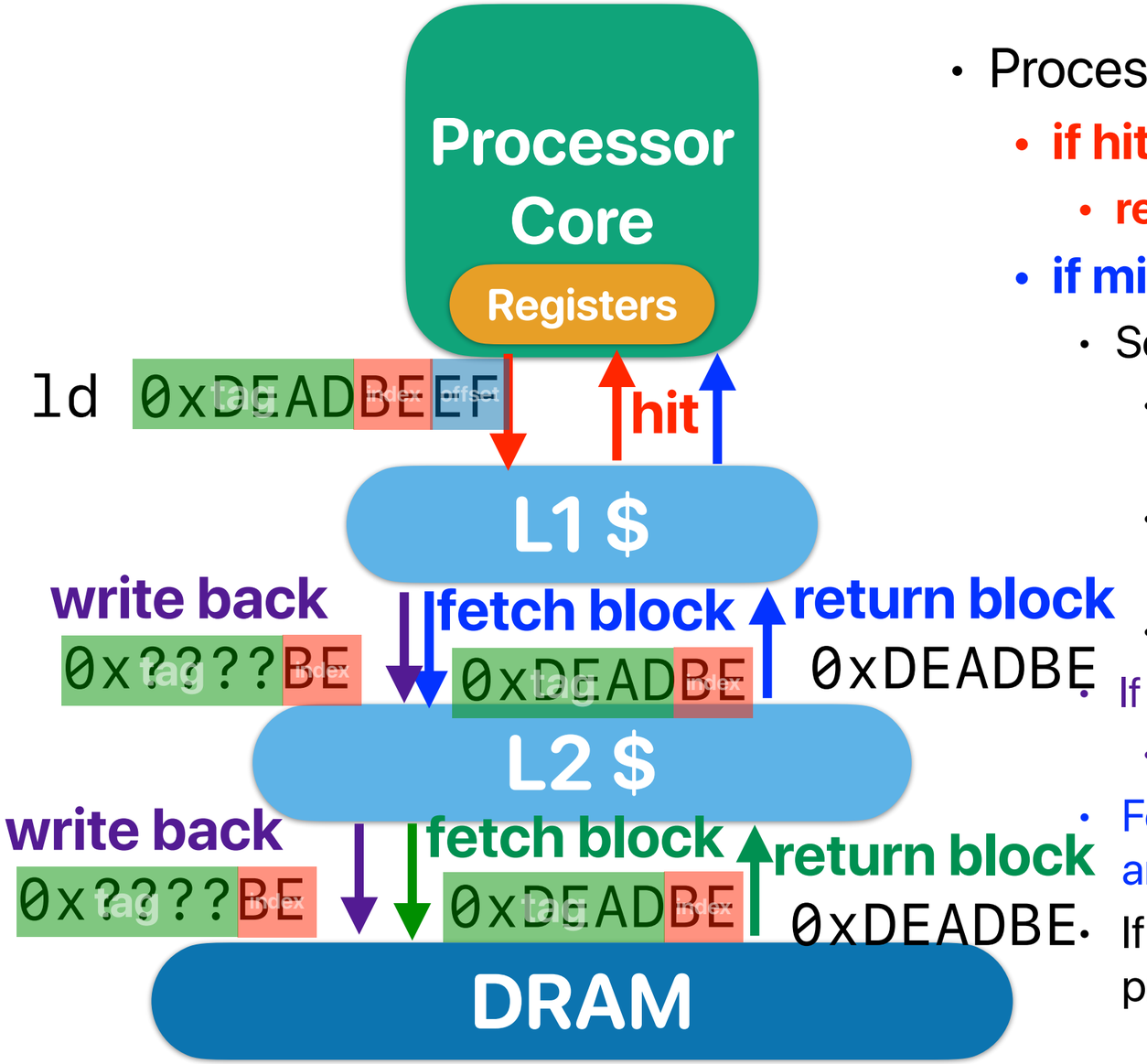
Performance gap between Processor/Memory



Recap: Memory Hierarchy

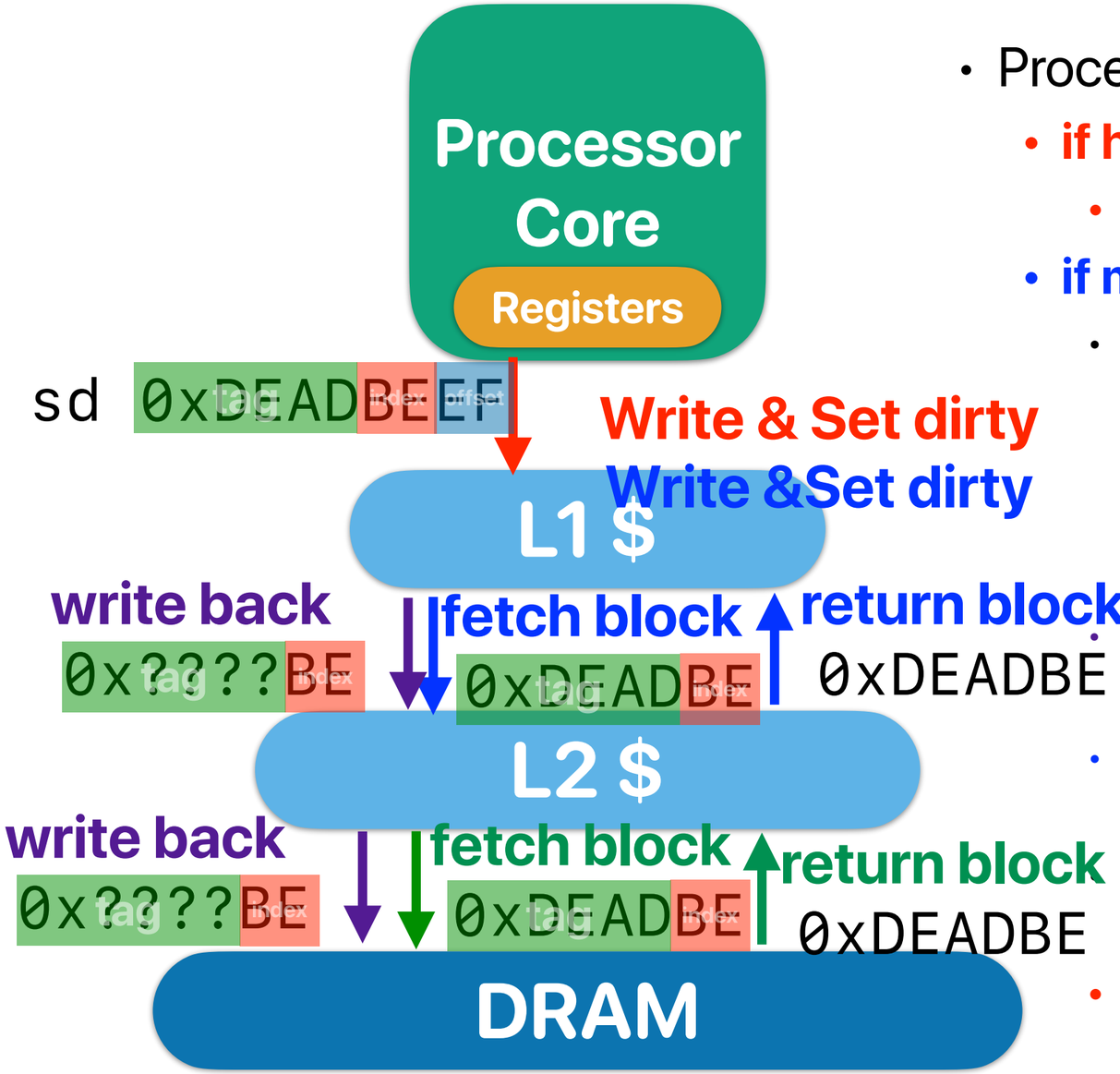


What happens when we read data



- Processor sends load request to L1-$
  - **if hit**
    - **return data**
  - **if miss**
    - Select a victim block
      - If the target "set" is not full — select an empty/invalidated block as the victim block
      - If the target "set is full — select a victim block using some policy
      - LRU is preferred — to exploit temporal locality!
    - If the victim block is "dirty" & "valid"
      - **Write back** the block to lower-level memory hierarchy
    - Fetch the requesting block from lower-level memory hierarchy and place in the victim block
    - If write-back or fetching causes any miss, repeat the same process

What happens when we write data



- Processor sends load request to L1-$
  - **if hit**
    - **return data — set DIRTY**
  - **if miss**
    - Select a victim block
      - If the target "set" is not full — select an empty/invalidated block as the victim block
      - If the target "set is full — select a victim block using some policy
      - LRU is preferred — to exploit temporal locality!
    - If the victim block is "dirty" & "valid"
      - **Write back** the block to lower-level memory hierarchy
    - Fetch the requesting block from lower-level memory hierarchy and place in the victim block
    - If write-back or fetching causes any miss, repeat the same process
  - **Present the write "ONLY" in L1 and set DIRTY**

Outline

- Virtual memory
- Architectural support for virtual memory
- Advanced hardware support for virtual memory

Let’s dig into this code

```
int main(int argc, char *argv[])
{
    int i,j;
    double **a;
    double sum=0, average;
    int dim=32768;
    if(argc < 2)
    {
        fprintf(stderr, "Usage: %s dimension\n",argv[0]);
        exit(1);
    }
    dim = atoi(argv[1]);
    a = (double **)malloc(sizeof(double *)*dim);
    for(i = 0 ; i < dim; i++)
        a[i] = (double *)malloc(sizeof(double)*dim);
    for(i = 0 ; i < dim; i++)
        for(j = 0 ; j < dim; j++)
            a[i][j] = rand();
    for(i = 0 ; i < dim; i++)
        for(j = 0 ; j < dim; j++)
            sum+=a[i][j];
    average = sum/(dim*dim);
    fprintf(stderr,"average: %lf\n",average);
    for(i = 0 ; i < dim; i++)
        free(a[i]);
    free(a);
    return 0;
}
```

Poll close in 1:30

What will happen?

- If we execute the code on the right-hand side code on a machine with only 16 GB of physical memory installed and the dim is "49512" (requires 49512*49512*8 bytes ~ 19 GB memory at least), What will happen?
  A. The program will crash in one of the malloc function call
  B. The program will crash due to a "segmentation fault" that caused by accessing NULL pointer
  C. The program will be killed automatically by the OS as it uses more than installed physical main memory
  D. The program will finish without any issue

```
int main(int argc, char *argv[])
{
    int i,j;
    double **a;
    double sum=0, average;
    int dim=32768;
    if(argc < 2)
    {
        fprintf(stderr, "Usage: %s dimension\n",argv[0]);
        exit(1);
    }
    dim = atoi(argv[1]);
    a = (double **)malloc(sizeof(double *)*dim);
    for(i = 0 ; i < dim; i++)
        a[i] = (double *)malloc(sizeof(double)*dim);
    for(i = 0 ; i < dim; i++)
        for(j = 0 ; j < dim; j++)
            a[i][j] = rand();
    for(i = 0 ; i < dim; i++)
        for(j = 0 ; j < dim; j++)
            sum+=a[i][j];
    average = sum/(dim*dim);
    fprintf(stderr,"average: %lf\n",average);
    for(i = 0 ; i < dim; i++)
        free(a[i]);
    free(a);
    return 0;
}
```

```
#include <stdlib.h>
#include <assert.h>
#include <sched.h>
#include <sys/syscall.h>
#include <time.h>

//#define dim 32768
//#define dim 49152
int main(int argc, char *argv[])
{
    int i,j;
    double **a;
    double sum=0, average;
    int dim=32768;
    if(argc < 2)
    {
        fprintf(stderr, "Usage: %s dimension\n",argv[0]);
        exit(1);
    }
    dim = atoi(argv[1]);
    a = (double **)malloc(sizeof(double *)*dim);
File memory_allocation.c not changed so no update needed
BunnyMACProRetina [/Users/bunny/Dropbox/CSE203/GitHub/demo/virtual_memory] -bunny- ./memory_allocation

BunnyMACProRetina [/Users/bunny/Dropbox/CSE203/GitHub/demo/virtual_memory] -bunny-
```

What will happen?

