Lecture II
System Programming

§1. Introduction

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

¶1. 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 1.1: (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`. In the future, you can call Setup to look for any other software that you might need.
(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 shell commands:

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

`rm, mv, cp, ln, cat,`

`printenv, export, alias,`

`pushd, popd, dirs`

(iv) Take the following actions:
– Create a directory called "os" in your home directory in cygwin.
– cd to "os" and create two subdirectories: "lect" and "hw". Download our lecture notes into lect.
– cd to "hw" and create a directory called "h1" for your first homework. Download homework 1 there.
– cd to "os" and create a link (ln) called "curr" to hw/h1. Thus, you will have instant access from os to the current homework.
(vi) Customization. Your general unix environment is customized in two ways: by the values of environmental variables, and by aliases. These customizations is usually stored in special shell script files such as `$HOME/.profile` or `$HOME/.bashrc`. Note: if your shell is not bash, then you have analogues such as `$HOME/.cshrc` for csh shell. Moreover, instead of `.profile`, you might have `.bash_profile`.

The `.profile` file is executed only once, when you first log into the system. So it is known as a **login script**. But scripts such as `.bashrc` are automatically read each time you start up a shell and are known as **shell startup scripts**. Since Unix tend to start up new shells for each task, shell startup scripts are read very often. You should decide which commands are better in shell startup scripts and which should be in login scripts. But making a mistake in this decision is normally not very consequential – it just makes your shell start-up time slower slightly. Typically, you should to force `.profile` to read `.bashrc` by including a statement such as

```
if [ -f ~/.bashrc ]; then
    source ~/.bashrc
fi
```

Please take a look at these script files. Notice that they are hidden files (dot-files) and are stored in your home directory (value of the HOME environmental variable).

Another common device is to have these shell scripts read other shell scripts. E.g., have a file `.env` to set environmental variables, and have `.profile` read this file.

Actually, there are also system versions of these shell scripts, e.g., `/etc/profile` might contain initializations suitable for all users. These may be read before individual user scripts are read.

(v) The commands `pushd, popd, dirs` forms a group. This allows you to juggle between moving among a group of directories that are pertinent to your current task. Pushd and popd allows you to maintain a stack of "directories" so that you can jump to any one of them with a short command, and to reorder this stack. To view this stack, just type `dirs`.

I like to alias them as follows:

`alias pu=pushd; alias po=popd`

If you like this, put this alias in your `.bashrc` file.

In the simplest case, if you just shuttle between two directories, you just need to issue the commands `pu` or `po` to switch back and forth. All these 3 commands take small integer arguments (including +0 and -0).
(vi) Other useful unix commands or standard programs: `which, whence, file, wc` ♢

Exercise 1.2: 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 1.3: Read up about environmental variables in Unix.

1. What are the roles of the following variables?
   `PATH, HOME, PWD, PS1`
2. Read about 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.)
3. Read about the .profile or .bashrc files. How can you use these files?

♢

Exercise 1.4: Read up about Unix file systems.

1. What are the following special directories: "/", ".", ".." ?
2. How do you create, move and delete directories?
3. How do you create, move (rename) and delete files? (NOTE: for this question, "files" do not include directories)
4. What is one operation you can do with a files which cannot be done on directoris?
5. Basic file permissions are based on "permission bits" where each bit, if turned on, gives permission for a category of user to perform a specific action. There are (i) three categories of users with respect to a file, and (ii) three kinds of actions that can be performed on file. This gives rise to 9 permission bits. Please describe these bits. [Besides these 9 bits, more sophisticated permission control are also used.]

6. How do you change permission bits on a file?
7. What information is displayed when you do "ls -l" on a file?

◊

End Exercises

¶2. On Text Editors. 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.*

Incidentally, for the same reasons as above, there many tasks that you want to perform on a computer that is best done using a powerful command shell (like bash). For instance, if you want to navigate directories, and search files, the using a shell (with keyboard commands) are vastly superior to the usual "Window Explorer".

§2. Unix

We will learn Unix as we go – please read our webpage FAQs for this.

Here are some self-test to see if you have a working knowledge of Unix.

1. Your Unix run-time environment is customized by setting certain variables called **environment variables**.
   (a) Are such variable special in anyway?
   (b) How do you find out what are your environment variables?
   (c) How do you set their values?
   (d) Name 3 such variables
   (e) How are each used?
2. Each of the following special characters are related to Unix processes. For each of the following, describe one situation where they are used.

