Main Memory

COMP 3361: Operating Systems I

Winter 2015

http://www.cs.du.edu/3361

1 Background

- A program must be brought (from disk) into memory for it to run
- A typical instruction-execution cycle
  - fetch
  - decode (may result in more fetch operations)
  - execute
- We shall focus on the "**fetch**" part
  - generating memory addresses from where data/instruction is to be read/written

2 Memory Manager

- Keep track of occupied/available **main memory (RAM)**
  - cache management done in hardware
  - register management done by compilers
- Allocate memory to processes when they need it
  - tell a process which part of free memory it should use
- Deallocate memory when a process is done using it
  - mark the memory areas as being available

Two Programs

address	instruction
16383	NOP
	…
28	ADD …
24	MOV …
	…
0	JMP 28

Program A (16KB)

address	instruction
16383	NOP
	…
28	CMP …
24	CALL …
	…
0	JMP 24

Program B (16KB)

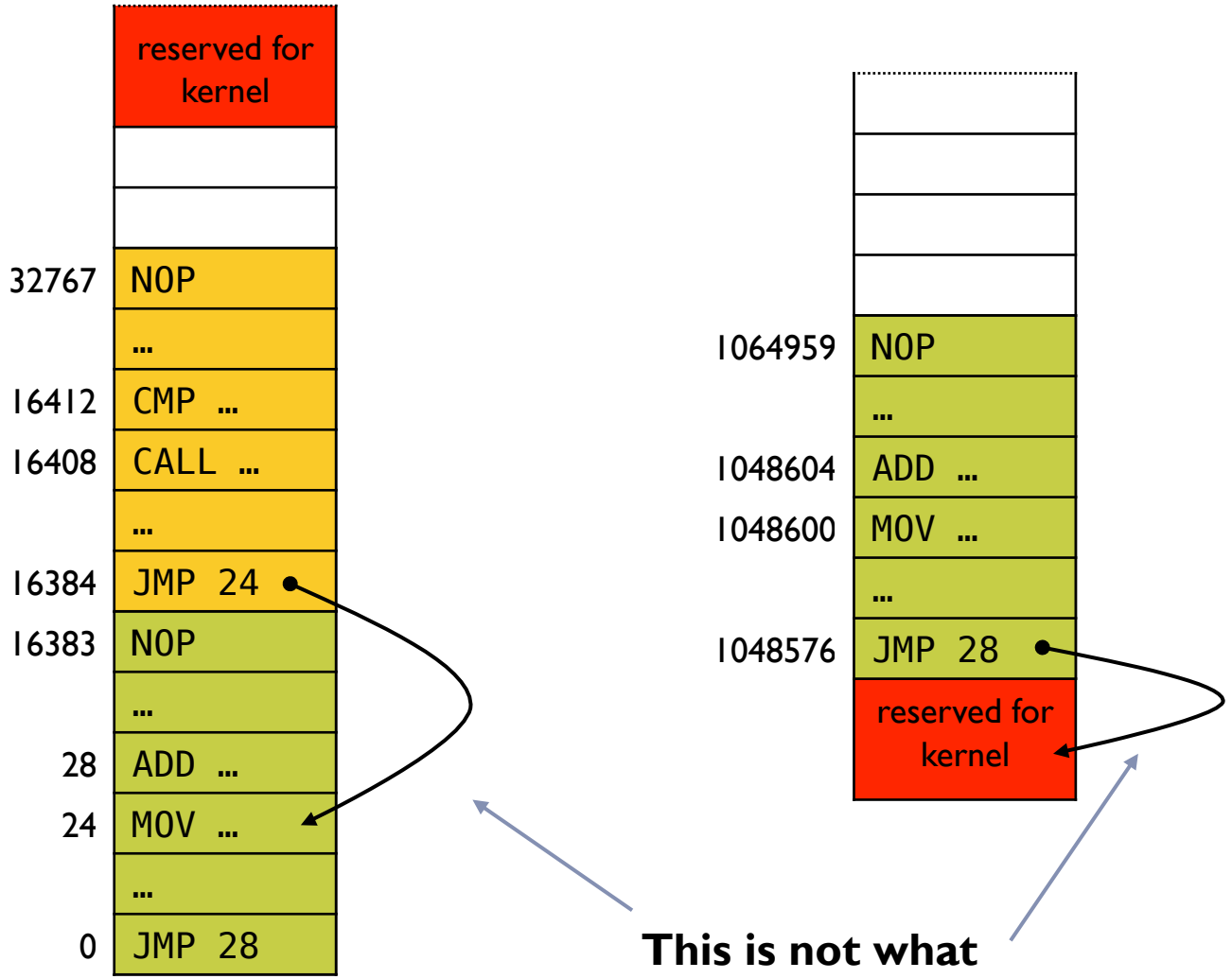
Both programs assume they will be placed at memory address zero

4 Running a Program

Program A no longer exists in memory!