- If we execute the code on the right-hand side code on a machine with only 16 GB of physical memory installed and the dim is “49512” (requires 49512*49512*8 bytes ~ 19 GB memory at least), What will happen?
  A. The program will crash in one of the malloc function call
  B. The program will crash due to a “segmentation fault” that caused by accessing NULL pointer
  C. The program will be killed automatically by the OS as it uses more than installed physical main memory
  D. The program will finish without any issue

```
int main(int argc, char *argv[])
{
    int i,j;
    double **a;
    double sum=0, average;
    int dim=32768;
    if(argc < 2)
    {
        fprintf(stderr, "Usage: %s dimension\n",argv[0]);
        exit(1);
    }
    dim = atoi(argv[1]);
    a = (double **)malloc(sizeof(double *)*dim);
    for(i = 0 ; i < dim; i++)
        a[i] = (double *)malloc(sizeof(double)*dim);
    for(i = 0 ; i < dim; i++)
        for(j = 0 ; j < dim; j++)
            a[i][j] = rand();
    for(i = 0 ; i < dim; i++)
        for(j = 0 ; j < dim; j++)
            sum+=a[i][j];
    average = sum/(dim*dim);
    fprintf(stderr,"average: %lf\n",average);
    for(i = 0 ; i < dim; i++)
        free(a[i]);
    free(a);
    return 0;
}
```

Let’s dig into this code

```c
#define _GNU_SOURCE
#include <unistd.h>
#include <stdio.h>
#include <stdlib.h>
#include <assert.h>
#include <sched.h>
#include <sys/syscall.h>
#include <time.h>

double a;

int main(int argc, char *argv[])
{
    int i, number_of_total_processes=4;
    number_of_total_processes = atoi(argv[1]);
    // Create processes
    for(i = 0; i< number_of_total_processes-1 && fork(); i++);
    // Generate rand seed
    srand((int)time(NULL)+(int)getpid());
    a = rand();
    fprintf(stderr, "\nProcess %d. Value of a is %lf and address of a is %p\n",getpid(), a, &a);
    sleep(10);
    fprintf(stderr, "\nProcess %d. Value of a is %lf and address of a is %p\n",getpid(), a, &a);
    return 0;
}
```

Poll close in 1:30

Consider the following code …

- Consider the case when we run multiple instances of the given program at the same time on modern machines, which pair of statements is correct?
  - ① The printed "address of a" is the same for every running instances
  - ② The printed "address of a" is different for each instance
  - ③ All running instances will print the same value of a
  - ④ Some instances will print the same value of a
  - ⑤ Each instance will print a different value of a
  - A. (1) & (3)
  - B. (1) & (4)
  - C. (1) & (5)
  - D. (2) & (3)
  - E. (2) & (4)

```c
#define _GNU_SOURCE
#include <unistd.h>
#include <stdio.h>
#include <stdlib.h>
#include <assert.h>
#include <sched.h>
#include <sys/syscall.h>
#include <time.h>

double a;

int main(int argc, char *argv[])
{
    int i, number_of_total_processes=4;
    number_of_total_processes = atoi(argv[1]);
    for(i = 0; i< number_of_total_processes-1 && fork(); i++);
    srand((int)time(NULL)+(int)getpid());
    fprintf(stderr, "\nProcess %d. Value of a is %lf and address
of a is %p\n",getpid(), a, &a);
    sleep(10);
    fprintf(stderr, "\nProcess %d. Value of a is %lf and address
of a is %p\n",getpid(), a, &a);
    return 0;
}
```

```
終端機 — -tcsh — 102×25
BunnyMACProRetina [/Users/bunny/Dropbox/CSE203/GitHub/demo/virtual_memory] -bunny- make clean
rm -f virtualization
BunnyMACProRetina [/Users/bunny/Dropbox/CSE203/GitHub/demo/virtual_memory] -bunny-
```

Demo revisited

```
    sleep(10);
    fprintf(stderr, "\nProcess %d: Value of a is %lf and address of a is %p\n",(int)getpid(), a, &a);
    return 0;
}
File virtualization.c not changed so no update needed
BunnyMACProRetina [/Users/bunny/Dropbox/CSE203/GitHub/demo/virtual_memory] -bunny- make
gcc -O3 virtualization.c -o virtualization
BunnyMACProRetina [/Users/bunny/Dropbox/CSE203/GitHub/demo/virtual_memory] -bunny- ./virtualization 4

Process 19719: Value of a is 1671139616.000000 and address of a is 0x104967050

Process 19720: Value of a is 1671156423.000000 and address of a is 0x104967050

Process 19718: Value of a is 1671122809.000000 and address of a is 0x104967050

Process 19721: Value of a is 1671173230.000000 and address of a is 0x104967050

Process 19719: Value of a is 1671139616.000000 and address of a is 0x104967050

Process 19721: Value of a is 1671173230.000000 and address of a is 0x104967050

Process 19720: Value of a is 1671156423.000000 and address of a is 0x104967050

Process 19718: Value of a is 1671122809.000000 and address of a is 0x104967050
BunnyMACProRetina [/Users/bunny/Dropbox/CSE203/GitHub/demo/virtual_memory] -bunny-
```

Different values

Different values are preserved

The same memory address!

Consider the following code …

- Consider the case when we run multiple instances of the given program at the same time on modern machines, which pair of statements is correct?
  - ① The printed "address of a" is the same for every running instances
  - ② The printed "address of a" is different for each instance
  - ③ All running instances will print the same value of a
  - ④ Some instances will print the same value of a
  - ⑤ Each instance will print a different value of a
  - A. (1) & (3)
  - B. (1) & (4)
  - C. (1) & (5)
  - D. (2) & (3)
  - E. (2) & (4)

```c
#define _GNU_SOURCE
#include <unistd.h>
#include <stdio.h>
#include <stdlib.h>
#include <assert.h>
#include <sched.h>
#include <sys/syscall.h>
#include <time.h>

double a;

int main(int argc, char *argv[])
{
    int i, number_of_total_processes=4;
    number_of_total_processes = atoi(argv[1]);
    for(i = 0; i< number_of_total_processes-1 && fork(); i++);
    srand((int)time(NULL)+(int)getpid());
    fprintf(stderr, "\nProcess %d. Value of a is %lf and address
of a is %p\n",getpid(), a, &a);
    sleep(10);
    fprintf(stderr, "\nProcess %d. Value of a is %lf and address
of a is %p\n",getpid(), a, &a);
    return 0;
}
```

Virtual Memory

Virtual memory

This approach is called demand paging + swapping

Program A

0f00bb27	00c2e800
509cbd23	00000008
00005d24	00c2f000
0000bd24	00000008
2ca422a0	00c2f800
130020e4	00000008
00003d24	00c30000
2ca4e2b3	00000008

(Data)

Processor

Program B

0f00bb27	00c2e800
509cbd23	00000008
00005d24	00c2f000
0000bd24	00000008
2ca422a0	00c2f800
130020e4	00000008
00003d24	00c30000
2ca4e2b3	00000008

(Data)

PC for A

PC for A

load

0x0

0f00bb27
509cbd23
00005d24
0000bd24

2ca422a0
130020e4
00003d24
2ca4e2b3

00c2e800
00000008
00c2f000
00000008

00c2f800
00000008
00c30000
00000008

Virtual Memory

$2^{64}-1$

page

PC for B

PC for B

load

load

0x0

0f00bb27
509cbd23
00005d24
0000bd24

2ca422a0
130020e4
00003d24
2ca4e2b3

00c2e800
00000008
00c2f000
00000008

00c2f800
00000008
00c30000
00000008

Virtual Memory

$2^{64}-1$

00c2f800
00000008
00c30000
00000008

00c2e800
00000008
00c2f000
00000008

2ca422a0
130020e4
00003d24
2ca4e2b3

00c2e800
00000008
00c2f000
00000008

0f00bb27
509cbd23
00005d24
0000bd24

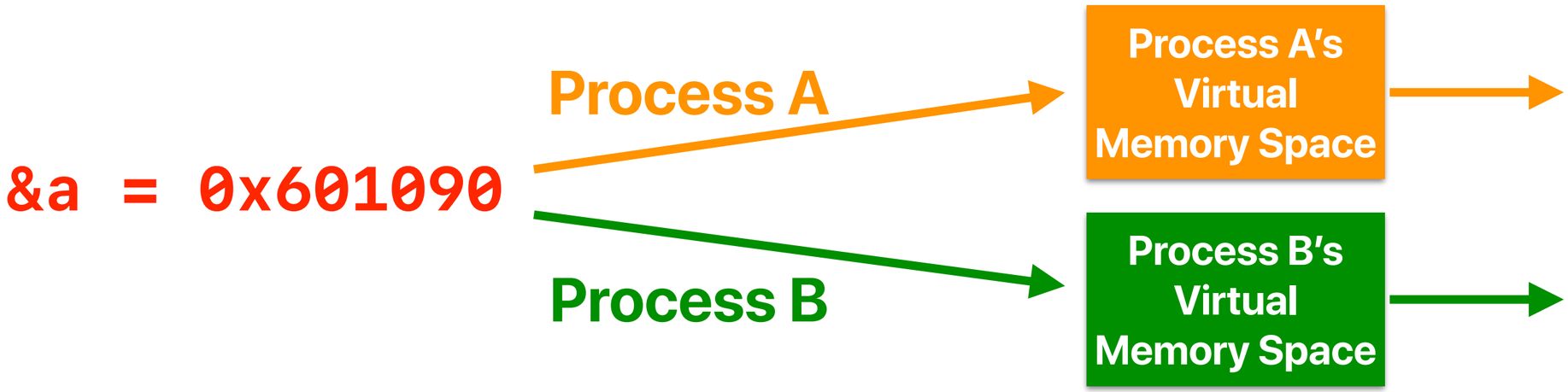
2ca422a0
130020e4
00003d24
2ca4e2b3

Physical Memory

0f00bb27
509cbd23
00005d24
0000bd24

Swap

Demo revisited




```c
#define _GNU_SOURCE
#include <unistd.h>
#include <stdio.h>
#include <stdlib.h>
#include <assert.h>
#include <sched.h>
#include <sys/syscall.h>
#include <time.h>

double a;

int main(int argc, char *argv[])
{
    int i, number_of_total_processes=4;
    number_of_total_processes = atoi(argv[1]);
    for(i = 0; i< number_of_total_processes-1 && fork(); i++);
    srand((int)time(NULL)+(int)getpid());
    fprintf(stderr, "\nProcess %d. Value of a is %lf and address of a is %p\n",getpid(), a, &a);
    sleep(10);
    fprintf(stderr, "\nProcess %d. Value of a is %lf and address of a is %p\n",getpid(), a, &a);
    return 0;
}
```

