

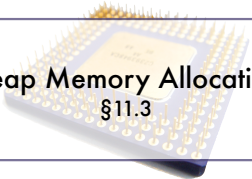


## Heap Memory Allocation & Memory Management §11.3 & Supplemental



## Homework/Administrivia

- ▶ **Floating Point:** You are only responsible for material from notes
  - ▶ Read §12.1–12.2 as necessary to understand that material; skip the parts we didn't cover
- ▶ **Read** §9.4.1–9.4.3 on two-dimensional arrays – *not covered in lecture*
- ▶ **Homework 6** – Due *this Sunday*, November 16, by 11:59 p.m. (sorry)
- ▶ **Exam 2 Bonus** next Friday, November 21, in class
  - ▶ Points possible: depends on your Exam 2 score. You can recover up to 30% of the points you missed. If you made  $\geq 90\%$ , it is worth 3 points.
  - ▶ Points possible =  $\max(3, (100 - \text{your score on Exam 2}) \times 0.3)$
  - ▶ Score will be *added* to your Exam 2 score. If you don't take it, your Exam 2 score stays the same.
  - ▶ Topics: procedures/stack frames and memory operands
  - ▶ Only about half the length of a normal exam
  - ▶ Closed-book, closed-notes; no make-ups given after November 21 (only in advance)



## Heap Memory Allocation §11.3



## Static vs. Dynamic Memory Allocation

- ▶ **Statically allocated memory** is reserved when the program is compiled/assembled
  - ▶ Assembly: `.data` section (among others) is put directly into `.exe` file and loaded as-is
  - ▶ C++: global variables (among other things)
- ▶ **Dynamically allocated memory** is allocated while the program runs, as needed
  - ▶ C++/Java: `new Something()`
  - ▶ Assembly: need to call a Windows API function (`HeapAlloc`)
- ▶ Static allocations have a fixed size; dynamic allocation sizes are specified at runtime
- ▶ Q. Why is dynamically allocated memory necessary?



## Dynamic Allocations: Stack vs. Heap

- ▶ In C++/Java:
  - ▶ Global variables are allocated statically
  - ▶ Local variables are allocated dynamically on the *stack* (remember `enter n, 0?`)
  - ▶ Objects created using `new Something()` are allocated dynamically on the *heap*
- ▶ The runtime stack is a fixed size
  - ▶ In MASM, you can use a directive like `.stack 4096` to change it
  - ▶ If you try to push too much onto the stack/too many recursive calls, stack overflow
- ▶ The amount of heap memory is not fixed – ask Windows to give you more, and it will try
  - ▶ A **heap** is a memory pool for a specific process
    - ▶ Has nothing to do with heap data structures



## How to Allocate Heap Memory

- ▶ Requires a sequence of two Windows API calls:
- ▶ **1. call `GetProcessHeap`**
  - ▶ Returns (in EAX) a *handle* to the process's default heap
    - ▶ Processes can have several heaps. In subsequent calls, you'll need to tell Windows which heap your memory is coming from. This *handle* is just an integer that uniquely identifies this heap.
- ▶ **2. push number of bytes to allocate**
  - push 0**
  - push the handle returned by `GetProcessHeap`**
  - call `HeapAlloc`**
  - ▶ Allocates the given number of bytes of memory
  - ▶ Returns (in EAX) the memory address of the allocated memory
- ▶ Both `GetProcessHeap` and `HeapAlloc` return 0 if an error occurs. *Check for this!*

## On Garbage Collection



- ▶ In Java, memory is *garbage collected* – it is automatically freed when possible

```
public void foo() {
    System.out.println(new Object()); // An Object is allocated and displayed.
    System.out.println("Done!");      // Now, the garbage collector can delete the object,
                                      // since it's no longer needed.
}
```
- ▶ In C++, memory is **not** garbage collected
  - ▶ If you use `new`, you must also use `delete` to free that memory
- ▶ **Memory allocated using *HeapAlloc* is not garbage collected**
  - ▶ You must use *HeapFree* to free that memory

## How to Free Heap Memory



- ▶ **1. call *GetProcessHeap***
  - ▶ Or if you already have a handle from a previous call, just use it
- ▶ **2. push the pointer returned by *HeapAlloc***  
**push 0**  
**push the handle returned by *GetProcessHeap***  
**call *HeapFree***
- ▶ *HeapFree* returns 0 if an error occurs. *Check for this!*

