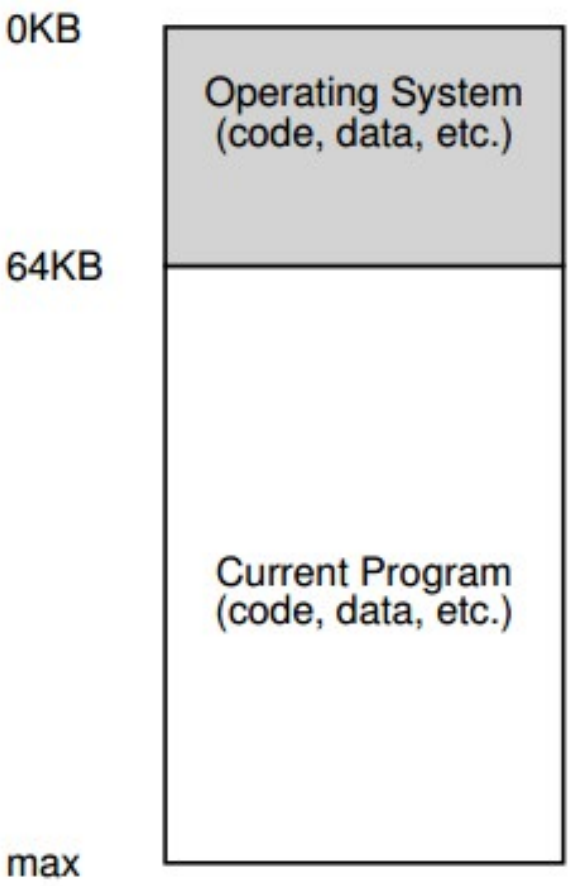
Overview of last lecture

- Condition variables, explicit queue.
  - Threads waiting on a condition, do so by **waiting** on a condition variable.
  - Threads wishing to let others know of a change in condition, do so by **signaling** on a condition variable.
- Condition variable is associated with a shared data and usually describes a predicate about it.
- Use different condition variables for different predicates about certain shared data. Since they are all about the same data, multiple condition variables can be used with the same mutex.
- Mesa vs Hoare semantics – always wait for a condition in a while loop.
- Semaphores – an object with a numeric value with two primitives
  - Wait (blocks until value is greater than 0, then decrements and returns)
  - Post (never blocks, increments the value by 1, wakes a sleeper if there is one)
- Semaphores are complete, they can be used for all synchronization needs.
  - Can be used to implement locks and condition variables.
- Reader-Writer locks: good for read-heavy scenarios, does not allow readers to block, unless there is a writer.
- Deadlocks: occur when four conditions hold: circular wait, hold and wait, no preemption, mutual exclusion.
- Wait-free/Lock-free data structures.
- Monitors: objects in which only one thread can be active at a time.

