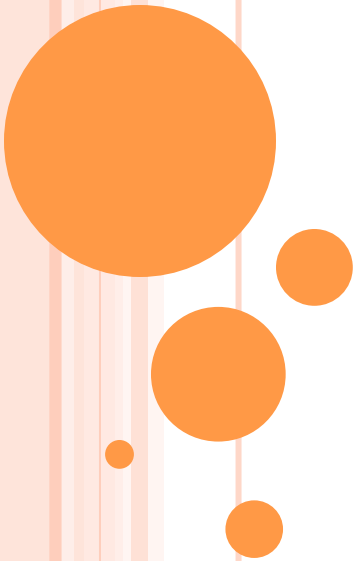


CS24: Introduction to Computing Systems

Spring 2015

Lecture 7

Last Time

- Dynamic memory allocation and the heap:
  - A *run-time facility* that satisfies multiple needs:
    - Programs can use widely varying, possibly large amounts of memory for computations
    - Allocated memory is available beyond a single function call
  - Allocate with: `void * malloc(size_t size)`
  - Release with: `void free(void *ptr)`
- C arrays: `T A[N]`
  - Allocates `N` × `sizeof(T)` bytes for the array
  - Element *i* resides at address `A + sizeof(T) * i`
  - `A[i]` is equivalent to `*(A + i)`
    - Adding *i* to a pointer moves forward *i* elements, not *i* bytes
    - C infers the element size from the pointer's type

Today: Implementing Vector-Add

- How to implement this function in IA32?

```
int * vector_add(int *a, int *b, int length) {
  int *result;
  int i;

  result = (int *) malloc(length * sizeof(int));
  for (i = 0; i < length; i++)
    result[i] = a[i] + b[i];

  return result;
}
```

- Note: This example works on Linux.
  - e.g. MacOS X introduces many additional complexities
    - Prepends an underscore _ on public symbol names
    - Must use relative/indirect references to external symbols
- If on a confusing platform, use **`gcc -S`** on test code to figure out what's needed

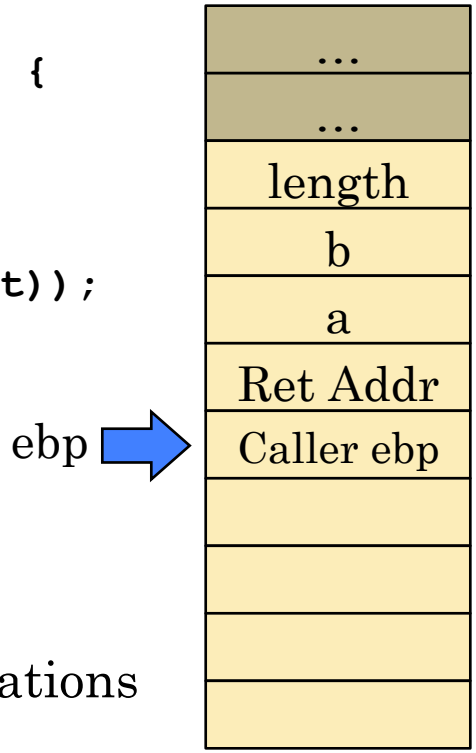
Vector-Add Arguments

- How to implement this function in IA32?

```
int * vector_add(int *a, int *b, int length) {
  int *result;
  int i;

  result = (int *) malloc(length * sizeof(int));
  for (i = 0; i < length; i++)
    result[i] = a[i] + b[i];

  return result;
}
```



- Using cdecl calling convention:
  - **a**, **b**, **length** pushed by caller; ebp + *offset* locations
  - Arguments are pushed last-to-first…
  - **a** stored at: **8(%ebp)**
  - **b** stored at: **12(%ebp)**
  - **length** stored at: **16(%ebp)**

Implementing Vector-Add (1)

- Initial code:
  - Put in placeholders where implementation is unknown

```
vector_add:
    # a      =  8(%ebp)
    # b      = 12(%ebp)
    # length = 16(%ebp)
    pushl   %ebp         # Save caller ebp
    movl    %esp, %ebp   # Set up stack frame ptr

    # TODO:  Save callee-save registers we alter

    # TODO:  Allocate space for result with malloc

    # TODO:  Implement vector-sum loop

    # TODO:  Restore callee-save registers we alter

    # Clean up stack before returning to caller
    movl    %ebp, %esp   # Remove local vars
    popl    %ebp         # Restore caller ebp
    ret
```