Main Memory

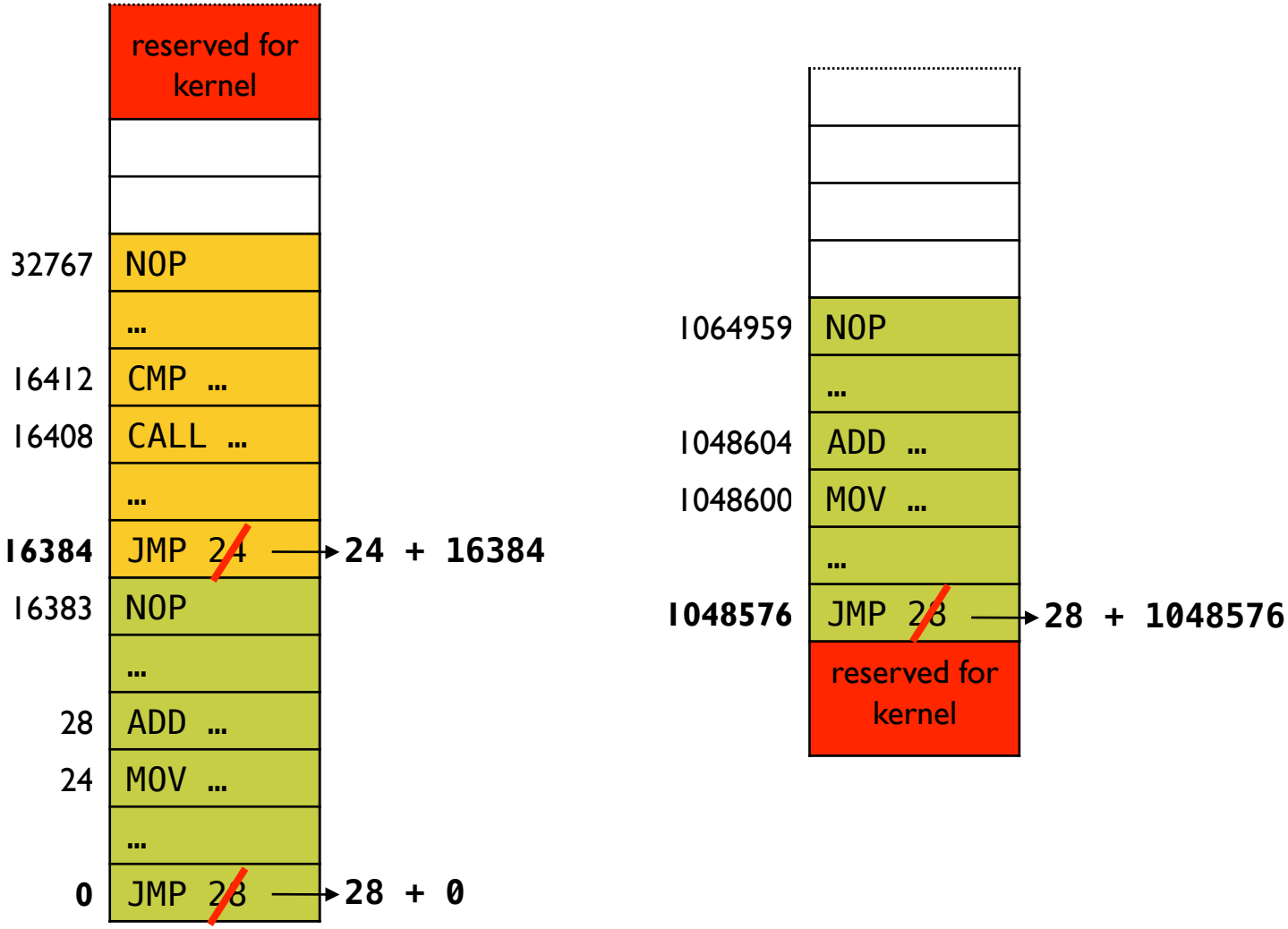
Critical Problems



Relocation Problem

- Instructions in both programs assume that the first instruction is at memory address zero
- **Relocation problem**: If program is placed at a location other than what was assumed, all address based calculations fail
- **Static relocation**
  - before starting execution, modify all addresses in program by adding the actual start address of the program
  - complicated since not all numbers are addresses

With Static Relocation



Process Address Space

- Static relocation works, but is complicated to do in software
- Disconnect the link between addresses in program instructions and physical memory
- Process address space: the set of addresses a process can use
  - each process will have its own address space
  - address *x* in one process is different from address *x* in another
    - addresses generated by a program will be called **logical (virtual) addresses**
  - each program is created **assuming**
    - it is the only one running
    - and it will be placed at physical memory address zero