CHARACTER	NAME
cntl-D	control-D
cntl-Z	control-Z
|	pipe
<, >	redirections
&	ampersand

3. For each of the following shell commands, (1) describe the meaning of the command, and (2) describe a situation where it is useful. Note that you can concatenate several shell commands into one command by placing a semicolon (;) between consecutive commands.

   HINTS: We suggest you actually test this command on your system. In Unix systems, you can get an online manual for a particular `COMMAND` by typing `> man COMMAND`. For instance, `> man ls` will tell you all the options available for `ls`.
   (a) ls `ls -a`
   (b) `ls -tl /bin`
   (c) `ls /bin/m*`

(d) `ls -sF`

(e) `ps -s`

(f) `ps -sW`

(g) `echo "Reading This File"`

(h) `which ls`

(i) `file hw1.c`

(j) `wc hw1.c hw2.c`

(k) `pushd "/cygdrive/c/Program Files"; ls`
NOTE: this will not work if you are not in Cygwin. Instead use `pushd /usr/bin; ls`.

(l) `popd`

(m) `alias ls="ls -sF --color=tty"`

(n) `ln -s "/cygdrive/c/Program Files/Vim/vim71/gvim.exe .`

§3. Mechanics of Compilation

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

¶3. Assembling and linking. 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 "only 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`.

¶4. Separate Compilation. Why would you want to do this round-about way to producing `a.exe` from `myprog.c`?

A: Your `myprog.o` file may refer to routines that are useful to other other programs as well. Suppose you have such a routine called

```
int useful(char *, int)
```

It is then good to separate out this routine, to store it in a file `useful.c` by itself. Then we can compile it using

```
gcc useful.c -c
```

to produce the object file `useful.o`. Note that `useful.c` need not have a `main()` routine since we do not intend to run `useful` by itself.

Now, you do not need to define `useful(...)` in `myprog.c` in order to use it. But we do have to tell the compiler that `useful(...)` has been defined elsewhere. To do this, we use the `extern` command near the beginning of the file `myprog.c`:

```
extern int useful(char *, int);
```

Let us suppose `useful(...)` is defined in a file `mylib.c` containing all the useful routines. You can then compile `mylib.c` to obtain `mylib.o`. Finally you can compile `myprog.c` using:

```
> gcc mylib.o myprog.c
```

This method of compiling `myprog.c` is known as **separate compilation**. Note that `mylib.o` might contain many other functions that is not needed by `myprog.c`, but no matter.

This saves time on re-compiling (an issue in large projects). that might be used by various programs.

Now, besides inserting the extern command, there might be other variables, constants or functions used by `useful()`. Every program invoking `useful()` must extern them as well. This is cumbersome.

¶5. Include Files The best way is to solve this problem is to collect all these declarations of mylib.c into a file called `mylib.h` called a **header file**. Then, `myprog.c` just have to include this header file if it wants to invoke functions in `mylib.c`:

`#include "useful.h"`

Note that the extension `.h` is standard for header files. Furthermore, unlike the standard header files, such as `stdio.h`, we enclose this header file between double-quotes (" . . . ") instead of angle brackets (< . . . >). These two different formats simply determine where `gcc` searches for library files: one is determined by PATH, and other is determined by the system.

Here are some of the most important include files that you may need:

```
#include <stdlib.h>     --standard library
#include <stdio.h>      --standard I/O
#include <unistd.h>     --unix standard
#include <fcntl.h>      --file control
#include <sys/types.h>  --system types
#include <sys/stat.h>   --system statistics
#include <signal.h>     --signal library
```

¶6. Libraries This brings us to the subject of libraries. Conceptually, `mylib.o` is already compiled library.

But in C programming, we add another facility: we can pack together many dot-o files into one single dot-a file (the "a" here stands for "archive"). The advantage of the dot-a format is that it supports a fast search for library routines, and possibly for compression to afford a smaller file. So we usually call dot-a files **libraries**.

To access `mylib.a` from any location, there are standard locations where the compiler looks for libraries. For instance, the compiler often search in the directory `/usr/lib`.

Every executable program refer to at least one standard library routines, namely, `main()`. This routine is part of the standard C library found in the library file named `libc.a`. Thus, the file `libc.a` is a concatenation a series of object files (`*.o`)!

