Virtual Memory

CSCI 237: Computer Organization
32nd Lecture, Friday, November 22, 2024

Kelly Shaw

Administrative Details

- Read CSAPP 9.9
- Glow quiz due today at 2:30pm
- Lab #6 due Friday, Dec. 6, at 5pm
- Watch video online before Monday
- Colloquium talk on Friday at 2:35pm in Wege
  - Jeremy Fineman, Georgetown
  - ” Beating Bellman-Ford: Faster Single-Source Shortest Paths with Negative Weights”

Last Time: Virtual Memory

- Address translation (Ch 9.6)
- End-to-end example of a simple memory system

Today

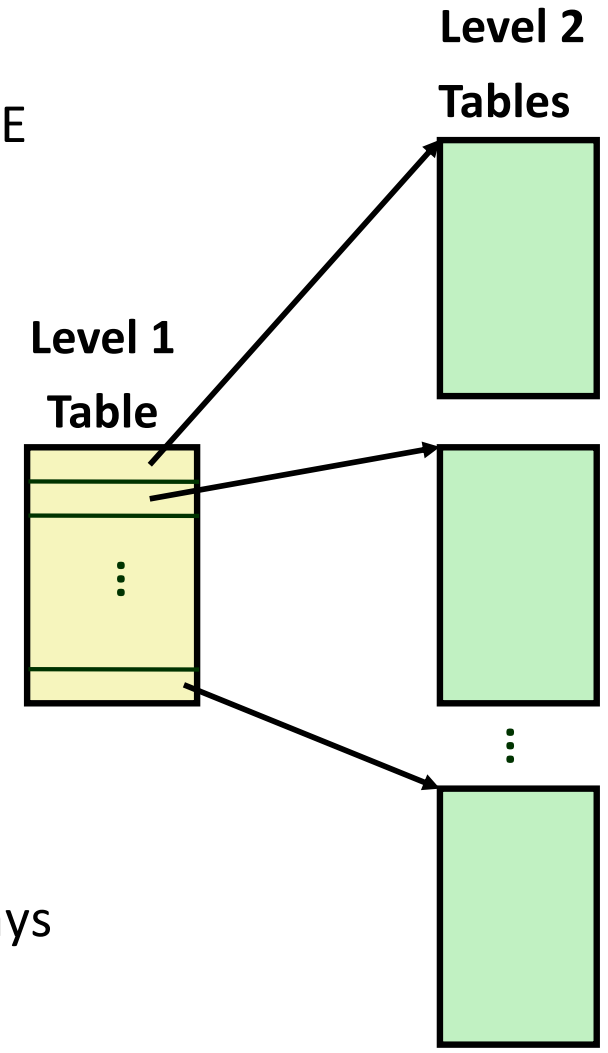
- Address translation (Ch 9.6)
- End-to-end example of a simple memory system
- Dynamic Memory Allocation (Ch 9.9)

Linear Page Table is HUGE and Sparse!

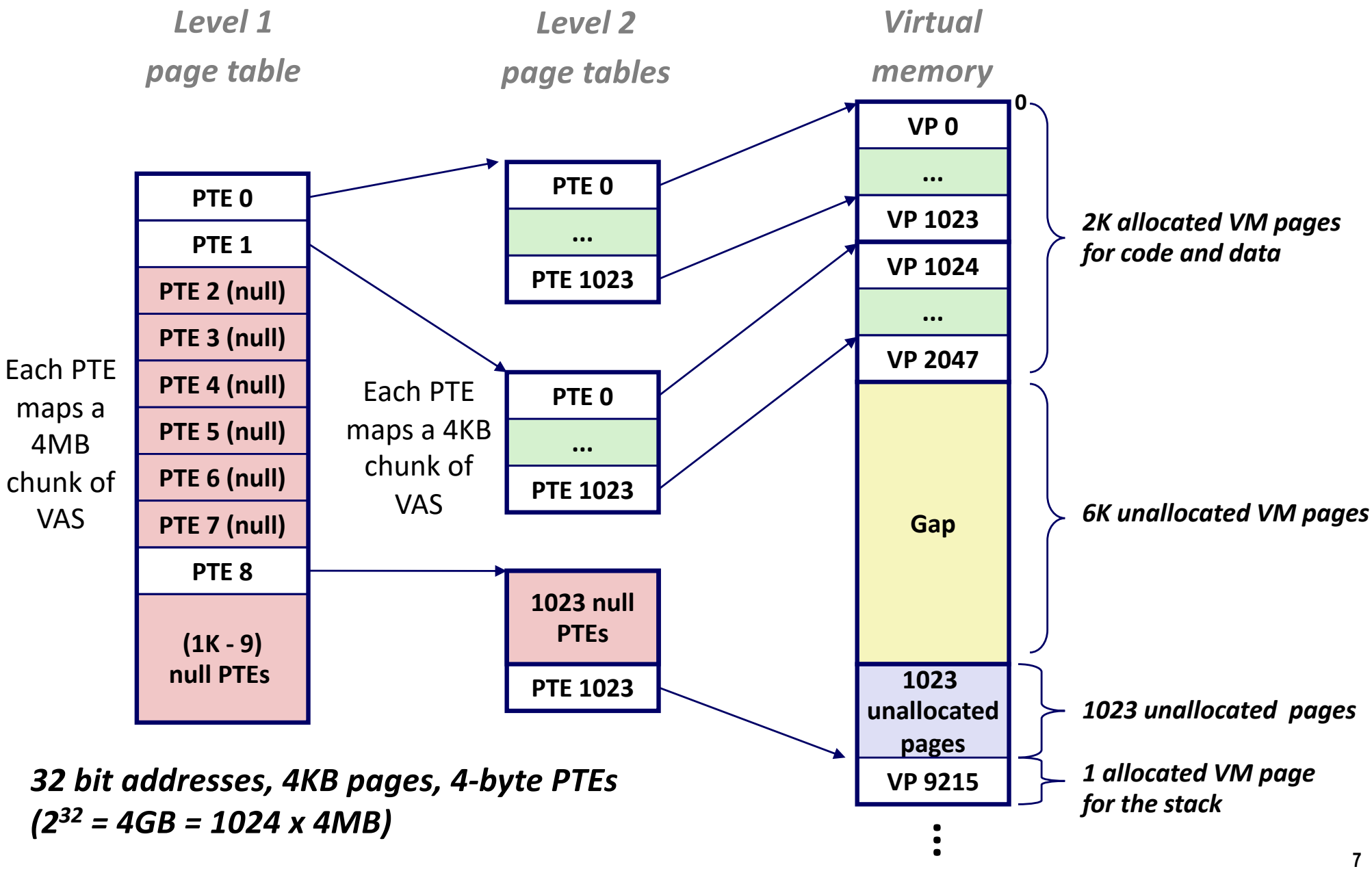
- Linear Page Tables
  - (# of virtual pages) * (size of PTE)
  - Would need to be placed in contiguous physical addresses in DRAM
  - But much of virtual address space is unallocated
    - E.g.) area from top of stack to top of heap
- Use a hierarchical data structure / tree instead
  - We don't have to create PTEs for unallocated virtual pages
  - We can have single top level page (root) in DRAM, with other pages being demand paged as needed