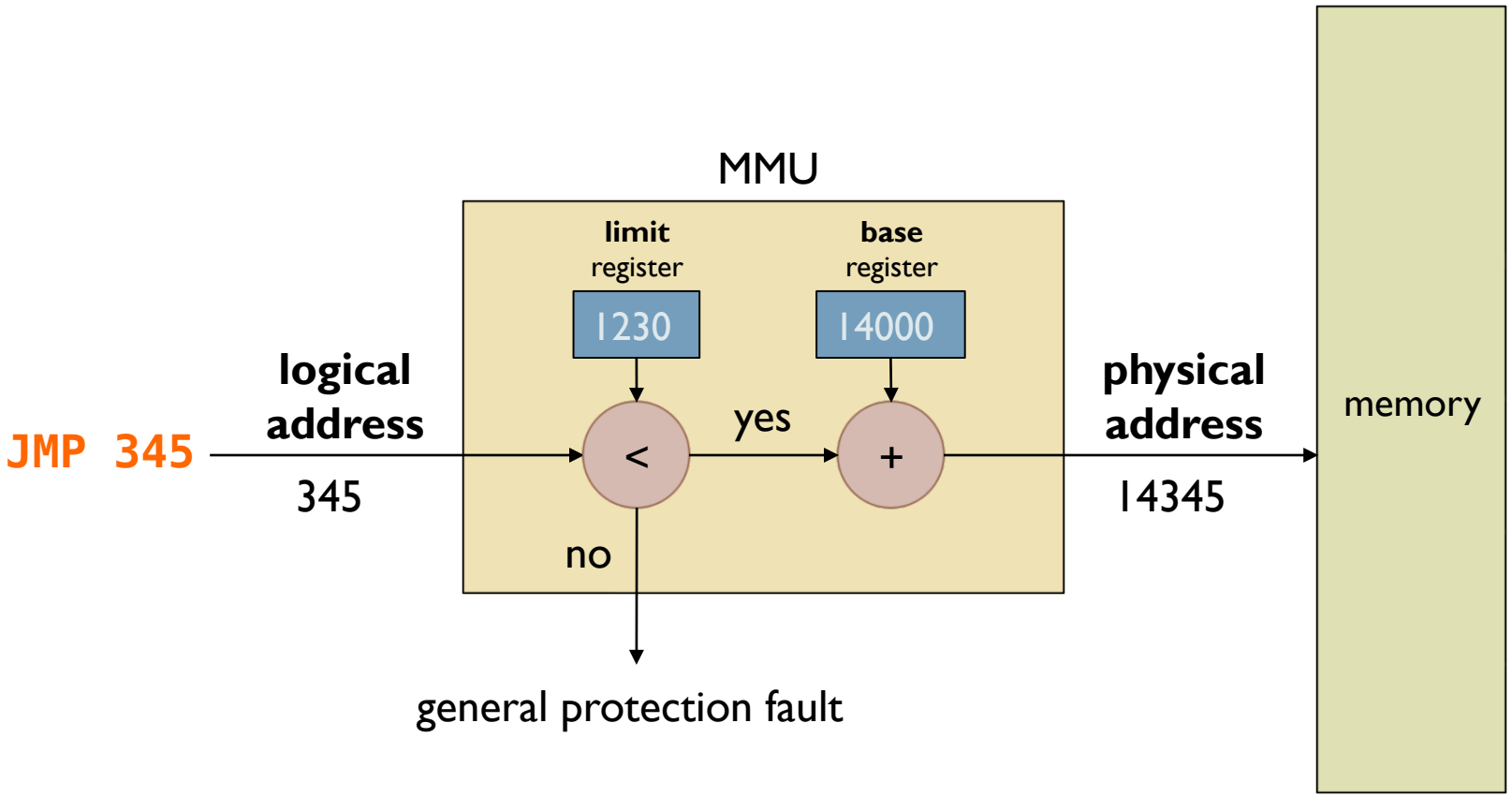
Dynamic Relocation

- Address translation is done by a hardware component called the **memory management unit (MMU)**
- MMU has two registers – **Base** and **Limit**

