INF-1100
Functions, compiler, libraries and the operating system

Åge Kvalnes

University of Tromsø, Norway

October 8, 2014

Functions

All programs are built as a collection of functions.

A function is a fragment of a program that:

- Implements a well-defined functionality.
- Can be invoked by another function.
- Returns control to the calling function when finished.

There are basically two reasons for using functions:

- Reuse code. Debug once.
- Program structure. Easier to read and understand a program that uses functions (less code).

Terminology: **function == procedure == subroutine**

The stack (x86)

Variables declared inside a function are stored on the stack. Think of the stack as an array. A CPU register (the stack pointer) is initialized with the memory address of the array. There are two operations defined on the stack:

- push(variable):
  stackpointer−−;
  $*stackpointer = variable;$
- variable = pop():
  $variable = *stackpointer;$
  $stackpointer++;$

On the x86, push and pop are *actual instructions*. The CPU updates the address in the stack pointer register as part of executing pop/push.

Functions: Invocation and return (x86)

A function is invoked by executing a **call** instruction.

Operand specifies the *address* of the function.

As part of executing the **call** instruction, the address of the following instruction is pushed onto the stack.

The function executes the **ret** instruction to return control back. (**ret** pops an address from the stack and writes it to the program counter.)

Functions: Arguments and return values (x86)

A function frequently needs some input values to perform its task. These values are called *arguments*.

On the x86, arguments to a function are passed via the stack. (push(arg0), push(arg1), etc.).

A function frequently needs to return a value back to the caller (the return value).

On the x86, return values are passed via a CPU register (%eax).

Functions: Use of registers (x86)

A function needs to use registers as part of its operation. The registers may contain values that are used by the caller. Thus the registers must be *stored* (in memory) before they are used by the function, and then *restored* before they are again used by the caller.

Callee save:

- The called function is in charge of storing and restoring registers.

Caller save:

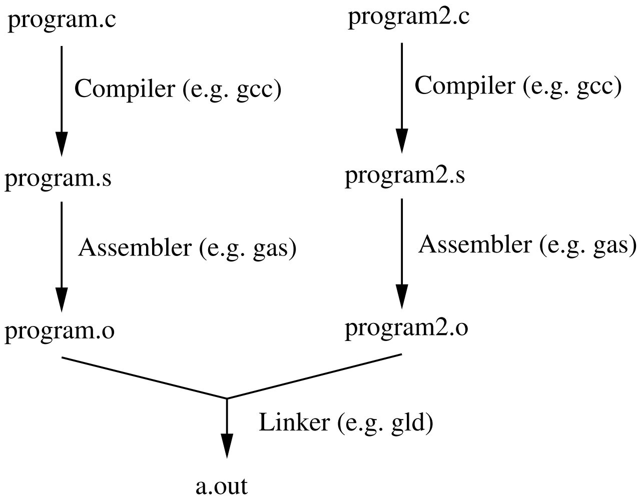
- The calling function is in charge of storing and restoring registers.

On the x86, callee save is used.

From C code to an executable, one file

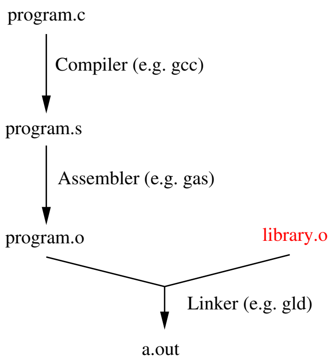
$$\begin{tikzcd}
\text{program.c} \arrow[d, "\text{Compiler (e.g. gcc)}"] \\
\text{program.s} \arrow[d, "\text{Assembler (e.g. gas)}"] \\
\text{program.o} \arrow[d, "\text{Linker (e.g. gld)}"] \\
\text{a.out}
\end{tikzcd}$$

From C code to an executable, many files


$$\begin{tikzcd}
\text{program.c} \arrow[d, "\text{Compiler (e.g. gcc)}"] & & \text{program2.c} \arrow[d, "\text{Compiler (e.g. gcc)}"] \\
\text{program.s} \arrow[d, "\text{Assembler (e.g. gas)}"] & & \text{program2.s} \arrow[d, "\text{Assembler (e.g. gas)}"] \\
\text{program.o} \arrow[dr, dash] & & \text{program2.o} \arrow[dl, dash] \\
& \text{a.out} \arrow[from=u, "\text{Linker (e.g. gld)}"] &
\end{tikzcd}$$

Reusing existing programs: Libraries

A library is a precompiled collection of functions. A library can be *linked* to your program. The *linker* will *resolve* references in your code to functions in the library.