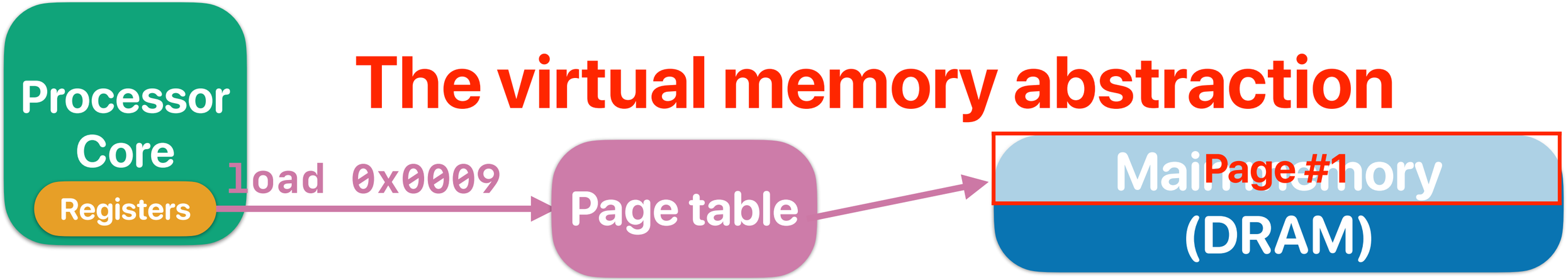
Virtual memory

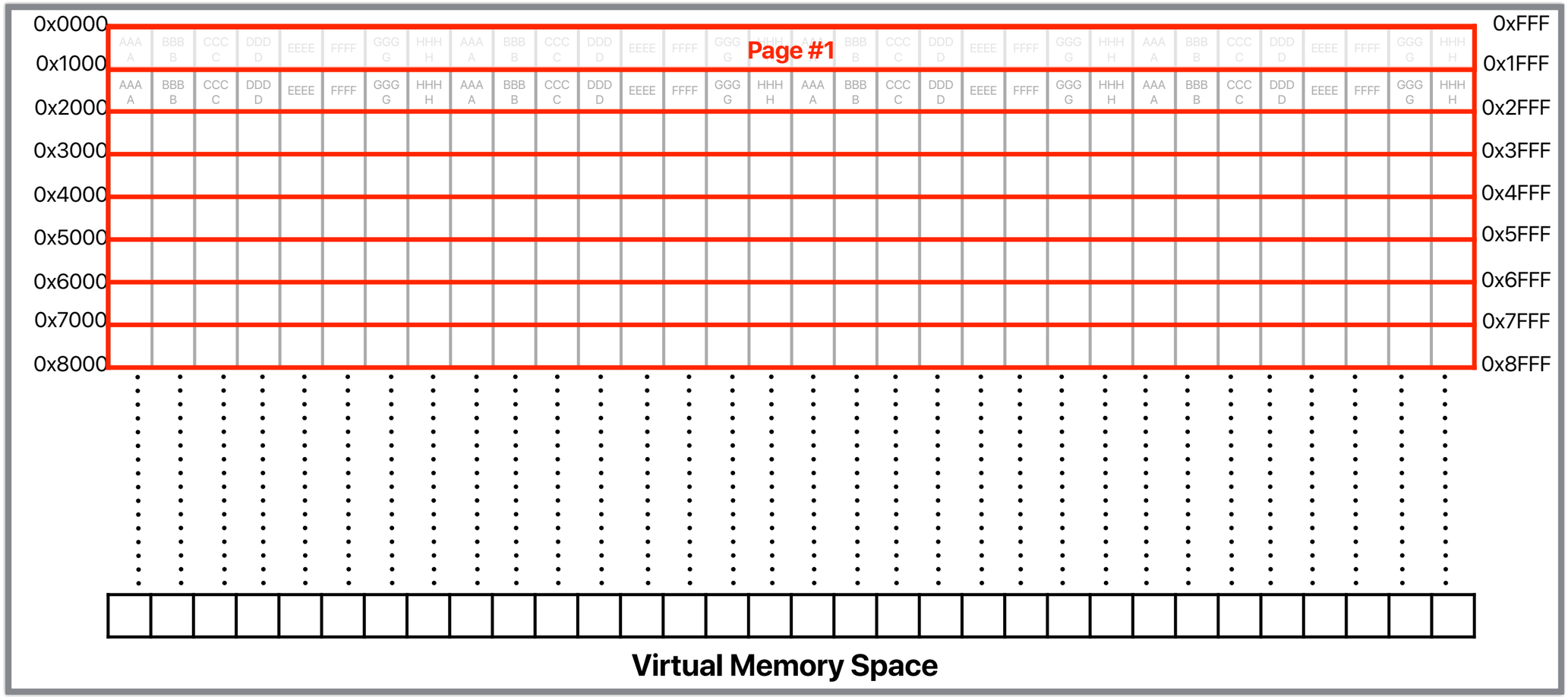
- An **abstraction** of memory space available for programs/software/programmer
- Programs execute using virtual memory address
- The operating system and hardware work together to handle the mapping between virtual memory addresses and real/physical memory addresses
- Virtual memory organizes memory locations into "**pages**"

Why Virtual memory?

- Allowing multiple applications to share physical main memory
  - Memory protection/isolation among programs/processes is automatically achieved
- Allowing applications to work even the installed physical memory or available physical memory is smaller than the working set of the application
  - Programmer does not need to worry about the physical memory capacity of different machines — make compiled program compatible
  - Multiple programs can work concurrently even through their total memory demand is larger than the installed physical memory

The virtual memory abstraction



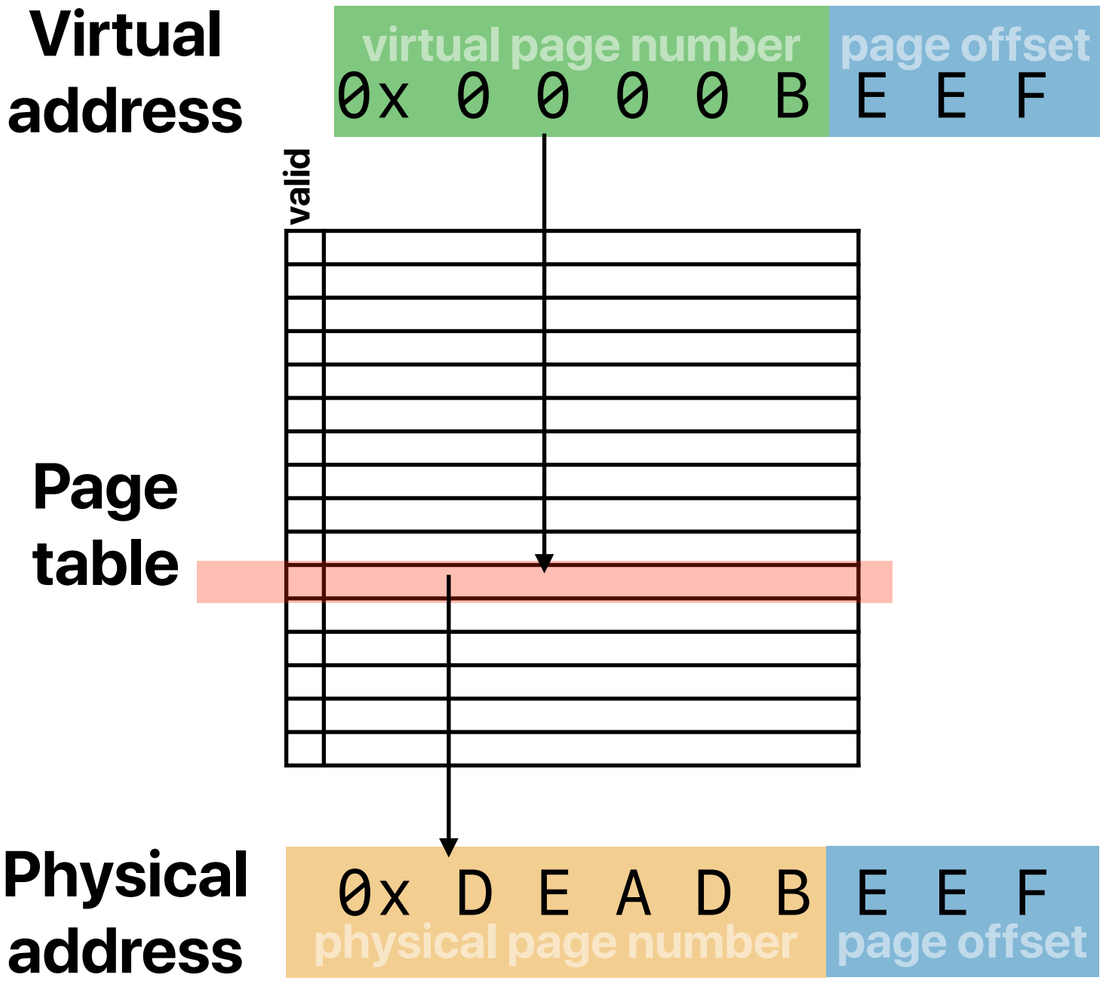


Demand paging

- Treating physical main memory as a "cache" of virtual memory
- The block size is the "page size"
- The page table is the "tag array"
- It's a "fully-associate" cache — a virtual page can go anywhere in the physical main memory

Address translation

- Processor receives virtual addresses from the running code, main memory uses physical memory addresses
- Virtual address space is organized into “pages”
- The system references the **page table** to translate addresses
  - Each process has its own page table
  - The page table content is maintained by OS



Poll close in 1:30

Size of page table

- Assume that we have **64-bit** virtual address space, each page is 4KB, each page table entry is 8 Bytes, what magnitude in size is the page table for a process?
  A. MB — 2^{20} Bytes
  B. GB — 2^{30} Bytes
  C. TB — 2^{40} Bytes
  D. PB — 2^{50} Bytes
  E. EB — 2^{60} Bytes