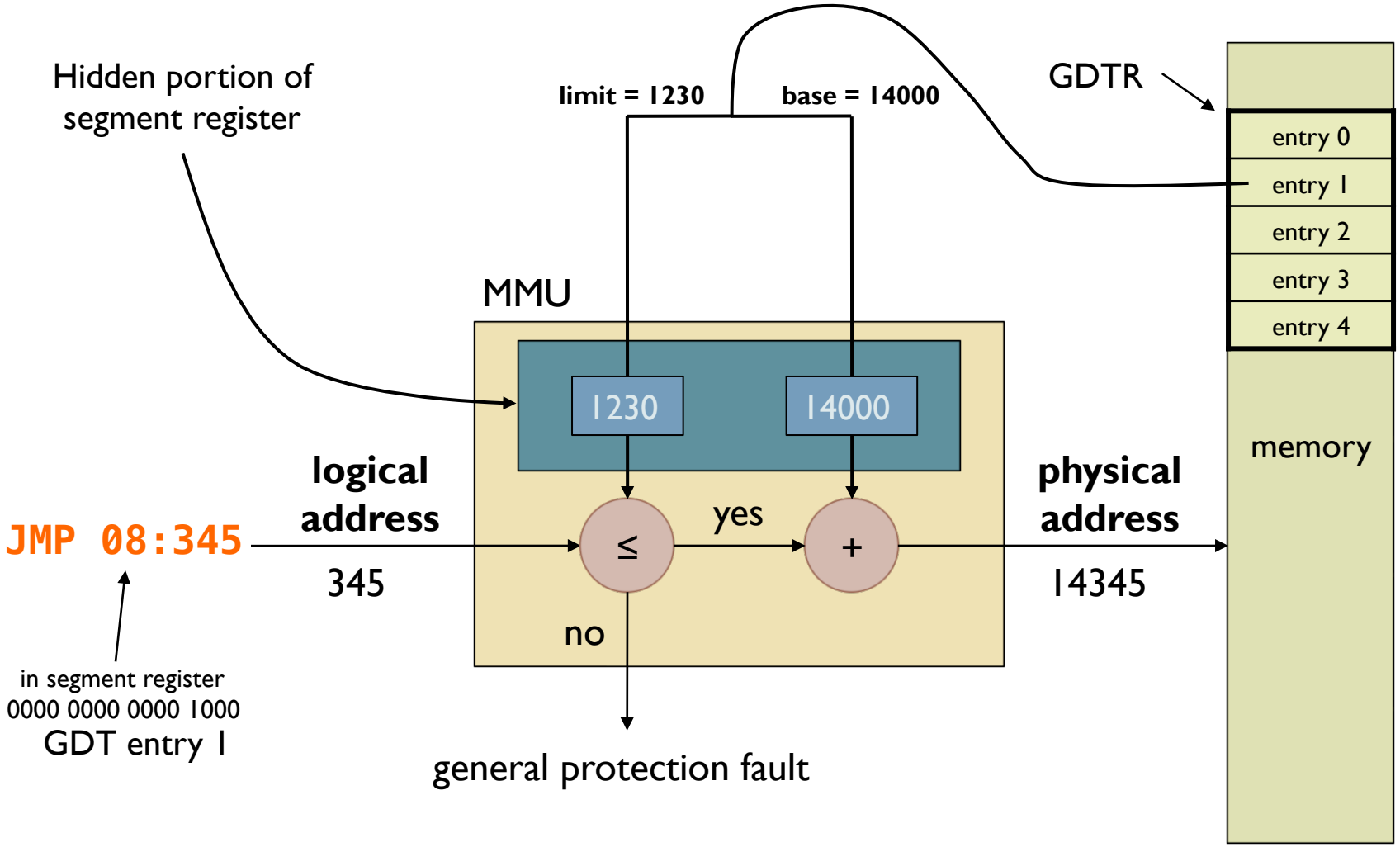
- When a program runs
  - load Base with the start address of program in physical memory
  - load Limit with size of program