Multi-Level Page Tables: Concrete Example

- Suppose:
  - 4KB (2^{12}) page size, 48-bit address space, 8-byte PTE
- Problem?
  - Would need a 512 GB page table!
    - $2^{48} / 2^{12} = 2^{36}$ = # entries in page table
    - 2^3 bytes per entry
    - $2^{48} * 2^{-12} * 2^3 = 2^{39}$ bytes in every page table
- Common solution: Multi-level page tables
  - No need to waste space for unallocated pages!
- 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)



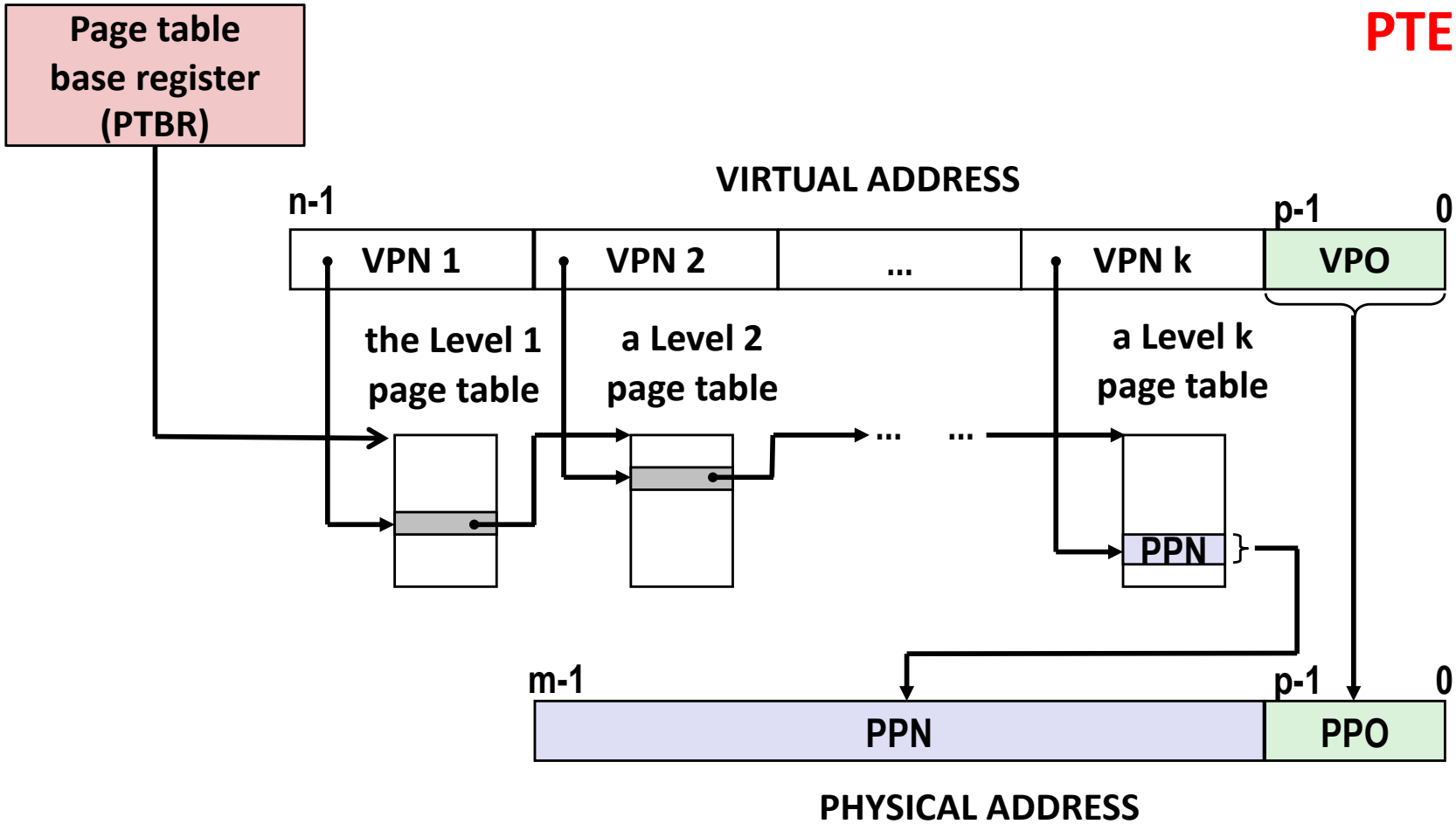
A Two-Level Page Table Hierarchy



32 bit addresses, 4KB pages, 4-byte PTEs
(2^{32} = 4GB = 1024 x 4MB)

Translating with a k-level Page Table

TLB caches PTEs from all levels



Address Translation Summary (Ch 9.6)

- Programmer's view of virtual memory:
  - Each process has its own private linear address space
  - Cannot be corrupted by other processes

