15-213 Recitation 10: Virtual Memory

21 March 2016

Ralf Brown and the 15-213 staff

Agenda

- Reminders
- Virtual Memory: Address Translation
- Virtual Memory: Memory Mapping
- Memory Allocation
- Malloclab Preview

That's **not** the kind of memory we'll be discussing....

Image: http://chimeraobscura.com/vm/ (used by permission)

Reminders

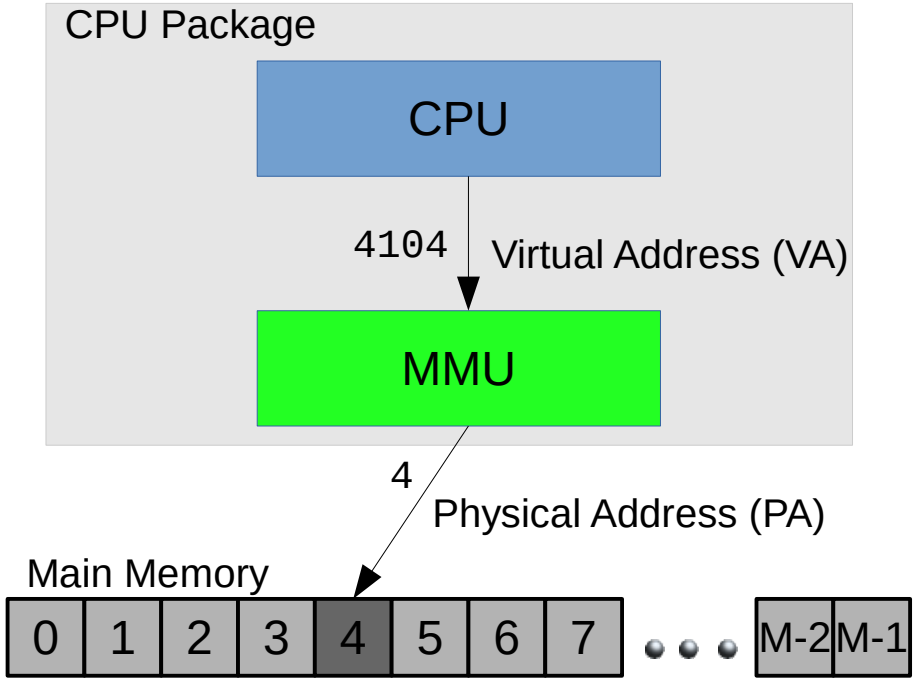
- Shell Lab is due **tomorrow!**
  - “But if you wait until the last minute, it only takes a minute!” - ***NOT!***
- Malloc Lab will be released **tomorrow!**

Virtual Memory

- So far, we've viewed memory as a linear array
- But if every program is loaded at 0x400000, how do we keep them from conflicting with each other?
- Answer: each process gets its own address space
  - the addresses it sees are not the addresses used by the RAM chips

