COEN 175

Lecture 13: Storage Allocation

Storage Allocation

- The compiler must ensure that storage will be allocated at run time for the program's data.
- Note that the compiler does not itself allocate space.
  - The compiler does not use `malloc` or `new`.
  - Rather, the compiler must generate code that will allocate the space at run time.
  - The compiler must also ensure that its generated code uses this space correctly.
  - Each object should be assigned its own memory location in accordance with the rules of the language.

Access Patterns

- Languages provide different allocation mechanisms depending upon how objects are used.
  - How many copies of an object exist?
  - What is the lifetime of an object?
  - How do we refer to objects?
- Simpler mechanisms are easier to implement and might be faster, but might be too restrictive.
- The three commonly used allocation mechanisms are **static**, **stack**, and **dynamic** allocation.

Static Allocation

- There is one and only one copy of an object.
- The lifetime of the object is the lifetime of the running program.
- Objects are simply referred to by name.
- Because of these properties, space for a statically allocated object can be laid out at **compile time**.
- What are some common examples?
  - Global variables and local variables declared as `static`
  - Code (your program is stored in memory too!)

Stack Allocation

- There can be multiple copies of an object. Each copy is called an **activation**.
- However, only the **most recently** allocated copy is (directly) accessible.
- Objects are still referred to by name.
- Each function is responsible for allocating and deallocating memory to hold these objects.
- What are some common examples?
  - Local variables and parameters

Dynamic Allocation

- There can be multiple copies of an object.
- The lifetime of objects is directly controlled by the programmer.
- Since the programmer can declare only a finite number of names, how do we refer to such objects?
  - We refer to such objects **indirectly** (i.e., using pointers).
- What are some common examples?
  - Objects created from `malloc` and `new`.

Language Support

- Does the C language support static allocation?
  - Yes ... global variables and code are statically allocated.
- Does the C language support stack allocation?
  - Yes ... C calls it "automatic" allocation.
  - Stack allocation is necessary to support recursion.
- Does the C language support dynamic allocation?
  - Surprisingly, no ... the **language** does not.
  - For that matter, the C language does not support I/O. Huh?
  - The C **standard library** does; these design decisions were key in allowing C to be easily ported to new platforms.

Memory Segments

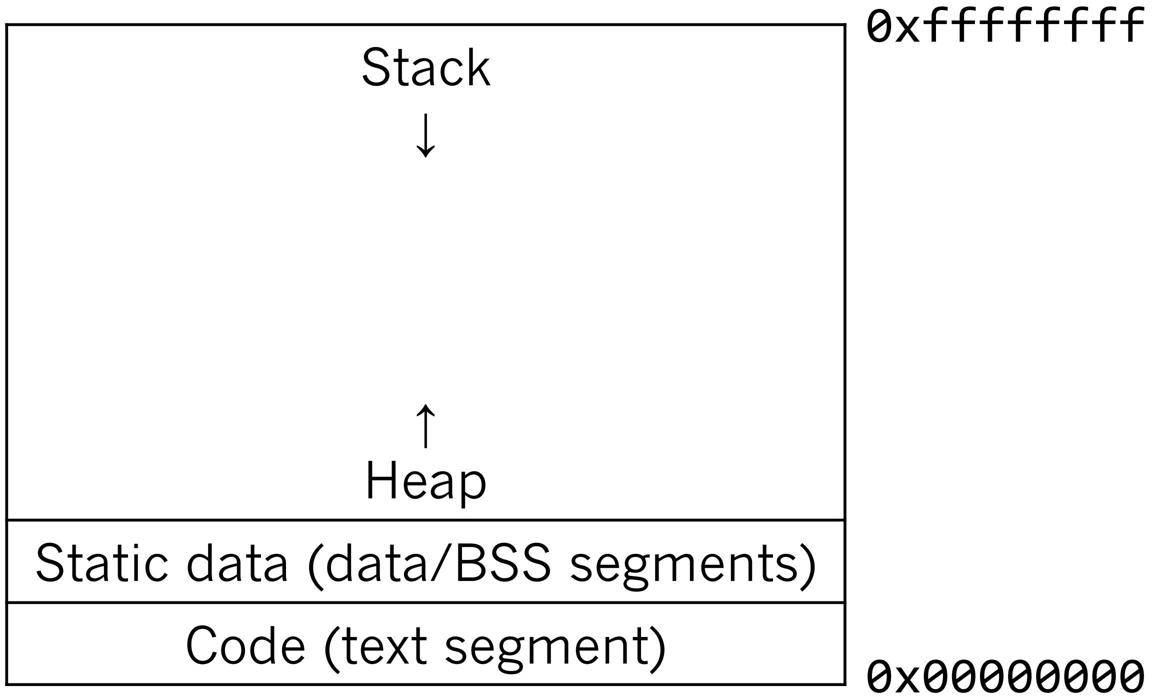
- Suppose we need to support static, stack, and dynamic allocation of data in our running program.
- How many different memory segments do we need?
  - Static data requires a segment.
  - Stack data requires a segment (“the run-time stack”).
  - Dynamic data requires a segment (“the heap”).
  - The code itself requires a segment.
- Don’t forget that your code is in memory as well!
- Trying to access an address not in one of your allocated segments will cause a **segmentation fault**.

Memory Layout

- Which segments have fixed size during execution?
    - The static data and code segments don't change size.
    - The stack and heap can grow and shrink.
- Suppose we have an entire 32-bit (or 64-bit) address space available to us.
- How might we place the four segments in memory?
    - Place the two fixed-size segments at either or both ends.
    - Place the other two segments at opposite ends in the remaining memory and have them grow towards each other.

Example Memory Layout

- The code and static data are placed at the start of the address space and the stack grows down.



Example

- Identify how each object in the following C program is allocated:

```
int x, *p;                      // x, p: static

int f(int y) {                  // f: static, y: stack
    static int z, *q;           // z, q: static

    p = malloc(10);             // *p: dynamic
    q = &z;                     // *q: static
}
```

- Note that `*q` and `z` refer to the same object.
    - We say that `*q` and `z` are **aliases** for each other.

Static Allocation

- Easiest of all allocation mechanisms because it can be done at compile time.
- There is only one copy of an object and the lifetime is that of the running program.
- The compiler simply uses an assembler directive to reserve space for an object.
  - Such a directive is also called a “pseudo-op.”
- The object is simply referred to by its name.

Example

- Using standard AT&T assembler syntax:
  - `.byte` *x* – reserve a byte with initial value *x*
  - `.word` *x* – reserve a 16-bit “word” with initial value *x*
  - `.long` *x* – reserve a 32-bit “long word” with initial value *x*
  - `.quad` *x* – reserve a 64-bit “quad word” with initial value *x*
- Usually each directive is given a **label** based on the name of the variable.
  - Support we have a declaration `int ival;`
  - We would output: `ival: .long 0`
  - Note that an “`int`” in C is a “`long`” in the assembler.

An Easier Way

- By default symbols declared in the assembler are not visible outside the source file.
  - We can mark symbols as visible using the `.globl` directive.
  - This directive tells the assembler to mark the symbol as global in the link map for the object file.
  - Declaring a global function or variable `static` in C just means no `.globl` directive is issued.
- An easier way to declare statically allocated data is to use the `.comm` directive:
  - `.comm` *name*, *size* – reserve “common” space with the given *name* and *size*