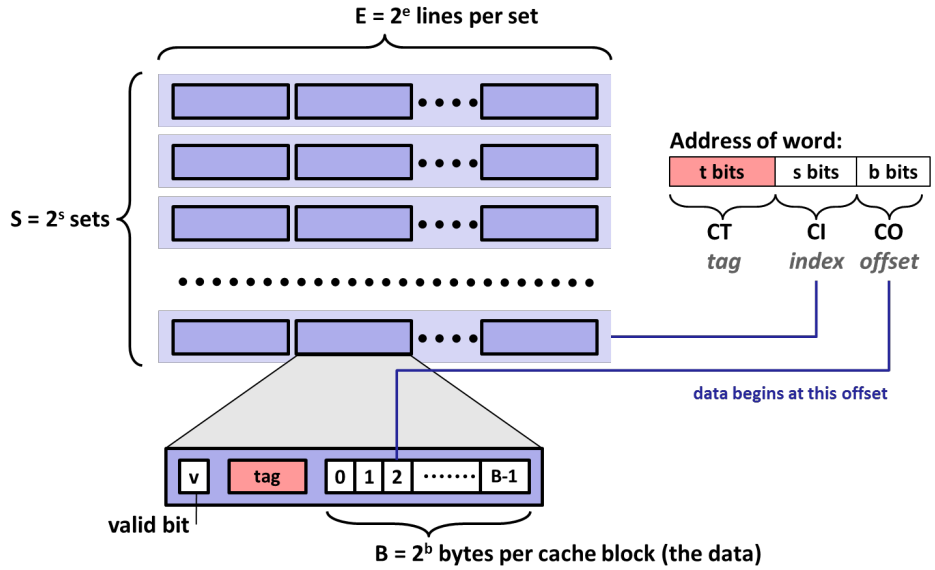
- System's view of virtual memory:
  - Uses memory efficiently by caching virtual memory pages
    - Efficient only because of locality
  - Simplifies memory management and programming
  - Simplifies protection by providing a convenient inter-positioning point to check permissions

Today

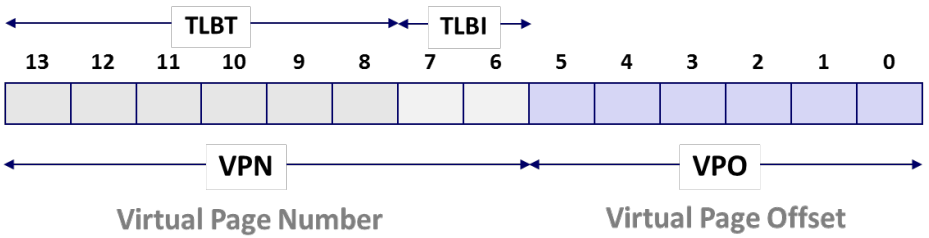
- Address translation (Ch 9.6)
- End-to-end example of a simple memory system
- Dynamic Memory Allocation (Ch 9.9)

Review of Symbols

- Basic Parameters
  - **N = 2^n** : Number of addresses in virtual address space
  - **M = 2^m** : Number of addresses in physical address space
  - **P = 2^p** : Page size (bytes)

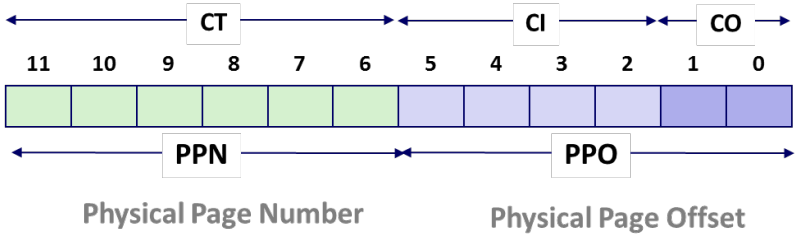


- Components of the *virtual address* (VA)
  - **TLBI**: TLB index
  - **TLBT**: TLB tag
  - **VPO**: Virtual page offset
  - **VPN**: Virtual page number



