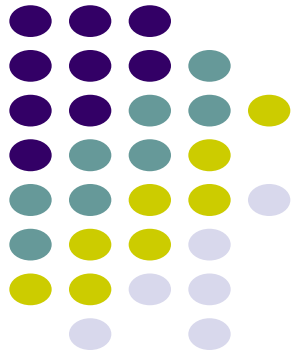
# Virtual Memory Background: Address Binding & Linking

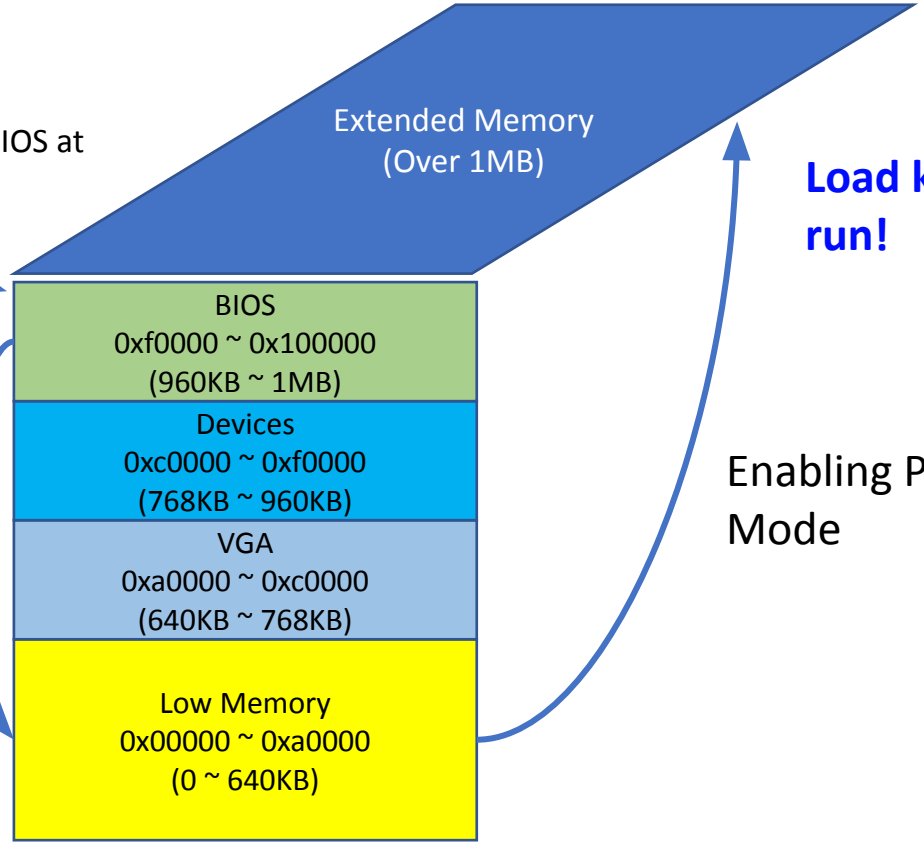
ECE 469, Jan 18

Aravind Machiry



# Booting


Map code in BIOS at f000:fff0




**Load kernel and run!**

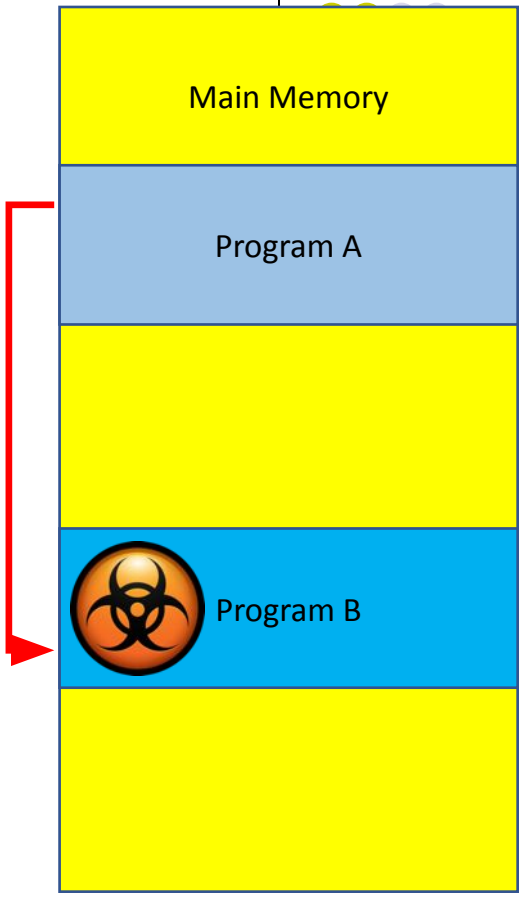
Enabling Protected Mode

Read Master Boot Record (MBR)
from the boot disk
and load it at 0x7c00

# Real mode

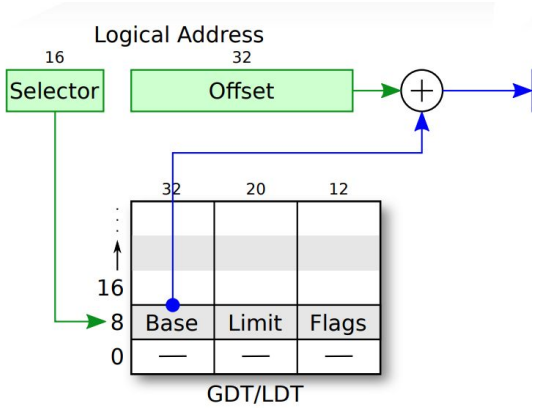
- Suppose two program runs at the same time
- Program A attempts to modify memory used by program B

No SECURITY!



# i386 Protected mode

- Look at GDT (Global Descriptor Table)
  - Indexed by a segment register

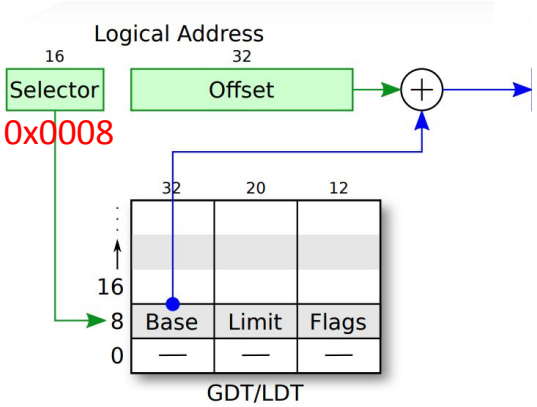


# i386 Protected mode

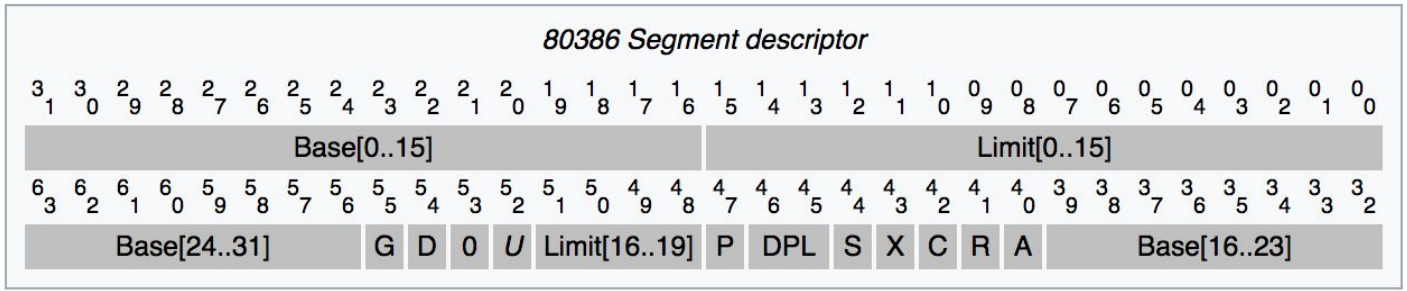
- Address 0x0008:0x00003400

- In the real mode:
  - 0x0008 * 16 + 0x3400 = 0x3480

- In the i386 protected mode
  - GDT[1].base + 0x3400
  - Access if 0x3400 is less than GDT[1].limit
  - Otherwise, raise an exception!

