Virtual memory & memory hierarchy

Hung-Wei Tseng

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;
}
```

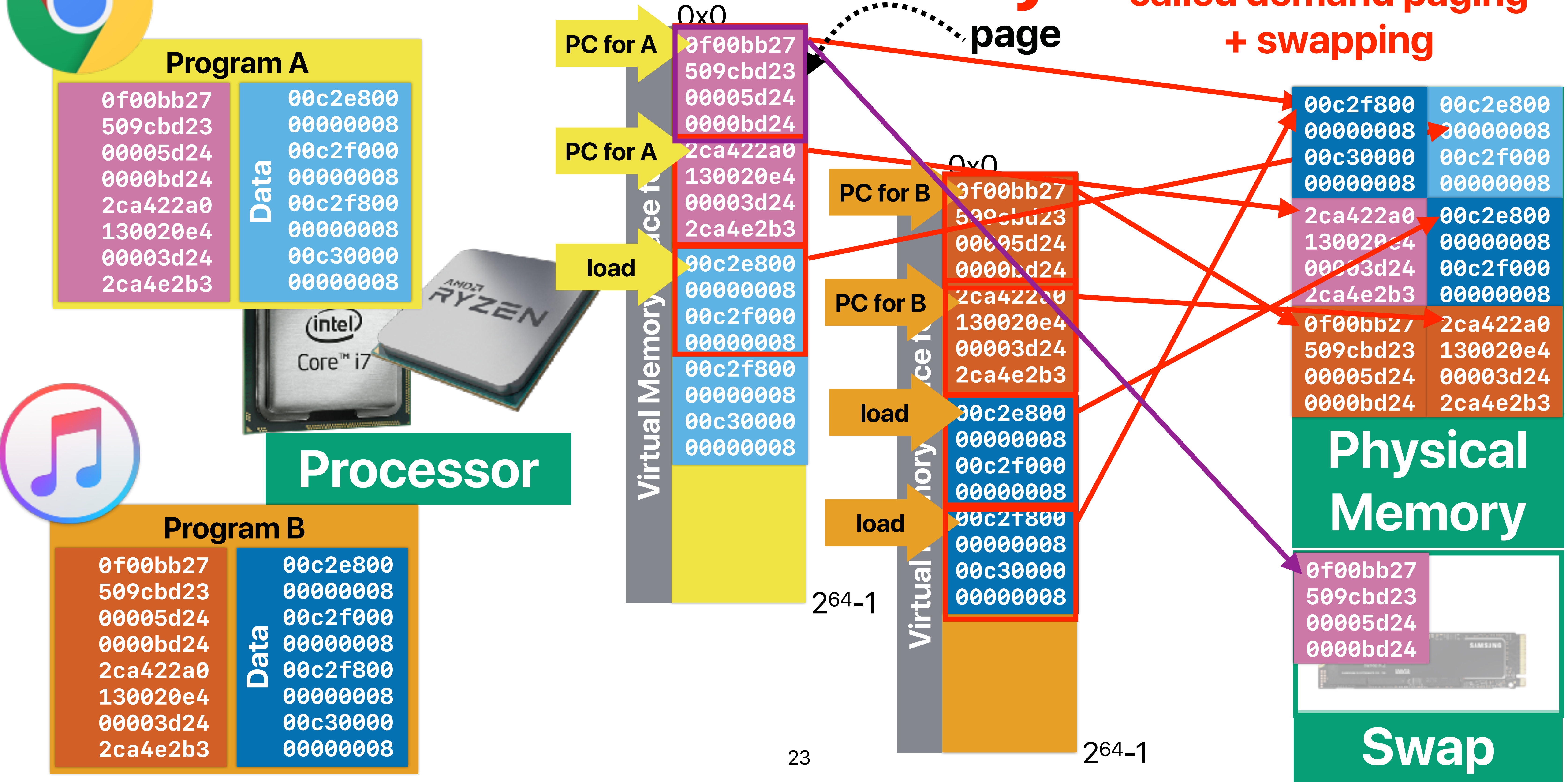
Let’s dig into this code

```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;
}
```

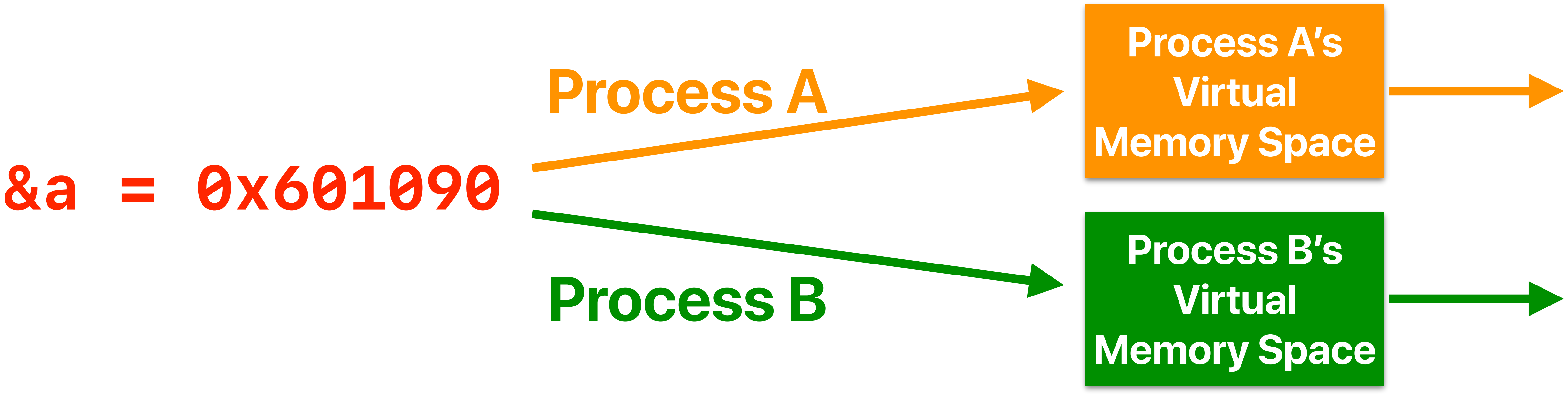
Virtual Memory

Virtual memory

This approach is called demand paging + swapping



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;
}
```

