Early Systems

- In early systems, the OS was a set of routines (a library, really) that sat in memory (starting at physical address 0 in this example), and there would be one running program (a process) that currently sat in physical memory (starting at physical address 64k in this example) and used the rest of memory.

Multiprogramming and Time Sharing

- The era of multiprogramming was born, in which multiple processes were ready to run at a given time, where the OS would switch between them, for example when one decided to perform an I/O. Doing so increased the effective utilization of the CPU as well as efficiency
- Soon enough, however, people began demanding more of machines, and the era of time sharing was born
- The notion of interactivity became important, as many users might be concurrently using a machine, each waiting for (or hoping for) a timely response from their currently-executing tasks.
- One way to implement time sharing would be to run one process for a short while, giving it full access to all memory, then stop it, save all of its state to some kind of disk (including all of physical memory), load some other process's state, run it for a while, and thus implement some kind of crude sharing of the machine
    - this approach has a big problem: it is way too slow, especially as memory grows
    - While saving and restoring register-level state (the PC, general-purpose registers, etc.) is relatively fast, saving the entire contents of memory to disk is brutally non-performant.
- what we'd rather do is leave processes in memory while switching between them, allowing the OS to implement time sharing efficiently
- Allowing multiple programs to reside concurrently in memory makes protection an important issue; you don't want a process to be able to read, or worse, write some other process's memory

The Address Space

- The OS creates an easy to use abstraction of physical memory. We call this abstraction the address space, and it is the running program's view of memory in the system.
- The address space of a process contains all of the memory state of the running program. For example, the code of the program (the instructions) have to live in memory somewhere, and thus they are in the address space.
    - The program, while it is running, uses a stack to keep track of where it is in the function call chain as well as to allocate local variables and pass parameters and return values to and from routines.
    - The heap is used for dynamically-allocated, user-managed memory, such as that you might receive from a call to malloc() in C or new in an object oriented language such as C++ or Java.

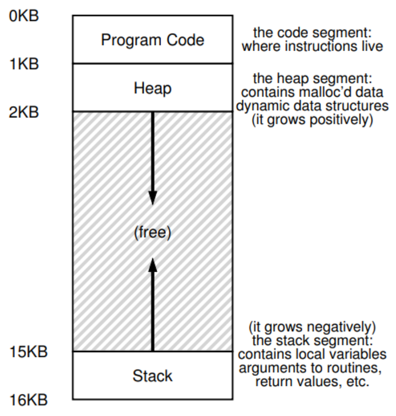

0KB
1KB
2KB
15KB
16KB
Program Code
Heap
(free)
Stack
the code segment: where instructions live
the heap segment: contains malloc'd data dynamic data structures (it grows positively)
(it grows negatively) the stack segment: contains local variables arguments to routines, return values, etc.

Figure 13.3: **An Example Address Space**

- As shown in the image, we have the two regions of the address space that may grow (and shrink) while the program runs.
- Those are the heap (at the top) and the stack (at the bottom). We place them like this because each wishes to be able to grow, and by putting them at opposite ends of the address space, we can allow such growth: they just have to grow in opposite directions.
- The heap thus starts just after the code (at 1KB) and grows downward (say when a user requests more memory via malloc()); the stack starts at 16KB and grows upward (say when a user makes a procedure call).
    - However, this placement of stack and heap is just a convention; you could arrange the address space in a different way if you'd like for ex. when multiple threads co-exist in an address space
- when we describe the address space, what we are describing is the abstraction that the OS is providing to the running program.
    - The program really isn't in memory at physical addresses 0 through 16KB; rather it is loaded at some arbitrary physical address(es).
- How can the OS build this abstraction of a private, potentially large address space for multiple running processes (all sharing memory) on top of a single, physical memory?
    - When the OS does this, we say the OS is virtualizing memory, because the running program thinks it is loaded into memory at a particular address (say 0) and has a potentially very large address space (say 32-bits or 64-bits); the reality is quite different
    - When, for example, process A in the figure above tries to perform a load at address 0 (which we will call a virtual address), somehow the OS, in tandem with some hardware support, will have to make sure the load doesn't actually go to physical address 0 but rather to physical address 320KB (where A is loaded into memory).