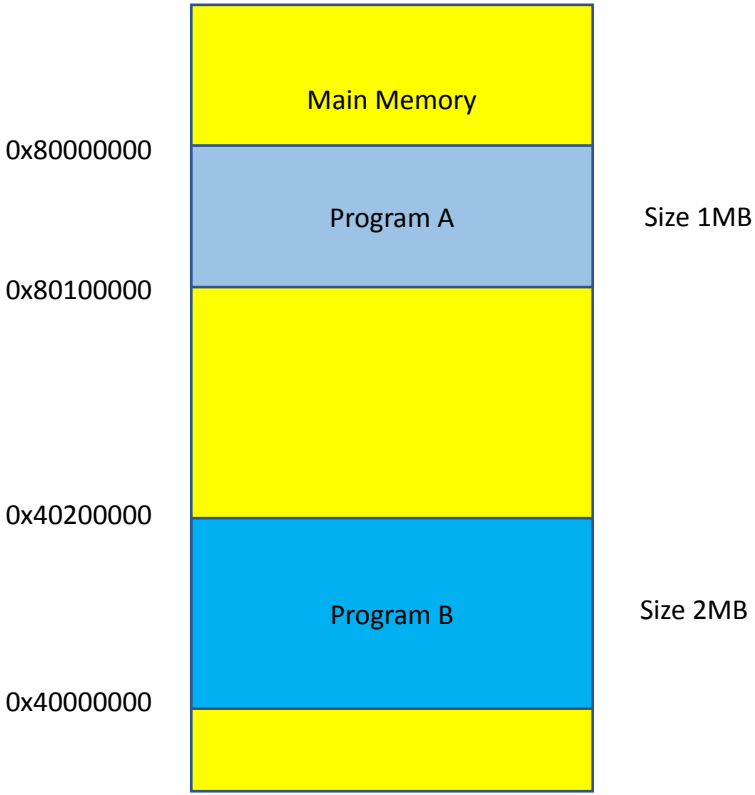


# i386 Protected mode




- G - Granularity (0 = byte, 1 = page)
  - 0: Limit will be byte granularity **(i.e., limit, only access $2^{20}$, 1MB)**
  - 1: Limit will be page granularity **(i.e., limit * 4096, $2^{20} * 2^{12} = 2^{32}$)**
- D – Default operand size (0 = 16-bit, 1 = 32-bit)
  - Set the values of IP/SP with respect to this bit
- R,X – Readable/Executable
- DPL – Descriptor Privilege Level (a.k.a. Ring Level)
  - 0 (highest priv), 1, 2, 3 (lowest priv)

# Segment Example




0x10:0 ~ 0x10:0x100000 are valid address for Program A

0x80000000 ~ 0x80100000

0x08:0 ~ 0x08:0x200000 are valid address for Program B

0x40000000 ~ 0x40200000

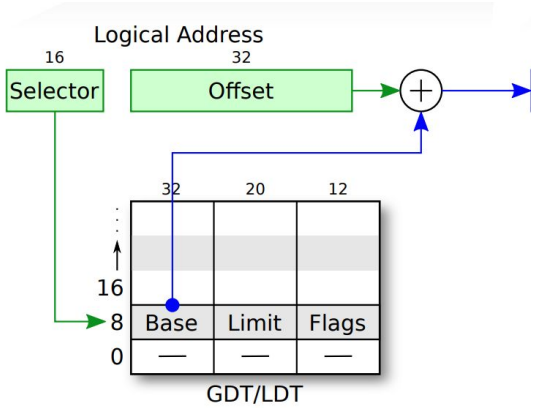
| GDT index | 32-bit Base | 20-bit Limit | 12-bit Flags |
|---|---|---|---|
| 16 | 0x80000000 | 0xfffff | G=0 |
| 8 | 0x40000000 | 0x00200 | G=1 |
| 0 | 0x0 | 0x0 | G=0 |

# Protected mode - Examples

- 0x8:0x8080
  - Base: 0x40000000
  - Limit (addr): 0x8000000 (**0x08000 * $2^{12}$**)
  - Offset: 0x8080

0x8080 < 0x8000000

Address: 0x40008080

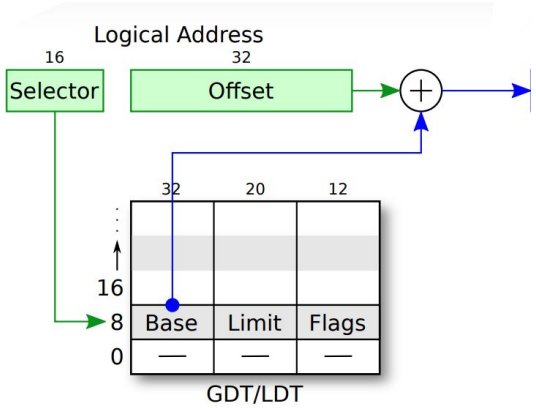



| GDT index | 32-bit Base | 20-bit Limit | 12-bit Flags |
|---|---|---|---|
| 16 | 0x31300000 | 0x1000 | G=0 |
| 8 | 0x40000000 | 0x08000 | **G=1** |
| 0 | 0x0 | 0x0 | G=0 |

# Protected mode - Examples

- 0x10:0x333
  - Base: 0x31300000
  - Limit (addr): 0x1000
  - Offset: 0x333

Address: 0x31310333

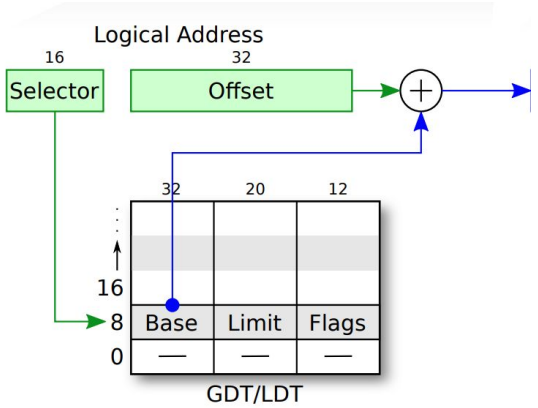



| GDT index | 32-bit Base | 20-bit Limit | 12-bit Flags |
|---|---|---|---|
| 16 | 0x31300000 | 0x1000 | G=0 |
| 8 | 0x40000000 | 0x08000 | **G=1** |
| 0 | 0x0 | 0x0 | G=0 |

# Protected mode - Examples

- 0x10:0x8080
  - Base: 0x31300000
  - Limit (addr): **0x1000**
  - Offset: **0x8080**

    Offset >= limit, Access denied!




| GDT index | 32-bit Base | 20-bit Limit | 12-bit Flags |
|---|---|---|---|
| 16 | 0x31300000 | 0x1000 | G=0 |
| 8 | 0x40000000 | 0x08000 | **G=1** |
| 0 | 0x0 | 0x0 | G=0 |

# Protected mode - Memory Privilege Levels

- DPL (Descriptor Privilege Level)

- Protected mode – four levels of memory privilege
  - 0 (00) – highest, OS kernel
  - 1 (01) – OS kernel