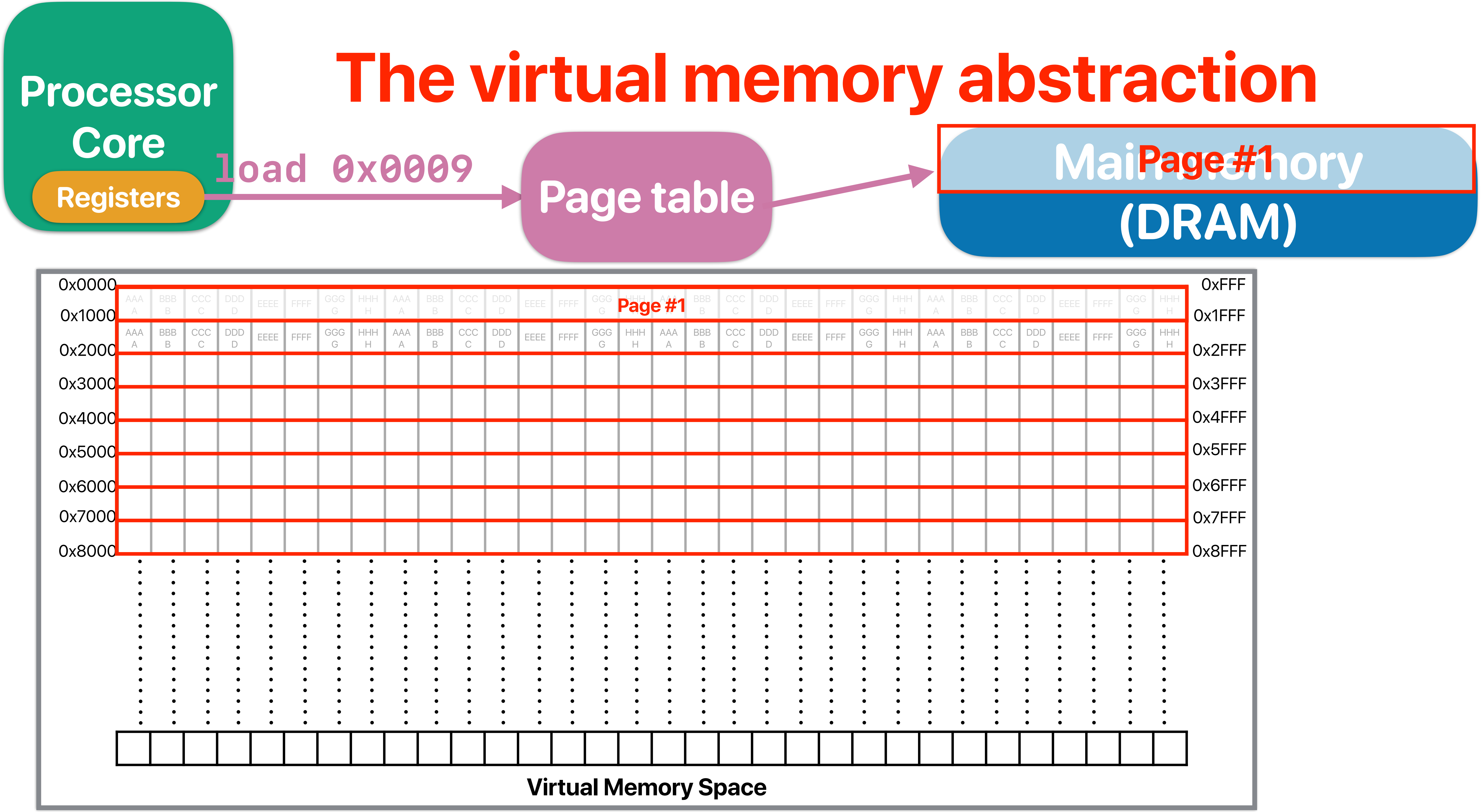
Virtual memory

- 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

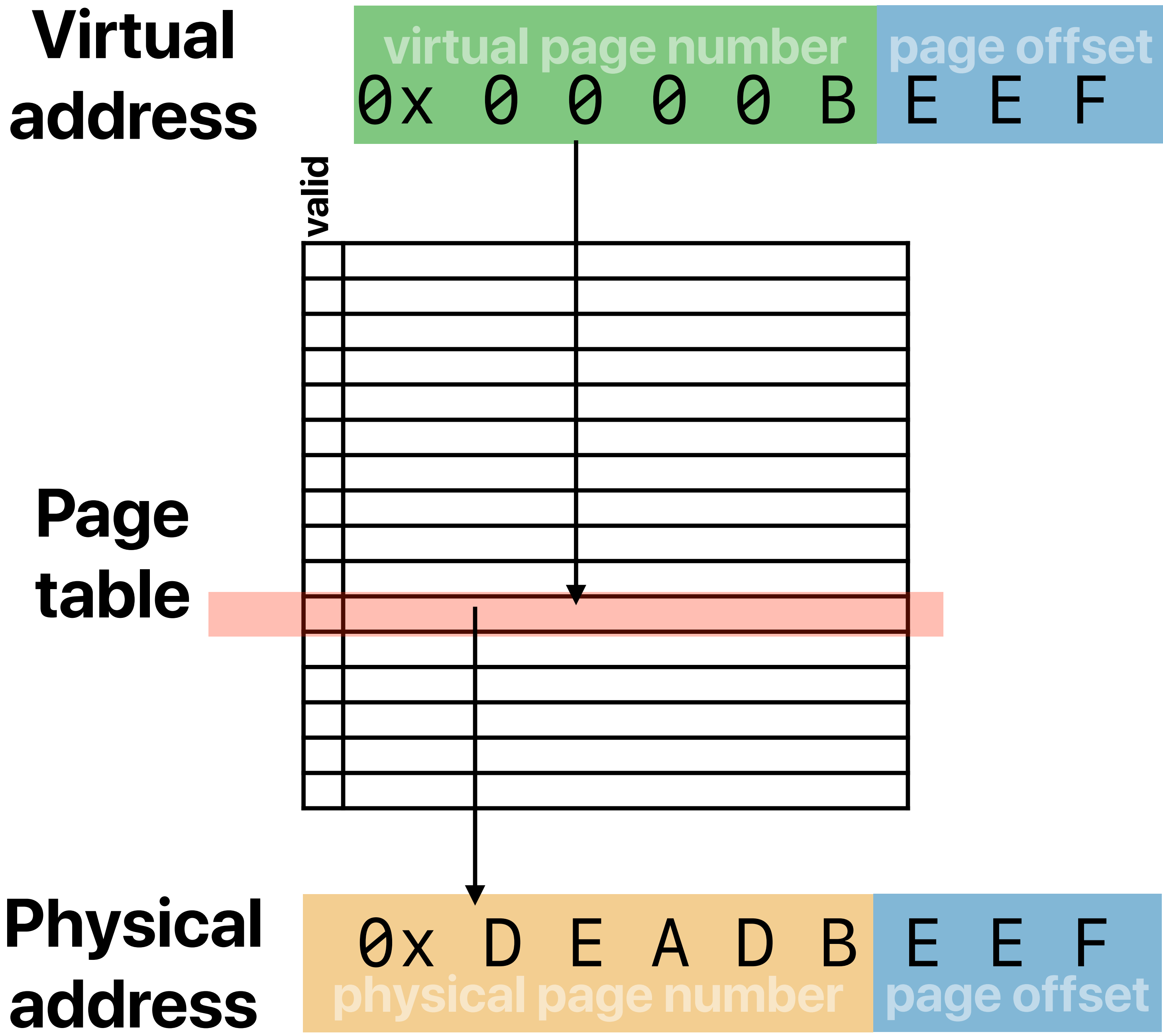


Demand paging

- 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



Conventional page table

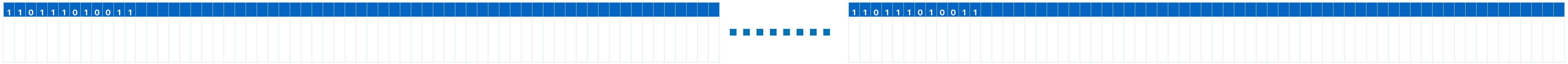
0x0 0xFFFFFFFFFFFFFFFF

— must be consecutive in the physical memory

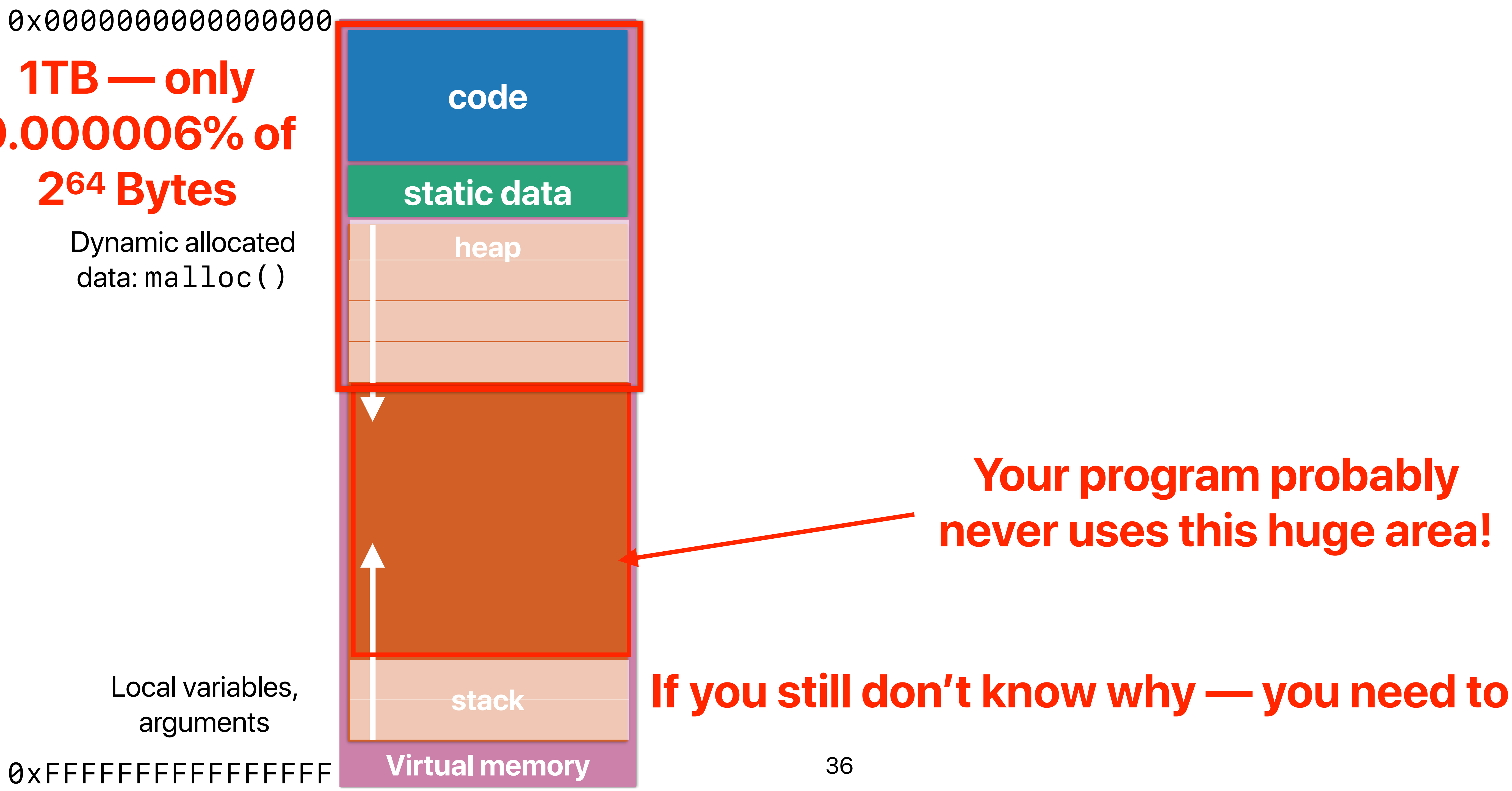
— need a big segment! — difficult to find a spot

— simply too big to fit in memory if address space is large!