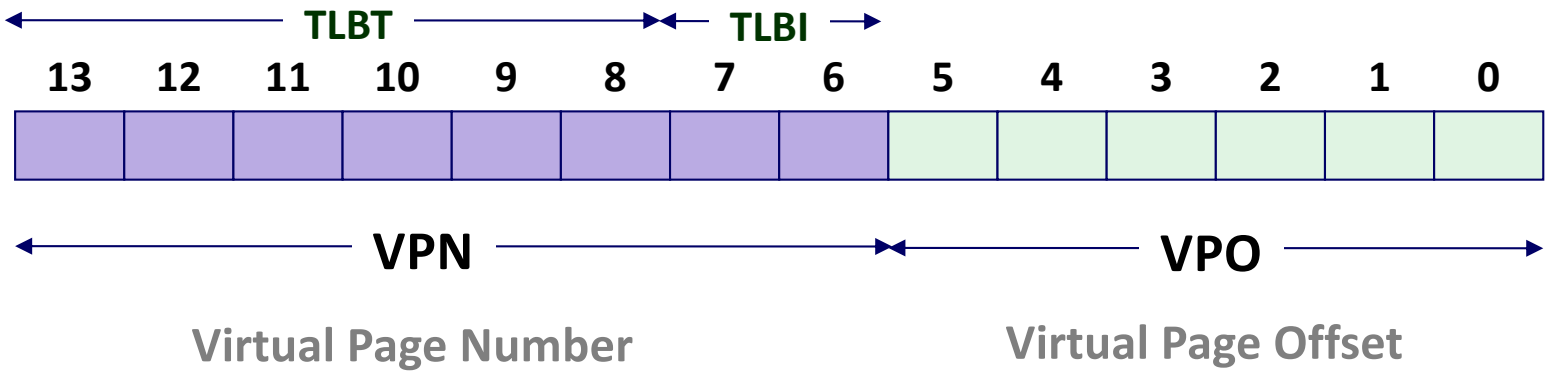
- Components of the *physical address* (PA)
  - **PPO**: Physical page offset (same as VPO)
  - **PPN:** Physical page number
  - **CO**: Byte offset within cache line
  - **CI:** Cache index
  - **CT**: Cache tag

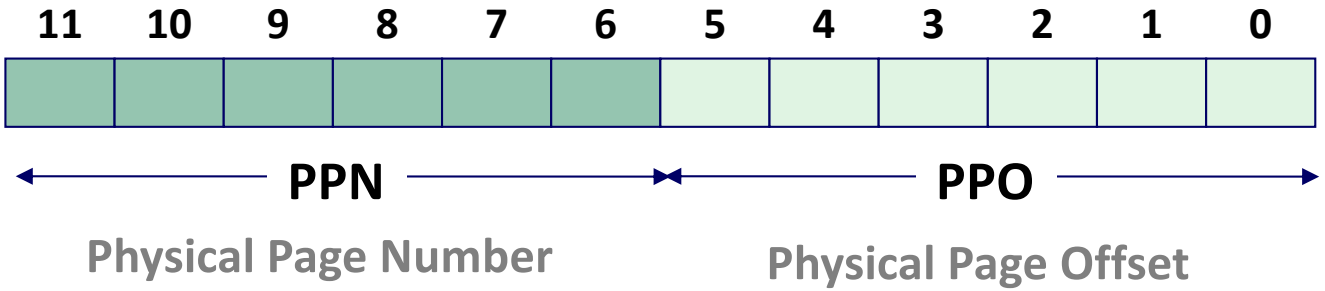
(bits per field for our simple example)



Simple Memory System Example

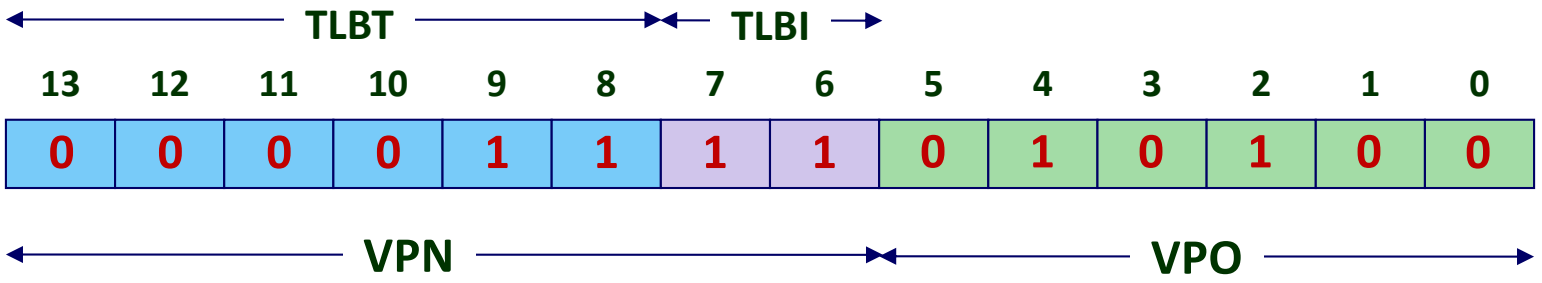
- Addressing Parameters
  - Page size = 64 bytes
  - 14-bit virtual addresses
  - 12-bit physical address
- TLB Parameters
  - 4 Sets
  - 4-way