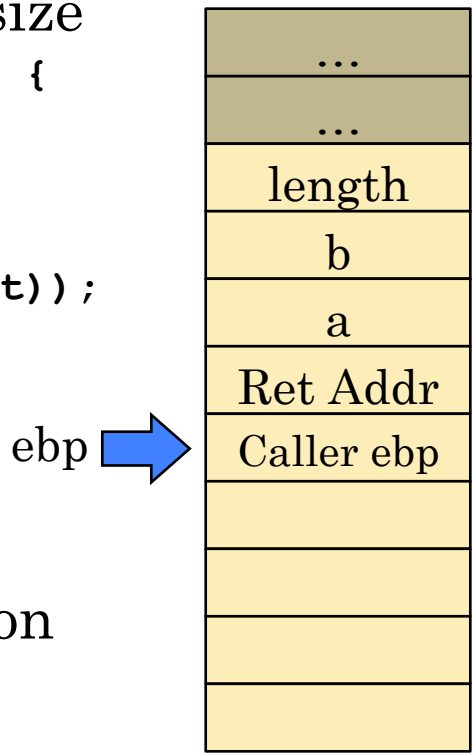
Calling Malloc

- Next task is to call **`malloc()`** with array size

```
int * vector_add(int *a, int *b, int length) {
  int *result;
  int i;

  result = (int *) malloc(length * sizeof(int));
  for (i = 0; i < length; i++)
    result[i] = a[i] + b[i];

  return result;
}
```



- **`malloc()`** also uses cdecl calling convention
  - Push **`length`** * 4 onto stack
  - Call **`malloc()`** subroutine
  - Memory address will be returned in **`eax`**
- We want to return this memory, so leave **`malloc()`**'s result in **`eax`** and return it to our caller as well

Implementing Vector-Add (2)

- Continuing our code:

```
vector_add:
    # a      =  8(%ebp)
    # b      = 12(%ebp)
    # length = 16(%ebp)
    ...

    # Allocate space for result with malloc
    movl    16(%ebp), %ecx
    shll    $2, %ecx     # ecx = length * 4
    pushl   %ecx
    call    malloc       # allocate space for result
    popl    %ecx         # cdecl:  caller removes args

    # TODO:  Implement vector-sum loop
```

- We return this memory to caller; leave pointer in **eax**

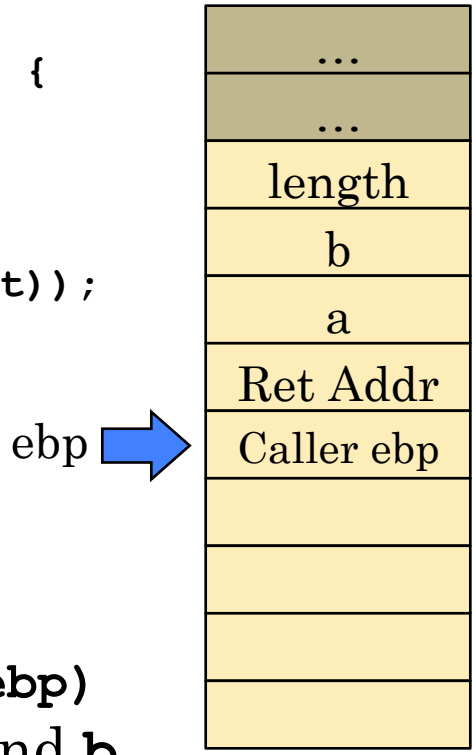
Implementing Our Loop

- Next, implement our loop!

```
int * vector_add(int *a, int *b, int length) {
  int *result;
  int i;

  result = (int *) malloc(length * sizeof(int));
  for (i = 0; i < length; i++)
    result[i] = a[i] + b[i];

  return result;
}
```



- Need a loop variable
  - Use **esi**. Compare directly to **length**, **16(%ebp)**
- Also need to access element *i* of arrays **a** and **b**
  - Use scaled, indexed memory addressing
  - **(R_b, R_i, s)** syntax accesses $M[R_b + R_i \times s]$
  - Need to store start-addresses of **a**, **b** into registers too

Implementing Vector-Add (3)

- Continuing our code:

```
vector_add:
    # a      =  8(%ebp)
    # b      = 12(%ebp)
    # length = 16(%ebp)
    ...

    # Implement vector-sum loop
    movl    8(%ebp), %ebx     # Start of a
    movl    12(%ebp), %ecx    # Start of b
    movl    $0, %esi          # Loop variable i
```