$\frac{2^{64}\ B}{2^{12}\ B}$ page table entries/leaf nodes

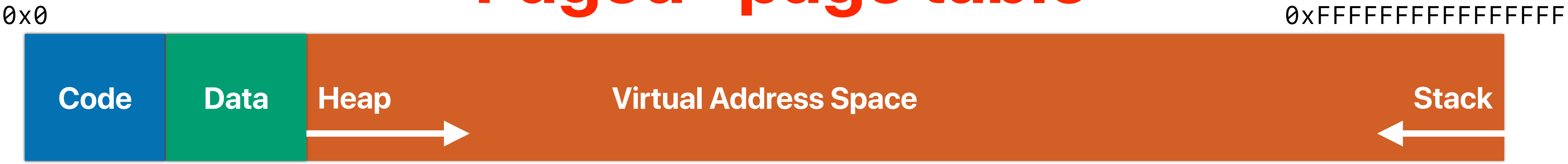


Do we really need a large table?



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

"Paged" page table

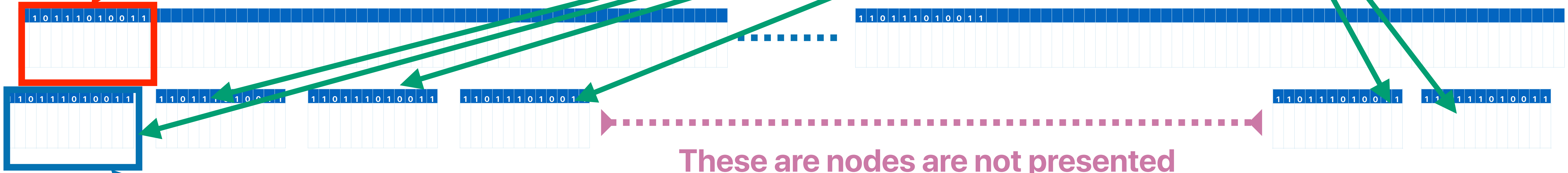


Break up entries into pages!
Each of these occupies exactly a page

— $\frac{2^{12}\ B}{2^{3}\ B} = 2^{9}$ **PTEs per node**

Otherwise, you always need to find more than one consecutive pages — difficult!

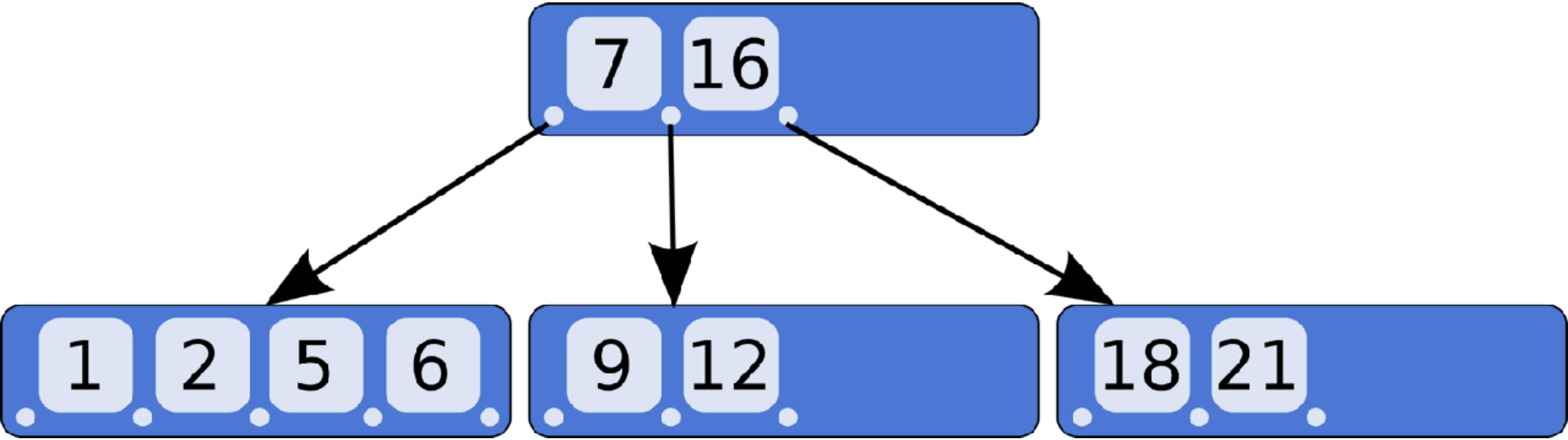
Question:
These nodes are spread out, how to locate them in the memory?



