EE360C Spring 2007

Linking and Loading

Adnan Aziz *UT Austin*

1 Programs and processes

Processors execute instructions—these are prototypically reg-reg operations, ld/st, and system instructions.
A program (binary) is a sequence of machine instructions:

```
add r0, r1, r2
brz r2 r7
ld r5 r6
...
```

A process is a running program—it's the image of the program in memory, along with additional state:

- A program counter—says where the program is in execution
- Stack—stack memory and stack counter
- Heap—memory that's persistent across function calls
- Miscellaneous—pending exceptions, privileges, etc.

2 Function calls

A function call is a sequence of instructions that ends in the program counter being set to the starting location of the function being called.

Before the actual change in PC takes place, arguments have to be pushed, return address has to be saved, stack offset has to be saved; right after the change in PC, the caller has to save any registers it's going to use.
When the function returns, it has to restore state, e.g., the registers it stored.
A trivial function—

```
int foo( int a ) {
  int y = a + 42;
  return y;
}
```

Using gcc this compiles to

```
        pushl   %ebp
        movl    %esp, %ebp
        movl    8(%ebp), %eax
        addl    \$42, %eax
        leave
        ret
```

There is some header information that basically acts as notes identifying labels (variable/function addresses).

3 System calls

Very often your process needs to perform actions that involve working with devices—e.g., sending a network packet, or read a file, or it may want to communicate with another process. You could let the process directly manipulate the file system, this is what ancient OSs did (MSDOS was basically a program loader). However, it's a bad idea—many security/stability issues.
The mechanism for talking to devices/other processes is the system call. Each modern OS has a set of 100—1000 system calls. In order to say do a write, your program first sets

up appropriate arguments, and then makes a call to a specific instruction, usually a TRAP, passing in the id of the sys call, and a pointer to the arguments.
The OS copies over these arguments, checks them out, and implements the call. When it's done, it returns a code to your process, and re-activates it.
Incidentally, a system call is very expensive, much more so than a function call. Some reasons—overhead of flushing the pipe, loading in the handler, copying the arguments, etc. (The OS does more than just implement system calls—it also schedules processes, and handles interrupts, etc.)

4 Libraries and linkage

Libraries are just files that contain sets of compiled functions. Some of the functions may be purely in user-space (i.e., there's no system call involved). Some may involve system calls (e.g., printf has some user code set up arguments, then calls the write system call). Some may conditionally require a system call (e.g., malloc).
Linkage is performed to put together a set of functions into a program. In its simplest form, the program is "statically" linked, that is the linker takes a list of files containing functions, and creates a single program at the end. In addition to the files being compiled, the linker has to know where to find the libraries—this can be a problem if you have functions that exist in multiple libraries.
The most complex thing the linker does is resolve symbols, that is figure out where in memory the location of a variable/function will be when all the code it put together; It does this using hash tables, a data structure we will study soon. (The linker also adds a bunch of header information about formats, DLL locations, offsets, etc. that the runtime system takes into account when spawning a process from the program.)
One problem with static linkage is that it can result in huge executables for tiny programs (e.g., if there's GUI code). Also, the program needs to be rebuilt whenever there's a change to a library.
For these reasons, dynamic linkage was invented.
In dynamic linkage, the linked program does not include the code for a function that exists in a DLL. Instead, it stores the name f of the function, and executes a system call asking for the code for f to be added to the running program. The system call also updates the code that calls the system call, so that it calls f directly (this can be done trivially since f is now in memory).

The problem with dynamic linkage is that a program that worked may fail after libraries are updated. For this reason, many commercial programs are not dynamically linked. (Another problem is that you need to specify the DLL path, it's not always standard.)

4.1 Include files

It is important to understand the role of include files. The include file has no executable code—it just has prototypes for structs and functions, as well as defines. The struct prototypes are needed to allow the compiler to know how much memory to allocate for variables, and in the call to new/malloc. The function prototypes exist to allow the compiler to do type checking, and catch simple bugs like missing arguments.
The actual implementation can be in a seperate .c file or in a library (the former is the case when you are working with your own code, the latter for standard functions). All the C functions that are part of the standard C library are in a single file, usually called `libc.a`. Somewhat confusingly, the prototypes for functions in the standard C library are not in one single header file—rather they are split, e.g., I/O functions are prototyped in `stdio.h`, time and date functions are in `time.h`, etc. The reason is simple—you usually don't need all the functions, and including lots of header files slows things down.
The library itself is just a file that contains object files along with a table identifying where the constituent functions are (to speed up the link process). You can create your own library in Linux using `ar cq libabc.a foo.o bar.o`; you link against it with `gcc -labc ...`. (If it's not in the local directory, you may need to specify the path to it, e.g., by adding `-L/home/adnan/libs/` as an argument to the gcc compile.)

5 Virtual memory

The addresses your program refers to—for data as well as code—are not the actual addresses for the physical RAM of your machine. Instead, the addresses are "virtual" and there is a translation from virtual to physical address done each time your program peforms a read or a write. (This includes references to data as well as code, e.g., the instruction fetch unit translates the branch target address to a physical address.)
The translation is defined by the "page table." It's not as huge as you might think, since the mapping does not map individual VAs to PAs—instead it maps pages, which are 2-8 kByte blocks. Processors accelerate the translation process by providing a Translation Lookaside

Buffer, essentially a very fast cache of commonly used page table entries. On a TLB miss, two things can happen—the processor hardware itself updates the TLB, or it traps to the OS which then updates the entry. (Note that the VA may not exist in the page table, in which case the hardware update routines generates a page-fault which is handled by the OS.)
Virtual memory was introduced to allow applications requiring lots of memory to access the disk as RAM. However, it has another compelling use, which is that it can protect processes from writing to each others memory space—each process has its own page table, and the OS manages the entries to ensure physical addresses don't overlap. When one process is switched out for another process (this is called a context switch), the OS invalidates the TLB entries to keep protection; another solution is to store the process ID as part of the TLB/page table entry.
In some special cases, you do want processes to share memory, e.g., you can use the page-table to implement shared code segments. (You need to keep special attributes along with the page-table entry, e.g,. that processes should not be allowed to write to those locations.) In another example, say one process is doing some processing on packets, and the other is to take those packets and do more processing. It's faster then for them to share the same memory, rather than copy from one process to another. Linux allows this through the `mmap` system call, which sets up page-table entries appropriately.

6 References

There are many books on compilers; I am partial to Michael Scott's [3] book myself. There is only one book I know that talks about nothing but linking and loading [2]. You can get a good idea about low-level implementation issues from this online article [1].

References

[1] A whirlwind tutorial on creating really teensy elf executables for linux. `http://www.muppetlabs.com/ breadbox/software/tiny/teensy.html`.

[2] J. Levine. *Linkers and Loaders*. Morgan-Kaufmann, 1999.

[3] M. Scott. *Programming Language Pragmatics*. Morgan-Kaufmann, 2006.