- Now we need to implement our **`for`**-loop
  - **`for`**-loop = **`while`**-loop + initialization and update
  - Turn **`while`**-loop into **`do`**-loop by factoring out first test

Implementing Vector-Add (4)

- Continuing our code:

```
        ...
        # Implement vector-sum loop
        movl    8(%ebp), %ebx       # Start of a
        movl    12(%ebp), %ecx      # Start of b
        movl    $0, %esi            # Loop variable i

        cmpl    16(%ebp), %esi      # First for-test
        jge     vadd_done           #   Is i >= length ?

vadd_loop:
        movl    (%ebx, %esi, 4), %edx   # edx  = a[i]
        addl    (%ecx, %esi, 4), %edx   # edx += b[i]
        movl    %edx, (%eax, %esi, 4)   # r[i] = edx

        incl    %esi                # i++
        cmpl    16(%ebp), %esi      # Subsequent for-tests
        jl      vadd_loop           #   Is i < length ?
vadd_done:
```

Loop Conditions

- AT&T syntax is really confusing for conditions
- C code: **`for (i = 0; i < length; i++)`**
- Assembly code:

```
        ...
        cmpl    16(%ebp), %esi      # First for-test
        jge     vadd_done           #   Is i >= length ?

vadd_loop:
        movl    (%ebx, %esi, 4), %edx   # edx  = a[i]
        addl    (%ecx, %esi, 4), %edx   # edx += b[i]
        movl    %edx, (%eax, %esi, 4)   # r[i] = edx

        incl    %esi                # i++
        cmpl    16(%ebp), %esi      # Subsequent for-tests
        jl      vadd_loop           #   Is i < length ?
vadd_done:
```

- **`cmpl Src2, Src1`** sets flags as for Src1 – Src2
  - For A *op* B, generally want to specify **`cmp B, A`**
  - This way, the conditional jump's type mirrors the C code

Finishing Up the Implementation

- Just need to save/restore our callee-save registers
- Code:

```
vector_add:
    # a      =  8(%ebp)
    # b      = 12(%ebp)
    # length = 16(%ebp)
    pushl   %ebp          # Save caller ebp
    movl    %esp, %ebp    # Set up stack frame ptr

    # Save callee-save registers.
    pushl   %ebx
    pushl   %esi

    ... (rest of implementation)

    # Restore callee-save registers.
    popl    %esi
    popl    %ebx

    # Clean up stack before returning to caller
    movl    %ebp, %esp    # Remove local vars
    popl    %ebp          # Restore caller ebp
    ret
```

Using Our Vector-Add Code

- Save assembly code into **`vector_add.s`**
- To make function callable, need to put this at top:

```
# Function to add two vectors together.
# (bla bla bla)
.globl vector_add
vector_add:
    ...
```

  - **`.globl`** makes the symbol visible to other functions
  - Many other directives you can add, but this is minimal to make the function callable.
- Assemble our function into an object file:
  - **`as -o vector_add.o vector_add.s`**
  - Can also include debug info by adding **`-g`** to args

Using Our Vector-Add Code (2)

- A simple test program, **va_main.c**:

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

/* Normally would declare this in file vector_add.h */
int * vector_add(int *a, int *b, int length);

int main() {
    int a[] = {1, 2, 3, 4, 5};
    int b[] = {6, 7, 8, 9, 10};
    int *res;
    int i;

    res = vector_add(a, b, 5);
    for (i = 0; i < 5; i++)
        printf("res[%d] = %d\n", i, res[i]);

    free(res);
    return 0;
}
```

Using Our Vector-Add Code (2)

- Compile **va_main.c**, link with **vector_add.o**

```
gcc -c va_main.c
gcc -o vadd va_main.o vector_add.o
```

- Run the program:

```
[user@host:~]> ./vadd
res[0] = 7
res[1] = 9
res[2] = 11
res[3] = 13
res[4] = 15
```

- Success! ☺
  - And if not, **gdb** is always available for debugging…

Vector-Add Function

- Using scaled, indexed memory access mode, very easy to implement array indexing in IA32
  - Once array's base address and index are in registers, very easy to retrieve/store a particular element
- Can also call C runtime functions from assembly
  - Called **`malloc()`** function by adhering to calling conventions
- Returned a heap-allocated chunk of memory to our caller
  - Now our functions can return results larger than **`eax`**
  - Results also last longer than a single procedure call since they aren't allocated on the stack
  - The program has to free the memory when it's done…

Finding Primes

- Our other program to return primes:
  - Write a function that takes an argument n
  - Returns a collection of all prime numbers $\leq n$
  - e.g. **`find_primes(int n)`**
- Size of result depends on inputs
- Need to build up result as we go
- Might be better to use a linked-list structure:

```
typedef struct IntList {
  int value;              /* Value for this node  */
  struct IntList *next; /* Pointer to next node */
} IntList;

IntList * find_primes(int n);
```

Finding Primes (2)

- A simple algorithm to generate result:

  for *i* = 2 **to** *n* **do**

  if *is_prime*(*i*) **then**

  Append new IntList node for *i* onto result.

  done

  return result

- Simple to implement, except possibly list part…
- To keep track of list pointers:

```
IntList *result = NULL;
IntList *last = NULL;
```

- Use **`malloc()`** to allocate new nodes for list

Finding Primes (3)

- To create and append new list nodes:

```
IntList *new;
...
if (is_prime(i)) {
    /* Allocate new node. */
    new = (IntList *) malloc(sizeof(IntList));
    new->value = i;
    new->next = NULL;

    /* Append new node onto existing list. */
    if (last != NULL)
        last->next = new;
    else
        result = new;

    /* Get ready to append next new node. */
    last = new;
}
```

Prime Number Linked-List

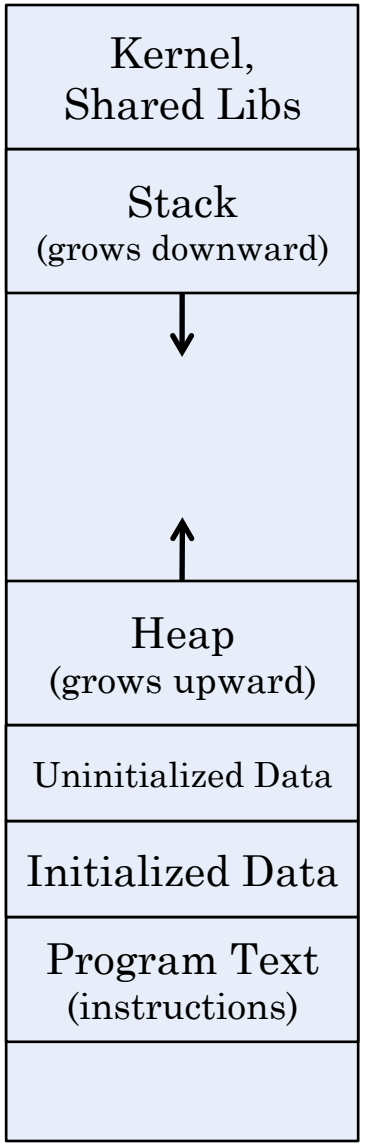
- There is no way to construct such a data structure on the stack!
  - Simply doesn't match the usage pattern for the stack
- Introduces another reason to use memory heap!
- Previous reasons:
  - Size of result depends on value of input(s)
  - Need result to live beyond lifetime of the procedure
- New reason:
  - Data structure simply cannot be represented on stack

Process Memory Layout

- Where do the stack and heap actually live?
- Where does the program itself live?
- Every program is laid out in memory following a specific pattern
  - Memory regions devoted to stack, heap, instructions, constants, global variables, etc.
- Some of these regions vary in size from program to program
  - Instruction data depends on program size
  - Program may not have global variables
