Sep 12, 2006
Lecture 2: System Programming

September 19, 2007

1 Introduction

In this lecture, we will introduce the basics of systems programming using the language of choice – C language. We also introduce some basic tools for building and managing large software projects (such as an operating system).

The two main operating systems we will learn in this course is Windows and Unix. Unix has many derivatives, including the well-known Linux. However, for teaching purposes (as well as in research) the easiest way to give universal access to a Unix-like environment is to use Cygwin. The reason is that most students' primary computing platform is Windows. If you already have Linux or Apple computers, then you do not need Cygwin. Apple's OS is really UNIX under the hood. Cygwin, originally from RedHat (a linux vendor), is a Unix emulator that sits on top of Windows. So it is really a Windows program, and as such it can share files with Windows (and Windows shares files with Cygwin). It is free and easy to use. In particular, you do not need to dual-boot to use cygwin.

In the Cygwin environment, you get free access to a large set of standard tools, including compilers, text editors, X-window system, spell checkers, graphics tools, X-window systems, networking, etc. We will need the following tools from Cygwin:

gcc (Gnu C compiler)
tar (program for archiving and sharing files)
make (program to organize tasks)
CVS (program to maintain files in a project shared by many users)

An operating system is a large piece of software. To build, maintain and debug large software, we need many kinds of tools. The make and CVS tools are two basic tools we will use.

Exercises

Exercise 0.1: (First Steps into Cygwin)

(i) Please read up the basic information about Cygwin on our class webpage, and download from the web a Cygwin Setup program.

(ii) We recommend a 2-step approach. First use Setup is to simply download a "base" Cygwin: just aaccept all the defaults, although you should choose a directory for storing the download history, e.g., `C:/download/cygwin-files/`. Next, call Setup again, and now choose the following additional downloads: gcc, tar, make, cvs.
(iii) Read up some basic information about Unix. Note that if you accept the defaults, you will be using the bash shell (which we assume in the following). Learn about the following 13 shell commands:

`ls, cd, pwd, mkdir, rmdir, rm,`

`mv, cp, cat, pushd, popd, ln, alias`

Create a directory called "os" in your home directory in cygwin, and create a subdirectory of "os" called "hw". ♢

Exercise 0.2: Write a C program which prints "Hello World!" on the screen. Use any C program you know. Store it as the file `hello.c` in the "hw" directory. Compile it using gcc, and execute it. If you are not familiar with C, read our webpage on C programming. ♢

Exercise 0.3: Learn about environmental variables in Unix, and about the roles of the variables:

`PATH, HOME, PS1`

Learn how to use the shell commands `printenv, export`. What does the following command accomplish?

```
> export PATH=.:/home/yap/bin/:‘printenv PATH‘
```

(Note the special reverse quotes ‘...‘ used here.) Learn about the .bashrc files. How can you use .bashrc files? ♢

End Exercises

On Text Editors. Furthermore, I strongly encourage you to learn some **keyboard-based editor** (KBE) for program development. The advantages of such editors are often hidden from the general public. Most users only know WYSIWYG (What-you-see-is-what-you-get) editors. The problem is that precisely in its name – you cannot get what you cannot see. In contrast, we do not need to "see" something in order to execute commands in KBE's. If you can touch type, you can accomplish tasks with more power and speed than any WYSIWYG editor. Why is this so? Because by its very nature, WYSIWYG editors requires you to "see" and hence coordinate your eye with your finger actions! This slows you down – in contrast, in KBE's you can issue commands directly from the keyboard without even looking at the screen. For instance, you can type in 3 key strokes,

`50d`

to delete the next 50 lines of text. No WYSIWYG editor can hope to match this in speed. Or, you can type

`:g/WYSIWYG/s//KBE/g`

to search the whole file for the string "WYSIWYG" and replace each by "KBE". In contrast, imagine a WYSIWYG editor would have you call up a search panel, locate the place to type your search string "WYSIWYG" and replacement string "KBE", and possibly to keep clicking the replace command for each occurence of WYSIWYG!