Address Translation Example

Virtual Address: 0x03D4



VPN 0x0F TLBI 0x3 TLBT 0x03 TLB Hit? Y Page Fault? N PPN: 0x0D

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

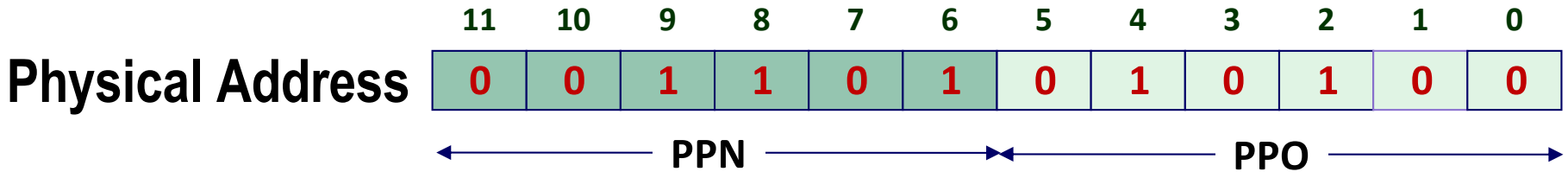
Address Translation Example

Cache

Idx	Tag	Valid	B0	B1	B2	B3
0	19	1	99	11	23	11
1	15	0	–	–	–	–
2	1B	1	00	02	04	08
3	36	0	–	–	–	–
4	32	1	43	6D	8F	09
5	0D	1	36	72	F0	1D
6	31	0	–	–	–	–
7	16	1	11	C2	DF	03

Idx	Tag	Valid	B0	B1	B2	B3
8	24	1	3A	00	51	89
9	2D	0	–	–	–	–
A	2D	1	93	15	DA	3B
B	0B	0	–	–	–	–
C	12	0	–	–	–	–
D	16	1	04	96	34	15
E	13	1	83	77	1B	D3
F	14	0	–	–	–	–

- 16 lines, 4-byte block size, direct mapped
- Physically addressed



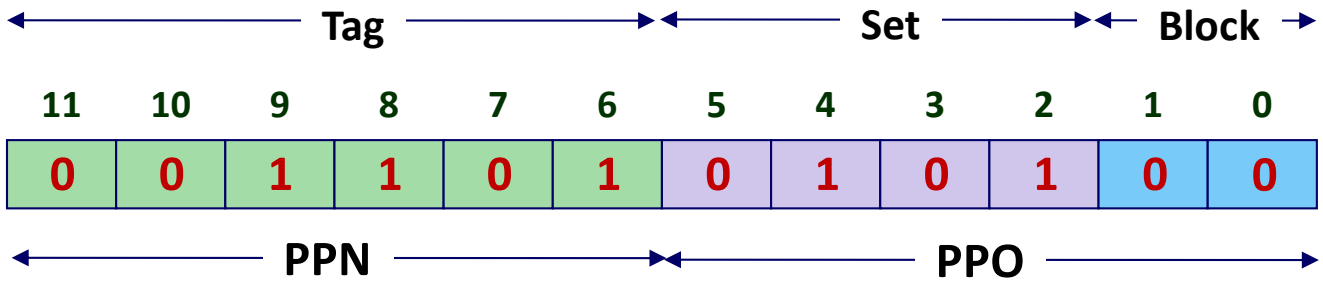
Address Translation Example

Cache

Idx	*Tag*	*Valid*	*B0*	*B1*	*B2*	*B3*
0	19	1	99	11	23	11
1	15	0	–	–	–	–
2	1B	1	00	02	04	08
3	36	0	–	–	–	–
4	32	1	43	6D	8F	09
5	0D	1	36	72	F0	1D
6	31	0	–	–	–	–
7	16	1	11	C2	DF	03

Idx	*Tag*	*Valid*	*B0*	*B1*	*B2*	*B3*
8	24	1	3A	00	51	89
9	2D	0	–	–	–	–
A	2D	1	93	15	DA	3B
B	0B	0	–	–	–	–
C	12	0	–	–	–	–
D	16	1	04	96	34	15
E	13	1	83	77	1B	D3
F	14	0	–	–	–	–