  - Compiler determines these details at compile-time
- Some regions depend on program behavior
  - e.g. heap usage or stack usage is a run-time behavior

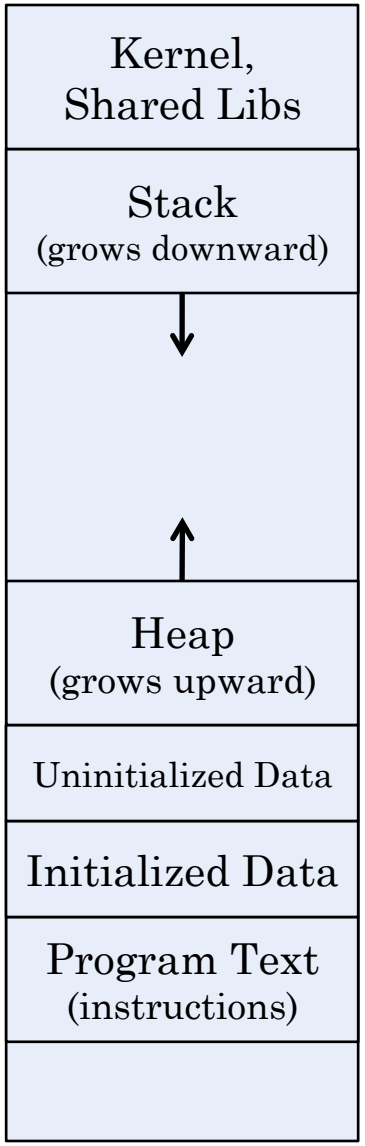
Process Memory Layout (2)

- A common memory layout:
- Program text (instructions), globals, and constants are placed by compiler
  - Reside toward bottom of address space
- Stack resides near top of address space
  - Grows downward as space is consumed
- Heap resides above fixed-size program data
  - Grows upward as space is consumed
- Arrangement gives stack and heap maximum room to grow



Process Memory Layout (3)

- Variable-size memory areas have soft and hard limits
  - When soft limit is hit, the region is extended, if possible
  - When hard limit is hit, the program is aborted by the operating system
- Book mentions **`brk`** value (§9.9)
  - "Break" address, where the program's memory heap ends
  - **`sbrk()`** ("set break") requests more heap space
- Also see **`getrlimit()`**/**`setrlimit()`** functions
  - Gets/sets limits on **`brk`**
  - Also limits on stack size, etc.



Stack and Heap Management

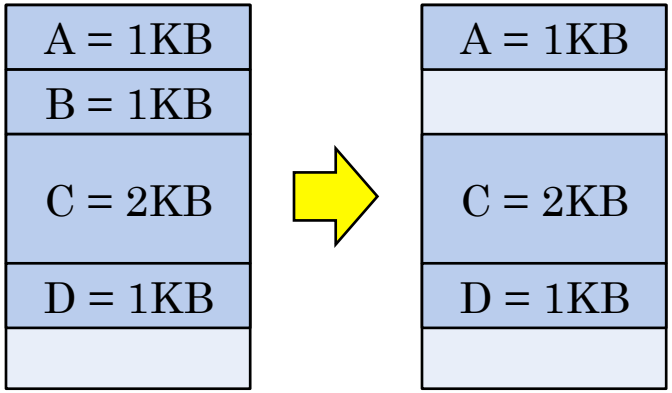
- Stack is relatively easy to manage
  - State consists of memory region + stack pointer
- Heap is *much* more complex!
  - Programs allocate and free blocks of memory, depending on their own needs
  - Heap must keep track of what memory is currently used, and what memory is available for use
  - Data structures for managing heap memory are necessarily complex
- Heap memory is managed by a heap allocator
  - Receives and attempts to satisfy allocation requests from a program
  - Manages these data structures internal to the heap

Heap Allocators

- Two kinds of heap allocators
  - Explicit allocators require applications to manually release memory when finished