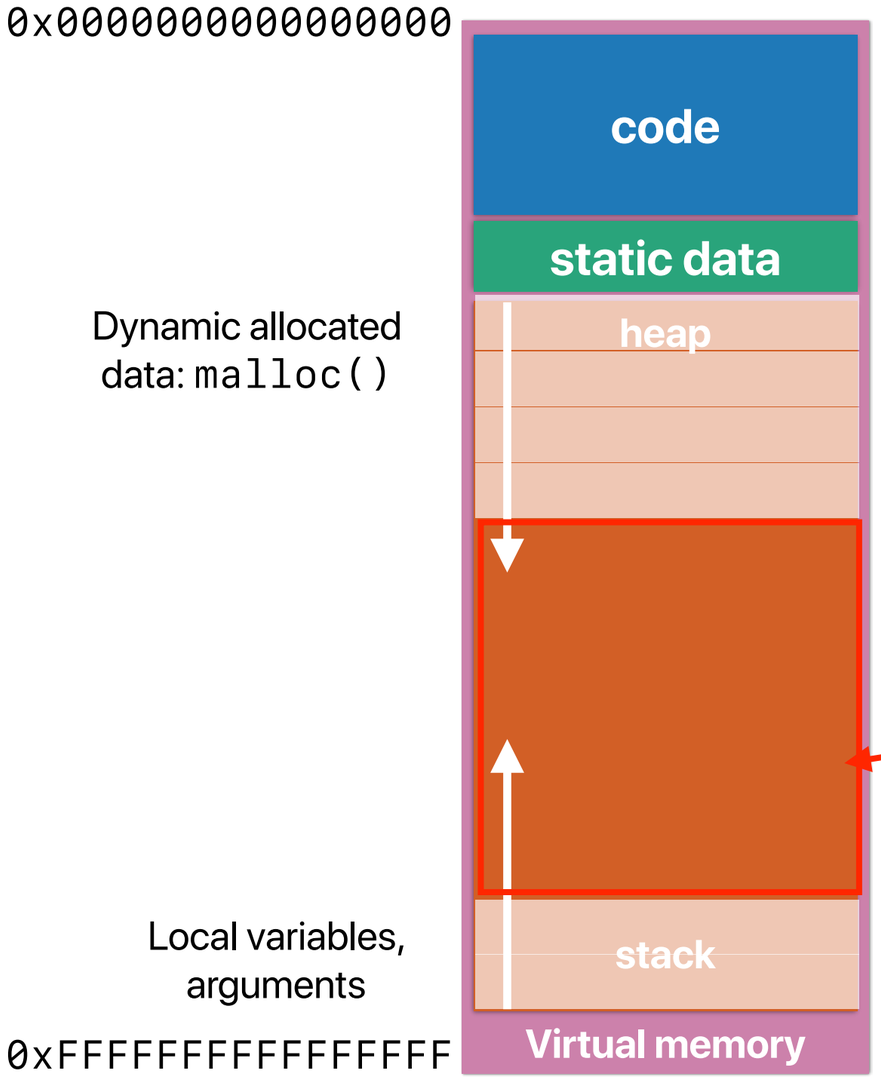
Size of page table

- Assume that we have **64-bit** virtual address space, each page is 4KB, each page table entry is 8 Bytes, what magnitude in size is the page table for a process?
  A. MB — 2^{20} Bytes
  B. GB — 2^{30} Bytes
  C. TB — 2^{40} Bytes
  D. PB — 2^{50} Bytes
  E. EB — 2^{60} Bytes

$$\frac{2^{64}\ Bytes}{4\ KB} \times 8\ Bytes = 2^{55}\ Bytes = 32\ PB$$

If you still don't know why — you need to take CS202

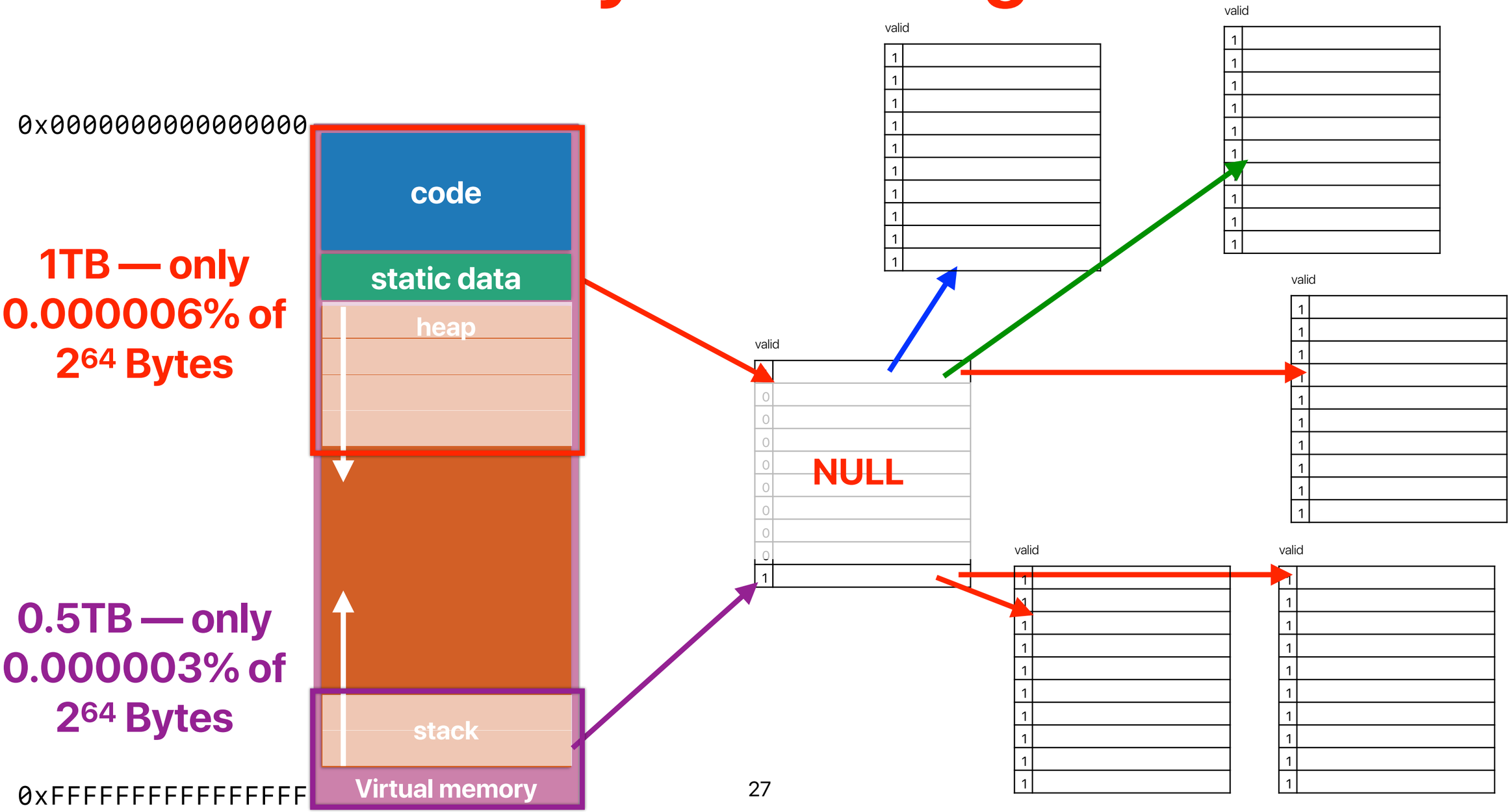
Do we really need a large table?



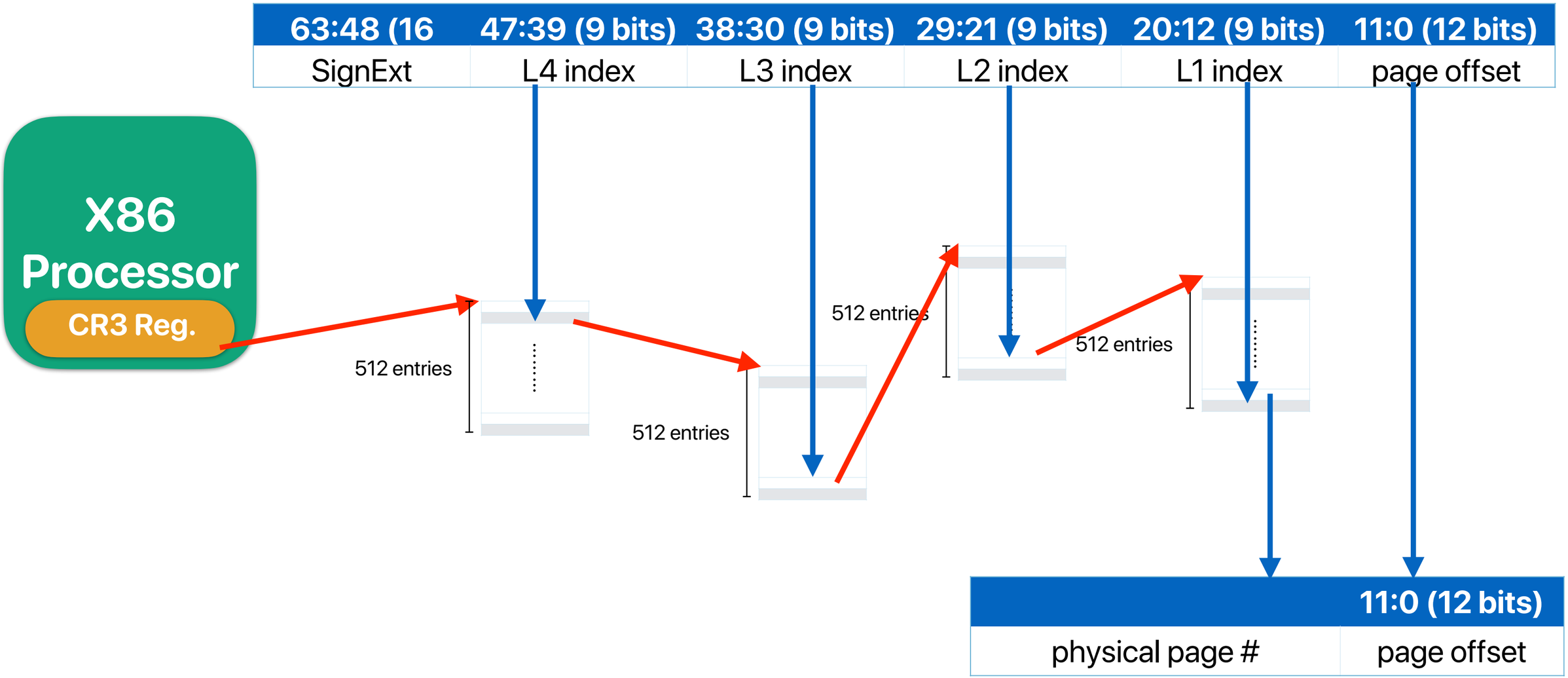
Your program probably never uses this huge area!

If you still don't know why — you need to take CS202

Do we really need a large table?