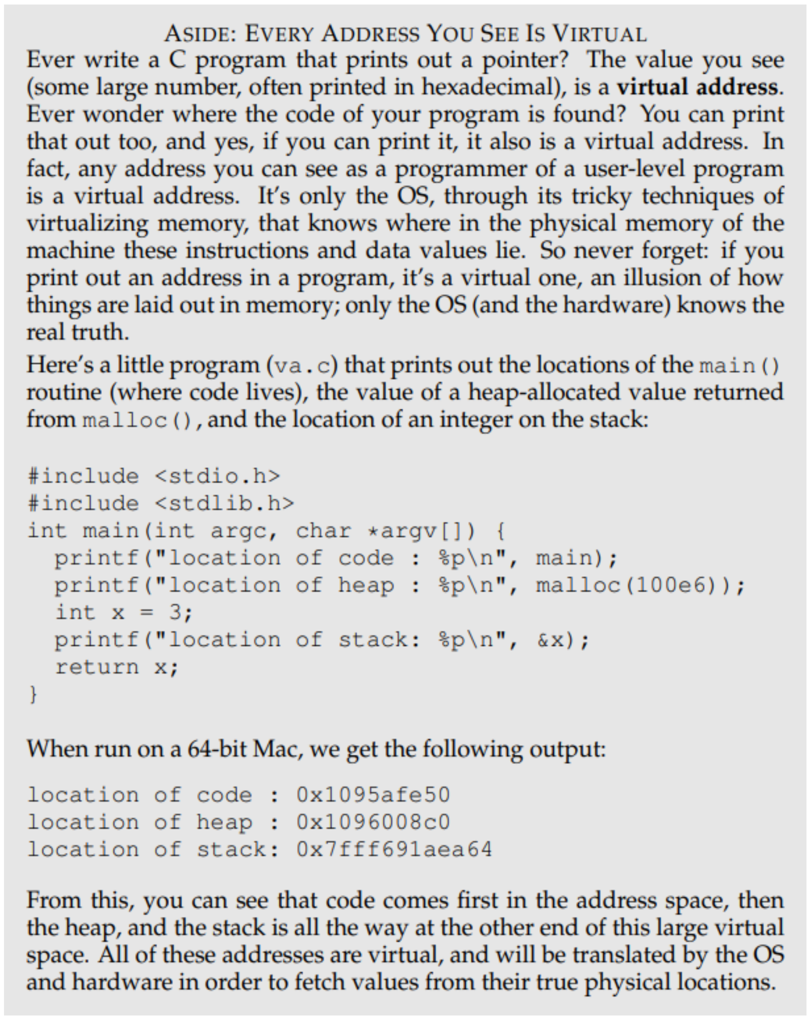
Goals

- One major goal of a virtual memory (VM) system is transparency
    - The OS should implement virtual memory in a way that is invisible to the running program. Thus, the program shouldn't be aware of the fact that memory is virtualized; rather, the program behaves as if it has its own private physical memory.
        - Behind the scenes, the OS (and hardware) does all the work to multiplex memory among many different jobs, and hence implements the illusion.
- Another goal of VM is efficiency.
    - The OS should strive to make the virtualization as efficient as possible, both in terms of time (i.e., not making programs run much more slowly) and space (i.e., not using too much memory for structures needed to support virtualization).
- Finally, a third VM goal is protection. The OS should make sure to protect processes from one another as well as the OS itself from processes
    - When one process performs a load, a store, or an instruction fetch, it should not be able to access or affect in any way the memory contents of any other process or the OS itself (that is, anything outside its address space).

    - Protection thus enables us to deliver the property of isolation among processes; each process should be running in its own isolated cocoon, safe from the ravages of other faulty or even malicious processes.

Summary

- The virtual memory system is responsible for providing the illusion of a large, sparse, private address space to each running program; each virtual address space contains all of a program's instructions and data, which can be referenced by the program via virtual addresses.
- The OS, with some serious hardware help, will take each of these virtual memory references and turn them into physical addresses, which can be presented to the physical memory in order to fetch or update the desired information.
- The OS will provide this service for many processes at once, making sure to protect programs from one another, as well as protect the OS. The entire approach requires a great deal of mechanism (i.e., lots of low-level machinery) as well as some critical policies to work;
- Isolation is a key principle in building reliable systems. If two entities are properly isolated from one another, this implies that one can fail without affecting the other.
    - Operating systems strive to isolate processes from each other and in this way prevent one from harming the other. By using memory isolation, the OS further ensures that running programs cannot affect the operation of the underlying OS.
    - Some modern OS's take isolation even further, by walling off pieces of the OS from other pieces of the OS. Such microkernels [BH70, R+89, S+03] thus may provide greater reliability than typical monolithic kernel designs



ASIDE: EVERY ADDRESS YOU SEE IS VIRTUAL

Ever write a C program that prints out a pointer? The value you see (some large number, often printed in hexadecimal), is a **virtual address**. Ever wonder where the code of your program is found? You can print that out too, and yes, if you can print it, it also is a virtual address. In fact, any address you can see as a programmer of a user-level program is a virtual address. It's only the OS, through its tricky techniques of virtualizing memory, that knows where in the physical memory of the machine these instructions and data values lie. So never forget: if you print out an address in a program, it's a virtual one, an illusion of how things are laid out in memory; only the OS (and the hardware) knows the real truth.

Here's a little program (`va.c`) that prints out the locations of the `main()` routine (where code lives), the value of a heap-allocated value returned from `malloc()`, and the location of an integer on the stack:

```
#include <stdio.h>
#include <stdlib.h>
int main(int argc, char *argv[]) {
  printf("location of code : %p\n", main);
  printf("location of heap : %p\n", malloc(100e6));
  int x = 3;
  printf("location of stack: %p\n", &x);
  return x;
}
```

When run on a 64-bit Mac, we get the following output:

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

From this, you can see that code comes first in the address space, then the heap, and the stack is all the way at the other end of this large virtual space. All of these addresses are virtual, and will be translated by the OS and hardware in order to fetch values from their true physical locations.