    - e.g. C **`malloc()`**/**`free()`**, C++ **`new`**/**`delete`**
  - Implicit allocators detect when a memory region is no longer used
    - Employ garbage collectors for finding unused regions
    - e.g. Python, Java, Scheme, …
- Today: begin looking at explicit heap allocators
- You will get to write one in HW3! ☺

Allocation and Fragmentation

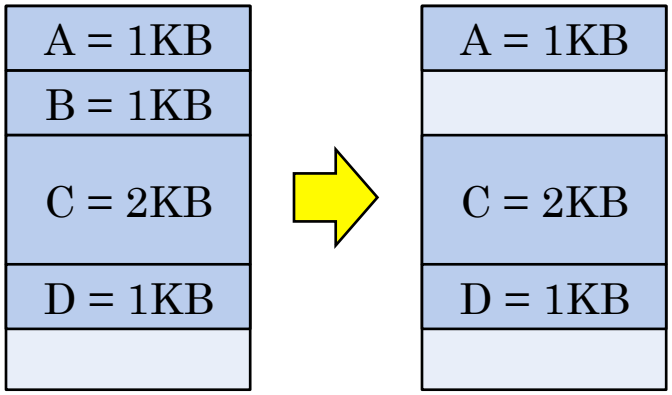
- Allocators must handle any sequence of allocation requests
  - May affect ability to satisfy requests
- Example: heap with 6KB total space
  - A = allocate 1KB
  - B = allocate 1KB
  - C = allocate 2KB
  - D = allocate 1KB
  - Free B



- Try to allocate 2KB?
  - 2KB of memory is *available*, but it's not contiguous!
- Heap memory can become <u>fragmented</u> from use
  - Allocator must minimize occurrence of fragmentation…
  - …but this also depends heavily on program's behavior!

Allocation and Fragmentation (2)

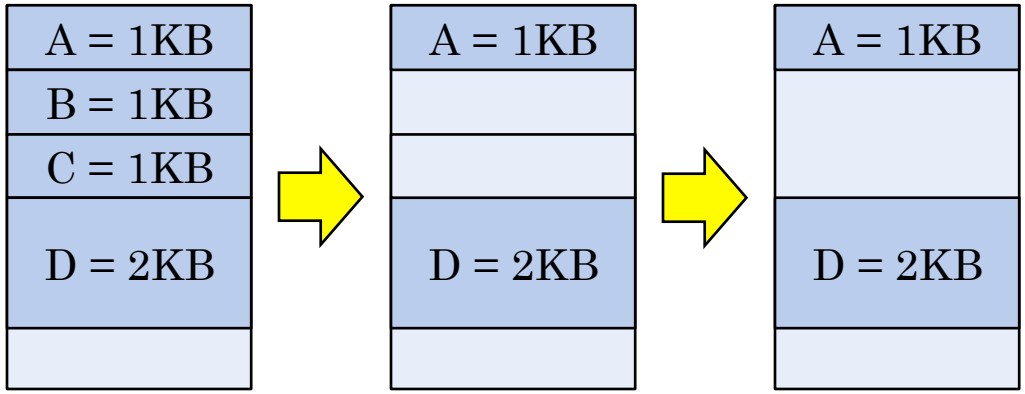
- Allocators cannot modify a memory block once in use
  - e.g. to move it to another part of memory
- Our example:
  - A = allocate 1KB
  - B = allocate 1KB
  - C = allocate 2KB
  - D = allocate 1KB
  - Free B
  - Try to allocate 2KB?



- Cannot modify allocated blocks to compact free space
  - Program has pointers into the allocated region
  - Allocator simply cannot find and update those pointers!
- Allocators can only manipulate free blocks
  - Program isn't using the free blocks, so it doesn't care…

Coalescing Free Blocks

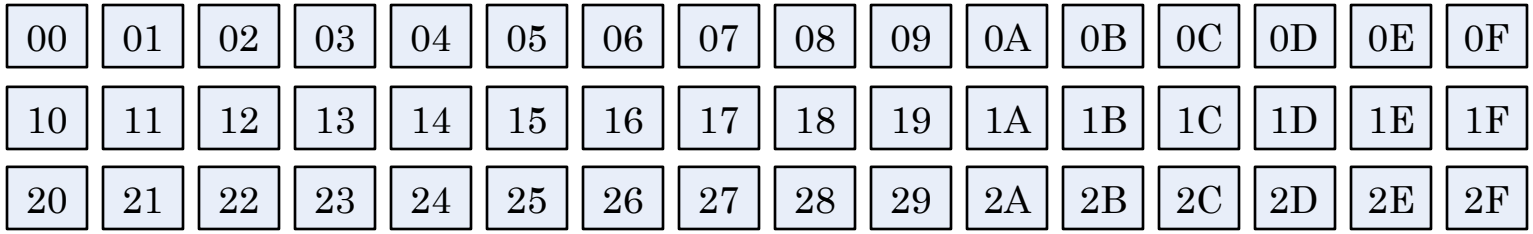
- Another sequence of allocations:
  - A = allocate 1KB
  - B = allocate 1KB
  - C = allocate 1KB
  - D = allocate 2KB
  - Free C
  - Free B
  - Try to allocate 2KB?



- Allocator needs to coalesce adjacent free blocks
  - Failure to do so leads to false fragmentation