- When a program refers to a memory address *m*, MMU
  - checks if *m* is less than the value in Limit
  - adds Base to *m* to determine the actual physical address

MMU with Base and Limit Registers



11 Relocation Today



Memory Allocation

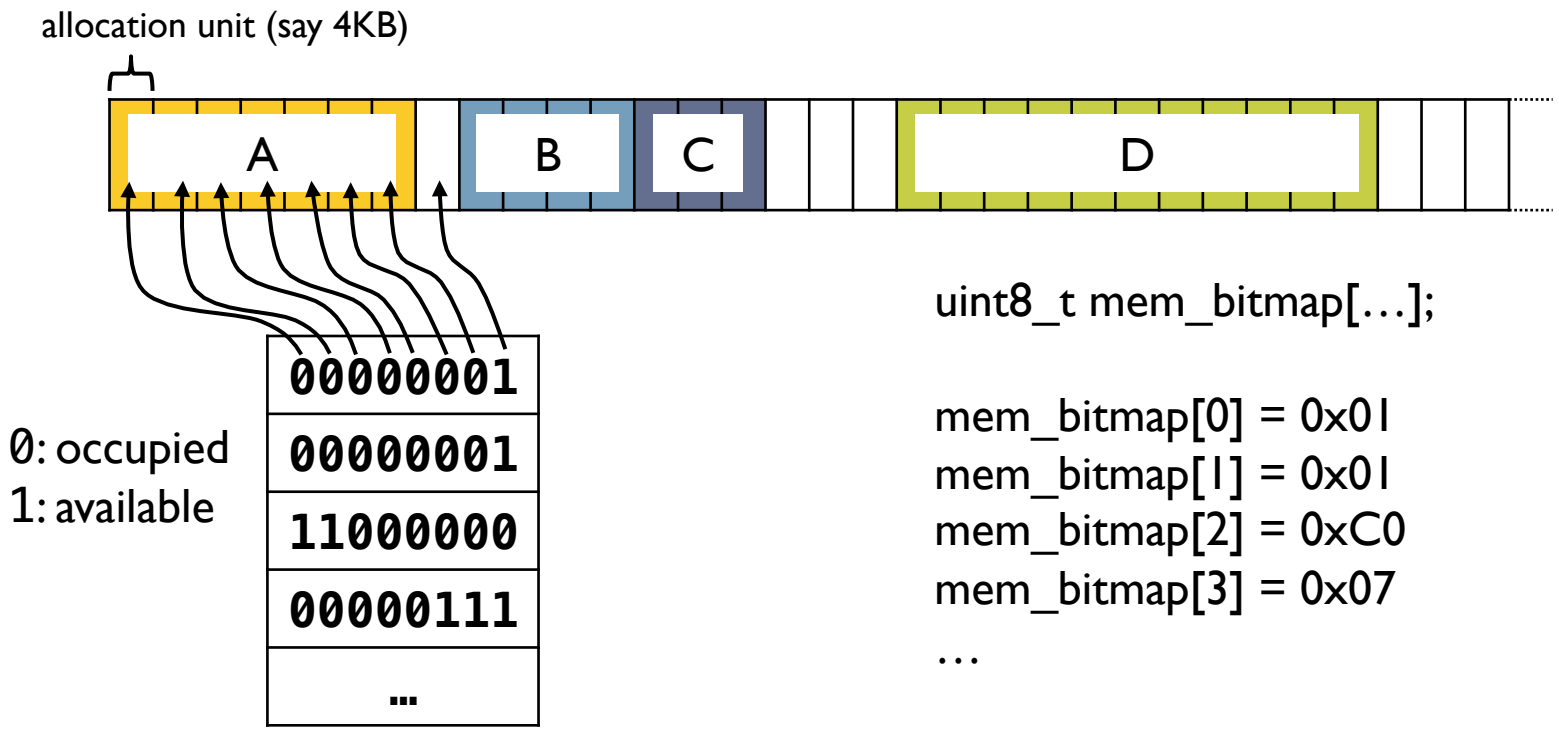
- Main memory is usually divided into two partitions:
  - resident operating system, usually held in low memory with interrupt vector
  - user processes held in high memory
- How to keep track of free memory?
  - quick allocation
  - quick de-allocation
- How to allocate memory areas?
  - contiguous
  - non-contiguous