Virtual Memory

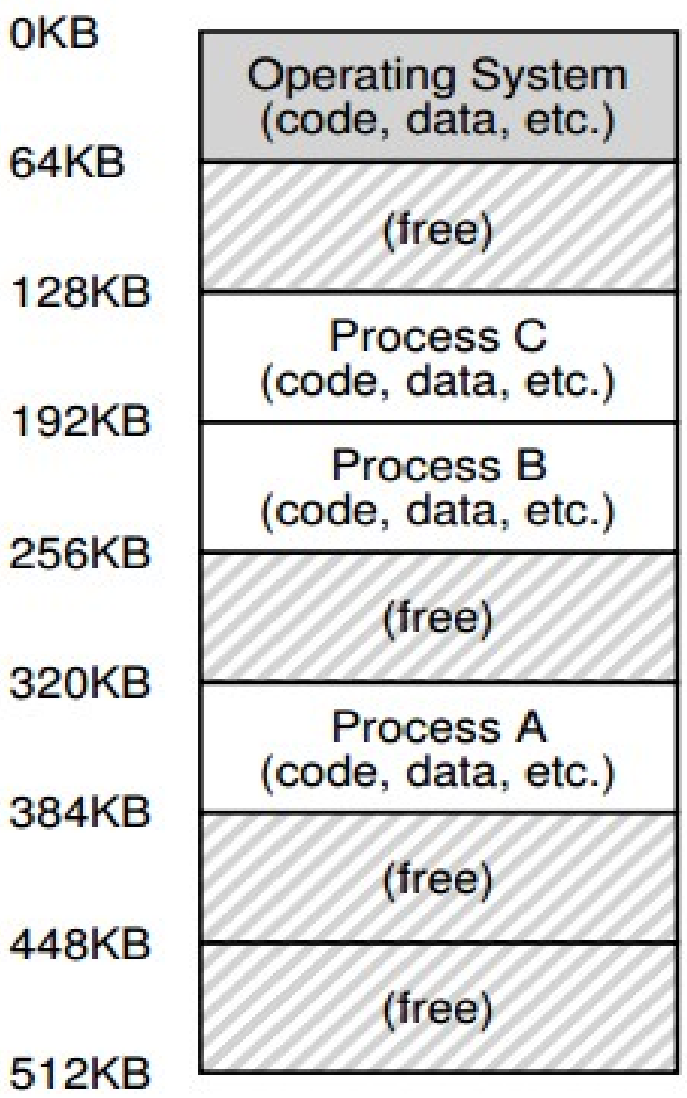
- Starts simple, gets complicated soon, easy to loose high level view. Therefore pay extra attention.
- We will follow major advances that lead to systems we have today.
- Any description in the middle does not reflect current systems.
- We will arrive at current systems at the end of the next lecture.

Virtual Memory – Early Operating Systems



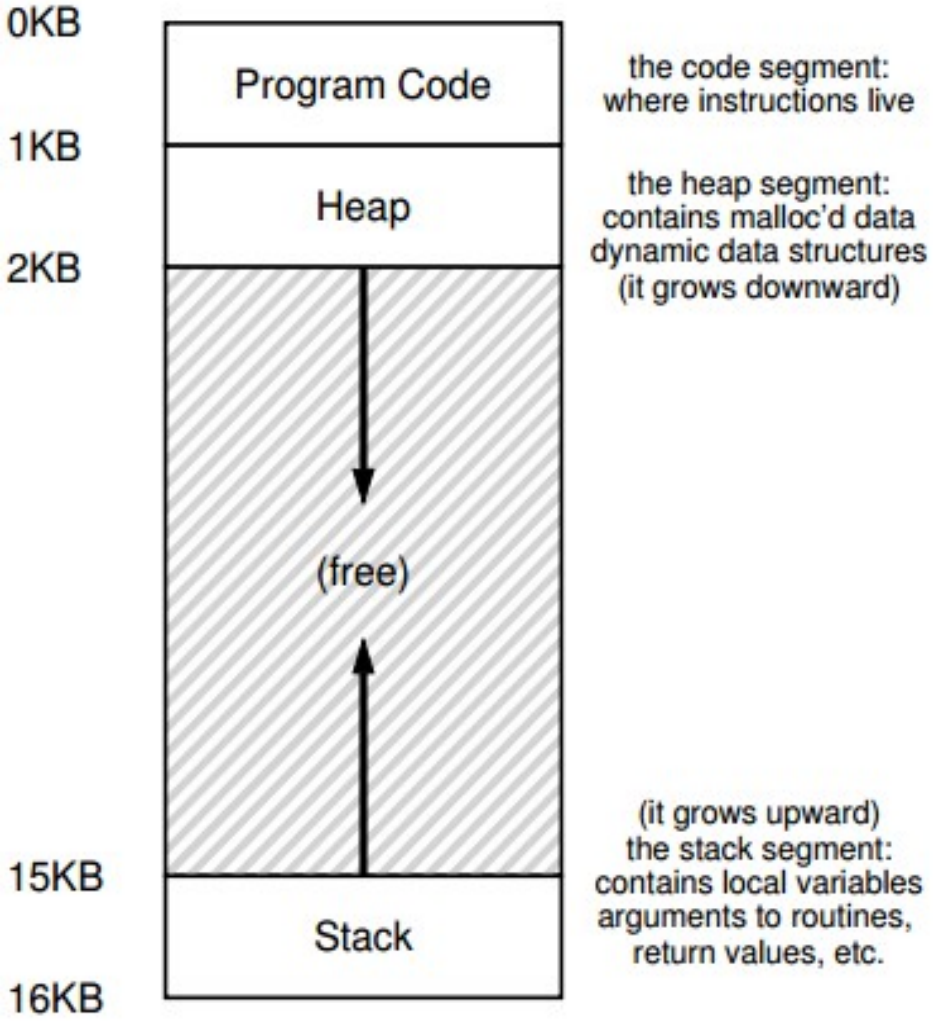
- OS was a library (a set of routines).
- Only one running program in phyiscal memory.