These are nodes are not presented if they are not referenced at all — save space

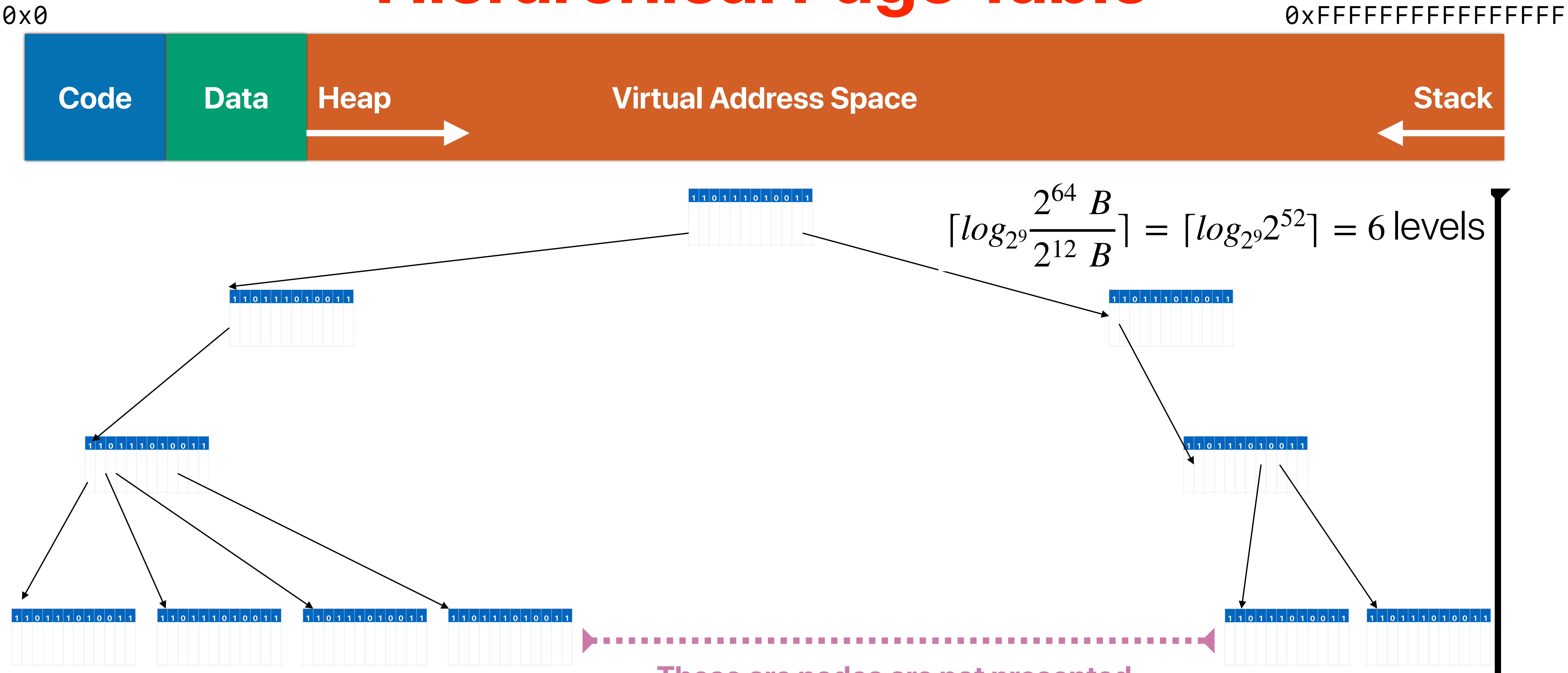
Allocate page table entry nodes "on demand"

B-tree



https://en.wikipedia.org/wiki/B-tree#/media/File:B-tree.svg

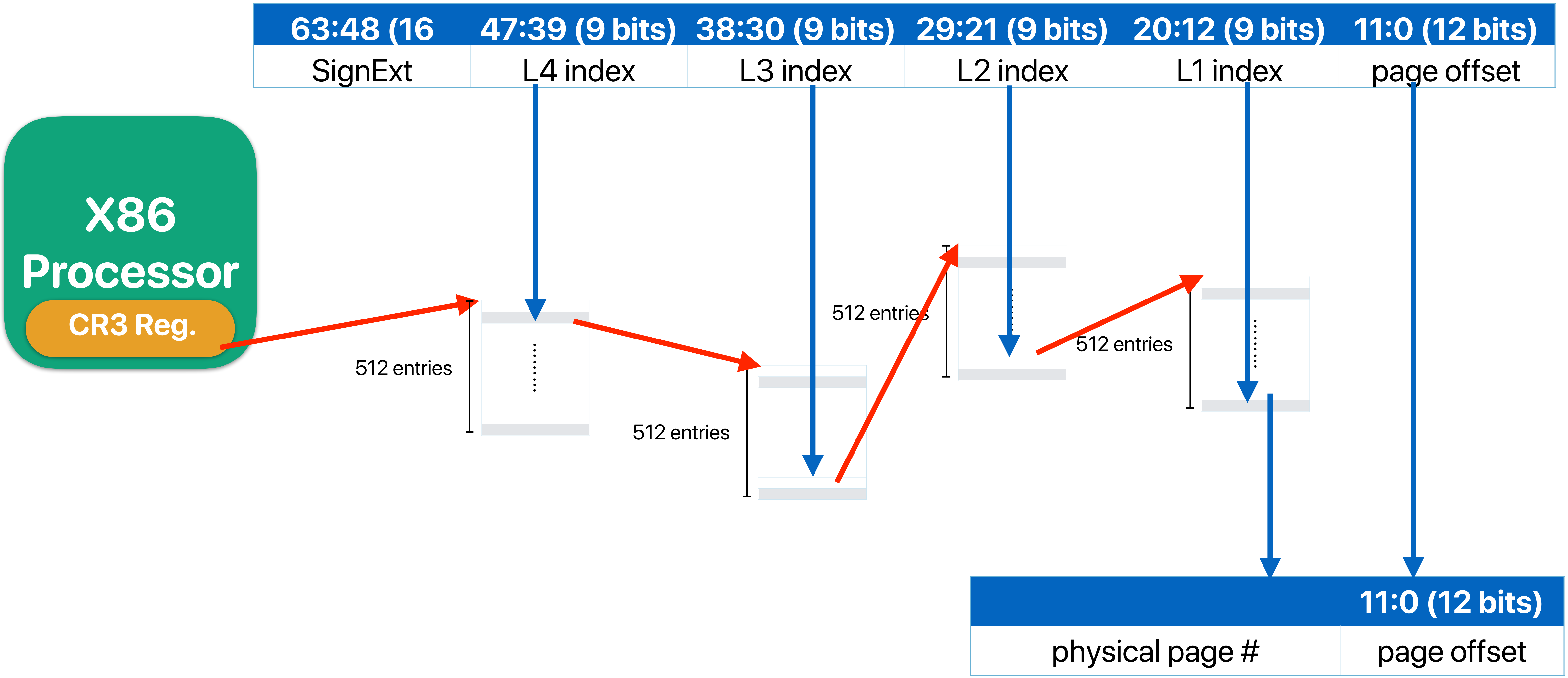
Hierarchical Page Table



$$\lceil log_{2^9}\frac{2^{64}\ B}{2^{12}\ B}\rceil = \lceil log_{2^9}2^{52}\rceil = 6 \text{ levels}$$

$$\frac{2^{64}\ B}{2^{12}\ B}$$ page table entries/leaf nodes (worst case)