¶7. Library Conventions 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.

¶8. Dynamic and Shared Libraries. So far, we have implicitly assumed the conventional library, also known as **static library**. There is a compiler option is to ask for **dynamic libraries** instead. Using static libraries, when you call a library routine, a copy of the assembled code for the routine will be incorporated into the object code for your program. With dynamic libraries, only a "stub" for that library routine is placed in your object code. At run-time, the actual library routine will be loaded. This is called **dynamic loading**. But the library routine might already be loaded (by some other process) and you can share this routine. Hence dynamic libaries are sometimes called **shared libraries**. Logically, dynamic loading and sharing are distinct features but in practice the two goes hand-in-hand. One consequence of using dynamic loading is that the object file is considerably smaller than one based on static libraries.

By convention, corresponding to a static library `libXXX.a`, the shared library files is named `libXXX.so`.

¶9. Modules and Header Files.

- In all programming languages, we need some method for dividing a large piece of code into functional units, called **modules**. The above discussion already gives us many tools for creating and using modules.
- In the simplest case, assume each module is stored in its own file.

  A **header file** is normally used to specify the "interface" to a module.

  In C language and its derived languages (C++, etc), the header files are normally given the ".h" extension. We call these **dot-h** files or header files, as described earlier. However, Java has a different solution to this problem and it does not use header files.
- For example, in our Toy OS projects, be writing various projects modifying an original OS. We start out by giving you four files

  ```
  the.h, the.c, tos.h, tos.c
  ```

  Note that the files occur in pairs: a dot-c and a doh-h file forms a pair. Each pair forms a module (the hardware module and the OS module). The dot-h file specifies the interface, and the dot-c file provides the implementation.

  When another program wants to use the module, it only need to include the dot-h file from that module to compile. When it is ready to link, it can be linked against the dot-o file of that module.

  We can compile stm.c into a stm.o file. Any OS that is written for STM machine can just link to stm.o. As long as the "hardware" does not change, we do not have to recompile stm.c.
- All the C libaries are organized in this way. For instance, to use the functions in the "standard I/O" library, you just have to include `stdio.h` in your program. You can then link to these precompiled routines such as `printf` and `scanf`.

- Modules might depend on other modules (the dependency diagram ought to be acyclic).

  The "make" program allows one to automatically maintain dependency information, so that if a dependent file is modified, the "dependee file" must also be recompiled.

- Now, it is possible to include the .h files many times. This will cause compiler errors.

  This could lead to the error of multiple definition. To avoid this, we insert an **include guard** into each .h file. For the file stm.h, we might write:

```
#ifndef H_STM
#define H_STM

... stm header information

#endif
```

  Here, `H_STM` is a unique variable associated with the file. Note the convention of using CAPS and also *H*-underscore.

- THE DOT-H SOLUTION: For each file (like stm.c), we create a file called "stm.h". The file stm.c must include stm.h, and so must any other module that depends on stm.c. The include statement is this:

```
#include <stdio>            // standard library includes first
#include "stm.h"            // user-defined includes follow
                            // NOTE: the <...> versus "..." construct
                            //   which distinguishes between standard
                            //   and user-defined libraries.

... stm.c implementation here ...
```

  The advantage of this is that we have separated the INTERFACE and IMPLEMENTATION into dot-h and dot-c files (resp).

- WHAT GOES INTO A DOT-H FILE?

  function headers (stubs), global and static declarations, type definitions, defined constants.

Exercises

Exercise 3.1: What is odd about this command line:

`> gcc myprog.c -lm -c`. ♢

Exercise 3.2: You want to find out more about the library file `libc.a`.

1. How do you locate it in your system?
2. The answer to your previous query comes back as:
   `/usr/lib/gcc/3.4.4/../../../lib64/libc.a`
   What is the canonical full name of this file?
3. Now you cd to the directory containing `libc.a`. How do you check if the function `fork()` is defined here?

◇

Exercise 3.3: Write a toy shell program called `tsh.c` that executes unix commands. It should know how to process command lines containing the special tokens "&" and ";", but not "—". We will provide the basic routine called `parseCommand` and also the skeleton of `tsh`. We want you to do separate compilation of `parseCommand.c` and `tsh.c` (see the provided Makefile to understand this).