Address translation in x86-64



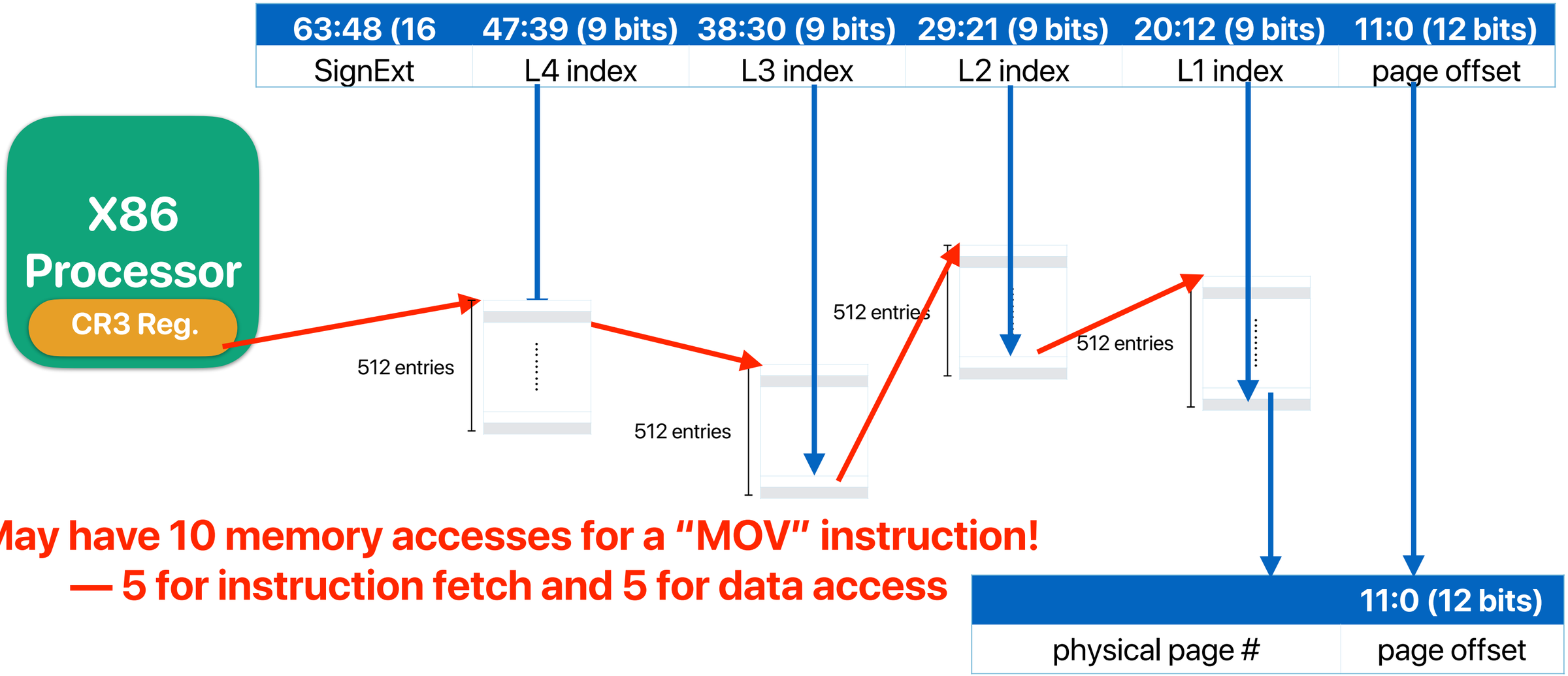
Translation Caching: Skip, Don't Walk (the Page Table)
Thomas W. Barr, Alan L. Cox, Scott Rixner

Poll close in 1:30

When we have virtual memory…

- If an x86 processor supports virtual memory through the basic format of the page table as shown in the previous slide, how many memory accesses can a **`mov`** instruction that access data memory once incur?
  A. 2
  B. 4
  C. 6
  D. 8
  E. 10

Address translation in x86-64



May have 10 memory accesses for a "MOV" instruction!

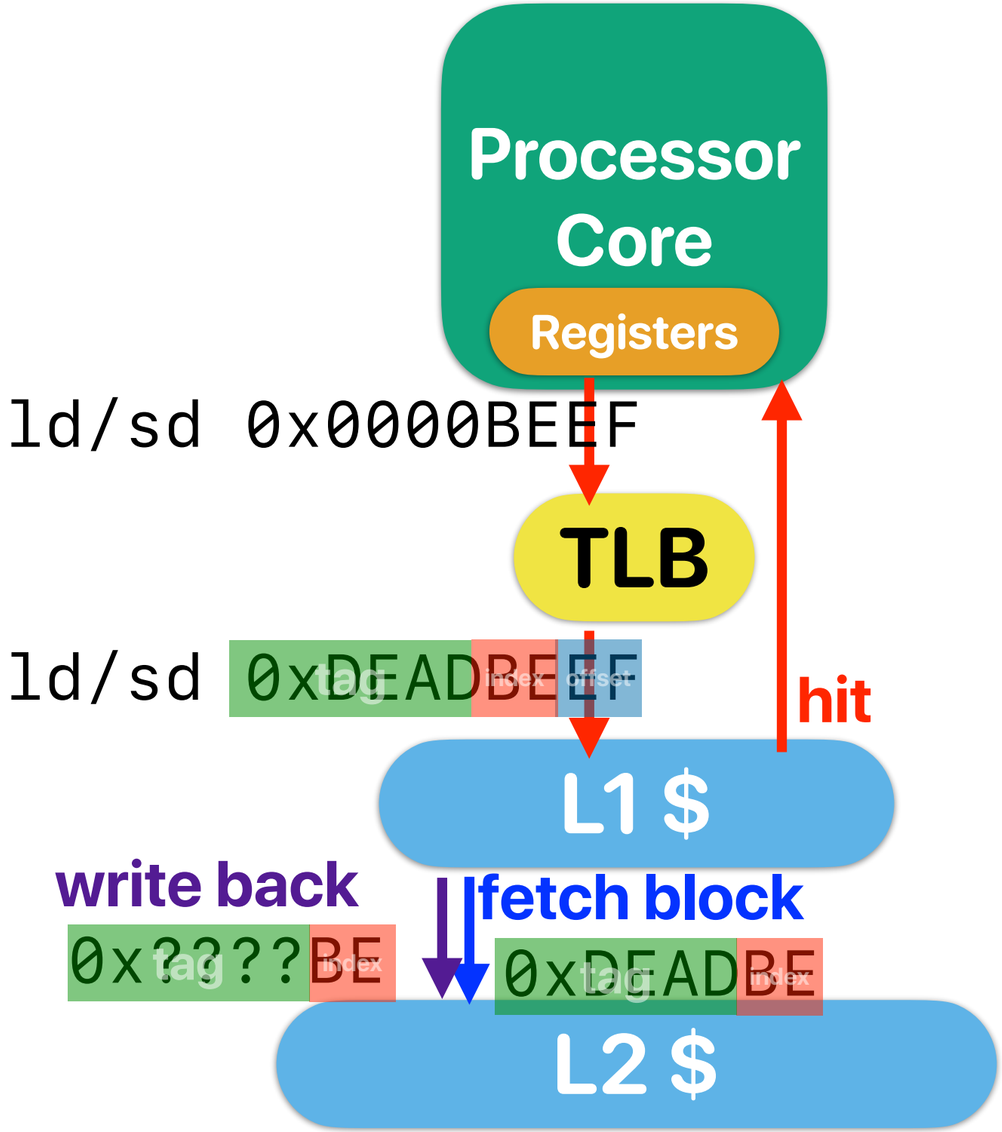
— 5 for instruction fetch and 5 for data access

When we have virtual memory…

- If an x86 processor supports virtual memory through the basic format of the page table as shown in the previous slide, how many memory accesses can a **`mov`** instruction that access data memory once incur?
  A. 2
  B. 4
  C. 6
  D. 8
  E. 10

Avoiding the address translation overhead

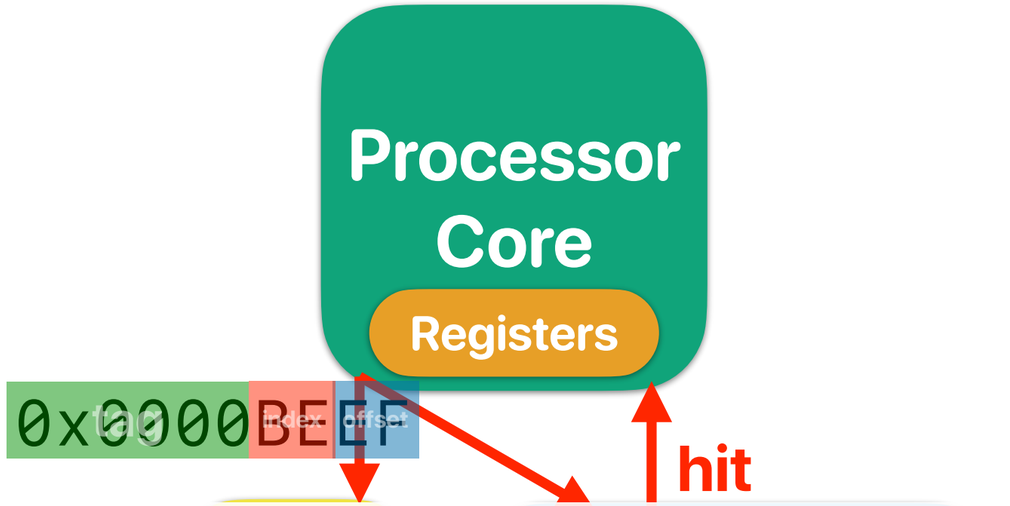
TLB: Translation Look-aside Buffer



- TLB — a small SRAM stores frequently used page table entries
- Good — A lot faster than having everything going to the DRAM
- Bad — Still on the critical path

TLB + Virtual cache

- L1 $ accepts virtual address — you don't need to translate
- Good — you can access both TLB and L1-$ at the same time and physical address is only needed if L1-$ misses
- Bad — it doesn't work in practice
  - Many applications have the same virtual address but should be pointing different **physical addresses**
  - An application can have "aliasing virtual addresses" pointing to the same **physical address**



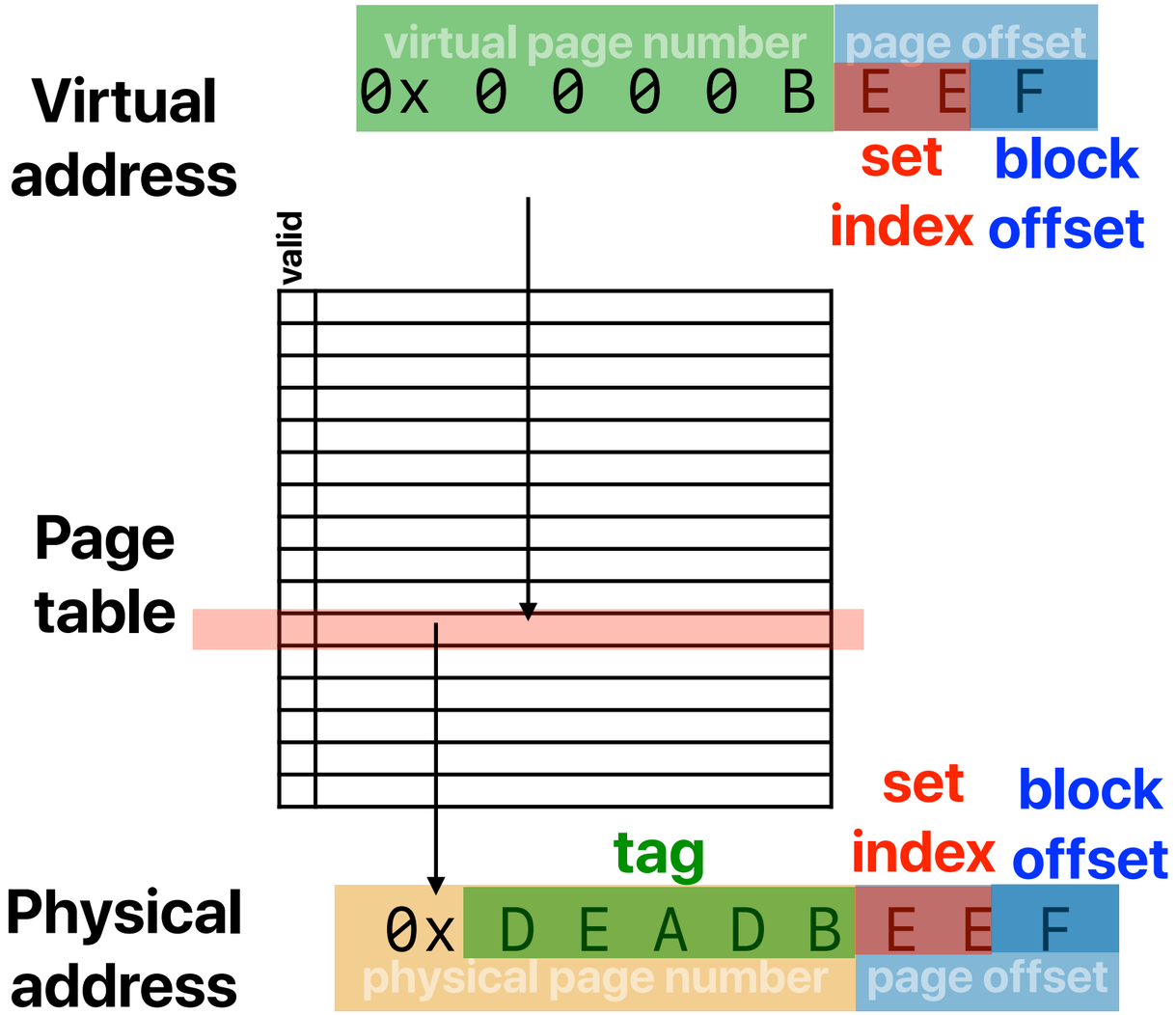
You really need "physical address" to judge if that's what you want