  ---

  - 2 (10) – highest user-level privilege
  - 3 (11) – user-level privilege

**Kernel: for privileged OS operations…**

**User: for unprivileged applications…**

*80386 Segment descriptor*

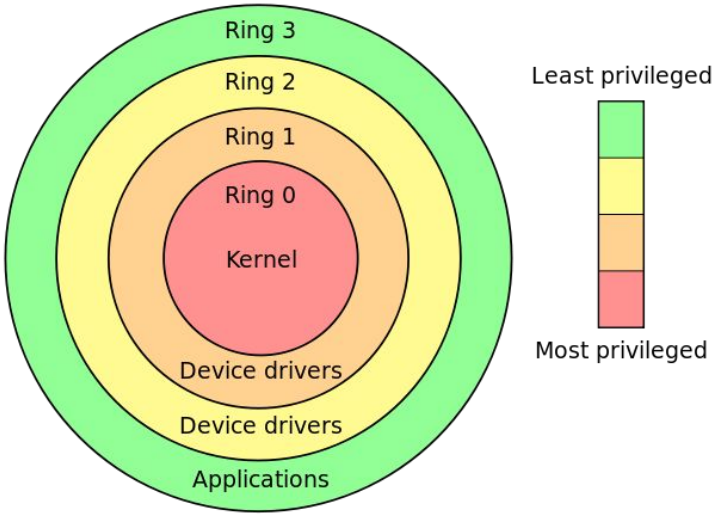
| 31 30 29 28 27 26 25 24 | 23 22 21 20 | 19 18 17 16 | 15 14 13 12 | 11 10 09 08 | 07 06 05 04 | 03 02 01 00 |
|---|---|---|---|---|---|---|
| Base[0..15] | | | Limit[0..15] | | | |
| 63 62 61 60 59 58 57 56 | 55 54 53 52 | 51 50 49 48 | 47 46 45 44 | 43 42 41 40 | 39 38 37 36 | 35 34 33 32 |
| Base[24..31] | G | D | 0 | *U* | Limit[16..19] | P | DPL | S | X | C | R | A | Base[16..23] |

# Protected mode - Memory Privilege Levels

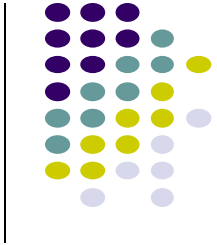
- DPL (Descriptor Privilege Level)
- Protected mode – four levels of memory privilege
  - 0 (00) – highest, OS kernel
  - 1 (01) – OS kernel

  ---

  - 2 (10) – highest user-level privilege
  - 3 (11) – user-level privilege
- Typically, 0 is for kernel, 3 is for user…

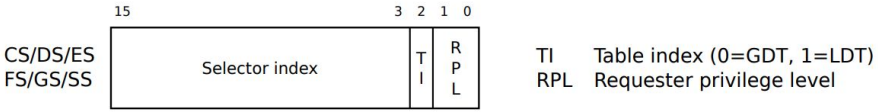


# DPL Defines Ring Level



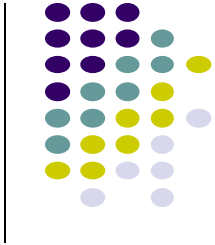
- CPL = Current Privilege Level
  - Defined in the last 2 bits of the %cs register
  - You can change %cs only via `lcall/ljmp/trap/int`

- Examples
  - %cs == 0x8 == 1000 in binary, last 2 bits are ZERO -> KERNEL!
  - %cs == 0x13 == 10011 in binary, last 2 bits are 3 -> USER!
  - %cs == 0x10 == 10000 in binary, last 2 bits are 0 -> KERNEL!
  - %cs == 0xb == 1011….




| GDT index | 32-bit Base | 20-bit Limit | 12-bit Flags |
|---|---|---|---|
| 16 **USER** | 0x31310000 | 0x1000 | G=0, **DPL=3** |
| 8 **KERNEL** | 0x40000000 | 0x80000 | G=1, **DPL=0** |
| 0 **KERNEL** | 0x0 | 0xfffff | G=1, **DPL=0** |

# DPL Defines Ring Level



- CPL = Current Privilege Level
  - Defined in the last 2 bits of the %cs register
  - You can change %cs only via `lcall/ljmp/trap/int`

- mov %ax, %cs □ **impossible!**

- Can only move down…
  - CPL==0, then ljmp 0x3:0x1234 is **OK to execute**
  - CPL==3, then ljmp 0x0:0x1234 is **not allowed**

| GDT index | 32-bit Base | 20-bit Limit | 12-bit Flags |
|---|---|---|---|
| 16 **USER** | 0x31310000 | 0x1000 | G=0, **DPL=3** |
| 8 **KERNEL** | 0x40000000 | 0x80000 | G=1, **DPL=0** |
| 0 **KERNEL** | 0x0 | 0xfffff | G=1, **DPL=0** |

# Ring 0 (Kernel) can go to Ring 3 (User)

- Then, how can we go back to kernel?
- We can switch from ring 0 to ring 3 via ljmp
  - ljmp 0x3:0x1234
- We cannot switch from ring 3 to ring 0 via ljmp
  - ljmp 0x0:0x1234 □ illegal instruction
- We use iret / sysexit / sysret to switch from ring 3 to ring 0
  - We will learn this in week 4

# Enabling Protected Mode: Create GDT

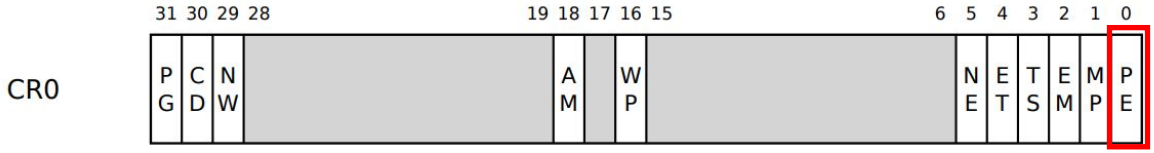
- In boot/boot.S
  - %cs to point 0 ~ 0xffffffff in DPL 0
  - %ds to point 0 ~ 0xffffffff in DPL 0

- Only kernel can access those two segment

```
# Bootstrap GDT
.p2align 2                                # force 4
gdt:
  SEG_NULL                # null seg
  SEG(STA_X|STA_R, 0x0, 0xffffffff) # code seg
  SEG(STA_W, 0x0, 0xffffffff)           # data seg
.set PROT_MODE_CSEG, 0x8          # kernel code segment selector
.set PROT_MODE_DSEG, 0x10         # kernel data segment selector
```

| GDT index | 32-bit Base | 20-bit Limit | 12-bit Flags |
|---|---|---|---|
| 16 | 0x0 | 0xfffff | G=1,W **DPL=0** |
| 8 | 0x0 | 0xfffff | G=1, XR **DPL=0** |
| 0 | 0 | 0 | 0 |

# Enabling Protected Mode: Change CR0




| | | | | | |
|---|---|---|---|---|---|
| PE | Protection enabled | ET | Extension type | NW | Not write-through |
| MP | Monitor coprocessor | NE | Numeric error | CD | Cache disable |
| EM | Emulation | WP | Write protect | PG | Paging |
| TS | Task switched | AM | Alignment mask | | |

Set PE (Protected enabled) to **1** will enable Protected Mode

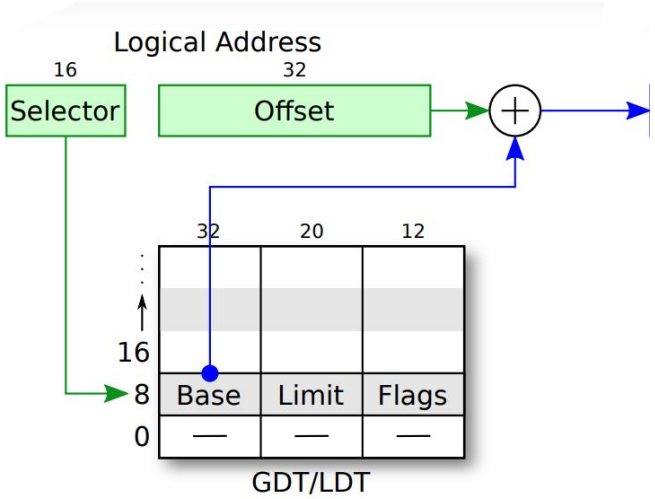
```
lgdt    gdtdesc
movl    %cr0, %eax
orl     $CR0_PE_ON, %eax
movl    %eax, %cr0
```

1. Load GDT
2. Read CR0, store it to eax
3. Set PE_ON (1) on eax
4. Put eax back to CR0

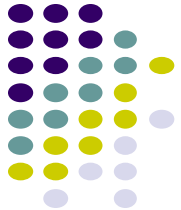
(PE_ON to CR0!!)