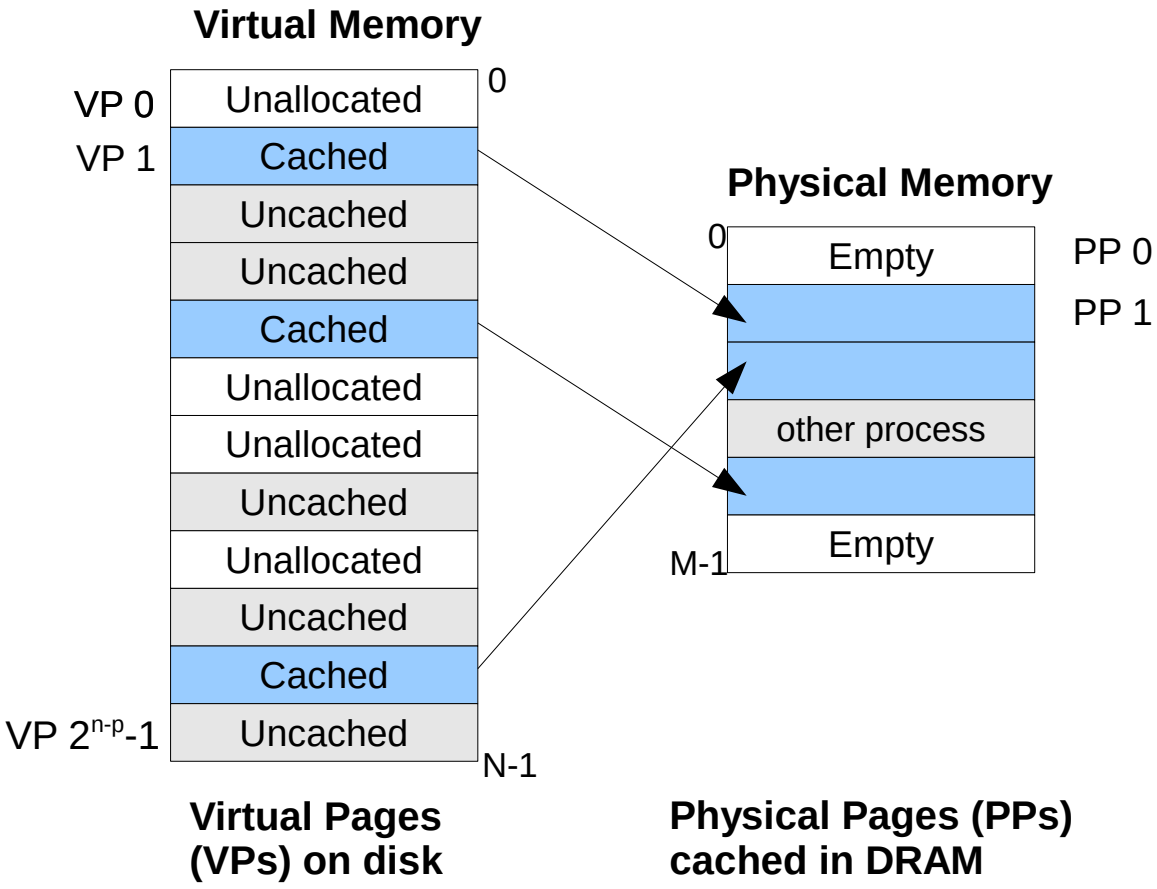
Virtual Memory: Address Translation

- Programs use virtual addresses
- These get mapped to the actual physical address in main memory by the memory-management unit
- Allows multiple programs to run in the same address range
- Protects programs from each other, operating system from programs
  - you can't change anything that isn't mapped to a virtual address
- Can share read-only memory



Virtual Memory as Caching

- VM can be thought of as an array of N contiguous bytes on disk, with the M bytes of physical memory as cache
- "cache blocks" called *pages*
  - size is $P = 2^p$ bytes, most commonly 4096, but may be 4MB or larger
  - large pages because misses are *very* expensive
- only read into RAM when accessed (demand paging)

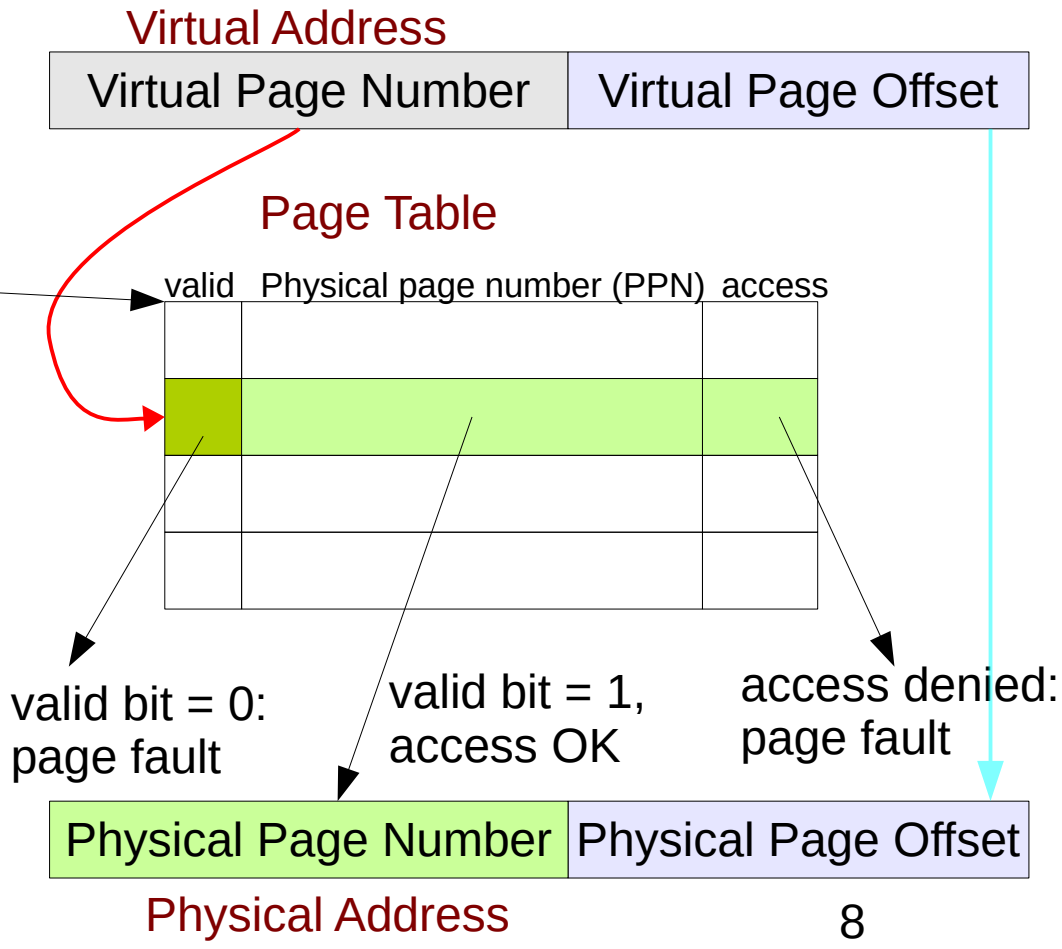


Address Translation: Concepts

- **Virtual Address Space**
  - V = {0, 1, ..., N-1}
  - virtual addresses are **n** bits long ($2^n = N$)
- **Physical Address Space**
  - P = {0, 1, ..., M-1}
  - physical addresses are **m** bits long ($2^m = M$)
- **Memory is divided into "pages"**
  - page size is P bytes; the offset into a page is **p** bits ($2^p = P$)
  - virtual page offset (VPO) and physical page offset (PPO) are the same!
    - no need to translate those bits

Address Translation: Page Tables

- Each process has its own private page table for address translation
  - Page Table Base Register holds physical address of PT
- A page table is an array of entries which specify for each virtual page
  - whether the page is in memory
  - the physical page number
  - access rights (executable, read-only, etc.)