Extra Credit Features:
(1) Allow the user can change the prompt string.
(2) Can you make "cd" work properly? ◇

End Exercises

§4. Run-Time Organization of C Programs

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

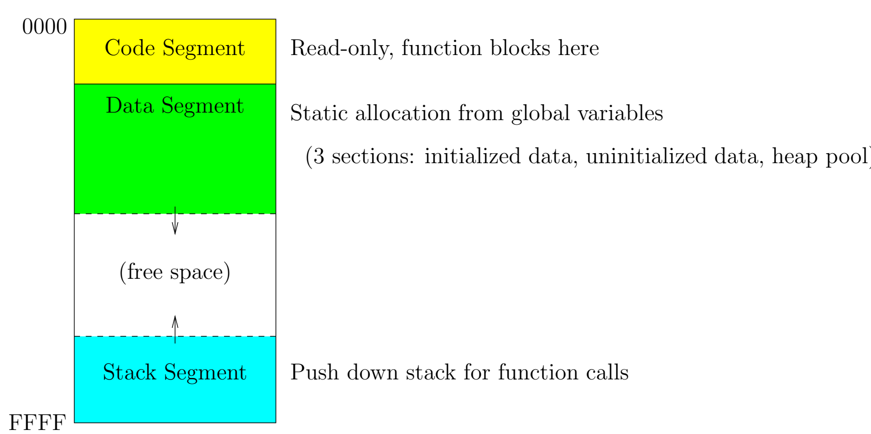

Figure 1: Process Image

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.

§5. 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/systeminfo.h>
```

```
/**************************************************
 * io.c: minimal version of "cat" program (from Kernighan and Pike)
 * USAGE:
 * 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 `int read(fd, buffer, maxsiz)` is actually a system call to read from an opened file (denoted by the `fd`). Read will read at most `maxsiz` characters into a character buffer `buffer`. It returns the number of read characters (-1 if error). Normally, this numer is equal to `maxsiz` unless the file terminated.

§6. 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()`Ḃoth 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>
```

§7. 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 7.1: Write the system call if you want to list the help message for `ls`. See answer at the end of this lecture. ♢

Exercise 7.2: 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" in "lp" refers to "list" since the arguments are explicitly listed by their list elements. The "p" in "lp" 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", argv);
```

where `argv` is an array of pointers 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 `argv` just before you make the system call.

Exercises

Exercise 7.3: Write a routine called

```
setup(char inputBuffer[], char *argv[],int *background)
```

which takes an input line `inputBuffer` and outputs the arguments as an array of strings `argv` as above. The last array element in `argv` is a null string. Thus `argv` can be used as argument to `execvp`. ♢

End Exercises

§8. 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.

¶10. Quirks. In cygwin, we find that if a process is not waited for, then its output status is queued. Therefore, subsequent waits will get a status from this queue! So subsequent waits may return instantly, because this queue is non-empty.

This can cause anomalies in our "toy shell" programming exercise. Recall that in this project, when we issue a "background command" (as indicated by the "`&`" character, the job is to be executed in the background. That means the parent process does not have to wait for the termination of this child (normally it waits). However, once you performed ANY background task, you will find that all subsequent command appear to execute in the background as well (as if the parent process did not wait for the child). Of course it did wait, but it returned instantly based on the queued statuses.

To solve this problem, you must save the CHILD's PID and then wait explicitly for this PID's termination. Note that wait() returns the PID of the terminated child, so you must compare it to the PID you are waiting for.

§9. Unix I/O

In Unix, all I/O is performed on files. The easiest functions to use are the high level formatted I/O functions.

¶11. Formatted I/O. We assume that you know the basics of formatted output using `printf` and formatted input using `scanf`. E.g.,

```
printf("Input an int, a float, and a string (separated by commas):  ");
int n;
float x;
char name[12];
scanf("%d, %f, %s", &n, &x, name);
```

Instead of `printf`, `scanf`, we also have the file version:

```
int fprintf(FILE *stream, char *format [, vars...]);
int fscanf(FILE *stream, char *format [, pointers...]);
```

and string version:

```
int sprintf(char *inputBuf, char *format [, vars...]);
int sscanf(char *inputBuf, char *format [, pointers...]);
```