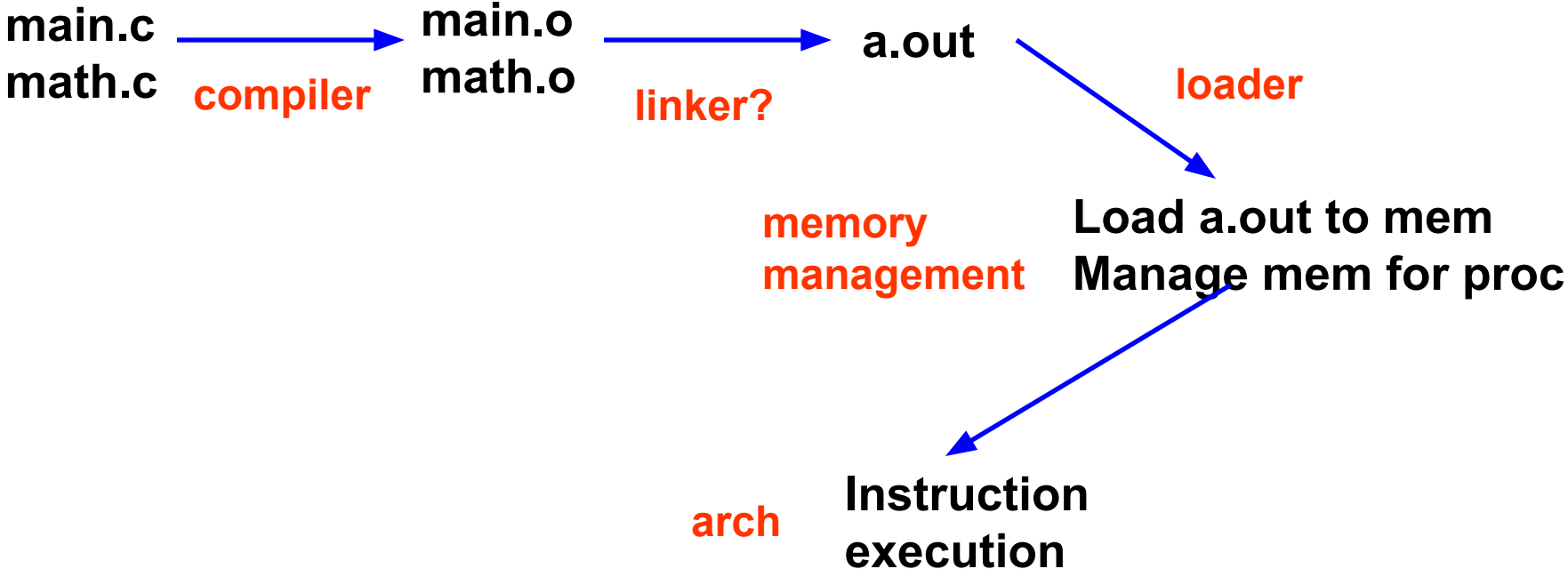
# Protected Mode Summary

- Segment access via GDT
  - Base + Offset if Offset < Limit * 4096 (if G == 1)
  - Base + Offset if Offset < Limit (if G == 0)
- Last two bits in %cs - CPL
  - Memory Privilege - Ring level
  - 0 for OS kernel
  - 3 for user application
- Changing CR0 to enable protected mode
  - CR0_PE_ON == 1, set via eax
- Changing CPL?
  - ljmp %cs:xxxxx, set the last 2 bits of %cs as 0 for kernel, 3 for user



# A gap among Architecture, Compiler and OS courses





# Example

*Main.c*:

```
extern float sin( );
main( )
{
  static float x, val;


  printf(“Type number: ”);
  scanf(“%f”, &x);
  val = sin(x);
  printf(“Sine is %f”, val);
}
```

*Math.c*:

```
float sin(float x)
{
  static float temp1, temp2, result;

  – Calculate Sine –

  return result ;
}
```

# Example (cont)

- Main.c uses externally defined sin() and C library function calls
  - printf()
  - scanf()

- How does this program get compiled and linked?

# Compiler

- Compiler: generates object file
  - Information is incomplete
  - Each file may refer to symbols defined in other files

# Components of Object File

- Header
- Two segments
  - Code segment and data segment
  - OS adds empty heap/stack segment while loading
- Size and address of each segment
  - Address of a segment is the address where the segment begins.

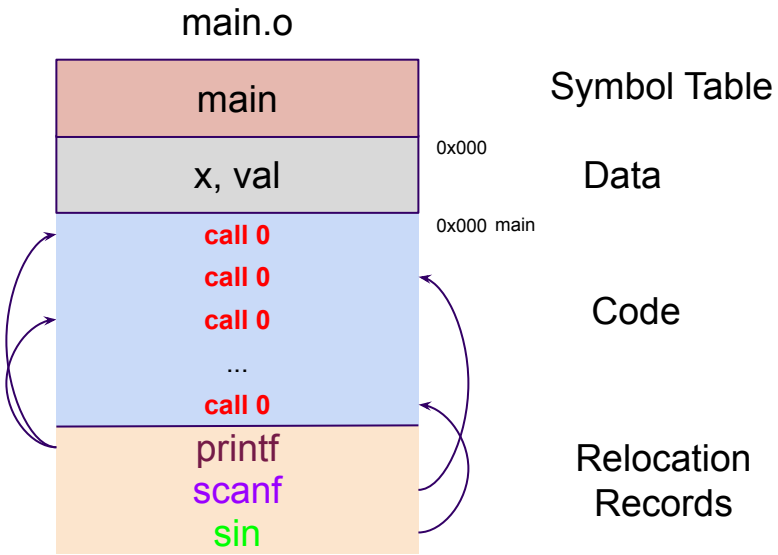
# Components of Object File (cont)

- Symbol table
  - Information about stuff defined in this module
  - Used for getting from the name of a thing (subroutine/variable) to the thing itself
- Relocation information
  - Information about addresses in this module linker should fix
    - External references (e.g. lib call)
    - Internal references (e.g. absolute jumps)
- Additional information for debugger

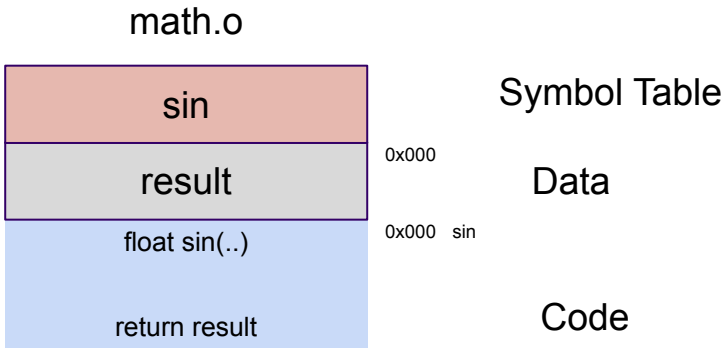
# What could the compiler not do?