Physical Address



Block ___0___ Set___0x5___ Tag ___0x0D___ Hit? __ Value: ____

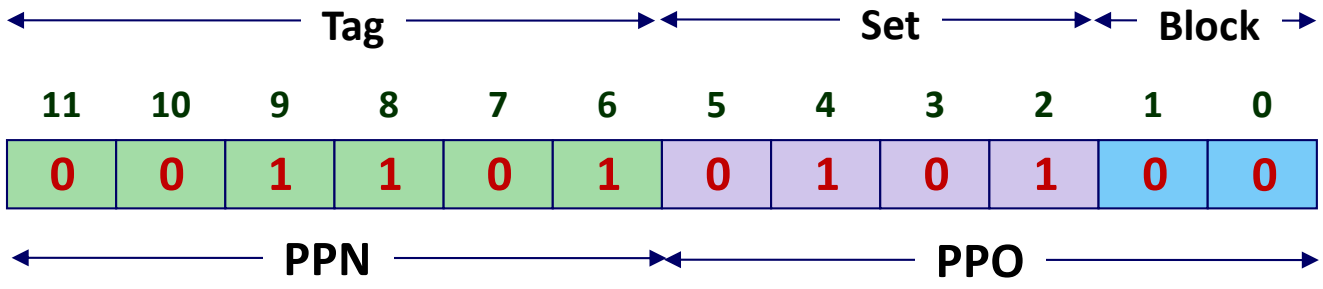
Address Translation Example

Cache

Idx	Tag	Valid	B0	B1	B2	B3
0	19	1	99	11	23	11
1	15	0	–	–	–	–
2	1B	1	00	02	04	08
3	36	0	–	–	–	–
4	32	1	43	6D	8F	09
5	0D	1	36	72	F0	1D
6	31	0	–	–	–	–
7	16	1	11	C2	DF	03

Idx	Tag	Valid	B0	B1	B2	B3
8	24	1	3A	00	51	89
9	2D	0	–	–	–	–
A	2D	1	93	15	DA	3B
B	0B	0	–	–	–	–
C	12	0	–	–	–	–
D	16	1	04	96	34	15
E	13	1	83	77	1B	D3
F	14	0	–	–	–	–

Physical Address



Block ___0___ Set__0x5__ Tag __0x0D__ Hit? __Y__ Value: ____

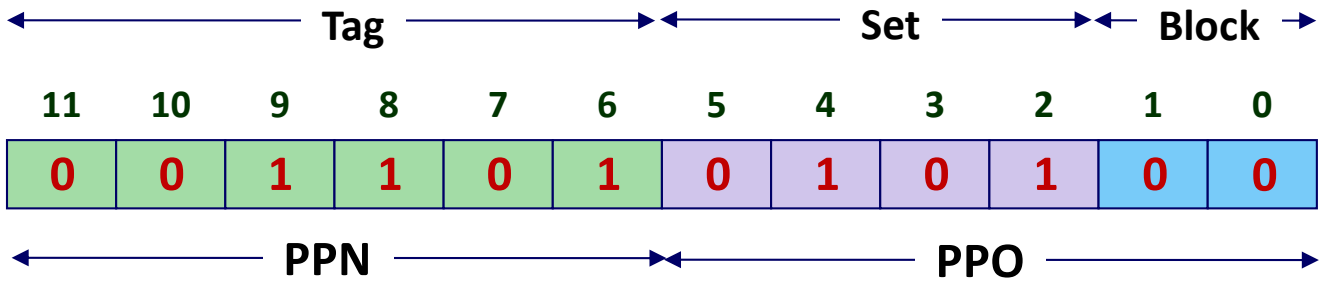
Address Translation Example

Cache

Idx	Tag	Valid	B0	B1	B2	B3
0	19	1	99	11	23	11
1	15	0	–	–	–	–
2	1B	1	00	02	04	08
3	36	0	–	–	–	–
4	32	1	43	6D	8F	09
5	0D	1	36	72	F0	1D
6	31	0	–	–	–	–
7	16	1	11	C2	DF	03

Idx	Tag	Valid	B0	B1	B2	B3
8	24	1	3A	00	51	89
9	2D	0	–	–	–	–
A	2D	1	93	15	DA	3B
B	0B	0	–	–	–	–
C	12	0	–	–	–	–
D	16	1	04	96	34	15
E	13	1	83	77	1B	D3
F	14	0	–	–	–	–

Physical Address



Block 0 Set 0x5 Tag 0x0D Hit? Y Value: 0x36

Practice on Your Own

The following problem concerns the way virtual addresses are translated into physical addresses.