  - Suitable free memory is available, but allocator can't find a region of the proper size to satisfy request
  - Ideally, coalescing free blocks will be *fast*

Data Alignment and Allocators

- Allocators often must care about data alignment
- So far, we have presented memory as an array of individually addressable bytes



- The processor may have a much larger data bus, e.g. $w = 32$ bits
  - Processor can read or write 32 bits of data at once
- To take full advantage of this, memory must also be able to read or write the same size data value

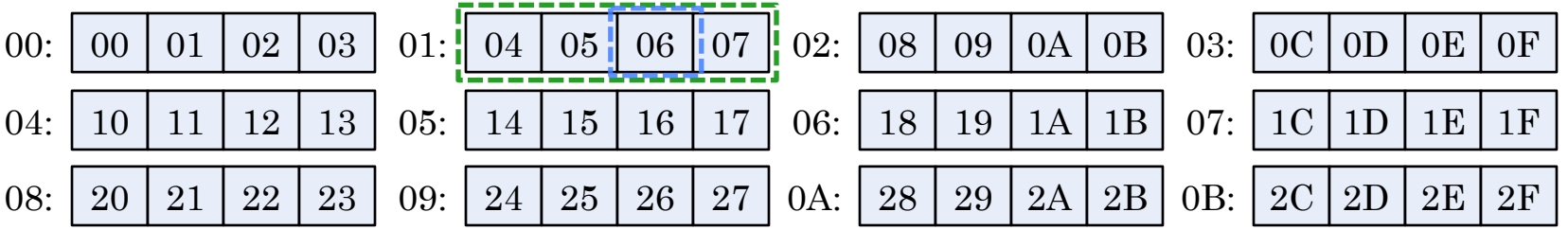
Data Alignment and Allocators (2)

- For e.g. 32-bit word size, memory looks like this:

  - Each addressable cell actually holds 4 bytes
  - Not possible to address individual bytes in each cell; must read or write the entire cell
  - (Memories often allow you to specify a mask saying which bytes in the cell you actually want to use…)
- The processor is able to present an abstraction that individual bytes are addressable…
  - …but, the CPU will still be interacting with memory in 32-bit words, not 8-bit bytes

Data Alignment and Allocators (3)

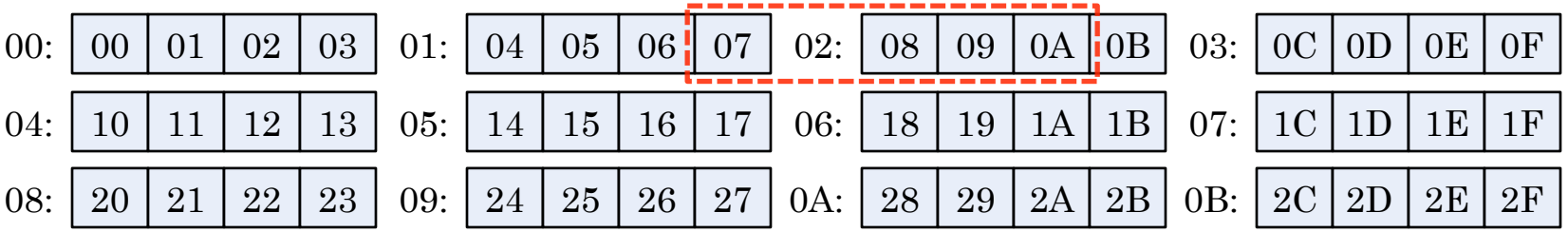
- For e.g. 32-bit word size, memory looks like this:



  - (Boxes contain byte-addresses; cell address is on left)
  - CPU presents an abstraction that each byte is individually addressable…
  - CPU maps the program's memory accesses to their actual addresses in physical memory
- Example: Write value 0xFF to byte-address 0x06
  - CPU writes 0x0000FF00 to memory address 1, and tells memory that only the 3rd byte should be written

Data Alignment and Allocators (4)

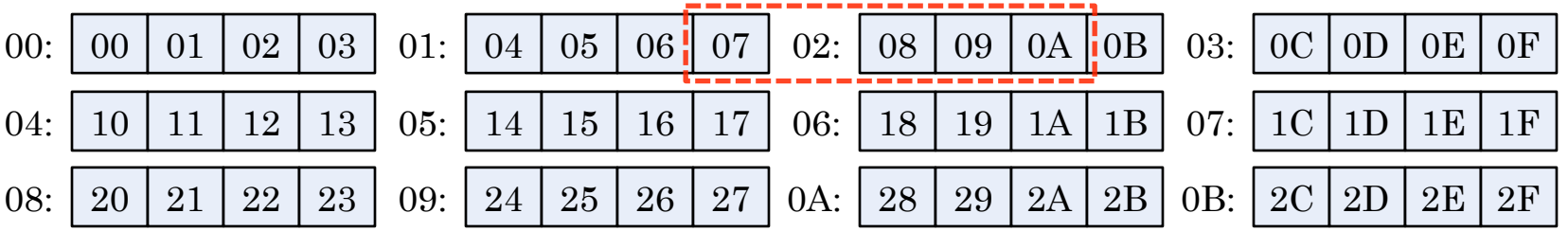
- For e.g. 32-bit word size, memory looks like this:



- What happens when a program tries to access a word that spans multiple memory cells?
  - e.g. read word starting at byte-address 7