¶12. Opening and Closing Files. Going to files, suppose you have a file named "myfile". You can open it for reading and writing by the following system calls:

```
#include <stdio.h>

FILE *fp; // FILE is a structure defined in stdio.h
ifp = fopen("myfile", "r"); // "r" for reading
ofp = fopen("myfile", "w"); // "w" for writing
                // Other possible modes "a", "r+", "w+", "a+"

// (1) Formatted I/O:
fprintf(ofp, "Please input an int:  ");
int n;
fscanf(ofp, "%d", n);
// (2) Character I/O:
char c = getc(stdin); // get a char from stdin
putc(stdout,c); // put it out on stdout
// (3) String I/O:
#define MAXLINE 80
char line[MAXLINE];
fgets(line, MAXLINE, ifp);
fputs(line, ofp);
```

Note that `fprintf` and `fscanf` are the file versions of `printf` and `scanf`. The only difference is that the file versions accept a file pointer as their first argument.

We also have `fclose` and `fcreat` to close and create new files.

We call `ifp, ofp` **file pointers**, pointing to a file table structure which the system maintains to store information for each opened file (e.g., read/write permissions, etc).

The remark that all I/O is performed on files is not surprising, until you realize that even I/O to terminals (screens) are done through files! In fact, there are three special files whose file pointers are called

`stdin, stdout, stderr`

for **standard input**, **standard output** and **standard error**. These are automatically opened each time you start a new process, and they all initially defaults to the terminal.

So instead of file pointers, Unix also gives an alternative method to keep track of opened files. These are called **file descriptors** and they are basically integers. They are just indices into the file structure table for the current process. Thus, the file descriptors for `stdin, stdout, stderr` are just 0, 1, and 2. Alternatively, use the more descriptive names of

`STDIN_FILENO, STDOUT_FILENO, STDERR_FILENO`

Using file descriptors, the above code corresponds to the following:

```
#include <stdio.h>

int fd; // fd is a file descriptor
fd = open("myfile", 0); // 0=read, 1=write, 2=read+write

write(fd, "Hello File!", 12);
```

Similarly, we have `close()` and `creat()` to close and create new files. NOTE: if you have a file descriptor `fd`, you can open a file using `fdopen(...)` given by:

`FILE * fdopen (int fd, const char *type);`

HINT: most system calls returns an error code if the call fails. The error code is stored in the variable `errno`. You should always check for error possibility, and use a useful error function found in the `stdio.h` called `perror()`. You can use this to print the system message associated with `errno`, and include your own message. E.g.,

`if (open("myfile", 0)<0) perror("failed to open myfile");`

¶13. Stream I/O and Buffers Actually, the functions in `stdio.h` are performing buffered I/O. Files which are automatically buffered are alled **streams**. Hence the `FILE` data type is also a stream data type. The buffered routines have names that are normally begins with an "`f`" (e.g., `fopen`, `fflush` etc. Because of the buffering, sometimes I/O may not do the expected thing (e.g., screen output may appear out of sequence). To solve this problem, you need to clear the buffer: for input buffers, the contents are discarded, for output buffers, the contents are written out to the output files. This is done by calling the functin `fflush()`. E.g. to flush the standard output (which is buffered), you say

```
fflush(stdout);
```

If you call `fflush()` with no arguments, all the buffers in the currently buffered files will be cleared.

¶14. Filters and Pipes The philosophy of Unix to write useful, re-usable program modules is often based on the concept of filters: a **filter** is basically a program P that transforms a data stream to produce a new data stream.

In order to use a filter P, we need to create a stream that servers as input to P, and produces the output of P as another stream. This output of P can be fed to another filter Q and so on. The process of turning the output of P into the input of Q is accomplished by a **pipe** and is easily done at the shell level as follows:

where we assume P produces an output stream with no explicit inputs.

§10. Make Program

A programming project usually involves many individual tasks. These tasks may require certain input files and certain output files, and they may depend on each other. The **make** program is a tool to collect these tasks, and to automate the dependency information.

Consider the following simple Makefile:

```
p=hello

gcc: \$(p).c \$(p).h
------>gcc \$(p).c  -o \$(p).exe
------>\$(p).exe
```

This program assigns the variable `p` to the string `hello`. Henceforth, to get this string, we simply type the sequence $\$(p)$. We see one **target** called `gcc`, with two **dependencies** named `hello.c, hello.h`. The actions associated with this target are given in the next two lines: first call `gcc` to compile `hello.c` and output the file as `hello.exe`. Next, call the executable program `hello.exe`.