Virtual Memory – Early Operating Systems



- Multiprogramming: multiple processes ready to run, OS would switch when a process would perform I/O.
- Time sharing: early designs would run a process for a short period, preempt it and save all of its state to permanent storage, load another process and let it run – too slow. A refinement: leave processes in memory and switch between them.
- Doing this properly required memory protection.

Virtual Memory – The Address Space



- As a programmer, do we usually care about memory layout?
- Unlike us, compiler writers do. How would you compile the code for the same program that would be loaded at process C and process B?
- Compiler writers need to have an easy to use abstraction of memory and should not worry about how the program is laid out in physical memory.
- This abstraction is the **address space** – running program's view of memory.
- Why does stack grow down?
- Code section for instructions.
- Stack section for function call chain.
- Heap section for dynamically allocated memory.
- Remember – an abstraction, not a physical layout, see previous slide for physical layout.

Virtual Memory – The Address Space

- All addresses we deal with as programmers are virtual.
- We usually do not use explicit memory addresses, but variables to denote them.
- Compilers convert those into virtual memory addresses using the **address space abstraction** as a reference.
- We can see virtual memory addresses when debugging a C or C++ program or by explicitly printing them out, as in figure.
- Every process in figure in the previous slide, when tries to perform a load at virtual address 0, the OS with hardware support makes sure that load goes to the correct physical address, which is different for each process.

```
#include <stdio.h>
#include <stdlib.h>
int main(int argc, char *argv[]) {
    printf("location of code  : %p\n", (void *) main);
    printf("location of heap  : %p\n", (void *) malloc(1));
    int x = 3;
    printf("location of stack : %p\n", (void *) &x);
    return x;
}
```

```
location of code  : 0x1095afe50
location of heap  : 0x1096008c0
location of stack : 0x7fff691aea64
```

Virtual Memory - Goals

- Should be transparent to the running program.
- Should be efficient.
- Should provide protection between processes.

Virtual Memory – Address Translation

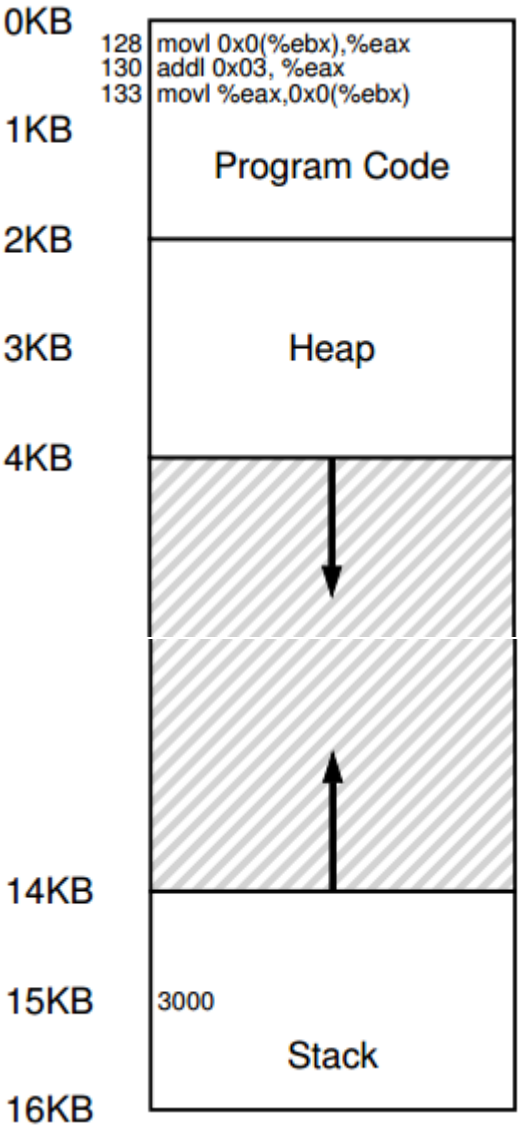
- Translation of virtual addresses to physical addresses is performed via hardware + OS support.
- Assumptions we make to simplify description:
  - Address space must be placed contiguously in physical memory.
  - Address space size is small, much less than physical memory, so that we can fit multiple address spaces into physical memory.
  - Each address space is of the same size.