Handling a Page Fault

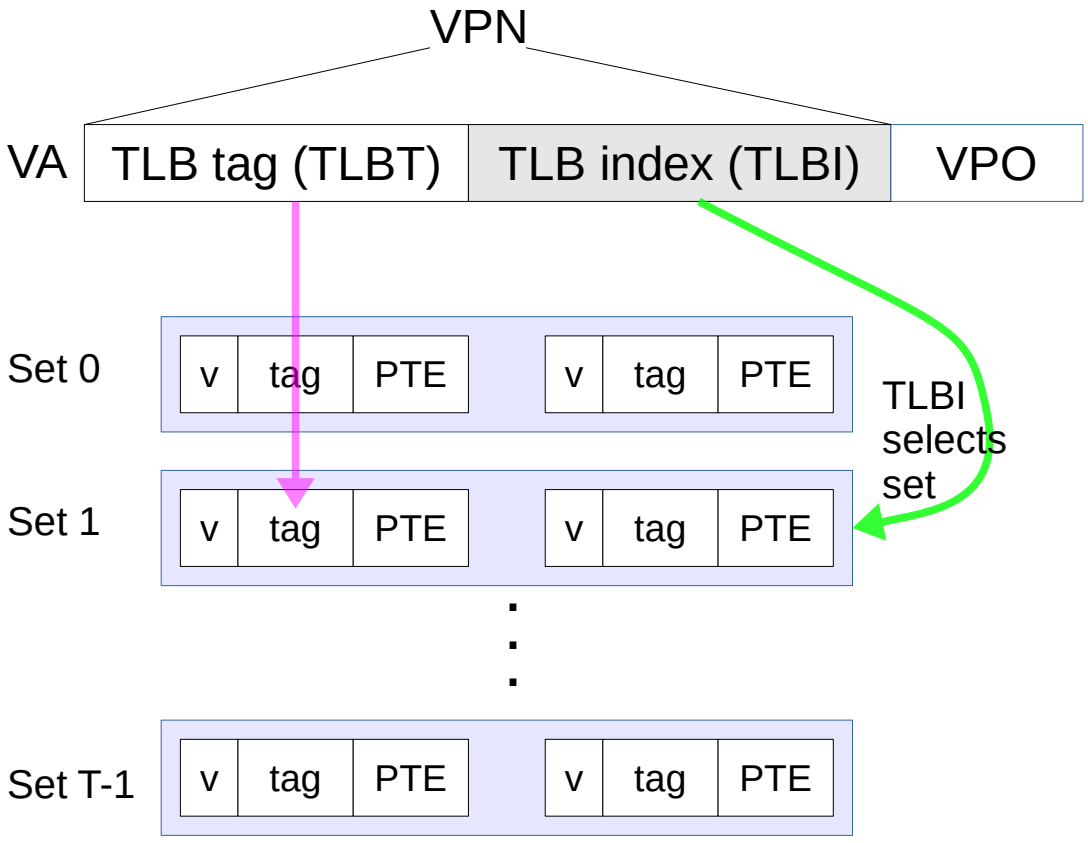
- When a page fault occurs, the CPU suspends the instruction that caused the fault and transfers control to a page-fault handler
- If the page is not in memory, the handler loads it and marks it as present in the page table entry
- the CPU then restarts the interrupted instruction and resumes as if the page had been present all along

Writing to a Read-Only Page

- Getting a page fault on writing to a read-only page lets us implement Copy-on-Write
- fork() doesn't copy a process's memory – only its page tables
  - but all pages are marked read-only in the copy
- The cloned process shares all of its memory with its parent, except those locations it modifies
  - the write causes a page fault, and the operating system then makes a fresh copy of that page and marks it writeable

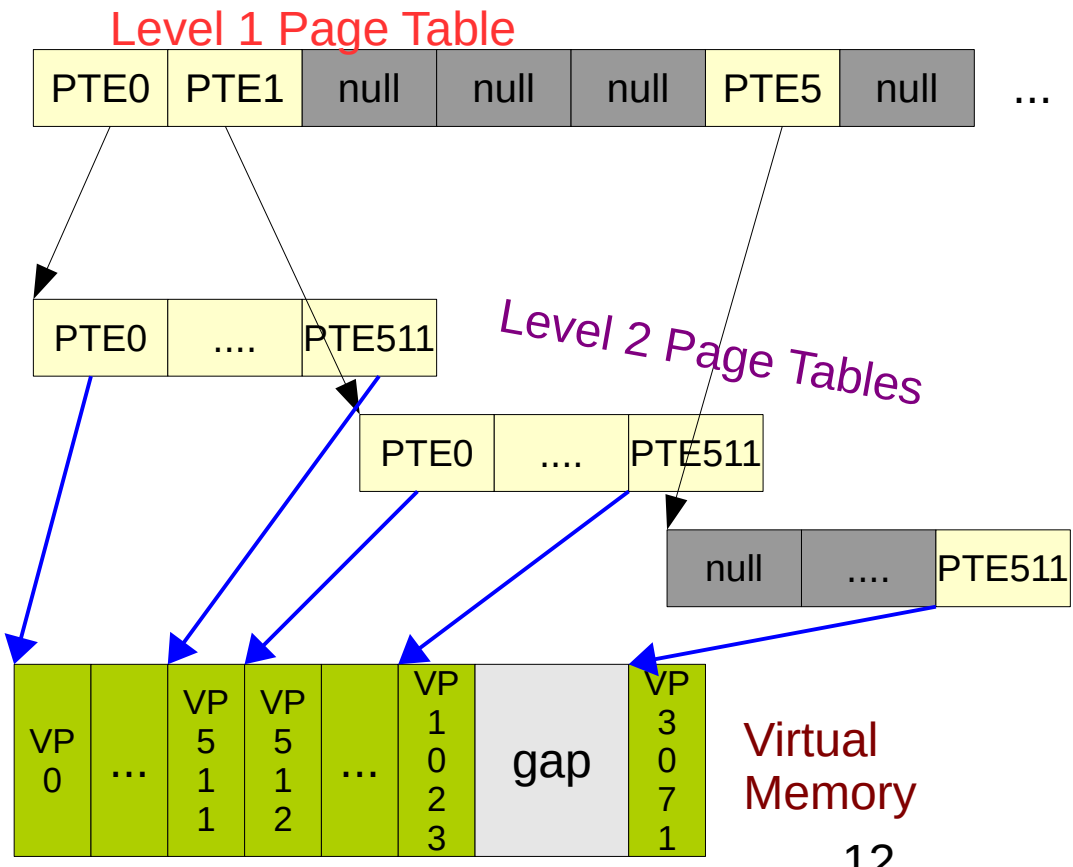
Translation Lookaside Buffer (TLB)