- Compiler does not know final memory layout
  - It assumes everything in .o starts at address zero
  - For each .o file, compiler puts information in the symbol table to tell the linker how to rearrange outside references safely/efficiently
    - For exported functions, absolute jumps, etc

# Compiler: main.c



# Compiler: math.c



# Linker functionality

- Three functions of a linker
  - Collect all the pieces of a program
  - Figure out new memory organization
    - Combine like segments
    - Does the ordering matter? (spatial locality for cache)
  - Touch-up addresses

- The result is a runnable object file (e.g. a.out)

# Linker – a closer look

- Linker can shuffle segments around at will, but cannot rearrange information within a segment

# Linker requires at least two passes

- Pass 1: decide how to arrange memory
- Pass 2: address touch-up

# Pass 1 – Segment Relocation

- Pass 1 assigns input segment locations to fill-up output segments
  - Read and adjust symbol table information
  - Read relocation info to see what additional stuff from libraries is required

# Pass 2 – Address translation

- In pass 2, linker reads segment and relocation information from files, fixes up addresses, and writes a new object file

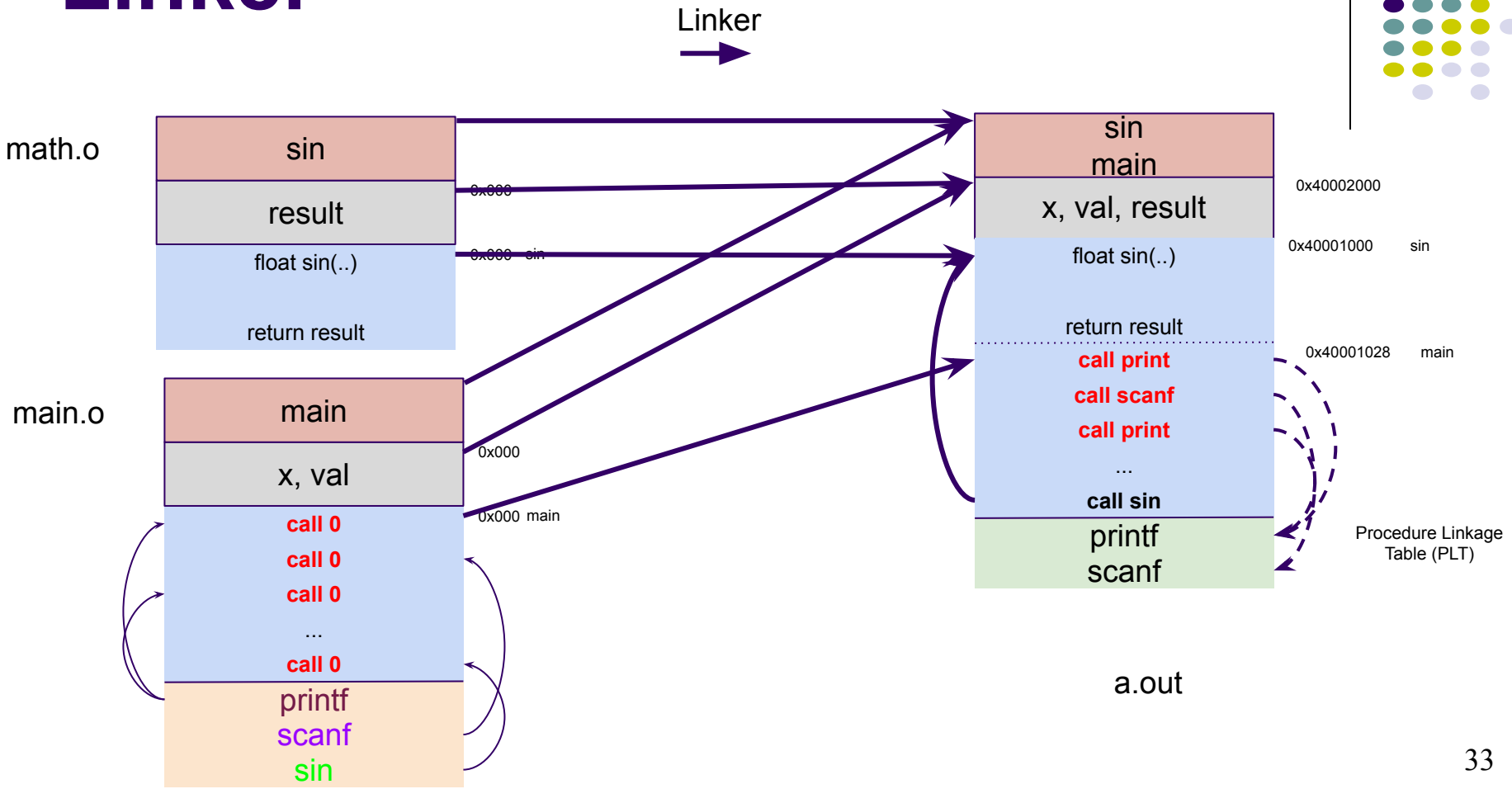
- Relocation information is crucial for this part

# Putting It Together

- Pass 1:
  - Read symbol table, relocation table
  - Rearrange segments, adjust symbol table

- Pass 2:
  - Read segments and relocation information
  - Touch-up addresses
  - Write new object file

# Linker


Linker
math.o
sin
result
float sin(..)
return result
0x000
0x000 sin
main.o
main
x, val
call 0
call 0
call 0
...
call 0
printf
scanf
sin
0x000
0x000 main
sin
main
x, val, result
float sin(..)
return result
call print
call scanf
call print
...
call sin
printf
scanf
0x40002000
0x40001000 sin
0x40001028 main
Procedure Linkage Table (PLT)
a.out

# Dynamic linking

- Static linking – each lib copied into each binary
- Dynamic linking:
  - Instead of system call wrapper code, a stub that finds lib code in memory, or loads it if it is not present
- Pros:
  - all procs can share copy (shared libraries)
    - Standard C library
  - live updates

# Dynamic loading

- Program can call dynamic linker via
  - dlopen()
  - library is loaded at running time
- Pros:
  - More flexibility -- A running program can
    - create a new program
    - invoke the compiler
    - invoke the linker
    - load it!

# Memory Usage Classification

- Memory required by a program can be used in various ways
- Some possible classifications
  - Role in programming language
  - Changeability
  - Address vs. data
  - Binding time

# Role in Programming Language

- Instructions
  - Specify the operations to be performed on the operands
- Variables
  - Store the information that changes as program runs
- Constants
  - Used as operands but never change

# Changeability

- Read-only
  - Example: code, constants
- Read and write
  - Example: Variables

# Address vs. Data

- Need to distinguish between addresses and data
- Why?
  - Addresses need to be modified if the memory is re-arranged

# Binding Time

- When is the space allocated?
  - Compile-time, link-time, or load-time
  - Static: arrangement determined once and for all
  - Dynamic: arrangement cannot be determined until runtime, and may change
    - malloc( ), free( )

# Classification – summary

- Classifications overlap
  - Variables may be static or dynamic
  - Code may be read-only or read and write
    - Read-only: Solaris
    - Read and write: DOS
- So what is this all about?
- What does memory look like when a process is running?

# Memory Layout

- Memory divided into segments
  - Code (called text in Unix terminology)
  - Data
  - Stack

- Why different segments?
  - To enforce classification
  - e.g. code and data treated differently at hardware level