Virtual Memory – Address Translation

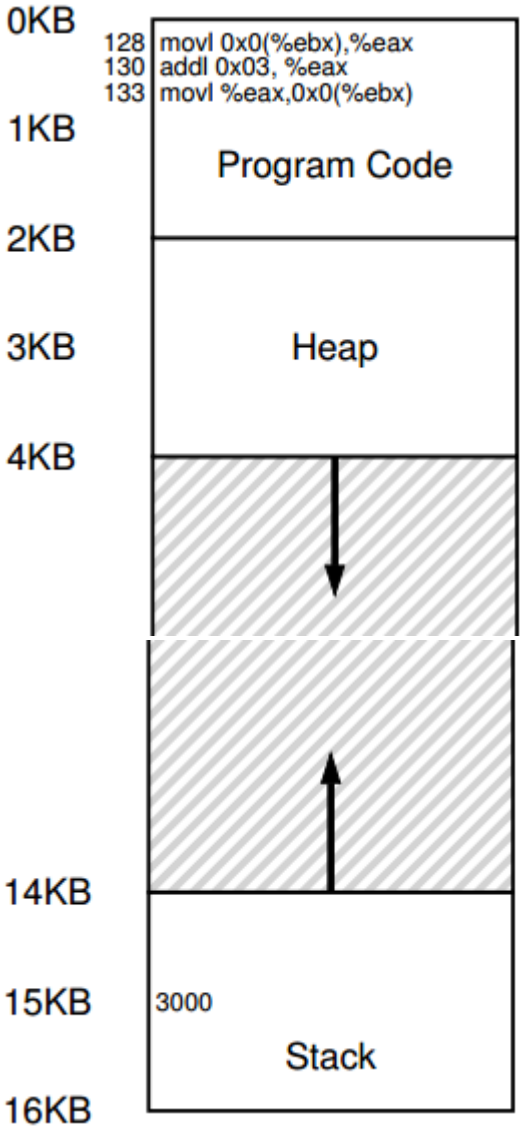
```
void func() {
  int x;
  ...
  x = x + 3;
  …
}
```

```
128: movl 0x0(%ebx), %eax
132: addl $0x03, %eax
135: movl %eax, 0x0(%ebx)

; address of x is in ebx
; load the value in ebx to eax
; increment by 3
; store eax back to x
```