- Hardware cache in MMU
- Stores complete page table entries for a small number of pages (*why not all?*)
- TLB hit eliminates memory access to get the page table entry needed for a memory access
- TLB misses go through L1/L2 cache like any other memory access



Multi-Level Page Tables

- A big address space requires a big page table (512 GB for a 48-bit address space and 4K pages!)
- Solution: a table of page tables
  - top-level page table (page directory) stays in memory, second level pages can be demand-paged like other data
  - unused areas of the address space don't need to allocate second-level page tables
- How many memory accesses would the address translation need on a TLB miss for a 2-level table?



Example: Address Translation

1 MB of virtual memory

256 KB of physical memory

4 KB page size

TLB: 8 entries, 2-way set associative

- How many bits are needed to represent the virtual address space?
- How many bits are needed to represent the physical address space?

- How many bits are needed to represent the page offset?
- How many bits are needed to represent the VPN?

- How many bits are in the TLB index?
- How many bits are in the TLB tag?

Example: Address Translation

1 MB of virtual memory

256 KB of physical memory

4 KB page size

TLB: 8 entries, 2-way set associative

- How many bits are needed to represent the virtual address space?
- How many bits are needed to represent the physical address space?
  - **20 virtual (1MB = 2^{20}), 18 physical (256 KB = 2^{18})**
- How many bits are needed to represent the page offset?
- How many bits are needed to represent the VPN?
  - **12 offset bits (4 KB = 2^{12}), 8 bits for VPN (20-12)**
- How many bits are in the TLB index?
- How many bits are in the TLB tag?
  - **2 index bits (4 sets = 2^{2}), 6 tag bits (8-2)**

Example: Address Translation with TLB

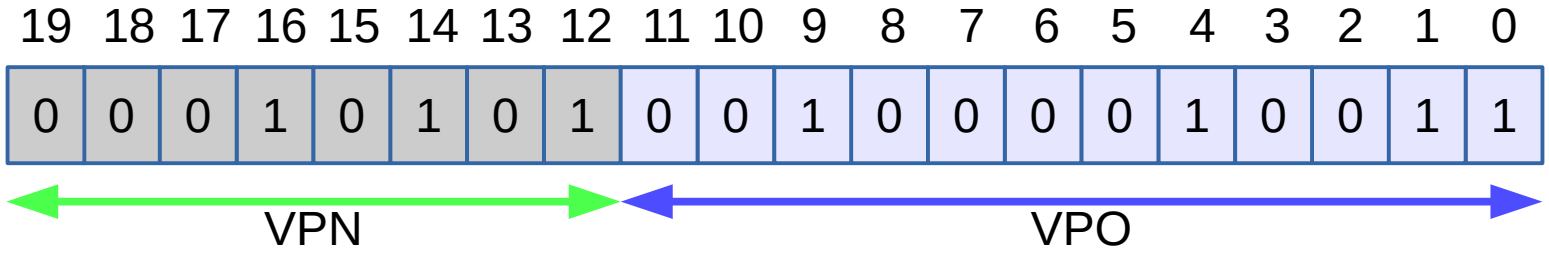
- Translate 0x15213, given the contents of the TLB and the first 32 entries of the page table below.

2-way set associative

Index	Tag	PPN	Valid
0	05	13	1
	3F	15	1
1	10	0F	1
	0F	1E	0
2	1F	01	1
	11	1F	0
3	03	2B	1
	1D	23	0

VPN	PPN	Valid	VPN	PPN	Valid
00	17	1	10	26	0
01	28	1	11	17	0
02	14	1	12	0E	1
03	0B	0	13	10	1
04	26	0	14	13	1
05	13	0	15	18	1
06	0F	1	16	31	1
07	10	1	17	12	0
08	1C	0	18	23	1
09	25	1	19	04	0
0A	31	0	1A	0C	1
0B	16	1	1B	2B	0
0C	01	0	1C	1E	0
0D	15	0	1D	3E	1
0E	0C	0	1E	27	1
0F	2B	1	1F	15	1

Address Translation with TLB: 0x15213

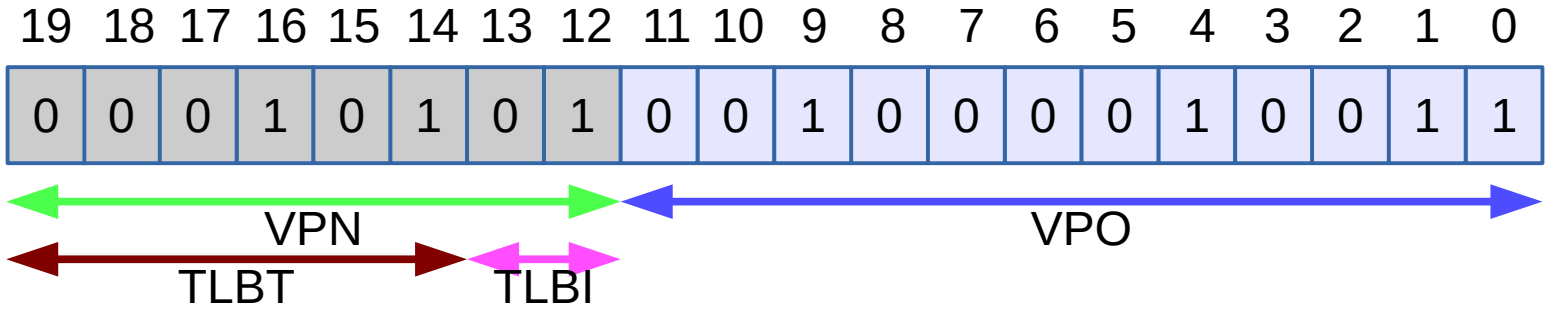


VPN = ?