Virtually indexed, physically tagged cache

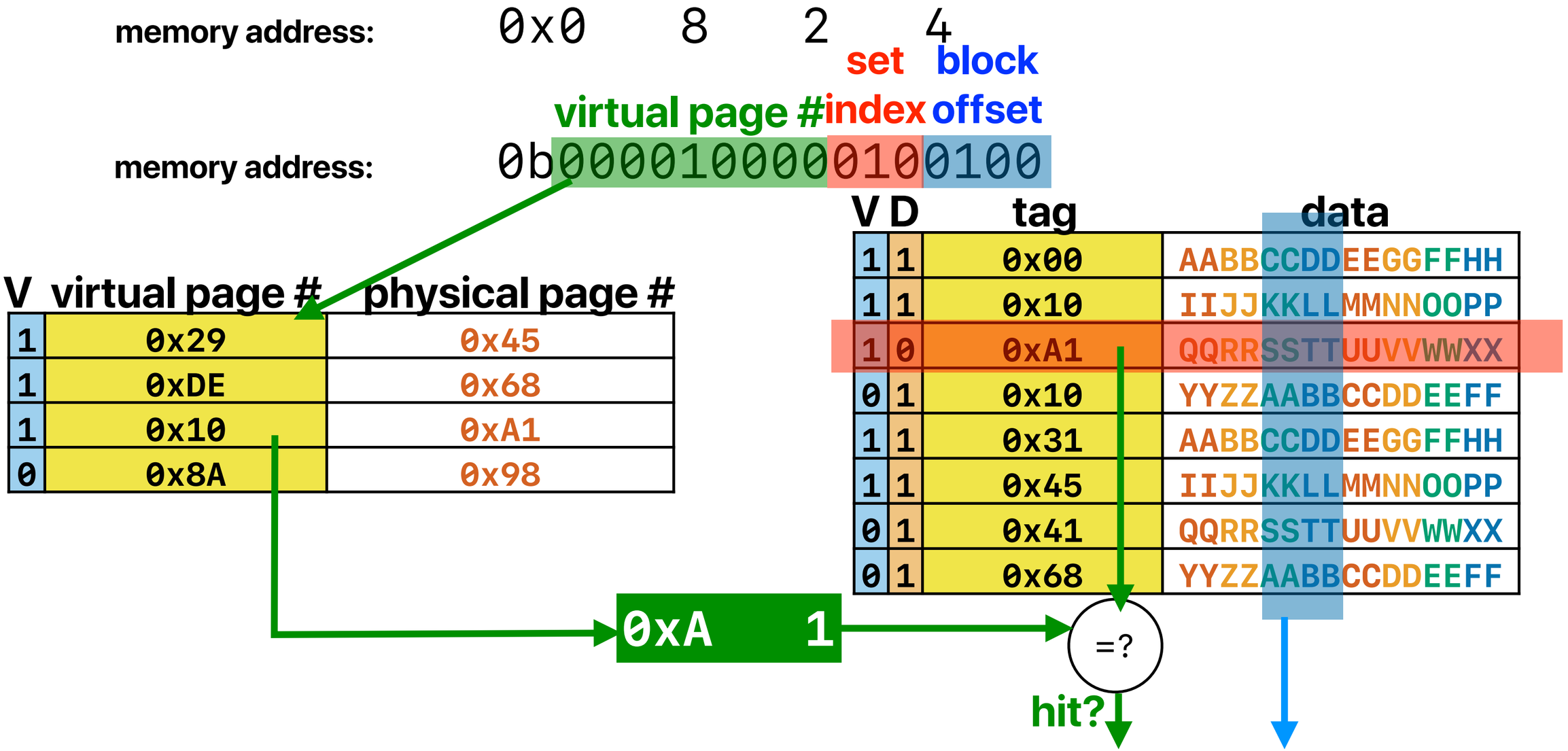
- Can we find physical address directly in the virtual address

  — Not everything — but the page offset isn't changing!
- Can we indexing the cache using the "partial physical address"?

  — Yes — Just make set index + block set to be exactly the page offset



Virtually indexed, physically tagged cache



Virtually indexed, physically tagged cache

- If page size is 4KB —

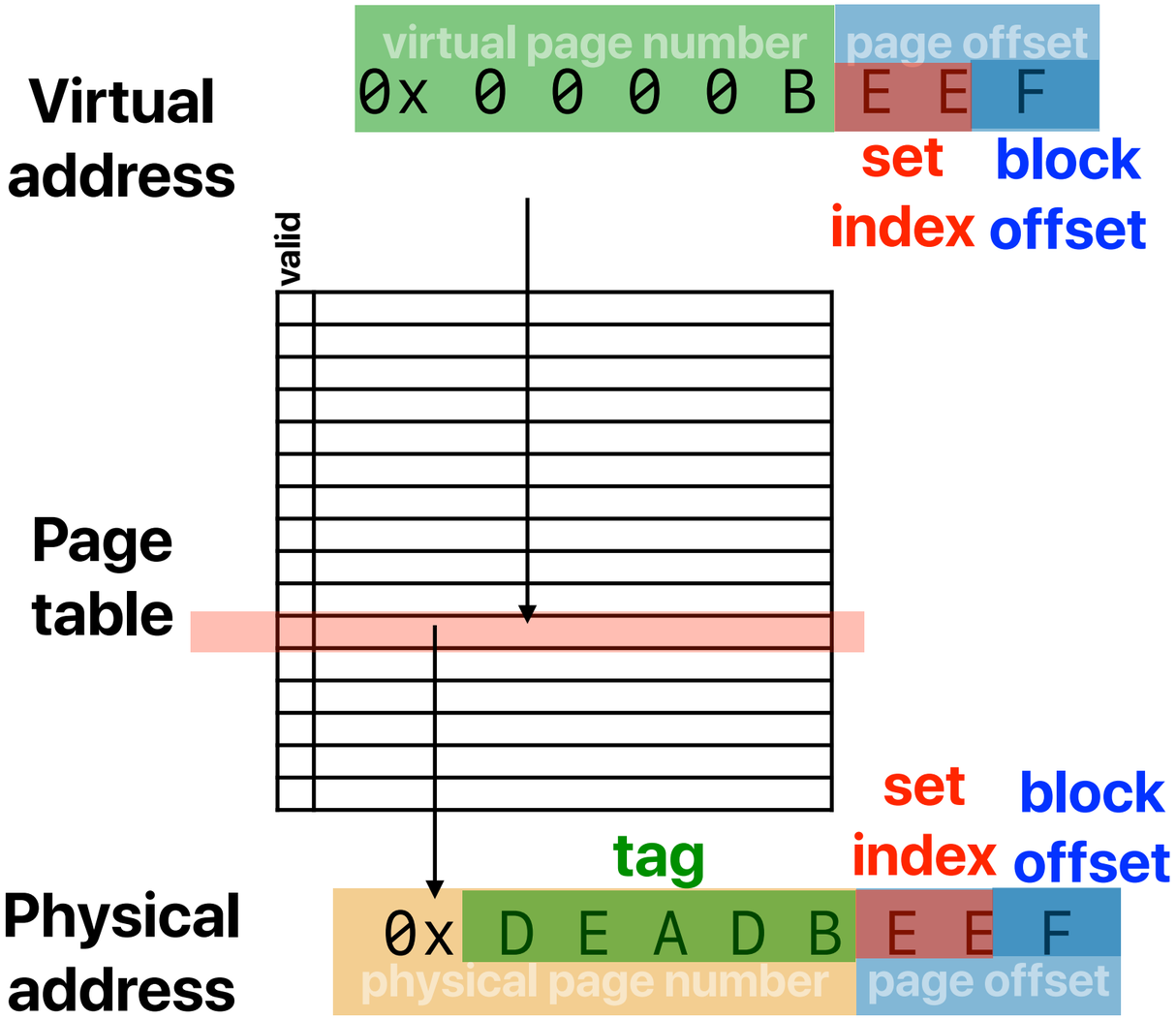
$$lg(B) + lg(S) = lg(4096) = 12$$

$$C = ABS$$

$$C = A \times 2^{12}$$

$$if\ A = 1$$

$$C = 4KB$$



Poll close in 1:30

Virtual indexed, physical tagged cache limits the cache size

- If you want to build a virtual indexed, physical tagged cache with 32KB capacity, which of the following configuration is possible? Assume the operating system use 4K pages.
  A. 32B blocks, 2-way
  B. 32B blocks, 4-way
  C. 64B blocks, 4-way
  D. 64B blocks, 8-way

Virtual indexed, physical tagged cache limits the cache size

- If you want to build a virtual indexed, physical tagged cache with 32KB capacity, which of the following configuration is possible? Assume the operating system use 4K pages.
  A. 32B blocks, 2-way
  B. 32B blocks, 4-way
  C. 64B blocks, 4-way
  D. 64B blocks, 8-way

$$lg(B) + lg(S) = lg(4096) = 12$$

$$C = ABS$$

$$32KB = A \times 2^{12}$$

$$A = 8$$

Exactly how Core i7 configures its own cache

Translation Caching: Skip, Don't Walk (the Page Table)