The learning curve for WYSIWYG editors is small (to its credit), but then there is not much more to learn after the initial introduction! In contrast, keyboard-based editors has a steep initial learning curve and you can keep learning new tricks for a long time. If you have no prior commitment to a particular KBE, I highly recommend the GVIM editor. This is a GUI-based enhancement of original Vi editor. Some people like the EMACS editor. Although both are equally powerful, I feel GVIM is more efficient (in terms of key strokes for accomplishing tasks). You can download GVIM through the Cgywin setup (the cygwin version may require X-windows, and so you may want to directly download a Windows-based GVIM from the internet). If you already know Vi, you can immediately use GVIM (but there are many, many new features which you can slowly learn to use). Note that the non-GUI version of GVIM is called VIM, but there is no reason to use VIM if you can get GVIM. One advantage of GVIM over VIM is that you can use the familiar cut-and-paste paradigm and other similar tasks of WYSIWYG editors. Admittedly, this is sometimes more convenient to use. *In short, real programmers use KBE's.*

2 Mechanics of Compilation

We assume that you have read our webpage on C programming, and can write and compile a "Hello World" program.

The simplest way to compile a C program `myprog.c` is to type:

```
> gcc myprog.c
```

Assuming successfull compilation, the output is an executable file called `a.out` (in unix) or `a.exe` (in cygwin). But in this invocation, you are calling more than just the compiler!

1. First, the compiler invokes a PREPROCESSOR to handle lines that begins with a "#". Macro calls are also handled by the preprocessor.

2. The compiler (gcc) translates your preprocessed program into assembly language *but in symbolic terms*. E.g., a store instruction will read "STORE VAR_A" where VAR_A is a variable name.

3. This output is processed by another program called the ASSEMBLER that converts "STORE VAR_A" into a binary word (with VAR_A resolved into

a numeric address). If you stop the process the this point (with the `-c` option) (see below).

4. Next, gcc calls the LINKER (or link editor) which builds the executable program by linking the object file to the library routines.

Suppose you want to prevent the compiler from calling the linker and loader. Then you type

```
> gcc myprog.c -c
```

(so -c means "just compile"). This will produce the output file `myprog.o`, called an "object file".

If you want to create an executable from myprog.o, you can now call the loader directly:

```
> ld -lc myprog.o
```

This will produce the `a.out` or `a.exe` you expected. NOTE: that `ld` must be explicitly given the library `-lc`.

Q: Why would you want to do this round-about way to producing `a.exe`? A: Your `myprog.o` file may contain routines that can be linked to other programs that wants to use them. This saves time on re-compiling (an issue in large projects).

This brings us to the subject of libraries. Every program refer to library routines. For instance, `main()` is actually the name of a library routine. The standard C library routines are found in a library file named `libc.a`. This library file is usually found in the system include directory called `/usr/lib`. The file `libc.a` is basically a list of a bunch of object files (`*.o`)!

CONVENTION: All library files are named `libXXX.a` where `XXX` is any identifier. To ask the compiler to search the library `libXXX.a`, you give it the option `-lXXX`. Thus, if `XXX=c`, we use the option `-lc` as in our illustration.

Suppose your program uses a mathematical library routine like "sqrt". This is in another library called `libm.a`. In this case, you MUST tell gcc to search the `libm.a`. You should now invoke

```
gcc myprog.c -lc -lm
```

If you have several libraries to search, give each one as a separate `-lXXX` option. Below, we provide more notes on Runtime libraries and Shared libraries.