TLBI = ?

TLBT = ?

Address Translation with TLB: 0x15213

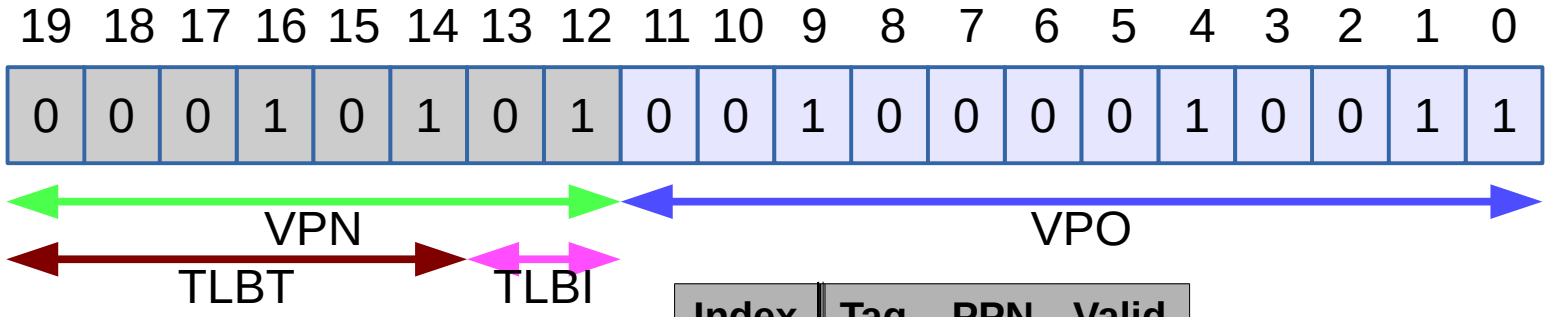


VPN = 0x15

TLBI = ?

TLBT = ?

Address Translation with TLB: 0x15213



VPN = 0x15

TLBI = 1

TLBT = 0x05

Index	Tag	PPN	Valid
0	05	13	1
	3F	15	1
1	**10**	**0F**	**1**
	0F	**1E**	**0**
2	1F	01	1
	11	1F	0
3	03	2B	1
	1D	23	0

TLB **Miss!**

We'll have to look it up in the page table....

Address Translation with TLB: 0x15213

19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	1	0	1	0	1	0	0	1	0	0	0	0	1	0	0	1	1

VPN = 0x15

TLBI = 1

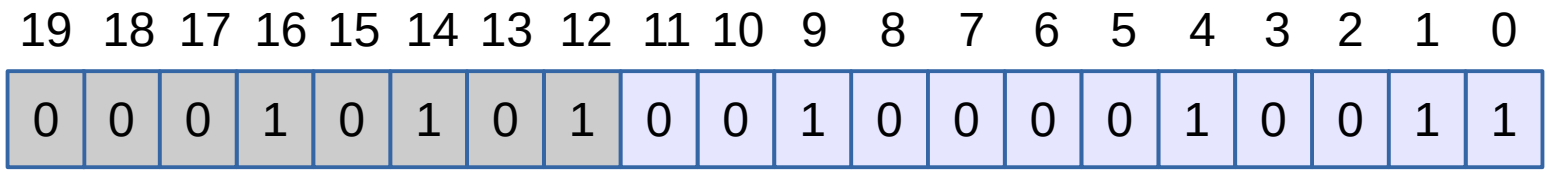
TLBT = 0x05

VPN	PPN	Valid	VPN	PPN	Valid
00	17	1	10	26	0
01	28	1	11	17	0
02	14	1	12	0E	1
03	0B	0	13	10	1
04	26	0	14	13	1
05	13	0	15	18	1
06	0F	1	16	31	1
07	10	1	17	12	0
08	1C	0	18	23	1
09	25	1	19	04	0
0A	31	0	1A	0C	1
0B	16	1	1B	2B	0
0C	01	0	1C	1E	0
0D	15	0	1D	3E	1
0E	0C	0	1E	27	1
0F	2B	1	1F	15	1

Page Table Hit
PPN = ?
Offset = ?

Physical Address:
?

Address Translation with TLB: 0x15213



VPN = 0x15

TLBI = 1

TLBT = 0x05

VPN	PPN	Valid	VPN	PPN	Valid
00	17	1	10	26	0
01	28	1	11	17	0
02	14	1	12	0E	1
03	0B	0	13	10	1
04	26	0	14	13	1
05	13	0	**15**	**18**	**1**
06	0F	1	16	31	1
07	10	1	17	12	0
08	1C	0	18	23	1
09	25	1	19	04	0
0A	31	0	1A	0C	1
0B	16	1	1B	2B	0
0C	01	0	1C	1E	0
0D	15	0	1D	3E	1
0E	0C	0	1E	27	1
0F	2B	1	1F	15	1