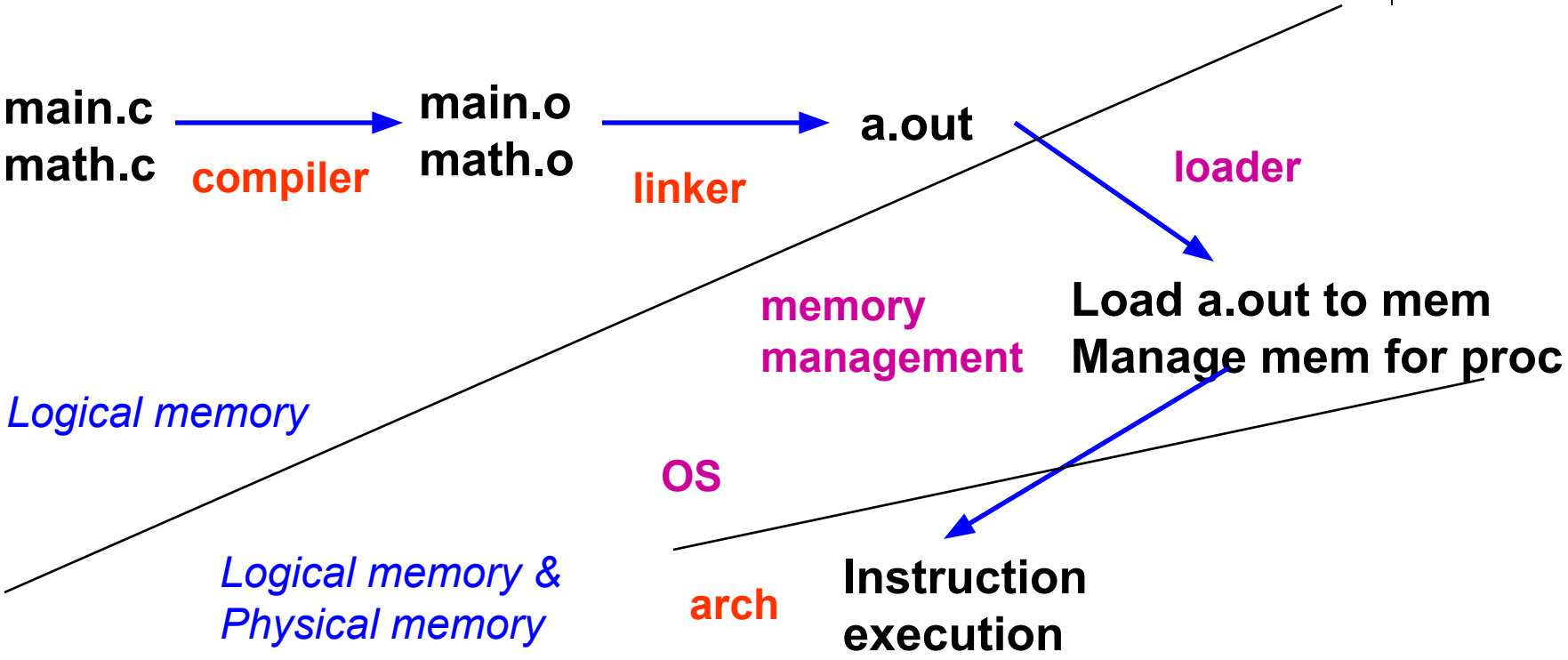
# The big picture

- a.out needs address space for
  - text seg, data seg, and (hypothetical) heap, stack
- A running process needs phy. memory for
  - text seg, data seg, heap, stack
- But no way of knowing where in phy mem at
  - Programming time, compile time, linking time
- Best way out?
  - Make agreement to divide responsibility
    - Assume address starts at 0 at prog/compile/link time
    - OS needs to work hard at loading/runing time

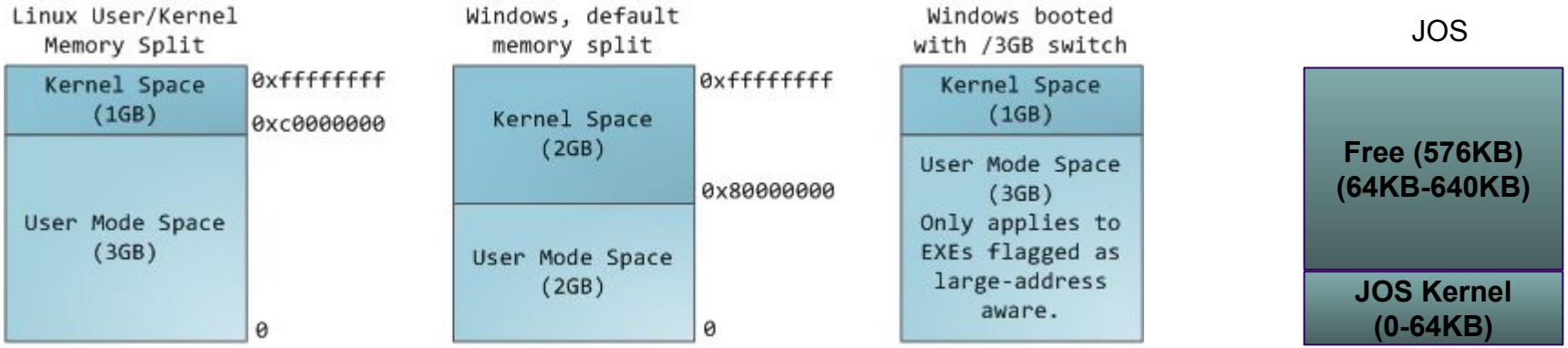
# Big picture (cont)

- OS deals with physical memory
  - Loading
  - Sharing physical memory between processes
  - Dynamic memory allocation