13 A DUMB Memory Manager

```
extern uint64_t total_memory;

uint32_t *alloc_base; // the returned memory base address
uint32_t total_memory_bytes;

/*** Initialize memory manager ***/
void init_memory_manager(void) {
    total_memory_bytes = total_memory * 1024;

    // allocated memory always begins at 4MB mark
    alloc_base = (uint32_t *)(0x400000);
}

/*** Allocate count bytes of memory ***/
void *alloc_memory(uint32_t count) {
    int i;

    // check if we have the requested memory
    if ((uint32_t)alloc_base + count > total_memory_bytes)
        return NULL;

    return alloc_base;
}
```

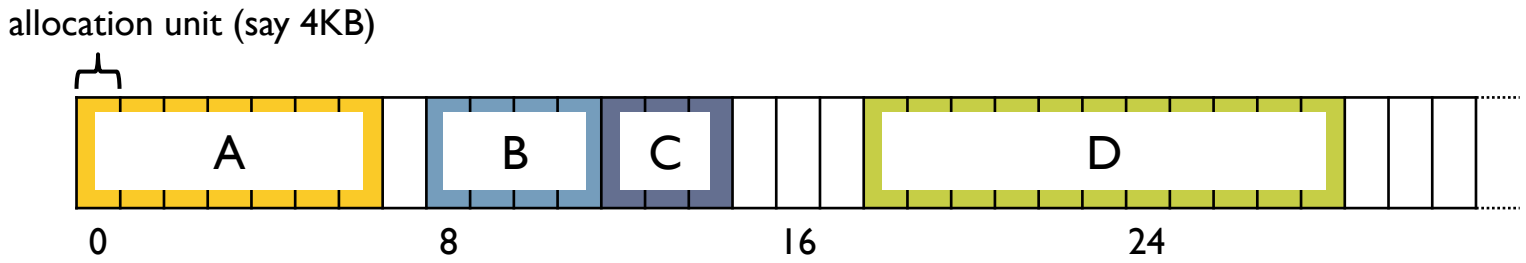
14 Tracking Memory Usage with Bitmaps

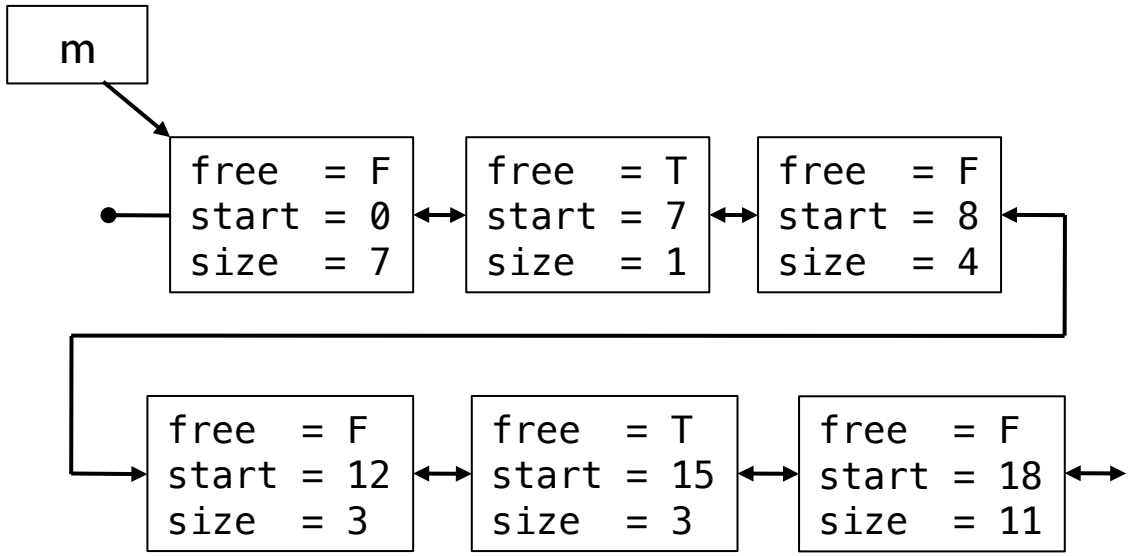
- Divide memory into **allocation units** or **frames**
  - memory will always be allocated in multiples of allocation units




```
uint8_t mem_bitmap[…];

mem_bitmap[0] = 0x01
mem_bitmap[1] = 0x01
mem_bitmap[2] = 0xC0
mem_bitmap[3] = 0x07
…
```