Page Table Hit
PPN = 0x18
Offset = 0x213

Physical Address:
0x18213

Memory Mapping

- VM areas are initialized by associating them with disk objects ("memory mapping")
- An area can get its initial values from (be "backed by")
  - a regular file on disk (e.g. an executable)
    - the first fault will read the corresponding section of the file into memory
  - an anonymous file (e.g. nothing)
    - the first fault will allocate a physical page filled with zero bytes
    - once written to (dirtied), the page is like any other
- Dirty pages are copied between memory and a special swap file.
- No virtual pages are copied into memory until accessed!
  - "demand paging" - crucial for time and space efficiency

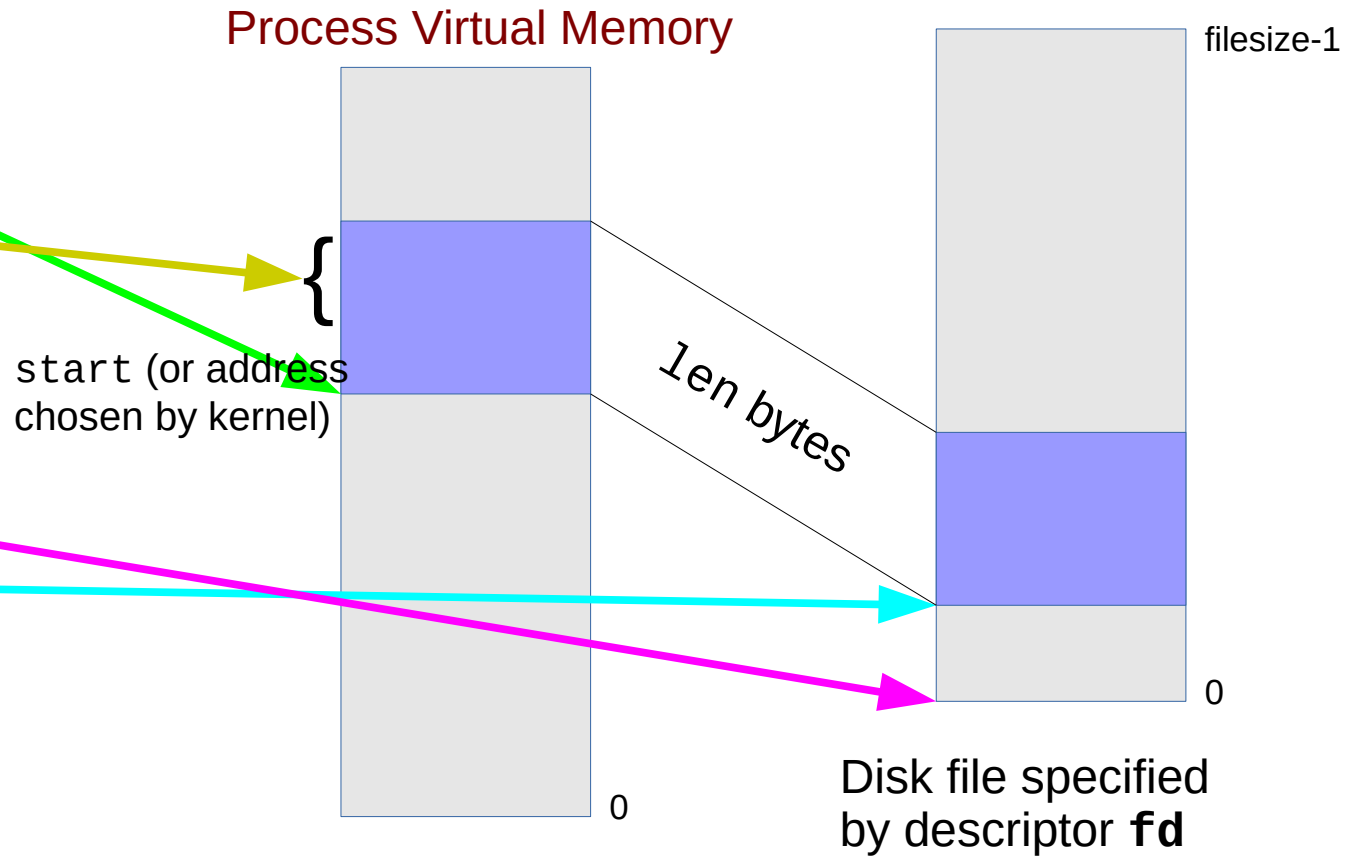
User-Level Memory Mapping

- `#include <sys/mman.h>`
- `void *mmap(void *start, size_t length, int prot, int flags, int fd, off_t offset);`
- map `length` bytes starting at offset `offset` of the file referenced by descriptor `fd`, preferably at address `start`
  - `start` = NULL for “pick an address” (i.e. “I don't care”)
  - `prot`: bit flags PROT_READ, PROT_WRITE, ...
  - `flags`: MAP_ANONYMOUS, MAP_PRIVATE, MAP_SHARED, ...
- returns a pointer to the start of the mapped area (which may differ from `start`) or MAP_FAILED

User-Level Memory Mapping: File Mapping

```
void *mmap(
        void *start,
        size_t len,
        int prot,
        int flags,
        int fd,
        off_t offset);
```

- Note that it is not necessary to map the entire file



User-Level Memory Mapping: Releasing a Mapping

- `#include <sys/mman.h>`
- `int munmap(void *start, size_t length);`
- Remove the mapping for the specified address range
  - future accesses become invalid, generate segmentation faults
  - required to free resources – closing the file descriptor used in mmap() does **not** unmap memory!
- Returns 0 on success, -1 on error (and sets `errno`)

Advantages of Memory Mapping Files

- ?