# Connecting the dots



# Process memory map

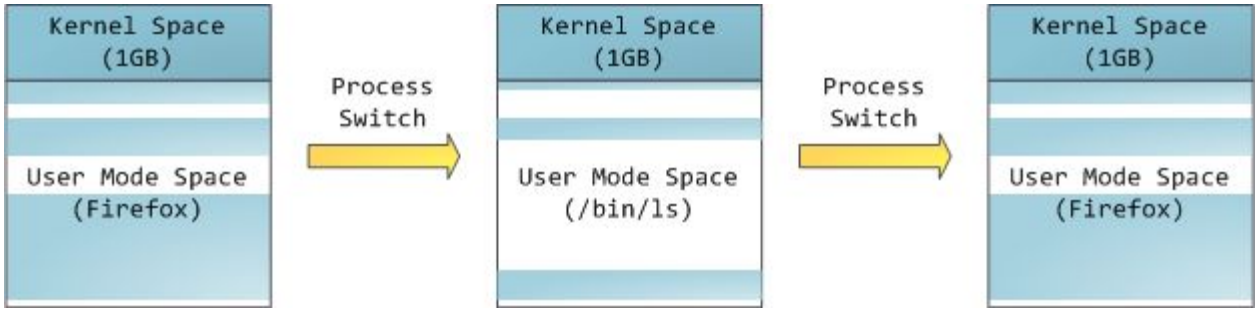



On ubuntu (check kernel map): sudo cat /proc/iomem



Copied from: https://manybutfinite.com/post/anatomy-of-a-program-in-memory/

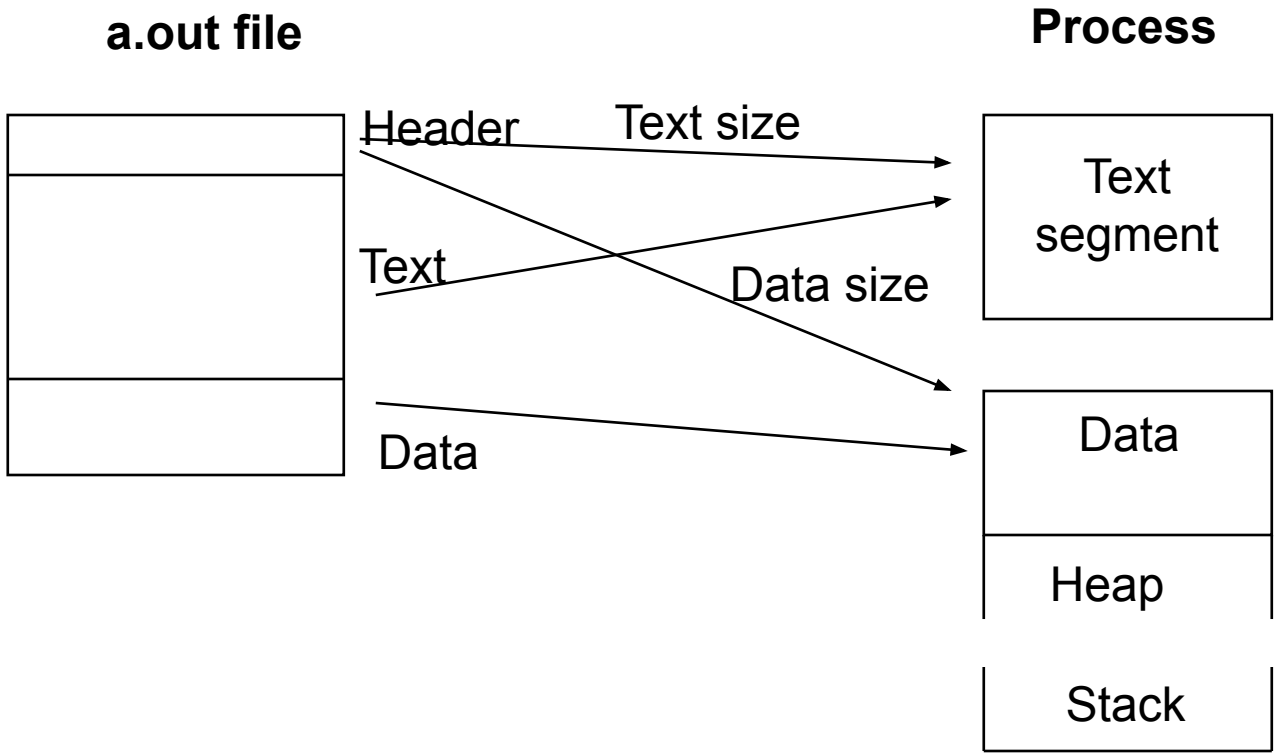
# Easier context-switch






Copied from: https://manybutfinite.com/post/anatomy-of-a-program-in-memory/

# Loading


a.out file
Process
Header
Text size
Text
Data size
Data
Text segment
Data
Heap
Stack

# Dynamic memory allocation during program execution

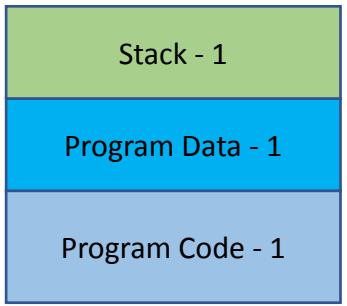
- Stack: for procedure calls
- Heap: for malloc()
- Both dynamically growing/shrinking

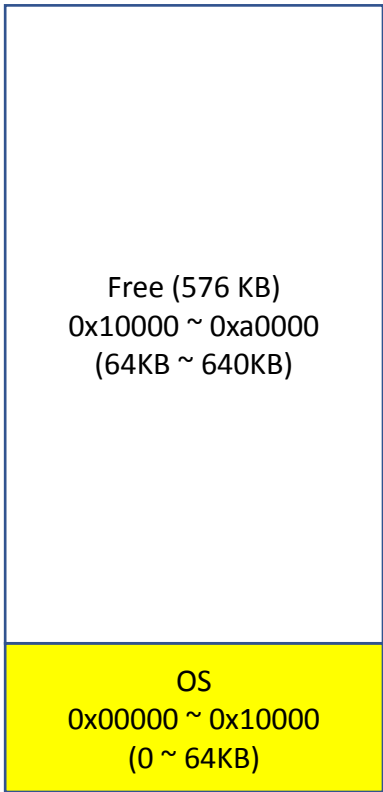
- Assumption for now:
  - Heap and stack are fixed size
  - OS has to worry about loading 4 segments per process:
    - Text
    - Data
    - Heap
    - stack

# Uniprogramming Environment

- Run one program

- The program can use memory space freely…

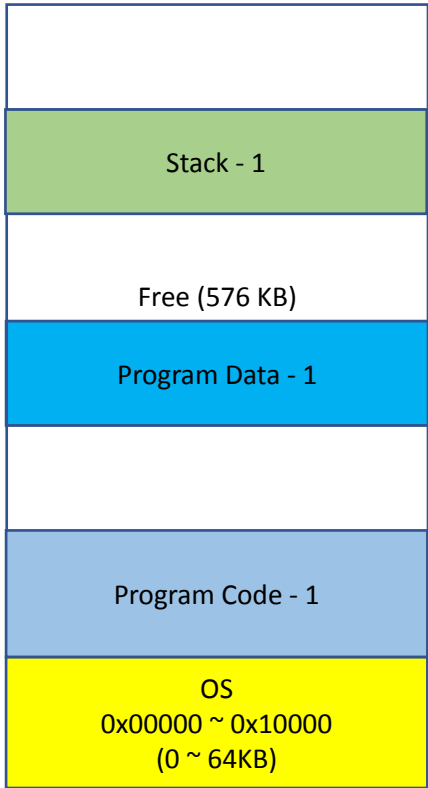


# Uniprogramming Environment

- Run one program

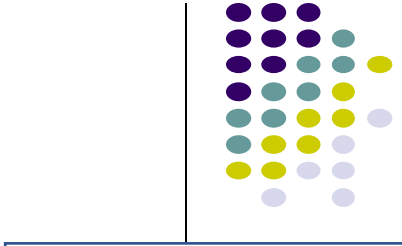
- The program can use memory space freely…

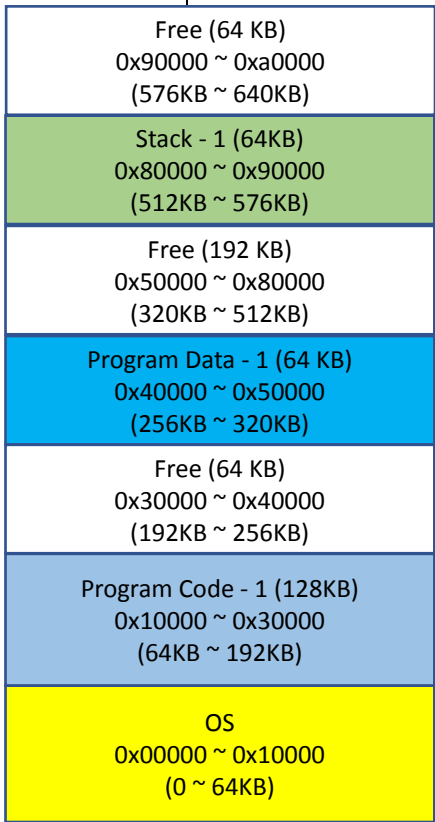


# Uniprogramming Environment

- Run one program

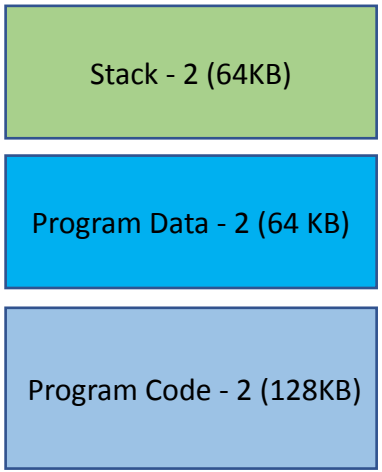
- The program can use memory space freely…