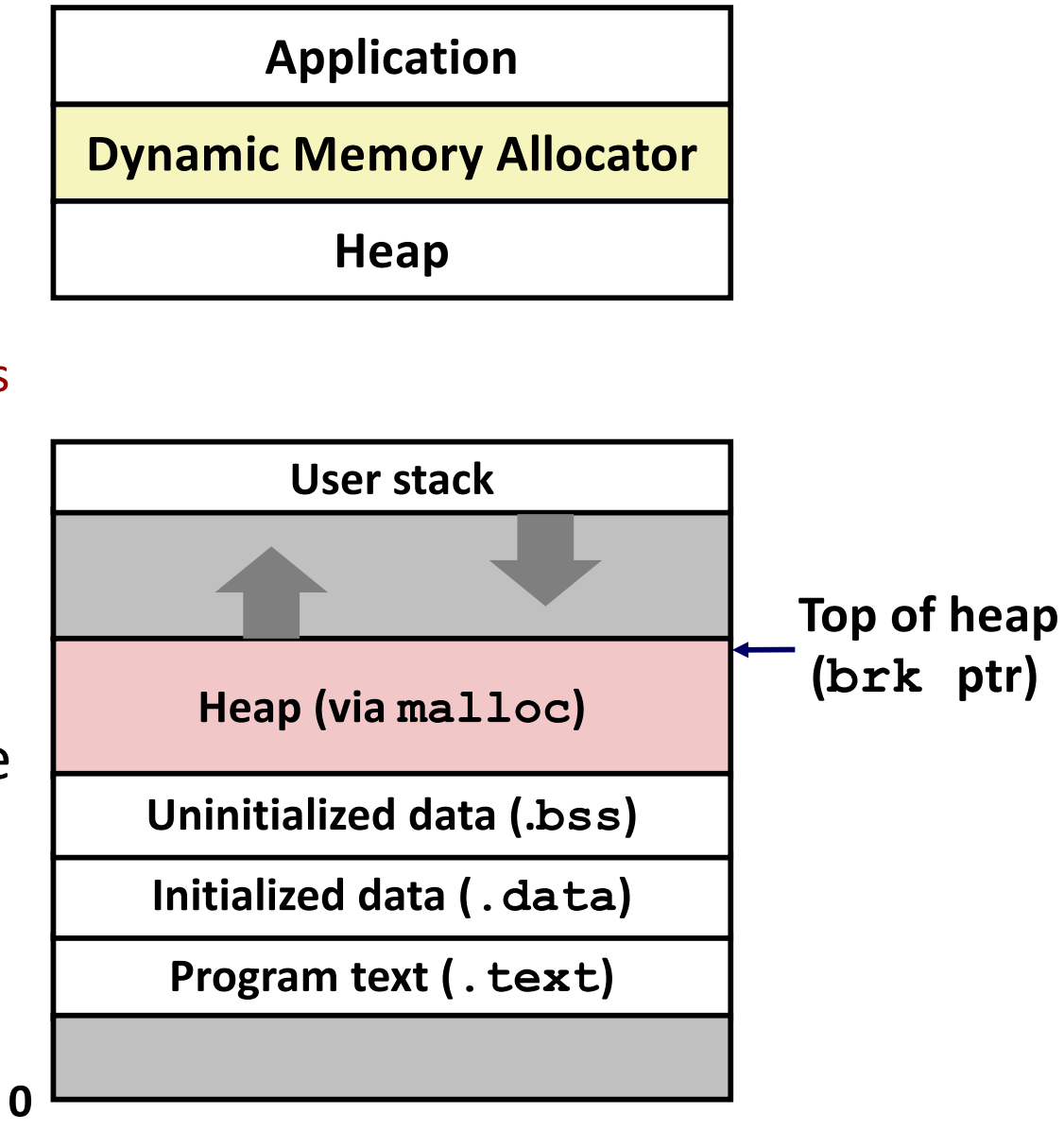
- The memory is byte addressable.
- Memory accesses are to 1-byte words (not 4-byte words).
- Virtual addresses are 16 bits wide.
- Physical addresses are 14 bits wide.
- The page size is 1024 bytes = 2^{10}
- The TLB is 4-way set associative with 16 total entries.
- For the virtual address 0x76BD, what is the
  - VPN
  - VPO
  - TLB Index
  - TLB Tag

Today

- Address translation (Ch 9.6)
- End-to-end example of a simple memory system
- Dynamic Memory Allocation (Ch 9.9)

Dynamic Memory Allocation

- Programmers use *dynamic memory allocators* (such as `malloc`) to acquire VM at run time.
  - For data structures whose size is only known at runtime.
- Dynamic memory allocators manage an area of process virtual memory known as the *heap*.



Dynamic Memory Allocation

- Allocator maintains heap as collection of variable sized *blocks*, which are either *allocated* or *free.*
- Types of allocators
  - ***Explicit allocator*:** application allocates and frees space
    - e.g., **`malloc`** and **`free`** in C
  - ***Implicit allocator:*** application allocates, but does not free space
    - e.g., garbage collection in Java, ML, and Lisp

- Will discuss simple explicit memory allocation today

The `malloc` Package

```
#include <stdlib.h>

void *malloc(size_t size)
```

- Successful:
  - Returns a pointer to a memory block of at least **`size`** bytes aligned to an 16-byte boundary (on x86-64)
  - If **`size == 0`**, returns NULL
- Unsuccessful: returns NULL (0) and sets **`errno`**

```
void free(void *p)
```

- Returns the block pointed at by **`p`** to pool of available memory
- **`p`** must come from a previous call to **`malloc`** or **`realloc`**