EXERCISES

Exercise 0.4: What is odd about this command line:

```
> gcc myprog.c -lm -c.
```

♢

END EXERCISES

3 Run-Time Organization of C Programs

The compiled code is loaded into memory in three segments (code, data and stack):

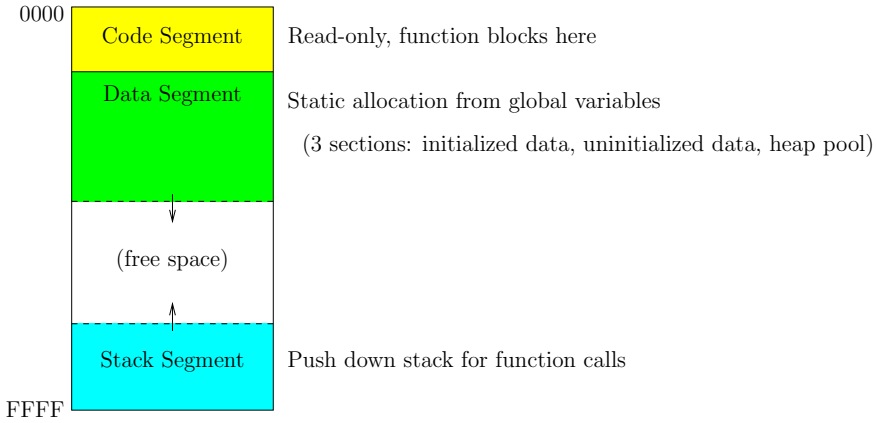

Figure 1: Process Image

As illustrated in Figure 1, these segments are laid out in a specific order: (starting from lowest address `0000` to highest `FFFF`) (1) code segment, (2) the data segment, (3) a gap of free space, and finally (4) the stack segment.

Because the gap is flexible, we can arrange the top of the stack in the stack segment to coincide with the highest address (`FFFF` in the figure). Then as the stack grows during runtime, it grows into the free space found it the gap.

The data segments are subdivided into initialized and uninitialized data sections, and a third section for heap allocation. The initialized and uninitialized data sections have fixed size. Like the stack, the heap can grow during run-time. So it is arranged to be adjacent to the gap of free space.

During runtime, we get an error when the gap between the heap and stack is reduced to nothing. We either have a "stack overflow" or an error while attempting to obtain space from the heap.

4 System Calls

In Unix and C, there is no difference between a system call and an ordinary function call! The main thing is to make sure that you have the proper header files included. Two common ones are

```
#include <unistd.h>
#include <sys/sysinfo.h>
```

```
/*************************************************
 * io.c: minimal cat version (from Kernighan and Pike)
 * This program will echo whatever you type.
 * To terminate, type control-D
 *************************************************/
```

```
#define SIZE 512 // arbitrary

int main() {
char buf[SIZE];
int n;

while ((n = read(0, buf, sizeof buf)) > 0)
write( 1, buf, n);
return(0);
}
```

The function `read(...)` is actually a system call.

5 On Processes

Processes are the main entities in an operating system. Indeed, one view of an OS is as a manager of processes. Homework 1 gives you an brief introduction to the programming of processes.

In order to do homework 1, you will need to know something about `exec*(...)` and `fork()`. Both are unix system calls. All unix system calls can be accessed from a C program. They are found in the Standard C Library (named `libc.a`), and sometimes you may need to include the header file:

```
#include <unistd.h>
```

6 Exec*

The "exec" system call is used to execute a program from within a process. Suppose your process `Q` is currently executing a program `q`. If `q` wants to execute another program `p` it can make the "exec" system call with suitable arguments. Then the current process image is replaced (overlaid) by the image for the program `p`. There is no return to the original program `q`.

To call "exec", you need to the following arguments: where to find the executable file for program `p`, the name of the program `p`, and any arguments that must be passed to `p`.

Passing these arguments to "exec" might be troublesome:
(1) The arguments to be passed to `p` is quite arbitrary, depending on `p`. For instance, exec does not know in advance how many arguments `p` takes.
(2) Second, it may not be obvious where the executable image for program `p` is located, even if we know its name. E.g., `p` might be standard system programs like `date` or `ls`, we may not know where `date` is located in the OS. This would require searching the directories in the environmental variable `PATH`.