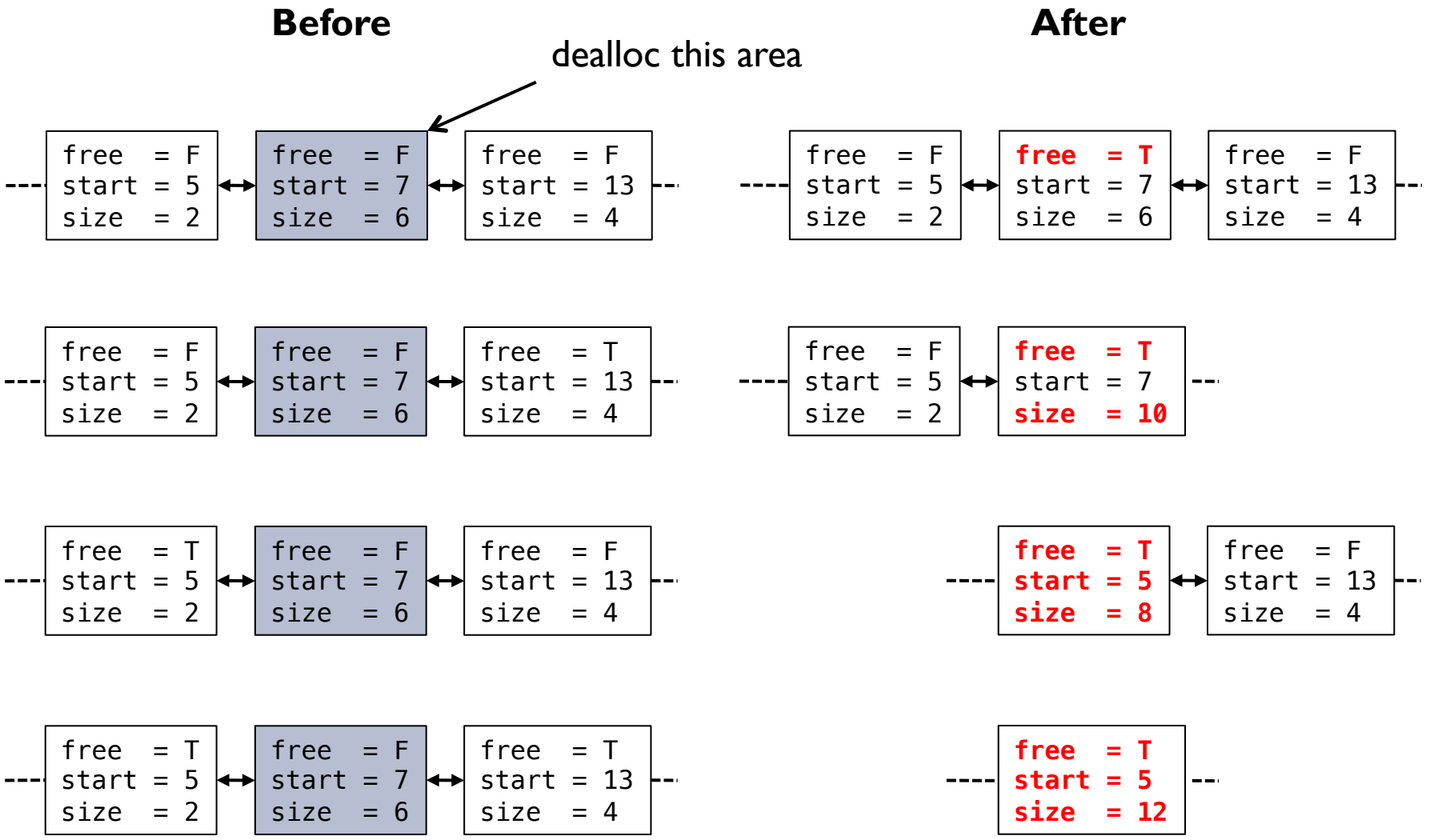
Tracking Memory Usage with Lists





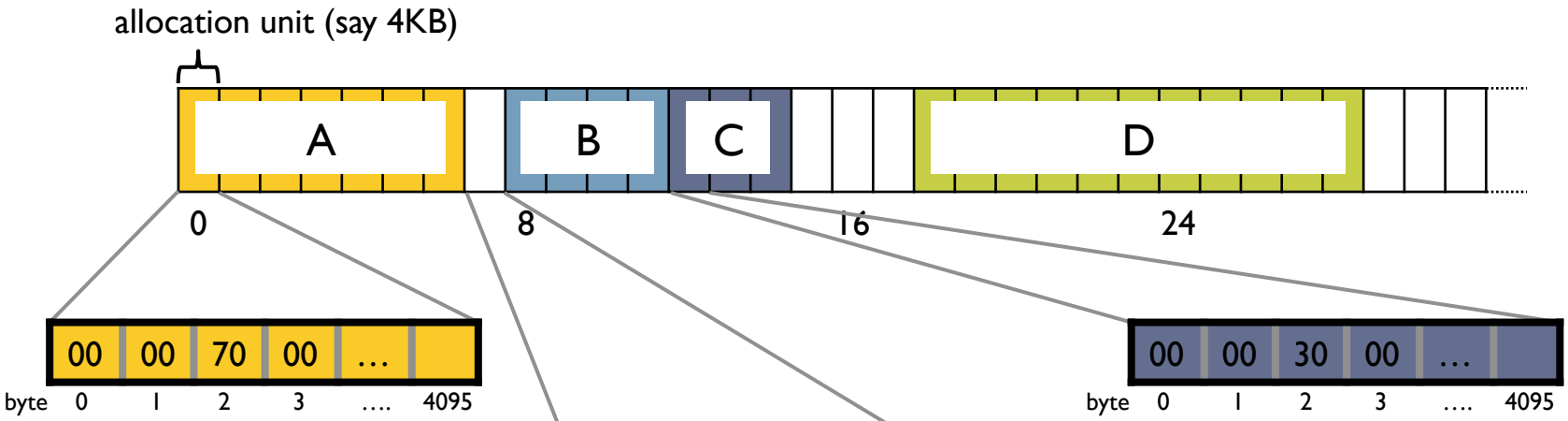

```
typedef struct _m_area {
        bool free;
        uint32_t start;
        uint32_t size;
        struct _m_area *next;
        struct _m_area *prev;
} m_area;

m_area *m;
```

List Update


Before
After
dealloc this area
free = F
start = 5
size = 2
free = F
start = 7
size = 6
free = F
start = 13
size = 4
free = T
start = 7
size = 6
free = T
start = 13
size = 4
free = T
start = 5
size = 2
free = F
start = 7
size = 6
free = F
start = 13
size = 4
free = F
start = 5
size = 2
free = T
start = 7
size = 10
free = T
start = 5
size = 2
free = F
start = 7
size = 6
free = T
start = 13
size = 4
free = T
start = 5
size = 8
free = F
start = 13
size = 4
free = T
start = 5
size = 2
free = F
start = 7
size = 6
free = T
start = 13
size = 4
free = T
start = 5
size = 12

Implicit Free Lists

- Store free/used bit and area size in first four bytes of the memory area



0x00007000
0x7000 bytes = 7x4KB
occupied (bit 0 = 0)

0x00001001
0x1000 bytes = 1x4KB
free (bit 0 = 1)

0x3000 bytes
occupied

What Next?

- Process A needs n frames of memory
- There are more than n frames of free memory
- All free areas are less than n frames in size!
- What to do?

References

- Chapter 3.1-3.2, Modern Operating Systems, A. Tanenbaum and H. Bos, 4th Edition.