Suppose we type `> make p=world` while at the directory containing this Makefile. What will happen? This will override any assignment to `p` within the Makefile, and `world` will be assigned to `p`. This override feature is very useful.

¶15. Meta Characters. Make files are text files and are line oriented. There are two special characters in processing make files.
– Comment character is #, and everything from this character and beyond within the current line is discarded.
– The continuation character is
. Conceptually, this means the newline character following
is discarded and thus two lines are joined into one.

¶16. Rules. Fundamentally, a makefile consists of a set of of **rules** where a rule has three parts

1. Target part
2. Dependency part
3. Action part

Each Target part is just a sequence of one or more names, separated by white spaces (but not newlines). thus target part must belong to one line of text. These names are called **pseudo-files** because the original intent is that they should be file names. But even if they do not refer to files, that is OK, as we will see.

Suppose the target and depency parts are both file names. If we make a particular target name, then we compare the last access date/time of the target file to the last access date/times of the dependent file. If any dependent file has a latter date/time, it means that we must take step to (1) recursivly execute any dependent file targets and (2) execute out the action associated with the current target.

Similarly, the Dependency part is a sequence of zero or more pseudo-files which are separated by white spaces (but not new lines).

Finally, the Action part is just a sequence of zero or more shell commands, each belonging to its own line. The line must have a tab as the first character.

The first line of the rule contains the Target part, followed by a colon (:), followed by the Dependency part. The remaining lines are the lines of the Action part.

E.g.

¶17. Variables. Besides rules, a make file can contain variable assignments. Variables are implicitly declared by their first appearence in an assignment.

¶18. How to use a make file. To use a make file, we say ”make target”.

References

Two useful references are:
Kernighan and Pike (Chapter 7).
Derek Kiong, Jump Start to C Programming and the Unix Interface

Exercises

Exercise 10.1: Write the `exec*(...)` command to list the help message for `ls`.

♢

Exercise 10.2: Which `exec*(...)` command would you use if you do not want to use the `PATH` searching facility, but know the exact location of the `ls` program?

G ve 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);
```

NOTE: this method might be safer in the sense that someone cannot change the search path and cause you to execute a wrong file.

♢

Exercise 10.3: Write a C program `mul.c` that takes two integers and output their gcd. ♢

Exercise 10.4: Write a C program `reverse.c` that reads an arbitrary sequence of arguments and prints them back in reverse order (one per line). ♢

Exercise 10.5: Write a C program `a2b.c` that takes two unix commands (programs), say A and B. a2b will execute A and then B in this order. Program A does not need any arguments but produce some output which B accepts as input. We should see the output of B on the screen. So this is equivalent to setting up a pipe from A to B: `A | B`.

E.g., if A=`ls` and B=`wc`, then we will list the current directory, send this output to `wc` which will print a 3 integer summary (number of lines, words, bytes).

HINT: use `fork()`, `exec*(...)`, `pipe` and `dup`. For simplicity, do not check for errors in these system calls, and assume that input of `a2b` are correctly formatted.

♢

Exercise 10.6: Write a Makefile that customizes the compilation of programs depending on whether one is in a unix or cygwin environment. Specifically in Unix, when I type "make p" I want to use gcc to compile the program "p.c" into an executable program called "p", and then run "p". But in cygwin, I want the executable to be called "p.exe". ♢

Exercise 10.7: Suppose you have a make file that must be used in CYGWIN as well as in a LINUX environment. E.g., you want execute a file that is named ”a.out” in LINUX but named ”a.exe” in CYGWIN. How can you make such switch of context convenient?

◊

Exercise 10.8: What does `gcc myprog.c -lm` really tell the compiler to do? ◊

Exercise 10.9: Suppose we have three programs, `prog1.c`, `prog2.c`, `prog3.c`. Only `prog1.c` has the main routine defined, but it calls routines defined in `prog2.c` and `prog3.c`.
(a) How do you compile such a collection of programs?
(b) What special C-programming constructs do you need in order to write these programs? ◊

Exercise 10.10: What is the other command that seems to go hand-in-hand with `fork()`? ◊

Exercise 10.11: Suppose you have a Makefile that is ready made. How can you determine what actions you can take in this Makefile? ◊

Exercise 10.12: Give two reasons why keyboard editors are more efficient and more productive than WYSIWYG editors, and one reason why people like WYSIWYG. ◊

End Exercises