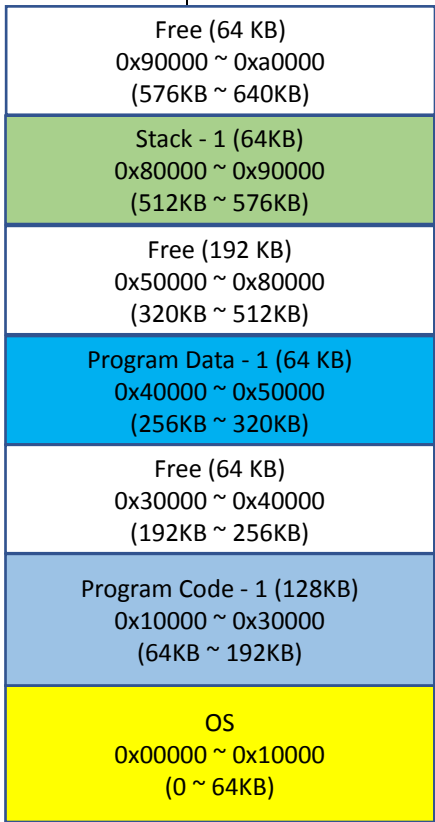




# Multi-programming Environment

- Run two programs

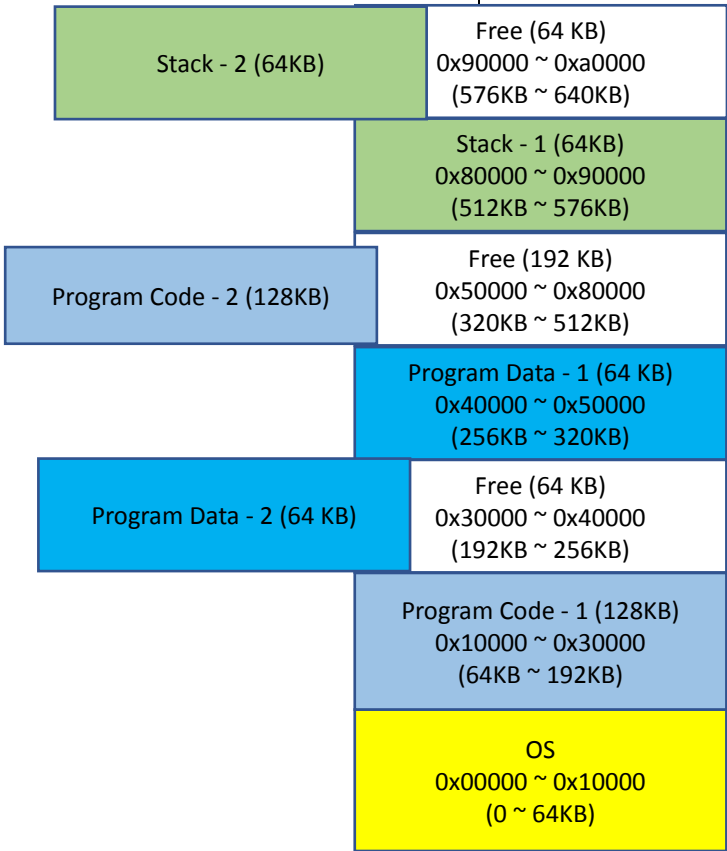


# Multi-programming Environment

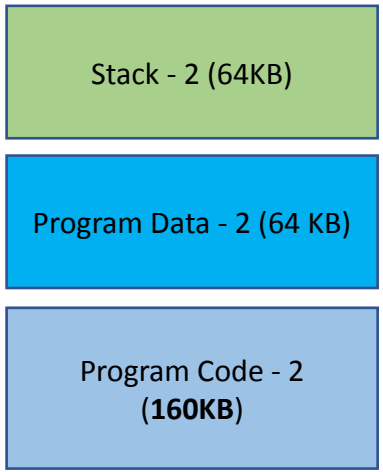
- Run two programs
- System's memory usage determines allocation
- Program need to be aware of the environment
  - Where does system loads my code?
  - You can't determine… system does..

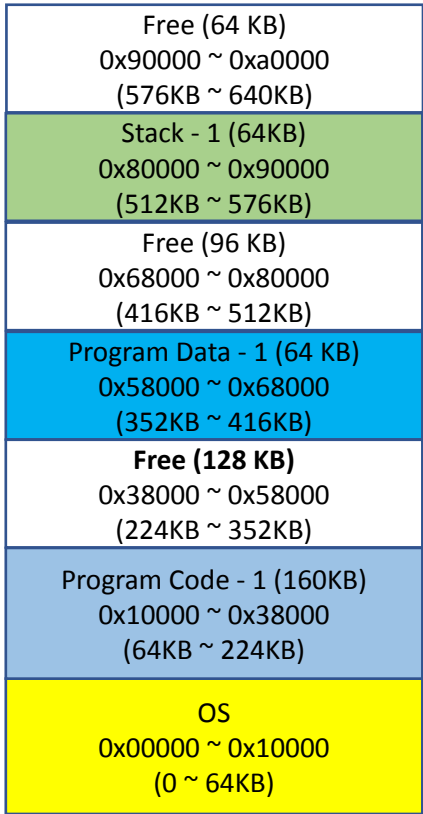
No Transparency…



# Multi-programming Environment

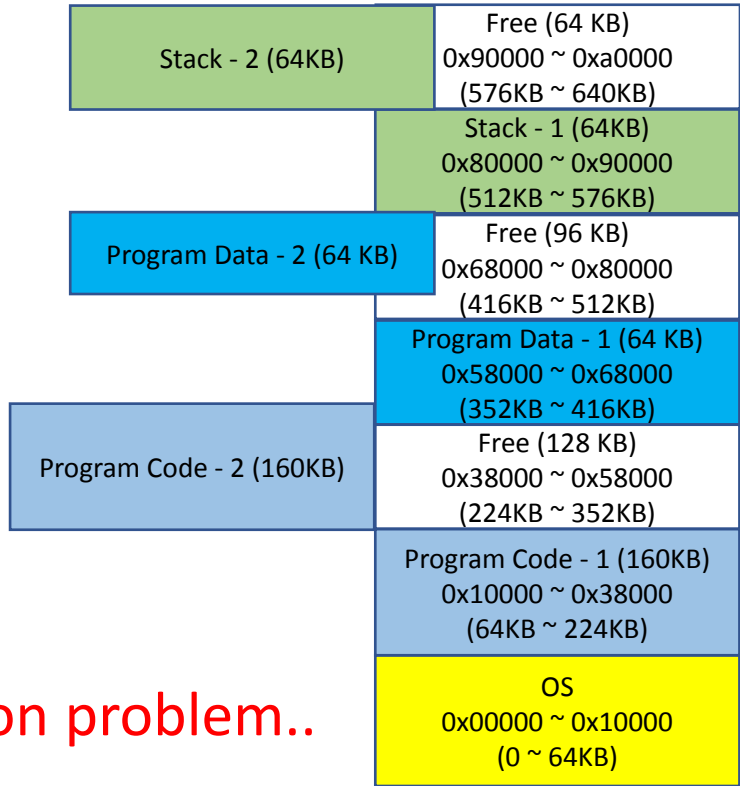
- Run two programs



# Multi-programming Environment

- Run two programs
  - Program size: 64KB + 64KB + 160K = 288KB
- Free mem: 64 + 96 + 128 = 288KB
- Cannot run Program – 2
  - Can't fit…

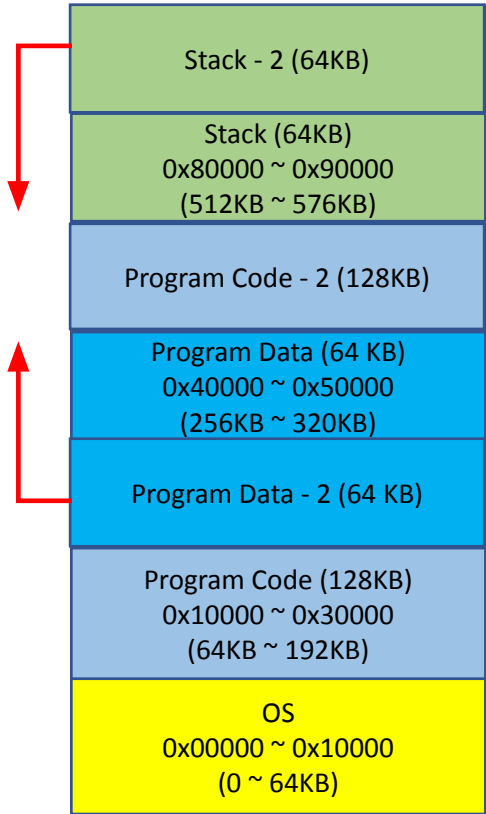



Not efficient.. Suffers memory fragmentation problem..

# Multi-programming Environment

- Run two programs
- What if Program-2's stack underflows?
- What if Program-2's data overflows?
- Without virtual memory
  - Programs can affect to the other's execution

No isolation. Security problem.



# Virtual Memory

- Three goals
  - Transparency: does not need to know system's internal state
    - Program A is loaded at 0x8048000. Can Program B be loaded at 0x8048000?
  - Efficiency: do not waste memory; manage memory fragmentation
    - Can Program B (288KB) be loaded if 288 KB of memory is free, regardless of its allocation?
  - Protection: isolate program's execution environment
    - Can we prevent an overflow from Program A from overwriting Program B's data?