Defining interfaces: Include files

```
int myfunction(int *p)
{
...
}
int myotherfunction(void)
{
  float f;
  myfunction(f);
}
```

If compiling the above, the compiler detects that myfunction is supplied with an argument of the wrong type: the compiler performs error checking on your code.

How can the compiler perform error checking when the called function resides in a precompiled (object file) library (or in a different C file)?
⇒ By defining interfaces in **include** files.

Include files

An include file defining the interface to a library typically contains:

- Definitions of functions (their names, argument types, and return types).

```
int mylibraryfunction(someinput_t *s);
```

- Definitions of composite data types.

```
typedef struct someinput someinput_t;
struct someinput {
  int input1;
  int input2;
};
```

- Useful macros (defines).

The operating system

The operating system is a program.

The operating system (OS) basically solves two problems:

- When writing a program, it would take a lot of effort to also write the code that controls I/O devices.
- It is desirable to be able to simultaneously run multiple programs on a computer. What program uses the computer when needs to be controlled.

Solution: Let the OS program handle all interfacing with the hardware and control what "normal" programs gets to execute when.

Normal programs call functions in the OS when they need to use I/O devices.

The operating system: Abstractions

The operating system defines convenient *abstractions* for utilizing the CPU and I/O devices. These typically include:

- Processes: Running programs.
- Files: Storing data on disk.
- Sockets: Sending/Receiving data over the network.

Abstraction example: Files

Hardware disk interface: Disk is divided into blocks of fixed size. Each block has an address (0..MAX). The disk controller offers an interface for reading and writing blocks.

OS file interface:

- Create new file with specific name.
- Read/Write data from/to file.
- Delete file.
- Organize files into a tree structure to ease management of many files.

Abstractions: Implementing the file abstraction

In the implementation of the file abstraction, the operating system typically uses three different data structures:

- Free block list. What disk blocks do not contain file data.
  - Typically placed on a fixed location on the disk.
- File description: Name of file, size of file, date created, etc.
  - Typically a fixed location on the disk contains many of these data structures.
- Per file disk block table: Information on what disk blocks a file occupies.
  - Typically placed in blocks taken from the free block list.

Managing free disk blocks

Problem: Keeping track of disk blocks that are in use.

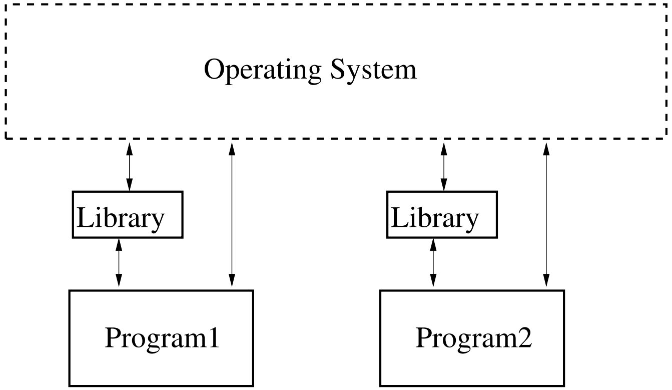
Approach: Use a bitmap.

- If bit N is 0, corresponding disk block is **not** in use.
- If bit N is 1, corresponding disk block is in use.

Interface:

- **int bitalloc(unsigned char *bitmap, int bitmapnbytes)**
- **int bitfree(unsigned char *bitmap, int bitmapnbytes, int freebit)**

The big picture


Operating System
Library
Library
Program1
Program2