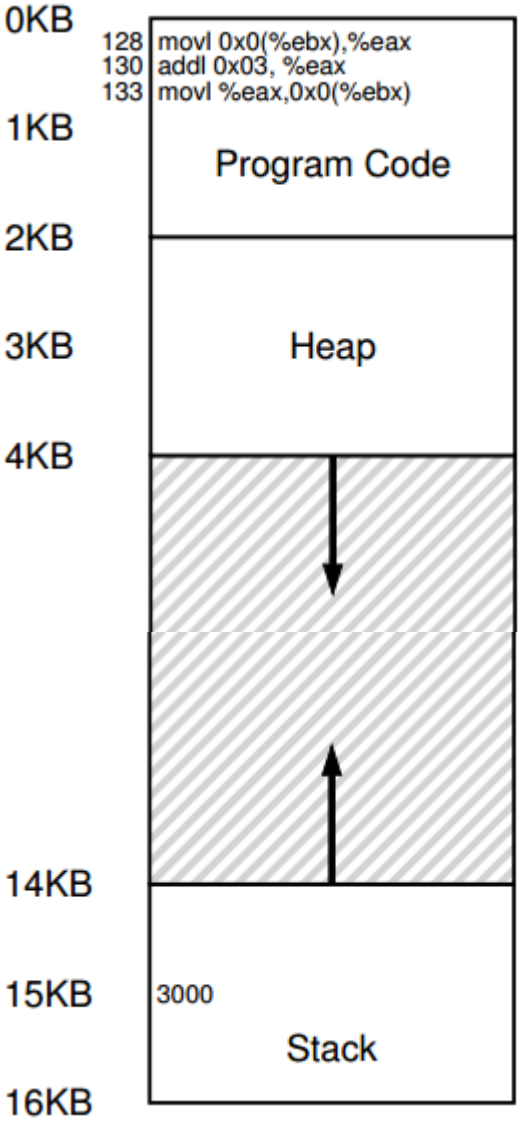
Virtual Memory – Address Translation

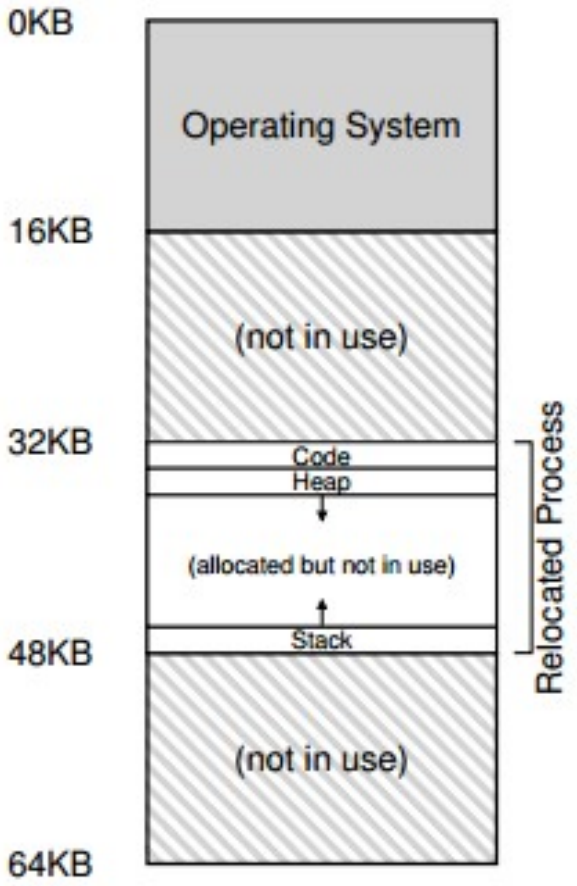



```
128: movl 0x0(%ebx), %eax
132: addl $0x03, %eax
135: movl %eax, 0x0(%ebx)
```

- What happens when the instructions on the left are run?
  - Fetch instr at 128
  - Execute (load from 15KB)
  - Fetch instr at 132
  - Execute instr (increment)
  - Fetch instr at 135
  - Execute instr (store to 15KB)