Thomas W. Barr, Alan L. Cox, Scott Rixner

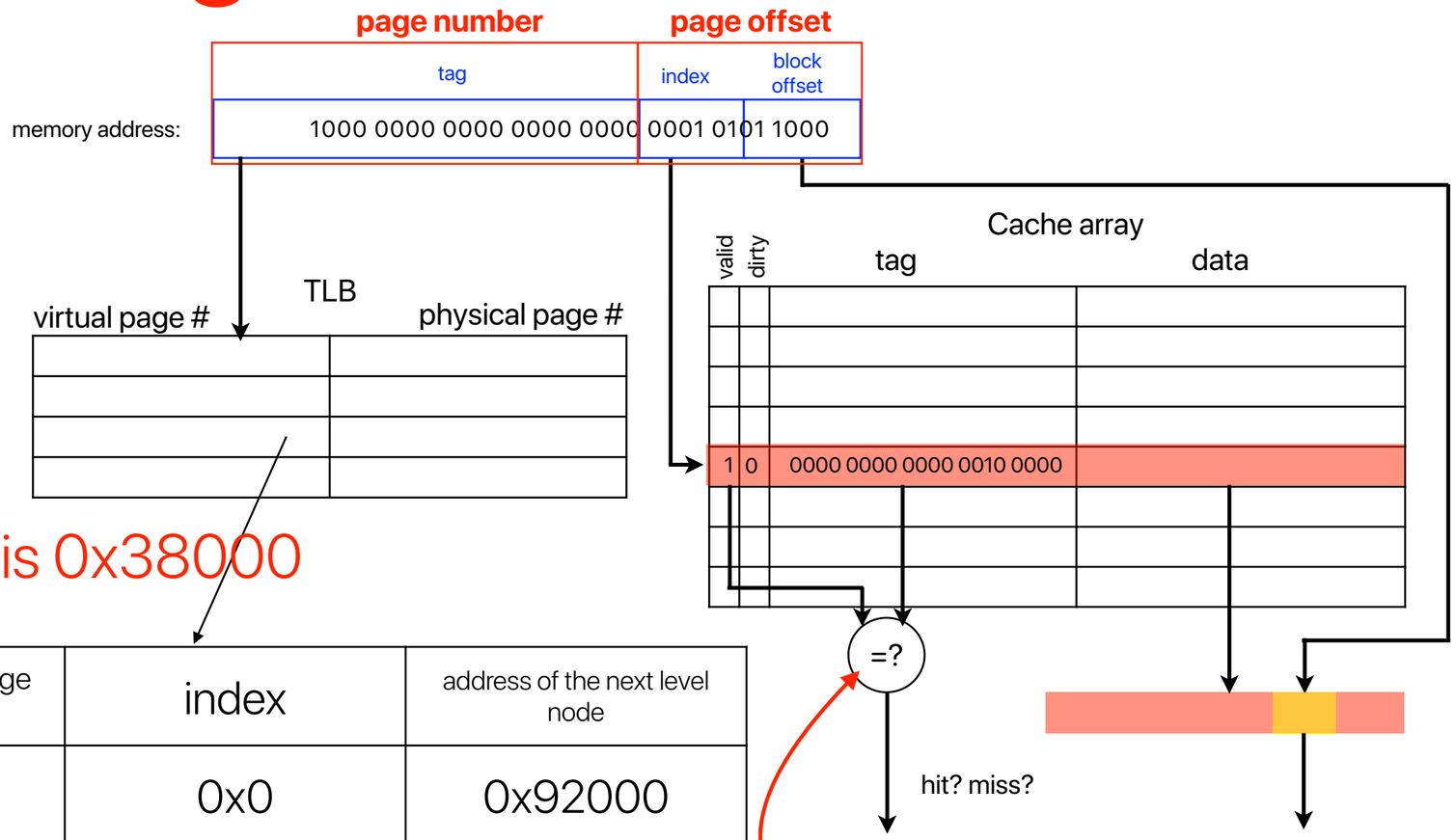
Why should we care about this paper?

- TLB miss is expensive
  - You have to walk through multiple nodes in the hierarchical page table
  - Each node is a memory access — 100 ns
- Modern processors use memory management units (MMUs)
  - MMUs have caches, but not optimized for the timing critical TLB miss
  - Page table caches
  - Translational caches

What this paper proposed

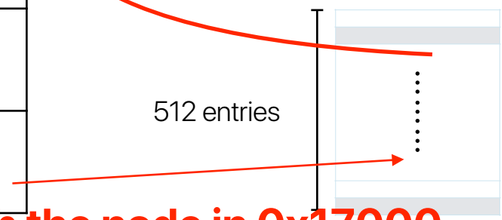
- Not really proposing anything. More an empirical analysis paper
- Design space exploration to find out the optimal solution

Page table caches



assume CR3 is 0x38000

Base address of page table node	index	address of the next level node
0x63000	0x0	0x92000
0x27000	0x23	0x87000
0x38000	0x10	0x63000
0x92000	0x0	0x17000

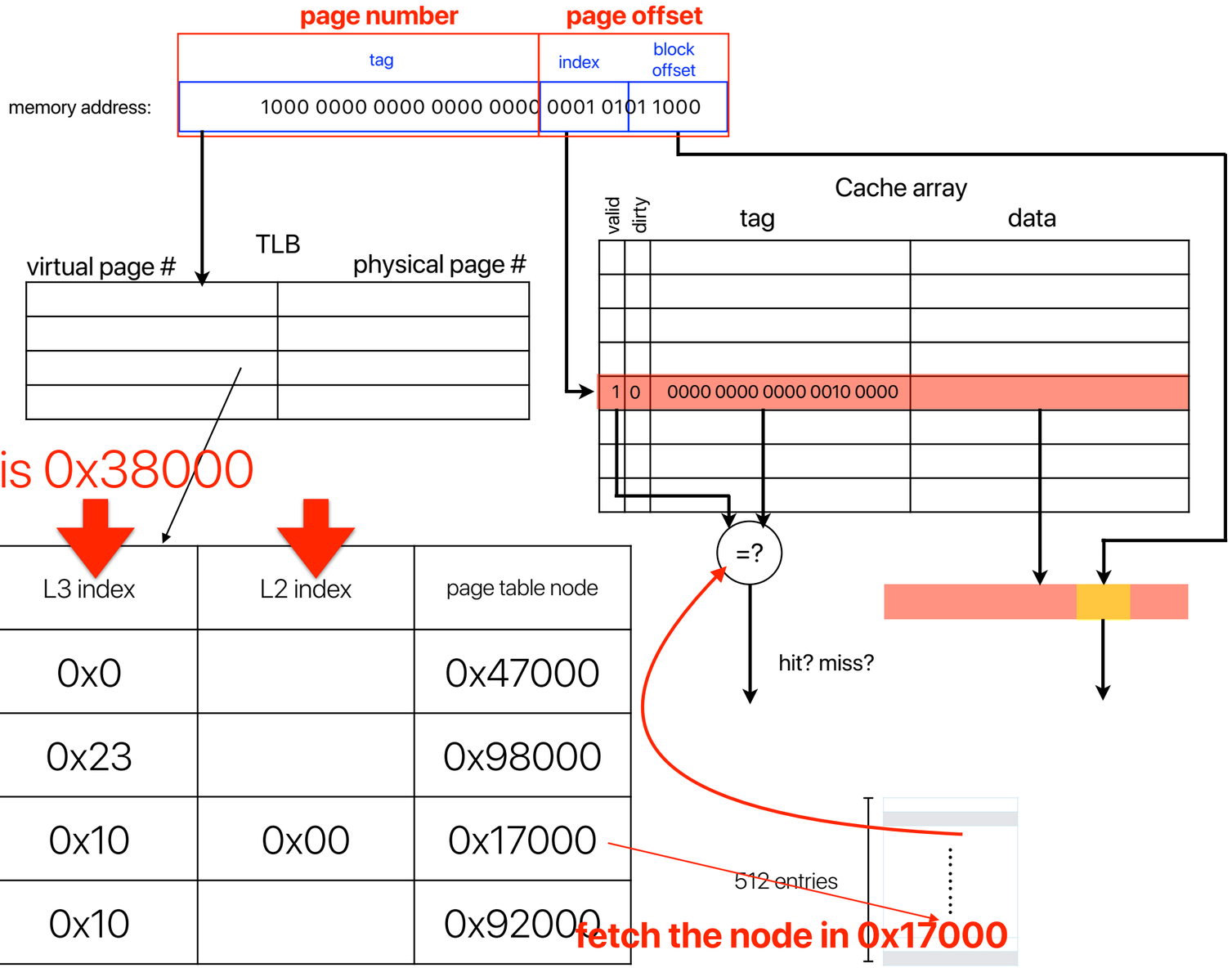


fetch the node in 0x17000

Page table caches

- PTC caches the addresses of “page table nodes”
- PTC uses the physical address of page table nodes as the index
  - Unified page table cache (UPTC)
  - Split page table cache (SPTC)
    - Each page level get a private cache location

Translation cache



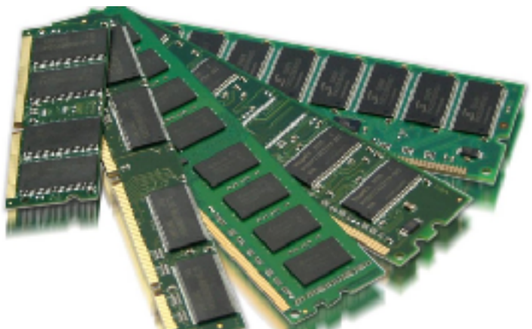
Translation caches

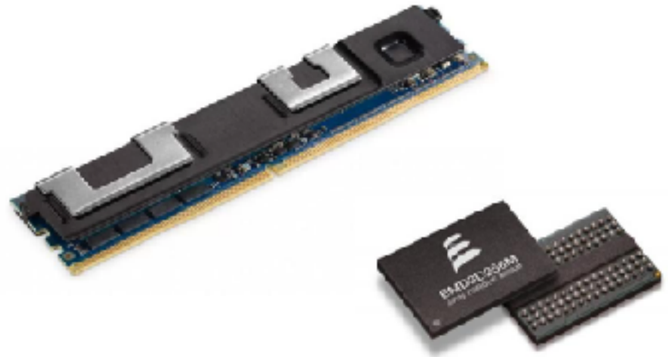
- Indexed by the prefix of the requesting virtual address
  - Split translational cache (STC)
  - Unified translational cache (UTC)
  - Translational-path Cache (TPC)
- Pros:
  - Allowing each level lookup to perform independently, in parallel
- Cons:
  - Less space efficient

Efficient Virtual Memory for Big Memory Servers

Arkaprava Basu, Jayneel Gandhi, Jichuan Chang, Mark D. Hill and Michael M. Swift

Applications with big memory footprints and high-density memory technologies





Why should we care about this paper?

- We care about big memory applications (e.g. memcached)
- Machines with TBs of physical memory are available
- Big-memory workloads pay high costs in paging
  - Up to 51% execution time burned due to TLB miss with 4KB page size
  - 10% of execution time in TLB miss even with 2MB page sizes