Advantages of Memory Mapping Files

- Can treat the file as an array, instead of having to use read(), etc.
  - but sizes of mapped files were limited by address space on 32-bit machines
- Multiple processes can load common data while keeping only one copy in RAM
  - using flag MAP_SHARED
- Possible to copy from one file to another without copying the bytes from kernel buffers into user buffers

Copying Files using `mmap`

- Copying without transferring data to user space

```
#include <sys/mman.h>
#include <unistd.h>
/* mmapcopy – uses mmap to copy   */
 *            file fd to stdout   */
void mmapcopy(int fd, int size)
{
  /* ptr to mem-mapped VM area */
  char *bufp;
  bufp = mmap(NULL, size, PROT_READ,
              MAP_PRIVATE, fd, 0);
  if (bufp == MAP_FAILED)
     fatal_error("mapping failed");
  }
  if (write(1, bufp, size) != size)
     fatal_error("write failed");
  munmap(bufp,size);
}
```

```
/* mmapcopy driver */
int main(int argc, char **argv)
{
  struct stat stat;
  int fd;
  /* check for req'd cmdline arg */
  if (argc != 2)
     print_usage_and_exit();
  /* copy the input arg to stdout */
  fd = open(argv[1], O_RDONLY, 0);
  if (fd == -1) fatal_error("open");
  if (fstat(fd, &stat) == -1)
     fatal_error("file stats") ;
  mmapcopy(fd, stat.st_size);
  exit(EXIT_SUCCESS);
}
```

Copying Files using `mmap`

- Copying without transferring data to user space

```
#include <sys/mman.h>
#include <unistd.h>
/* mmapcopy – uses mmap to copy   */
 *            file fd to stdout   */
void mmapcopy(int fd, int size)
{
  /* ptr to mem-mapped VM area */
  char *bufp;
  bufp = mmap(NUL
              MAP
  if (bufp == MA
     fatal_err
  }
  if (write(1, bufp, size) != size)
     fatal_error("write failed");
  munmap(bufp,size);
}
```

We're writing data we never explicitly read!

```
/* mmapcopy driver */
int main(int argc, char **argv)
{
  struct stat stat;
  int fd;
  /* check for req'd cmdline arg */
  if (argc != 2)
     print_usage_and_exit();
  /* copy the input arg to stdout */
  fd = open(argv[1], O_RDONLY, 0);
  if (fd == -1) fatal_error("open");
  if (fstat(fd, &stat) == -1)
     fatal_error("file stats") ;
  mmapcopy(fd, stat.st_size);
  exit(EXIT_SUCCESS);
}
```

Allocating Memory with `mmap`

- Remember how we said that a memory mapping can be backed by an anonymous file?
- mmap's MAP_ANONYMOUS flag lets us do just that
  - no file descriptor needed
  - physical pages will be allocated (and zeroed) as we access the virtual address space of the returned mapping
- The net effect of a MAP_ANONYMOUS mapping is that we've just allocated `length` bytes of memory for our process
  - most implementations of malloc() use this for large allocations

Dynamic Memory Allocation

- What is the simplest possible memory allocation scheme?
- What are its advantages and disadvantages?

Dynamic Memory Allocation

- What is the simplest possible memory allocation scheme?
  - reserve space on the stack
- What are its advantages and disadvantages?
  - ✔ extremely simple
  - ✔ super fast
  - ✔ automatic deallocation
  - ✗ can't retain after function returns
  - ✗ can't deallocate in arbitrary order
  - ✗ restricted amount of memory

General-Purpose Memory Allocation

- Required:
  - ?
- Desirable:
  - ?

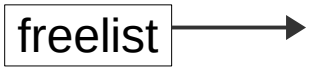
General-Purpose Memory Allocation

- Required:
  - allow arbitrary sizes
  - allow deallocation in arbitrary order
  - no restrictions on how long allocation can be kept
  - no restrictions on passing references to an allocation
  - allow program to use all available memory
- Desirable:
  - as fast as possible
  - as little memory overhead as possible

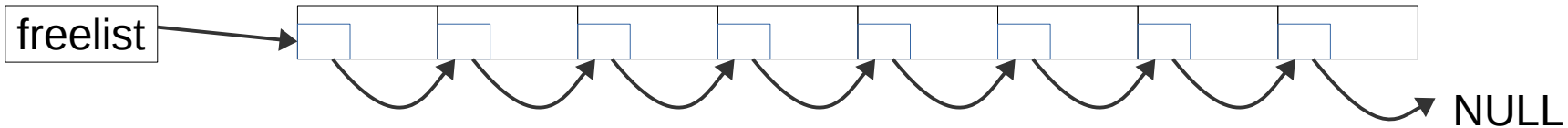
Special-Purpose: Fixed-Size Memory Allocators

- Can get much faster and simpler code if we restrict allocations to a specific size
  - useful if we have many instances of a particular struct
- Basic idea: keep a linked list of unallocated objects
  - when allocating an object, remove the first one on the list
    - if the list is empty, allocate another batch of objects from the OS
  - when done with an object, add it back to the list
- Drawback: hard to share memory between different allocation sizes
- We'll talk about more general versions of free lists that can handle variable sizes next week