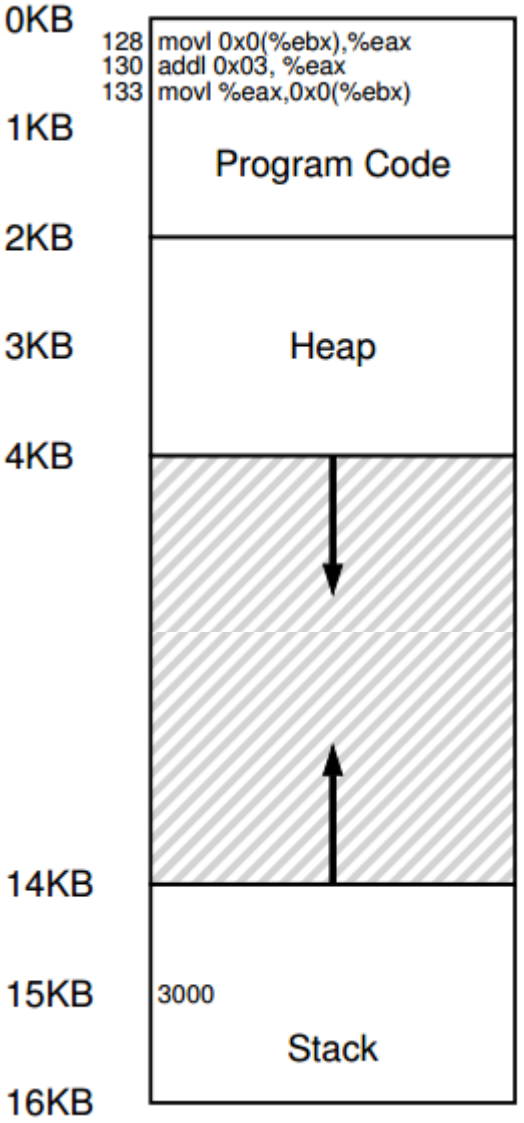
Virtual Memory – Address Translation Dynamic (Hardware-based) Relocation

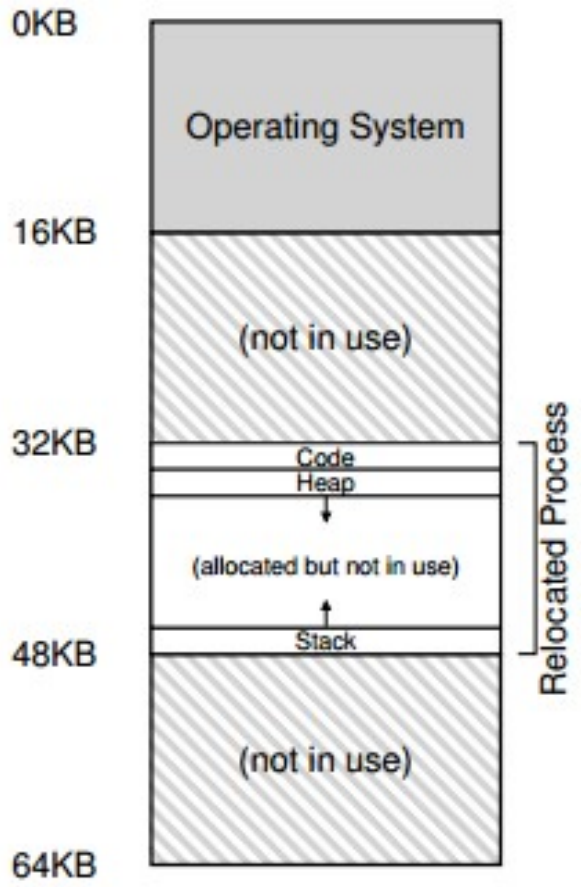




- Base and bounds == dynamic relocation.
- A pair of registers for currently running process.
- Each program compiled as if it is loaded at address 0. When process is loaded into memory, OS finds an empty memory slot and sets the base register to that value. E.g. the program on the left figure is loaded at physical address 32KB on the right figure, thus, base register value is 32KB.
- Each address reference is translated: physical address = virtual address + base.

Virtual Memory – Address Translation Dynamic (Hardware-based) Relocation






```
.28: movl 0x0(%ebx), %eax
.32: addl $0x03, %eax
.35: movl %eax, 0x0(%ebx)
```

- What happens when the instructions above are run?
  - Fetch instr at 128 => 32896
  - Execute (load from 15KB) => 47KB
  - Fetch instr at 132 => 32900
  - Execute instr (increment)
  - Fetch instr at 135 => 32903
  - Execute instr (store to 15KB) => 47KB

Virtual Memory – Address Translation Dynamic/Static Relocation

- What happened to **bounds** register? Every memory reference is checked to be within the bounds, trap to OS is made if not.
- **Base** and **bounds** can also be implemented as **base** and **size**.
- Static relocation: same idea done in software. Once the OS decide where to load the binary, **loader** rewrites the binary the by adding the value of a would be base register to all memory references.
- Disadvantages of static relocation:
  - No bounds register, no protection.
  - Once a program is loaded, it cannot be relocated, since binary rewriting is done at load time, hence **static** relocation. Whereas with dynamic relocation, we can change the base register value and relocate the program after it has been loaded but while it is not running.