For these two reasons, we provide several versions of the "exec" to provide various conveniences in making this system call. In fact, there are 7 variants:

```
execl, execle, execlp
execv, execve, execvp
exect
```

Note that they are placed in 3 groups (l, v, t). We shall collectively call them `exec*(...)`. To begin, we illustrate with an easy-to-use version: `execlp`.

Suppose your program `q` wants to list the files in the current directory. It can use the following system call:

```
execlp("ls", "ls", NULL);
```

The first argument tells us where to find the executable file for `ls`. This might be in `/bin` or `/usr/bin` or some other directory. The version `execlp` will search the directories of `PATH` to look for `ls`. The remaining arguments will be passed to the `ls` program. Since the number of arguments is unknown, we have the convention to adding `NULL` to the list of arguments to signal the end of the list.

So the `ls` program receives only one argument, namely, its own name. Recall the convention that all C programs can access their list of arguments from the `argv` argument in the `main` program. Moreover, the first argument is always the name of the program itself.

EXERCISES

Exercise 0.5: Write the system call if you want to list the help message for `ls`. See answer at the end of this lecture. ♢

Exercise 0.6: What is the system call if you do not want to use the `PATH` searching facility but know the exact location of the `ls` program? See answer at the end of this lecture. ♢

END EXERCISES

We just described the "lp" version of `exec*(...)`. The "l" refers to "list" since the arguments are explicitly listed by their list elements. The "p" refers to the facility of searching directories in `PATH`. Sometimes, it is convenient to gather the list of arguments into an array (or "vector"). This is the "v" version `exec*(...)`:

```
execvp("ls", argp);
```

where `argp` is an array of points to the arguments, structured exactly like `argv` in the standard `main()` subroutine.

This version is useful if this invocation of `ls` may have different number of arguments at runtime. You would have to assemble (at runtime) these arguments into `argp` just before you make the system call.

7 Fork

What if the process `Q` want to return to the calling program `q` after executing `p`? It can create another process `P` just for the sole purpose of executing `p`. The system call to create another process is called `fork()` which takes no arguments.

When the process `Q` calls `fork()`, a new process `P` is created that is identical to the calling process. We call `P` a **child process** of `Q` (the **parent process**). There is only one difference: `fork()` returns a **process ID** (of type `int`). The value returned to `Q` is the process ID of `P`, but the value returned to `P` is 0. Therefore if we want `P` to behave differently from `Q` we must have code that tests this returned value and branch on this:

```
int status;
if (fork() == 0)
    execlp("ls", "ls", NULL);   /* child */
wait(&status);                  /* parent */
```

The child process `P` will reach the `exec*(...)` statement in this code, and since there is no return from an `exec*(...)`, it will not see the following wait statement.

The parent process `Q` will execute the wait statement, meaning that it will wait until its child process terminates. In general, if it has many children, it waits until the first child that terminates. The value of the `status` variable returned by `wait` has the child's exit status stored in its low-order eight bits (where `0` indicates normal return). The next eight bits of `status` is the argument given to the `exit` statement or the `return` statement of main, whichever terminated the child process.

CODA

REFERENCE: Kernighan and Pike (Chapter 7).

Q: *Write the system call if you want to list the help message for* `ls`.

ANSWER:

```
execlp("ls", "ls", "--help", NULL);
```

Q: *What is the system call if you do not want to use the* `PATH` *searching facility, but know the exact location of the* `ls` *program? NOTE: this method might be safer in the sense that someone cannot change the search path and cause you to execute a wrong file.*

ANSWER: Give the absolute path name of the program file, instead of a relative path name. E.g., if the absolute path name for the `ls` program is `/usr/bin/ls.exe` then issue:

```
execlp("/usr/bin/ls.exe", "ls", NULL);
```