Other functions

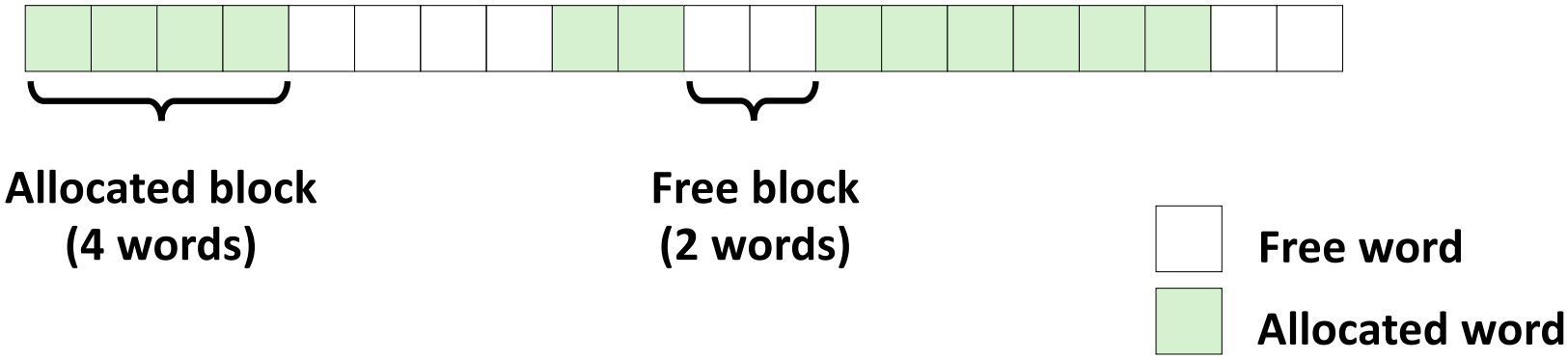
- **`calloc`:** Version of **`malloc`** that initializes allocated block to zero.
- **`realloc`:** Changes the size of a previously allocated block.
- **`sbrk`:** Used internally by allocators to grow or shrink the heap

`malloc` Example

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

void foo(int n) {
    int i, *p;

    /* Allocate a block of n ints */
    p = (int *) malloc(n * sizeof(int));
    if (p == NULL) {
        perror("malloc");
        exit(0);
    }

    /* Initialize allocated block */
    for (i = 0; i < n; i++)
        p[i] = i;


    /* Return allocated block to the heap */
    free(p);
}
```

Simplifying Assumptions Made in This Lecture

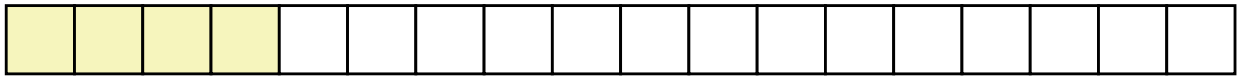
- Memory is word addressed.
- Words are int-sized (4 bytes).
  - Each box is a 4 byte word.
- Allocations are double-word (8 byte) aligned.



Allocation Example

```
#define SIZ sizeof(int)
```

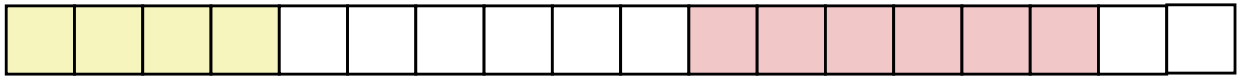
```
p1 = malloc(4*SIZ)
```



```
p2 = malloc(5*SIZ)
```

```
p3 = malloc(6*SIZ)
```

```
free(p2)
```



```
p4 = malloc(2*SIZ)
```

Constraints

- Applications
  - Can issue arbitrary sequence of **`malloc`** and **`free`** requests
  - **`free`** request must be to a **`malloc`**'d block

- Allocators
  - Can't control number or size of allocated blocks
  - Must respond immediately to **`malloc`** requests
    - *i.e.*, can't reorder or buffer requests
  - Must allocate blocks from free memory
    - *i.e.*, can only place allocated blocks in free memory
  - Must align blocks so they satisfy all alignment requirements
    - 16-byte (x86-64) alignment on Linux machines
  - Can manipulate and modify only free memory
  - Can't move the allocated blocks once they are **`malloc`**'d
    - *i.e.*, compaction is not allowed

Evaluating a Memory Allocator

- What does it mean for an allocator to be good?
  - How do we measure the “performance” of an allocator?
  - How do we measure “quality” of the allocations?

- As we talk about designs, think about the best and worst cases

Performance Goals: Throughput

- Given some sequence of `malloc` and `free` requests:
  - $R_0, R_1, ..., R_k, ..., R_{n-1}$

- Throughput:
  - Number of completed requests per unit time
  - Example:
    - 5,000 **`malloc`** calls and 5,000 **`free`** calls in 10 seconds
    - Throughput is 1,000 operations/second

Performance Goals: Peak Memory Utilization

- Given some sequence of `malloc` and `free` requests:
  - $R_0, R_1, ..., R_k, ..., R_{n-1}$
- *Def: Aggregate payload* P_k
  - **`malloc(p)`** results in a block with a ***payload*** of **`p`** bytes
  - After request R_k has completed, the ***aggregate payload*** P_k is the sum of currently allocated payloads
- *Def: Current heap size* H_k
  - Assume H_k is monotonically nondecreasing
    - i.e., heap only grows when allocator uses **`sbrk`**
- *Def: Peak memory utilization after k+1 requests*
  - $U_k = (max_{i \le k} P_i) \ / \ H_k$

Performance Goals: Peak Memory Utilization

- Given some sequence of `malloc` and `free` requests:
  - $R_0, R_1, \ldots, R_k, \ldots, R_{n-1}$
- *Def: Aggregate payload P_k*

- **Performance Goals:**
  - **1) Maximize throughput**
  - **2) Maximize peak memory utilization**
- **These goals are often conflicting!**

  - i.e., heap only grows when allocator uses **`sbrk`**
- *Def: Peak memory utilization after k+1 requests*
  - $U_k = (max_{i \le k} P_i) \ / \ H_k$