- The CPU must perform two reads, then assemble the result into a single word
  - Read memory cell at address 1, keep only the 4th byte
  - Read memory cell at address 2, keep bottom 3 bytes
  - Assemble these into a single word

Data Alignment and Allocators (5)

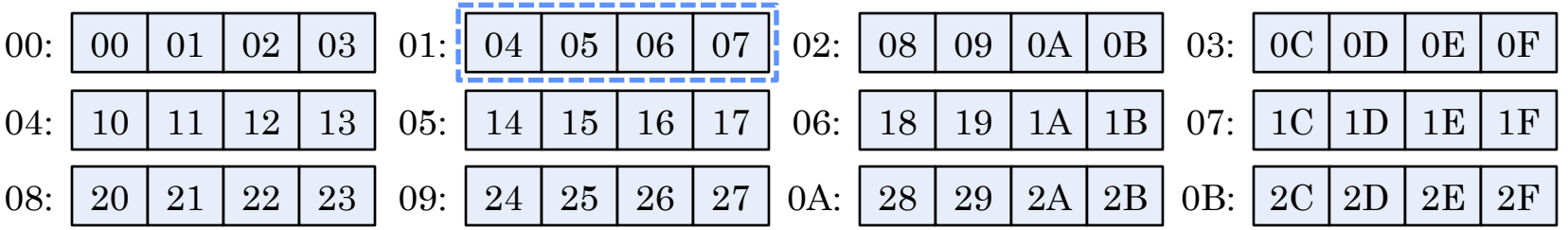
- For e.g. 32-bit word size, memory looks like this:



- This obviously complicates the CPU logic…
- Some CPUs simply disallow non-word-aligned memory accesses
  - If a program tries to perform such an access, the CPU reports an error
  - (IA32 supports non-word-aligned memory accesses…)
- *The compiler and the memory allocator must be aware of these data-alignment issues.*

Data Alignment and Allocators (6)

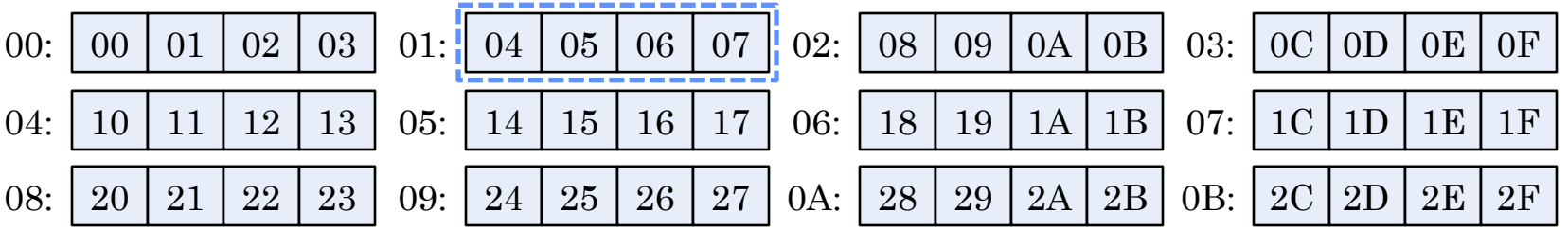
- For e.g. 32-bit word size, memory looks like this:



- Solution: the compiler positions all values so that they always start at word boundaries
  - e.g. on a 32-bit system, only store values at addresses that are evenly divisible by 4 bytes
- Programmers generally don't have to think about these word-alignment issues…
  - …but sometimes you can make your program *much* faster by knowing about them…

Data Alignment and Allocators (7)

- For e.g. 32-bit word size, memory looks like this:



- Similarly, allocators generally only hand out chunks of memory that are word-aligned
  - Request size is also rounded up to next word-aligned boundary
- Example: an allocator that keeps regions aligned on 8-byte boundaries, and a request for 53 bytes
  - Allocator returns block with address 0x00417E58 (bottom 3 bits of the address are 0), size is 56 bytes

Next Time

- More about how explicit allocators work
  - How do allocators track which memory regions are free and which are used?
  - How do allocators deal with fragmentation?
  - What approaches and data structures are used?