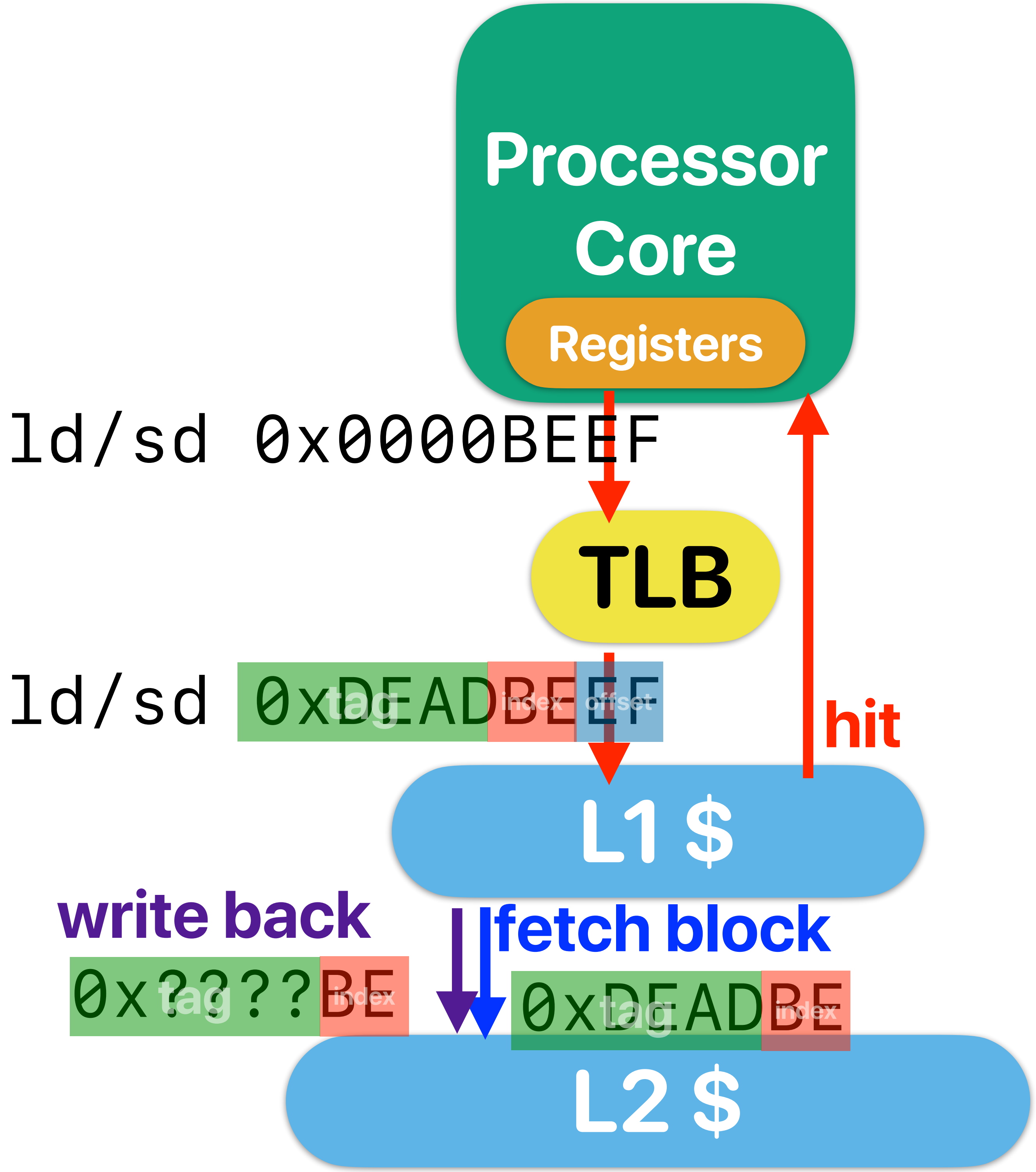
Address translation in x86-64



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

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**

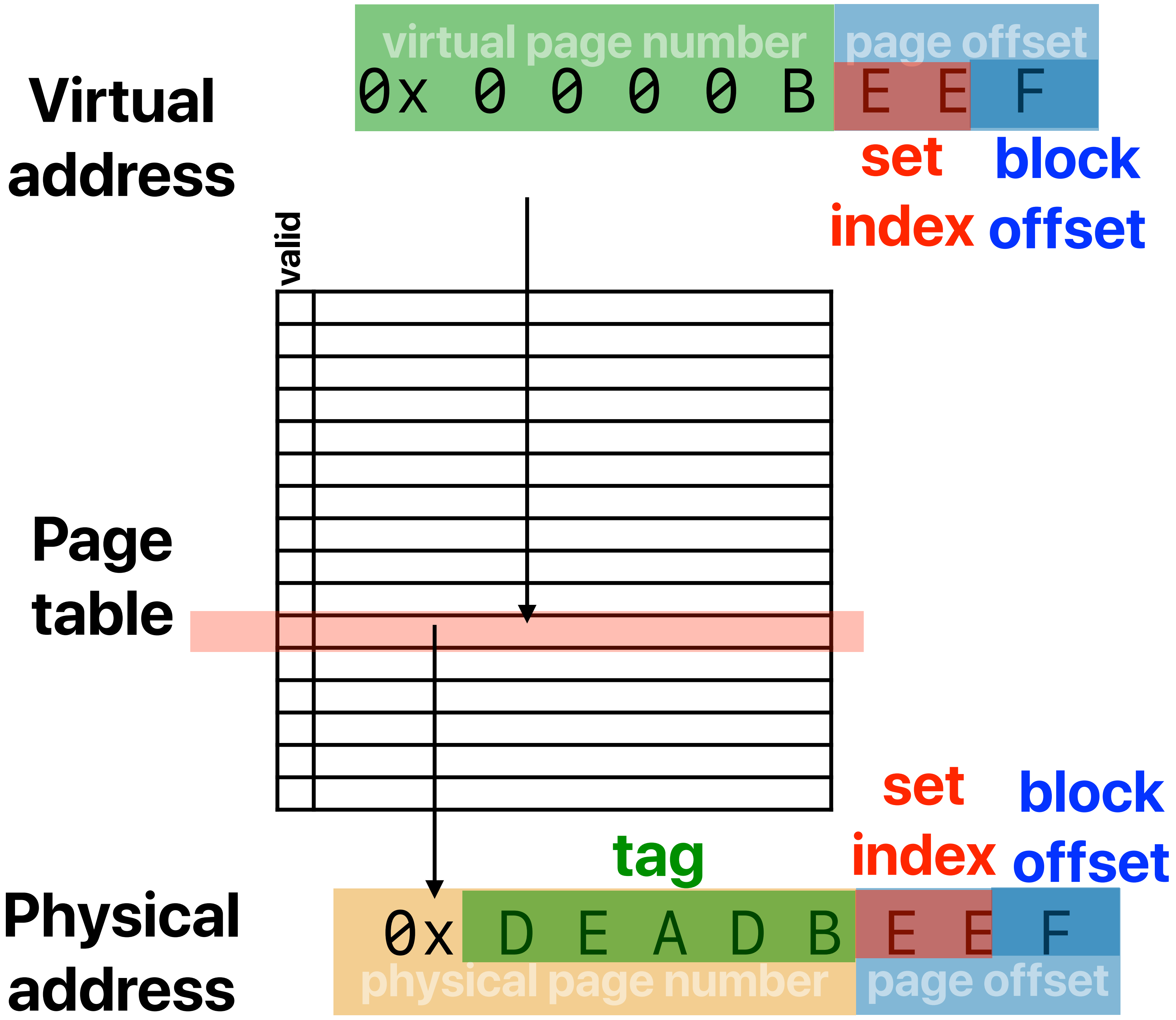
You really need "physical address" to judge if that's what you want

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

memory address: 0x0 8 2 4

memory address: 0b000010000 010 0100

virtual page # | set index | block offset

V	virtual page #	physical page #
1	0x29	0x45
1	0xDE	0x68
1	0x10	0xA1
0	0x8A	0x98

V	D	tag	data
1	1	0x00	AABBCCDDEEGGFFHH
1	1	0x10	IIJJKKLLMMNNOOPP
1	0	0xA1	QQRRSSTTUUVVWWXX
0	1	0x10	YYZZAABBCCDDEEFF
1	1	0x31	AABBCCDDEEGGFFHH
1	1	0x45	IIJJKKLLMMNNOOPP
0	1	0x41	QQRRSSTTUUVVWWXX
0	1	0x68	YYZZAABBCCDDEEFF

0xA 1

=?

hit?

