Sep 12, 2006
Lecture 3: Cygwin and Makefile

September 5, 2007

1 Introduction

In this lecture, we will introduce the basics of the Cygwin Operating System, and a basic tool called Makefile.

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. This is based on the assumption 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, 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.

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

gcc (Gnu C compiler)

tar (program for archiving and sharing files)

Makefile (program to organize tasks)

Furthermore, I strongly encourage you to learn some keyboard-based editor. The advantages of such editors over a WYSIWYG editor is the power and speed with which you can develop software using the latter. The learning curve for WYSIWYG editors is small, but there is also not much more to learn after the initial introduction! In contrast, keyboard-based editors can be very complex and you can keep learning new tricks. If you have no prior commitment to a keyboard editor, I strongly recommend GVIM (the GUI-based enhancement of VI). Some people like EMACS. You can download GVIM directly or obtain it through Cgywin (I think they only distribute VIM, which is the non-GUI version).

An operating system is a large piece of software. To build and maintain and debug large software, we need many kinds of tools. A simple but powerful tool I want you to learn is Makefile.

You can get basic information about these on our class webpage.

2 Exec

In homework 1, you need to make two systems calls: fork() and exec*(). Note that almost all calls to fork() is followed by an exec*() call. Another common system call is wait().

We will explain "exec" here. Basically, "exec" is called to execute a binary file (i.e., compiled, executable program). These binary files might perform basic tasks provided by an operating system, such as listing a directory.

There are actually 6 or 7 variants of exec(), and none of them are actually called "exec". We write exec*() to refer to all these variants:

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

These are available from the **Standard C Library** (libc.a). To make any of these system calls, your C program must include the header file "unistd.h". Here is the calling sequence for `execlp` which is used in our homework:

```
#include <unistd.h>

int execlp (
File,
Arg0 [, Arg1, ...], NULL)

const char *File, *Arg0, *Arg1, ...;
```

The first argument is the name of an executable file, and the remaining arguments `Arg0, Arg1`, etc, depends on the arguments needed by the executable file.

Since execlp does not know how many arguments are needed by your executable file, you must explicitly terminate your argument list with the NULL argument. Since NULL is literally zero, you can write 0 in place of NULL.

A standard convention in UNIX i that `Arg0` should be the name of the executable file itself. Hence, every executable program takes at least one argument. E.g.,

```
execlp("sleep", "sleep", "10", 0)
```

where the program "sleep" sends the calling process to sleep for 10 seconds, and

```
execlp("echo", "echo", "Hello World!", NULL)
```

will call the "echo" program to print on the screen the string "Hello World!".

Return Values. Upon successful completion, the exec subroutines do not return because the calling process image is overlaid by the new-process image. If the exec subroutines return to the calling process, the value of -1 is returned and the errno global variable is set to identify the error.

Other Variants. The first argument of exec*() is the name of the program to execute. E.g., execl("ls", ...) says that we want to execute the unix "ls" program. The remaining arguments depends on the variant AND on the program to be executed:

System Call	Argument Format	Environment Passing	Path Search
execl	list	auto	no
execv	array	auto	no
execle	list	manual	no
execve	array	manual	no
execlp	list	auto	yes
execvp	array	auto	yes

The arguments to the program to be executed can be given in ARRAY format or in LIST format:

E.g., execl("ls", "ls", "-lt", NULL);

In this example of the list format, we see that the list of arguments is terminated by a NULL argument.

The environment for executing the program can also be passed explicitly or implicitly inherited from the current environment.

Path searching uses the standard unix concept of a list of search paths defined in the environment. A given file name is searched for in each of the paths until one is found (or failure reported).

The exec subroutine, in all its forms, executes a new program in the calling process. The exec subroutine does not create a new process, but overlays the current program with a new one, which is called the new-process image. The new-process image file can be one of three file types:

* An executable binary file in XCOFF file format.

* An executable text file that contains a shell procedure (only the execlp and execvp subroutines allow this type of new-process image file).

* A file that names an executable binary file or shell procedure to be run.

2.1 Examples

1. To run a command and pass it a parameter, enter:

```
execlp("li", "li", "-al", 0);
```

The execlp subroutine searches each of the directories listed in the PATH environment variable for the li command, and then it overlays the current process image with this command. The execlp subroutine is not returned, unless the li command cannot be executed.

Note: This example does not run the shell command processor, so operations interpreted by the shell, such as using wildcard characters in file names, are not valid.

2. To run the shell to interpret a command, enter:

```
execl("/usr/bin/sh", "sh", "-c", "li -l *.c", 0);
```

This runs the sh command with the -c flag, which indicates that the following parameter is the command to be interpreted. This example uses the execl subroutine instead of the execlp subroutine because the full path name /usr/bin/sh is specified, making a path search unnecessary.

Running a shell command in a child process is generally more useful than simply using the exec subroutine, as shown in this example. The simplest way to do this is to use the system subroutine.

3. The following is an example of a new-process file that names a program to be run:

```
#!  /usr/bin/awk -f  for (i = NF; i > 0; --i) print $i
```

If this file is named reverse, entering the following command on the command line:

reverse chapter1 chapter2

This command runs the following command:

/usr/bin/awk -f reverse chapter1 chapter2

Note: The exec subroutines use only the first line of the new-process image file and ignore the rest of it. Also, the awk command interprets the text that follows a # (pound sign) as a comment.