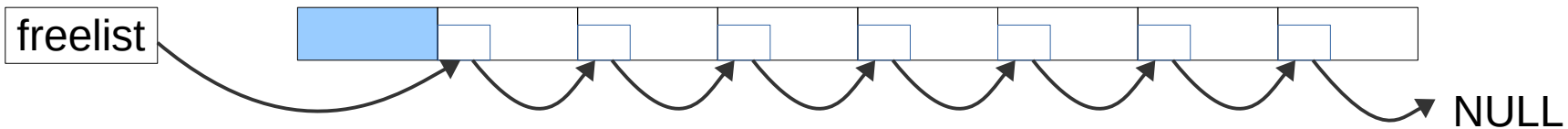
Fixed-Size Free-List Memory Allocation



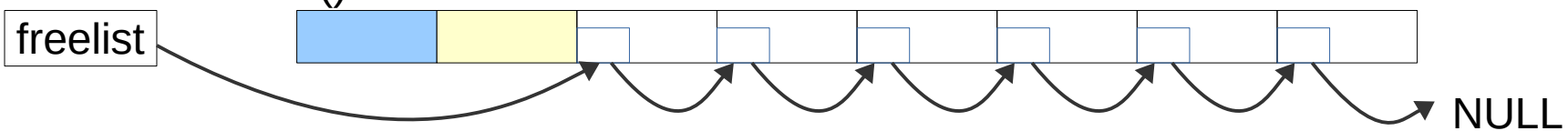
when called, alloc() requests a block of memory from the OS and sets up a list of empty objects:



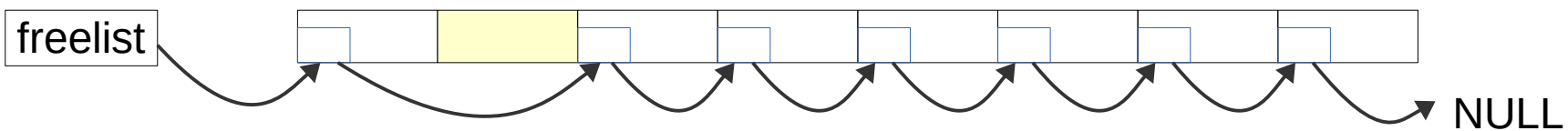
alloc() now returns the first item, and updates the freelist pointer:



next call to alloc() allocates another item:



freeing the first item puts it back on the free list:



Malloclab Preview

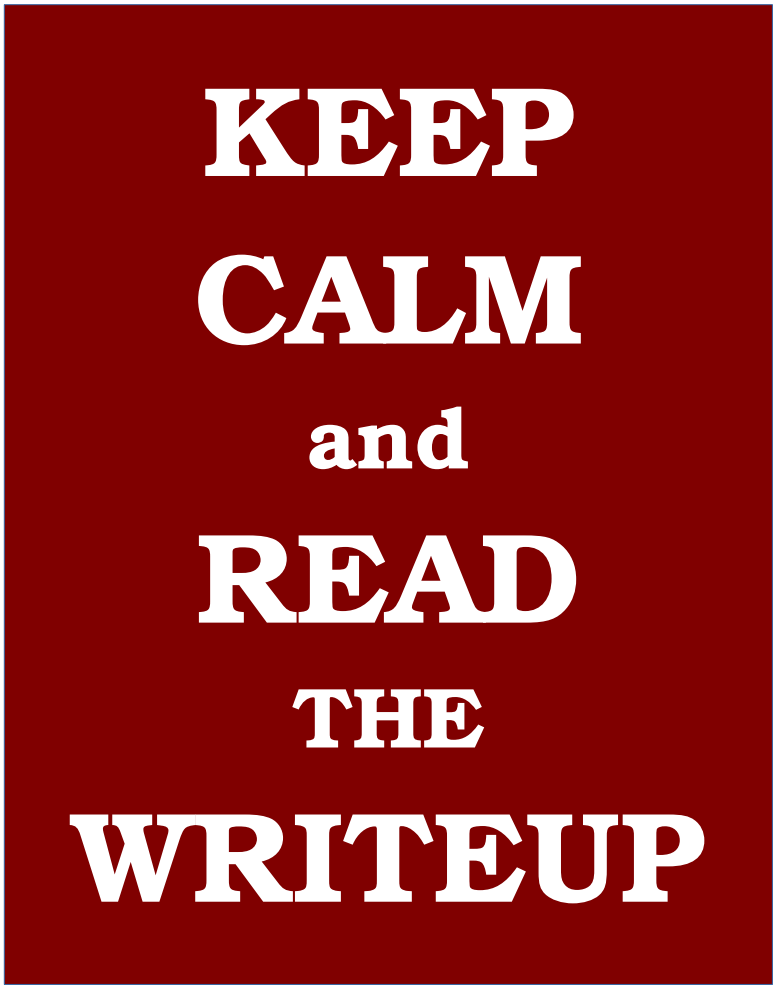
- You will be writing your own memory allocator, with specific speed and memory-overhead constraints
  - macros and inline functions will help you achieve the speed goals
- Functions to implement:
  - `int mm_init(void);`
  - `void *malloc(size_t size);`
  - `void *realloc(void *ptr, size_t newsize);`
  - `void *calloc(size_t n, size_t size);`
  - `void free(void *ptr);`
  - `void mm_checkheap(int verbose);`

C: Macros and Inline Functions

- Macros: text is substituted by the preprocessor before the compiler sees it
  - use #define to avoid magic numbers: `#define TRIALS 100`
  - use function-like macros for short, heavily-used code snippets
    - `#define GET_BYTE_ONE(x)  ((x) & 0xff)`
    - `#define GET_BYTE_TWO(x)  (((x)>>8) & 0xff)`
- Inline functions: ask the compiler to insert the complete body of the function in every location where it is called
  - `inline int fn(int a, int b) { ... body ... }`
  - easier to write safely than a function-like macro, and just as fast when the compiler inlines the code (may not be inlined if the resulting code becomes too large or recursive calls are made)

If You Get Stuck

- ***Please read the writeup!***
- CS:APP Chapter 9
- View lecture notes and course FAQ at http://www.cs.cmu.edu/~213
- Office hours Sunday through Thursday 5:00-9:00pm in WeH 5207
- Post a **private** question on Piazza


KEEP
CALM
and
READ
THE
WRITEUP