Virtually indexed, physically tagged cache

- If page size is 4KB —

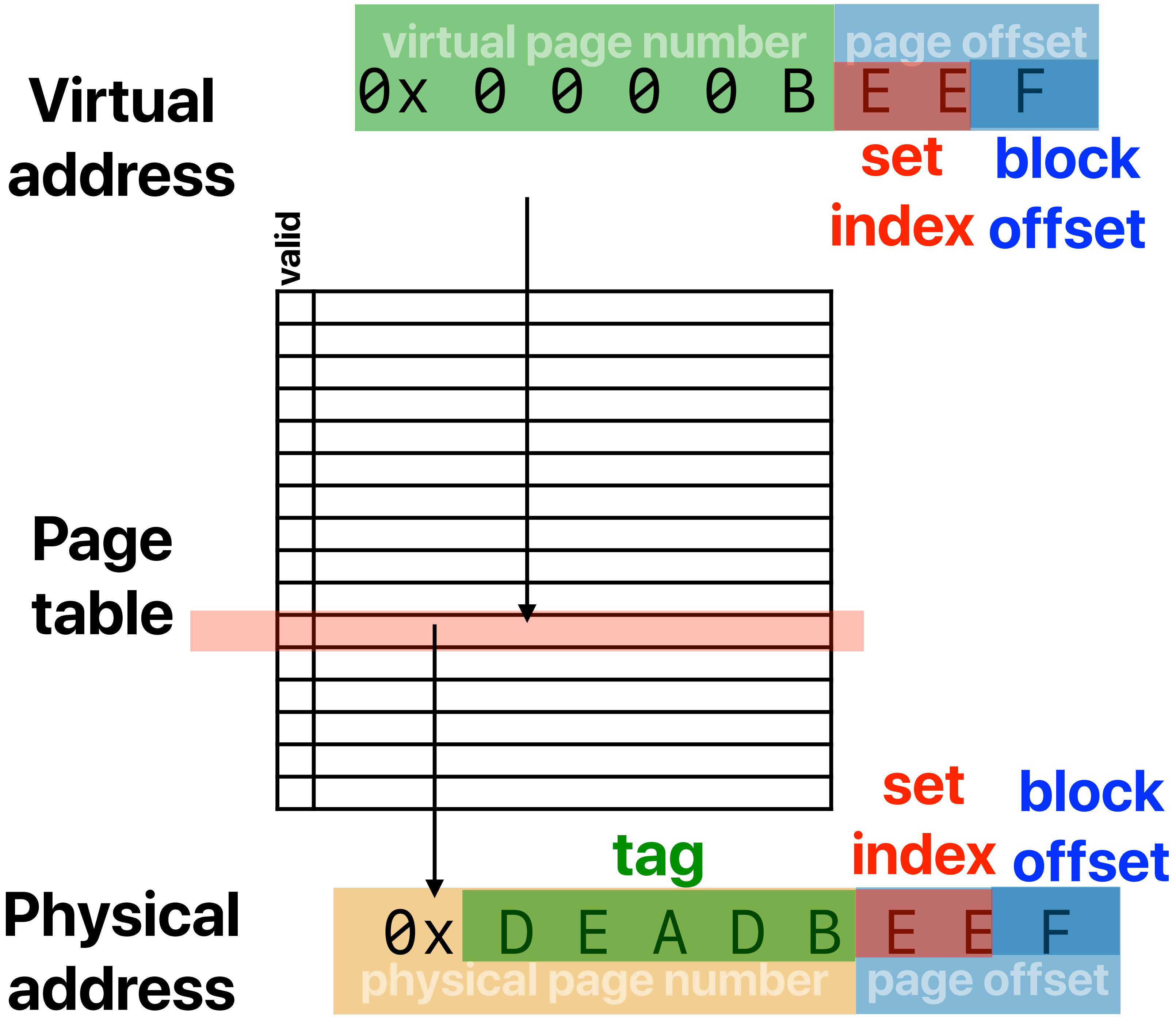
$$lg(B) + lg(S) = lg(4096) = 12$$

$$C = ABS$$

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

$$if\ A = 1$$

$$C = 4KB$$



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

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

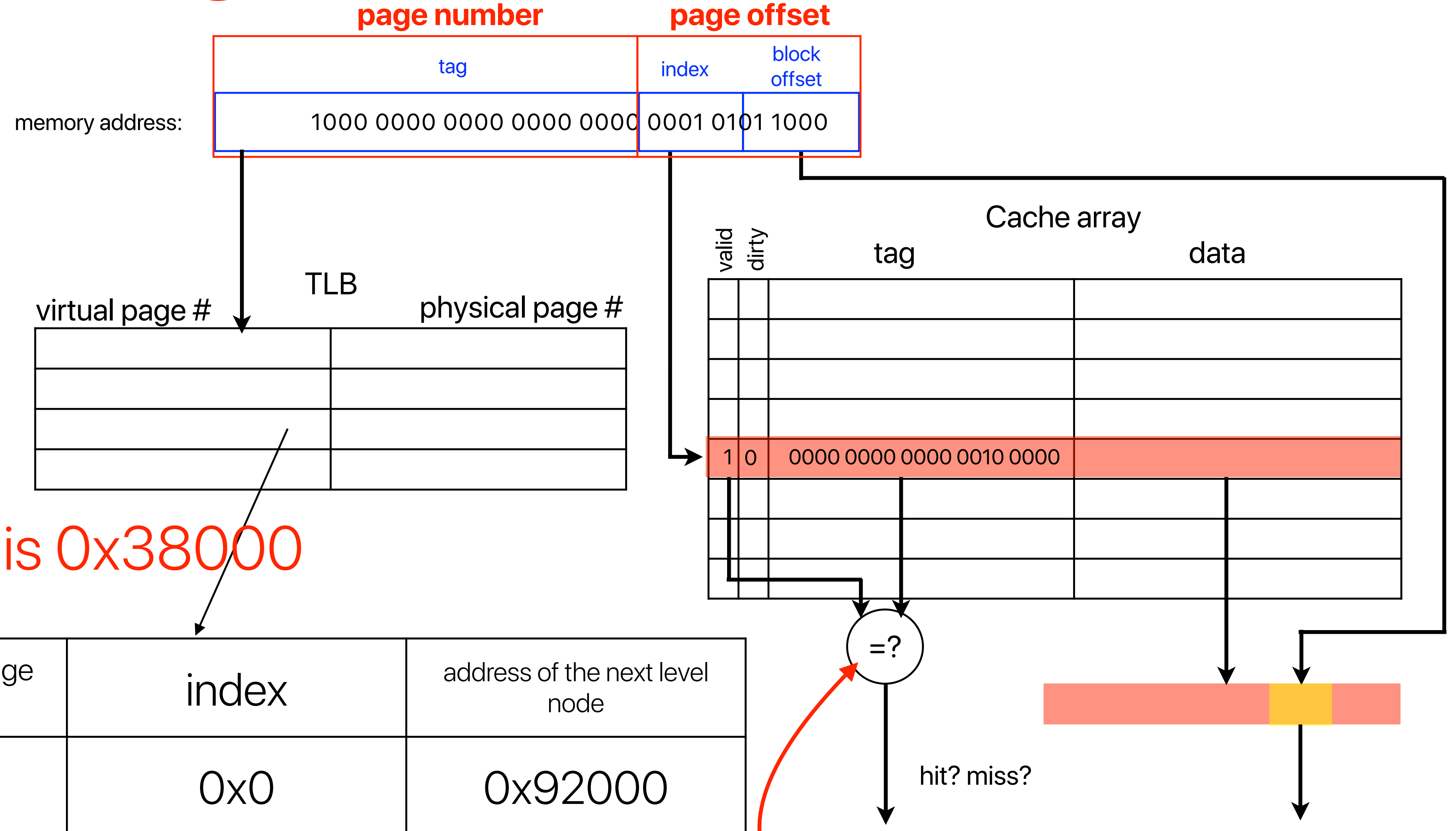
Why should we care about this paper?

- 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

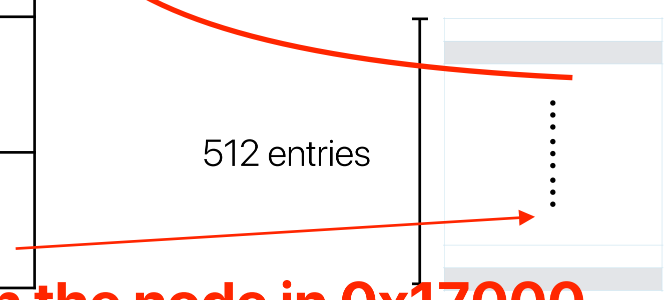
- Not really proposing anything. More an empirical analysis paper
- Design space exploration to find out the optimal solution