## Whiteboard Example



- ▶ The `exit` macro you have been using in your programs is actually shorthand for

```
push 0 ; This is your program's "exit code"
call ExitProcess
```
- ▶ *ExitProcess* is a Windows API function that terminates a process
- ▶ A nonzero exit code indicates that your program terminated due to an error
- ▶ **Q. Write a program that:**
  - ▶ Allocates 8 bytes of memory on the heap
  - ▶ Sets all 8 bytes to `FFh`
  - ▶ Frees the memory
  - ▶ Terminates with an exit code of 1 if an error occurs and 0 otherwise

## Windows Dev Center



Windows | Dev Center - Desktop

Dashboard Get started Design Develop Test

Desktop technologies Server and system API index Samples

▶ *HeapAlloc*, *HeapFree*, *ExitProcess*, *WriteConsole*... all of these Windows API functions are documented in the Windows Dev Center

- ▶ But you'll want a good book on Win32 programming if you begin to use these in real life...
- ▶ Also, in real life, you would do Win32 programming in C or C++, not assembly language

### HeapAlloc function

Allocates a block of memory from a heap. The allocated memory is not movable.

**Syntax**

```
LPVOID WINAPI HeapAlloc(
    _In_ HANDLE hHeap,
    _In_ DWORD dwFlags,
    _In_ SIZE_T dwBytes
);
```

**Parameters**

*hHeap* [in]

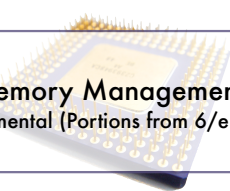
**ExitProcess function**

Ends the calling process and all its threads.

**Syntax**

```
VOID WINAPI ExitProcess(
    _In_ UINT uExitCode
);
```

## Memory Management Supplemental (Portions from 6/e §2.3)



## Real vs. Protected Mode



- ▶ Recall from a couple of months ago:
  - ▶ x86 processors can run in **real-address mode** or **protected mode**
  - ▶ Processors boot in real-address mode
    - ▶ For backward compatibility with the original 8086
  - ▶ MS-DOS used real-address mode
  - ▶ Windows, Linux, Mac OS X switch the processor to protected mode at startup

## Memory Addresses



- Surprise... not all memory addresses correspond to RAM storage!
- E.g., in real-address mode:
  - ROM BIOS (Basic Input-Output System) starts at address F0000h
  - Video memory starts at address A0000h
- Hardware can be *memory-mapped*
  - Hardware devices connected to the address/data bus
  - Intercept requests to read/write certain addresses
- BIOS can provide a (partial) *memory map*
  - E.g., identifies addresses corresponding to available RAM

[http://wiki.osdev.org/Detecting\\_Memory\\_\(x86\)](http://wiki.osdev.org/Detecting_Memory_(x86)) <http://www.urwik.org/orig-grub/mem64mb.html>

## Real-Address Mode (1)



- Real-address mode** (backward-compatible with the 8086)
  - Not designed for multitasking
    - Only one running program
  - 20-bit memory addresses
    - 00000h through FFFFFh
    - So only 1 MB of memory can be addressed
  - Programs can access any memory address
    - Including addresses corresponding to memory-mapped hardware
  - MS-DOS used real-address mode

## Real-Address Mode (2)



- Problem:** 8086 processor had 20-bit memory addresses but only 16-bit registers
- Solution:** *Segmented memory*
  - Segment register holds 16-bit *segment* value, 16-bit general-purpose register holds 16-bit *offset* value
    - Segment-offset address written as *segment:offset*
    - Recall: segment registers are CS, DS, ES, FS, GS
  - The actual (*linear* or *absolute*) memory address is  $segment \times 10h + offset$
  - 08F1:0100 corresponds to the linear address 09010h
  - 07FF:1020 also corresponds to linear 09010h

## Protected Mode (1)



- Protected mode**
  - Designed for multitasking
    - OS has more privileges than application programs (four *privilege rings*)
  - 32-bit memory addresses
    - 00000000h through FFFFFFFFh
    - So 4 GB of memory can be addressed
  - Each process is assigned its own area of memory
  - One process cannot access another process's memory
  - Applications cannot access memory-mapped hardware