Virtual Memory – Address Translation

- Duties of the OS: the OS should manage free memory by maintaining some kind of a **free list**:
  - Find a memory slot to load a new program.
  - Reclaim memory from a terminated process.
- There is only one pair of base and bounds registers. Since every program is loaded at a different address, they different values for different programs, just like PC and SP. Thus, the OS should save and restore (to/from where?) values of these registers when a context switch occurs.

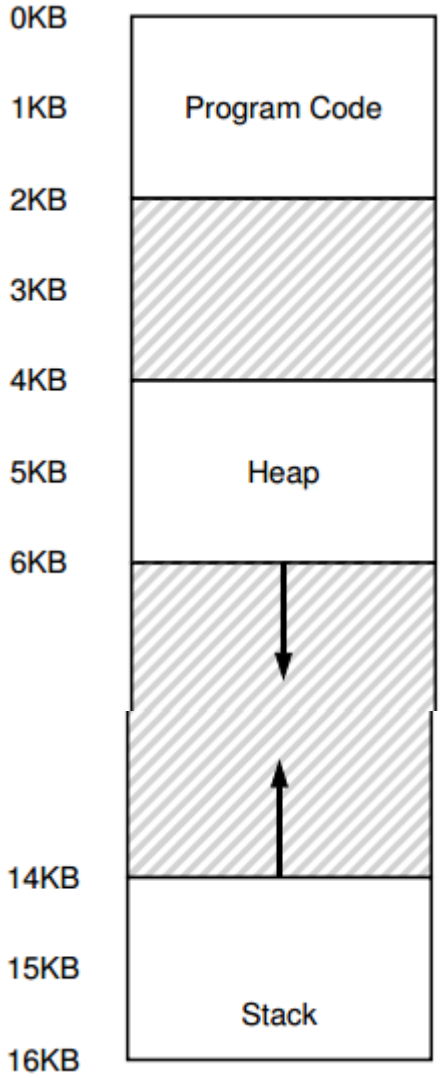
Virtual Memory - Segmentation

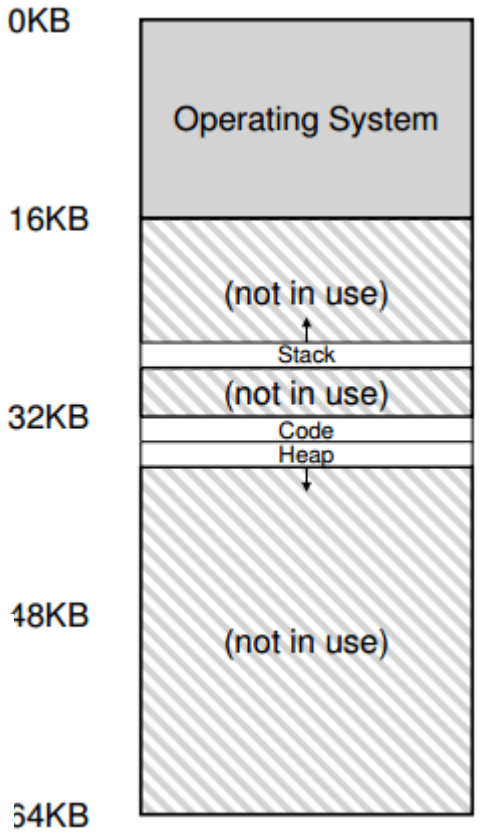
- Dynamic relocation
  - Leads to **internal fragmentation**. Unused space between stack and heap is wasted. Even though we may have enough aggregate memory for a new process, we are unable to use it.
  - Does not allow a program with large address space to be loaded into memory. How can we load a program with 32-bit address space (4GB memory) to 1GB physical memory?
- Segmentation is our first attempt to fix that.