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

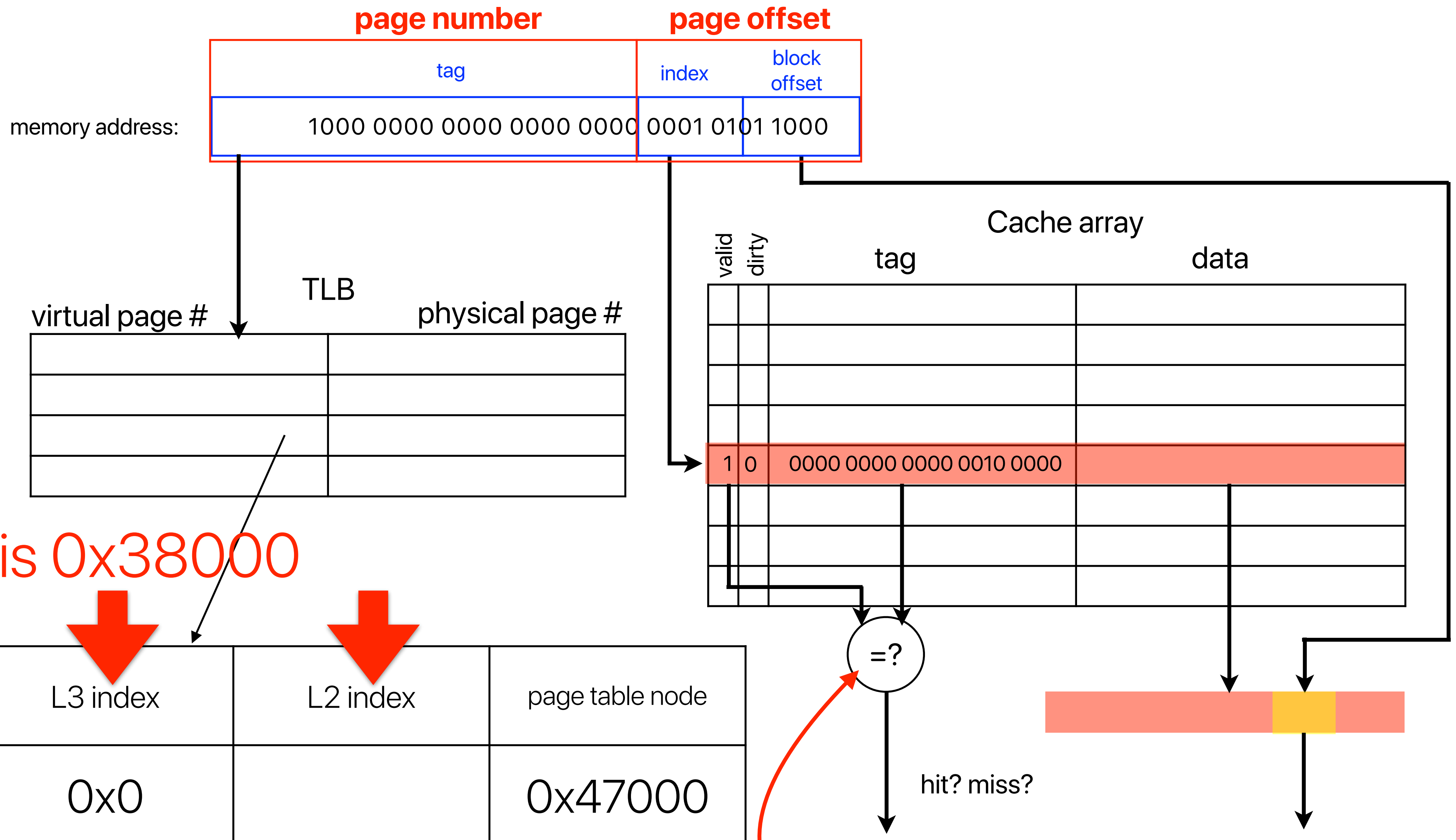


fetch the node in 0x17000

Page table caches

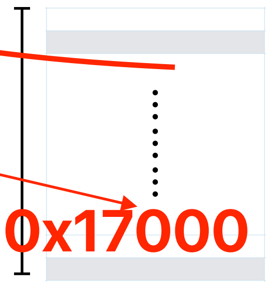
- PTC caches the addresses of “page table nodes”
- PTC uses the physical address of page table nodes as the index
    - Unified page table cache (UPTC)
    - Split page table cache (SPTC)
        - Each page level get a private cache location

Translation cache



assume CR3 is 0x38000

L4 index	L3 index	L2 index	page table node
0x0	0x0		0x47000
0x30	0x23		0x98000
0x0	0x10	0x00	0x17000
0x0	0x10		0x92000



fetch the node in 0x17000

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

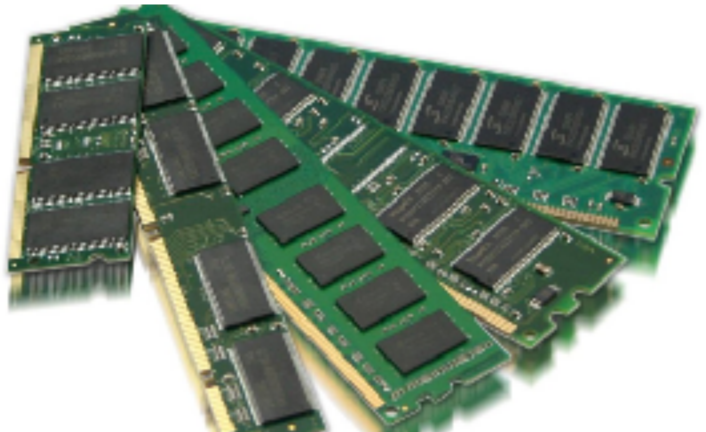
Arkaprava Basu, Jayneel Gandhi, Jichuan Chang, Mark D. Hill and Michael M. Swift

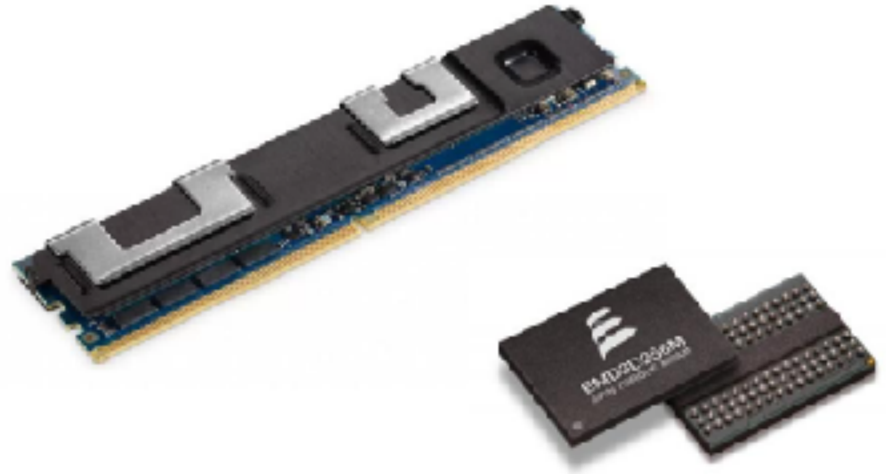
Applications with big memory footprints and high-density memory technologies

MySQL

redis

GRAPH 500





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

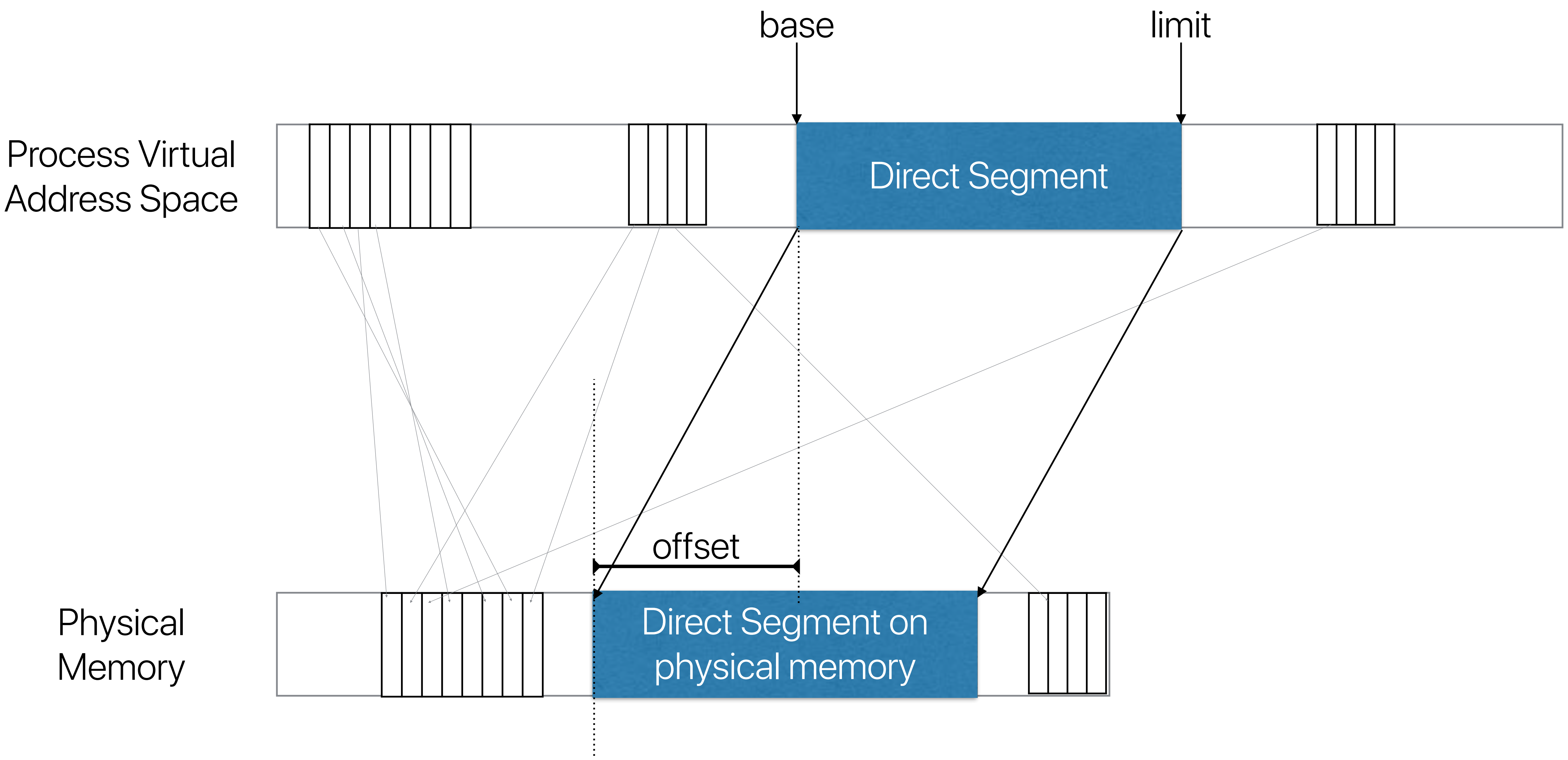
- Few swapping — **since the computer has big memory**
- Few copy-on-write — **since the workload is mostly reads**
- Does not require per-page protection