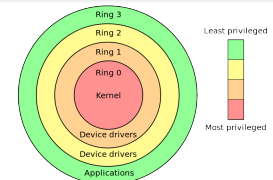


Image Source: [http://en.wikipedia.org/wiki/File:Priv\\_rings.svg](http://en.wikipedia.org/wiki/File:Priv_rings.svg)

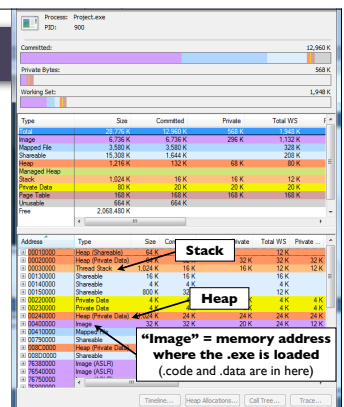
## Protected Mode (2)



- In (32-bit) protected mode, there are 32-bit memory addresses and 32-bit registers
- Modern operating systems use the **flat** segmentation model
  - Essentially, every segment starts at memory address 00000000h
  - So, 32-bit memory addresses exactly correspond to a linear address
  - This is why `mov eax, OFFSET f00` puts the memory address of `f00` into EAX:
    - The value stored into EAX is the number of bytes from the start of the data segment to `f00`
    - But the data segment starts at memory address 00000000h
    - So the offset *is* the linear memory address!
- In protected mode, segment registers used for a different purpose
  - Unless you're writing an operating system, you can basically ignore them

## Address Space

- Recall:
  - Executables (.exe files) contain data and text sections corresponding to .data and .code sections of an assembly language program
  - To run a program, the operating system loads the executable (.exe file) into memory and sets EIP to point to the first instruction; it also does other things, like reserve memory for the stack
  - Memory addresses are 32 bits; each memory address corresponds to one byte of memory
  - Think of memory as a giant, 4 GB array of bytes, indexed 00000000h through FFFFFFFFh
  - VMMMap (from [www.sysinternals.com](http://www.sysinternals.com)) can display the address space of a running process





- Does it seem weird that your .data section is almost *always* at memory address 00405000h? Like, that memory address is *always* available for your program to use?

## Protected Mode – Virtual Memory (1)

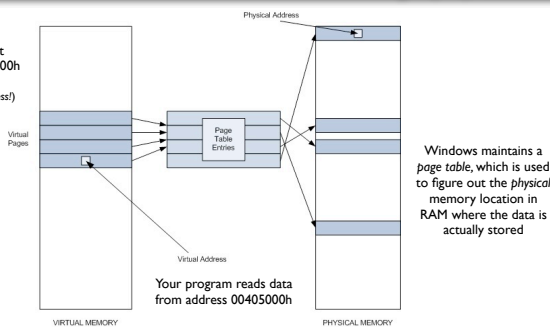


- In the flat memory model, 32-bit addresses correspond exactly to a linear address, but...
- The memory addresses you use in your assembly language program (like 00405000h) are *virtual* memory addresses! The *physical* address—where you'll find the data in RAM—is completely different.**
- Modern operating systems use *virtual memory* or *paging*
  - A *page table* is used to determine how linear addresses are mapped to physical addresses
    - Memory is divided into 4KB blocks called *pages*
  - Pages can be temporarily stored on disk; that memory can then be used by another process
    - Gives the illusion that the computer has more memory than the amount of physical RAM installed

## Protected Mode – Virtual Memory (2)



Windows tells you your .data section is at memory address 00405000h  
(That's actually a virtual address!)



## Virtual Address Space



- In Windows 32-bit operating systems...
- Each process (running program) has its own 32-bit virtual address space
  - Each process has 4 GB of virtual memory
- The bottom 2 GB is **user-mode space**
  - Addresses 00000000h–7FFFFFFFh
  - Your executable is loaded at address 00400000h
  - Data, stack, and heap are all in this address range
  - Code from EXEs and DLLs is marked read-only (why?)
- The top 2 GB is **system space** (or **kernel space**)
  - Addresses 80000000h–FFFFFFFFh
  - OS code is here; your program cannot read/write this space