Characteristics of Big Memory Apps

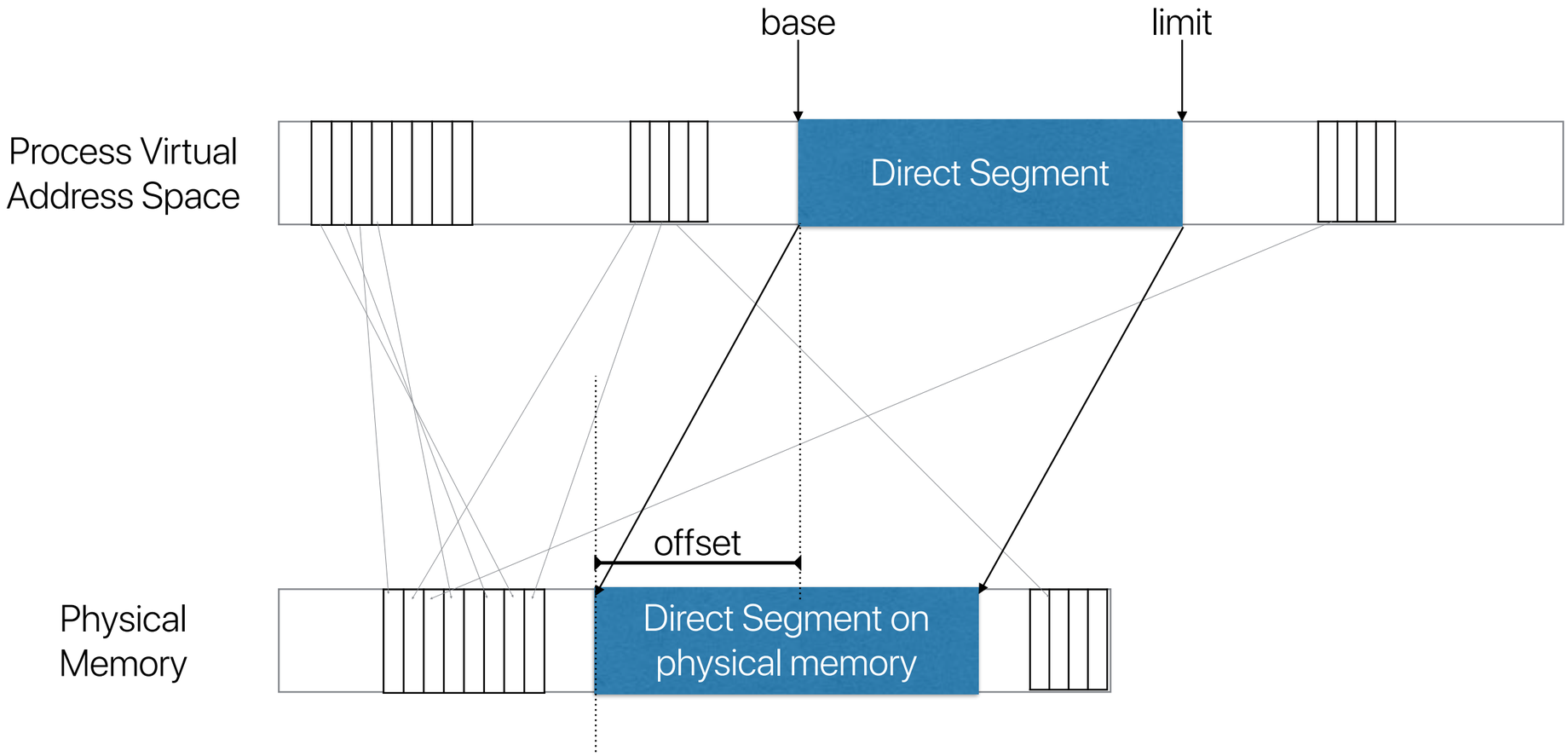
- Few swapping — since the computer has big memory
- Few copy-on-write — since the workload is mostly reads
- Does not require per-page protection

— remind yourself how you usually allocate memory?

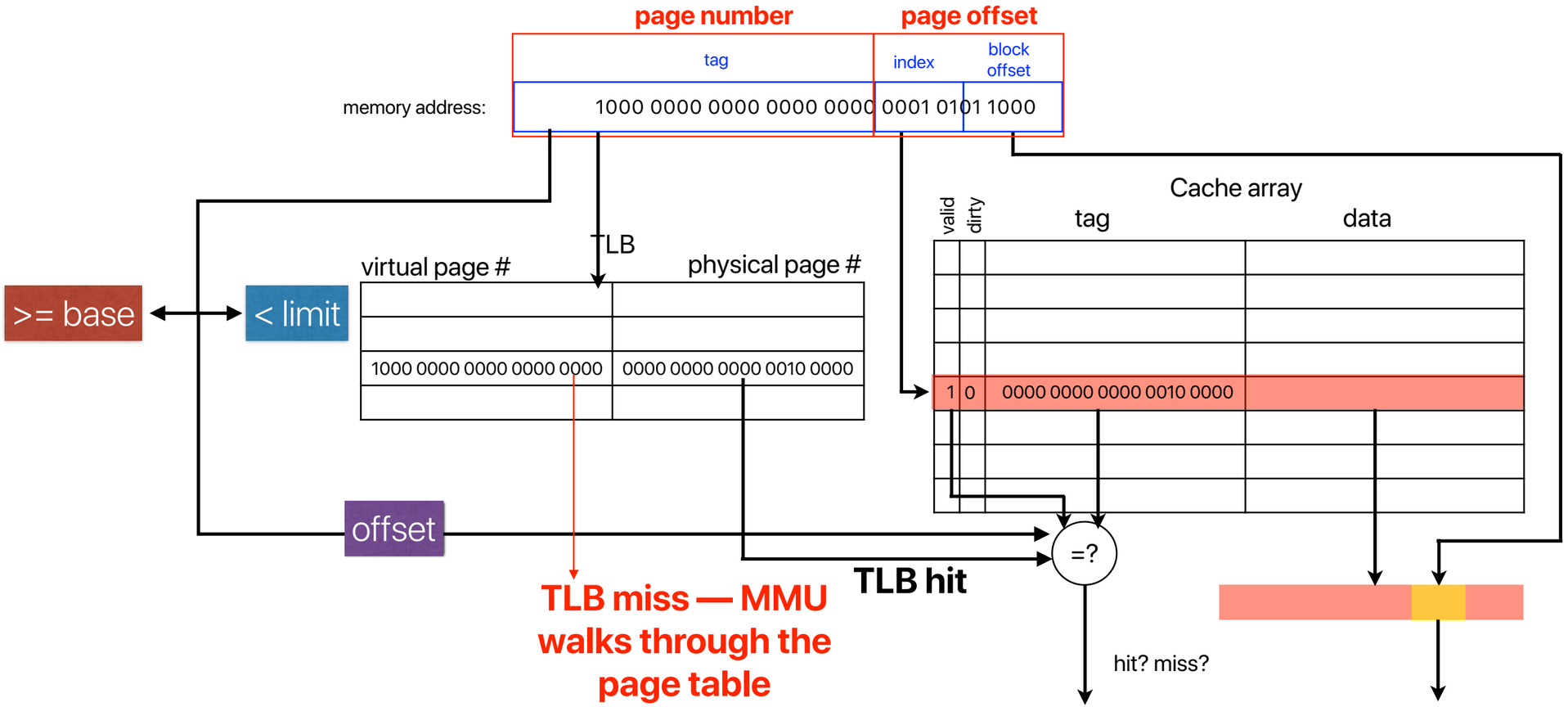
What this paper proposed?

- Mapping part of a process's linear virtual address to a "direct segment" rather than a page
- Direct segment
  - Similar to classic segmentation: adding base, offset, limit registers to each core
  - If the virtual address falls in the range between base and limit, no TLB access is necessary
- Virtual memory outside a direct segment still uses conventional demand paging

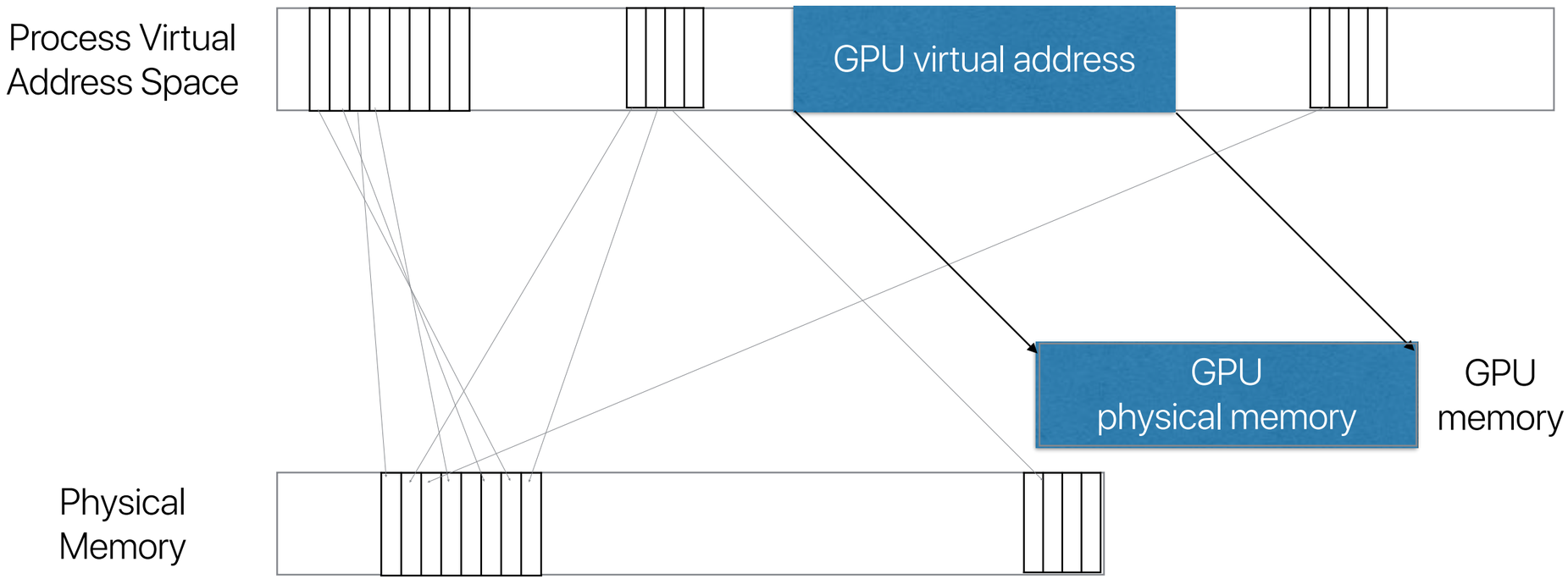
Direct Segment



Architecture overview



Nvidia's unified virtual memory



OS/Software support

- The operating system provides a primary region abstraction
  - 64-bit memory space is big enough to find a contiguous primary region
- The operating system allocates a physical memory region for the primary region using direct segment
  - extending libc function interface — think about your projects
- Optimizing the physical memory allocation
  - memory compaction
  - program architectural registers: base, offset, limit
  - growing/shrinking segments

Why direct-segment instead of large pages

- Comparing direct-segment with using large pages that x86-64 supports, how many of the following statements is/are correct
  - ~~①~~ Direct segment works better when the memory allocator favors sparse virtual memory allocation
  - ② Direct segment is more flexible and efficient in terms of accommodating various sizes of contiguous physical memory region allocated for a contiguous virtual memory region
  - ③ Direct segment is less architecture dependent than large pages
  - ④ Direct segment requires more modifications to the application

  A. 0
  B. 1
  C. 2
  D. 3
  E. 4

In addition, software that depends on sparse virtual memory allocations may waste physical memory if mapped with direct segments. For example, *malloc()* in glibc-2.11 may allocate separate

System software has two basic responsibilities in our proposed design. First, the OS provides a *primary region* abstraction to let applications specify which portion of their memory does not benefit from paging. Second, the OS provisions physical memory for a

Third, large page sizes are often few and far apart. For example in x86-64, the large page sizes correspond to different levels in the hardware-defined multi-level radix-tree structure of the page table. For example, recent x86-64 processors have only three page sizes (4KB, 2MB, 1GB), each of which is *512 times* larger than the previous. This constant factor arises because 4KB pages that hold page tables contain 512 8-byte-wide PTEs at each node of the page table. Such page-size constraints make it difficult to introduce and flexibly use large pages. For example, mapping a 400GB physical memory using 1GB pages can still incur substantial number of TLB misses, while a 512GB page is too large. A direct segment overcomes this shortcoming, as its size can adapt to application or system needs.

Announcement

- Assignment #2 due next Monday
- Reading quiz due next Wednesday
- Project is up — check the website
- Office Hours on Zoom (the office hour link, not the lecture one)
  - Hung-Wei/Prof. Usagi: M 8p-9p, W 2p-3p (cancelled this week), but can answer questions through e-mails. Will make up later in the quarter
  - Quan Fan: F 1p-3p

Computer
Science &
Engineering

203

つづく