— **remind yourself how you usually allocate memory?**

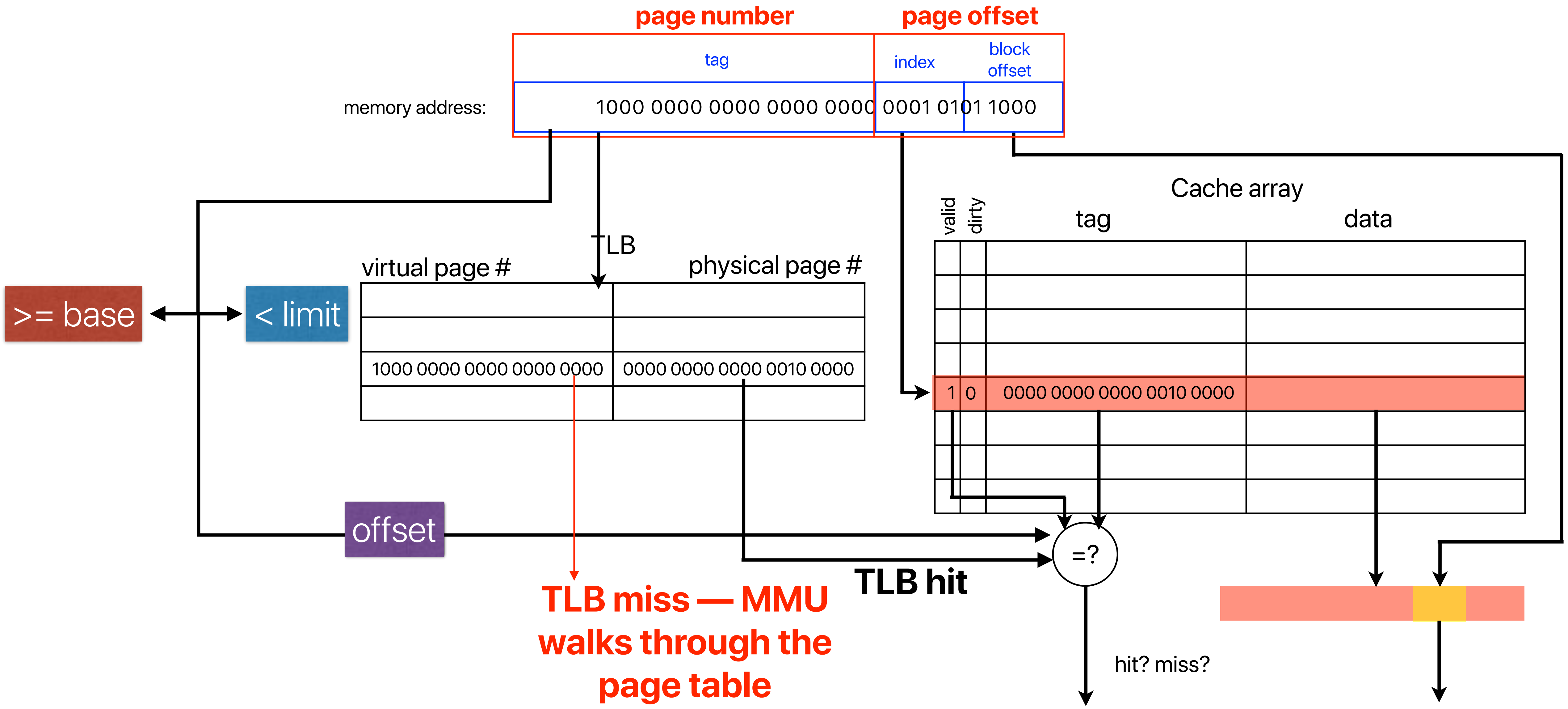
What this paper proposed?

- 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

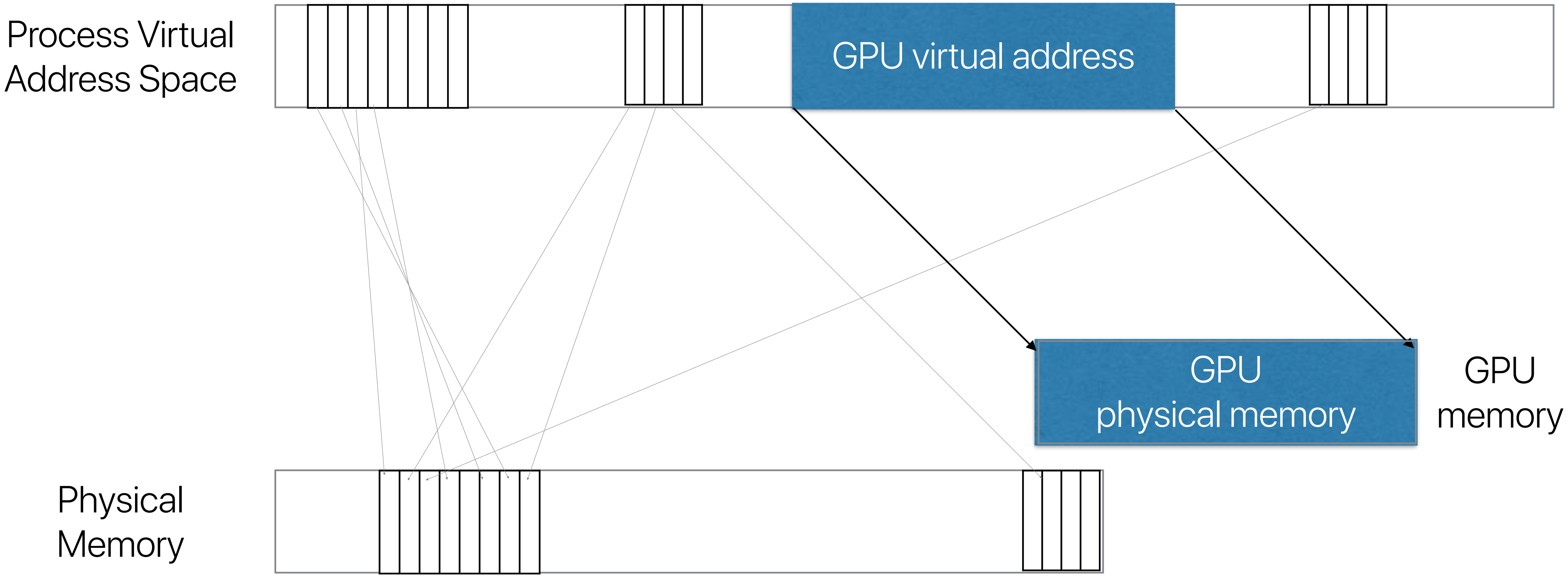
Direct Segment



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