Virtual Memory - Segmentation

- Segmentation: generalized base and bounds
  - Have one base and bounds pair **per logical segment**, instead of just one for the whole address space.
  - Logical segments: code, heap and stack.
- With this setup, segments can be located at arbitrary memory offsets.

Virtual Memory – Segmentation





Segment	Base	Size
Code	32K	2K
Heap	34K	2K
Stack	28K	2K

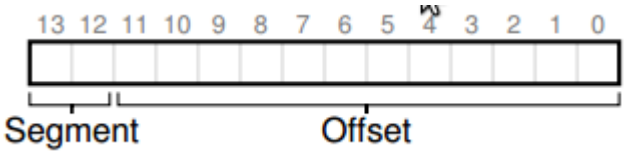
- We can place each segment **independently** in physical memory. Only used memory is allocated, thus **sparse address space** can be accommodated. SAMPLE RELOCATION.

Virtual Memory – Segmentation

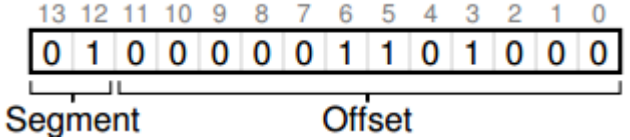
- Given a virtual address how does hardware know which segment it refers to? Two approaches:
  - Implicit – by noticing how the address was formed
    - If generated from the program counter then assume code segment
    - If based off of the stack or base pointers, stack segment
    - Anything else must be heap segment
  - Explicit – use top few bits of the virtual address to specify the segment.

Virtual Memory - Segmentation

- We have three segments, how many bits do we need to represent three different segments?



- Example, address from heap: 4200



- What's the segment number? 01
- What's the offset? 0000 0110 1000 (0x68, 104)

Virtual Memory – Segmentation

```
1 // get segment number
2 Segment = (VirtualAddress & SEG_MASK) >> SEG_SHIFT
3 // now get offset
4 Offset = VirtualAddress & OFFSET_MASK
5 if (Offset >= Bounds[Segment])
6   RaiseException(PROTECTION_FAULT)
7 else
8   PhysAddr = Base[Segment] + Offset
9   Register = AccessMemory(PhysAddr)

SEG_MASK = 0x3000
SEG_SHIFT = 12
OFFSET_MASK = 0XFFF
```

Virtual Memory – Segmentation

- What about stack? It has been relocated to 28KB. What makes stack special? It grows backwards, from 28KB to 26KB. Hardware needs extra bit to note that.

Segment	Base	Size	Grows Positive?
Code	32K	2K	1
Heap	34K	2K	1
Stack	28K	2K	0

- Hardware needs to translate virtual address references to stack segment differently.
- Example: access 15KB
  - In binary 11 1100 0000 0000 (hex 0x3C00)
  - Which segment? 11 → stack segment
  - What's the offset? 1100 0000 0000 → 3072 (3KB)
  - Calculate negative offset: offset - max segment size = 3072 – 4096 = -1024
  - Add it to stack segment base: 28KB – 1024 = 27KB.

Virtual Memory – Segmentation

- Observe that while stack and code segments change as the process runs, code segment is immutable.
- This leads to an optimization – **code sharing** – that is still in use today. Imagine the same program being loaded twice. How would code sharing help?
- Support from hardware, why do we need it?

Segment	Base	Size	Grows Positive?	Protection
Code	32K	2K	1	Read-Execute
Heap	34K	2K	1	Read-Write
Stack	28K	2K	0	Read-Write

- How would the mapping algorithm change?

Virtual Memory – Segmentation

- In comparison to dynamic relocation, the OS now has more registers to save and restore.
- One problem with segmentation is **external fragmentation** – the address space becomes fragmented with many tiny segments and the OS is unable to find contiguous segment of requested size.
- Many algorithms to deal with it: **best-fit**, **first-fit**, **buddy**, etc.